LIMES DATENTECHNIK GMBH

FLAM – Der vollständige Kurs

Alle Einheiten des Schulungsmaterials

limes datentechnik gmbh

FLAMv5.21-42969

Begrüßung und Vorstellungsrunde.
Diese Unterlage ist für diese Schulung zusammengestellt und bleibt Ihnen danach vollständig erhalten – auch die Teile, die wir aus Zeitgründen nicht besprechen. Sie können also später damit weiterarbeiten und nachschlagen.

AGENDA

THEMA	SEITE
FLAM-Grundlagen	5
Werkzeuge: FLCL und FLCC	21
Doku und Support	36
Die Kommandosprache CLE/P	47
Die eingebauten Funktionen	62
Dateien auswählen, Namen bilden	75
Administration	88
Betrieb und Wartung	99
Die drei Zugangswege	113
Zugang: Batch-Utility	124
Zugang: Subsystem und Filesystem	137
Zugriff aus dem Programm heraus	150

Die Leseordnung des vollständigen Materials. Jeder Punkt ist eine Einheit, die Seitenzahlen verweisen in diese Unterlage.
Die Reihenfolge ist die Lernordnung: Grundlagen, dann die Zugangswege, dann die Funktionsthemen. Eine Einheit setzt voraus, was vor ihr steht, und sagt es auf ihrer Titelfolie.

AGENDA

THEMA	SEITE
Netzwerk: SSH und FLIES	163
FLAM4 – Architektur und Schnittstellen	176
Das FLAM5-Archiv	189
Indizierung, Filter und Suche	205
Verwaltung archivierter Daten	221
Topfdesign	234
Segment-Store und Cloud	250
Konvertierung mit FLUC	264
Kompression	283
Dateiformate und Sätze	295
Tabellen und CSV	309
Der Format-Compiler	324

AGENDA

THEMA	SEITE
Daten erkennen ohne Beschreibung	339
Zeichensätze	355
XML-Verarbeitung	376
Fremdformate und Austausch	388
Krypto und Schlüssel	399
PGP und Signaturen	413
Sicherheit im Betrieb	426
SAF und RACF	439
Hilfsfunktionen	452
Einbettung in Batch-Jobs	463
Performance und Optimierung	479
Beschleuniger auf z/OS	495



Erste Einheit jeder Schulung. Ziel ist nicht Vollständigkeit, sondern ein tragfähiges Modell: FLAM ist ein Zugriffsverfahren, und es gibt genau drei Wege hinein.
Wer diese Einheit verlassen hat, muss den Satz "ein Parameterstring, drei Wege" erklären können. Alles Weitere hängt daran.



Ziel drei ist das eigentliche Lernziel der Einheit; die anderen vier sind Vokabular, das man dafür braucht.



Die häufigste Fehlannahme im Raum: "FLAM ist wie ZIP". Wer das denkt, sucht später die falschen Lösungen.
Der Unterschied ist nicht die Kompression, sondern der Zugriff: bei einem Archivierer müssen Sie auspacken, um an einen Satz zu kommen. Bei FLAM kommen Sie an den Satz, ohne auszupacken – und genau das erlaubt Archive mit Suche, Updates und Zugriffskontrolle.
Wenn jemand fragt "warum nicht einfach gzip plus openssl": weil danach kein Zugriff mehr möglich ist, nur noch Wiederherstellung.

EIN PARAMETERSTRING, DREI WEGE

Dieselbe Anweisung, dreimal verpackt:

```
read.text(file='eingabe.txt' ccsid='IBM-1141')
write.binary(file='ausgabe.bin')
```

WEG	VERPACKUNG	WER ÄNDERT ETWAS
Batch-Utility	flcl CONV ... in JCL oder Shell	Betrieb
Subsystem, Filesystem	SUBSYS= im DD-Statement, Mount	Betrieb
API	Parameter eines Aufrufs (FLCBYT, FLUC-UP, Satz-IF)	Entwicklung

Das FLCL-Handbuch ist damit auch das API-Handbuch – es beschreibt die Sprache, nicht das Werkzeug.

FLAMv5.21-42969

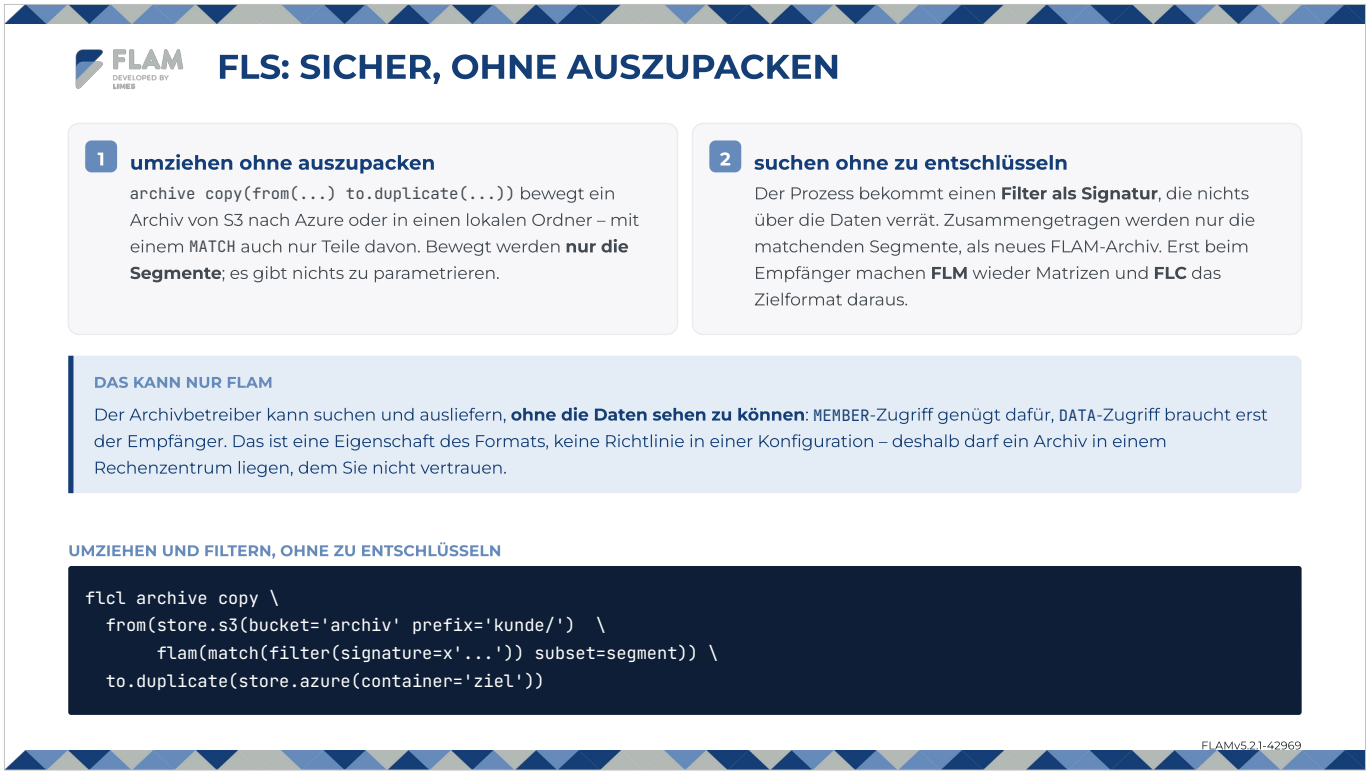
Das ist die tragende Folie der ganzen Schulung, und sie steht so auch auf flam.de: "da aber die gleichen Parameterstrings auch für die API's und Subsysteme genutzt werden ... sind diese Dokumente die Grundlage für die gesamte FLAM Infrastruktur".
Konsequenz für die Teilnehmer: Sie lernen die Sprache einmal. Ob der String in einer JCL steht, in einem DD-Statement oder als Parameter eines API-Aufrufs, ist eine Frage der Verpackung, nicht des Inhalts.
Konsequenz für das Nachschlagen: das FLCL-Handbuch ist auch das API-Handbuch. Das überrascht regelmäßig.



Diese Folie ist wichtiger als sie aussieht, weil die drei Kürzel die Arbeitsteilung erklären – und weil ein Kunde die Kürzel in Fehlermeldungen sieht, lange bevor er die Architektur kennt.
Die Reihenfolge in der Tabelle ist die Reihenfolge des Datenflusses von außen nach innen: FLC nimmt die Originaldaten, FLM macht daraus Segmente, FLS transportiert sie.
FLM ist die eigentliche Zugriffsmethode. Sie serialisiert eine Matrix in ein Segment – das heißt komprimieren und verschlüsseln – und deserialisiert umgekehrt: entschlüsseln, dekomprimieren, und aus dem Segment wieder eine Matrix machen, auf der man den klaren Datensatz sieht.
FLC ist die Originaldatenverarbeitung. Beim Lesen neutralisiert sie: aus einem plattform- und anwendungsspezifischen Format wird eine neutrale FLAM-Matrix, also eine Elementliste. Beim Schreiben genau umgekehrt: aus der neutralen Elementliste wird wieder ein konkretes Format.
Und hier der praktische Hinweis, der im Support Zeit spart: viele dieser Module heißen historisch ORG statt FLC – ORG für Originaldatenverarbeitung. Wenn ein Kunde eine Fehlermeldung aus einem ORG-Modul sieht, dann geht es um genau diese Schicht: das Konvertieren der Originaldaten beim Schreiben und das Neutralisieren beim Lesen.
FLS: das S steht für Secure. FLS-Komponenten sehen nie die klaren Daten, sie arbeiten immer auf dem verschlüsselten Segment. Die nächste Folie führt das aus.



Die Folie zum Datenfluss. Von links nach rechts vorlesen und dabei betonen, wo die Grenze liegt: bis zur Matrix sind es klare Daten, ab dem Segment nicht mehr.
Beim Lesen: FLC nimmt die Originaldatei in ihrem plattform- und anwendungsspezifischen Format und neutralisiert sie zu einer FLAM-Matrix, also einer Elementliste. FLM serialisiert die Matrix in ein Segment – komprimieren und verschlüsseln. Ab hier arbeitet nur noch FLS.
Beim Schreiben genau umgekehrt: FLM deserialisiert das Segment zur Matrix, FLC macht daraus wieder ein konkretes Format.
Der Satz, der hängen bleiben soll: die Matrix ist die neutrale Mitte. Deshalb ist die Zielplattform frei wählbar – das Format der Ausgabedatei entsteht erst in FLC beim Herausschreiben, es steckt nicht im Segment.



Das S in FLS steht für Secure. Der Kern: FLS-Komponenten sehen nie die klaren Daten. Gedacht war FLS primär für den sicheren Austausch der Daten im Netz.
Zwei Beispiele, die heute schon laufen.
Erstens das Umziehen. `archive copy(from(...) to.duplicate(...))` verschiebt ein Archiv von S3 nach Azure oder in einen lokalen Ordner; mit einem MATCH auch nur Teile. Bewegt werden nur die Segmente. Weil nichts entpackt wird, besteht dieser Weg ausschliesslich aus FLS-Komponenten – und deshalb gibt es für ein DUPLICATE auch keine Kompressions- oder Verschlüsselungsparameter zu setzen.
Zweitens die Suche. Der Prozess bekommt einen Filter als Signatur mit, die nichts über die Daten verrät. Zusammengetragen werden nur die matchenden Segmente, als neues FLAM-Archiv. Erst beim Empfänger kommen FLM und FLC zum Tragen: FLM macht aus den Segmenten wieder Matrizen, FLC das Zielformat.
Daraus die Aussage, die vor einem Sicherheitsverantwortlichen zählt: der Archivbetreiber kann suchen und ausliefern, ohne die Daten sehen zu können.
MEMBER-Zugriff genügt dafür, DATA-Zugriff braucht erst der Empfänger.
Wenn jemand nach dem FLS-Server fragt, siehe die nächste Folie – das ist die ehrliche Antwort und sie gehört dazu.



Diese Folie ist eine ehrliche Folie, und sie gehört genau hierher, weil sonst die Frage kommt: wo ist denn der FLS-Server, also der FLSSRV?
Die Antwort: es gibt ihn, aber wir liefern ihn nicht aus, und wir werden ihn in der geplanten Form auch nicht ausliefern. Geplant war FLSSRV für drei eigene Transportwege – IPN für IP-native, IPS für IP-Secure über TLS, und MQS für MQ Series von IBM. Der FLSSRV besteht ausschliesslich aus FLS-Komponenten; der Client dann aus FLM und FLC.
Der Grund für die Entscheidung ist kein technischer Mangel, sondern die Realität heutiger Netze: ein proprietärer Server auf einem eigenen Port, der Daten transportiert, in die niemand hineinschauen kann, stört genau die Systeme, die dafür da sind, das zu verhindern. IPS und Firewalls lassen so etwas nicht durch, und das ist auch richtig so. Wer das gegen seine Netzabteilung durchsetzen muss, hat verloren.
Deshalb der andere Weg: wir sprechen die Protokolle, die ohnehin erlaubt sind.
REST und SSH heute, weitere Standardprotokolle später. Die Sicherheit kommt dabei nicht vom Transportweg, sondern von den Daten selbst – das Segment ist verschlüsselt, unabhängig davon, worüber es reist. Das ist der Punkt: wir brauchen keinen eigenen sicheren Kanal.
Wichtig zur Abgrenzung, sonst klingt es nach einer Lücke: FLS ist mehr als der Server. Alles auf der vorigen Folie – DUPLICATE, Subset, Signatursuche – sind FLS-Komponenten und laufen heute. Zur Netzfähigkeit von FLAM gibt es außerdem einen eigenen Foliensatz; die Fähigkeiten sind da, nur die zentrale Komponente ist bis heute nicht fertig.

WAS BEIM EIN- UND AUSPACKEN PASSIERT



- jede Stufe ist einzeln abschaltbar – Sie zahlen nur, was Sie nutzen
- derselbe Weg läuft rückwärts: die FLAMFILE ist plattformneutral, das Zielformat entsteht erst beim Schreiben
- der Zugriff dekodiert nur das benötigte Segment, nicht die Datei
- bei **FLAM5** werden die MACs bei Update, Insert und Delete **korrekturgerechnet** – Vollständigkeit und Integrität bleiben geschützt; bei **FLAM4** werden sie dabei wertlos

FLAMv5.21-42969

Vier Schritte, immer in dieser Reihenfolge, und jeder einzeln abschaltbar.
Wichtig ist die Umkehrbarkeit: derselbe Weg läuft rückwärts, und deshalb ist die Zielplattform frei wählbar – die FLAMFILE ist neutral, das Format der Originaldatei entsteht erst beim Herausschreiben.
Der MAC ist der Punkt, an dem später die Stolperfalle sitzt: satzweise Änderungen und ein Integritätsnachweis über die ganze Datei schließen sich aus.



Das ist die Folie, auf die man zurückkommt, wenn später jemand fragt, was FLAM5 eigentlich von FLAM4 unterscheidet. Alles andere folgt daraus.
FLAM4 kennt nur das ganze Record. Verschlüsselung, Komprimierung, Integrität – alles gilt für den Record als Block. Indizierte Bereiche gab es nur dort, wo VSAM oder ISAM sie mitgebracht haben.
FLAM5 zerlegt den Record in eine Tabelle mit beliebig vielen Spalten. Und ab da ist jede Entscheidung eine Spaltenentscheidung: welche Spalte verschlüsselt wird, wie sie komprimiert wird, ob und wie sie indiziert wird.
Die Indizierung geht sogar auf **Teile** einer Spalte – man kann etwa nur das Länderkennzeichen einer IBAN indizieren und danach suchen, ohne die IBAN selbst preiszugeben.
Jetzt die Folgerung, und die ist die eigentliche Nachricht: sobald die Daten in Spalten liegen, ist das **logische** Format eine Frage der Ausgabe.
Voraussetzung ist, dass die Daten nur eine unendliche Dimension haben – also beliebig viele Zeilen bei feststehender Spaltenstruktur. Für kommerzielle Massendaten trifft das fast immer zu.
Damit wandelt FLAM5 zwischen **logischen** Formaten: FB und VB auf der einen Seite, CSV, XML und JSON auf der anderen. Beides in beide Richtungen.
Der Vergleich macht es klar: **FLAM4 wandelt nur zwischen physischen Dateiformaten** – Text gegen Record. FLAM5 kann das auch, und mehr davon (Unicode und Weiteres), aber der Unterschied ist die logische Ebene.
Und daraus folgt der Satz, den man im Raum stehen lassen sollte: FLAM5 ist damit ein **Multi-Format-Konverter** – ein Enterprise-Service-Bus für persistente Datenbestände. Nicht für Nachrichten im Fluss, wie die üblichen ESB, sondern für die Bestände, die ohnehin auf der Platte liegen.
Ein Nachtrag, der im Gespräch mit der Anwendungsentwicklung wichtig wird:
FLUC kann dieselbe Wandlung auch **memory to memory** – ohne dass eine Datei im Spiel ist. Damit gilt dieselbe Technologie und derselbe Parameterstring auch für das **Online-Geschäft**: eine einzelne Transaktion, etwa eine Instant Payment, wird im Speicher gewandelt. Wer die Formatlogik einmal als Parameterstring beschrieben hat, benutzt sie im Batch und im Dialog – das ist selten und für ein Haus mit beidem ein Argument.

SEGMENT UND MEMBER: WARUM ZUGRIFF FUNKTIONIERT

- ein **Member** ist eine Originaldatei im Archiv – mit Namen, Attributen und eigenem Schlüssel
- ein **Segment** ist eine autarke Portion Nutzdaten: für sich dekodierbar, ohne Nachbarn
- der Zugriff findet das Segment und dekodiert nur dieses – im Speicher, nie als Datei
- damit bleiben die Kosten pro Treffer etwa konstant, auch wenn das Archiv wächst

Nur das gerade benötigte Segment existiert im Klartext, und nur im Hauptspeicher. Genau das fordern mehrere Sicherheitsstandards.

Vertiefung: Foliensaetze FLAM5-Archiv und Archiv-Verwaltung (Matrix, Bloom-Filter, Versionierung).

FLAMv5.21-42969

Das ist die technische Begründung für das Versprechen von Folie 3. Ohne diese Folie bleibt "Zugriff ohne Auspacken" eine Behauptung.
Die Zahl, die hängen bleiben soll: die Kosten pro Treffer bleiben etwa konstant, wenn das Archiv wächst. Deshalb wird aus einem komprimierten Backup ein abfragbares Archiv.
Details kommen in den Decks FLAM5-Archiv und Archiv-Verwaltung; hier nur das Modell.
Wenn jemand nach dem Speicherbedarf fragt – das gehört in die Notizen, nicht auf die Folie: es sind **nie mehr als vier Matrizen gleichzeitig im Speicher**. Beim Vollzugriff sind es das Verzeichnis, das Member-Segment (der Index) und **zwei** Datensegmente: eines, das über die Daten wandert und Updates und Deletes ausführt, und eines am Ende für die Inserts, also neue Records. Wer nur liest (Unload) oder nur schreibt (Load), hat drei. Diese Zahl ist die Antwort auf "wie viel Hauptspeicher braucht das denn" – sie hängt von der Segmentgröße ab, aber nicht von der Archivgröße.

PLATTFORMEN UND RELEASE-POLITIK

- vom Mainframe bis Embedded und Cloud – z/OS und z/OS USS, BS2000/OSD, AIX, Solaris, Linux (x86, IBM Z, Power), Windows
- dazu **Micro Focus Enterprise Server** (EDZ): dieselben Satzformate und DD-Namen wie auf dem Host, damit eine migrierte Anwendung unverändert bleibt
- **Rolling Release** auf Basis von Continuous Integration: ein Parameter oder Bugfix kann **jeden Montag** bereitstehen
- eine Version wird in der Regel 13 Monate unterstützt und steht mindestens so lange zum Download
- garantierte Rückwärtskompatibilität – deshalb ist „immer aktuell“ die empfohlene Betriebsart

Erste Frage bei jedem Problem: läuft die aktuelle Version? Bei einem Rolling Release ist das keine Ausrede, sondern die kürzeste Diagnose.

FLAMv5.21-42969

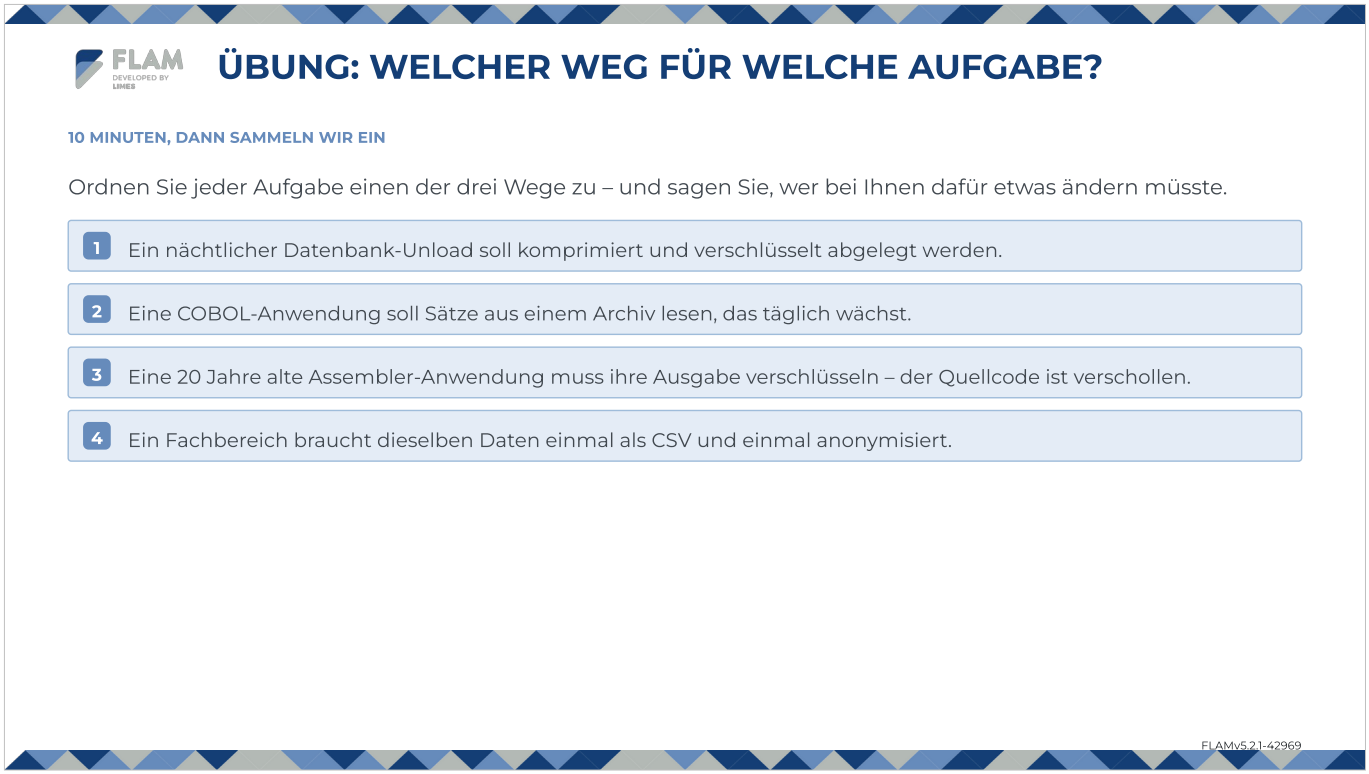
Rolling Release ist der Punkt, der im Betrieb Rückfragen erzeugt: es gibt keine Fix-Releases neben der Linie. Ein Fehler wird mit der nächsten Version ausgerollt, und deshalb lautet die erste Supportfrage immer, ob die aktuelle Version im Einsatz ist.
Rückwärtskompatibilität ist die andere Hälfte: eine 1985 geschriebene FLAMFILE ist heute lesbar. Das erlaubt es, immer die neueste Version zu fahren.
Zur Plattformliste eine Bemerkung, weil im Raum oft alte Zahlen kursieren:
HP-UX, NonStop und z/VSE sind nicht mehr im Support. Bestehende Installationen laufen weiter und die alten Handbücher bleiben verfügbar, aber neue Versionen kommen dort nicht mehr. Wer danach fragt, fragt meist, weil er selbst noch so eine Maschine hat – dann ist das ein Migrationsgespräch und kein Supportversprechen.
Und die Zahlen zur Kadenz, weil sie im Betrieb die eigentliche Nachricht sind: wir arbeiten mit **Continuous Integration**, und daraus folgt, dass ein neuer Parameter oder ein Bugfix **jeden Montag** im Rahmen des Rolling Release bereitstehen kann. Die Erweiterung einer bestehenden Komponente dauert wenige Wochen, eine ganz neue Komponente wenige Monate. Das ist der Grund, warum "immer aktuell" bei uns eine empfohlene Betriebsart ist und nicht ein Risiko.



Diese Folie beantwortet die Frage, mit der Teilnehmer nach der Schulung an ihren Arbeitsplatz zurückgehen: wie probiere ich das aus.
Wichtig: die Demo-Lizenz kann jeder herunterladen und FLAM damit bis 1 MiB nutzen. Wer größere Dateien testen will, braucht eine Test-Lizenz – auf drei oder sechs Monate befristet, mit allen Feature aktiv, und für 0 Euro. Dafür wendet sich der Kunde an sales@flam.de.
Beim Kauf ist keine Neuinstallation notwendig, nur die Lizenz wird getauscht.
Die zweite Tabelle nicht vorlesen, sondern den Satz darunter sagen: die Bemessung ist verhandelbar, und das ist ein echter Unterschied zu Standardprodukten. Wer eine Konzernlizenz hat, zählt keine Installationen und keine Kapazitäten mehr – begrenzt werden nur die Feature und die Liste der Konzerngesellschaften. Details gehören ins Vertriebsgespräch, nicht in die Schulung; hier geht es nur darum, dass die Teilnehmer wissen, dass es diese Möglichkeiten gibt.



Die Folie davor sagt, **was** lizenziert wird. Diese sagt, **wie** bezahlt wird – und das ist die Frage, die im Anschluss aus dem Einkauf kommt.
Vier Modelle, und sie unterscheiden sich nicht im Funktionsumfang, sondern in der Zahlungsform:
Kauf ist das unbefristete Nutzungsrecht, einmalig bezahlt.
Miete ist ein zeitlich befristetes Nutzungsrecht und kostet ein Fünftel des Kaufpreises. Besteht ein Wartungsvertrag, kommen 50 Prozent Nachlass darauf. Das ist der Weg für Projekte mit absehbarem Ende.
Miet-Kauf ist der interessanteste für den Mittelstand: ein zeitlich **unbefristetes** Nutzungsrecht, aber über eine Mindestlaufzeit bezahlt. Bei drei Jahren ein Drittel des Kaufpreises pro Jahr, bei vier Jahren ein Viertel, bei fünf ein Fünftel. Am Ende gehört das Nutzungsrecht dem Kunden, ohne dass er zu Beginn die ganze Summe aufbringen muss.
Verbrauch ist das Modell, das die Bemessung ganz abschafft: bezahlt werden die prozessierten Megabyte, gemessen über unser Usage-Log. Und dann spielen Kapazitäten, Anzahl der Installationen, Plattform und Feature **keine Rolle mehr**. Für einen Kunden, der nicht weiß, wie sich seine Nutzung entwickelt, ist das das risikoärmste Modell – und für einen, der FLAM auf vielen kleinen Maschinen betreibt, oft das billigste.
Der Hinweis auf das Usage-Log lohnt hier doppelt: es ist dasselbe Log, das im Performance-Teil zur Analyse dient. Wer es einschaltet, sieht seinen Verbrauch, bevor er das Modell wählt.
Nicht in die Preisdiskussion einsteigen. Die Folie soll nur sicherstellen, dass niemand die Schulung mit der Vorstellung verlässt, es gäbe nur Kauf.



Zehn Minuten, dann einsammeln. Ziel ist nicht die richtige Antwort, sondern dass die Teilnehmer die Frage "wer muss etwas ändern" selbst stellen.
Die dritte Zeile ist die interessante: eine Anwendung, die nicht angefasst werden darf, führt fast immer zum Subsystem – das ist der Aha-Moment.

WAS BLEIBT

- FLAM ist ein Zugriffsverfahren: es liefert Daten aus, ohne sie auszupacken
- ein Parameterstring, drei Wege – die Sprache lernt man einmal
- `FLC/ORG` konvertiert die Originaldaten, `FLM` serialisiert die Matrix ins Segment, `FLS` transportiert es – nur `FLS` sieht die klaren Daten nie
- Segment und Member sind der Grund, warum Zugriff und Suche funktionieren
- Rolling Release plus Rückwärtskompatibilität: immer aktuell fahren

WEITER IN DIESER REIHENFOLGE

Werkzeuge: FLCL und FLCC	Kommandos bauen, interaktiv nachschlagen
Doku und Support	wo steht es, wie kommt ein Ticket zu uns
Zugang Batch / Subsystem / API	die drei Wege im Detail

FLAMv5.2.1-42969

Die Lernziele noch einmal vorlesen und fragen, ob alle fünf erreicht sind.
Nächste Einheiten in der empfohlenen Reihenfolge: Werkzeuge (FLCL und FLCC), dann Doku und Support, dann die Zugangswege im Detail.

FLAM SCHULUNG, EBENE 0

Kommandos bauen statt Handbuch lesen

Die Kommandozeile, das Framework darunter und die GUI darüber

Falk Reichbott · limes datentechnik gmbh

Zweite Einheit. Nach dem Modell kommt das Werkzeug.
Der Satz, der hängen bleiben soll: es gibt über 650 Manpages und niemand liest es durch – FLCC ist das interaktive Handbuch, und es liefert den String, den Sie danach überall einsetzen.



Ziel vier ist das Arbeitsziel der Einheit: am Ende soll jeder einen String im FLCC bauen und in eine JCL übernehmen können.

DIE 15 KOMMANDOS VON FLCL

GRUPPE	KOMMANDOS	WOFÜR
Archiv	ARCHIVE, FLAM	FLAM5-Archive und FLAMFILEs anlegen, lesen, verwalten
Konvertierung	CONV, ICNV, XCNV	Formate, Zeichensätze, Codierungen wandeln
Tabellen und CSV	BCSV, ACSV, FCSV	binäre Daten und Tabellen nach CSV, mit und ohne Copybook
Prüfen und Vergleichen	DIFF, XCHK, HASH	Unterschiede, Struktur- und Integritätsprüfung
Suchen	GREP	in Dateien und Archiven, auch mit RegEx
Schlüssel	KEY	Schlüssel und Keysets verwalten
Umgebung	INFO, UTIL	Auskunft über Installation, Lizenz und Dateien; `UTIL` erledigt Datei-Aufgaben über Wildcards: suchen, löschen, umbenennen

`flcl ARCHIVE` allein hat 15 Unterfunktionen – COMP, DECO, LIST, UPDATE, DELETE, FILTER, PRUNE, REKEY, VERIFY und weitere.

FLAMv5.21-42969

Nicht vorlesen. Die Teilnehmer sollen sehen, dass die Oberfläche überschaubar ist – das Volumen liegt in den Parametern, nicht in den Kommandos.
Wenn Sie Zeit sparen müssen: nur CONV, ARCHIVE, INFO und GREP erklären, die anderen als "gleiche Sprache, anderes Verb" abhandeln.

DIE SPRACHE: CLE/P

```
flcl CONV read.text(file='ein.txt' ccsid='IBM-1141')
         write.binary(file='aus.bin')
```

ELEMENT	SCHREIBWEISE	BEDEUTUNG
Kommando	CONV	was getan wird
Objekt	read.text(…)	gruppiert zusammengehörige Parameter
Overlay	read.text / read.binary	Auswahl einer von mehreren Varianten
Zuweisung	file='ein.txt'	Zeichenkette in Hochkommata
Array	file[a b c]	mehrere Werte, Klammer statt Gleichheitszeichen
Zahl	blksize=32760	ohne Hochkommata

CLE/P ist Open Source und liegt auf unserem GitHub – dieselbe Kommandozeile können Sie für eigene Werkzeuge nutzen.

FLAMv5.21-42969

Diese Folie ist die Vokabelliste. Wer sie versteht, kann jeden String lesen, auch einen, den er noch nie gesehen hat.
Das Wichtigste ist die Verschachtelung: ein Objekt in Klammern enthält weitere Parameter. Genau deshalb sind die Strings lang – nicht weil die Sprache kompliziert ist, sondern weil sie vollständig ist.
CLE/P ist Open Source und liegt auf unserem GitHub; das überrascht viele.

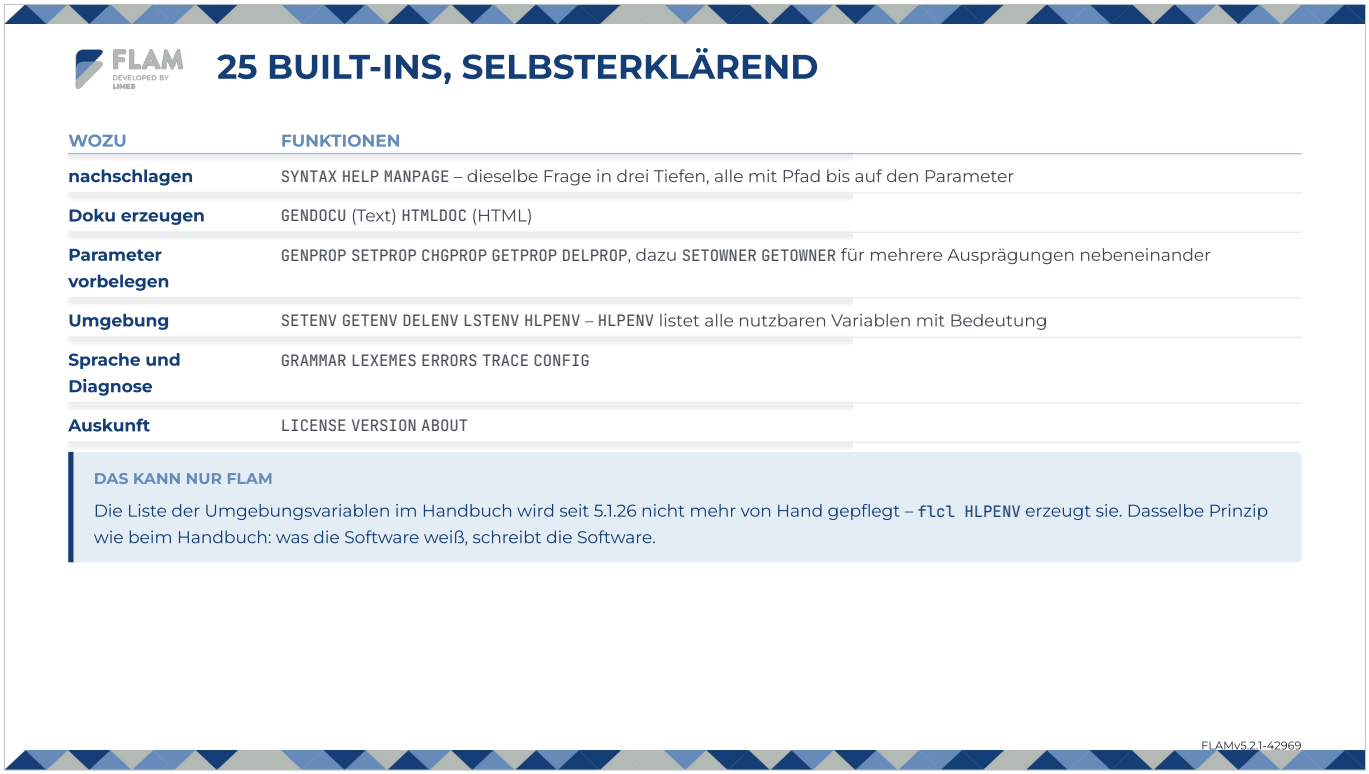
DIE EINGEBAUTE HILFE

AUFRUF	WAS ES TUT
`flcl HELP`	Übersicht aller Kommandos
`flcl HELP CONV`	Hilfe zu einem Kommando
`flcl HELP CONV.read.text.ccsid`	bis auf den einzelnen Parameter hinunter
`flcl MANPAGE CONV`	die Manpage dazu
`flcl GENDOCU CONV=conv.txt`	Teil des Handbuchs erzeugen
`flcl SYNTAX / GRAMMAR / LEXEMES`	die Sprache selbst erklären

- die Software erzeugt ihr Handbuch selbst – die Doku passt damit immer zum installierten Stand
- `HELP` geht bis auf den einzelnen Parameter hinunter, nicht nur bis zum Kommando
- das gesamte Handbuch oder einzelne Teile sind erzeugbar, auch für die eigene Hausdokumentation

FLAMv5.21-42969

Das ist der Teil, der im Betrieb am meisten Zeit spart und am wenigsten bekannt ist: flcl kann das Handbuch selbst erzeugen, und zwar genau für den Stand, der installiert ist. Deshalb kann die Doku nicht veralten.
Praktisch: HELP mit Pfad geht bis auf den einzelnen Parameter hinunter.
Vorführen, nicht erzählen.

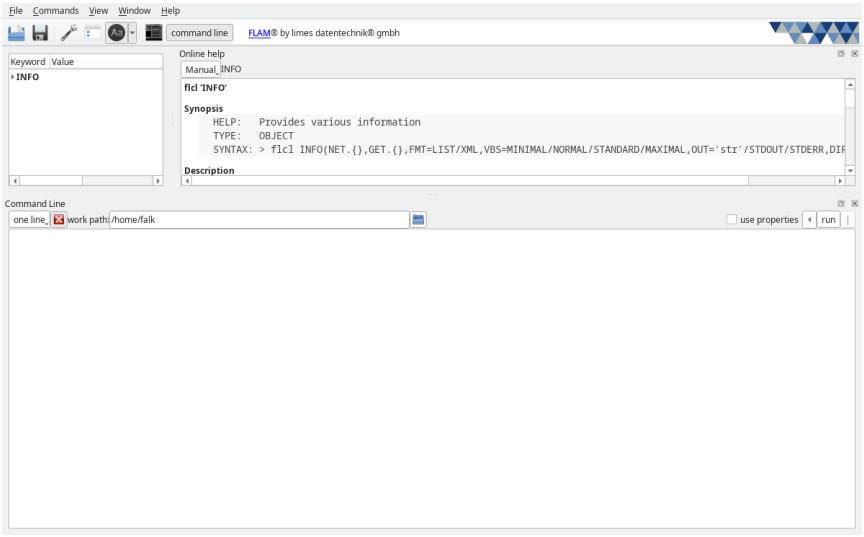


Fünfundzwanzig Built-in Functions, und die meisten Anwender kennen nur HELP.
Sie sind kein Beiwerk: Kapitel 2 des Handbuchs besteht aus ihnen, und sie sind der Grund, warum man für die täglichen Fragen kein Handbuch braucht.
Nicht alle vorlesen. Die Gruppen erklären und je Gruppe einen Vertreter zeigen.
Gruppe eins, nachschlagen: SYNTAX, HELP, MANPAGE. Der Unterschied ist die Tiefe – SYNTAX gibt die eine Zeile, HELP die Erklärung, MANPAGE den ganzen Text. Alle drei nehmen einen Pfad und gehen bis auf den Parameter hinunter.
Gruppe zwei, Doku erzeugen: GENDOCU macht das Handbuch oder einen Teil davon als Text, HTMLDOC als HTML. Das ist der Weg zur eigenen Hausdokumentation.
Gruppe drei, Properties – und das ist die betrieblich wichtigste Gruppe.
GENPROP erzeugt eine Property-Datei mit allen Parametern und ihren Defaults, SETPROP aktiviert sie, CHGPROP ändert einzelne Werte, GETPROP zeigt den aktuellen Stand, DELPROP nimmt sie zurück. Damit belegt ein Haus seine Standards einmal vor und die Kommandos werden kurz. Mit SETOWNER und GETOWNER kann man mehrere Ausprägungen nebeneinander halten – typisch Test gegen Produktion.
Gruppe vier, Umgebung: SETENV, GETENV, DELENV, LSTENV und HLPENV. HLPENV ist der praktische Tipp, weil dort alle nutzbaren Umgebungsvariablen samt Bedeutung aufgelistet werden – das Handbuch pflegt diese Liste seit 5.1.26 nicht mehr von Hand, weil HLPENV sie erzeugt.
Gruppe fünf, Sprache und Diagnose: GRAMMAR und LEXEMES geben die Grammatik und die Lexeme der Kommandosprache aus, ERRORS die Liste der Completion Codes, TRACE schaltet die Ablaufverfolgung ein, CONFIG zeigt und ändert die Konfiguration.
Gruppe sechs, Auskunft: `LICENSE`, `VERSION`, `ABOUT`. `LICENSE` ist die, die man im Betrieb wirklich braucht – sie zeigt, welche Feature aktiv sind.



Der Kern dieser Einheit. Die Begründung offen aussprechen: die Menge der Parameter ist so gross, dass niemand das Handbuch durchliest. FLCC macht die Struktur klickbar und liefert am Ende einen String.
Wichtig für Entwickler im Raum: derselbe String ist der API-Parameter. FLCC ist damit auch das Werkzeug für API-Aufrufe, nicht nur für die Kommandozeile.
Und das geht weiter, als die meisten annehmen: **FLCC bietet nicht nur die FLCL-Kommandos an, sondern auch die übrigen API-Strings** – `format`, `state` und die anderen. Man kann sich damit also **alle** CLP-Strings für unsere Schnittstellen bauen, nicht nur die für das Batch-Utility. Wer gegen die API programmiert, klickt sich seinen Formatstring hier zusammen, statt ihn aus dem Handbuch abzuschreiben. Das deckt auch die Unterprogramme ab.
Drei Dinge zur Bedienung, die im Raum immer gut ankommen:
FLCC kann die **Samples aus dem Handbuch laden**. Man muss also nicht bei null anfangen, sondern nimmt das passende Beispiel und ändert es ab.
Die **Fenster sind frei einrichtbar** – die Doku etwa rechts oben, damit Baum, Hilfe und erzeugter String gleichzeitig sichtbar sind. Genau so ist der Screenshot auf der nächsten Folie aufgebaut.
Und FLCC ist **barrierefrei**: es gibt einen **Flat-Edit-Mode**, der die Bedienung durch Blinde erheblich vereinfacht. Das ist bei öffentlichen Auftraggebern regelmäßig ein Vergabekriterium und lohnt die Erwähnung.
Vorführen: ein CONV-Kommando im Baum zusammenklicken, den String kopieren, in eine JCL setzen, ausführen.

FLCC: PARAMETERBAUM

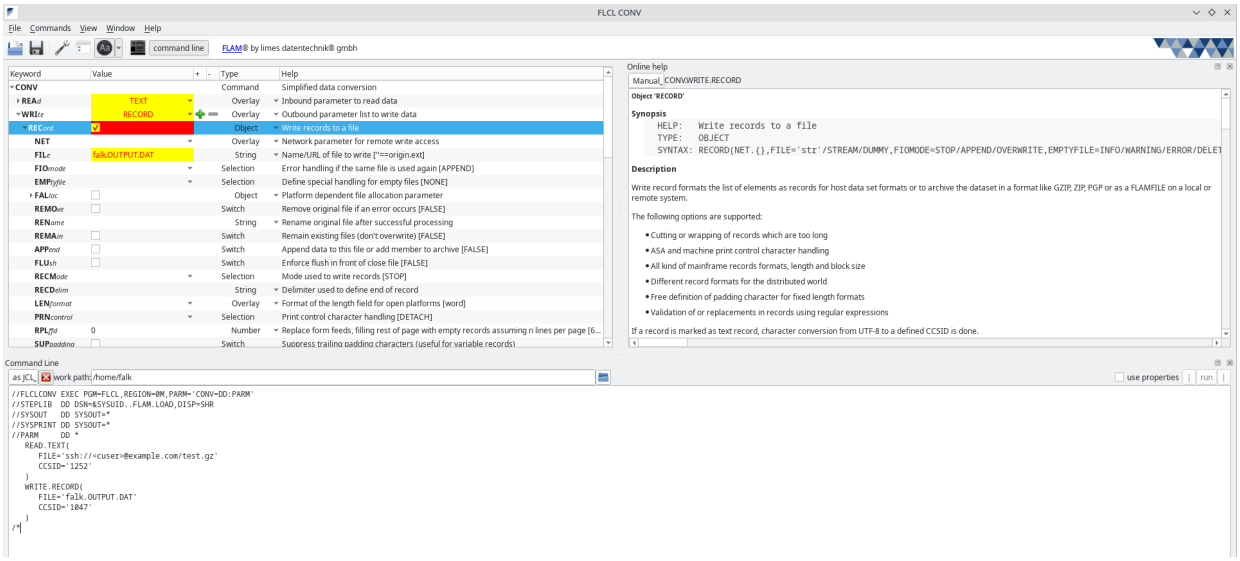


FLCC: links der Parameterbaum, rechts die Online-Hilfe mit Synopsis und Syntax, unten die Kommandozeile mit Arbeitspfad, Property-Schalter und Ausführen

FLAMv5.21-42969

Live vorführen, wenn ein Rechner da ist; das Bild ist der Rückfall.
Zeigen: links die Objekte, rechts die Werte, unten der wachsende String.
Betonen, dass FLCC nur anbietet, was an dieser Stelle erlaubt ist – das ist der Unterschied zu einem Texteditor mit Handbuch daneben.

FLCC: DERSELBE STRING, DREI ZIELE



Der aufgeklappte Parameterbaum eines CONV-Kommandos mit dem erzeugten String

Die Brücke zur Drei-Wege-Folie. Der String aus dem FLCC wandert unverändert in alle drei Verpackungen.
Wenn Zeit ist: einmal in eine JCL kopieren und laufen lassen. Der Effekt "das war es schon?" ist genau der gewünschte.

DASSELBE ÜBER DIE DREI WEGE

BATCH-UTILITY

```
//CONV EXEC PGM=FLCL,PARM='CONV'
//FLAMPAR DD *
  read.text(file='ein.txt' ccsid='IBM-1141');
  write.binary(file='aus.bin');
```

SUBSYSTEM

```
//DDNAME DD DSN=datei,SUBSYS=(FLAM,'CRYPTOM=AES')
```

API

```
pvRed = fcbopen("read.text(file='ein.txt' ccsid='IBM-1141')", ...)
```

Der Inhalt der Klammern ist identisch. Nur die Verpackung wechselt.

FLAMv5.21-42969

Diese Folie kommt in jedem Funktionsdeck in derselben Form wieder. Hier ist sie zum ersten Mal, also einmal ausführlich erklären.
Die Botschaft: der Inhalt der Klammern ist identisch. Was sich unterscheidet, ist nur, wer den String übergibt.



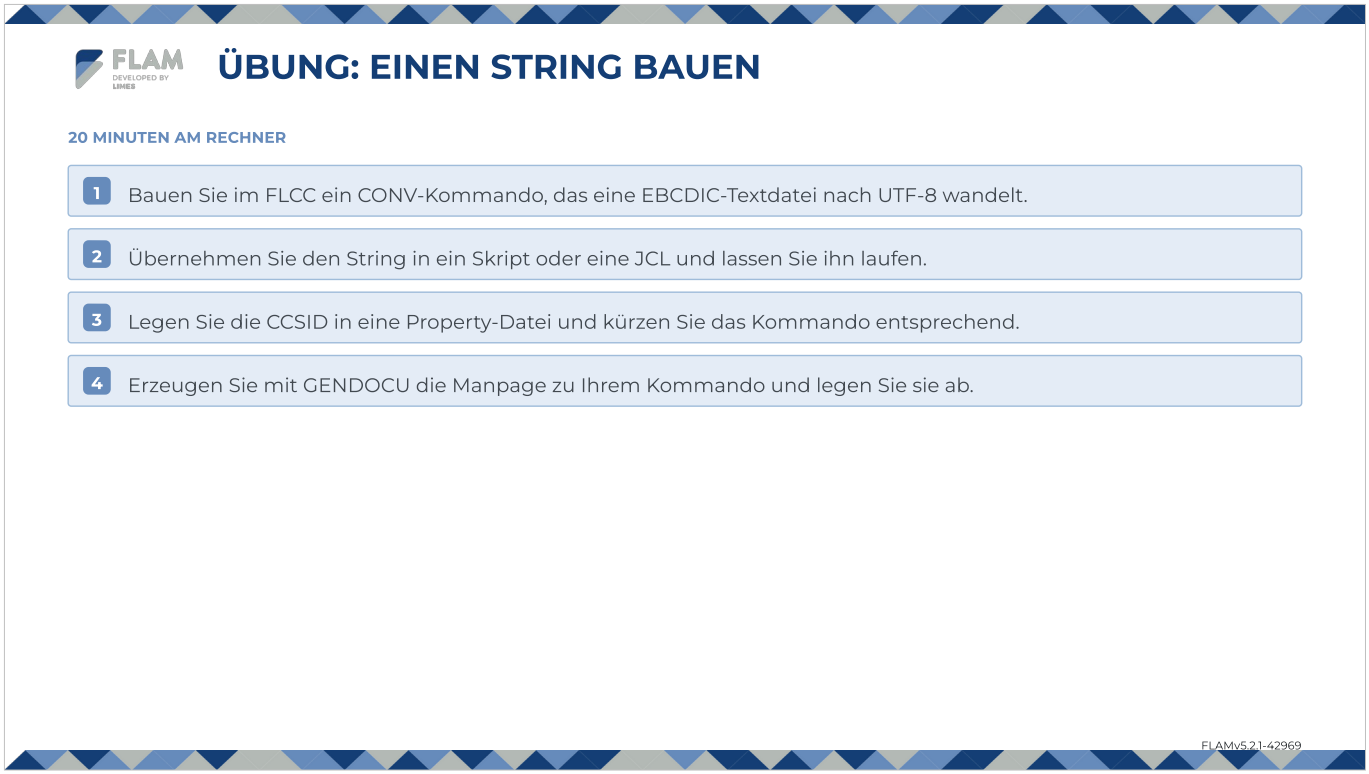
Der Grund, warum Produktionsstrings kurz sind: alles Wiederkehrende steht in einer Property-Datei oder einer Konfiguration, nicht im Kommando.
Die Reihenfolge ist die häufigste Fehlerquelle im Betrieb – wenn ein Parameter "nicht wirkt", überschreibt ihn fast immer eine spätere Quelle.
Das gehört auf das Whiteboard.
Und ein Weg, der zusätzlich zu den Property-Dateien existiert und den kaum jemand kennt: **jeder Parameter lässt sich über seinen Pfad als Umgebungsvariable vorbelegen.** Die Regel ist mechanisch – Pfad nehmen, alles groß schreiben, Punkte durch Unterstriche ersetzen. Aus `CONV.WRITE.TEXT.METHOD` wird also `CONV_WRITE_TEXT_METHOD`.
Man kann davor noch Programm und Owner setzen, um es einzugrenzen: `HUGO_FLCL_CONV_READ_TEXT_ENL2LF=ON` gilt nur für den Owner `hugo`. Die ownerspezifische Angabe schlägt die programmspezifische, und die schlägt die globale.
Das ist der bequemste Weg, in einem Testsystem einen Standard zu setzen, ohne eine Datei anzulegen – und `flcl GENPROP` erzeugt die Liste aller Properties, wenn man wissen will, was überhaupt vorbelegbar ist.



Zwei eingebaute Funktionen, die kaum jemand kennt und die dem Werkzeug einen Teil seiner Ausdruckskraft geben: `flcl lexemes` und `flcl grammar`.
`grammar` gibt die Grammatik der Sprache aus, `lexemes` die Lexeme – also was ein Zahl, ein String, ein Kommentar ist. Das klingt akademisch, aber zwei Abschnitte darin braucht man in der Produktion regelmäßig.
Erstens die vordefinierten Konstanten. Sie sind in Wertausdrücken benutzbar, und es sind mehr, als man erwartet: `NOW`, `MINUTE`, `HOUR`, `DAY`, `YEAR` für Zeitrechnungen; `KiB`, `MiB`, `GiB`, `TiB` für Größen; `PI` als Fließkommazahl; `RND1` bis `RND8` für Zufallszahlen. Und eine ganze Reihe von Zeitstempeln in fertigen Formaten – `LCSTAMP`, `LCDATE`, `LCYEAR`, `LCMONTH`, `LCTIME` für die lokale Zeit, dieselben mit `GM` davor für Greenwich.
Und man kann damit **rechnen**: `bufsiz=4MiB` *ist gültig. Das ist deutlich lesbarer als `4194304`, und es steht so in der JCL.*
Zweitens der Zugriff auf die Umgebung. In einem String wird `<name>` durch den Wert der Umgebungsvariablen ersetzt. Auf z/OS trifft das auch exportierte JCL-Symbole und Systemsymbole – damit wird eine Prozedur wiederverwendbar, ohne dass die JCL angefasst wird.
Wichtig ist der Unterschied, weil hier alle einmal hineinlaufen: **die Konstanten sind Werte, keine Textbausteine.** `file=LCSTAMP` erzeugt eine Datei mit dem Zeitstempel als Namen. `file='/pfad/<LCSTAMP>.csv'` tut das **nicht** – in einem String wird nur die Umgebungsvariable ersetzt, und `LCSTAMP` ist keine. Wer den Zeitstempel im Dateinamen braucht, nimmt die Namens-Tokens (`[base]`, `[indN]`) oder setzt sich vorher eine Umgebungsvariable. Nachgemessen, nicht vermutet – beide Fälle laufen so.



Alle aus Supportfällen. Die erste kostet die meisten Tickets.
Zu den Anführungszeichen, weil das in jeder Shell-Diskussion aufkommt: sie sind **nur nötig, wenn der String selbst Syntaxelemente enthält** – also Klammern, Leerzeichen, Operatoren. Sonst kann man sie weglassen, und ein String darf direkt hinter dem Gleichheitszeichen beginnen; er endet dann am nächsten Trenner.
Und es gibt nicht nur das einfache Hochkomma: `'`, `"` und der Backtick sind alle gültig. Das ist praktisch, weil in Bash, PowerShell und JCL jeweils ein anderes Zeichen Ärger macht – man nimmt einfach das, das dort nicht stört. Nachzulesen in `flcl lexemes`.



Zwanzig Minuten am Rechner. Wer keinen FLCC hat, arbeitet mit `flcl HELP`.
Ziel ist der Moment, in dem der selbst gebaute String läuft – danach ist die Hemmschwelle weg.
Auflösung:
1. Im FLCC das Kommando wählen, im Parameterbaum die benötigten Objekte aufklappen und die Werte setzen. Was inaktiv ist, gehört nicht zum gewählten Overlay – das ist die Lernhilfe.
2. Die Kommandozeilen-Ansicht liefert den String. Umschalten auf die strukturierte Ansicht zum Lesen, auf "als JCL" für den Host.
3. Derselbe String läuft in der Shell, in der JCL und als erster Parameter eines `fcbopen`. Das ist der Satz "ein Parameterstring, drei Wege" in der Praxis – und der Grund, warum das FLCL-Handbuch auch das API-Handbuch ist.
4. Wer prüfen will, ob der String stimmt, ohne ihn laufen zu lassen: `NORUN` im Kommando. Der Parser prüft dann die Syntax und beendet.

WAS BLEIBT

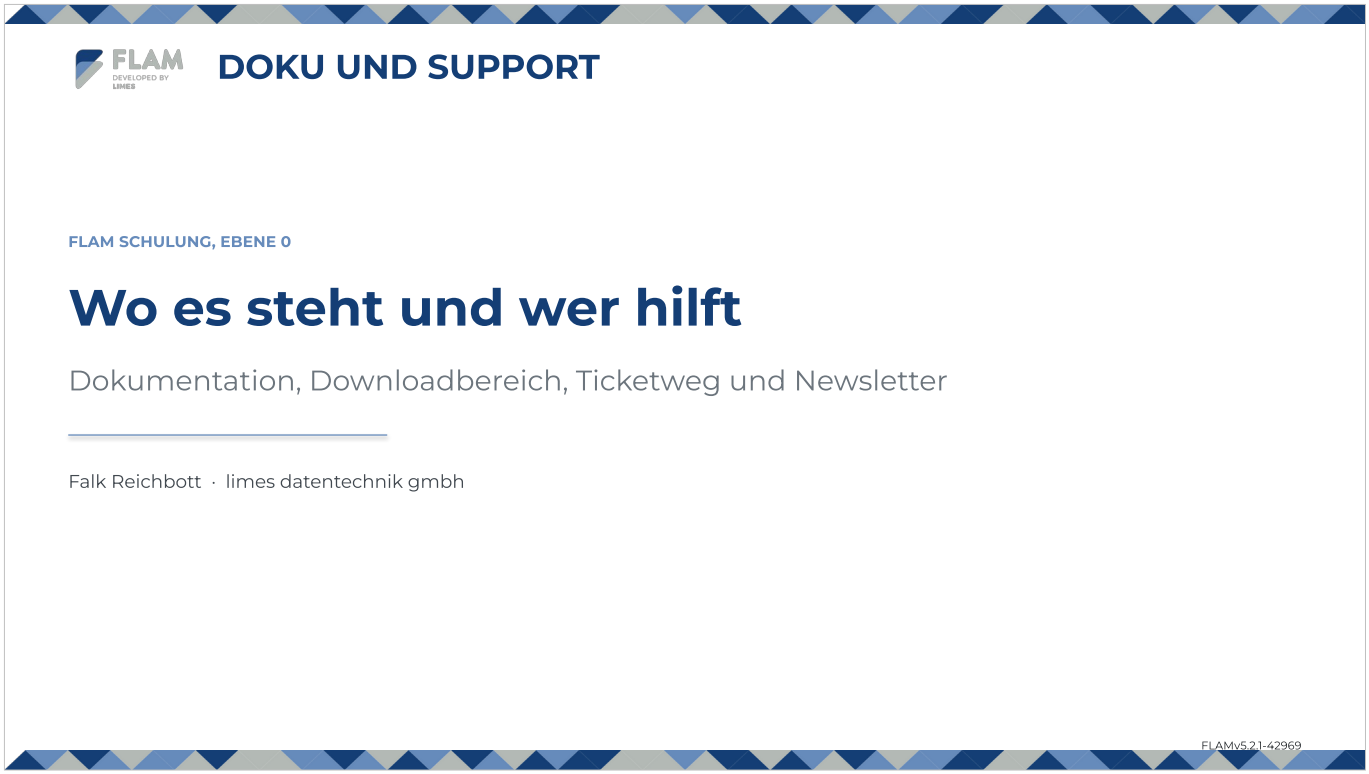
- 15 Kommandos, eine Sprache – das Volumen liegt in den Parametern
- CLE/P: Kommando, Objekt, Overlay, Array. Wer das liest, liest jeden String
- die Software erzeugt ihr Handbuch selbst; HELP geht bis zum Parameter
- FLCC ist das interaktive Handbuch und liefert den String für alle drei Wege
- Vorbelegung hat eine Reihenfolge – sie ist die häufigste Fehlerquelle

WEITERLESEN

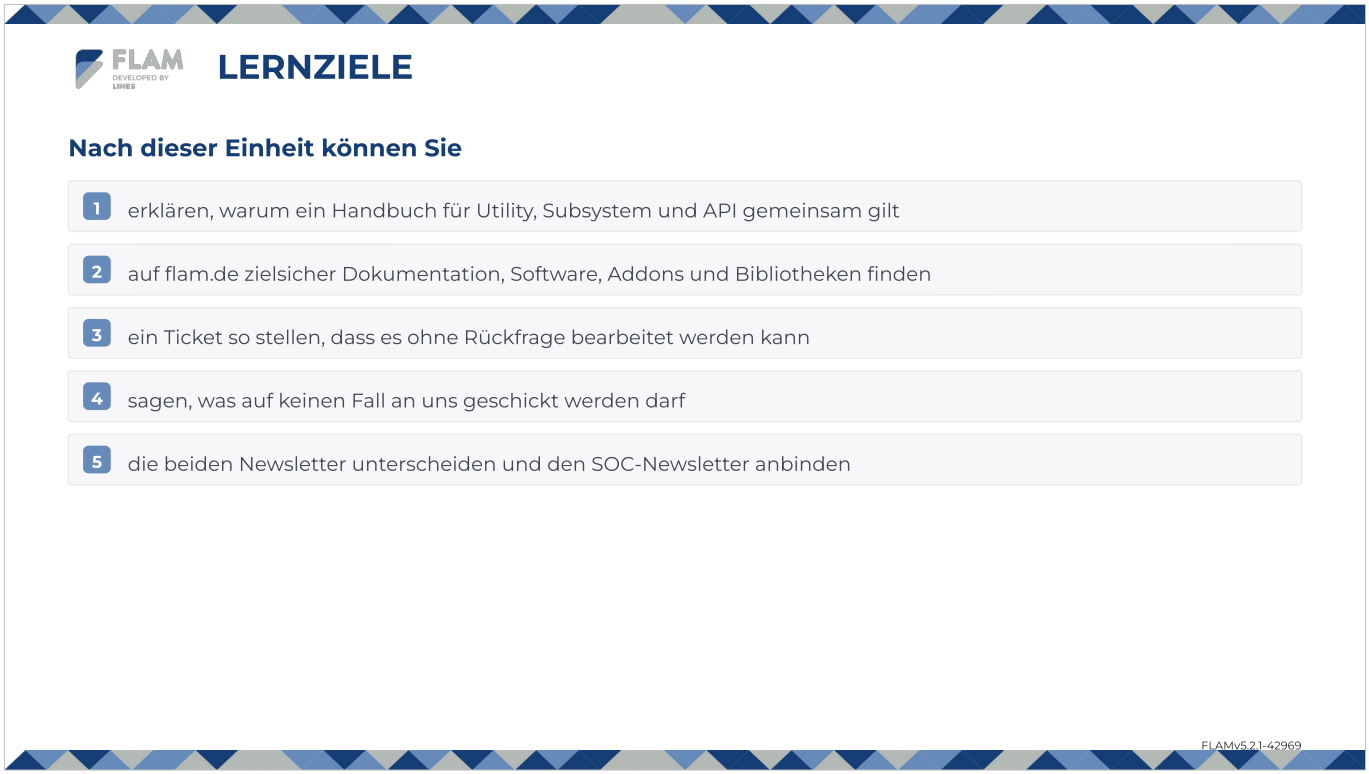
FLCL-Handbuch	flam.de/downloads/documentation
CLE/P Quellen und API	unser GitHub-Bereich
Hilfe im Werkzeug	flcl HELP, MANPAGE, GENDOCU, SYNTAX, GRAMMAR, LEXEMES

FLAMv5.21-42969

Lernziele durchgehen. Danach folgt Doku und Support – also wo das alles liegt und wie ein Ticket zu uns kommt.



Dritte Einheit der Ebene 0. Unspektakulär, aber die mit dem höchsten Nutzwert nach der Schulung: wo finde ich es, und wen frage ich.
Diese Einheit auf keinen Fall am Ende des Tages einplanen – sie enthält die Informationen, die die Teilnehmer mitnehmen und weitergeben.



Ziel zwei ist das, was im Alltag zählt. Ziel vier verhindert einen Datenschutzvorfall – das ist kein Formalismus.

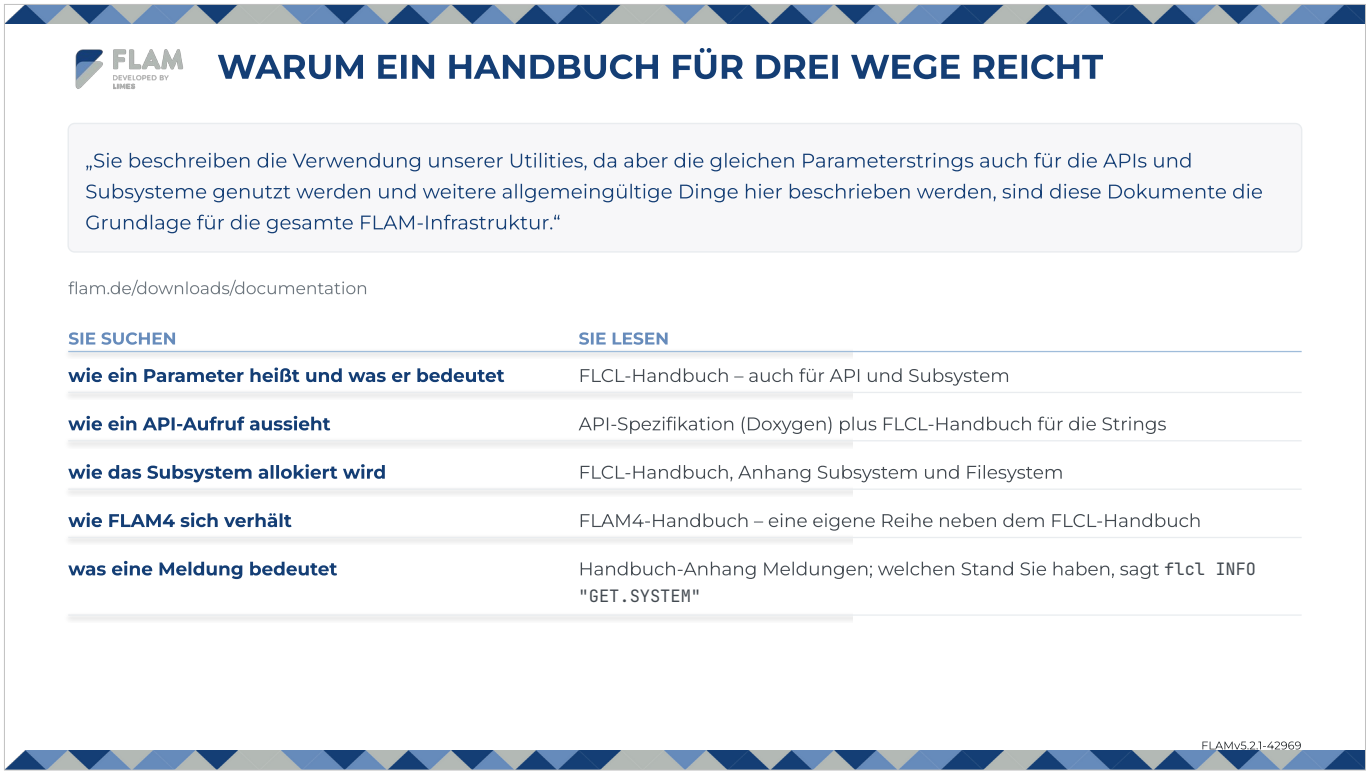
DIE SOFTWARE ERZEUGT IHR HANDBUCH SELBST

- das Handbuch wird aus derselben Parametertabelle erzeugt, aus der auch der Kommandoparser entsteht
- damit stimmt die Dokumentation immer mit dem Softwarestand überein
- die Doku ist **Teil der Installation** – im Zweifel gilt die installierte, nicht die online gelesene
- online steht immer nur die neueste Version
- Teile davon können Sie selbst erzeugen: `flcl GENDOCU`

Ein Handbuch, das die Software selbst schreibt, kann nicht veralten – aber es beschreibt die Sprache, nicht Ihren Anwendungsfall. Dafür ist die Schulung da.

FLAMv5.21-42969

Das ist der Grund, warum unsere Doku nicht veraltet – und warum sie manchmal spröde klingt: sie kommt aus derselben Tabelle, aus der auch der Parser gebaut wird.
Praktische Folge für den Betrieb: die Doku im Installationspaket ist immer die richtige. Online steht nur die neueste Version.

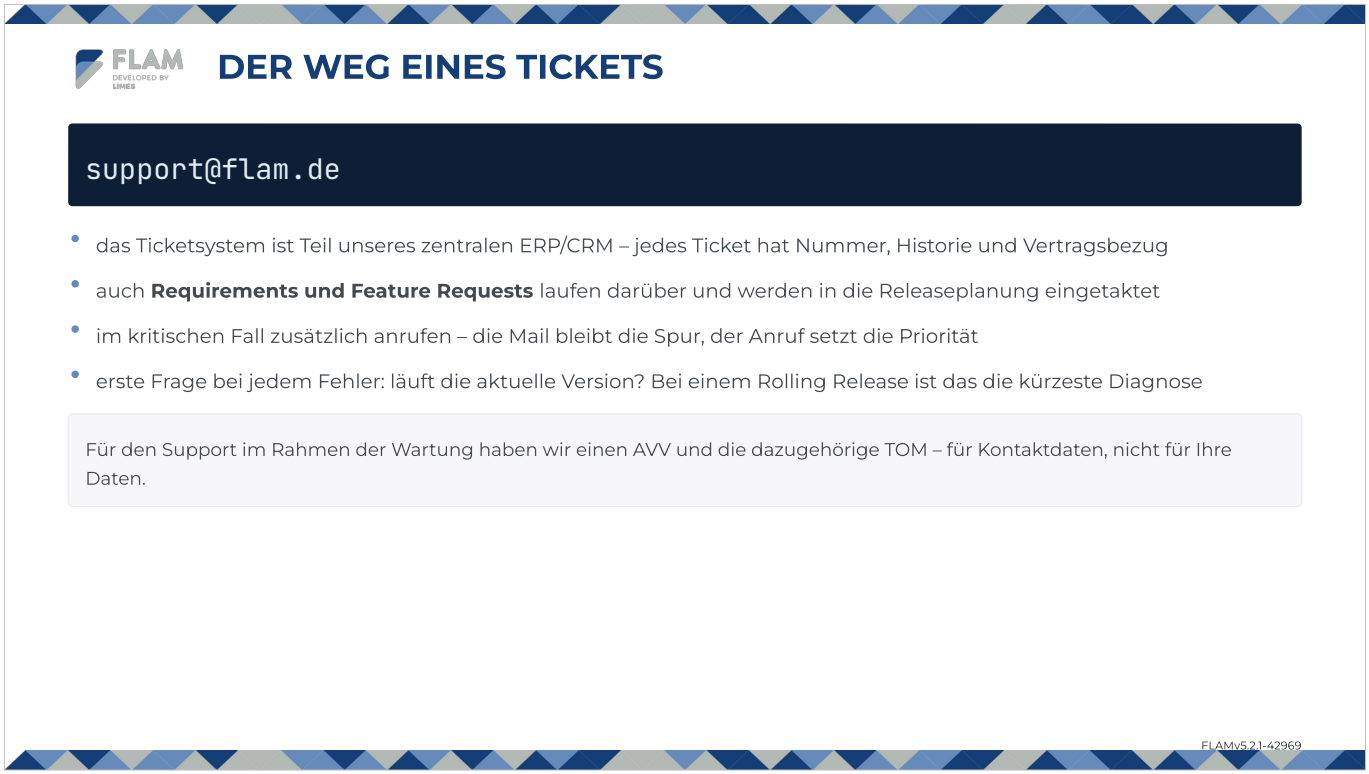


Wörtlich von flam.de: "Sie beschreiben die Verwendung unserer Utilities, da aber die gleichen Parameterstrings auch für die API's und Subsysteme genutzt werden ... sind diese Dokumente die Grundlage für die gesamte FLAM Infrastruktur." Das ist dieselbe Aussage wie in den Grundlagen, hier aber aus der Sicht des Nachschlagens. Entwickler im Raum suchen erfahrungsgemäß ein "API-Handbuch" und finden es nicht – weil es das FLCL-Handbuch ist.

DER DOWNLOADBEREICH AUF FLAM.DE

REITER	INHALT
Dokumentation	alle Handbücher tabellarisch, FL5- und FLAM4-Reihe, API-Spezifikation – und der Excel-Bogen für die Schulungsplanung
Software	Installationspakete für alle Plattformen und Versionen, in der Regel 13 Monate verfügbar
Add-ons	Erweiterungen zum Hauptpaket, zum Beispiel das Java Byte Interface mit JNI-DLL, Java-Klassen und Stream-Implementierungen
Bibliotheken	unsere Anpassungen an Open-Source-Projekten auf GitHub – mit unseren Anpassungen – und der CLE/P als eigenes Open-Source-Projekt

Vier Reiter, und die Teilnehmer kennen meist nur einen. Besonders "Libraries" ist unbekannt – dort liegen unsere Anpassungen an externen Bibliotheken, und der CLE/P als eigenes Open-Source-Projekt.
Der Excel-Bogen für die Schulungsplanung liegt gleich oben unter Dokumentation – darauf hinweisen, er ist der Einstieg für die nächste Schulung.



Der zentrale Satz: das Ticketsystem ist Teil unseres ERP/CRM, es ist kein Postfach nebenher. Damit hat jedes Ticket eine Nummer, eine Historie und einen Bezug zum Wartungsvertrag.
Feature Requests gehen denselben Weg – das überrascht regelmäßig und ist eine gute Nachricht: Anforderungen aus dem Betrieb landen in der Releaseplanung, nicht in einem Wunschzettel.

WAS INS TICKET GEHÖRT – UND WAS NICHT

BITTE MITSCHICKEN	BITTE NICHT SCHICKEN
Version und Plattform (`flcl INFO GET.SYSTEM`)	Dumps aus dem Produktivbetrieb
der vollständige Parameterstring	produktive Kundendaten, auch nicht „nur ein Auszug“
die Meldung im Original, nicht abgetippt	Screenshots statt Text, wenn Text möglich ist
ein anonymisiertes Minimalbeispiel	Zugangsdaten, Schlüssel, Keysets
was Sie erwartet haben und was passiert ist	

Wer sensible Daten an uns schickt, hat diese schon beim Losschicken kompromittiert. Bitte keine Dumps und keine produktiven Kundendaten.

FLAMv5.21-42969

Die linke Spalte spart eine Rückfragerunde, die rechte verhindert einen Datenschutzvorfall.
Den letzten Punkt deutlich sagen, er ist unangenehm und wichtig: wer sensible Daten an uns schickt, hat sie beim Losschicken schon kompromittiert. Wir sind kein Rechenzentrum und haben keinen Zugang zu Kundensystemen.
Für Fehlerbilder reichen fast immer die Version, der Parameterstring, die Meldung und ein anonymisiertes Minimalbeispiel.

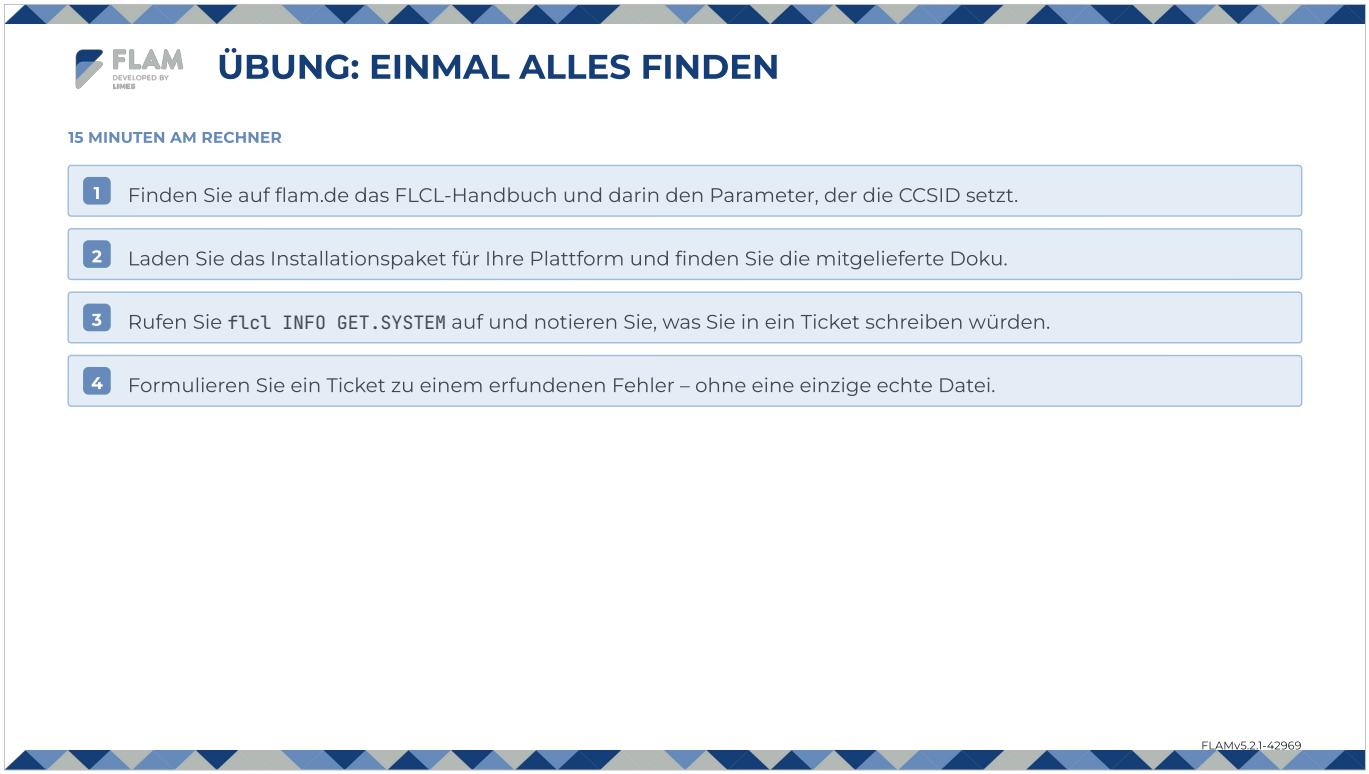


Der zweite ist der, den kaum jemand kennt und den jede regulierte Organisation braucht: der SOC-Newsletter meldet neu erkannte Sicherheitslücken in unserer Software **und in den von uns verwendeten externen Bibliotheken**, nach den Vorgaben von BSI und NIST.
Das ist die Antwort auf die DORA- und NIS2-Frage „wie erfahren wir von Schwachstellen bei Ihnen“.
Konkret anbieten, die Adresse des SOC aufzunehmen.

OPEN SOURCE UND VERÖFFENTLICHUNGSPFLICHTEN

- alle genutzten und angepassten Open-Source-Projekte liegen einsehbar auf unserem GitHub
- damit kommen wir den Veröffentlichungspflichten der jeweiligen Lizenzen nach
- Beispiel Mehrwert: der libiconv haben wir zusätzliche **EBCDIC-Codepages** hinzugefügt
- **CLE/P** ist unser eigenes Projekt und steht kostenfrei als Open Source zur Verfügung – mit API-Spezifikation und Doxygen-Doku
- Fehler und Feature Requests dazu bitte ebenfalls über support@flam.de

Zwei Botschaften. Erstens: wir kommen den Pflichten der Open-Source-Lizenzen nach und legen unsere Anpassungen offen – prüfbar auf GitHub. Zweitens: ein Teil davon ist ein Mehrwert für den Kunden, etwa die EBCDIC-Codepages in der libiconv, die es dort im Original nicht gibt.
CLE/P ist unser eigenes Projekt und frei nutzbar; wer eine eigene plattformübergreifende Kommandozeile braucht, kann sie nehmen.



Fünfzehn Minuten am Rechner. Der Effekt ist gross und der Aufwand klein: wer den Weg einmal gegangen ist, geht ihn im Alltag wieder.
Bei Aufgabe 4 darauf achten, dass wirklich niemand echte Daten anhaengt – das ist der Lernpunkt.
Auflösung:
1. Der Parameter heißt `CCSID` und steht in jedem read- und write-Objekt.
Schneller als das Handbuch ist `flcl HELP CONV.read.text.ccsid`.
2. Die Doku liegt im Paket, nicht daneben – das ist der Lernpunkt: sie passt damit zur installierten Version, auch wenn online schon eine neuere steht.
3. Ins Ticket gehören Version und Build, Plattform, aktive Lizenz und die wirksamen Umgebungsvariablen – also genau die Ausgabe von `flcl INFO GET.SYSTEM`. Das erspart die erste Rückfragerunde.
4. Erwartet wird: Was war die Absicht, was war der Aufruf, was kam heraus, was sollte herauskommen – plus die INFO-Ausgabe. Und keine echten Daten. Wer Testdaten braucht, konstruiert sie; wer Produktionsdaten anhängt, hat ein Datenschutzproblem geschaffen, um ein Softwareproblem zu melden.

WAS BLEIBT

- die Software schreibt ihr Handbuch selbst – die installierte Doku gilt
- ein Handbuch für drei Wege, weil der Parameterstring derselbe ist
- vier Reiter im Downloadbereich: Dokumentation, Software, Add-ons, Bibliotheken
- Tickets an support@flam.de – mit Version, String und Meldung, ohne Produktivdaten
- zwei Newsletter: Releases für den Betrieb, Schwachstellen für das SOC

WEITERLESEN

Support und Wartung	flam.de/support
Handbücher und Schulungsbogen	flam.de/downloads/documentation
Software und Lizenzen	flam.de/downloads/software
Open Source und CLE/P	flam.de/downloads/libraries

FLAMv5.21-42969

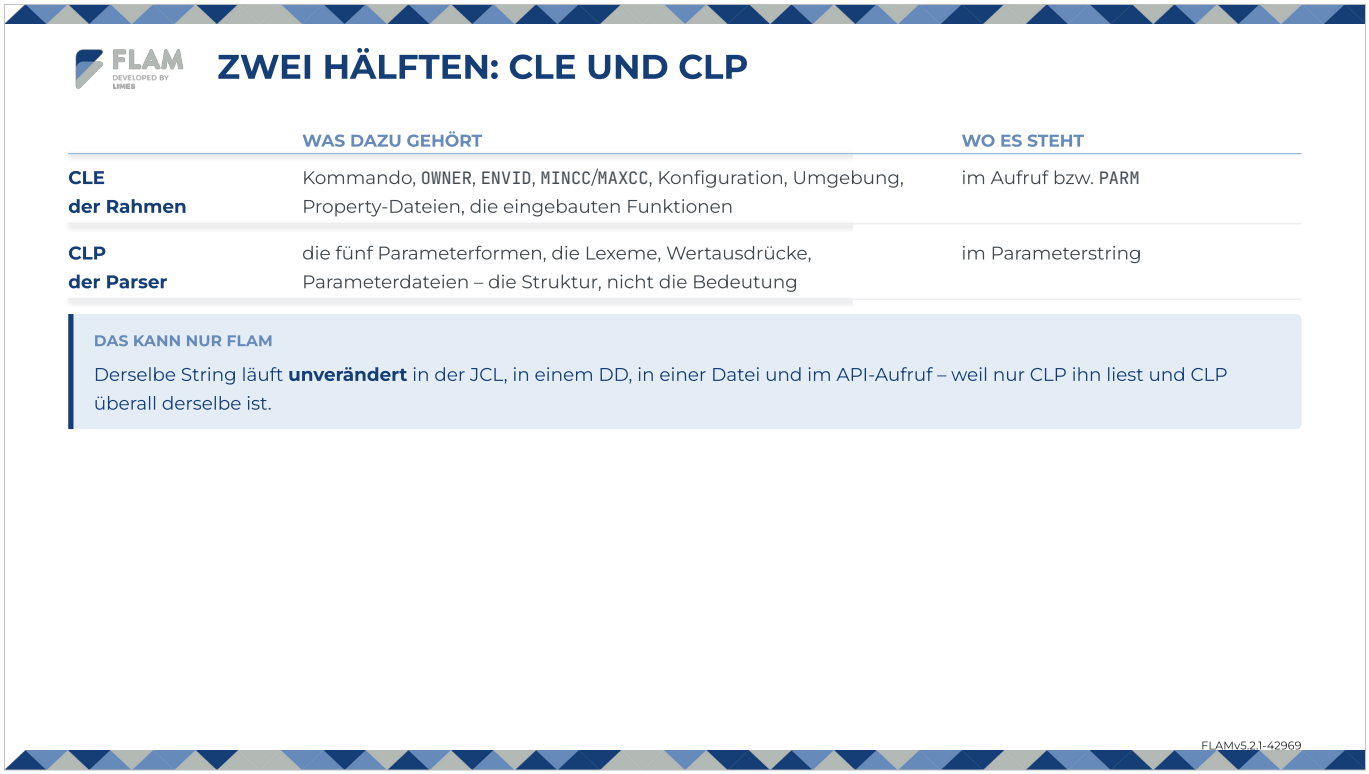
Lernziele durchgehen. Damit ist die Ebene 0 abgeschlossen; danach folgen die drei Zugangswege im Detail.



Diese Einheit ist die Referenz zur Sprache. Sie wird selten von vorn bis hinten vorgetragen – man zeigt sie einmal und schlägt später darin nach.
Der Bogen: die zwei Hälften CLE und CLP, die fünf Parameterformen, die Lexeme, das Rechnen im String, Parameterdateien, und zuletzt die Fehlerausgabe des Parsers.
Der Satz, der die Einheit rechtfertigt: **jeder Parameterstring in FLAM ist in derselben Sprache geschrieben** – in der JCL, in der Shell, im Subsystem, im API-Aufruf. Wer sie einmal liest, liest sie überall.
Vorausgesetzt wird *Werkzeuge: FLCL und FLCC*; dort steht die Vokabelliste als Übersicht und die eingebaute Hilfe.



Das dritte Ziel ist das, um das es wirklich geht: einen fremden String lesen, ohne ihn nachzuschlagen.
Das fünfte ist der Punkt, der in der Produktion Zeit spart – eine Fehlermeldung des Parsers sagt genau, wo er stehengeblieben ist und was er bis dahin verstanden hat. Wer das einmal gesehen hat, sucht nicht mehr.



Die Trennung erklärt, warum manches am Kommando steht und manches im String – eine Frage, die in jeder Schulung kommt.
CLE ist der Rahmen um den Aufruf: welches Kommando läuft, unter welchem Owner, mit welcher Umgebung und welchem Returncode-Fenster. Dazu gehören auch die eingebauten Funktionen – `HELP`, `SYNTAX`, `MANPAGE`, `GENDOCU`, `GENPROP`, `LEXEMES`, `GRAMMAR`.
CLP ist der Parser für den Parameterstring. Er kennt die Formen, die Lexeme und die Wertausdrücke, und er kennt die Struktur der Kommandos – aber nicht ihre Bedeutung.
Der praktische Nutzen der Trennung: `MAXCC` und `OWNER` stehen im PARM, weil sie CLE betreffen. Alles, was die Verarbeitung beschreibt, steht im String, weil es CLP betrifft. Deshalb kann derselbe String unverändert in die JCL, in ein DD, in eine Datei oder in einen API-Aufruf.
Und beides ist Open Source – CLE/P liegt auf unserem GitHub und wird von anderen Werkzeugen benutzt.

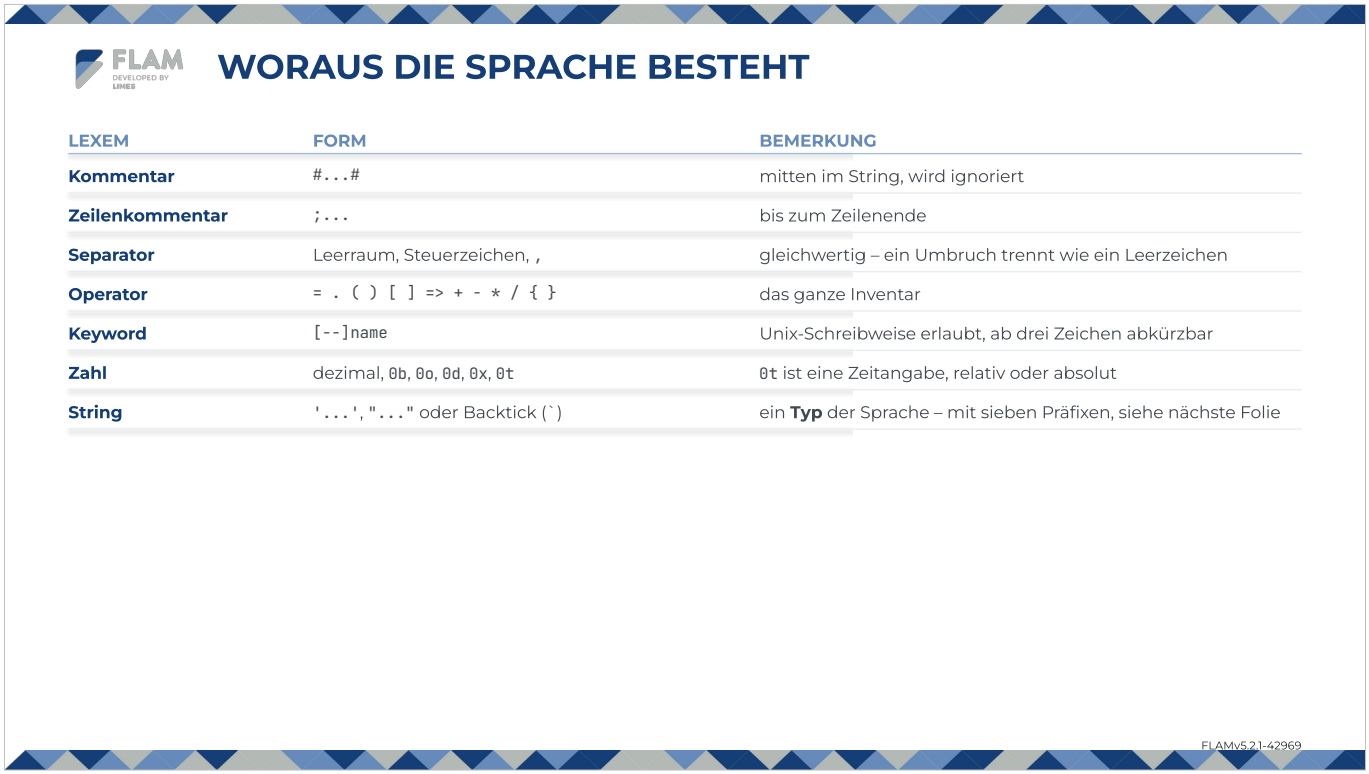
FÜNF FORMEN, MEHR GIBT ES NICHT

FORM	SCHREIBWEISE	BEISPIEL
Schalter	`KEYWORD`	`suptws`
Zuweisung	`KEYWORD=wert`	`ccsid='UTF-8'`
String	`'...'`, `"..."`, Backtick (`)	`file='ein.txt'`
Zahl	dezimal, `0b`, `0o`, `0d`, `0x`, `0t`	`reclength=80`
Float	`[+|-]d.dE d`	`faktor=1.5E2`
Auswahl	vordefiniertes Schlüsselwort	`prncontrol=REPLACE`
Objekt	`KEYWORD(...)`	`read.text(file='ein.txt')`
Overlay	`KEYWORD.variante`	`read.record` / `read.text`
Array	`KEYWORD[...]`	`file['a' 'b' 'c']`

Die Klammer **ersetzt** beim Array das Gleichheitszeichen. `file[a b c]` ist richtig, `file=[a b c]` nicht.

FLAMv5.21-42969

Diese Folie ist die wichtigste der Einheit. Wer die fünf Formen kennt, kann jeden String zerlegen – auch einen, den er noch nie gesehen hat.
Langsam durchgehen und an jeder Zeile das Beispiel zeigen:
Ein **Schalter** ist ein Keyword allein. Er sagt ja, ohne einen Wert.
Eine **Zuweisung** ist Keyword, Gleichheitszeichen, Wert – und dahinter stehen die **skalaren Typen** der Sprache, die man auseinanderhalten muss, weil der Parser sie auseinanderhält: ein **String** in Anführungszeichen, eine **Zahl** dezimal oder in einer der anderen Basen, ein **Float** mit Punkt und Exponent, und die **Auswahl** – ein vordefiniertes Schlüsselwort wie `REPLACE`, das aussieht wie ein Keyword und ein Wert ist. Der Schalter ist der fünfte skalare Typ, aber er hat keine Zuweisung, deshalb steht er in der Tabelle darüber.
Ein **Objekt** gruppiert Parameter in Klammern. Es ist die Form, die die Strings lang macht – nicht weil die Sprache kompliziert ist, sondern weil sie vollständig ist.
Ein **Overlay** ist eine Auswahl: genau eine von mehreren Varianten, und der Punkt trennt. `read.text` und `read.record` sind zwei Varianten desselben Overlays.
Ein **Array** nimmt mehrere Werte. Die Klammer ersetzt das Gleichheitszeichen – `file[a b c]`, nicht `file=[a b c]`. Es gibt auch die Form ohne Klammern, aber dann ist das Ende nicht mehr sicher erkennbar; das Handbuch empfiehlt ausdrücklich die eingeklammerte Form, und wir auch.
Der Rest der Sprache – Lexeme, Rechnen, Dateien – hängt nur noch an diesen fünf.



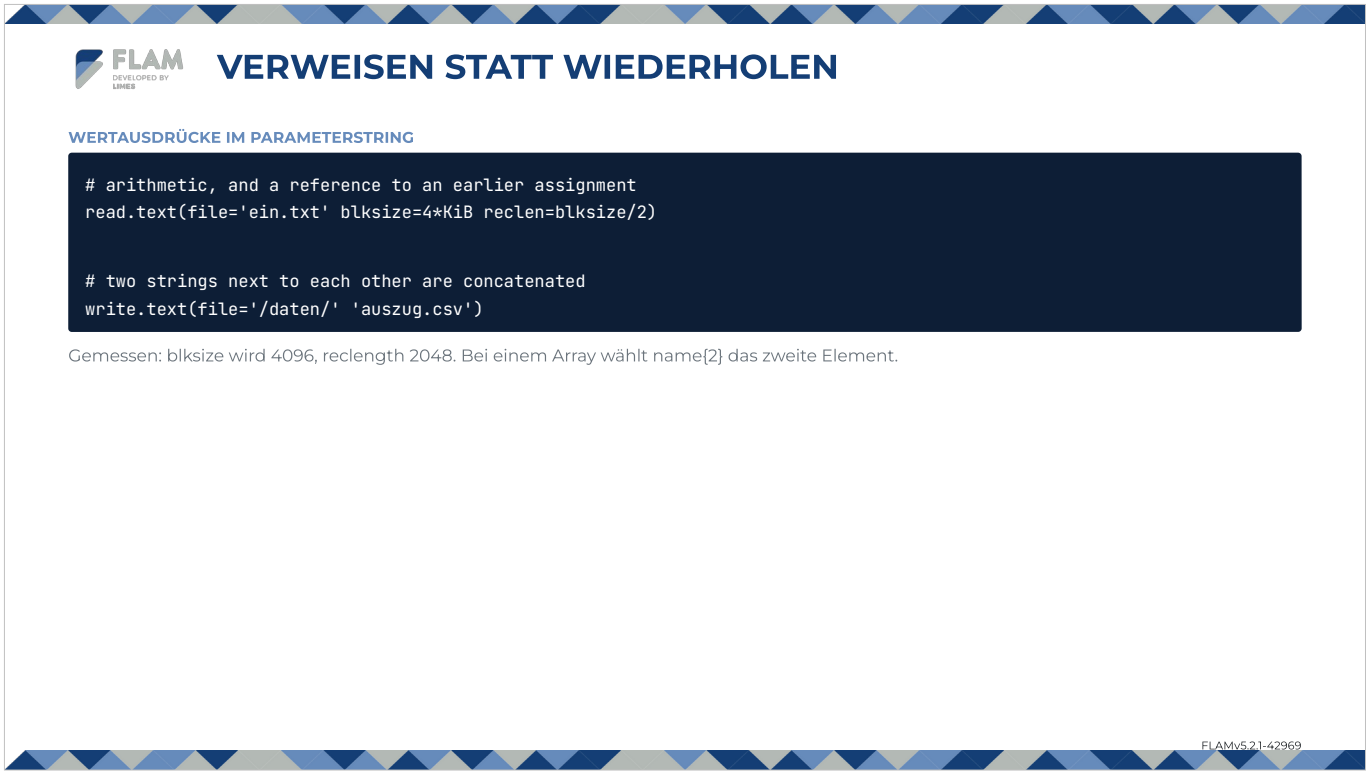
Die Lesehilfe für alles, was zwischen den fünf Formen steht. Nicht vorlesen – drei Zeilen zeigen genügt, der Rest ist zum Nachschlagen.
Die Überschrift ist mit Absicht so: hier steht, woraus die **Sprache** besteht. Der String ist einer ihrer **Typen** und kommt in der Tabelle als Zeile vor – nicht die Tabelle beschreibt den String, sondern der String ist ein Eintrag darin. Und eine Kleinigkeit dazu, die kaum bekannt ist: ein String darf auch in Backticks eingeschlossen werden, nicht nur in `'` oder `"` – so steht es in `flcl lexemes`, und auf einer Tastatur ohne Anführungszeichen ist das Gold wert. Deshalb steht der Backtick auf der Folie ausgeschrieben da: als Zeichen allein würde man ihn leicht übersehen.
Zwei Dinge daraus braucht man wirklich oft:
Kommentare. `#...#` mitten im String, `;` bis zum Zeilenende. Das ist der Grund, warum eine ausgelagerte Parameterdatei lesbar bleibt – man kommentiert, warum eine Angabe da steht.
Separatoren. Leerraum, Steuerzeichen und das Komma sind gleichwertig.
Ein Zeilenumbruch trennt also genauso wie ein Leerzeichen, und deshalb darf ein String in der JCL über beliebig viele Zeilen gehen.
Und eine Kleinigkeit, die Umsteiger freut: ein Keyword darf mit `-` oder `--` beginnen. Wer aus der Unix-Welt kommt, kann `--file=...` schreiben.
Die Kurzform gilt überall: drei Zeichen genügen, solange sie eindeutig sind – aber nicht in Skripten, denn ein neues Keyword kann die Eindeutigkeit morgen aufheben.
Das betrifft nicht nur Parameter: **Kommandos, Funktionen und Objekte dürfen genauso abgekürzt werden.** `CNV` ist `CONV`, `SAV.FIL` ist `SAVE.FILE`, `CMP` ist `COMPRESS`. Unsere Handbuchbeispiele benutzen beide Schreibweisen, und wer die Regel nicht kennt, hält das für zwei verschiedene Objekte. Der Rat bleibt derselbe: im Skript ausschreiben, auf der Kommandozeile abkürzen.



Der Teil, der eine JCL lesbar macht.
Eine Zahl kann in fünf Basen geschrieben werden, und das Log zeigt die Basis hinterher wieder an: `blksize=0x1000` erscheint als `x+1000`. Bei Größen, die ohnehin in Zweierpotenzen gedacht werden, ist die Hexschreibweise näher an der Wahrheit als die Dezimalzahl.
Wichtiger ist das Rechnen: `4*KiB*` *ist gültig und steht für 4096. Das liest sich in der JCL deutlich besser, und man sieht die Absicht statt einer Zahl, die man nachrechnen muss. Dasselbe gilt für Zeitangaben –* `NOW-DAY` *ist gestern.*
Und die Multiplikation darf man weglassen: eine Zahl direkt vor einer Konstanten ist eine Multiplikation, `4KiB` also dasselbe wie `4*KiB*`.
Gemessen: `blksize=4*KiB*` *ergibt 4096.*



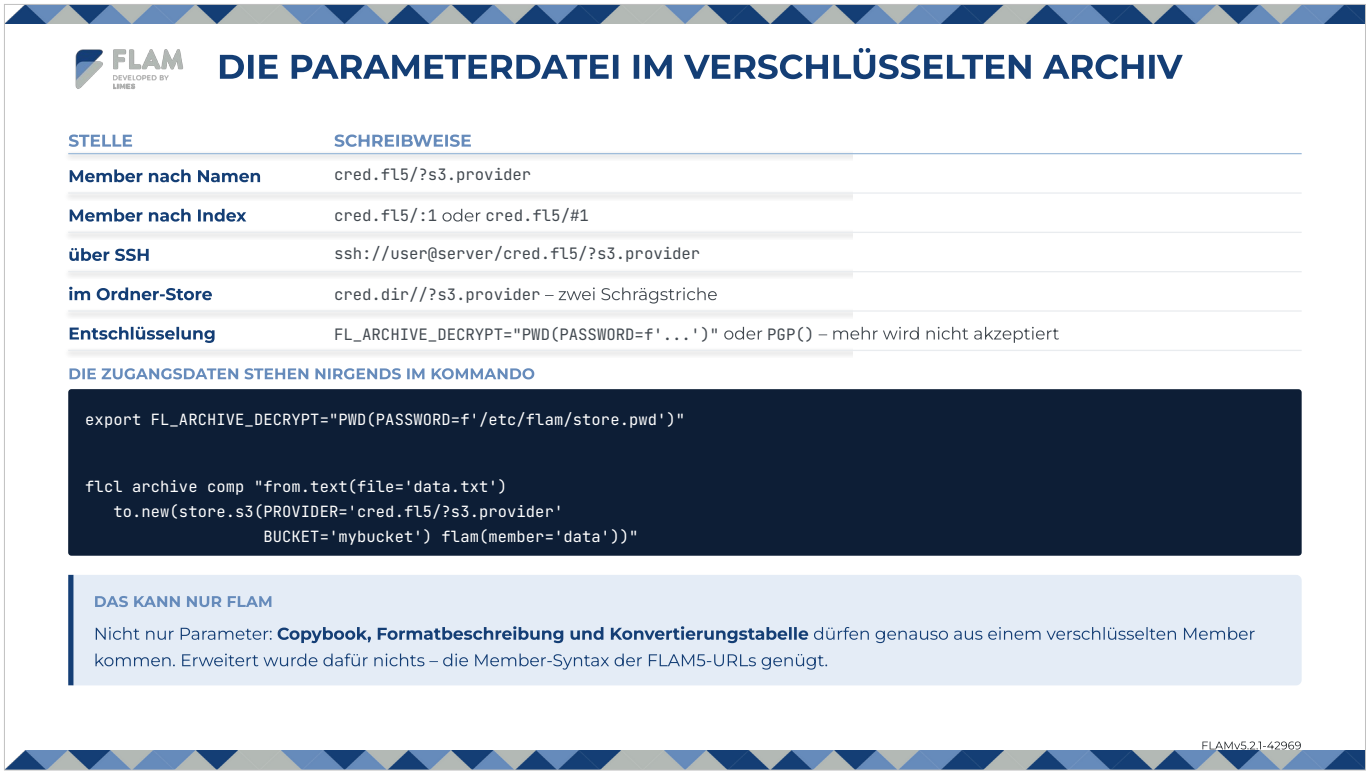
Die Folie, auf der etwas steht, das kaum jemand kennt und das im Zahlungsverkehr Gold wert ist: **`f'...'` liest den Wert aus einer Datei.** Erst die Reihe: ohne Präfix ist es der Normalfall. `s'...'` ist nullterminiert, `c'...'` ein Binärstring im lokalen Zeichensatz, `a'...'` in ASCII, `e'...'` in EBCDIC, `x'...'` in Hexschreibweise. Das braucht man, wenn ein Wert ein Byte ist und kein Text: gemessen ergibt `padcharacter=x'40'` das EBCDIC-Leerzeichen, und der Parser meldet es als `EBCDIC-BLANK(x'40)`.
Und dann `f'...'`: der Inhalt der Datei wird der Wert. Damit steht ein Passwort **nicht** im Parameterstring, nicht in der JCL und nicht in der Prozessliste – es steht in einer Datei, die man mit Dateirechten schützen kann. Das Log zeigt statt des Werts `***SECRET***`. *Gemessen mit `password=f'/pfad/pw.txt'`.*
Zum Schluss die Handgriffe: zwei Hochkommata hintereinander sind ein Hochkomma. Statt `'` gehen auch `"` und der Backquote – praktisch, wenn die Shell eine Sorte schon verbraucht. Und direkt hinter dem Gleichheitszeichen darf ein String ohne Anführungszeichen stehen; er endet dann am nächsten Separator, und Keywords haben Vorrang. In der Schulung sollte man dazu sagen: lesbar ist es, sicher nicht – wer einen Pfad mit Sonderzeichen hat, setzt Anführungszeichen.



Eine Folie für die, die lange Strings pflegen.
Ein Wert darf ein Ausdruck sein: `+ - /` *und Klammern. Und – das ist der Teil, den fast niemand kennt – ein Keyword im Wert bedeutet* **den Wert einer früheren Zuweisung im selben String**.
Gemessen:
`blksize=4*KiB reclen=blksize/2` *ergibt 4096 und 2048. Die Halbierung steht damit einmal da, und wer die Blockgröße ändert, ändert sie an einer Stelle.*
Bei einem Array kommt der Index in geschweifte Klammern: `name{2}`.
Für Strings gilt dasselbe in einfacherer Form: zwei Strings hintereinander werden aneinandergehängt, mit oder ohne `+`. Damit lässt sich ein Dateiname aus einem Präfix und einem Namen zusammensetzen, ohne ihn zweimal zu schreiben.
Der Rat für die Praxis: sparsam einsetzen und kommentieren. Ein Verweis, den der nächste nicht sieht, ist schwerer zu lesen als eine wiederholte Zahl.



Die Folie, mit der man eine unübersichtliche JCL aufräumt.
Nicht nur der ganze Parameterstring kann in einer Datei stehen – **jedes Objekt, jedes Overlay und jedes Array einzeln**. Damit wird aus einer langen Zeile eine kurze mit Verweisen.
Und jetzt die Unterscheidung, die man wirklich können muss, weil sie klein aussieht und drei verschiedene Dinge bedeutet:
Bei einem Objekt oder Overlay sind `=` und `=>` dasselbe – beide sagen "der Inhalt steht in dieser Datei". So steht es in der Grammatik (`flcl grammar`), und deshalb ist die Schreibweise hier Geschmackssache.
Bei einem String sind sie verschieden. `file='DD:PIN'` weist den String `DD:PIN` zu. `file=>'DD:PIN'` liest die **Datei** DD:PIN und macht aus **jeder Zeile** einen Eintrag – bei einem String-Array also eine Dateiliste, eine Zeile je Name oder Wildcard. Nachgemessen: eine Datei mit zwei Zeilen ergibt zwei `FILE=`-Einträge.
Und `file=f'DD:PIN'` ist wieder etwas anderes: dort wird der **ganze Inhalt** der Datei ein einziger String. Das ist die Form für Passphrasen – und der Unterschied zu `=>` ist genau der zwischen "eine Zeile je Element" und "alles ein Wert".
Was das bringt, sind drei Dinge:
Die Datei darf **Kommentare** enthalten – `;` bis zum Zeilenende und `#...#` mitten im Text. In einer JCL-Zeile kann man nicht erklären, warum `PRNCONTROL=REPLACE` dort steht; in einer Parameterdatei schon.
Die Datei ist **wiederverwendbar**. Dieselbe Leseseite kann von mehreren Jobs benutzt werden, und wenn sich das Format ändert, ändert sich die Datei – nicht die JCL.
Und die Datei darf **woanders liegen**: ein DD-Name auf z/OS, ein Pfad, oder eine URL. Bei einer URL gibt man die CCSID mit, dann wird die Datei beim Lesen umkodiert – so kann eine Parameterdatei auf einem Unix-System von einem Host-Job benutzt werden.
Gemessen mit einer Datei, die Kommentare und drei Angaben enthält.



Die Folie, die aus der vorigen eine Sicherheitsfunktion macht – und sie ist neu.
Das Problem zuerst: Ein Kommando trägt Angaben, die nicht jeder lesen soll – die Schlüssel eines Objektspeichers, ein Passwort, die Adresse eines internen Dienstes. Im Kommando stehen sie im Joblog und in der Prozessliste.
In einer gewöhnlichen Parameterdatei sind sie aus dem Kommando heraus, aber im Dateisystem drin.
Die Lösung ist ein Satz: **die Parameterdatei darf ein Member eines FLAM5-Archivs sein, und das Archiv darf verschlüsselt sein.** Damit ist das Archiv ein Credential-Store – die Geheimnisse liegen verschlüsselt, das Kommando nennt nur einen Membernamen, und weder Kommando noch Joblog noch Prozessliste zeigen den Inhalt.
Und das Schöne daran: **am CLP musste dafür nichts erweitert werden.** Die Member-Syntax, die FLAM5-URLs ohnehin haben, reicht – `archiv/?name` nach dem Namen, `archiv/:1` nach dem Index. Das Archiv darf auch entfernt liegen, über SSH oder in einem Ordner-Store; bei einem Ordner-Store steht der zweite Schrägstrich, und danach beginnt der Member.
Entschlüsselt wird über die Umgebungsvariable `FL_ARCHIVE_DECRYPT`, auf z/OS auch als Systemsymbol `&FLARCDC`. Sie trägt die DECRYPT-Angabe und sonst nichts: `PWD(PASSWORD=...)` oder `PGP()`.
Warum nur diese beiden? Das ist der Teil, den man im Raum erzählen sollte, weil er zeigt, wie hier gedacht wird: Das Parameterobjekt um DECRYPT herum trägt auch PREPROCESS und POSTPROCESS, und die **führen Kommandos aus**. Eine Variable, die bis dorthin reichte, würde aus jedem Lesen einer Parameterdatei einen Weg machen, Programme zu starten. Aus demselben Grund ist innerhalb von PGP die Angabe einer Krypto-Engine abgelehnt – eine Engine ist ein Modul, das geladen und ausgeführt würde.
Das Passwort muss auch nicht in der Variablen stehen: `PASSWORD=f'...'` holt es aus einer Datei mit eigenen Rechten. Und auf z/OS braucht man oft gar keines – ist das Archiv für einen OpenPGP-Schlüssel verschlüsselt, liegt der private Schlüssel in ICSF und es gibt kein Passwort, das irgendwo stehen könnte.
Und der Teil, der oft unterschätzt wird: **das gilt nicht nur für Parameter.** Gelesen wird über die Datei-zu-String-Schicht, und die bedient jede Texteingabe – also auch ein **COBOL-Copybook**, ein PL/I-Include, eine Assembler-Struktur, eine **Formatbeschreibung** oder eine **Konvertierungstabelle**. Damit kann auch die Beschreibung Ihrer Daten verschlüsselt liegen, nicht nur das Passwort dazu. Nachgemessen: eine Row-Spezifikation aus einem verschlüsselten Member, Konvertierung läuft durch.
Eine Abgrenzung, die im Gespräch sofort kommt: **`FL_ARCHIVE_DECRYPT` ist nicht die Entschlüsselung des Archivs, das ein Kommando verarbeitet.** Ein `DECO` eines verschlüsselten Archivs nimmt sein `DECRYPT()` aus dem Kommando. Die Variable gilt nur für ein Archiv, aus dem eine **Eingabe gelesen** wird.
Eine Eigenschaft noch, die man kennen muss: **die Member-Angabe ist die Entscheidung.** Ein Name ohne Member ist eine gewöhnliche Datei. Ein Name mit Member ist ein Archiv – und wenn es sich nicht öffnen lässt, der Member fehlt oder die Entschlüsselung scheitert, **bricht das**

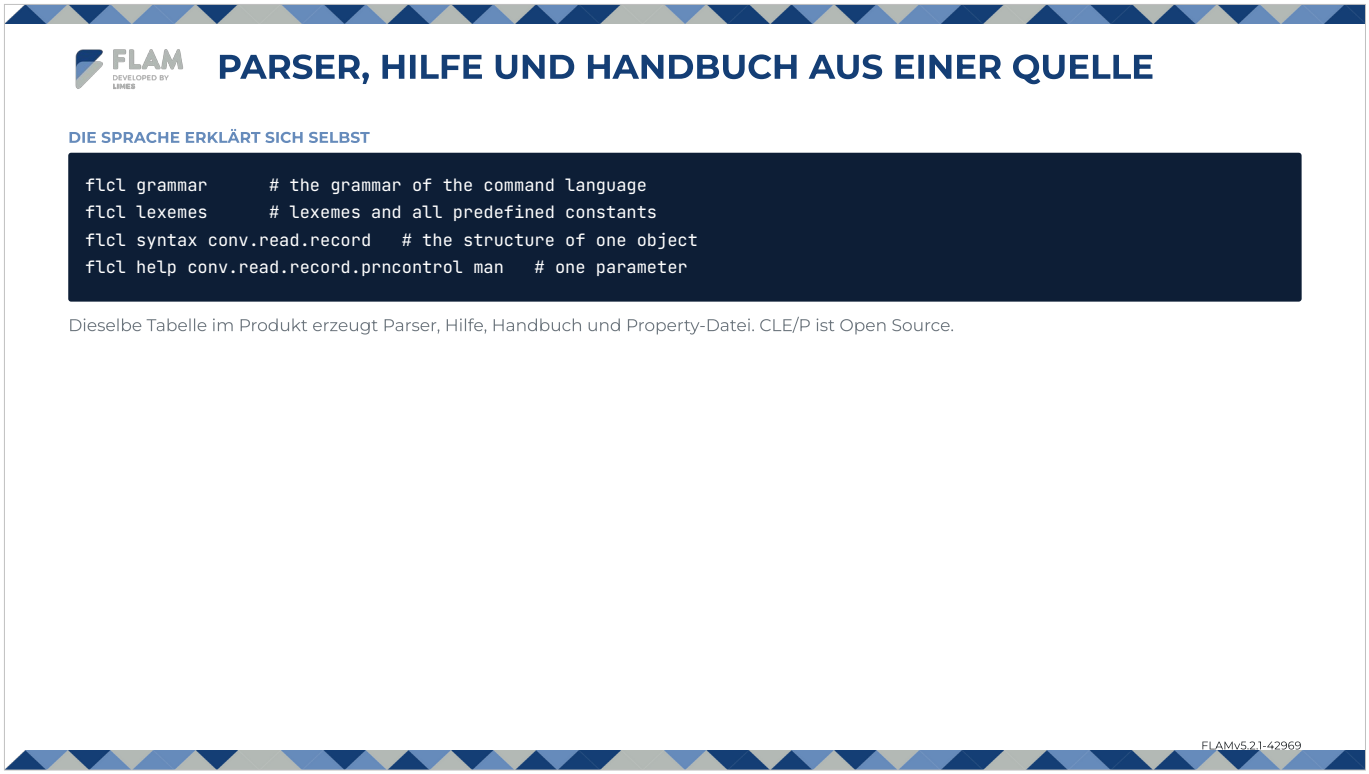
WAS DER PARSER SAGT, WENN ETWAS FEHLT

EIN VERTIPPTES KEYWORD – DIE AUSGABE VON UNTEN NACH OBEN LESEN

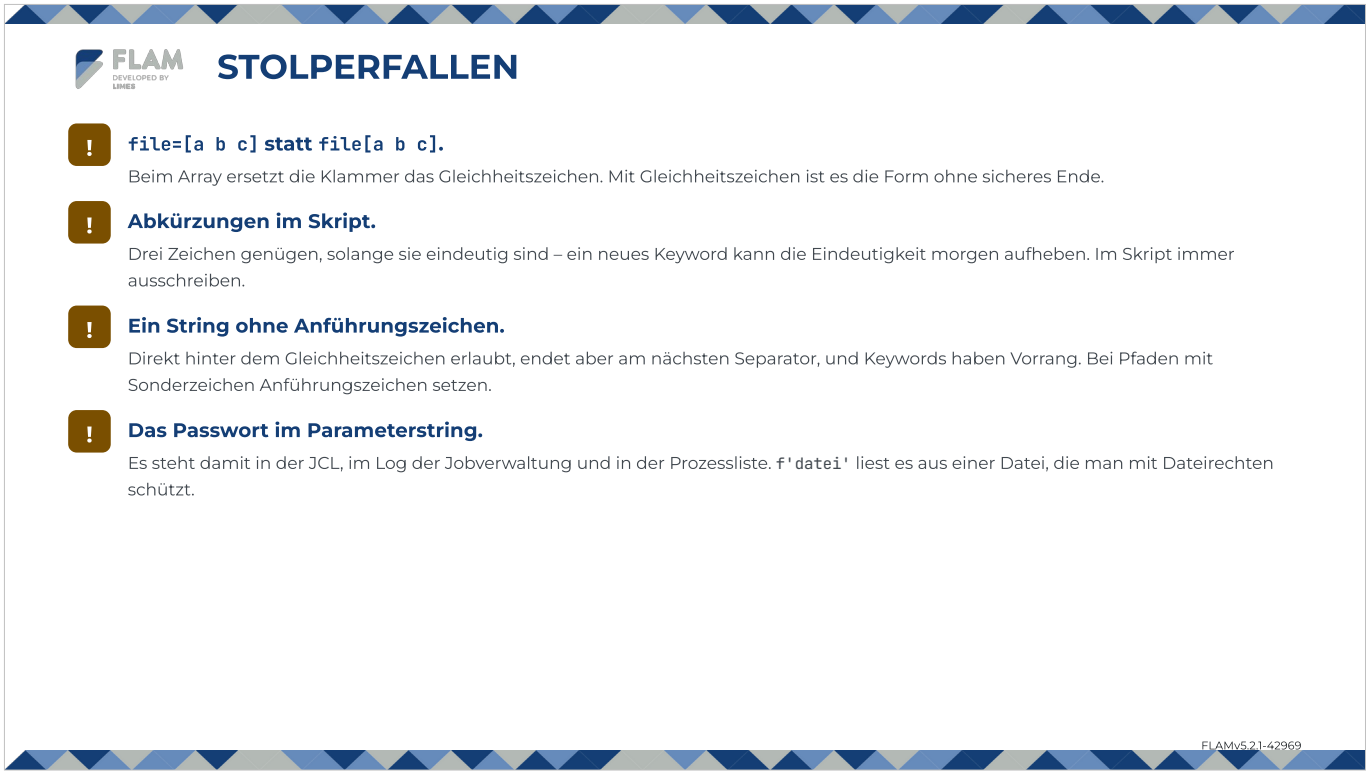
```
SYNTAX-ERROR:
Character ')' missing to enclose object (CONV.READ)
 - String (d'ccsidx=1141) given -> possible wrong keyword
  Cause: Row=1 Column=31 from command line
  "read.text(file='/tmp/ein.txt' ccsidx=1141)"
                                 ^^^^^^^^^^^
After successful parsing of arguments below:
  CONV.READ.TEXT(
  CONV.READ.TEXT.FILE=d'/tmp/ein.txt
```

FLAMv5.21-42969

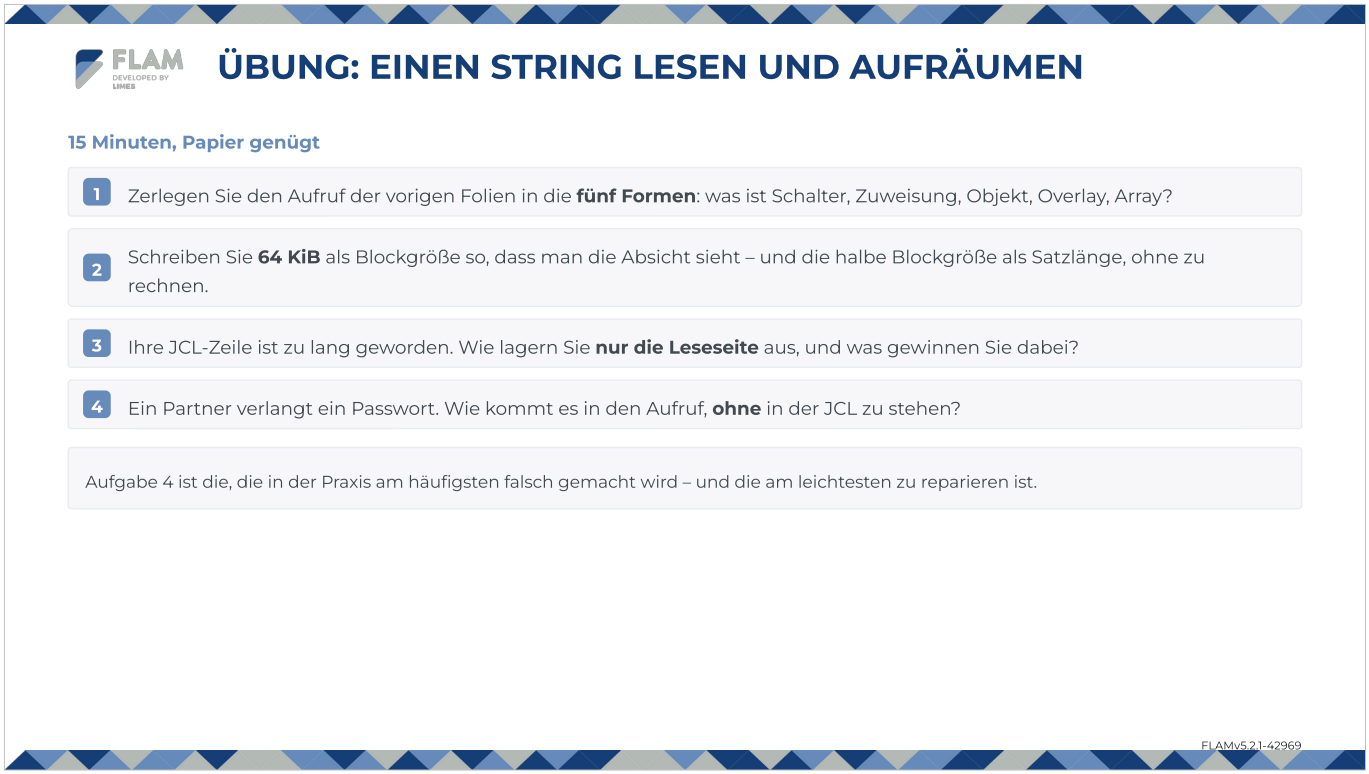
Diese Folie spart in der Produktion mehr Zeit als jede andere dieser Einheit – und sie wird übersehen, weil niemand eine Fehlermeldung liest.
Die Meldung hat drei Teile, und man liest sie **von unten nach oben**:
Unten steht, **was erfolgreich verstanden wurde**. Das ist die eigentliche Information: bis hierhin war alles in Ordnung.
In der Mitte steht die **Stelle** – Zeile, Spalte, die Eingabe selbst und darunter eine Markierung unter dem Zeichen bzw. dem Wort, an dem es abbrach.
Oben steht, **was erwartet wurde**. Im Beispiel: eine schließende Klammer fehlt – und der Parser sagt gleich dazu, was wahrscheinlich wirklich los ist, nämlich ein falsch geschriebenes Keyword.
Der Satz für den Raum: der Parser weiß nicht, was Sie gemeint haben, aber er weiß genau, wo er stehengeblieben ist. Das ist mehr, als die meisten Werkzeuge bieten.
Beide gezeigten Meldungen sind echte Ausgaben.



Die Folie, die erklärt, warum die Hilfe nie veraltet – kurz halten, die Doku-Einheit hat das ausführlich.
Die Kommandostruktur steht **einmal** in einer Tabelle im Produkt. Daraus entstehen der Parser, die eingebaute Hilfe, die Handbuchseiten und die Property-Dateien. Es gibt also keine zweite Stelle, an der die Beschreibung eines Parameters liegt – und damit auch keine, die man vergessen kann.
Zwei eingebaute Funktionen machen das nachprüfbar, und beide sollte man im Raum einmal laufen lassen: `flcl grammar` gibt die Grammatik der Sprache aus, `flcl lexemes` die Lexeme samt der vordefinierten Konstanten. Alles, was auf den vorigen Folien stand, kommt von dort.
Und CLE/P ist Open Source. Wer eigene Werkzeuge baut, kann dieselbe Kommandozeile benutzen – inklusive der Hilfe und der Handbuchgenerierung.



Alle vier aus Supportfällen und aus Schulungen.



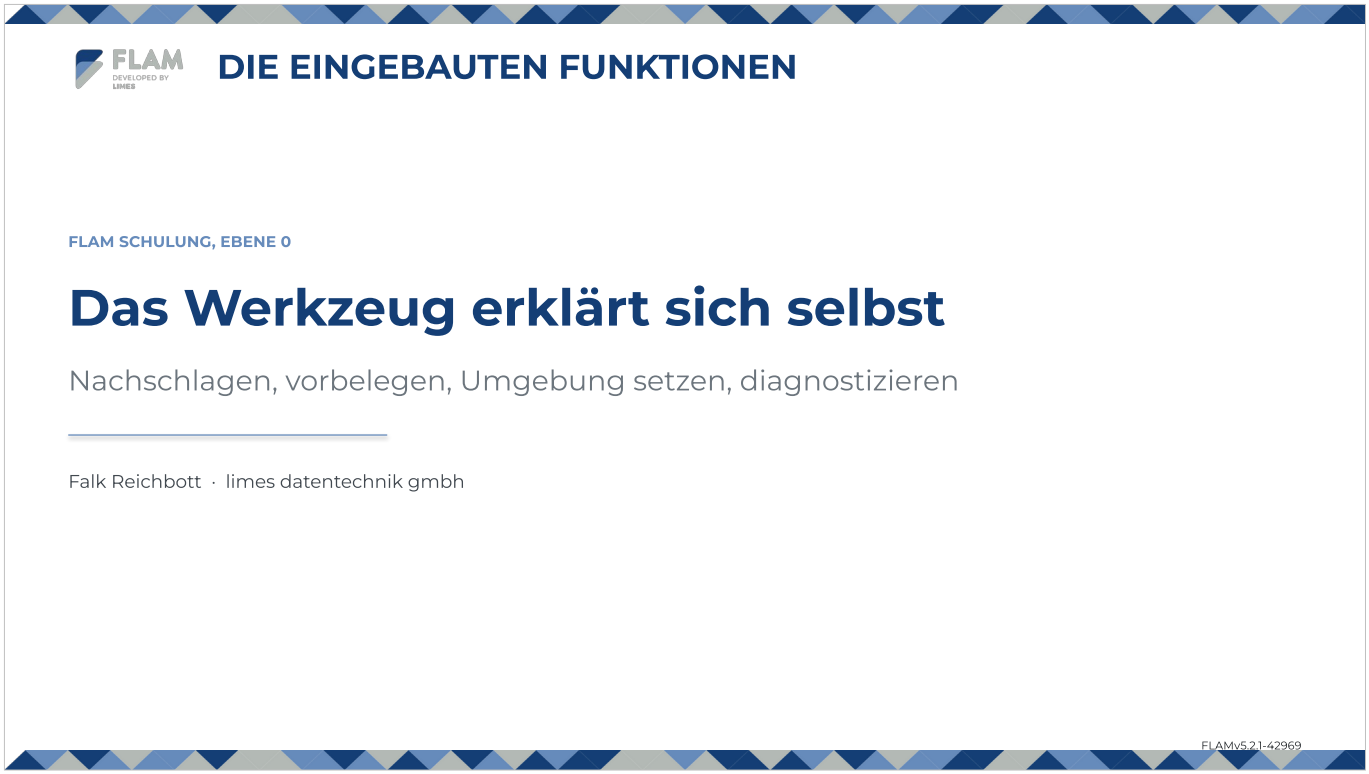
Fünfzehn Minuten. Die Übung ist absichtlich Lesearbeit – es gibt nichts zu tippen, es gibt etwas zu verstehen.
Die Lösungen:
Aufgabe 1. `read.record` ist ein Overlay (eine von mehreren Varianten), `file=...` eine Zuweisung, `suptws` ein Schalter, `hash(...)` ein Objekt, und `file['a' 'b']` wäre ein Array.
Aufgabe 2. `blksize=64*KiB*` – *und wer die Hälfte als Satzlänge will, schreibt `reclen=blksize/2` statt der Zahl.*
Aufgabe 3. Die Leseseite in eine Datei und mit `read.record=>'DD:RDPAR'` verweisen. In der Datei die Angaben kommentieren – das ist der eigentliche Gewinn.
Aufgabe 4. `password=f'/pfad/pw.txt'`. Das Passwort steht dann in einer Datei mit Dateirechten, nicht im Aufruf; das Log zeigt `***SECRET***`.

WAS BLEIBT

- **CLE** ist der Rahmen, **CLP** liest den Parameterstring – deshalb läuft derselbe String auf jedem Weg
- **fünf Formen**: Schalter, Zuweisung, Objekt, Overlay, Array – mehr gibt es nicht
- Zahlen in fünf Basen, und man kann **rechnen**: `blksize=4*KiB`
- `f'datei'` holt den Wert aus einer Datei – der Griff für **Passwörter**
- jedes Objekt darf über `=>` aus einer **eigenen Datei** kommen, mit Kommentaren
- die Fehlermeldung nennt **Spalte und Stand** – von unten nach oben lesen

FLAMv5.21-42969

Lernziele durchgehen. Und der Verweis nach vorn: die eingebauten Funktionen haben ihre eigene Einheit, die Dateiauswahl ebenso – beides ist Sprache, aber beides ist groß genug für eine eigene halbe Stunde.



Die Schwesterneinheit zur Kommandosprache: dort ging es um den Parameterstring, hier um den Rahmen.
Sechsundzwanzig Funktionen, die nichts mit Daten zu tun haben und trotzdem den Alltag bestimmen: nachschlagen, vorbelegen, Umgebung setzen, diagnostizieren. Der Bogen führt von der Rückgabe des Kommandos über die Property-Dateien bis zu dem, was in ein Ticket gehört.
Der Satz für den Anfang: **das Werkzeug erklärt und konfiguriert sich selbst.** Es gibt keine Konfigurationsdatei, die man erraten muss, und keine Parameterliste, die man von einer Webseite abschreibt.



Das zweite Ziel ist das betrieblich wichtigste: wer `MAXCC` falsch setzt, verliert einen echten Fehler.
Das vierte ist der Griff, mit dem man einen wiederkehrenden Aufruf kurz hält, ohne ein Skript zu schreiben.

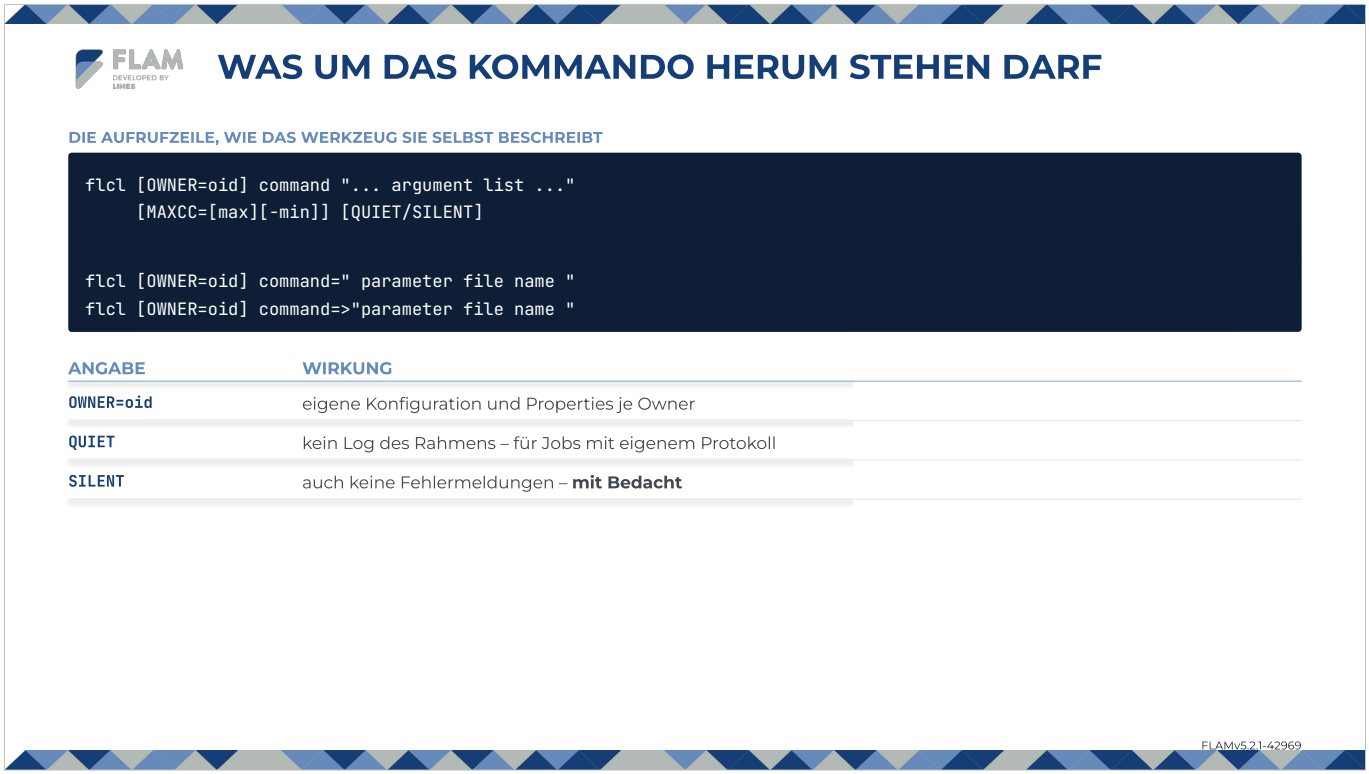
SECHSUNDZWANZIG FUNKTIONEN, FÜNF GRUPPEN

GRUPPE	FUNKTIONEN	WOFÜR
Nachschlagen	`SYNTAX`, `HELP`, `MANPAGE`	die Struktur und die Bedeutung eines Parameters
Erzeugen	`GENDOCU`, `HTMLDOC`, `GRAMMAR`, `LEXEMES`, `ERRORS`	Handbuch, Grammatik, Returncodes
Vorbelegen	`GENPROP`, `SETPROP`, `CHGPROP`, `GETPROP`, `DELPROP`	Angaben, die immer gleich sind
Umgebung	`SETENV`, `GETENV`, `DELENV`, `LSTENV`, `HLPENV`, `CONFIG`, `SETOWNER`, `GETOWNER`	was gilt, ohne im Aufruf zu stehen
Diagnose	`VERSION`, `ABOUT`, `LICENSE`, `TRACE`	was in ein Ticket gehört

Keine dieser Funktionen verarbeitet Daten. Sie sind der Rahmen – und der Rahmen ist der Teil, der in der Produktion Zeit kostet oder spart.

FLAMv5.21-42969

Die Übersicht. Nicht vorlesen – die Gruppen nennen und sagen, dass die Einheit sie in dieser Reihenfolge durchgeht.
Der Punkt, den man dazu machen sollte: keine dieser Funktionen verarbeitet Daten. Sie sind das Drumherum, und genau deshalb kennt sie kaum jemand – bis man sie braucht.
Eine Einordnung, die hilft: die ersten zwei Gruppen ersetzen das Handbuch, die dritte ersetzt Kopieren und Einfügen, die vierte ersetzt Skripte um den Aufruf herum, und die fünfte ist das, was man in ein Ticket schreibt.



Die Folie erklärt die Zeile, die in jeder JCL steht, und die drei Angaben, die man dort setzen kann. `OWNER=` wählt eine eigene Konfiguration. Auf einer Maschine mit mehreren Mandanten oder mehreren Anwendungen ist das der Weg, ihre Vorbelegungen auseinanderzuhalten – jeder Owner hat seine eigenen Properties und Umgebungsangaben.
`MAXCC=` ist die Returncode-Klammer, und die hat eine eigene Folie, weil sie gefährlich ist. `QUIET` schaltet das normale Log des Kommandozeilen-Rahmens ab, `SILENT` zusätzlich die Fehlermeldungen. In einem Job, der sein eigenes Protokoll schreibt, ist `QUIET` sinnvoll. `SILENT` sollte man nur setzen, wenn man sicher ist – danach sieht man nicht mehr, warum etwas schiefging.
Und die dritte Form ist die wichtigste für lange Strings: statt der Argumentliste darf ein **Dateiname** stehen. `command="datei"` oder `command=>"datei"` – damit ist der ganze Parameterstring ausgelagert.

MAXCC: DIE KLAMMER UM DEN RETURNCODE

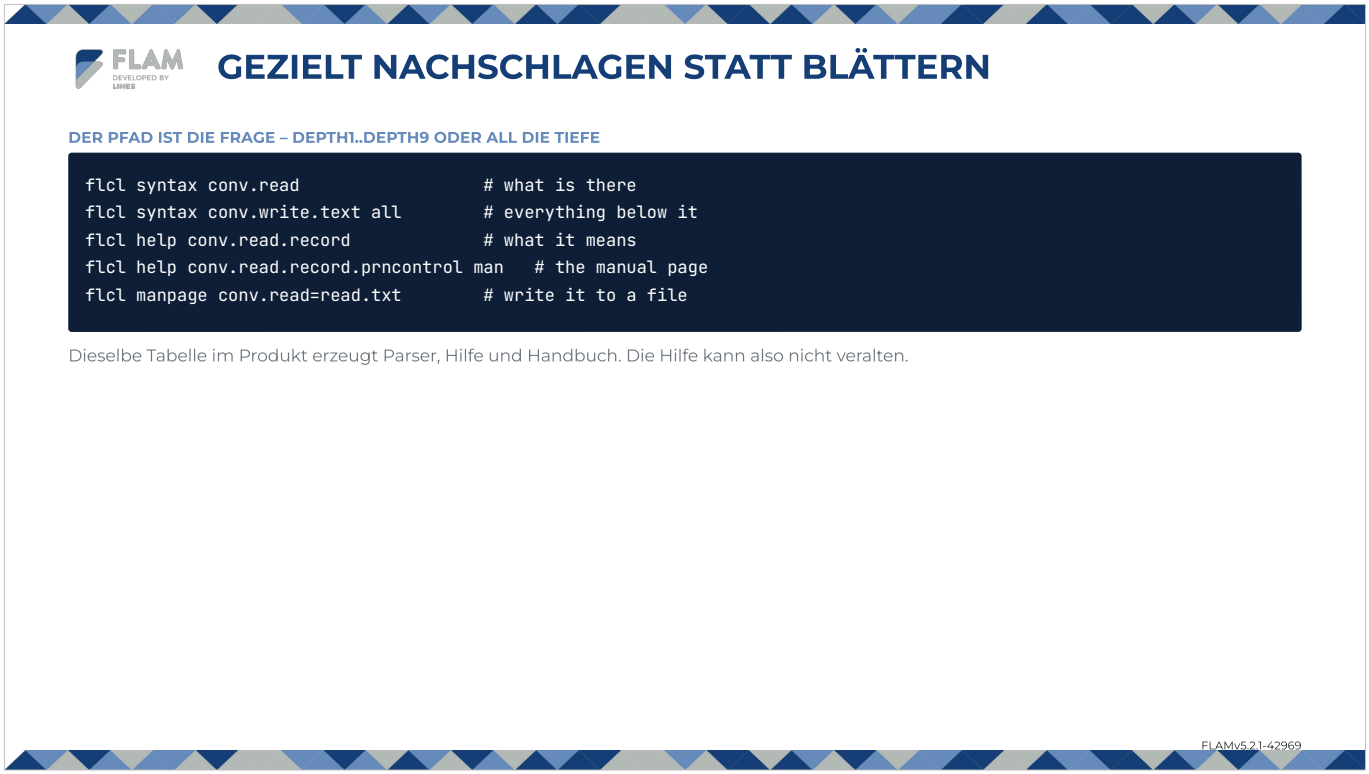
ANGABE	LAUF MIT CC 1	LAUF MIT CC 8
ohne	**1**	**8**
`MAXCC=0`	**0** – Warnung verschwindet	0 – **auch der Fehler**
`MAXCC=4`	1	**4** – der Fehler wird kleiner gemeldet
`MAXCC=8-4`	**0** – unter der Grenze	**8** – bleibt sichtbar

DAS KANN NUR FLAM

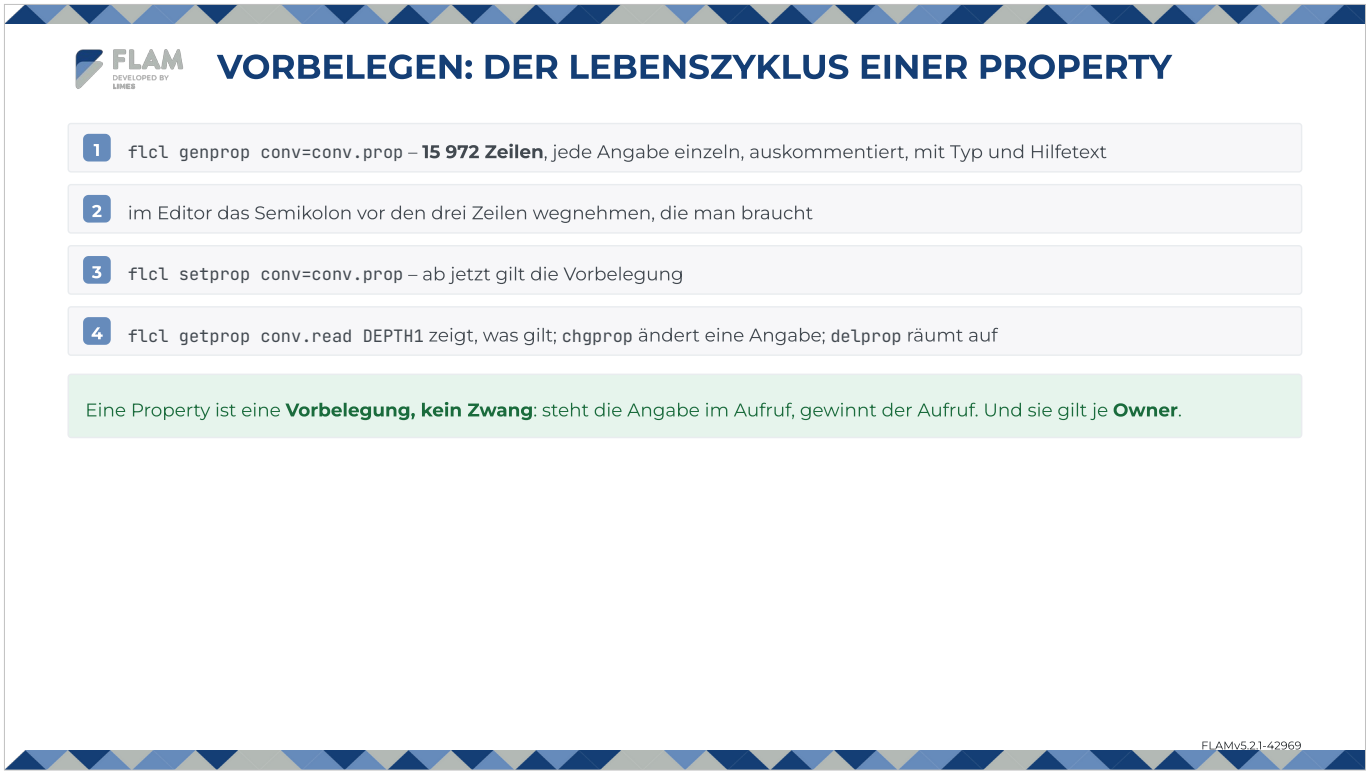
`-min` gibt 0 zurück, wenn der Code **kleiner** als die Grenze ist – Kleinigkeiten verschwinden, Fehler bleiben. Das ist die Angabe, die man will.

FLAMv5.21-42969

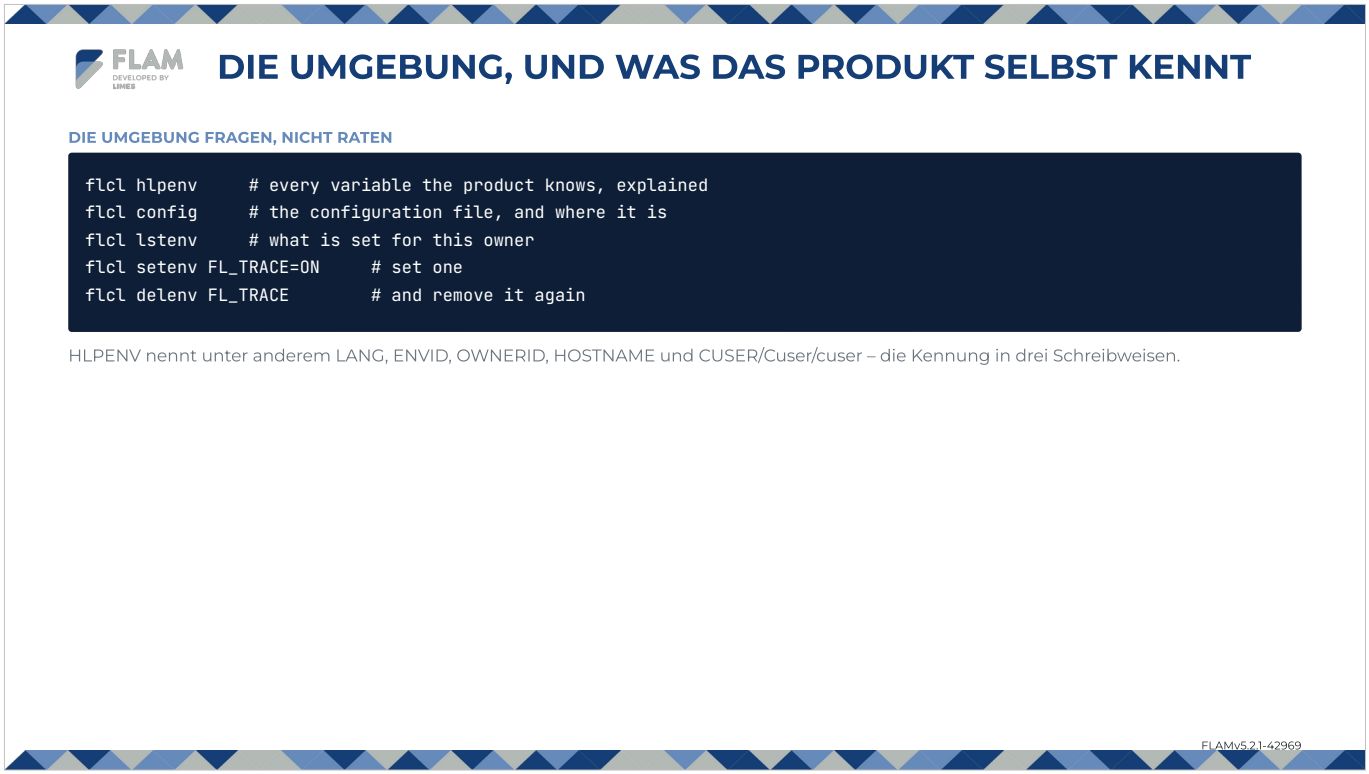
Die wichtigste Folie der Einheit, und sie enthält eine Falle, die in Produktionsjobs sitzt. `MAXCC=max` deckelt den zurückgegebenen Code. Das ist gedacht, um Warnungen zu unterdrücken – ein Lauf mit Completion Code 1 gibt mit `MAXCC=0` eine 0 zurück, und der Job läuft weiter.
Die Falle: der Deckel gilt auch für **echte Fehler**. Gemessen: ein Lauf, der ohne Angabe mit 8 endet, endet mit `MAXCC=4` als **4**. Der Fehler ist nicht weg, er ist nur kleiner geworden – und wenn die Jobsteuerung erst ab 8 reagiert, reagiert sie nicht mehr.
Der zweite Teil, `-min`, arbeitet anders, als der Name vermuten lässt: ist der Code **kleiner** als min, wird 0 zurückgegeben. Das ist die freundlichere Variante: `MAXCC=8-4` gibt für einen Lauf mit 1 eine 0 und für einen Lauf mit 8 die 8. Kleinigkeiten verschwinden, Fehler bleiben.
Die Empfehlung für den Raum: **nicht deckeln, sondern eine Untergrenze setzen.** `MAXCC=8-4` statt `MAXCC=4`. Alle drei Zeilen der Tabelle sind gemessen.



Drei Funktionen, die dasselbe Ziel von drei Seiten angehen. Der Unterschied ist die Tiefe.
`SYNTAX` gibt die **Struktur**: welche Parameter es an dieser Stelle gibt, in einer Zeile. Das ist die Funktion für die Frage "wie heißt das nochmal".
`HELP` gibt die **Bedeutung** – je Parameter eine Zeile Erklärung. Mit `MAN` dahinter die ganze Handbuchseite zu genau diesem Parameter.
`MANPAGE` schreibt die Handbuchseite in eine Datei, auch für eine ganze Kommandogruppe.
Der Griff, der die Sache praktisch macht, ist der **Pfad**. Man fragt nicht das Kommando, sondern die Stelle: `conv.read.record.prncontrol`. Und `DEPTH1` bis `DEPTH9` oder `ALL` bestimmen, wie viele Ebenen mitkommen – sonst erschlägt einen die Antwort.
In der Schulung einmal vormachen: `flcl syntax conv.read` zeigt die Varianten, `flcl help conv.read.record.prncontrol man` erklärt eine einzelne Angabe. Das ist schneller als jedes PDF.



Die Folie für alle, die denselben Parameter in zwanzig Jobs wiederholen.
Der Weg hat drei Schritte, und `GENPROP` ist der, der überrascht:
`flcl genprop conv=datei` schreibt **jeden** Parameter des Kommandos als auskommentierte Zeile in eine Datei, mit Typ und Hilfetext daneben.
Gemessen: für CONV sind das **15 972 Zeilen**. Man liest sie nicht – man sucht die drei, die man braucht, und nimmt das Semikolon davor weg.
`SETPROP` liest die Datei ein und macht sie wirksam. `CHGPROP` ändert eine einzelne Angabe ohne Datei. `GETPROP` zeigt, was gerade gilt, und `DELPROP` räumt auf.
Was eine Property ist: eine **Vorbelegung**, kein Zwang. Steht die Angabe im Aufruf, gewinnt der Aufruf. Deshalb ist es der richtige Ort für Hausstandards – die Blockgröße, die CCSID, der Virenscanner – und der falsche für alles, was sich je Lauf unterscheidet.
Ein Hinweis, der Ärger erspart: eine Property gilt je **Owner**. Wer sie für den ganzen Rechner setzen will, muss den Owner kennen, unter dem die Jobs laufen.



Zwei Dinge auf einer Folie, und das zweite kennt fast niemand.
Erstens die Verwaltung: `SETENV` setzt eine Variable in der Konfiguration des Owners, `GETENV` und `LSTENV` zeigen sie, `DELENV` entfernt sie.
`CONFIG` zeigt die ganze Konfigurationsdatei samt ihrem Ort – das ist die erste Frage bei jedem "bei mir geht es nicht".
Zweitens, und das ist der Fund: **`HLPENV` listet jede Umgebungsvariable, die das Produkt überhaupt beachtet**, mit Erklärung. Man muss also nicht suchen, welche Variable es gibt – man fragt.
Ein paar aus der Liste, die im Raum meist eine Frage auslösen: `LANG` bestimmt Sprache und CCSID, `ENVID` den Umgebungsqualifier für die Schlüssellabel-Vorlagen, `OWNERID` den Owner, wenn keiner gesetzt ist, und `CUSER`/`Cuser`/`cuser` geben die Benutzerkennung in drei Schreibweisen – praktisch in Dateinamen.
Der Rat: die Umgebung gehört in eine kontrollierte Startumgebung, nicht in das Profil des Anwenders. Die Sicherheitseinheit sagt, warum.

DER OWNER TRENNT DIE KONFIGURATIONEN

EIN RECHNER, MEHRERE VORBELEGUNGEN

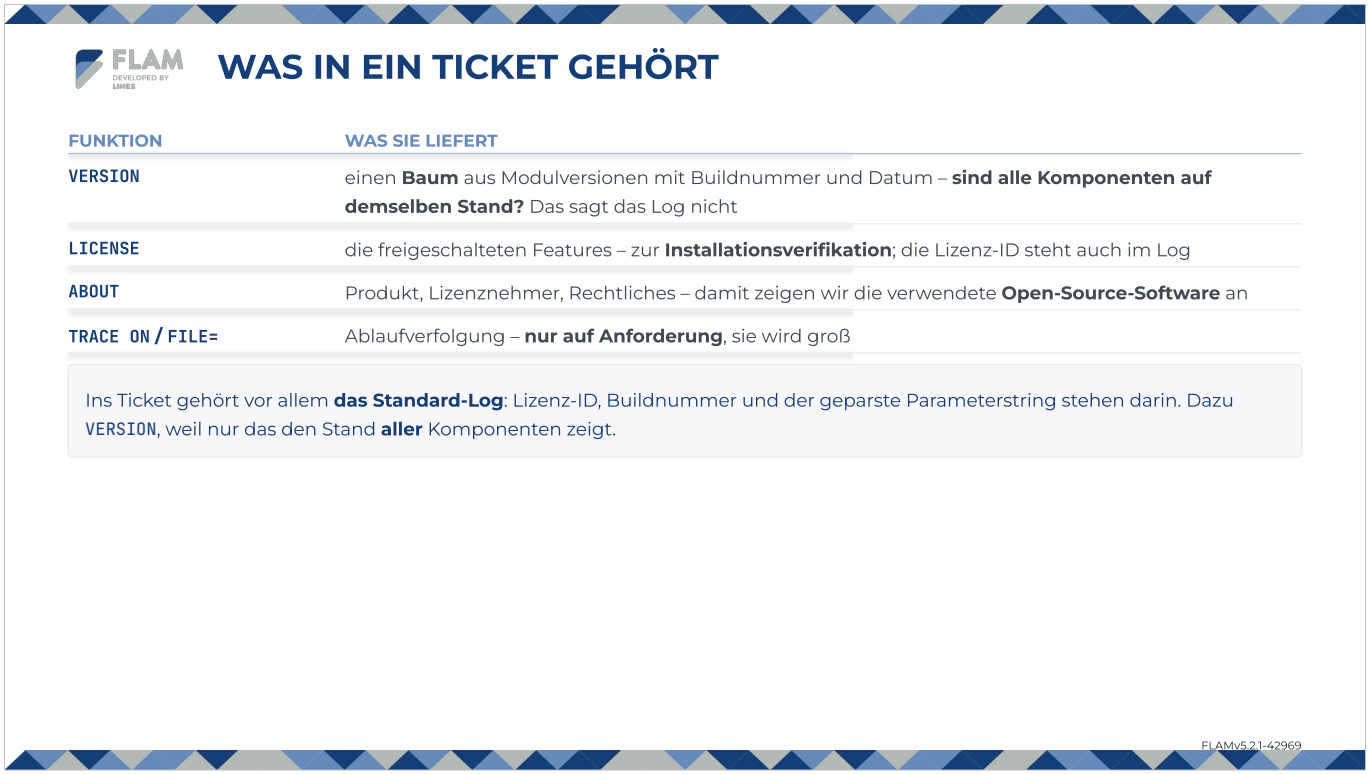
```
flcl getowner                 # who am I running as
flcl setowner zahlungsverkehr # set it for good
flcl OWNER=test CONV "..."    # once, for this call only
```

In der Konfigurationsdatei beginnt jede Zeile mit dem Ownernamen – deshalb stören sich zwei Anwendungen nicht.

Für einen Test denselben Aufruf mit `OWNER=test` gegen eine andere Vorbelegung fahren – ohne die produktive Konfiguration anzufassen.

FLAMv5.21-42969

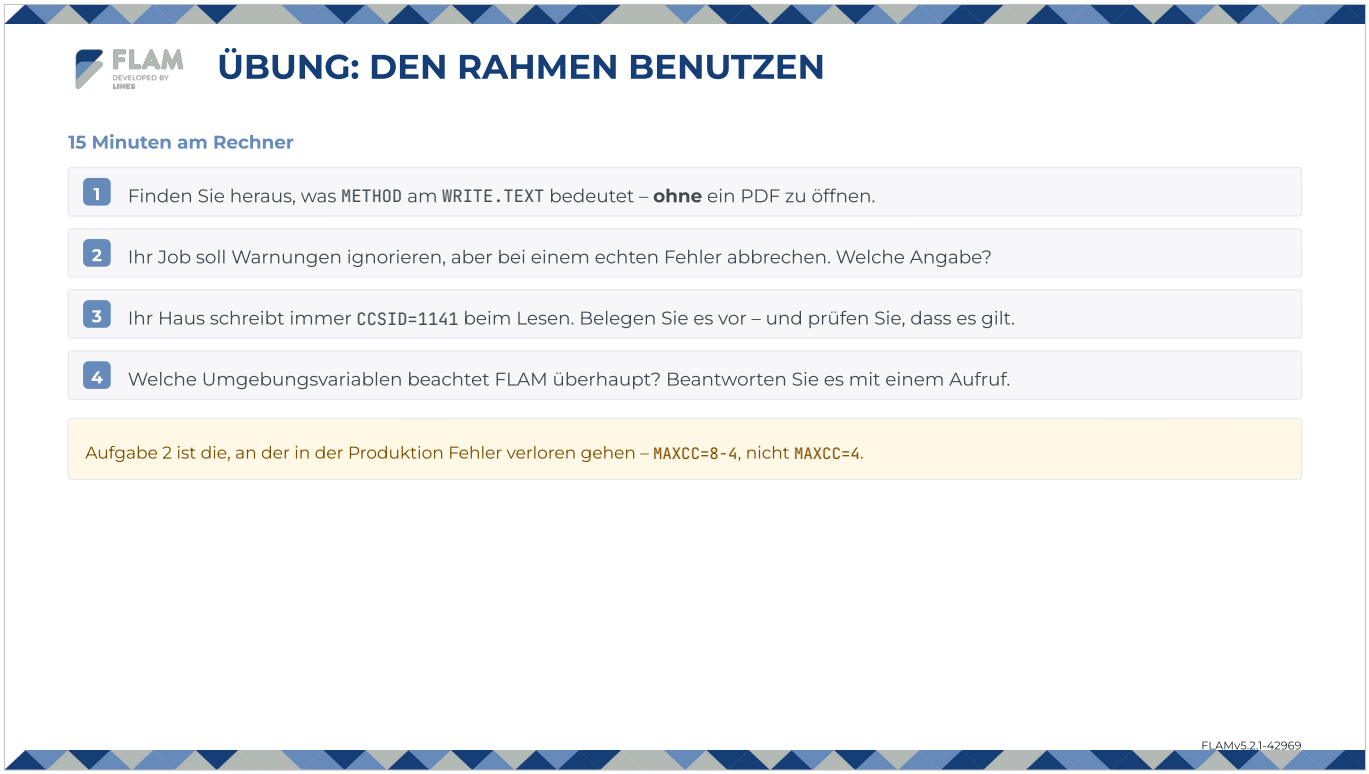
Eine kurze Folie mit einer Antwort auf eine Frage, die in jedem Rechenzentrum kommt: wie halte ich zwei Anwendungen auf einer Maschine auseinander?
Über den Owner. `SETOWNER` setzt ihn, `GETOWNER` zeigt ihn, und `OWNER=` am Aufruf überschreibt ihn für einen einzelnen Lauf. Jeder Owner hat seine eigenen Properties und seine eigenen Umgebungsangaben in derselben Konfigurationsdatei – man sieht es an den Zeilen, die `CONFIG` ausgibt:
sie beginnen mit dem Ownernamen.
Was das praktisch bringt: die Zahlungsverkehrsanwendung bekommt ihre Vorbelegungen, das Archiv seine, und niemand ändert versehentlich die des anderen. Und für einen Test kann man denselben Aufruf mit `OWNER=test` gegen eine andere Vorbelegung fahren, ohne etwas umzustellen.



Die Folie, die dem Support Arbeit erspart – und damit dem Anwender Wartezeit.
`VERSION` gibt nicht eine Zeile, sondern einen **Baum**: die Version jedes beteiligten Moduls mit Buildnummer und Datum. Das ist die Angabe, mit der der Support anfängt, weil sie sagt, welcher Code lief.
`ABOUT` nennt Produkt, Lizenznehmer und Rechtliches, `LICENSE` die freigeschalteten Features – die Frage "darf das diese Installation überhaupt" ist damit in einer Zeile beantwortet.
`TRACE ON` schaltet die Ablaufverfolgung ein, `TRACE FILE=name` schreibt sie in eine Datei. Das ist die Angabe, die der Support anfordert, wenn die Logmeldungen nicht reichen – und man sollte sie **nur auf Anforderung** einschalten, weil sie groß wird und langsam macht.
Und jetzt die Einordnung, die man geben muss, damit niemand die falsche Sammelarbeit macht: **das Wichtigste für ein Ticket ist unser Standard-Log.** Dort stehen die Lizenz-ID – über die kennen wir die freigeschalteten Features –, die Buildnummer und der **geparste Parameterstring**. Wer das Log schickt, hat damit schon fast alles geschickt. Die einzige Angabe, die das Log nicht hergibt, ist der **Abgleich der Buildnummern über alle Komponenten**; dafür braucht es die Ausgabe von `VERSION`.
Wofür die drei Funktionen also wirklich da sind: `VERSION` und `LICENSE` dienen dem **Kunden** zur Installationsverifikation – sind alle Komponenten auf demselben Stand, und arbeite ich mit meiner produktiven Lizenz? Und `ABOUT` erfüllt unsere Pflicht, die verwendete Open-Source-Software anzuzeigen.
Für ein Ticket also: **das Log**, dazu `VERSION` für die Konsistenz der Installation, und der Parameterstring, falls er nicht im Log steht. Details in der Doku-Einheit.



Alle vier aus der Produktion.



Fünfzehn Minuten am Rechner.
Die Lösungen:
Aufgabe 1. `flcl help conv.write.text.method man` – der Pfad ist die Frage, `man` gibt die Handbuchseite.
Aufgabe 2. `MAXCC=8-4`. Damit verschwinden die Codes unter 4, und ein echter Fehler bleibt 8. `MAXCC=4` wäre falsch, weil es die 8 auf 4 drückt.
Aufgabe 3. `flcl genprop conv=conv.prop`, im Editor die Zeile für die CCSID freigeben, `flcl setprop conv=conv.prop`. Und danach mit `flcl getprop conv.read DEPTH1` nachsehen.
Aufgabe 4. `flcl hlpenv` – die Frage ist damit beantwortet, ohne zu suchen. Wer die Liste einmal gesehen hat, fragt sie künftig zuerst.

WAS BLEIBT

- `MAXCC=8-4` **statt** `MAXCC=4` – ein Deckel verkleinert auch echte Fehler
- der **Pfad** ist die Frage: `flcl help conv.read.record.prncontrol man`
- `GENPROP` schreibt **jede** Angabe auskommentiert heraus – man streicht das Semikolon vor den dreien, die man braucht
- eine Property ist eine **Vorbelegung**, der Aufruf gewinnt – und sie gilt je **Owner**
- `HLPENV` **nennt jede Umgebungsvariable, die das Produkt beachtet**
- ins Ticket gehört vor allem **das Log** – Lizenz-ID, Buildnummer und geparster Parameterstring stehen darin; `VERSION` zeigt den Stand **aller** Komponenten

FLAMv5.21-42969

Lernziele durchgehen. Und der Hinweis: die Rangfolge der fünf Orte, an denen eine Angabe herkommen kann, steht in der Einheit *Administration* – hier ging es darum, mit welcher Funktion man sie anfasst.

FLAM SCHULUNG, EBENE 0

Ein Aufruf statt einer Schleife

Wildcards, Member, Directory-Walk – und wie der Ausgabename entsteht

Falk Reichbott · limes datentechnik gmbh

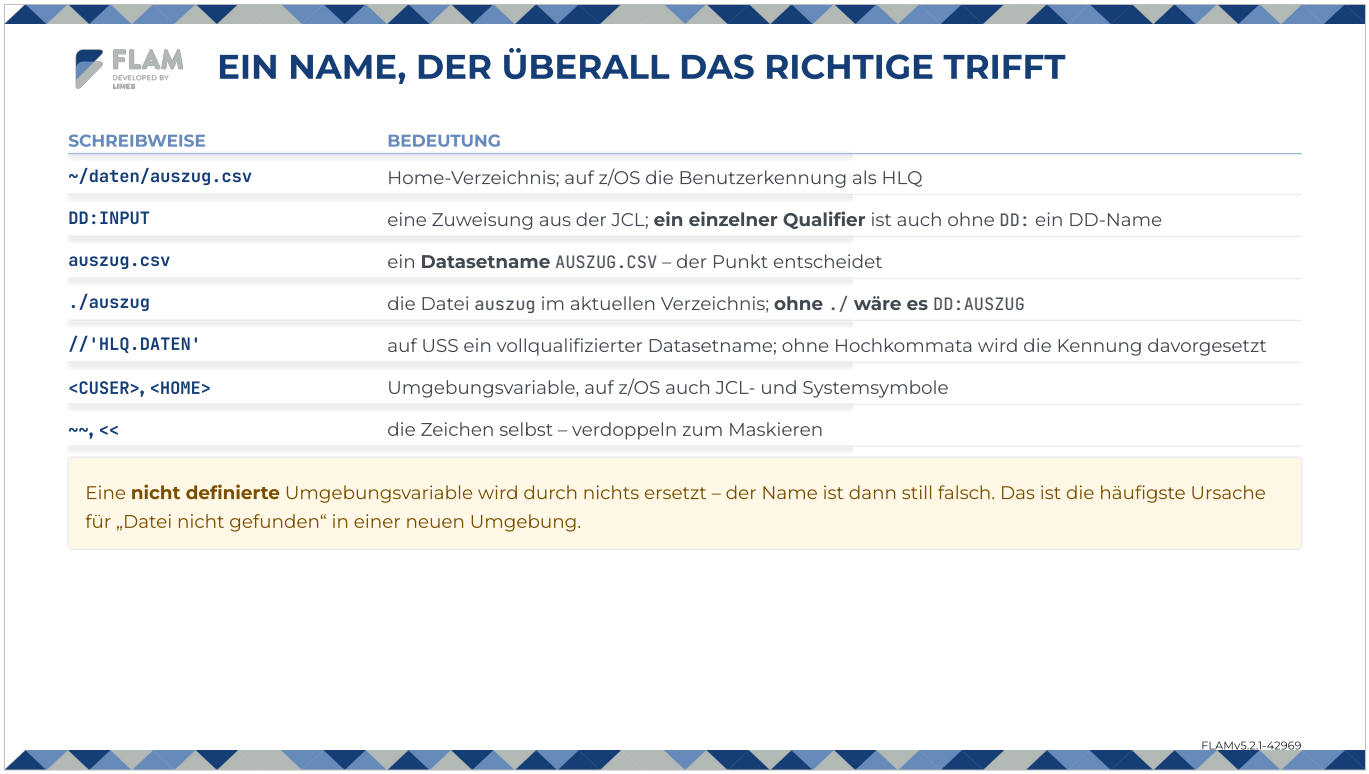
Die dritte Einheit zur Sprache, und die mit dem unmittelbarsten Nutzen: sie ersetzt Schleifen. Der Bogen: erst die Auswahl – Wildcards, der Dateiname selbst, Member in Archiven, der Directory-Walk. Dann die andere Richtung: wie der Ausgabename aus dem Eingabenamen gebildet wird.
Der Satz für den Anfang: **wer diese Einheit kennt, schreibt keine Schleife mehr.** Ein Aufruf verarbeitet einen ganzen Ordner und benennt jede Ausgabe nach einer Regel – das ist in der JCL eine Zeile statt eines REXX-Programms.



Das vierte Ziel ist das, das die Einheit rechtfertigt: eine Regel statt einer Liste. Wer sie einmal geschrieben hat, braucht sie in jedem zweiten Job.



Die Vokabelliste der Auswahl. Drei Zeilen zeigen genügt, der Rest ist zum Nachschlagen – aber die letzte Zeile muss sitzen.
Das `*` *und das `?` kennt jeder. Interessant sind die anderen:*
Das `+` verlangt **mindestens ein** Zeichen – damit trifft man alles außer dem leeren Rest, was `*` *nicht kann.*
Das `%` ist genau ein Zeichen und kommt aus der FLAM4- und ISPF-Welt. Wer von dort kommt, schreibt weiter `%` und es funktioniert – nur nicht in URLs.
Das `#` ist genau eine Ziffer, gilt aber **nicht auf MVS**, weil das Doppelkreuz dort Teil eines Qualifiers sein kann. Dafür gibt es das `=`:
genau eine Ziffer, auf allen Plattformen. Wer portabel schreiben will, nimmt `=`.
Und das Dach `^` ist die Maskierung. `dat^.txt` *meint die Datei, die wirklich* `dat.txt` heißt. Das braucht man selten und ist dann die Rettung.
Der Satz, der eine Supportrunde erspart: **ein Muster ist kein regulärer Ausdruck.** `.` *trifft hier nichts. Für Inhalte gibt es reguläre Ausdrücke, für Namen Wildcards – zwei Sprachen, ein Schalter dazwischen.*



Die Folie, die eine Prozedur portabel macht.
Jeder Dateiname geht durch dieselbe Abbildung, bevor irgendetwas geöffnet wird. Vier Regeln daraus braucht man wirklich:
Die Tilde steht für das Home-Verzeichnis, auf z/OS für die Benutzerkennung als High-Level-Qualifier. `~/daten/auszug.csv` auf Unix und `~.DATEN.AUSZUG` auf dem Host sind dieselbe Idee.
DD-Namen dürfen das Präfix `DD:` tragen, brauchen es aber seit 5.1.22 nicht mehr: **steht dort nur ein Qualifier** – kein Punkt, kein Schrägstrich – und gibt es eine Allokation dieses Namens, dann ist es ein DD-Name. Das Präfix macht es nur eindeutig, und in einem Aufruf, der nicht sichtbar aus einer JCL kommt, ist Eindeutigkeit das Argument dafür.
Auf MVS entscheidet die Form des Namens, und zwar in dieser Reihenfolge: **ein Schrägstrich** macht ihn zum Pfadnamen, **ein Punkt** zum Datasetnamen, und **ein einzelner Qualifier** – weder Punkt noch Schrägstrich – zum DD-Namen. Drei Beispiele nebeneinander, und der Unterschied ist jedes Mal nur ein Zeichen: `auszug.csv` ist das Dataset `AUSZUG.CSV`, `auszug` ist `DD:AUSZUG` und wird allokiert, `./auszug` ist die Datei `auszug` im aktuellen Verzeichnis. Wer eine USS-Datei ohne Endung meint, **muss** also `./auszug` schreiben; ohne das `./` versucht FLAM, `DD:AUSZUG` zu allokieren.
Auf USS beginnt ein Datasetname mit `//`, und mit Hochkommata dahinter ist er vollqualifiziert; ohne Hochkommata wird die Benutzerkennung davorgesetzt.
Umgebungsvariablen in spitzen Klammern werden ersetzt – auf z/OS auch exportierte JCL-Symbole und Systemsymbole. Damit wird eine Prozedur wiederverwendbar, ohne die JCL anzufassen.
Zwei Warnungen: ist die Variable nicht definiert, wird sie durch **nichts** ersetzt – der Name ist dann still falsch. Und wer ein `<` oder ein `~` wirklich im Namen braucht, schreibt es **doppelt**.

MEMBER IN EINEM CONTAINER AUSWÄHLEN

DER SCHRÄGSTRICH ÖFFNET DEN CONTAINER

```
# all XML members of a ZIP, written into a FLAM archive
flcl conv "read.xml(file=test.zip/?*.xml)
           write.flam(file=test.adc
                      member=<CUSER>.XMLPDS([base])
                      ccsid=1141)"

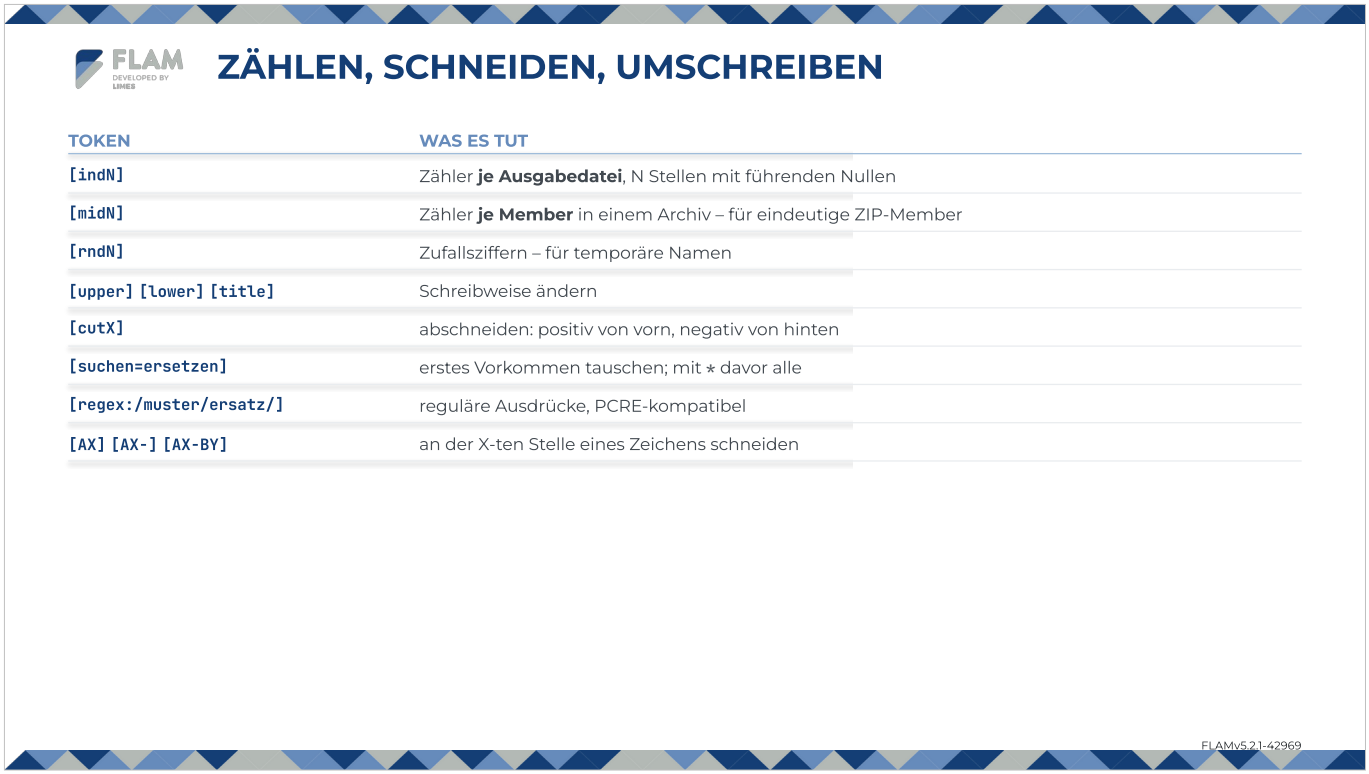

# a PDS member on z/OS, with and without a pattern
FILE='~.TEST.XMIT(MYTEXT)'      FILE='~.QUELLE(A*)'
```

FLAMv5.21-42969

Die Folie, die zeigt, dass ein Archiv sich wie ein Ordner anfühlt.
Ein Container – ein ZIP, ein PDS/E, ein FLAM-Archiv – wird mit einem Schrägstrich geöffnet, und dahinter steht das Muster für die Member.
`test.zip/?*.xml` *sind alle XML-Dateien im ZIP; das* `?` verlangt mindestens ein Zeichen vor der Endung.
Auf z/OS steht der Membername in Klammern, wie man es dort erwartet: `~.TEST.XMIT(MYTEXT)`, und mit Wildcards `~.QUELLE(A*)`.
Der Punkt für den Raum: das gilt **beim Lesen wie beim Schreiben**. Beim Schreiben bildet dieselbe Regel den Membernamen, und dafür gibt es die Namens-Token der nächsten Folien – `member='[base].txt'` schreibt jedes Element unter seinem Basisnamen in das Archiv.
Und es gilt für alle drei Containerarten mit derselben Schreibweise. Wer ein ZIP auspacken kann, kann auch ein PDS lesen.



Jetzt die andere Richtung, und das ist der Teil, der Schleifen ersetzt.
Ein Ausgabename ist ein **Muster** aus Text und Token. Text steht so da, wie er dasteht. Ein Token in eckigen Klammern beschreibt eine Verarbeitungsregel auf dem Eingabenamen.
Die sechs Token dieser Folie sind die, die man täglich braucht: `[path]` ist der Pfad ohne den letzten Schrägstrich, `[name]` der Dateiname, `[base]` der Dateiname ohne Endung, `[ext]` die Endung, `[member]` der Teil in Klammern – also der PDS-Membername – und `[copy]` der ganze Name, was man braucht, wenn man nur etwas anhängen will.
Das Beispiel unten ist gemessen: drei Dateien `f1.bin` bis `f3.bin`, gelesen über einen Directory-Walk, ergeben `f1.txt`, `f2.txt` und `f3.txt`.
Der Satz dazu: die Regel steht **einmal** im Aufruf und gilt für jede Datei, die der Walk findet. Man muss also nicht wissen, wie viele es sind und wie sie heißen.



Die zweite Hälfte der Token-Liste. Hier liegen die Griffe, mit denen man Eindeutigkeit erzwingt.
`[indN]` ist der **Zähler je Ausgabedatei**: N ist die Mindestbreite, mit führenden Nullen, und 0 heißt kürzest mögliche Darstellung. Gemessen:
`[base|upper]_[ind2]` ergibt `F1_03`, `F2_02`, `F3_01`.
`[midN]` ist der Zähler **je Member innerhalb eines Archivs**. Den braucht man, wenn dieselbe Satzbeschreibung mehrfach in ein ZIP geschrieben wird und das ZIP nach ISO 21320-1 eindeutige Membernamen haben muss.
`[rndN]` sind Zufallsziffern – für temporäre Namen.
`[upper]`, `[lower]`, `[title]` ändern die Schreibweise. Das braucht man beim Weg vom Host nach Unix fast immer.
`[cutX]` schneidet ab: positiv von vorn, negativ von hinten. Damit hält man einen Membernamen unter acht Zeichen.
Und dann die Ersetzungen: `[suchen=ersetzen]` tauscht das erste Vorkommen, mit einem Stern davor alle. `[regex:/muster/ersatz/]` ist die volle Form mit regulären Ausdrücken – gemessen ergibt `[name|regex:/^f([0-9])\.bin$/auszug_$1.csv/]` aus `f1.bin` die Datei `auszug_1.csv`.
Zuletzt der Baukasten für Namen mit Trennzeichen: `[AX]`, `[AX-]` und `[AX-BY]` schneiden an der X-ten Stelle eines Zeichens. Damit zerlegt man einen Datasetnamen an den Punkten, ohne einen regulären Ausdruck zu schreiben.

REGELN VERKETTEN MIT DEM STRICH

MEHRERE REGELN IN EINEM TOKEN, GETRENNT DURCH |

```
# measured: F1_03.txt, F2_02.txt, F3_01.txt
file='/ziel/[base|upper]_[ind2].txt'


# a data set name becomes a path name
file='/ziel/[member|lower].csv'


# measured: f1.bin -> auszug_1.csv
file='/ziel/[name|regex:/^f([0-9])\.bin$/auszug_$1.csv/]'
```

Die Schlüsselwörter sind nicht groß/klein-empfindlich. Eine eckige Klammer im Namen wird mit einem Dach maskiert.

FLAMv5.21-42969

Die Folie, auf der aus den Token eine kleine Sprache wird.
In einem Token dürfen mehrere Regeln stehen, getrennt durch einen senkrechten Strich. Jede Regel bekommt das Ergebnis der vorigen – das ist eine Kette, und sie liest sich von links nach rechts.
Die beiden Beispiele sind gemessen. Das erste macht aus einem Basisnamen Großbuchstaben und hängt einen zweistelligen Zähler an. Das zweite baut aus einem Datasetnamen einen Pfadnamen: der Membername in Kleinbuchstaben, eine Endung dahinter.
Der Rat: eine Kette mit zwei oder drei Regeln ist lesbar, eine mit sechs nicht. Wenn es lang wird, gehört der Aufruf in eine Parameterdatei, wo man ihn kommentieren kann – die Sprach-Einheit zeigt das.
Und eine Kleinigkeit, die Zeit spart: die Schlüsselwörter in den Token sind **nicht** groß/klein-empfindlich, `[BASE]` geht genauso wie `[base]`. Wer eine eckige Klammer im Namen braucht, stellt ein Dach davor.

EIN AUFRUF FÜR EINEN GANZEN ORDNER

DIRECTORY-WALK MIT AUSNAHMEN

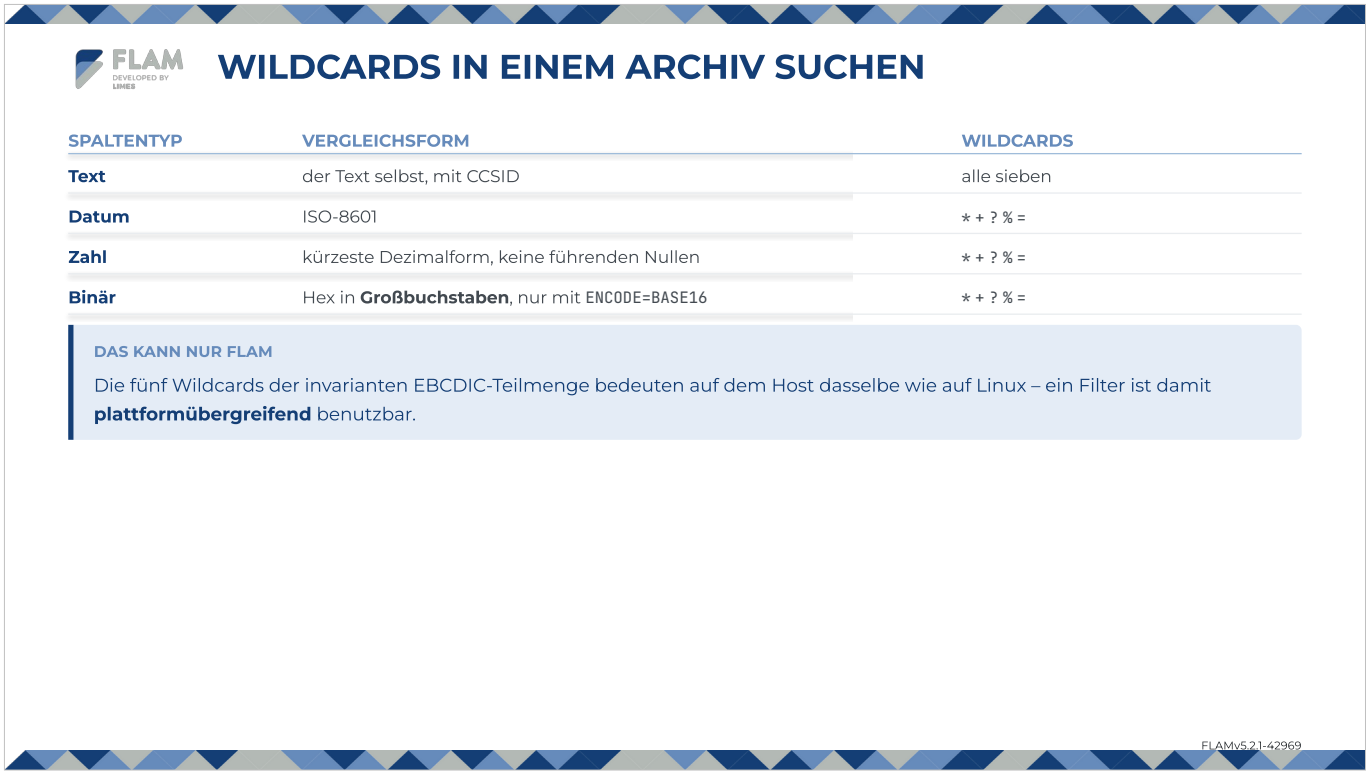
```
flcl CONV "read.record(file='/daten/ein/*.bin'
                       exclude['_test.bin' '.tmp']
                       recformat=FIX reclength=80 ccsid=1141)
           write.text(file='/daten/aus/[base|lower].csv'
                      ccsid='UTF-8' method=UNIX)"

# CC 0 = everything processed
# CC 4 = SOME FILES NOT PROCESSED -- the log names them
```

Completion Code 4 heißt: einzelne Dateien wurden nicht verarbeitet. Wer die 4 als Erfolg behandelt, verliert genau die Dateien, die Arbeit machen.

FLAMv5.21-42969

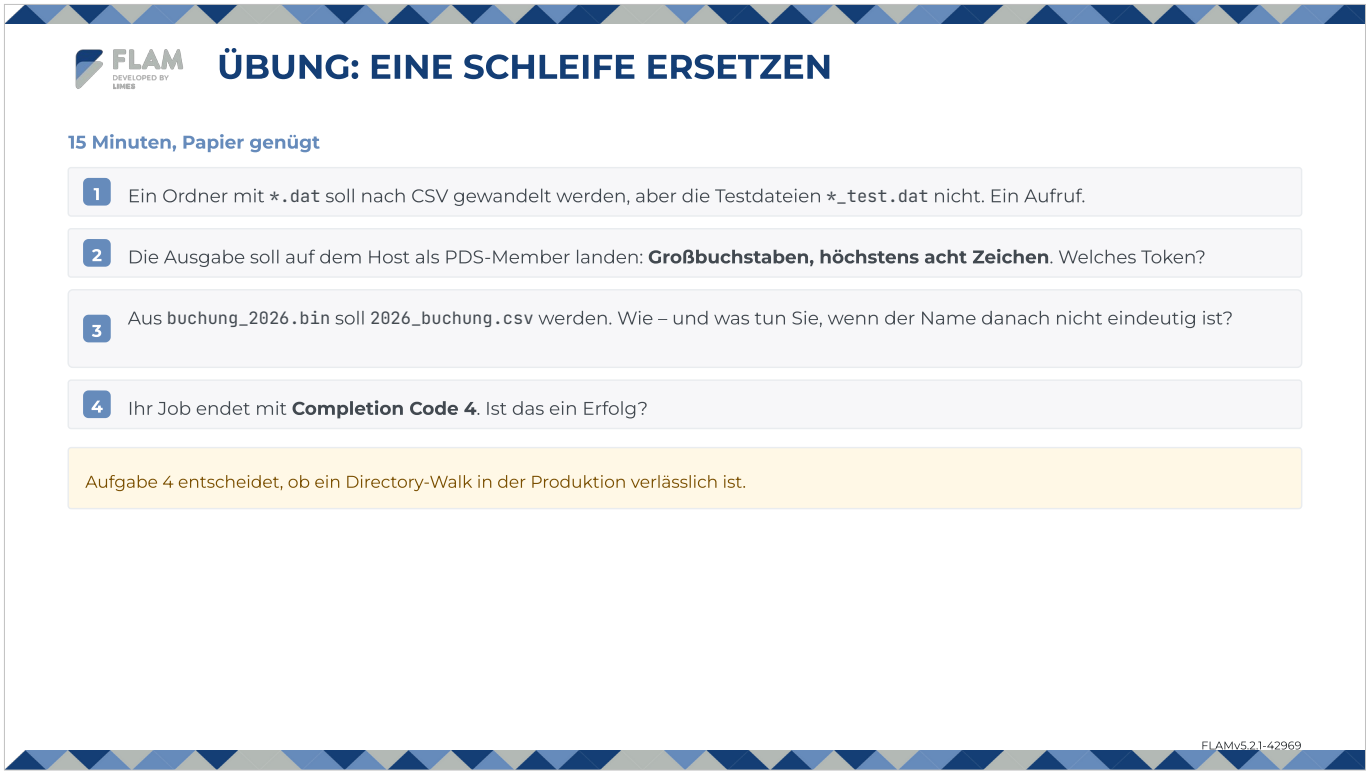
Die Folie, auf der das Ganze zusammenkommt – und die eine Falle enthält, die in der Jobsteuerung sitzt.
Ein Muster im Eingabenamen löst einen **Directory-Walk** aus: FLAM verarbeitet jede Datei, die passt, und bildet den Ausgabenamen aus der Regel. Das ersetzt die Schleife.
`EXCLUDE` nimmt einzelne heraus, und es ist ein Array – man darf mehrere Muster angeben. Gemessen: mit `exclude['*2.bin*']` *werden zwei von drei Dateien verarbeitet, und der Lauf endet mit* **0**. Eine ausgeschlossene Datei ist also kein Befund, sie war nie dabei.
Und jetzt die Falle: **Completion Code 4 heißt, dass einzelne Dateien nicht verarbeitet wurden** – weil eine gesperrt war, weil eine Konvertierung fehlschlug, weil ein Ziel existierte. Der Lauf ist damit nicht gescheitert, aber er ist auch nicht vollständig. Wer die 4 in der Jobsteuerung als Erfolg behandelt, verliert genau die Dateien, die Arbeit machen. Die Log-Zeile nennt sie beim Namen.
Der Rat für die Praxis: die 4 auswerten und die genannten Dateien nacharbeiten. Die Betriebs-Einheit zeigt, wie man das in der JCL aufsetzt.



Eine Folie für die, die schon mit Archiven arbeiten – sie verbindet diese Einheit mit der Suche. Wildcards gelten nicht nur für Namen, sondern auch für **Werte** in einem Filter, mit dem man in einem Archiv sucht. Und dort gibt es eine Einschränkung, die man kennen muss.
Eine Spalte ist im Archiv nicht als Text gespeichert, sondern in einer neutralen Form. Für den Vergleich wird sie in eine feste Textform gebracht: ein Datum nach ISO-8601, eine Zahl als kürzeste Dezimalform ohne führende Nullen, eine Binärspalte als Hex in Großbuchstaben. Das Muster muss also zu dieser Form passen, nicht zu der, in der die Daten einmal angeliefert wurden.
Und die Einschränkung: nur eine **Textspalte** trägt eine CCSID. Deshalb sind nur dort das Doppelkreuz und das Dach sicher erkennbar – beide sind in EBCDIC variante Zeichen und liegen je Codepage anders. Für alle anderen Typen bleiben `*`, `+`, `?`, `%` *und* `=`, *und die sind invariant.*
Der Satz für den Raum: **wer plattformübergreifend sucht, nimmt die invariante Teilmenge.** Dann bedeutet das Muster auf dem Host dasselbe wie auf Linux.



Alle vier aus Supportfällen.



Fünfzehn Minuten, Papier genügt.
Die Lösungen:
Aufgabe 1. `file='/ein/*.dat' *exclude*['_test.dat']`, und der Ausgabename `'/aus/[base].csv'`. Eine Zeile statt einer Schleife.
Aufgabe 2. `[base|upper|cut8]` – Großbuchstaben, dann auf acht Zeichen gekürzt. Die Reihenfolge in der Kette ist wichtig: erst umschreiben, dann kürzen.
Aufgabe 3. `[name|regex:/.../.../]`, oder einfacher mit `[suchen=ersetzen]`, wenn es nur ein festes Stück ist. Und der Hinweis:
wenn der Name danach nicht mehr eindeutig ist, hilft `[ind0]`.
Aufgabe 4. Nein – die 4 heißt, dass einzelne Dateien nicht verarbeitet wurden. Der Job muss sie aus dem Log nehmen und nacharbeiten. Das ist die Aufgabe, die in der Praxis am teuersten ist, wenn man sie falsch löst.

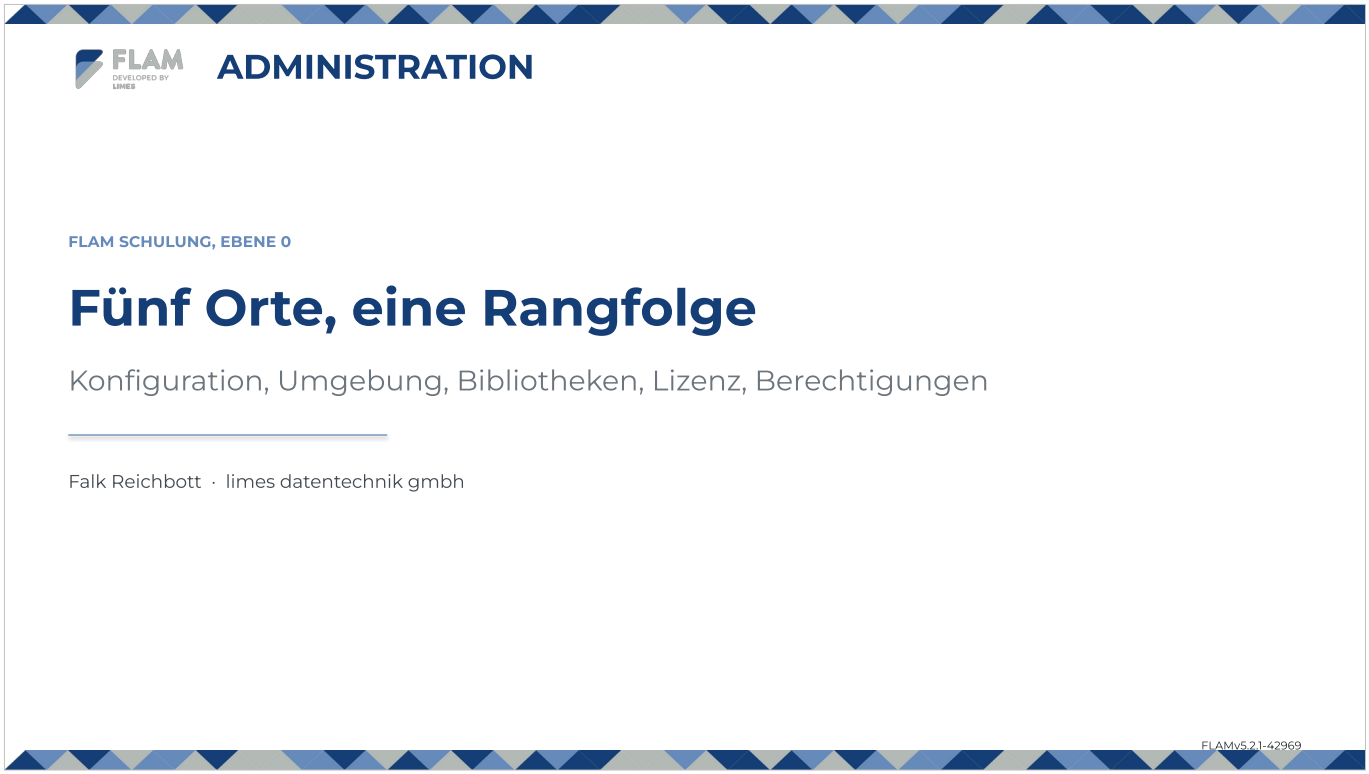
WAS BLEIBT

- **sieben Wildcards**, und = ist die portable Ziffer – # gilt nicht auf MVS
- ein Muster ist **kein** regulärer Ausdruck
- `~`, `DD:`, `//''` und `<variable>` machen einen Namen plattformgerecht – eine undefinierte Variable wird **nichts**
- der Schrägstrich öffnet einen Container: ZIP, PDS, FLAM-Archiv
- der Ausgabename kommt aus einer **Regel**: `[base]`, `[indN]`, `[regex:/…/…/]`, verkettet mit `|`
- **Completion Code 4 heißt: nicht alle Dateien verarbeitet**

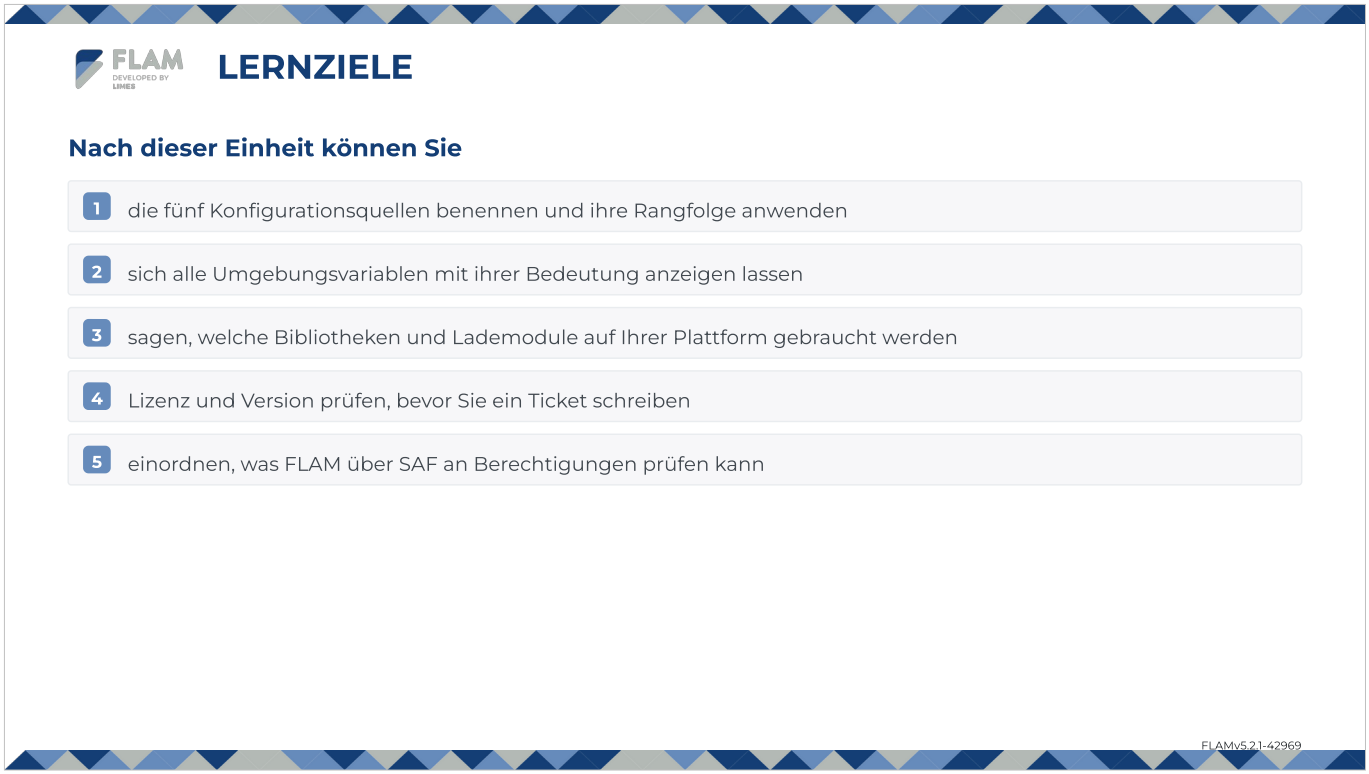
Zum Nachlesen im Handbuch: *1.4 Filename Mapping*, *3.2 Wildcards*, *3.3 Filename Handling*, *3.4 Directory Support* und *3.5 Input to Output Name Mapping*.

FLAMv5.2.1-42969

Lernziele durchgehen. Der Verweis nach vorn: die Netzwerk-Einheit zeigt, dass derselbe Name auch eine URL sein darf – die Auswahl gilt dann über SSH genauso.



Vierte Einheit der Ebene 0, und die erste, die sich an den Betrieb richtet.
Der rote Faden: es gibt genau fünf Orte, an denen eine Einstellung stehen kann, und sie haben eine feste Rangfolge. Wer die kennt, sucht nie wieder lange.



Ziel eins ist das operative Ziel: die Rangfolge im Kopf zu haben spart im Betrieb die meisten Rückfragen.

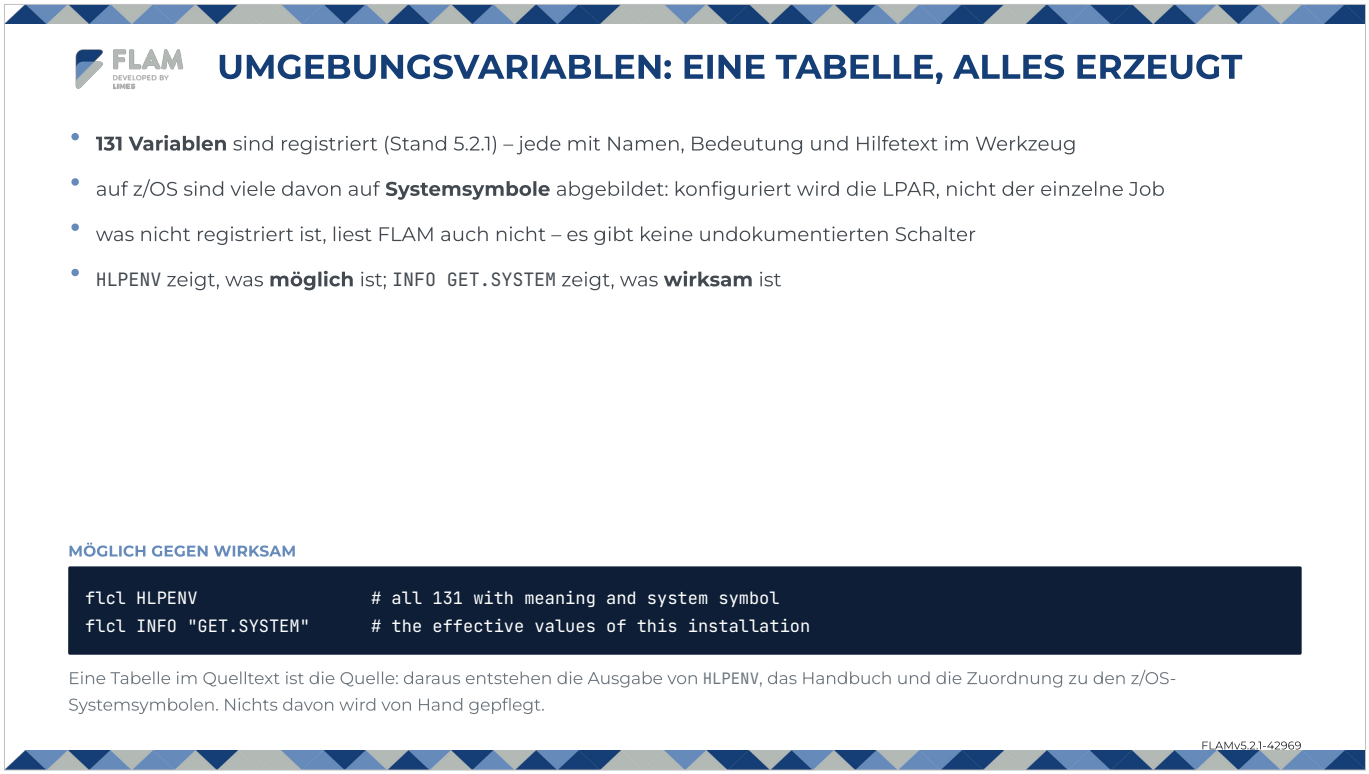
WOHER EINE EINSTELLUNG KOMMT

RANG	QUELLE	WOFÜR GEDACHT
1	Property-Datei	Hausstandards, dauerhaft: CCSID, Blockgroesse, Reportpfade
2	Konfiguration / Owner	mehrere Ausprägungen nebeneinander: Test, Produktion, Mandant
3	Umgebungsvariablen	plattform- und installationsabhängige Pfade
4	Parameterdatei	langer String, ausgelagert – auf z/OS über FLAMPAR
5	Kommandozeile	der Einzelfall, gewinnt gegen alles davor

„Der Parameter wirkt nicht“ heißt fast immer: es wirkt ein anderer. Erst die Rangfolge prüfen, dann den Parameter.

FLAMv5.21-42969

Die Rangfolge auf das Whiteboard. Von oben nach unten gewinnt der jeweils spätere Eintrag.
Das ist die Ursache für die häufigste Betriebsmeldung "der Parameter wirkt nicht": es wirkt einer, nur nicht der, den man gerade angesehen hat.
Praxis: bei Unklarheit `flcl INFO GET.SYSTEM` und die aktive Konfiguration ausgeben lassen, statt zu raten.



Die Zahl nennen, sie wirkt: 131 registrierte Variablen (Stand 5.2.1, gezählt über `flcl HLPENV`).
Niemand muss sie kennen – aber jede ist im Werkzeug selbst beschrieben.
`flcl HLPENV` ist der Befehl, den der Betrieb braucht. Er listet Name, Bedeutung und, auf z/OS, das zugeordnete Systemsymbol in Klammern.
Der Punkt, der das glaubwürdig macht: es gibt **eine** Tabelle im Quelltext, und alles andere ist daraus erzeugt – die Ausgabe von `HLPENV`, das Handbuch und, für einen Teil der Variablen, die Zuordnung zu den z/OS-Systemsymbolen.
Seit 5.1.26 pflegt das Handbuch die Liste nicht mehr von Hand. Eine Umgebungsvariable, die nicht in dieser Tabelle steht, gibt es nicht – weder im Werkzeug noch in der Doku.
Die Zahl deshalb als Momentaufnahme nennen und nicht als Eigenschaft: mit der Tabelle wächst sie.
Auf z/OS ist die Besonderheit, dass viele Variablen auf Systemsymbole abgebildet sind – damit konfiguriert man die LPAR und nicht jeden Job.
Der zweite Befehl daneben ist der für die Diagnose: `INFO GET.SYSTEM` zeigt nicht die möglichen, sondern die **wirksamen** Werte. Der Unterschied ist im Fehlerfall alles.

AUF Z/OS: BIBLIOTHEKEN UND LADEMODULE

WAS	WO	WOFÜR
Lademodule	STEPLIB oder LINKLIST	flcl, FLUC-UP, FLAM-UP, Satzschnittstelle
Parameter	FLAMPAR (DD)	der Parameterstring, ausgelagert aus der JCL
Started Task	eigene Prozedur	nur für das Subsystem – ohne sie kein SUBSYS=
USS-Anteil	LIBPATH	für die Nutzung aus USS und aus Java über JNI
Berechtigungen	RACF-Klasse $FLAM	optional, ab Version 5.1.21 aktivierbar

Das Subsystem braucht einen laufenden Dienst, nicht nur eine Bibliothek. Hier müssen Betrieb und Anwendungsentwicklung zusammenarbeiten.

FLAMv5.2.1-42969

Die drei Zeilen, die im Betrieb gebraucht werden. Was danach kommt, hängt davon ab, welcher der drei Wege genutzt wird.
Wichtig: für das Subsystem läuft eine Started Task. Wer das Subsystem nutzen will, braucht also nicht nur eine Bibliothek, sondern einen laufenden Dienst – das ist der Punkt, an dem Betrieb und Anwendungsentwicklung zusammenarbeiten müssen.

AUF LINUX, UNIX UND WINDOWS

- Installationsanleitungen liegen für **sieben Plattformen** bei: Linux, Windows, AIX, Solaris, BS2000/OSD, z/OS und z/OS USS
- Linux: **vier Paketformate** je Architektur – `rpm` (Red Hat, CentOS, Fedora, SUSE), `apt` (Debian, Ubuntu), `pac` (Arch) und `tar` als generischer Weg
- Windows: **MSI**; z/OS: **SMP/E** – jedes echte Paketformat ist signiert
- **vier Pakete**: die Basis (`flam`), die Oberflächen (`flamgui` – FLCC und FLGT), die Doku (`flamdoc`) und die Lizenz (`flamlic`)
- dazu `flam4` – das **alte** FLAM4-Utility, wenn ein Haus 100 % abwärtskompatibel bleiben will
- Standardablage ist `/opt/limes/flam/bin` – dieser Pfad muss in die `PATH`-Variable

Reihenfolge: **FLAM5-Basis**, dann optional Oberflächen und Doku – und nur wenn es **100 % abwärtskompatibel** sein muss, darüber das alte FLAM4. Die installierte Doku gehört zur installierten Version; das ist der Grund, sie mitzuinstallieren und nicht online zu lesen.

FLAMv5.21-42969

Kurz halten. Der einzige Punkt, der regelmäßig Rückfragen erzeugt, ist der PATH – und dass die Dokumentation ein eigenes Paket ist.
Zu den Paketformaten die vollständige Auskunft, weil hier regelmäßig zu wenig gesagt wird: für Linux gibt es je Architektur **vier** Wege – `rpm` für Red Hat, CentOS, Fedora und SUSE, `apt` für Debian und Ubuntu, `pac` für Arch, und `tar` als generischen Weg für alles andere. Wer nach "gibt es ein RPM" fragt, bekommt also ein Ja, und zwar nicht als Ausnahme.
Nachsehen kann das jeder selbst: der Downloadbereich ist nach Plattform, Architektur und Wortbreite gegliedert, und darunter liegen die Paketformate.
Auf Windows ist es MSI, auf z/OS SMP/E.
Die Installationsanleitung liegt für sieben Plattformen bei.
Die Aufteilung in Pakete hat je einen Grund, und beide kommen als Frage.
Die Oberflächen sind ein eigenes Paket, weil nicht jeder Server eine grafische Oberfläche hat – auf einem Batch-Server installiert man FLCC und FLGT gar nicht. **FLAM4 ist ein eigenes Paket**, das man über FLAM5 legt:
FLAM5 bringt ein neues FLAM4-Utility mit, das in den Grenzfällen nicht hundertprozentig so arbeitet wie das alte – beim Input-Output-Name-Mapping und beim Memberzugriff. Wer daran nichts anpassen will, ersetzt das neue Utility durch das aktuelle alte aus dem `flam4`-Paket.
Falls jemand die HP-UX-Anleitung im Paket findet: die stammt aus früheren Releases. HP-UX ist nicht mehr im Support.

LIZENZ UND VERSION PRÜFEN

- die Auslieferung enthält die **Demo-Lizenz** – alles läuft, aber nur bis **1 MiB je Datei**
- die Demo-Lizenz **untersagt die kommerzielle und produktive Nutzung**, sie ist ohne Gewähr und ohne Support und kann Funktionen in einem Vorserienstand enthalten
- für den produktiven Einsatz wird eine **kommerzielle Lizenz** erworben: ein **eigenes Paket** über die bestehende Installation, keine Neuinstallation
- `flcl license` und `flcl INFO GET.SYSTEM` zeigen Version, Build, Plattform, aktive Lizenz und Konfiguration
- eine produktive Lizenz-ID trägt **Kundennamen und Vertragsnummer** – steht dort DEMO, greift die Demo-Lizenz
- die Lizenz ist ein Singleton im Prozess – ein Wechsel wirkt erst im nächsten Lauf

Vor jedem Ticket: `flcl INFO GET.SYSTEM` ausführen und die Ausgabe mitschicken. Das ersetzt die erste Rückfragerunde.

FLAMv5.21-42969

Diese Folie ist die Vorbereitung für jedes Ticket. Wer `flcl INFO GET.SYSTEM` laufen lässt, hat die Hälfte der Rückfragen schon beantwortet.
Das Lizenzmodell zuerst, weil es die häufigste kaufmännische Frage im Raum ist. **Was Sie installieren, läuft sofort – mit der Demo-Lizenz.** Sie ist funktional vollständig: jede Funktion, jede Plattform, keine Zeitschranke.
Begrenzt ist nur die Menge, nämlich 1 MiB je Datei.
Und der Satz, der dazugehört und den man nicht verschweigen darf: **die Demo-Lizenz untersagt die kommerzielle und die produktive Nutzung.** Sie ist ohne Gewähr und ohne Support, und sie kann Funktionen enthalten, die noch in einem Vorserienstand sind. Sie ist zum Ausprobieren und zum Testen da, nicht für den Betrieb. Für den produktiven Einsatz – und für die Kompatibilität mit FLAM4-Beständen – wird eine **kommerzielle Lizenz** erworben; die kommt als **eigenes Paket** dazu und wird über die vorhandene Installation gelegt. Die Software selbst wird dabei nicht neu installiert, und das gilt auch für die Verlängerung.
Woran man eine produktive Lizenz erkennt: die Lizenz-ID trägt Kundennamen und Vertragsnummer. Steht dort DEMO, arbeitet man noch mit der Demo-Lizenz – auf dem Host der klassische Fall, dass die Lizenzbibliothek nicht in der STEPLIB steht.
Bei einem Lizenzproblem ist die Reihenfolge: greift überhaupt eine Lizenz, welche, und passt sie zur Plattform.

BERECHTIGUNGEN ÜBER SAF

- ab Version **5.1.21** können zahlreiche zusätzliche Prüfungen gegen die System Authorization Facility aktiviert werden
- sie laufen über eine eigene RACF-Klasse `$FLAM` mit Ressourcen je Funktionsbereich
- die Prüfungen sind **optional** – erst einschalten, wenn die Ressourcen definiert sind
- damit lässt sich steuern, wer verschlüsseln, entschlüsseln oder auf Schlüssel zugreifen darf

Vertiefung: Foliensatz Sicherheit – SAF und SKMS.

Nur einordnen, nicht vertiefen – die Details stehen im Deck sicherheit-saf-skms.
Der Punkt für den Betrieb: die Prüfungen sind optional und werden aktiv eingeschaltet. Wer sie einschaltet, muss die Ressourcen vorher definieren, sonst stehen Jobs.
Ausgangspunkt war ein Exit zum Virenscannen; daraus wurde ein allgemeiner Weg, FLAM-Funktionen an die Sicherheitsinfrastruktur zu binden.

STOLPERFALLEN

! **Eine Property-Datei wirkt auch dann, wenn niemand mehr von ihr weiß.**
Bei unerklärlichem Verhalten zuerst `flcl INFO GET.SYSTEM` und die aktive Konfiguration prüfen, nicht den Parameter.

! **Ohne Started Task kein Subsystem.**
Das `SUBSYS=` im DD-Statement scheitert, wenn der Dienst nicht läuft – und der Fehler klingt nicht danach.

! **Die Demo-Lizenz fällt spät auf.**
Alles funktioniert, bis eine Datei größer als 1 MiB ist. Vor einem Lasttest also die Lizenz prüfen.

! **Umgebungsvariablen aus einer Shell reisen nicht in einen Job.**
Auf z/OS gehören sie in die Konfiguration oder auf Systemsymbole, nicht in ein Profil.

FLAMv5.21-42969

Alle vier aus dem Betrieb. Die erste und die dritte kosten die meiste Zeit.



Fünfzehn Minuten am eigenen System. Ziel ist, dass jeder danach seine Installation beschreiben kann, ohne jemanden zu fragen.
Auflösung, damit niemand hängen bleibt:
1. `flcl INFO GET.SYSTEM` – die Ausgabe nennt Version und Build, das Lizenzmodul, den Owner, die geltende Konfigurationsdatei und die wirksamen Umgebungsvariablen.
2. `flcl HLPENV` listet alle 131. Die drei, die fast jeden betreffen, sind `LANG` (die CCSID – auf z/OS aus `&FLANGLC`, `&FLANGCC`, `&FLANGCS`), `FL_DEFAULT_CHAR_MODE` (das Fehlerverhalten der Zeichensatzwandlung) und `FL_ZEDC_SUPPORT` auf z/OS. Wichtig ist der Unterschied: `HLPENV` sagt, was möglich ist, `INFO GET.SYSTEM` sagt, was gesetzt ist.
3. Auf z/OS ist es die Bibliothek in der `STEPLIB`, meist `FLAM.LOAD`; unter Unix das Installationsverzeichnis samt `LD_LIBRARY_PATH`. Gepflegt wird sie in der Regel von der Systemtechnik, nicht von der Fachabteilung – genau das soll die Aufgabe klären.
4. Eine Started Task braucht es nur für Subsystem, LE-lose Schnittstellen und zIIP. Wer keines davon nutzt, braucht keine – das ist die häufigste falsche Annahme.

WAS BLEIBT

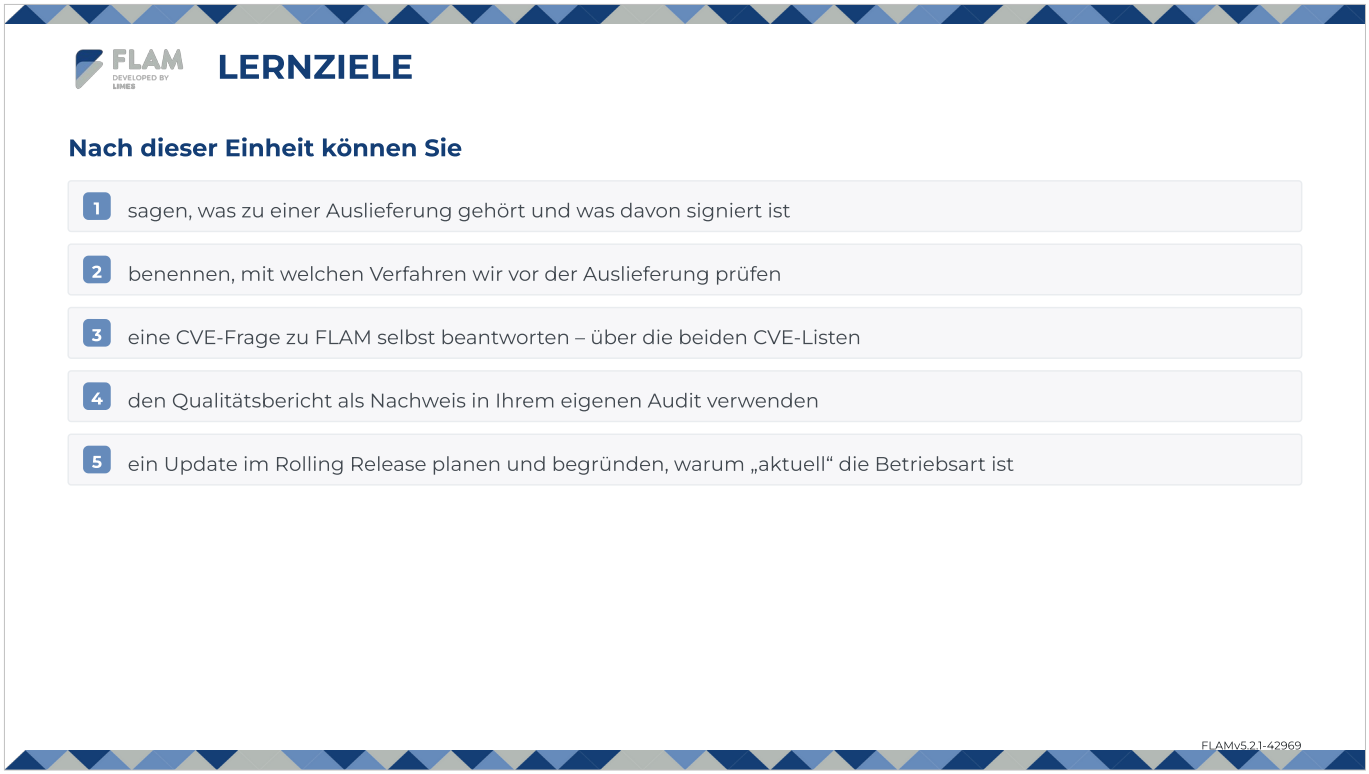
- fünf Konfigurationsquellen mit fester Rangfolge – der spätere gewinnt
- 131 Umgebungsvariablen, alle im Werkzeug beschrieben: `flcl HLPENV` – wirksam ist, was `INFO "GET.SYSTEM"` zeigt
- auf z/OS: Lademodule, FLAMPAR, Started Task für das Subsystem, LIBPATH für USS
- `flcl INFO GET.SYSTEM` vor jedem Ticket – Version, Lizenz, Konfiguration
- SAF-Prüfungen sind optional und müssen vorbereitet werden

FLAMv5.21-42969

Lernziele durchgehen. Danach folgt Betrieb und Wartung – also was wir liefern, wie wir es prüfen und wie Sie aktualisieren.



Fünfte Einheit der Ebene 0. Zielgruppe ist der Betrieb, nicht der Einkauf – also keine Werbung für unsere QA, sondern die Frage: was bekommen Sie, was ist mitgeprüft, und wie aktualisieren Sie. Der Satz, der hängen bleiben soll: es gibt zu jeder CVE eine Antwort, und zwar in einer von zwei Listen.



Ziel drei und vier sind die, nach denen in Audits gefragt wird. Ziel fünf ist das operative.

WAS ZU EINER AUSLIEFERUNG GEHÖRT

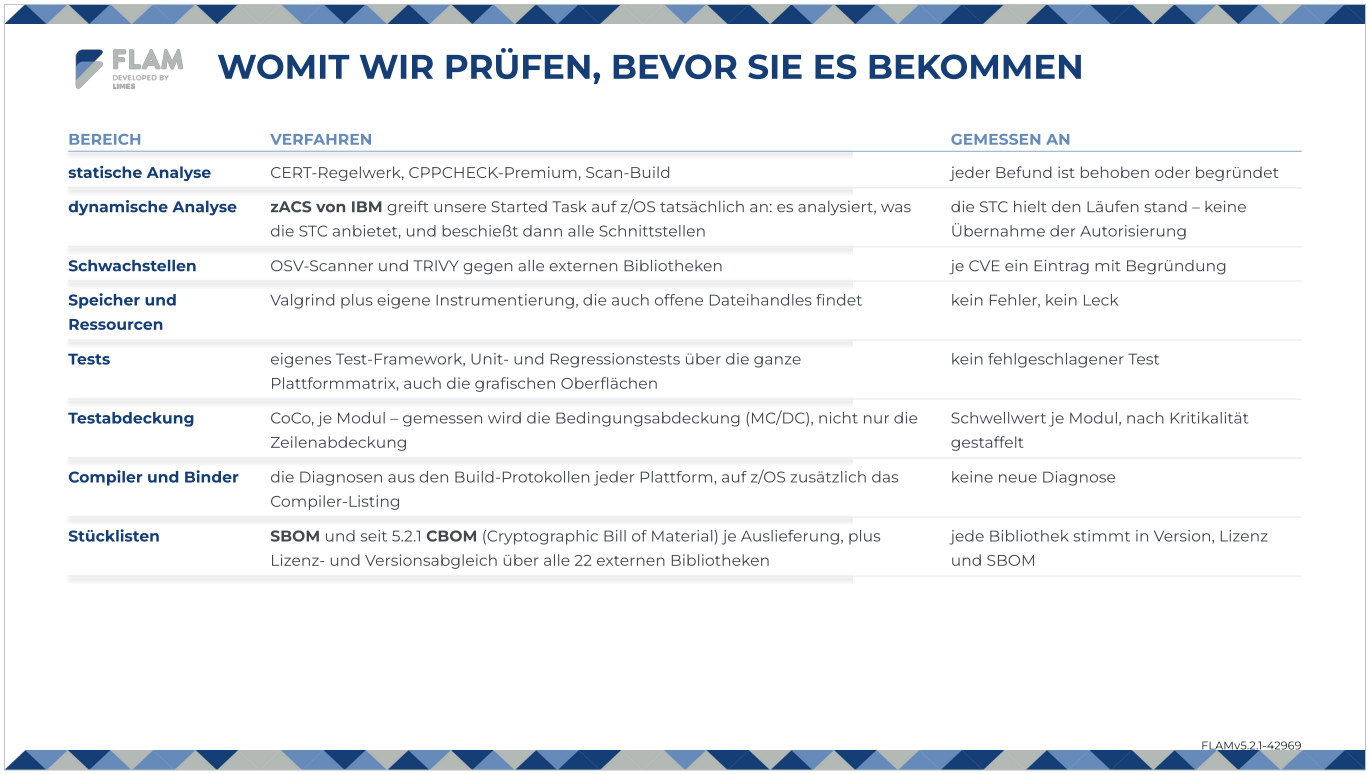
- das **Installationspaket** je Plattform – Ergebnis einer einzigen, automatisierten Kette; jedes echte Paketformat (MSI, Unix-Pakete, SMP/E) ist **signiert**
- die **Dokumentation im Paket**, erzeugt von der Software selbst und damit zum Stand passend – seit 5.2.1 liegen auch **diese Schulungsunterlagen** dort
- die **SBOM** und seit 5.2.1 die **CBOM** – Software und Cryptographic Bill of Materials
- der **Qualitätsbericht** mit den Befunden der Pruefwerkzeuge und ihrer Bewertung
- die **CVE-Listen** zu den externen Bibliotheken, mit Begründung je Eintrag

Alle QA-Maßnahmen führen als atomare Funktion zu einem signierten Installationspaket. Nichts wird von Hand zusammengestellt.

Auf z/OS ist SMP/E der Weg, und damit ist auch dort signiert. Ohne Signatur bleiben nur die alten Übertragungsformen ohne Paketstruktur – TAR-Balls und XMIT – die wir allein wegen eingespielter Kundenprozesse weiter anbieten.

FLAMv5.2.1-42969

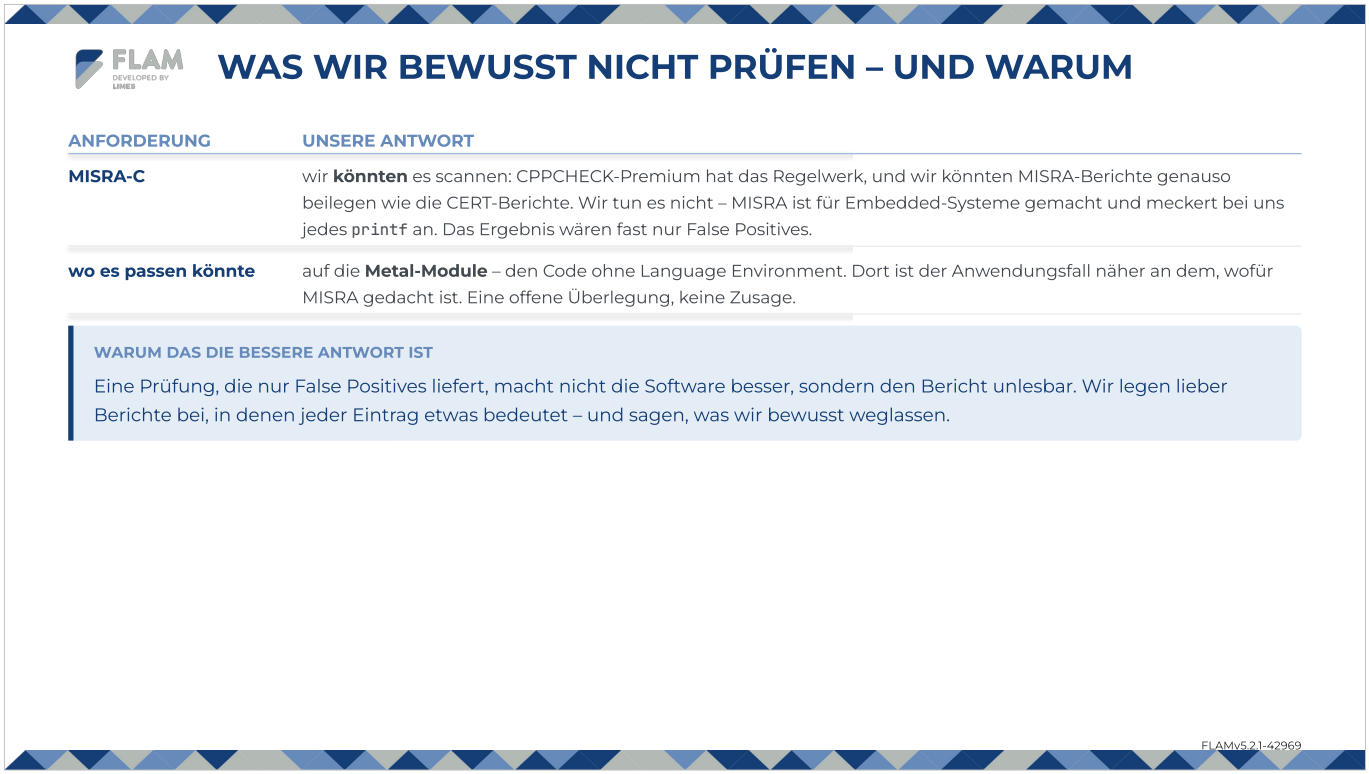
Der Kern: alle QA-Maßnahmen müssen als **eine** Funktion zu **einem** signierten Installationspaket führen. Das ist keine Marketingaussage, sondern die Anforderung, die uns Kunden aus regulierten Branchen gestellt haben – und der Grund, warum bei uns nichts von Hand zusammengestellt wird.
Wichtig für den Betrieb: die Dokumentation liegt im Paket. Damit passt sie zur installierten Version, auch wenn online schon eine neuere steht.
Und seit 5.2.1 gilt das auch für die Schulungsunterlagen: der Foliensatz, den Sie hier vor sich haben, wird mit ausgeliefert. Wer die Schulung ein Jahr später nachliest, hat den Stand, der zu seiner installierten Version gehört.



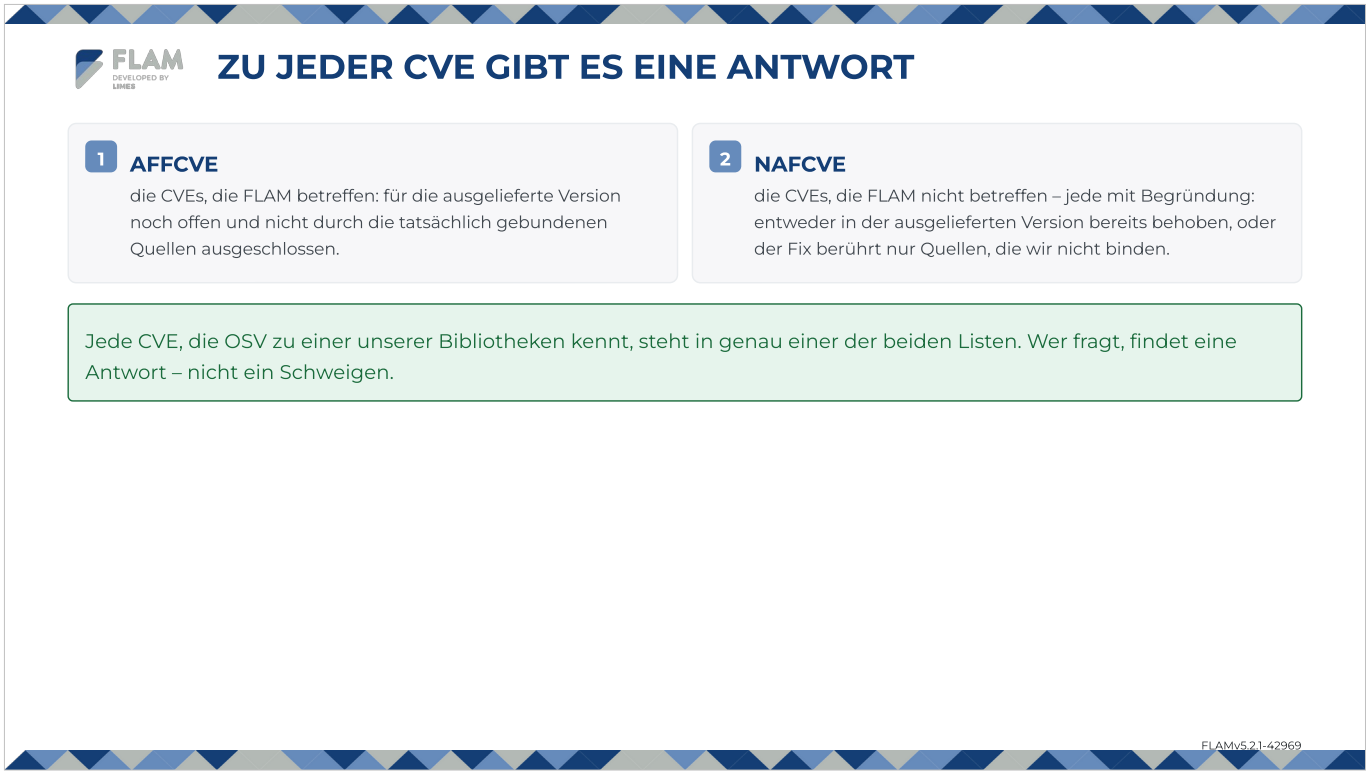
Nicht als Leistungsschau vortragen, sondern als Antwort auf die Frage, die im Audit kommt: welche Verfahren, und wie belegt.
Die Namen stehen alle im Qualitätsbericht, mit den jeweiligen Befunden und deren Bewertung.
Wer im Raum aus der Revision kommt, will diese Folie abfotografieren.
zACS ist der Punkt, der einen Mainframe-Kunden überzeugt. Dafür muss man kurz erklären, worum es überhaupt geht – im Raum sitzen oft Leute, die z/OS-Speicherschutz nie gebraucht haben.
Der Speicherschutz auf dem Host arbeitet mit Schlüsseln. Eine Anwendung läuft in **Key 8**, das System in **Key 0**. Eine Started Task schreibt man immer dann, wenn man für IBM ein Stück Betriebssystem fertigschreiben muss:
man braucht Autorisierung und muss nach Key 0 – das ist der Host-Gegenwert zu Root- oder Kernelrechten.
Und daraus folgt die Disziplin, um die es geht: eine STC darf nur dann nach Key 0 gehen oder sich autorisieren, wenn nichts mehr schiefgehen kann. Der Schritt muss atomar durchlaufen, danach ist man sofort wieder normale Anwendung. Wer autorisiert stehen bleibt und dann abstürzt, hinterlässt einen Zustand, in dem kritische Zugriffe möglich sind.
Genau darauf zielt zACS: er penetriert die Started Task und versucht, sie im autorisierten Zustand zum Brechen zu bringen, um die Autorisierung zu übernehmen. Das zwingt zu einer Programmierweise, die man sonst nirgends braucht – jeden Zeiger mit `VSMLOC` prüfen, jede Länge prüfen, bevor geschrieben wird.
Zur Einordnung: das ist **kein** statischer Analysator, sondern ein dynamischer Schwachstellen-Scanner. Von der Art her eher wie Valgrind als wie CPPCHECK – er beobachtet ein laufendes System, statt Quelltext zu lesen. Und die Aussage ist entsprechend: unsere STC hat den Läufen standgehalten. Das ist ein Prüfergebnis, keine Garantie.
Die Spalte "gemessen an" ist wichtiger als die Werkzeugnamen: sie sagt, dass jede Dimension einen Schwellwert oder eine Regel hat, gegen die gemessen wird. Ein Werkzeug ohne Kriterium ist im Audit wertlos.



Diese Folie ist der Unterschied zwischen "wir prüfen viel" und "wir können es belegen" – und sie ist die Folie, die einem Auditor tatsächlich genügt.
Der Kern in einem Satz: die Auslieferung wird von einem deterministischen Gatter freigegeben, nicht von einer Person, die Berichte durchsieht.
Die drei Eigenschaften einzeln betonen.
Erstens deterministisch: das Gatter selbst enthält keine KI und keine Bewertung. Es liest Berichte, normalisiert sie und vergleicht gegen Schwellwerte und hinterlegte Begründungen. Zweimal derselbe Eingang gibt zweimal dasselbe Urteil.
Zweitens fail-closed: eine fehlende Dimension ist ein Fehler, kein Schweigen.
Wenn ein Bericht nicht da ist, weil ein Lauf ausgefallen ist oder die Testinfrastruktur nicht erreichbar war, dann sagt das Gatter FEHLT und blockt.
Das ist der Punkt, an dem die meisten QA-Ketten in der Praxis versagen – eine nicht gelaufene Prüfung liest sich sonst wie eine bestandene.
Drittens die Frische: ein Bericht, der älter ist als der zulässige Zeitraum, gilt nicht. Sonst gibt ein alter grüner Bericht ein grünes Urteil für einen neuen Stand.
Und die Begründungen: jede akzeptierte Abweichung steht in einem handgepflegten Verzeichnis, mit einer maschinenlesbaren Verankerung und dem Begründungstext daneben. Werkzeuge dürfen Entwürfe vorschlagen, aber eingetragen wird von Hand – damit die Änderung im Diff sichtbar ist und gelesen wird.



Diese Folie kommt aus einer echten Kundenanforderung: MISRA-C stand auf einer Anforderungsliste, und die Frage kommt regelmäßig.
Die Antwort ist keine Ausrede, sondern eine Abgrenzung: MISRA-C ist für Embedded-Systeme gemacht. Auf unseren Code angewandt meckert es jedes `printf` an. Die Regeln passen für ein Werkzeug, das mit dem Betriebssystem und mit Dateien arbeitet, einfach nicht – was dabei herauskommt, sind praktisch ausschließlich False Positives.
Wichtig ist der technische Teil der Antwort, weil er zeigt, dass es keine Frage des Aufwands ist: wir könnten es scannen. CPPCHECK-Premium hat das MISRA-Regelwerk, und wir könnten MISRA-Berichte genauso ausliefern, wie wir die CERT-Berichte ausliefern. Wir tun es nicht, weil ein Bericht aus tausenden False Positives keine Aussage ist – er verdeckt die echten Befunde.
Wo es sinnvoll sein könnte, auch sagen: auf die Metal-Module, also den Code ohne Language Environment. Dort ist der Anwendungsfall näher an dem, wofür MISRA gedacht ist. Das ist eine offene Überlegung, keine Zusage.
Der Schlusssatz ist der, der im Audit trägt: eine Prüfung, die nur False Positives liefert, macht die Software nicht besser, sondern den Bericht unlesbar. Wir liefern lieber Berichte, in denen jeder Eintrag etwas bedeutet.



Das ist die Folie, die eine Auditfrage in zwei Minuten beendet – und die häufigste Frage aus regulierten Häusern.
Jede CVE, die OSV zu einer unserer externen Bibliotheken kennt, landet in genau einer der beiden Listen. Es gibt kein Drittes und kein Schweigen.
Das Beispiel nennen: wir binden libcrypto, aber nicht die TLS-Schicht. Eine CVE, deren Fix nur die TLS-Schicht berührt, betrifft uns nicht – und genau das steht mit dieser Begründung in der Liste.



Kurz halten, die Details stehen in den Anleitungen. Der Punkt für die Schulung ist die Bandbreite: sieben Plattformen, und die FLAMFILEs sind untereinander austauschbar.
Unter Linux ist es mehr als "ein Paket": es gibt **signierte Repositories** auf `repo.flam.de` – apt für Debian-basierte Systeme, yum, dnf und zypper für die RPM-Welt, pacman für Arch. Wer den GPG-Schlüssel importiert, aktualisiert FLAM danach wie jedes andere Systempaket. Das ist für einen Betrieb, der Paketquellen ohnehin verwaltet, der bequemste Weg – und der Signaturprüfung kommt er gratis mit.
Auf z/OS ist **SMP/E** der Weg, und das Paket ist signiert und vollständig.
Das XMIT darin enthält nur die Sample-Jobs für SMP/E selbst.
Die XMIT-Installation als Ganzes führen wir nur noch aus einem Grund weiter: **die eingespielten Prozesse unserer Kunden.** Manche bauen sich ihre SMP/E-Installation selbst aus zwei Teilen – das XMIT enthält die LOAD-Bibliothek, eine FLAMFILE den Rest. Wer so arbeitet, soll das weiterhin können.
Der letzte Punkt: die Doku ist ein eigenes Paket. Wer sie weglässt, liest später online die falsche Version.



Der Teil, der im Betrieb Widerstand erzeugt: es gibt keine Fix-Linie. Ein Fehler wird mit der nächsten Version ausgerollt.
Die Gegenleistung ist die Rückwärtskompatibilität – deshalb ist ein Update kein Risiko für die Bestandsdaten. Eine FLAMFILE aus den Achtzigern ist heute lesbar.
Praktischer Rat: eine Version im Testsystem vorlaufen lassen, aber nicht Versionen überspringen wollen. 13 Monate sind das Fenster.
Und jetzt die Einschränkung des Satzes "es gibt keine Fix-Linie", weil sie oft die eigentliche Frage im Raum beantwortet: **es gibt einen last-stable-Branch neben dem Master.** Wir können einen Fix dort einbauen und als neuen Build zur letzten Revision ausliefern – der Kunde bekommt also seinen Fix, ohne einen Versionssprung mitzumachen.
Technisch geht das in beide Richtungen: entweder auf dem last-stable fixen und in den Master mergen, oder im Master fixen und auf den last-stable cherry-picken. Was passt, entscheiden wir am Fall.
Für den Host gibt es zusätzlich den schnellen Weg: **ein Patch als XMIT-Load**, wenn nur ein Lademodul getauscht werden muss. Das ist die Antwort auf "wir können nicht innerhalb einer Woche eine ganze Installation durch die Freigabe bringen".
Der Grund, warum das überhaupt geht, ist Continuous Integration. Und daraus folgt eine Zusage, die man ruhig nennen darf, weil sie stimmt: **wünscht sich ein Kunde am Mittwoch einen neuen Parameter oder meldet einen Bug, steht das am Montag zur Verfügung.** Eine neue Komponente geht in etwa einem Monat. Das ist der eigentliche Vorteil der Rolling-Release-Linie – nicht dass es keine Fix-Releases gibt, sondern dass man sie kaum braucht.

WAS DER BETRIEB IM ALLTAG BRAUCHT

MITTEL	WAS ES LIEFERT	WANN
Returncode	die Ampel: gut, Warnung, Fehler	immer, in jedem Job
Report	was verarbeitet wurde und was nicht, je Element	wenn ein Lauf erklärt werden muss
Logging	der Verlauf im Detail, gestuft	gezielt bei der Fehlersuche, nicht dauerhaft
`flcl INFO GET.SYSTEM`	Version, Lizenz, Konfiguration, Umgebung	vor jedem Ticket

Logging ist mächtig und schreibt viel. Gezielt einschalten, nicht dauerhaft auf der höchsten Stufe fahren.

FLAMv5.21-42969

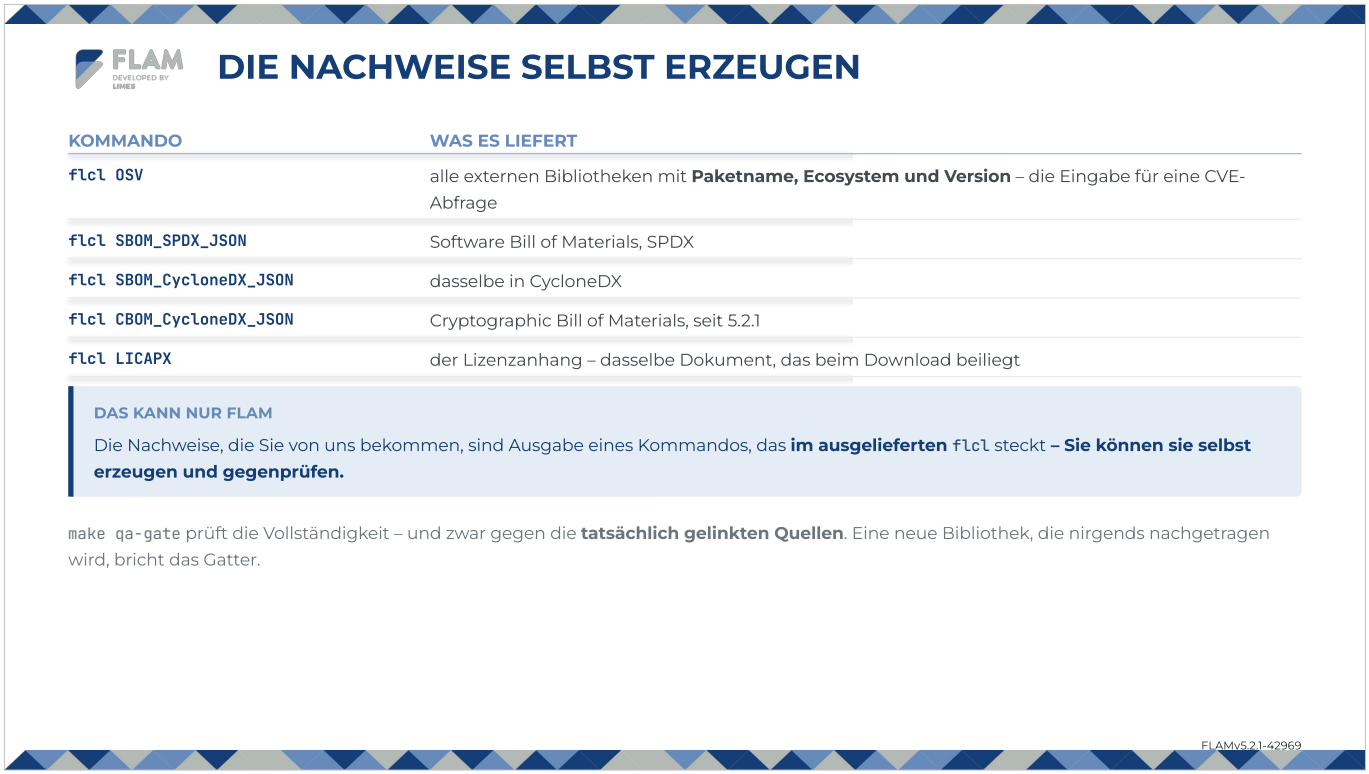
Drei Dinge, die den Unterschied zwischen einem Ticket und einer Selbstloesung machen.
Der Returncode ist die Ampel, der Report die Begründung, das Logging der Verlauf. Wer alle drei kennt, braucht uns seltener.
Beim Logging warnen: es ist mächtig und kann bei falscher Einstellung viel schreiben – nicht dauerhaft auf der höchsten Stufe fahren.

WENN IHR AUDITOR FRAGT

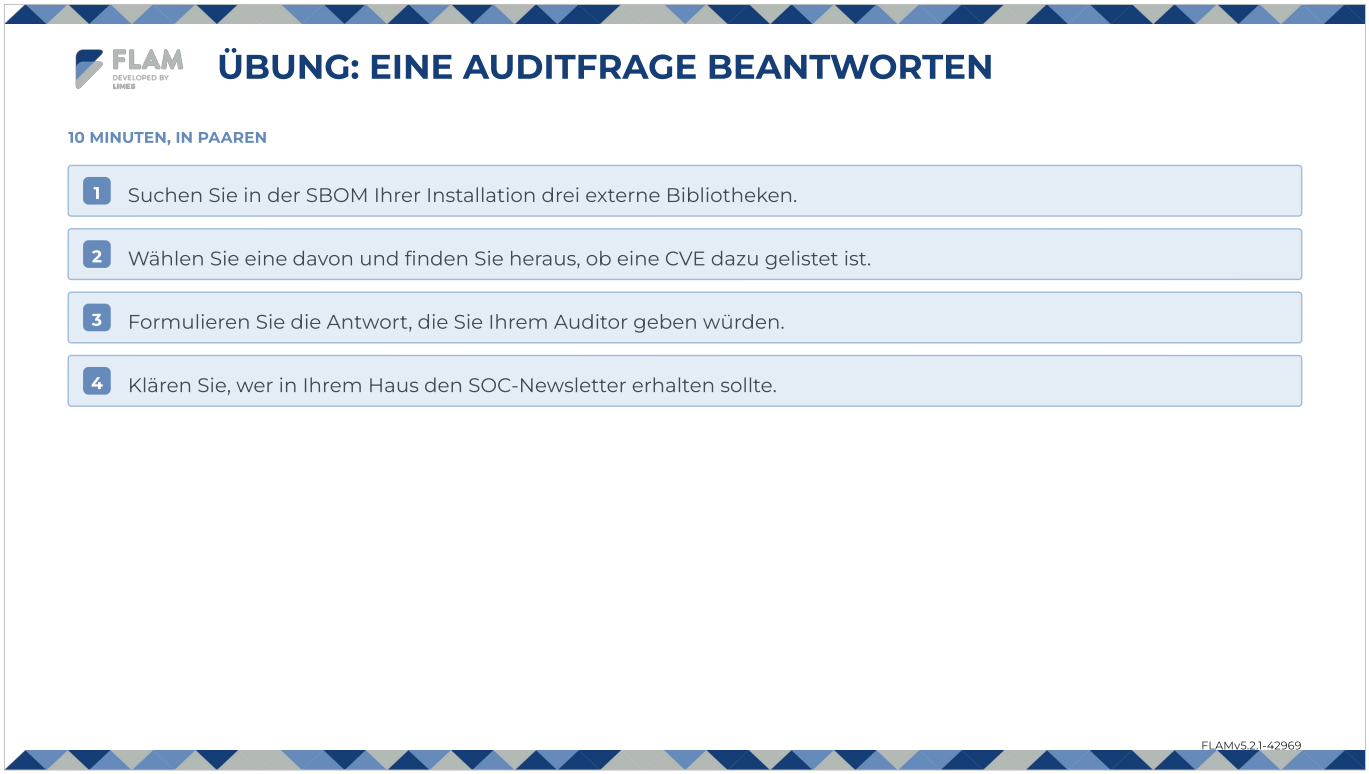
FRAGE IM AUDIT	WORAUS SIE ANTWORTEN
Welche Prüfverfahren nutzt der Hersteller?	Qualitätsbericht, mit Befunden und deren Bewertung
Welche Komponenten sind enthalten?	SBOM je Auslieferung
Ist CVE-XXXX-YYYY relevant?	AFFCVE oder NAFCVE, mit Begründung
Wie erfahren wir von Schwachstellen?	SOC-Newsletter nach den Vorgaben von BSI und NIST
Wer betreibt die Software?	Sie – wir haben keinen Zugang zu Ihren Systemen
Gibt es ein Qualitätsmanagement?	ISO 9001 zertifiziert

FLAMv5.21-42969

Diese Folie ist ein Geschenk an die Teilnehmer: sie können damit in ihrem eigenen Haus arbeiten.
Wichtig und ehrlich: wir sind kein Rechenzentrum und haben keinen Zugang zu Kundensystemen.
Der Kunde betreibt die Software eigenverantwortlich. Das ist keine Ausrede, es ändert nur, welche Kontrollen anwendbar sind.
Der SOC-Newsletter ist die Antwort auf die Schwachstellenfrage aus DORA und NIS2 – Adresse aufnehmen lassen.



Diese Folie ist ein kleines Geschenk an den Auditor im Raum, und fast niemand weiß davon: **die Werkzeuge, mit denen wir unsere Nachweise erzeugen, stecken im ausgelieferten `flcl`.** Es sind versteckte Kommandos – sie stehen nicht in der Hilfe, aber sie funktionieren, und Kunden dürfen sie benutzen.
Wir nutzen sie selbst, um SBOM, CBOM, LICAPX, ABOUT, VERSION und die OSV-Liste zu bestimmen. Das ist der Punkt, der Vertrauen schafft: die Nachweise, die Sie von uns bekommen, sind nicht handgeschrieben, sondern Ausgabe eines Kommandos, das Sie selbst laufen lassen können.
`flcl OSV` gibt die Liste aller externen Bibliotheken mit **Paketname, Ecosystem und Version**. Genau diese drei Angaben braucht man für eine CVE-Abfrage bei einem OSV-Server – das Ecosystem ist der Teil, den die meisten SBOM-Werkzeuge weglassen und der die Abfrage erst brauchbar macht.
Das SBOM gibt es in **zwei Formaten**, SPDX und CycloneDX, das CBOM in CycloneDX. Wer ein drittes Format braucht, kann es sich aus der OSV-Liste selbst bauen.
`flcl LICAPX` erzeugt den Anhang mit den Open-Source-Lizenzen – dasselbe Dokument, das beim Download beiliegt.
Und der Punkt, der die Vollständigkeit trägt: **`make qa-gate` prüft, dass nichts fehlt** – keine Lizenz vergessen, LICAPX vollständig, OSV-Listen und SBOM vollständig. Und diese Prüfung fußt auf den **tatsächlich gelinkten Quellen**, nicht auf einer gepflegten Liste. Eine Bibliothek, die hinzukommt und nirgends nachgetragen wird, bricht das Gatter.



Zehn Minuten, in Paaren. Der Effekt ist, dass die Teilnehmer merken: sie können die Frage selbst beantworten, ohne uns anzurufen.

WAS BLEIBT

- eine automatisierte Kette, ein signiertes Paket – Doku, SBOM und Qualitätsbericht gehören dazu
- geprüft wird statisch, dynamisch, auf Speicher, auf Abdeckung und auf Schwachstellen
- zu jeder CVE gibt es eine Antwort: AFFCVE oder NAFCVE, mit Begründung
- Rolling Release plus Rückwärtskompatibilität – 13 Monate Planungsfenster
- Returncode, Report, Logging und `flcl INFO GET.SYSTEM` sind die Werkzeuge des Betriebs

Lernziele durchgehen. Damit ist die Ebene 0 vollständig; es folgen die drei Zugangswege im Detail.

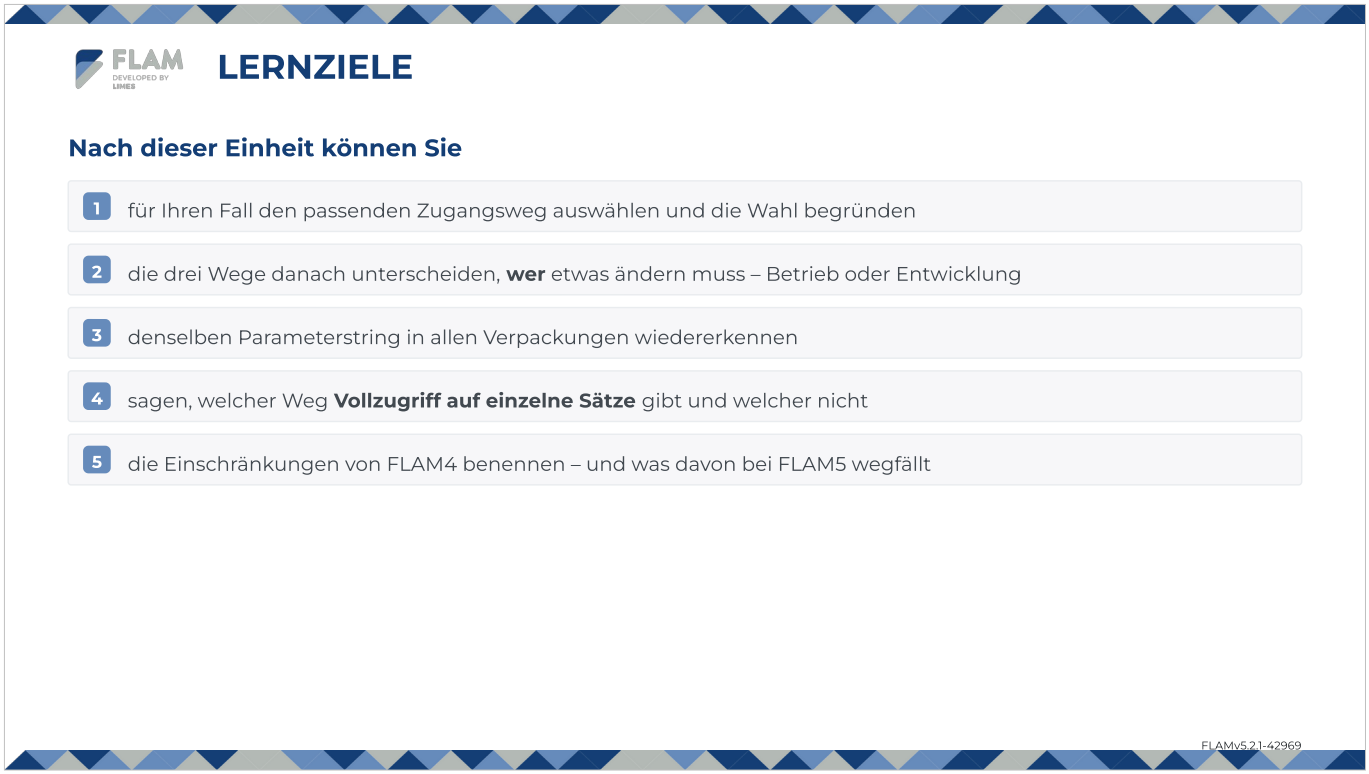
FLAM SCHULUNG, EBENE 0/1

Ein Parameterstring, drei Verpackungen

Batch, Subsystem und API im Vergleich – und was FLAM4 von FLAM5 unterscheidet

Falk Reichbott · limes datentechnik gmbh

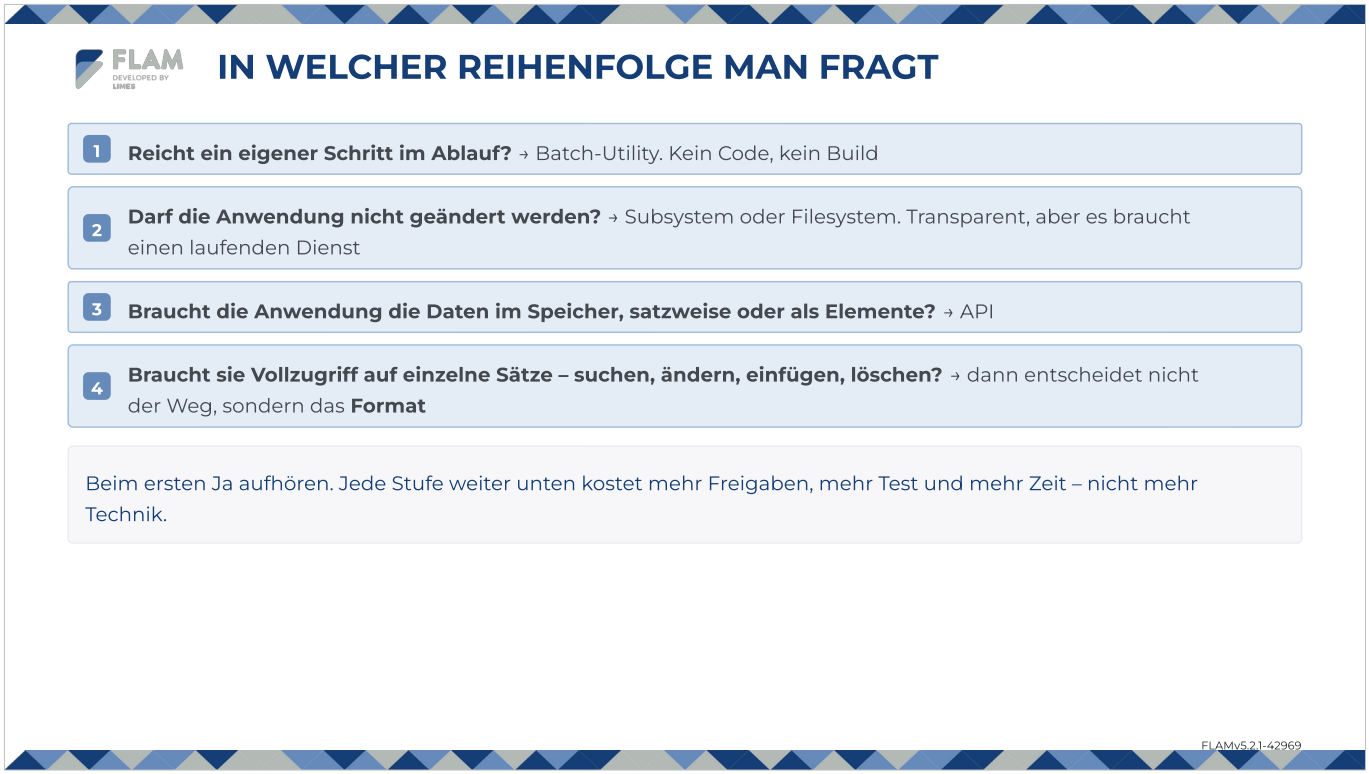
Diese Einheit kommt direkt nach den Grundlagen und ist das Bindeglied: dort haben die Teilnehmer gelernt, dass es einen Parameterstring und drei Wege gibt. Hier sehen sie, was die drei Wege unterscheidet und wie derselbe Fall auf jedem aussieht.
Danach behandelt jede der drei folgenden Einheiten einen Weg im Detail.
Der Bogen: das Kriterium, dann ein Fall viermal gelöst, dann die Frage, die im Raum am häufigsten kommt – wer kann eigentlich einzelne Sätze ändern.



Ziel 1 ist das Arbeitsziel: am Ende soll jeder für seinen eigenen Fall den Weg begründen können.
Ziel 4 ist das, was am meisten falsche Erwartungen verhindert.



Die Tabelle, um die sich alles dreht. Sie stand vorher in jedem Ebene-1-Deck einmal; hier steht sie einmal richtig.
Die entscheidende Spalte ist die zweite: **wer** muss etwas ändern. Das ist in fast jedem Projekt die eigentliche Frage, weil sie über Aufwand, Freigaben und Zeitplan entscheidet – nicht die Technik.
Batch und Subsystem sind Betriebsthemen: eine JCL, ein Skript, ein DD-Statement, ein Mount. Das API ist ein Entwicklungsthema: Quellcode, Bindung, Test, Freigabe.
Daraus folgt die Reihenfolge, in der man in einem Projekt fragt: geht es im Batch? Wenn nein, geht es transparent über das Subsystem? Wenn nein, dann das API. Wer umgekehrt anfängt, baut oft Wochen an der falschen Stelle.
Die dritte Spalte ist die, nach der der Betrieb fragt – was muss dafür laufen.



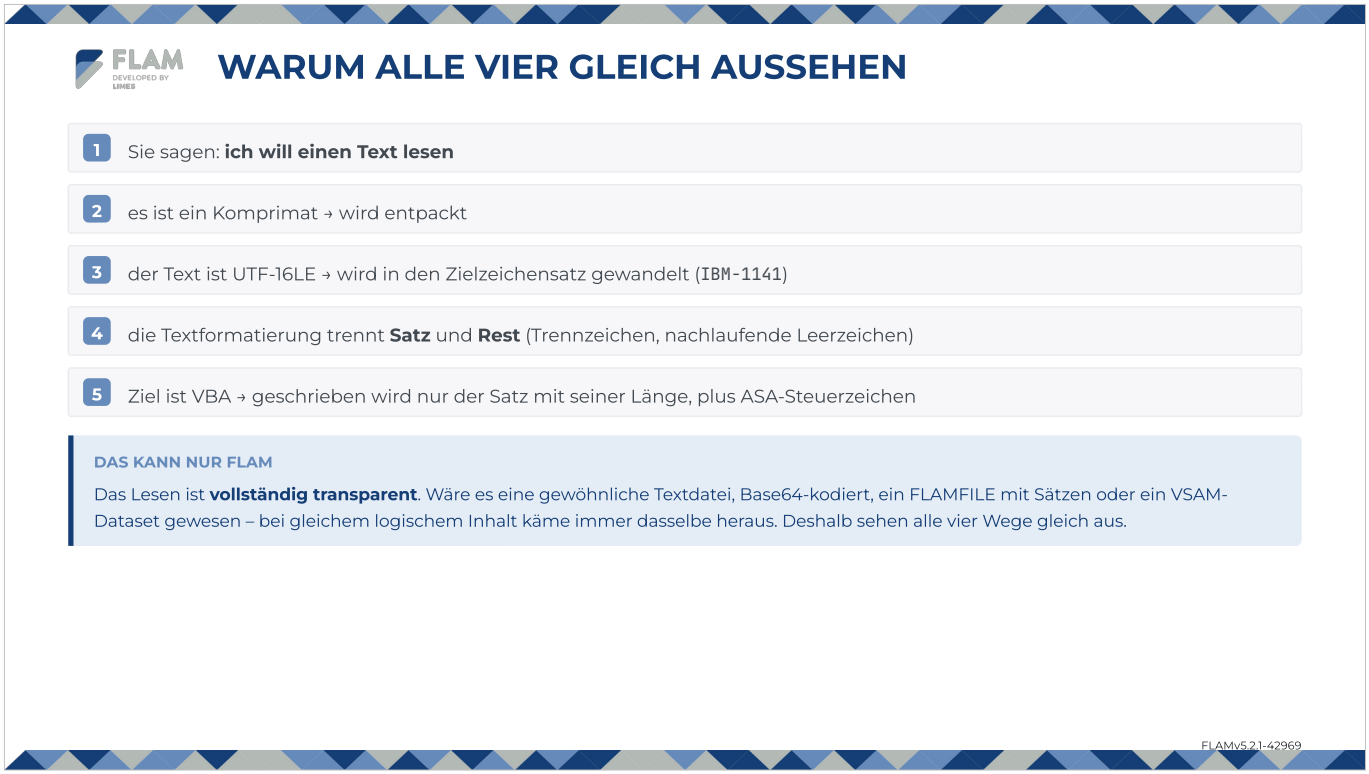
Die praktische Folie für ein Projektgespräch. Von oben nach unten durchgehen, und beim ersten Ja aufhören.
Der Grund für die Reihenfolge ist nicht technische Eleganz, sondern Aufwand:
jede Stufe weiter unten kostet mehr Freigaben, mehr Test und mehr Zeit.
Punkt vier ist die Ausnahme von der Reihenfolge: wer Vollzugriff auf einzelne Sätze braucht, muss nicht von oben anfangen, sondern kann direkt zur letzten Zeile springen. Dazu die Folie danach.



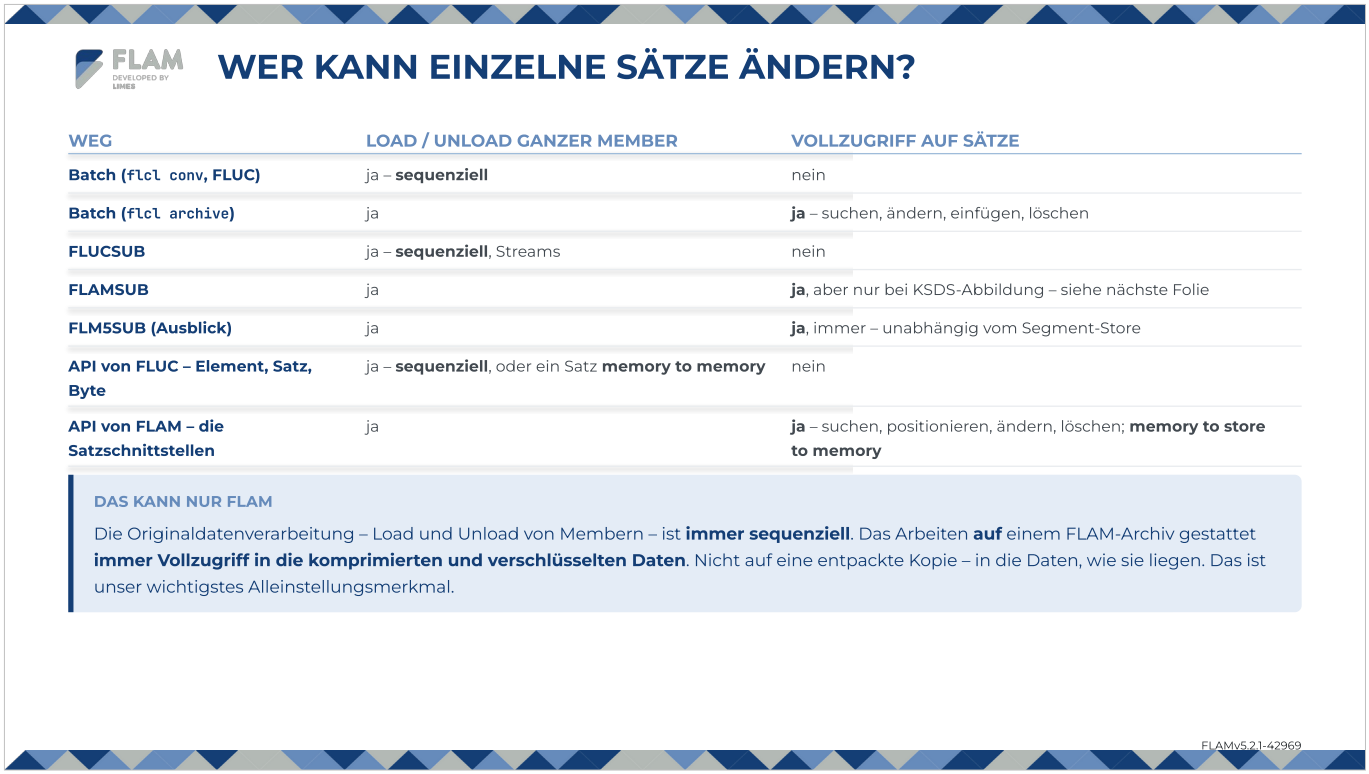
Das beste Beispiel, das wir haben: derselbe Fall wird viermal gelöst, und die Lösungen unterscheiden sich nur in der Verpackung.
Die Aufgabe langsam vorlesen, sie ist absichtlich unangenehm: in einem XMIT-PDS auf dem Host, Satzformat FB80, liegt ein Member DOCUMENT. Das ist eine **binäre GZIP-Datei**. Darin steckt ein Text im SWIFT-Format, kodiert in **UTF-16LE**, unter Windows erzeugt, mit **CRLF** als Satztrenner.
Gewünscht ist: jeden Satz korrekt in EBCDIC in einer **VBA-Datei**.
Die Teilnehmer einmal überlegen lassen, wie viele Schritte sie dafür brauchen würden. Die üblichen Antworten sind drei bis fünf Steps.
Und dann die Auflösung: es ist **ein** Aufruf, und man sagt darin nicht ein einziges Mal, dass entpackt werden soll. Man sagt nur, dass man einen Text lesen will.



Nicht alle vier vorlesen. Auf die Gemeinsamkeit zeigen: in jedem der vier Kästen steht derselbe Parameterstring, `read.text(...)`. Was sich unterscheidet, ist ausschliesslich die Verpackung.
FLCL im Batch: ein Job-Step, `write.record` mit `falloc(recf=VBA)`.
Das Subsystem: der Trick ist, dass hier **IEBGENER** die Arbeit macht – ein IBM-Utility, das von FLAM nichts weiß. Das `SUBSYS=(FLUC,...)` steht am Eingabe-DD, und der Parameterstring liegt in einem eigenen DD. Statt `write.record` steht hier `format.record`, weil das Ziel nicht eine Datei ist, sondern die Schnittstelle zum Utility.
Das Unterprogramm: ein `flucconv`-Aufruf mit demselben String. Verfügbar als DLL für C und C++ und als Lademodul-Bibliothek für COBOL und PL/I.
Das Byte-Interface: `fcbopen` mit den zwei Strings von der API-Einheit, danach eine gewöhnliche Leseschleife.
Die Frage, die hier kommt: warum steht beim Subsystem und beim Byte-Interface `format.record` und beim Batch `write.record`? Weil im ersten Fall die Daten in den Speicher des Aufrufers gehen und im zweiten in eine Datei. Das ist genau die Unterscheidung von Datei-String und Format-String.



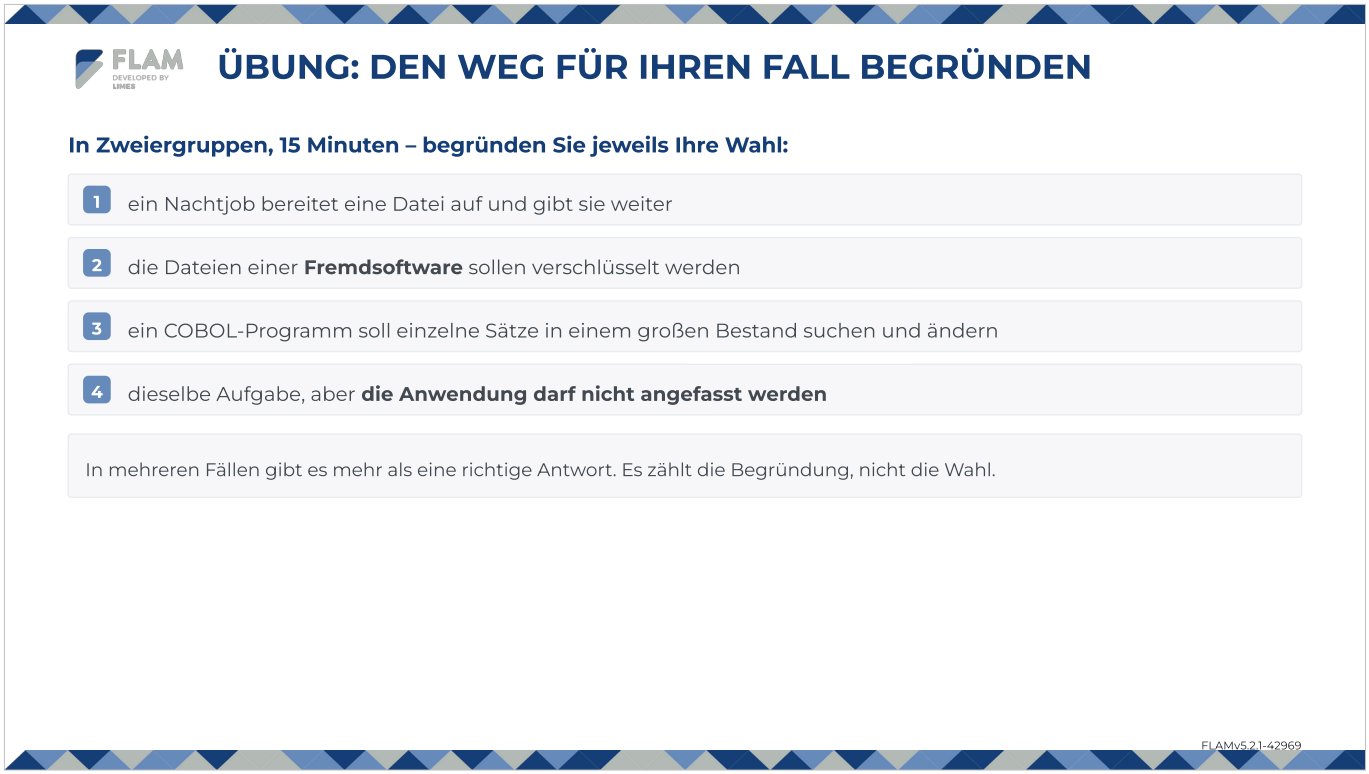
Die Auflösung des Beispiels, und die Folie mit der eigentlichen Botschaft der ganzen Ebene 1. Die Kette einmal von oben nach unten erzählen: der Anwender sagt, dass er einen **Text** lesen will. Daraus folgt alles andere von selbst.
Wir stoßen auf ein Komprimat, also wird entpackt. Wir erkennen, dass der entpackte Text UTF-16LE ist, also wird gewandelt – in den Zielzeichensatz der Plattform, hier IBM-1141. Die Textformatierung teilt die Daten in Paare aus Satz und Rest, wobei der Rest das Trennzeichen und mögliche nachlaufende Leerzeichen ist. Weil satzorientiert in eine VBA-Datei geschrieben wird, schreiben wir nur den Satz mit seiner Länge und setzen das ASA-Steuerzeichen davor.
Und jetzt der Satz, der die Einheit tragen soll: das Lesen ist **vollständig transparent**. Wäre die Datei eine gewöhnliche Textdatei mit Trennzeichen gewesen, oder Base64-kodiert, oder ein FLAMFILE mit Sätzen, oder ein satzorientiertes VSAM-Dataset – das Ergebnis wäre bei gleichem logischem Inhalt immer dasselbe.
Wenn Zeit ist, den Zusatz aus Altfolie 35 bringen: an denselben Aufruf kann man beliebig viele weitere Writes hängen. Etwa dieselben Daten zusätzlich als UTF-8 mit 0x0A als GZIP per SSH auf einen Unix-Server und als ZIP mit CP1252 für Windows – in einem Aufruf.



Die Folie, die die häufigste falsche Erwartung auflöst – und ich hatte sie selbst falsch im Kopf, deshalb steht sie jetzt so deutlich hier.
Die falsche Verkürzung lautet: "Vollzugriff auf einzelne Sätze gibt nur das API". Das ist nicht richtig. **Auch das Subsystem gibt Vollzugriff** – FLAMSUB heute unter Bedingungen, FLM5SUB künftig ohne. Das ist der große Unterschied zwischen FLAM und FLUC: FLUC ist Konvertierung und damit sequenziell, FLAM ist eine Zugriffsmethode.
Die Regel, die man sich merken sollte und die für alles gilt, was danach kommt:
Erstens: die **Originaldatenverarbeitung** – also Load und Unload von Membern – ist **immer sequenziell**. Das ist der FLUC-Weg. Eine ganze Datei rein, eine ganze Datei raus.
Zweitens: das **Arbeiten auf einem FLAM-Archiv** gestattet **immer den Vollzugriff in die komprimierten und verschlüsselten Daten**. Nicht auf eine entpackte Kopie – in die Daten, wie sie liegen.
Der zweite Satz ist unser wichtigstes Alleinstellungsmerkmal. Wenn Sie einen Satz aus dieser Einheit mitnehmen, dann diesen.
Und weil "das API" zu grob ist: auch dort muss man **FLUC von FLAM** unterscheiden. Die FLUC-Schnittstellen – Element, Satz, Byte – lesen oder schreiben **sequenziell**, oder sie wandeln einen einzelnen Satz **memory to memory**, ohne dass eine Datei im Spiel ist. Das ist der Weg für das Online-Geschäft. Vollzugriff gibt es dort nicht, und das ist keine Einschränkung, sondern die Natur einer Konvertierung.
Die FLAM-Satzschnittstellen arbeiten dagegen **gegen das Archiv** und können deshalb den Vollzugriff: **memory to store to memory**. Wer also "Einzelsatzzugriff über das API" sagt, meint immer die FLAM-Seite.



Jetzt die Einschränkungen von FLAM4, und die muss man kennen, weil die großen Bestände bei den Kunden FLAM4-Archive sind. Der Reflex im Raum ist oft: "Satzschnittstelle, das kennen wir, das kann nur wenig" – weil sie die alte FLAM4-Satzschnittstelle und deren Grenzen kennen.
Erste Grenze, das Dataset. Vollzugriff geht nur gegen Dataset-Typen, die selbst Vollzugriff anbieten – auf dem Host also nur VSAM-KSDS. Und zwar nur, wenn auch das FLAMFILE eine KSDS ist. Man kann RRDS, ESDS und KSDS eins zu eins durch ein FLAMFILE ersetzen, wenn dieses wieder KSDS, ESDS oder RRDS ist. LDS ginge auch, braucht aber niemand.
Zweite Grenze, das Kompressionsverfahren, und die ist die unangenehme: beim **ADC geht nur Update in place**. Kein Vergrößern, kein Verkleinern eines Satzes. Genau deshalb sind CX8 und VR8 bei Kunden weiter im Einsatz – nicht aus Gewohnheit, sondern weil dort mehr geht.
Dritte Grenze, der Integritätsschutz, und hier wird es fein: bei CX8 und VR8 gibt es Verschlüsselung, aber **ohne MAC**. Beim ADC gibt es Verschlüsselung **mit** MAC – aber sobald man Vollzugriff macht, also in place ändert, ist der MAC nicht mehr gültig und muss ignoriert werden.
Und dann der Kontrast, auf den die Folie hinausläuft: bei FLAM5 fällt das alles weg. Der Vollzugriff geht immer, unabhängig von Plattform und Segment-Store. Und Integrität und Vollständigkeit bleiben dabei gewahrt – das war die Anforderung, die den Aufbau von FLAM5 bestimmt hat.
Der Nebensatz für die Technikinteressierten: VSAM-KSDS unterstützen wir als Segment-Store nicht und vermutlich nie. Lokal nutzen wir USS-Pathnames, also Folder und File – und das zFS darunter liegt physisch in einem VSAM-LDS, also in der schnellen, flachen Ausprägung.



15 Minuten in Zweiergruppen, dann zwei Fälle im Plenum.
Die Übung ist absichtlich so gebaut, dass die Begründung zählt und nicht die Wahl – es gibt in mehreren Fällen mehr als eine richtige Antwort.
Auflösung:
1. Ein Nachtjob, der eine Datei aufbereitet und weitergibt: **Batch**. Kein Code, kein Build, keine Freigabe für eine Anwendungsänderung.
2. Eine Fremdsoftware, deren Dateien verschlüsselt werden sollen:
Subsystem oder Filesystem. Man darf und kann nichts ändern.
3. Ein COBOL-Programm, das einzelne Sätze in einem großen Bestand sucht und ändert: **API mit FLAM5-Archiv**. Über FLAMSUB ginge es nur bei KSDS-Abbildung und dann ohne gültigen MAC.
4. Dieselbe Aufgabe, aber die Anwendung darf nicht angefasst werden:
FLAMSUB heute unter den Bedingungen der vorigen Folie – oder warten auf FLM5SUB. Das ist die ehrliche Antwort, und sie ist besser als eine falsche Zusage.

WAS BLEIBT

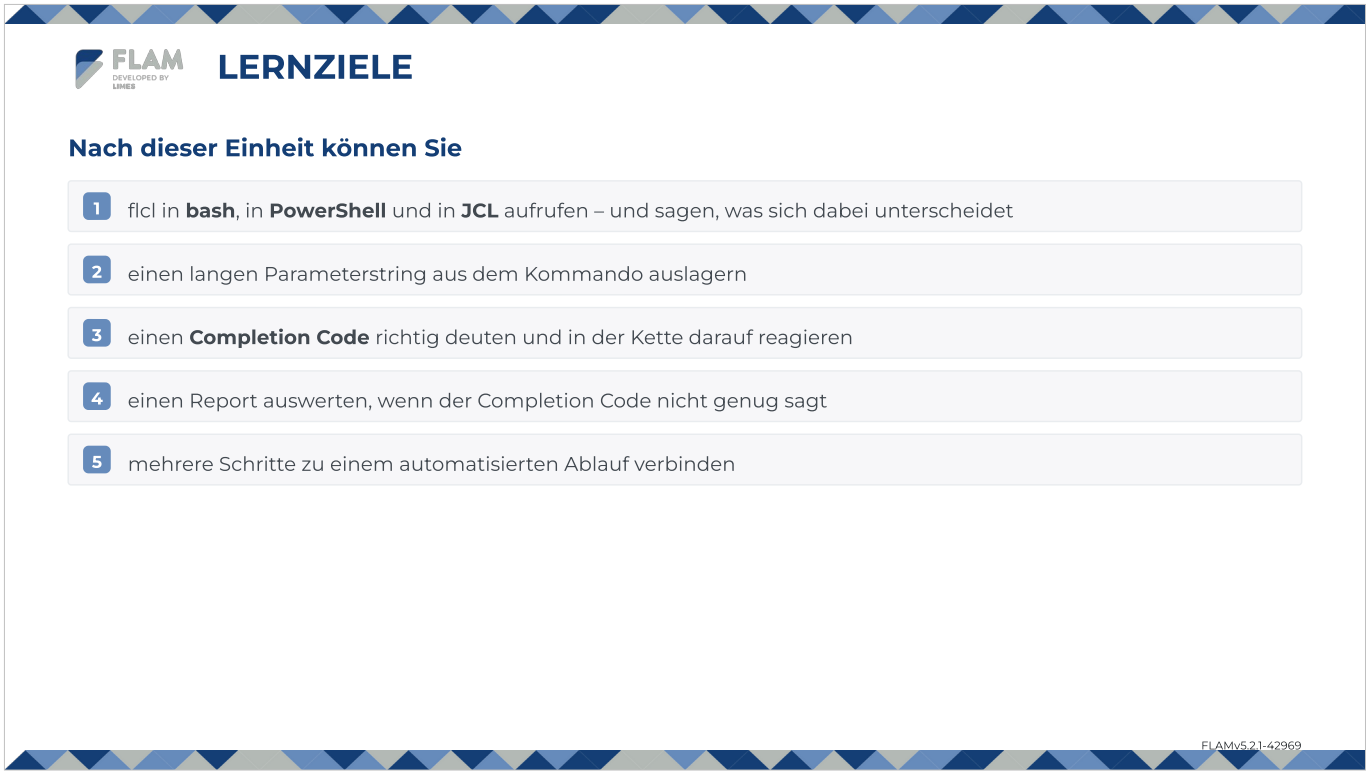
- derselbe **Parameterstring** in allen drei Wegen – was sich unterscheidet, ist die Verpackung
- die Frage ist **wer ändern muss**: Batch und Subsystem sind Betrieb, das API ist Entwicklung
- beim ersten Ja aufhören – jede Stufe weiter unten kostet mehr Freigaben und Zeit
- **Load und Unload sind immer sequenziell; das Arbeiten auf einem Archiv gibt immer Vollzugriff**
- die Grenzen, die Kunden von der Satzschnittstelle kennen, sind **FLAM4**-Grenzen

Die nächsten drei Einheiten nehmen je einen Weg im Detail: Batch-Utility, Subsystem und Filesystem, dann die APIs.

Die zwei Merksätze sind der erste und der vierte Punkt. Danach gehen die drei folgenden Einheiten je einen Weg im Detail durch.



Erster der drei Zugangswege, und der, mit dem achtzig Prozent der Kunden anfangen und aufhören.
Der Bogen: Aufruf in drei Umgebungen, Parameter auslagern, Ergebnis auswerten, automatisieren.
Der wichtigste Teil ist das Auswerten, und dort steckt die eine Aussage, die sitzen muss: die Completion Codes 1 und 2 sind Erfolg. Wer sie als Fehler behandelt, bricht Jobs ab, die durchgelaufen sind.



Ziel drei und vier sind die betrieblich wichtigen – ohne sie ist eine FLAM-Stufe in einer Kette nicht ueberwachbar.

DERSELBE AUFRUF, DREI WELTEN

LINUX, AIX, SOLARIS, Z/OS USS – BASH

```
flcl CONV "read.text(file='ein.txt' ccsid='IBM-1141') \
           write.binary(file='aus.bin')"
```

WINDOWS – POWERSHELL

```
flcl CONV "read.text(file='ein.txt' ccsid='IBM-1141') `
           write.binary(file='aus.bin')"
```

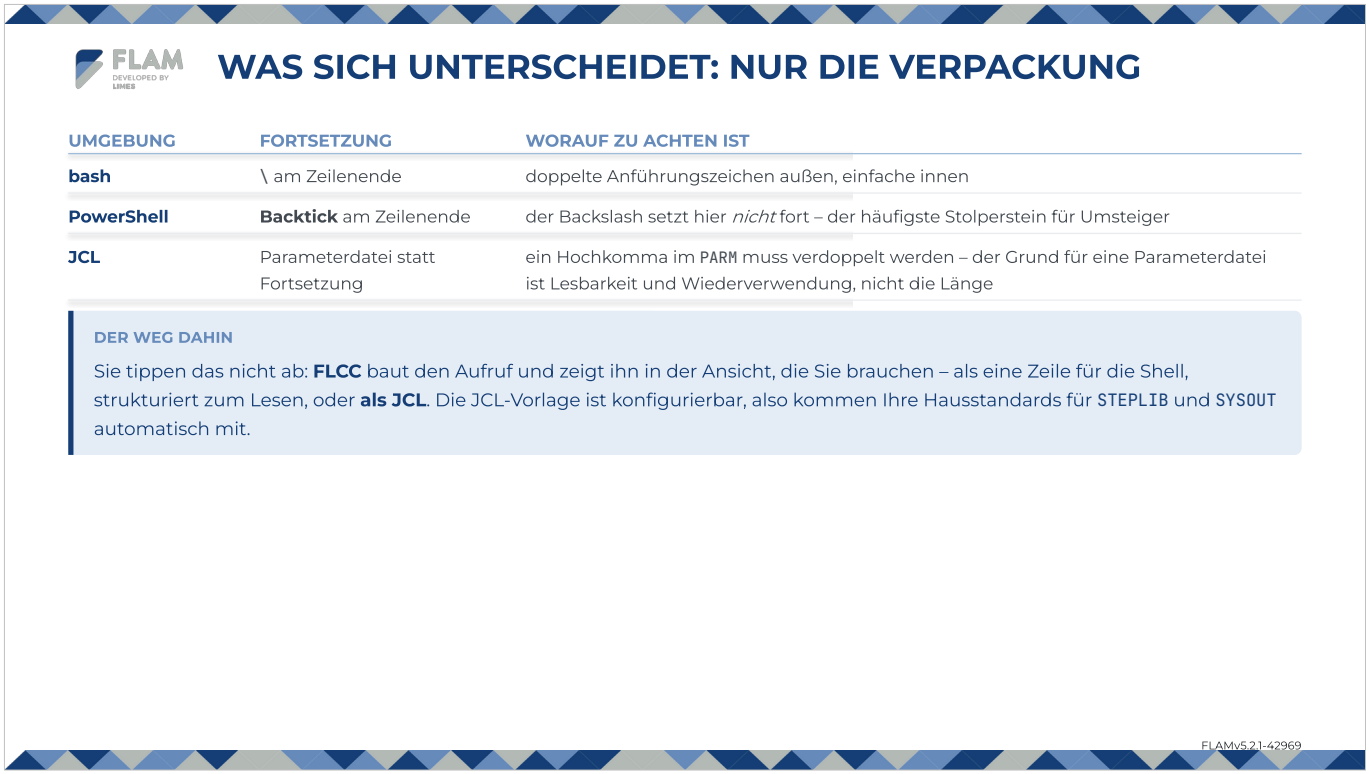
Z/OS – JCL

```
//CONV    EXEC PGM=FLCL,PARM='CONV MAXCC=-2'
//STEPLIB  DD DSN=FLAM.LOAD,DISP=SHR
//SYSPRINT DD SYSOUT=*
//FLAMPAR  DD *
  read.text(file='ein.txt' ccsid='IBM-1141');
  write.binary(file='aus.bin');
```

Der Parameterstring ist in allen drei Fällen identisch.

FLAMv5.21-42969

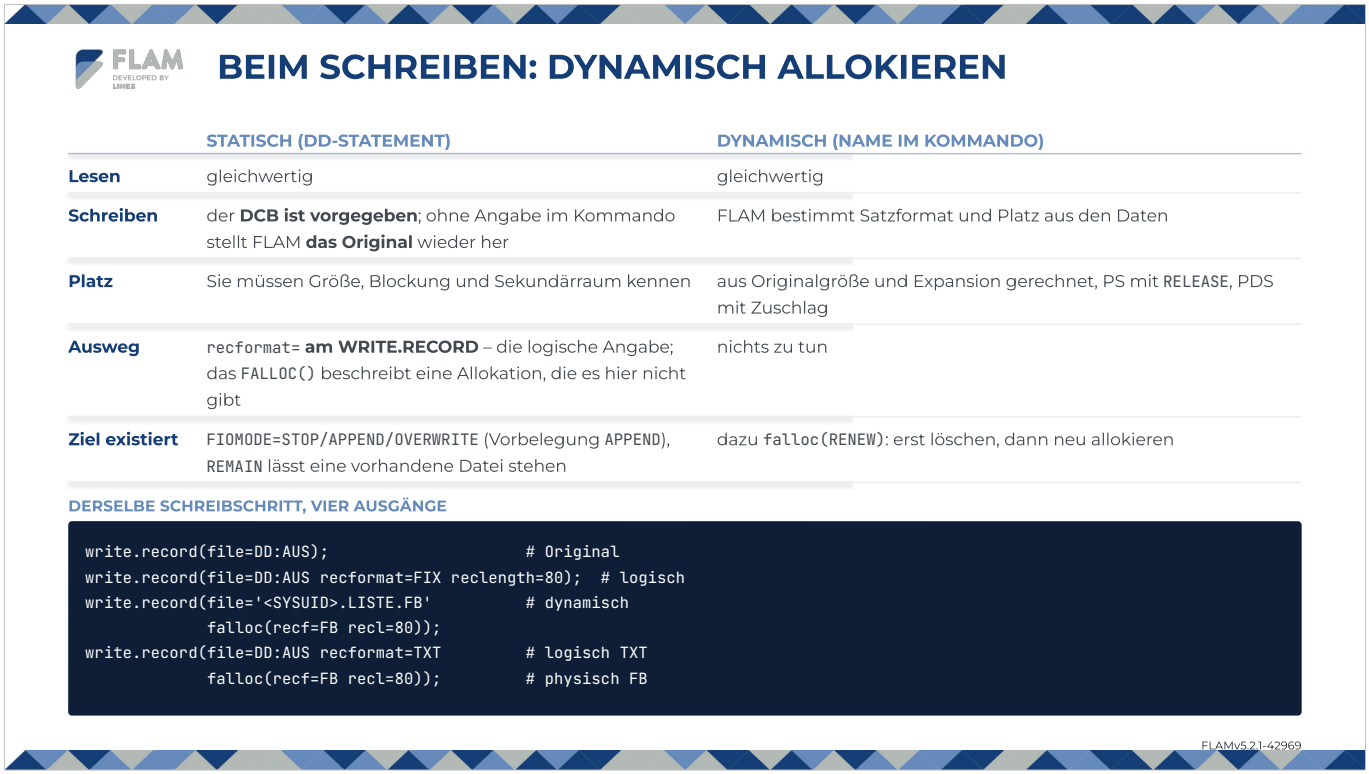
Die Kernfolie der Einheit. Drei Umgebungen, ein Parameterstring.
Langsam machen und den Teilnehmern zeigen, dass in allen drei Kästen buchstäblich dieselbe Zeichenfolge steht. Alles, was sich unterscheidet, ist die Verpackung der jeweiligen Umgebung – nicht FLAM.
Bash: doppelte Anführungszeichen außen, einfache innen, Backslash am Zeilenende für die Fortsetzung.
PowerShell: genauso, aber die Fortsetzung ist ein Backtick. Wer aus der Unix-Welt kommt, stolpert hier – und wer den Aufruf aus FLCC kopiert, bekommt ihn ohnehin als eine Zeile und braucht keine Fortsetzung.
JCL: der String steht nicht im PARM, sondern in einer Parameterdatei. Auf die verdoppelten Hochkommata hinweisen, wenn der String doch ins PARM soll – das ist die häufigste Ursache für angebliche Syntaxfehler, und es ist JCL, nicht FLAM.
Wenn Zeit ist: alle drei einmal laufen lassen. Der Effekt "das ist wirklich dasselbe" trägt die ganze Einheit.



Die Nachschlagefolie zur vorigen. Diese drei Zeilen sind der ganze Unterschied, und sie sind die drei häufigsten Supportfälle beim Einstieg.
Bash: Backslash setzt fort, einfache Anführungszeichen innen sind literal.
PowerShell: der Backtick setzt fort, nicht der Backslash. Das ist der Klassiker.
JCL: ein Hochkomma im PARM muss verdoppelt werden – deshalb gehört ein ernstgemeinter String in eine Parameterdatei und nicht ins PARM.
Zur Länge, falls die Frage kommt: **von unserer Seite gibt es keine Grenze.** FLAM5 verarbeitet einen beliebig langen Parameterstring, und FLAM4 nimmt, was z/OS maximal übergibt. Die alte Grenze von 100 Zeichen ist Geschichte, und mit `PARMDD` gibt es ohnehin einen zweiten Weg, den String aus einer DD-Karte zu lesen. Der Grund für eine Parameterdatei ist also nicht die Länge, sondern Lesbarkeit, Wiederverwendung und die verdoppelten Hochkommata – `FLAMPAR` oder `DD:PARM`.
Drei Vorbelegungen bei den DD-Namen, die kaum bekannt sind und eine JCL kürzer machen: steht am `READ` keine Datei, sucht FLAM **`FLAMIN`**, am `WRITE` **`FLAMOUT`**, und für eine FLAMFILE zusätzlich **`FLAMFILE`** – das Letzte aus Verträglichkeit mit dem alten FLAM-Utility. Wer die drei Allokationen so benennt, braucht im Kommando keinen Dateinamen.
Und dann der Punkt, der den Teilnehmern die meiste Tipparbeit erspart: FLCC baut den Aufruf und gibt ihn in der Ansicht, die man braucht. Eine Zeile für die Shell, strukturiert zum Lesen, oder direkt als JCL. Die JCL-Vorlage ist konfigurierbar – wer Hausstandards für STEPLIB, CEEOPTS und SYSOUT hat, legt sie einmal in FLCC ab und bekommt sie ab dann in jedem generierten Job.
Das ist die Brücke zur Einheit über die Werkzeuge: FLCC ist unser interaktives Handbuch, und hier ist es außerdem der Generator.



Produktionsstrings sind lang. Wer sie in die JCL schreibt, ändert die JCL bei jeder Kleinigkeit – und braucht dafür eine Freigabe.
Deshalb: FLAMPAR oder eine Parameterdatei, und alles Wiederkehrende in eine Property-Datei. Die JCL bleibt dann stabil.
Zwei Bequemlichkeiten, die kaum jemand kennt und die die JCL kürzer machen:
Erstens: **steht im `PARM` nur das Kommando, liest FLCL von selbst die Allokation `FLAMPAR`.** `PARM='CONV'` genügt also, wenn die Parameter unter `//FLAMPAR DD ` *stehen – so steht es auch in den Beispielen des Handbuchs.*
Das Gleichheitszeichen gehört dort **nicht** hin: `PARM='CONV='` sagt "Parameterdatei", und der Name dahinter fehlt. Nachgefahren mit 5.2.1:
`Command file: No file name provided`, Completion Code 36 (Falk hatte den Verdacht, 2026-09-16).
Zweitens: **das `DD:` vor einem DD-Namen ist optional.** Alles, was nur einen Qualifier hat, wird als DD-Name gelesen. `PARM='CONV=PARM'` und `PARM='CONV=DD:PARM'` sind dasselbe.
Zum Owner die Begründung, weil sie oft fehlt: **er dient vor allem der Mandantentrennung bei großen Betreibern**, die für mehr als einen Kunden arbeiten – IBM und Kyndryl, T-Systems und vergleichbare. Jeder Mandant bekommt seine eigene Konfiguration unter seinem Owner, im selben System und ohne dass sich die Einstellungen ins Gehege kommen. Test gegen Produktion ist der einfachere Fall derselben Mechanik.
Und die dritte Stufe, die auf dieser Folie oft fehlt und in der Praxis die wichtigste ist: **nicht nur der ganze Parameterstring, auch ein einzelnes String-Argument darf aus einer Datei kommen** – `pass=f'DD:PASS'`. Damit steht eine Passphrase weder in der JCL noch im Log noch in der Prozessliste. Der Operator `f'...'` gilt überall, wo ein String erwartet wird; `=>` gilt für den Parameterstring eines Kommandos, eines Overlays oder eines Objekts (Falk, 2026-09-16).
Rangfolge nicht vergessen (siehe Deck Administration): die Kommandozeile gewinnt gegen alles davor.
Der Satz, mit dem man die Folie verlässt: **Ziel ist eine JCL, die sich nicht mehr ändert.** Alles, was sich ändert, gehört daneben.
Und die Frage, die im Betrieb dazugehört: **was passiert, wenn das Ziel schon da ist?** `FIOMODE` entscheidet es – `APPEND` ist die Vorbelegung und hängt an, `OVERWRITE` schreibt drüber, `STOP` bricht ab. `REMAIN` ist die vorsichtige Variante: eine vorhandene Datei bleibt unangetastet. Und `falloc(RENEW)` ist die andere Richtung: die vorhandene Datei wird gelöscht, damit neu allokiert werden kann – der Weg, wenn sich die DCB-Angaben geändert haben.



Die Folie, die einen Supportfall verhindert, den wir regelmäßig sehen.
Beim **Lesen** ist es gleichgültig, ob eine Datei statisch über ein DD-Statement oder dynamisch über ihren Namen angesprochen wird. Beim **Schreiben** nicht.
Eine statische Allokation gibt den **DCB vor** – Satzformat, Satzlänge, Blockgröße. FLAM bekommt damit keine Vorgabe, wie es die Daten formen soll, und tut dann das Naheliegende: es versucht, **das Original wiederherzustellen**. Genau das ist der Fall, den Kunden nicht erwarten. Ein Beispiel: im Archiv liegt eine Textmatrix. Das Original war eine gewöhnliche Textdatei, also ein Bytestrom mit Zeilentrennern. Beim Schreiben in ein FB-Dataset entsteht daraus wieder ein Bytestrom – die Trenner stehen mit in den Daten, und jeder Satz ist bis auf den letzten randvoll. Wer Sätze erwartet hat, findet eine Bytewurst.
Der Ausweg, wenn es eine statische Allokation sein muss: das Satzformat am `WRITE.RECORD` selbst angeben – `recformat=` und `reclength=`. Das ist die **logische** Angabe: sie sagt, wie die Daten in Sätze geschnitten werden. Das `FALLOC()` beschreibt dagegen die **Allokation**, und die gibt es bei einem DD-Statement gar nicht – man würde also etwas beschreiben, das nicht stattfindet. Bis 5.2.0 war `falloc(recformat=…)` der einzige Weg und wirkte ersatzweise auch als logische Angabe; das bleibt gültig, ist aber nicht mehr die richtige Formulierung. Dann weiß FLAM, dass es Sätze schreiben soll – den Nettosatz je Satz –, und nicht die Form des Originals. War das Original selbst ein Satz-Dataset, passt es ohnehin.
Der bessere Weg ist die dynamische Allokation, und das steht auch so im Handbuch: "In all other cases, we recommend using dynamic allocation when writing new files." Der Grund ist nicht Bequemlichkeit, sondern die Platzrechnung: FLAM arbeitet den ersten Block im Open vollständig durch, kennt also die Größe des Originals und die Expansion oder Verdichtung, die die Daten durchlaufen. Aus beidem ergeben sich die Space-Angaben. Bei einer PS-Datei kommt `RELEASE` dazu, damit ungenutzter Platz beim Schließen zurückgeht; bei einem PDS legt FLAM etwas drauf und gibt nichts zurück.
Wer statisch allokiert, muss das alles selbst wissen – und rät meistens.

COMPLETION CODES RICHTIG DEUTEN

CODE	BEDEUTUNG	REAKTION IN DER KETTE
`0`	vollständig erfolgreich	weiter
`1`	erfolgreich, aber im Log steht eine relevante Warnung – etwa ein Schlüssel, der bald abläuft	**weiter** – das Log auswerten
`2`	erfolgreich, nur das Aufräumen danach ist fehlgeschlagen	**weiter** – kann ignoriert werden
`4`	Warnung: einige Dateien wurden nicht verarbeitet	fachlich entscheiden, Report auswerten
`8`	Fehler in der Ausführung	abbrechen, Report sichern
`12 … 48`	die Ausführung wurde nie erreicht: Mapping, Syntax, Konfiguration, Lizenz, Speicher	der Aufruf oder die Umgebung ist falsch – nicht die Daten
`64`	fataler Fehler	abbrechen, Support einschalten
`> 64`	Special Condition Code, vom Kommando gezielt für die Jobsteuerung gesetzt	auswerten wie vorgesehen

1 und 2 sind Erfolg. `MAXCC=-2` macht daraus `0`. Das sind die **Completion Codes des Programms** – nicht die Returncodes der API-Aufrufe aus `FLMRTC.h`; die kommen in der Einheit zum API.

FLAMv5.21-42969

Die wichtigste Folie der Einheit für den Betrieb, und die mit dem größten Schadenspotenzial, wenn man sie falsch liest.
Zuerst die Abgrenzung, die alles entscheidet: das hier sind die **Completion Codes des Programms**, wie sie in Anhang G des Handbuchs stehen. Sie sind nicht dasselbe wie die Returncodes der API-Aufrufe – die stehen in CINC/FLMRTC.h und kommen in der Einheit zum API. Wer die zwei Tabellen verwechselt, deutet jeden Lauf falsch.
Die Struktur der Tabelle: bis einschließlich 2 war der Auftrag erfolgreich.
4 heißt Warnung, es wurden einige Dateien nicht verarbeitet. 8 heißt Fehler in der Ausführung. Alles über 8 bis 64 heißt, dass es die Ausführung nie erreicht hat – Mapping, Syntax, Konfiguration, Lizenz. 64 ist der fatale Fehler, und über 64 liegen die Special Condition Codes, die ein Kommando gezielt für die Jobsteuerung setzt.
Das ist die eigentliche Botschaft: **1 und 2 sind Erfolg.** 1 heißt, im Log steht eine relevante Warnung – etwa dass ein Schlüssel bald abläuft. 2 heißt, das Aufräumen nach dem Auftrag ist schiefgegangen; die Aufgabe selbst ist erledigt und kann ignoriert werden.
Der praktische Rat: `MAXCC=-2` setzen, dann werden 1 und 2 zu 0. Die Meldung bleibt im Log – wer sie braucht, wertet das Log aus, nicht den Code.
Feinheit, die im Support hilft: relevant für Code 1 ist nur eine Warnung, die eine tatsächlich benutzte Komponente geschrieben hat. Warnungen von Komponenten, die aus der Verarbeitung herausgefallen sind, weil das Format nicht passte, stehen im Log, machen aber keine 1.



Der Report ist das, was in einer Batch-Kette den Unterschied macht: er sagt, welches Element verarbeitet wurde und welches nicht.
Logging ist die Stufe darunter und gehört in die Fehlersuche, nicht in den Dauerbetrieb – es schreibt viel.
Empfehlung für die Kette: Report immer schreiben, Logging nur bei Bedarf einschalten.
Der Fehlertrace ist der dritte Baustein und der, den man im Supportfall braucht. Das Beispiel unten ist ein echter Completion Code 8, und es lohnt sich, ihn im Raum **von unten nach oben** zu lesen – so ist er gebaut:
`uiFlmDelArc` wollte Rechte ermitteln, dafür musste `uiFlmArcRights` den Segment-Store öffnen, das ging über `uiFlmSegOpn` und `uiFlmStoOpn` bis zu `segstore_objects_open`, und ganz unten steht der eigentliche Grund:
"blob not found (key=root)" – das Root-Objekt fehlt, also existiert der Segment-Store nicht.
Der Punkt für die Teilnehmer: **die oberste Meldung nennt das Symptom, die unterste die Ursache.** Wer nur die erste Zeile ins Ticket kopiert, schickt "Object not found" – und das ist die unbrauchbarste Information des ganzen Traces. Mit `MESSAGE(ERRTRACE)` steht der ganze Weg im Log, mit Quelldatei, Funktion und Kontext.
Genau diesen Trace möchten wir im Ticket sehen. Er erspart die erste Rückfragerunde vollständig.

MEHRERE SCHRITTE VERBINDEN

- flcl fügt sich in Ihre Automatisierung ein und bringt keine eigene mit: JCL, BASH, BAT, PERL, Scheduler
- was in **ein** Kommando passt, gehört in ein Kommando: ein `READ` mit mehreren `WRITE`, ein `XCNV` mit mehreren `OUTPUT`
- **Wildcards, Verzeichnisdurchlauf und Dateiarrays** löst flcl selbst auf – keine Schleife im Skript: `file[a b c]`
- **Pre- und Post-Processing** führt eigene Kommandos vor und nach der Konvertierung aus – je Datei oder je Auftrag, und dort, wo die Datei liegt
- `flcl UTIL run.list` / `run.remove` / `run.files` ersetzt die Aufräum- und Suchschritte der Kette

PRE- UND POST-PROCESSING IM VERZEICHNISDURCHLAUF

```
flcl xcnv "input(sav.fil(fio.blk(file='/in/*.txt' \
       prepro(command='touch [name]')))) \
  output(sav.fil(fio.blk(file='[path]/out/[base].gz' comp.gzip() \
       postpro(command='sha1sum [name]' on=success perfile))))"
```

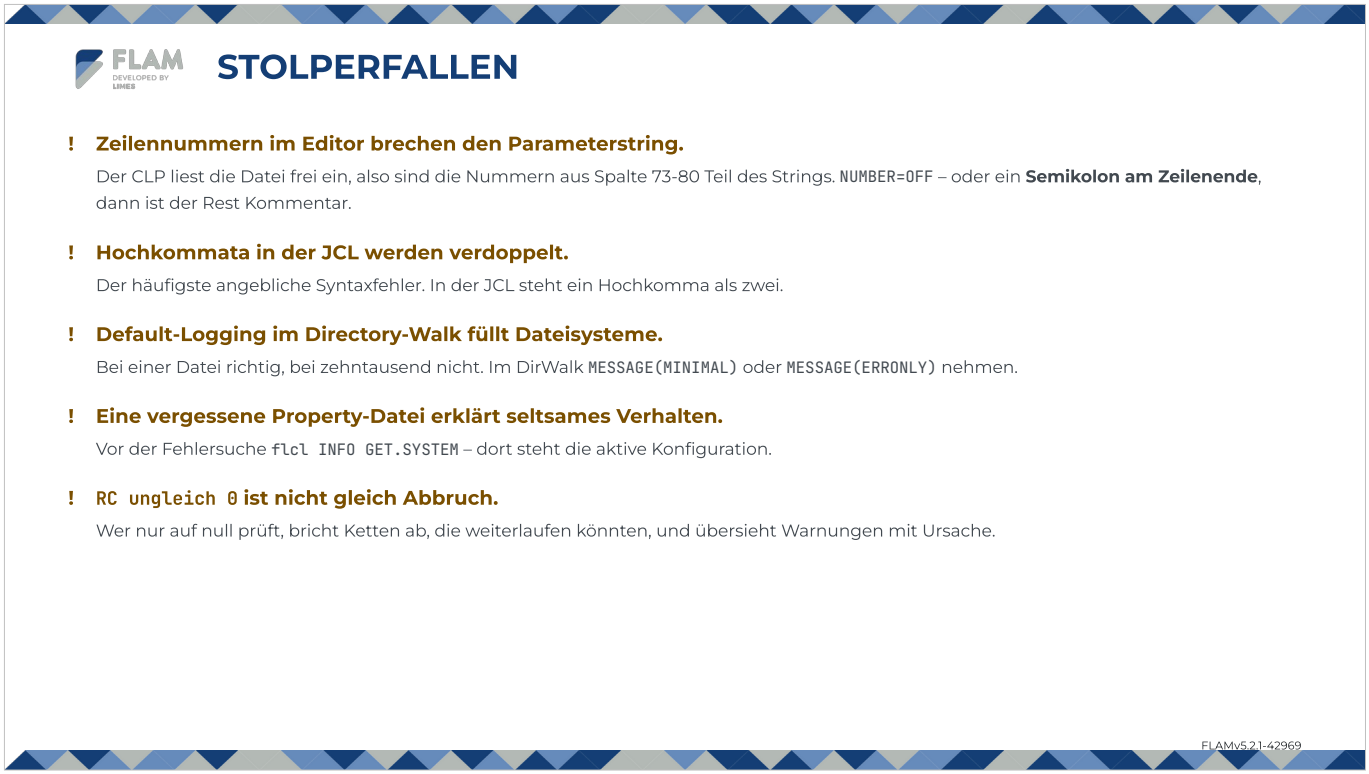
Beim Schreiben läuft Post-Processing standardmäßig nur im finalen Close – bei n Eingaben zu n Ausgaben braucht es `PERFILE`.

FLAMv5.21-42969

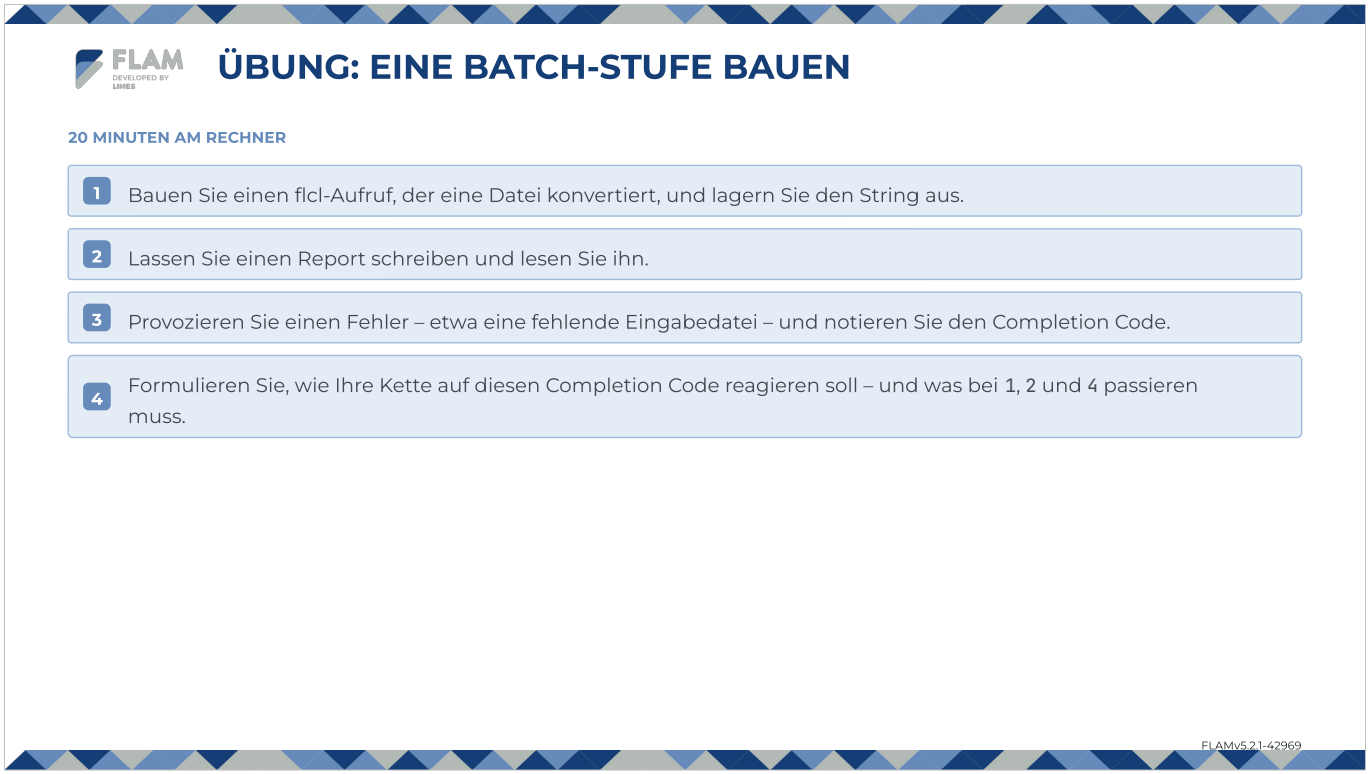
Der Punkt ist, dass FLAM sich in vorhandene Automatisierung einfügt und keine eigene mitbringt: JCL, Shell, Scheduler, was auch immer da ist. Es gibt bewusst keinen FLAM-Scheduler und keine FLAM-Skriptsprache.
Und ein Weg, der zu diesem Zugang gehört und in der API-Einheit nur erwähnt wird: **die Unterprogramme.** Damit ruft ein eigenes Programm **jedes Kommando als Funktion** – also Ganzdateiverarbeitung, inklusive Directory-Walk. Zum Batch-Zugang gehören deshalb drei Dinge: die **FLCL** für die Kommandozeile, die **FLCC** zum Bauen der Strings, und die **Lademodule mit den Kommandofunktionen** für die Integration in eine Anwendung. Wer die Utilities in einem eigenen Werkzeug nutzen will, nimmt das Unterprogramm und nicht die Satzschnittstelle – das ist die häufigste Verwechslung überhaupt.
Die Reihenfolge der Punkte ist die Reihenfolge, in der man sie braucht.
Erstens: was in einem Kommando zusammengefasst werden kann, gehört in ein Kommando – ein READ mit mehreren WRITE, oder ein XCNV mit mehreren OUTPUT.
Das ist der große Hebel und der Grund, warum aus einer Kette ein Step wird.
Zweitens: Wildcards, Verzeichnisdurchlauf und Dateiarrays. flcl löst sie selbst auf, man braucht keine Schleife im Skript.
Drittens, und das kennt fast niemand: Pre- und Post-Processing. Seit 5.1.21 kann man eigene Kommandos vor und nach einer Konvertierung ausführen, je Datei oder je Auftrag, auf der Lese- und auf der Schreibseite. Der interessante Teil: die Kommandos laufen dort, wo die Datei liegt – schreibt man über SSH, laufen sie auf dem entfernten Rechner. Damit fallen die kleinen Hilfsschritte der alten Kette weg.
Die Falle beim Post-Processing nennen: beim Schreiben läuft es standardmäßig nur im finalen Close, also beim Verzeichnisdurchlauf nur einmal für die letzte Datei. Das ist richtig, wenn alles in ein Archiv geht. Wer n Eingaben zu n Ausgaben verarbeitet und je Datei etwas tun will, braucht den Schalter PERFILE (seit 5.1.24).
Viertens die Aufräumarbeiten: `flcl UTIL` erledigt mit `run.list`, `run.remove` und `run.files` genau das, wofür auf dem Host sonst ein IDCAMS-Step steht.
Wichtig, weil ich es hier selbst falsch hatte: es gibt **kein** Kommando `flcl RUN`. `RUN` ist ein Overlay von `UTIL`, und `command=>'datei'` ist eine Parameterdatei, keine Kommandofolge.



Die Folie, nach der im Betrieb am häufigsten gefragt wird – und die, die einen Haufen selbstgeschriebener Schleifen ersetzt.
Die Dateiangabe ist ein Array. Man kann also mehrere Muster nebeneinander angeben, sogenannte Dateilisten, und aus der zusammengesetzten Treffermenge mit `EXCLUDE` wieder ausschließen. `EXCLUDE` ist ebenfalls mehrfach angebbar.
Die Wildcards kurz durchgehen: `` und `?` wie üblich, `+` verlangt mindestens ein weiteres Zeichen. Auf MVS ist der Punkt ein Qualifier-Trenner, deshalb matcht ein einfaches `` nur innerhalb eines Qualifiers – über Qualifier hinweg braucht es `` **oder `++`. Ein Wildcard im Namen selbst wird mit dem Zirkumflex `^` maskiert.**
GDG kann man gezielt ansprechen: `file='~.**(-1)**'` **liest von allen GDGs unter dem eigenen SYSUID die Vorgängergeneration.**
Auch in Archive kann man hineingreifen: `member=` oder ein `?` in der URL wählt Member nach Muster, ein `#` oder `:` nach Index, inklusive Bereichen – `member=#3,5,8-10,15`.
Das `DIR`-Objekt sind die Schalter dazu: `RECURSIVE` für Unterverzeichnisse, `ARCHIVE` um in Archive hineinzugehen, `HIDDEN`, `LINK`, `TAPE`, `ONEFILESYSTEM`.
Und jetzt die zwei Warnungen, die auf dem Host wirklich wehtun – die stehen so auch im Handbuch:
Ein **Wildcard im ersten Qualifier** löst eine Suche über die Catalog Search Interface (CSI) aus. Das vergleicht *jeden* Datensatz des Katalogs gegen das Muster und kann sehr lange dauern.
Und: ein Muster kann eine Datei treffen, die **auf Band liegt oder migriert** ist. Dann wartet der Lauf auf eine Konsoleneingabe. Bänder sind per Default ausgeschlossen, aber das verhindert die Konsolenanfrage nicht in jedem Fall – Vorsicht mit Wildcards am Anfang des Musters.



Alle aus Supportfällen der Batch-Nutzung.
Zur ersten: der ISPF-Editor kann bei FB80-Daten in den Spalten 73-80 Zeilennummern mitführen. Der CLP liest die Parameterdatei frei ein, also stehen diese Nummern mitten im Parameterstring und er meldet einen Syntaxfehler an einer Stelle, die im Editor völlig korrekt aussieht. Zwei Wege heraus:
`NUMBER=OFF` setzen, oder **ein Semikolon an das Zeilenende** – dann ist der Rest der Zeile ein Kommentar und die Nummer stört nicht mehr. Das Semikolon ist der robustere Weg, weil es auch dann noch trägt, wenn jemand anderes die Datei später mit `NUMBER=ON` anfasst.
Zur Logging-Falle: der Unterschied zwischen Einzeldatei und Directory-Walk ist hier groß. Das Default-Logging schreibt pro Datei mehrere Zeilen – bei einer Datei ist das gewollt, bei zehntausend füllt es das Dateisystem und macht den Lauf langsamer. Im Directory-Walk gehört deshalb `MESSAGE(MINIMAL)` oder `MESSAGE(ERRONLY)` in den Aufruf.



Zwanzig Minuten. Ziel ist eine Stufe, die man in eine echte Kette hängen könnte – also mit Report und mit einer Reaktion auf den Completion Code.
Beim vierten Punkt darauf achten, dass sie nicht auf "ungleich null" prüfen:
1 und 2 sind Erfolg.
Auflösung:
1. Der Aufruf ist derselbe in bash, PowerShell und JCL – nur die Verpackung unterscheidet sich: Backslash, Backtick, Parameterdatei.
2. Provoziert man einen Fehler mit einer fehlenden Eingabedatei, kommt `8`:
die Ausführung lief, aber mit Fehler. Wichtig ist, was *nicht* kommt – eine `1` oder `2` wäre Erfolg gewesen.
3. Die Kette muss also drei Fälle unterscheiden: `0` bis `2` weiterlaufen, `4` fachlich entscheiden (einige Dateien wurden nicht verarbeitet), ab `8` abbrechen. Wer auf "ungleich null" prüft, bricht bei `1` und `2` grundlos ab.
4. `MAXCC=-2` macht aus `1` und `2` eine `0`. Die Meldung bleibt im Log – also Log auswerten, nicht den Code.

WAS BLEIBT

- Shell und JCL unterscheiden sich in der Umgebung, nicht in der Sprache
- Ziel ist eine JCL, die sich nicht ändert – alles Veränderliche liegt daneben
- Completion Codes sind gestuft: 1 und 2 sind Erfolg**,** 4 ist eine Warnung – „ungleich null“ ist keine Auswertung
- der Report erklärt den Lauf, das Logging die Ursache
- flcl fügt sich in Ihre Automatisierung ein und löst Wildcards selbst auf

Lernziele durchgehen. Danach folgt derselbe Inhalt über Subsystem und Filesystem – transparent, ohne Aufruf.

FLAM SCHULUNG, EBENE 1

Die Anwendung merkt nichts

Ein DD-Statement auf z/OS, ein Mount auf Linux und Unix

Falk Reichbott · limes datentechnik gmbh

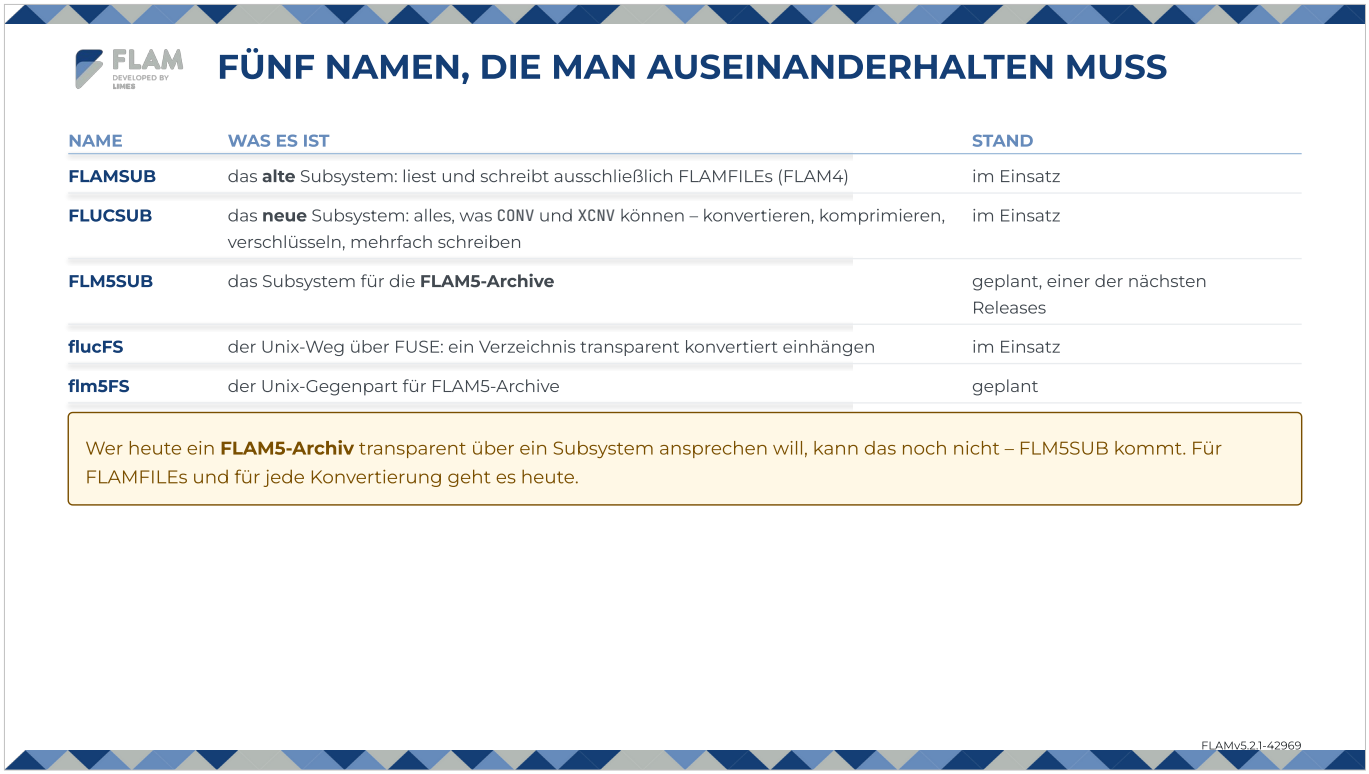
Zweiter Zugangsweg, und der mit dem größten Aha-Effekt: die Anwendung wird nicht angefasst und merkt nichts.
Das ist die Antwort auf „die Anwendung darf nicht geändert werden“ – ein Satz, der in jedem Bestandsprojekt fällt.



Ziel fünf ist das, was einen Einsatz scheitern lässt, wenn man es nicht vorher weiß.



Wörtlich aus dem Handbuch: alle Aufrufe des Anwendungsprogramms für Dateioperationen – öffnen, schließen, Sätze lesen und schreiben – werden vom Subsystem entgegengenommen und verarbeitet. Änderungen im rufenden Programm sind in der Regel nicht notwendig.
Das gilt sprachunabhängig: COBOL, PL/I, C, Java, Assembler. Der Grund ist, dass die Umleitung unterhalb der Sprache passiert.
Zweiter Effekt, der oft den Ausschlag gibt: Komprimierung heißt weniger File-I/O, der Lauf wird also schneller, während er zusätzlich geschützt wird.
Und jetzt der wichtigste Teil dieser Folie, weil er den Einsatzbereich verdoppelt. „Die Anwendung darf nicht geändert werden“ hat zwei ganz verschiedene Ursachen, und beide sind gleich häufig.
Die erste ist die bekannte: eigene Altsoftware. Ein Assembler-Programm aus den Achtzigern, das läuft, das keiner mehr anfassen will und für das es niemanden mehr gibt, der es könnte. Da ist das Subsystem die einzige Stelle, an der man ansetzen kann.
Die zweite ist die, die man leicht übersieht und die heute mindestens so wichtig ist: **fremde Standardsoftware**. Da darf man nicht ändern, nicht weil es zu riskant wäre, sondern weil man den Quellcode nicht hat. Die Alternative ist nicht „selbst ändern“, sondern „auf den Hersteller warten“.
Zwei Beispiele, die man erzählen sollte, weil sie sofort verstanden werden.
Ein Kunde wollte, dass alle SAP-Dateien verschlüsselt werden. Gelöst mit flucFS – ohne SAP anzufassen, ohne ein Requirement bei SAP einzureichen, ohne darauf zu warten.
Und auf dem Host: bei IBM können bis zur Erfüllung eines Kundenwunsches durchaus sieben Jahre vergehen. In der Zwischenzeit bilden Kunden die gewünschte Funktionalität in der Regel transparent zur IBM-Software über die Subsysteme ab. Das ist kein Notbehelf, das ist der praktikable Weg.
Wer im Raum Standardsoftware betreut, hört an dieser Stelle auf, das Thema für ein Mainframe-Legacy-Thema zu halten.



Wer von "dem Subsystem" spricht, meint je nach Vorgeschichte zwei verschiedene Dinge – und erwartet dann vom einen, was nur das andere kann.
Deshalb stehen die Namen hier einzeln.
Von oben nach unten durchgehen, und dabei die Zeitachse betonen – zwei davon gibt es heute, zwei kommen.
FLAMSUB ist das alte, und es kann genau eine Sache: FLAMFILEs lesen und schreiben. Das ist keine Kritik, das war die Aufgabe.
FLUCSUB ist das neue und kann alles, was `CONV` und `XCNV` können – die nächste Folie führt das aus, und die Liste überrascht die meisten.
Die Formulierung ist mit Absicht so gewählt: "alles, was die Satzschnittstelle kann" hört ein Kunde als die **FLAM4**-Satzschnittstelle und denkt damit an deren Grenzen. Gemeint ist der volle Umfang der Konvertierung.
FLM5SUB für die FLAM5-Archive ist geplant und kommt in einem der nächsten Releases. Das ist wichtig für die Erwartungssteuerung: wer heute ein FLAM5-Archiv transparent über ein Subsystem ansprechen will, kann das noch nicht.
Auf der Unix-Seite dasselbe Bild: flucFS gibt es, flm5FS ist der geplante Gegenpart.
Wenn jemand nach den Namen fragt – und die Frage kommt, weil sie auf den ersten Blick unlogisch aussehen – ist die Antwort lehrreich, weil sie die Architektur erklärt.
Warum nicht FLAMSUB für das neue Subsystem? Weil FLAMSUB schon existiert, das alte FLAM4-Subsystem, und es wird auch weiter existieren. Wir lösen es nicht ab. Ein Kunde mit FLAM4-Beständen behält sein FLAMSUB und bekommt FLM5SUB daneben, wenn er FLAM5-Archive transparent ansprechen will. Deshalb braucht das neue einen eigenen Namen.
Und auf der Unix-Seite: dort wäre flamFS frei, weil es für das alte FLAM4FILE nie ein FUSE-Dateisystem gegeben hat. Wir nennen es trotzdem flm5FS – damit auf beiden Plattformen derselbe Name für dieselbe Sache steht. Der Preis ist ein Name, der isoliert betrachtet ungewohnt aussieht; der Gewinn ist, dass man nicht zwei Namensschemata im Kopf halten muss.
Beide Zeilen sind ein Ausblick. Solange sie nicht ausgeliefert sind, gehören sie als Ausblick vorgetragen und nicht als Zusage.
Wenn im Raum jemand aus dem Mittelstand sitzt, lohnt der strategische Zusatz:
flm5FS ist nicht nur der Unix-Gegenpart zu FLM5SUB, es ist die Einstiegsphase in die Cloud. Diese Kunden sitzen oft noch auf Client-Server. Über ein Dateisystem können sie ihre Daten sicher in Cloud-Storages auslagern, **ohne dass das viel Impact in der Firma hat** – kein Anwendungsumbau, ein Mount.
Deshalb steht es in unserer Cloud-Planung vor den Standard-APIs für Cloud-Anwendungen: die Reihenfolge folgt nicht der technischen Schwierigkeit, sondern der Frage, welcher Kunde als nächster einsteigen kann.
Der Satz, der die Folie zusammenhaelt: FLAM archiviert, FLUC konvertiert – und die Namen folgen dieser Trennung, nicht der Chronologie.



Diese Folie ist der eigentliche Inhalt der Einheit, und sie stand vorher nicht drin – der Foliensatz hatte FLUCSUB auf "Format und Zeichensatz wandeln" verkürzt. Das ist deutlich zu wenig.
Der Grund, warum die Liste so lang ist, ist ein architektonischer und lohnt den Satz: das Subsystem geht über die LE-lose Metal-Schnittstelle (FLCRECXB) in die Satzschnittstelle (FLCRECLB). Damit steht im Subsystem alles zur Verfügung, was auch `CONV` und `XCNV` leisten – also lesen aus dem Speicher mit format-Punkt-irgendwas, und danach beliebig viele Writes mit dem vollen Umfang der Konvertierung. Es ist keine abgespeckte Variante, sondern derselbe Motor hinter einem DD-Statement.
Die Liste einmal durchgehen, aber nicht vorlesen. Die drei, die im Raum Reaktionen auslösen, sind: Virenscan im Datenstrom, PGP, und remote lesen und schreiben per SSH.
Multi-Format-Konvertierung konkret machen: aus einem FB-Record, zu dem ein Copybook gehört, wird eine CSV-Datei – oder JSON, oder XML. Ohne die Anwendung anzufassen.
FLAMSUB dagegen kann nur FLAMFILEs lesen und schreiben. Wichtig ist dabei die Nebenbemerkung, weil sie oft Verwirrung auslöst: der FLUC **versteht** FLAM4 als Originaldatum – genauso wie ein PGP-File oder ein ZIP-Archiv. FLUCSUB kann FLAMFILEs also auch, es ist für ihn nur eine von vielen Eingaben.



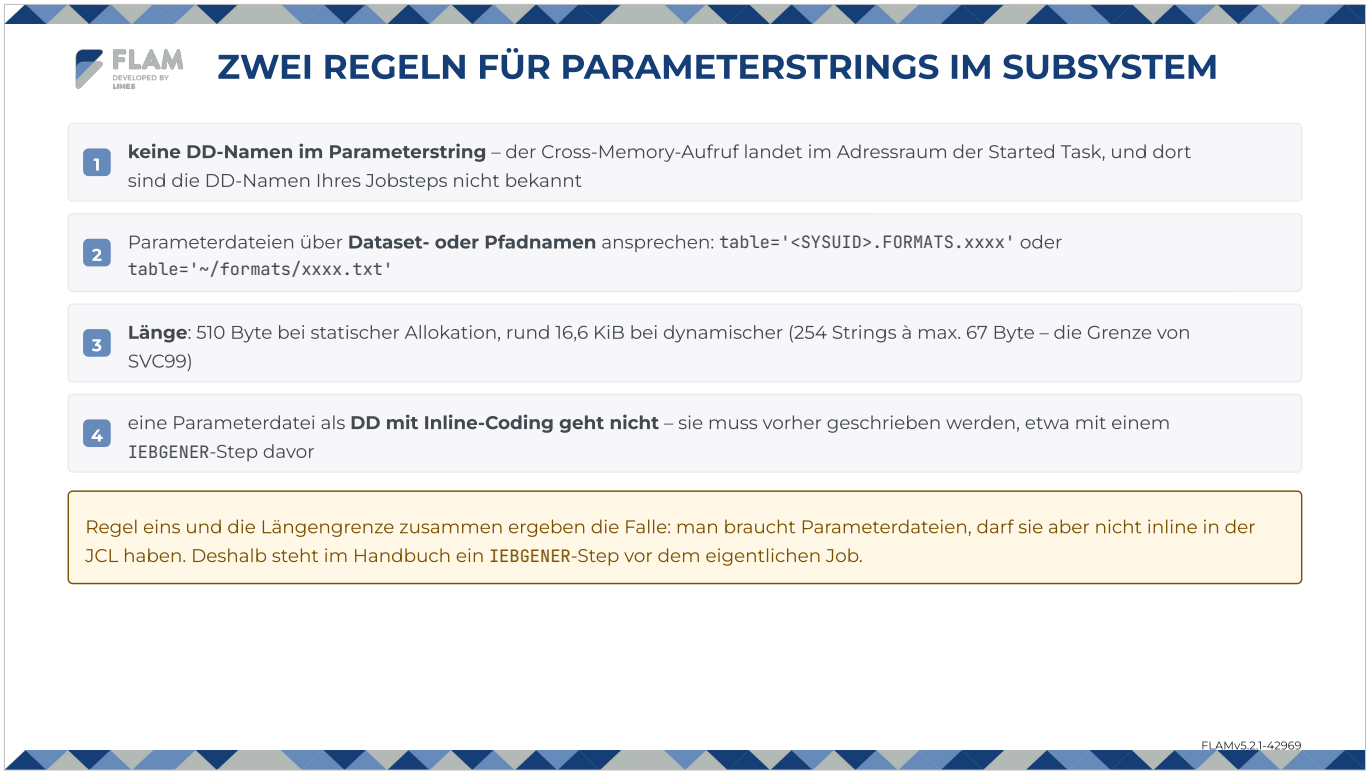
Das Beispiel, das den Nutzen von FLUCSUB in einem Bild erklärt, und es kommt aus der Praxis.
Die Ausgangslage: ein altes Dataset soll aus Sicherheitsgründen nicht mehr im Klartext liegen. Es wird durch ein klassisches FLAMFILE ersetzt – die Anwendung merkt davon nichts, weil das Subsystem davorsteht.
Und dann kommt der eigentliche Trick: im selben Aufruf zweigt man die Daten ab. Aus demselben Lesevorgang entsteht zusätzlich eine CSV-Datei in UTF-8 mit 0x0A als Trennzeichen, gepackt als GZIP, per SSH direkt beim Data Warehouse abgelegt – und als Post-Processing wird dort der Import gestartet.
Auf die Zeitachse hinweisen, das ist der Punkt: das passiert alles in einem Lesevorgang, nicht in einer Kette. Die Anwendung schreibt einmal.
Die drei Zeilen unten sind das Muster: `format.record()` sagt, wie die Anwendung die Daten liefert, danach zwei Writes – einer sichert, einer liefert.



Die Folie, auf der das Subsystem konkret wird – und sie zeigt bewusst **FLUC** zuerst, weil die Einheiten davor von FLUC sprechen.
Oben ein echter Schreibschritt aus unserer Auslieferung (`JOBLIB/FLUCSUB1.jcl`): IEBGENER schreibt nach `SYSUT2`, und weil dort `SUBSYS=(FLUC,'PARMDD=FLUCDD')` steht, geht der Satzstrom durch FLUC. Der Parameterstring liegt in einer eigenen Allokation – das ist die Regel im Subsystem, denn im DD-Statement ist für einen langen String kein Platz.
Was der Schritt tut, ist der Satz, der hängen bleiben soll: **ein unveränderter Utility schreibt eine komprimierte Datei auf einen fernen Unix-Rechner.** Zeichensatzwandlung von 1141 nach UTF-8, Zeilenenden nach Unix-Art, Trailing Blanks weg, gzip, und die Übertragung per SSH – alles im DD-Statement, kein Programm geändert.
Zur URL die Regel mitnennen, weil sie hier sichtbar ist: **ein Schrägstrich hinter dem Rechner heißt relativ zum Heimatverzeichnis**, zwei Schrägstriche heißen absolut.
Unten dasselbe Muster für **FLAM4** – die Syntax wörtlich aus `doc/man_flam4_appendix_flamsub.txt`, Abschnitt "Encryption using FKMEFILE". Nicht improvisieren: die doppelten Hochkommata sind der Teil, den alle falsch machen.
Die drei Voraussetzungen deutlich nennen: Lademodule erreichbar, Started Task läuft, Lizenz vorhanden. Fehlt eine davon, ist der Fehler nicht selbsterklärend.
Und die Eigenschaft, die im Betrieb Ärger erspart: **ungültige Parameter weist FLAM als JCL-Fehler ab**, nicht erst zur Laufzeit – der Job kommt also nicht erst nach zwei Stunden Rechenzeit zurück.
Das Semikolon am Zeilenende ist das Kommentarzeichen der Kommandosprache.
Mit `NUMBER=ON` im Editor werden die Zeilennummern dahinter zum Kommentar – deshalb steht es in allen unseren Beispielen.



Das Gegenstück außerhalb von z/OS, und es ist erfreulich einfach: ein Verzeichnis wird an einer anderen Stelle eingehängt, und alles, was dorthin geschrieben wird, wird transparent konvertiert und verschlüsselt.
Das Beispiel wörtlich aus dem Handbuch. Vorführen, wenn ein Linux da ist – der Effekt „ich schreibe mit cat und es ist verschlüsselt“ überzeugt sofort.
Abhängigkeit nennen: die FUSE-Bibliothek im Userspace muss vorhanden sein.
Der eigentliche Reiz sind die zwei Sichten auf dieselben Daten. Im physischen Verzeichnis liegen **echte PGP-Dateien** – die kann der Kunde mit seinen gewohnten Mitteln verwalten, sichern, auf Band schreiben, weitergeben. Auf der anderen Seite des Mount-Points sieht die Anwendung die klaren Daten. Kein Sonderformat, das niemand außer uns lesen kann.
Deshalb steht die Lösung oft auf zentralen Dateiservern mit **Net-HSM**: der Schlüssel liegt im HSM, die Dateien liegen als PGP im Dateisystem, und wer den Mount nicht hat, sieht nichts.
Und die Zeitachse: für die FLAM5-Archive ist **flm5FS** der geplante Gegenpart, den gibt es noch nicht.



Diese Folie fehlte, und sie ist der Grund für einen guten Teil der Supportfälle beim Subsystem. Beide Punkte stehen im Handbuch unter "Restrictions", aber niemand liest Anhänge.
Regel eins, keine DD-Namen im Parameterstring. Der Grund erklärt sich aus der Architektur: das Subsystem nutzt die LE-lose Schnittstelle von FLAMSTC, und ein Cross-Memory-Aufruf gibt die Konvertierung an einen Server weiter, dessen Adressraum die Started Task kontrolliert. In diesem Adressraum sind die DD-Namen des Jobsteps schlicht nicht bekannt. Es funktionieren nur solche DD-Namen, die statisch über FLAMSTC alloziert wurden – deshalb ist es sinnlos, im Subsystem mit DD-Namen zu arbeiten.
Die Konsequenz für die Praxis: Parameterdateien über Datasetnamen oder Pfadnamen ansprechen, nicht über DD.
Regel zwei, die Länge. Bei statischer Allokation sind es 510 Byte. Bei dynamischer Allokation mit FLAM5-Komponenten sind es rund 16,6 KiB, und die Zahl kommt so zustande: 254 Parameterstrings mit je maximal 67 Byte, das ist die Grenze von SVC99.
Daraus folgt, dass Parameterdateien im Subsystem noch wichtiger sind als im Batch. Und hier die Falle, die beides verbindet: eine Parameterdatei als DD mit Inline-Coding geht wegen Regel eins nicht. Man muss die Datei vorher schreiben – typischerweise mit einem IEBGENER-Step davor, so steht das Beispiel auch im Handbuch.
Wer diese drei Sätze kennt, spart sich die zwei häufigsten Tickets.

WAS AUF DIESEM WEG NICHT GEHT

- das FLUC-Subsystem arbeitet auf **Streams**, also rein sequenziellem I/O – **Ganzdateiverarbeitung**, keine Änderung einzelner Teile
- der Weg braucht **einen laufenden Dienst**: Started Task auf z/OS, FUSE-Mount auf Linux – also Betrieb, nicht nur eine Bibliothek
- es ist **kein** Weg, um im Archiv zu suchen oder einzelne Sätze über einen Schlüssel zu finden – das ist die Satzschnittstelle
- die Anwendung kann nicht steuern, wie verschlüsselt wird – das entscheidet die Allokation

Nur für FLAM4FILEs: dort hebt ein verändernder Zugriff den Integritätsschutz über die ganze Datei auf. Für **FLAM5-Archive gilt das nicht mehr** – dort bleiben Integrität und Vollständigkeit bei vollem Zugriff gewahrt (siehe Einheit *Das FLAM5-Archiv*). Ein Subsystem dafür (**FLM5SUB**) ist geplant.

Diese Folie ist die wichtigste der Einheit, weil sie Projekte vor einer falschen Annahme bewahrt. Zwei Dinge sind dabei sauber zu trennen, weil sie oft in einen Topf geworfen werden.
Erstens: die Aussage zum Integritätsschutz gilt nur für die **alten FLAM4FILEs**. Dort macht ein verändernder Zugriff den MAC über das Ganze kaputt, weil er nicht nachgerechnet werden kann. Für die neuen **FLAM5-Archive gilt das nicht mehr** – dort bleiben Integrität und Vollständigkeit bei vollem Zugriff gewahrt. Wer also aus dieser Folie mitnimmt "FLAM kann kein Update mit Integritätsschutz", hat das Gegenteil gelernt. Der Verweis geht auf die Einheit zum FLAM5-Archiv.
Zweitens die eigentliche Grenze des FLUC-Subsystems, und die ist eine andere als der MAC: das Subsystem arbeitet auf **Streams**, also rein sequenziellem I/O. Das heißt Ganzdateiverarbeitung – man kann keine Teile ändern. Das ist kein Krypto-Thema, sondern die Natur der Subsystem-Schnittstelle.
Drittens, organisatorisch und oft unterschätzt: der Weg braucht einen laufenden Dienst, also Betrieb – nicht nur eine Bibliothek.
Und der Ausblick, damit niemand die Grenze für endgültig hält: FLM5SUB für die FLAM5-Archive ist geplant.

STOLPERFALLEN

! **Ohne Started Task kein Subsystem.**
Das `SUBSYS=` scheitert, wenn der Dienst nicht läuft – und der Fehler klingt nicht danach.

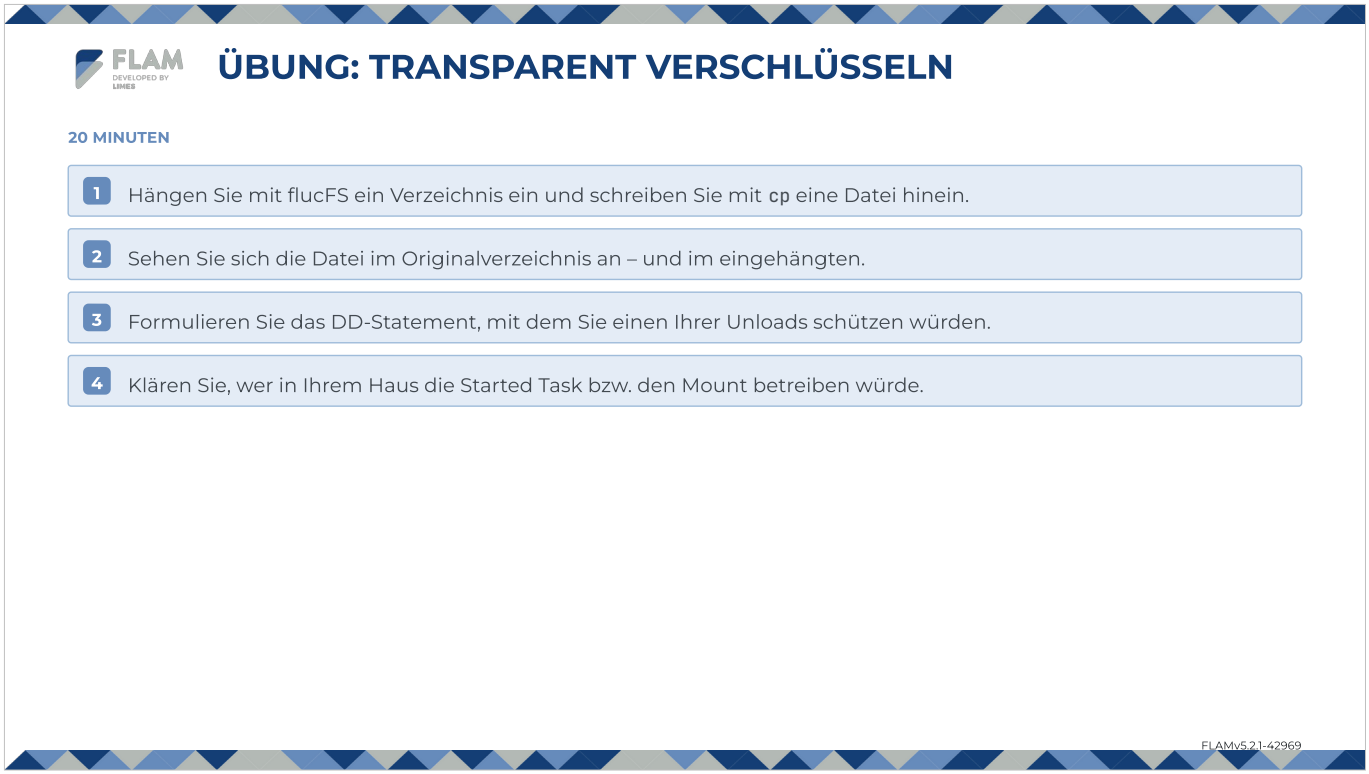
! **Eine FLAMFILE ohne `SUBSYS=` gelesen liefert die komprimierten Bytes.**
Das ist keine Korruption, das ist eine fehlende Allokation.

! **Fehlende DCB-Angaben treffen Utilities.**
IEBGENER, SORT und eigene Assembler-Routinen brauchen sie gelegentlich im DD-Statement; ohne sie liefert das Subsystem sie bei `DISP=SHR` selbst.

! **Der Job-User braucht Zugriff auf den Schlüssel.**
Bei ICSF heißt das Label und Service. Sonst scheitert der Job beim Open, nicht beim Schreiben.

FLAMv5.21-42969

Alle vier aus Supportfällen. Die erste kostet die meisten Tickets, die zweite die meiste Zeit.



Zwanzig Minuten. Auf Linux mit flucFS, auf z/OS am Beispiel eines DD-Statements auf dem Papier, wenn kein System zur Verfügung steht.
Aufgabe 4 ist die wichtige: sie klärt, wer im eigenen Haus den Dienst betreiben würde.
Auflösung:
1. Das `SUBSYS=(FLAM,...)` im DD-Statement ist die ganze Änderung. Die Anwendung wird nicht angefasst und nicht neu gebunden.
2. Ohne laufende Started Task scheitert das `SUBSYS=` – und die Meldung klingt nicht nach "Dienst fehlt". Das ist die häufigste Ursache für Tickets zu diesem Thema.
3. Liest man die Datei ohne `SUBSYS=`, bekommt man die komprimierten Bytes.
Das ist keine Korruption, sondern eine fehlende Allokation – der Lernpunkt der Aufgabe.
4. Die Anwendung kann nicht steuern, wie verschlüsselt wird; das entscheidet die Allokation. Und bei einem **FLAM4FILE** hebt wahlfreies Ändern den Integritätsschutz über die ganze Datei auf – bei einem FLAM5-Archiv nicht, dort bleiben Integrität und Vollständigkeit gewahrt. Wer beides braucht, nimmt also die Satzschnittstelle und ein FLAM5-Archiv.
Und die zwei Regeln nicht vergessen: keine DD-Namen im Parameterstring, und die Längengrenze von 510 Byte bei statischer Allokation.

WAS BLEIBT

- die Anwendung wird nicht angefasst und bemerkt keinen Unterschied
- **FLAMSUB** (alt) kann nur FLAMFILEs – **FLUCSUB** (neu) kann alles, was `CONV` und `XCNV` können
- auf Unix **flucFS** über FUSE, mit echten PGP-Dateien im physischen Verzeichnis
- das Subsystem arbeitet auf **Streams**: Ganzdateiverarbeitung, keine Änderung einzelner Teile
- der Weg braucht einen laufenden Dienst – Betrieb, nicht nur eine Bibliothek

Für die **FLAM5-Archive** sind **FLM5SUB** und **flm5FS** geplant. Heute gilt der transparente Weg für FLAMFILEs und für jede Konvertierung.

FLAMv5.21-42969

Lernziele durchgehen. Die vier Namen noch einmal nennen, das ist der Punkt, den man aus dieser Einheit falsch mitnehmen kann.
Danach die dritte Einheit: dasselbe im Programm, mit Zugriff auf einzelne Sätze.

FLAM SCHULUNG, EBENE 1

Satzweiser Zugriff aus dem Programm heraus

Byte, Satz, Element, Unterprogramm – plus C++ und Java

Falk Reichbott · limes datentechnik gmbh

Dritter Zugangsweg, und der einzige, bei dem der Anwendungsquellcode angefasst wird. Der Gegenwert ist der direkte Zugriff auf einzelne Sätze aus der Anwendung heraus.
Wichtig zur Abgrenzung, weil die Verkürzung naheliegt: das API ist **nicht** der einzige Weg zum Vollzugriff. `flcl archive` kann im Batch suchen und ändern, und `FLAMSUB` gibt den Vollzugriff auch transparent, bei FLAM4 allerdings nur bei KSDS-Abbildung. Was das API auszeichnet, ist der Zugriff **im Programm** – die Anwendung entscheidet je Satz, was passiert. Der Überblick dazu steht in der Einheit *Die drei Zugangswege*.
Der Bogen: vier APIs, ein Auswahlkriterium, dann jedes einzeln – und am Ende die Fehlerbehandlung, weil die in der Praxis den Unterschied macht.



Ziel eins ist das Entscheidende: die Wahl des richtigen APIs. Wer sie falsch trifft, baut Wochen an der falschen Stelle.



Hier ist die Ordnung wichtig, weil die Folie sonst mehr verwirrt als erklärt. Zwei Achsen, und man muss beide im Kopf haben.
Erste Achse: die Einheit, in der Ihr Programm denkt – ein Byte-Strom, ein Satz oder ein Element. Alle drei arbeiten auf einer **geöffneten Datei**: öffnen, lesen oder schreiben, schließen. Das ist der Charakter einer I/O-Schnittstelle.
Zweite Achse, und die fehlt in den meisten Erklärungen: FLUC oder FLAM. Bei der Satz- und der Elementschnittstelle macht das den entscheidenden Unterschied:
Über **FLUC** geht nur **sequenzielles** Lesen oder Schreiben. FLUC verarbeitet Originaldaten, und ein Strom hat keine Position, an die man springt.
Über **FLAM** geht der **Vollzugriff** – suchen, ändern, einfügen, löschen.
Denn dort liegt ein Archiv mit Verzeichnis und Index darunter.
Zur Byteschnittstelle die ehrliche Auskunft, weil sie im Raum gefragt wird:
es gibt sie derzeit nur auf FLUC-Basis, also für Ströme aus Originaldaten, und damit nur sequenziell. Ob wir eine Byteschnittstelle mit Vollzugriff, also mit Seek, auch für FLAM anbieten, ist **noch nicht entschieden**. Für ODATA könnte sie nötig werden. Machbar ist es, trivial nicht.
Und der Gedanke, der dahinter steht und den man mitgeben sollte:
Vollzugriff ist mit Records oder einzelnen Elementen sinnvoll, mit Bytes kaum. Ein Byte-Offset in komprimierten und verschlüsselten Daten hat mit Zeilen einer Tabelle und mit Indizierung wenig zu tun.
Zur Elementschnittstelle gehört ein Satz, der ihren Zweck erklärt, denn sonst wirkt sie wie eine dritte Variante ohne Grund: **sie gibt es für FLUC und für FLAM, und sie arbeitet auf den neutralisierten Elementen** – also auf den Daten, nachdem FLAM sie aus ihrem Format gelöst hat, und bevor sie in ein Zielformat gegossen werden.
Der Nutzen wird an einem Fall deutlich, den jeder kennt, der Daten umlagert:
eine Originaldatei mit FLUC lesen und in ein FLAM-Archiv schreiben. Ohne die Elementschnittstelle würde aus den neutralen Elementen erst ein Copybook-Record gebaut, und aus diesem Record danach wieder neutrale Elemente – eine Wandlung in den Speicher und aus dem Speicher zurück, für nichts. Mit ihr werden die **Elemente durchgereicht**: vom Lesen der Originaldatei direkt in das Archiv.
Das ist keine Bequemlichkeit, das ist ein eingesparter Arbeitsgang pro Satz.
Bei Massendaten ist das der Unterschied, den man in der CPU-Zeit sieht.
Was auf dieser Folie **nicht** steht und wohin es gehört: die **Unterprogramme**. Sie sind keine I/O-Schnittstelle, sondern die Möglichkeit, **jedes Kommando als Funktion** zu rufen – also Ganzdateiverarbeitung, inklusive Directory-Walk. Das ist ein ganz anderer Charakter und steht deshalb bei der Einheit *Batch*. Zwei Folien weiter kommen sie noch einmal vor, damit sie als Programmierschnittstelle aufgeführt sind.



Der Einstieg für alles, was nicht Mainframe ist. Wer `fopen` und `fgets` kennt, kennt das Muster – der Unterschied steckt im Öffnen.
Und hier ist der Punkt, der die ganze Einheit trägt und den man beim ersten Lesen des Handbuchs überliest: **`fcbopen` nimmt zwei Strings**, nicht einen.
Der erste ist der Datei- oder Konvertierungsstring – genau der aus dem FLCL-Handbuch und aus dem FLCC, also `read....` bzw. `write....` oder `input(...)`/`output(...)` von XCNV. Er beschreibt, was mit der Datei passiert.
Der zweite ist der **Formatstring**. Er beschreibt, in welcher Form Ihr Programm die Daten haben will: `format.text()`, `format.record()`, `format.binary()`, `format.table()`, `format.element()`. Das ist die Schnittstelle zu Ihrem Speicher, und die gibt es bei flcl nicht – dort ist das Ziel eine Datei, hier ist es Ihre Variable.
Die zwei Fragen also getrennt: was passiert mit der Datei, und wie will ich die Daten sehen.
Und hier die Verwechslung, die beim Lesen der Funktionsliste fast zwangsläufig passiert, weil die Namen so ähnlich sind: `fcbopenv` ist **nicht** eine Variante von `fcbopen`. Es öffnet einen **Element-Konverter** auf einem bereits offenen Handle, und es ist nur bei `format.element()` sinnvoll. Man kann mehrere gleichzeitig offen haben, und die Lese- und Schreibfunktionen mit `v` am Ende nehmen so ein Konverter-Handle zusätzlich mit. Damit wandelt man je Element – etwa eine Zahl aus einem XML in eine gepackte BCD-Darstellung.
Jetzt die Einordnung, die erklärt, warum es genau zwei Strings sind und nicht einer – die gilt für **alle** Schnittstellen und lohnt einmal ausführlich:
Die **Lese- und Schreibstrings für File-I/O entsprechen eins zu eins den Strings der Utilities.** Deshalb ist das FLCL-Handbuch auch das API-Handbuch, und deshalb baut die FLCC sie mit.
Anders ist es beim Lesen **aus** dem Speicher und Schreiben **in** den Speicher: dort wird nur der sinnvolle Teil des FLUC angeboten. Genau dafür gibt es das Schlüsselwort `format`. Und das ist unabhängig davon, ob am anderen Ende ein FLAM-Archiv oder Originaldaten liegen.
Das Beispiel oben ist die **FLUC-Byteschnittstelle**. Es liest aus einer Datei – das ist die Originaldatenverarbeitung des FLUC – und schreibt über den Formatstring in den Speicher. Auch das ist FLUC: ein `WRITE.TEXT()` mit FIODAT, also nach Memory.
Und so hängt es allgemein zusammen, das ist der Satz für das Whiteboard:
FLUC liest das Original und erzeugt daraus eine neutrale Matrix im Speicher. Aus dieser Matrix schreibt entweder wieder FLUC – in eine Datei oder in den Speicher – oder FLAM schreibt sie als komprimierte und verschlüsselte Segmente weg. Rückwärts genauso: FLAM liest die Segmente und macht die neutrale Matrix, und das Originaldaten-Schreiben des FLUC wandelt diese neutrale Elementliste in das Format, das im Speicher gebraucht wird – Copybook, C-Struktur, JSON.
Wer das einmal verstanden hat, versteht alle Schnittstellen gleichzeitig:
es ist immer dieselbe Mitte. Was sich unterscheidet, ist nur, wer die Matrix füllt und wer sie leert.
Der Satz zum Schluss: die Formatstrings konfigurieren auch das FLUC-Subsystem auf z/OS und flucFS auf Unix. Das ist die Brücke zur vorigen Einheit.

DER FORMATSTRING: DIE SICHT IHRES PROGRAMMS

FORMATSTRING	WAS IHR PROGRAMM BEKOMMT BZW. LIEFERT
`format.binary()`	Bytes, unverändert
`format.char()`	Zeichen, mit Zeichensatzwandlung
`format.text()`	Zeilen ohne Trennzeichen
`format.record()`	Sätze – der Fall für COBOL und PL/I
`format.xml()`	XML-Ereignisse
`format.table()`	Zeilen und Spalten nach Row-Spec
`format.element()`	einzelne Elemente in der **neutralen FLAM5-Darstellung** – die Grundlage für Element-Konverter

JE ELEMENT WANDELN – ZEICHENKETTE AUS XML NACH GEPACKT DEZIMAL

```
void* f = fcbopen("read.xml(file='/daten/ein.xml')",
                  "format.element()");
void* c = fcbopenv(f, "conv.float(from(format.str(chrset=UTF-8))"
                  " to(format.bcd(SIGNED TYPE=ZONED"
                  " INTDIGITS=8 FRACDIGITS=2)))");
```

FLAMv5.21-42969

Die Nachschlagefolie für Entwickler, und die Folie, die die Wahl der Schnittstelle innerhalb des Byte-Interfaces entscheidet.
Die sieben Formate einmal durchgehen, aber nicht alle gleich tief. Die drei oberen sind der Alltag, `TABLE` und `ELEMENT` sind die interessanten.
`format.element()` ist der Punkt, an dem das Byte-Interface über eine gewöhnliche Datei-API hinausgeht: die Anwendung bekommt einzelne Elemente in der neutralen FLAM5-Darstellung, und mit einem Element-Konverter wandelt man je Element in die Darstellung, die die Anwendung braucht.
Das Beispiel unten ist echt und aus Anhang A: eine Zahl, die als Zeichenkette in einem XML steht, wird beim Lesen in eine gepackte Dezimalzahl mit acht Vorkomma- und zwei Nachkommastellen gewandelt. Genau das schreibt sonst jeder selbst – und macht dabei Fehler.
Der State-String am Ende: beim Lesen gibt der Open zusätzliche Metadaten zurück, beim Schreiben kann man sie hineingeben. Damit überträgt man den Dateizustand von einer Seite auf die andere, ohne ihn selbst zu beschreiben.



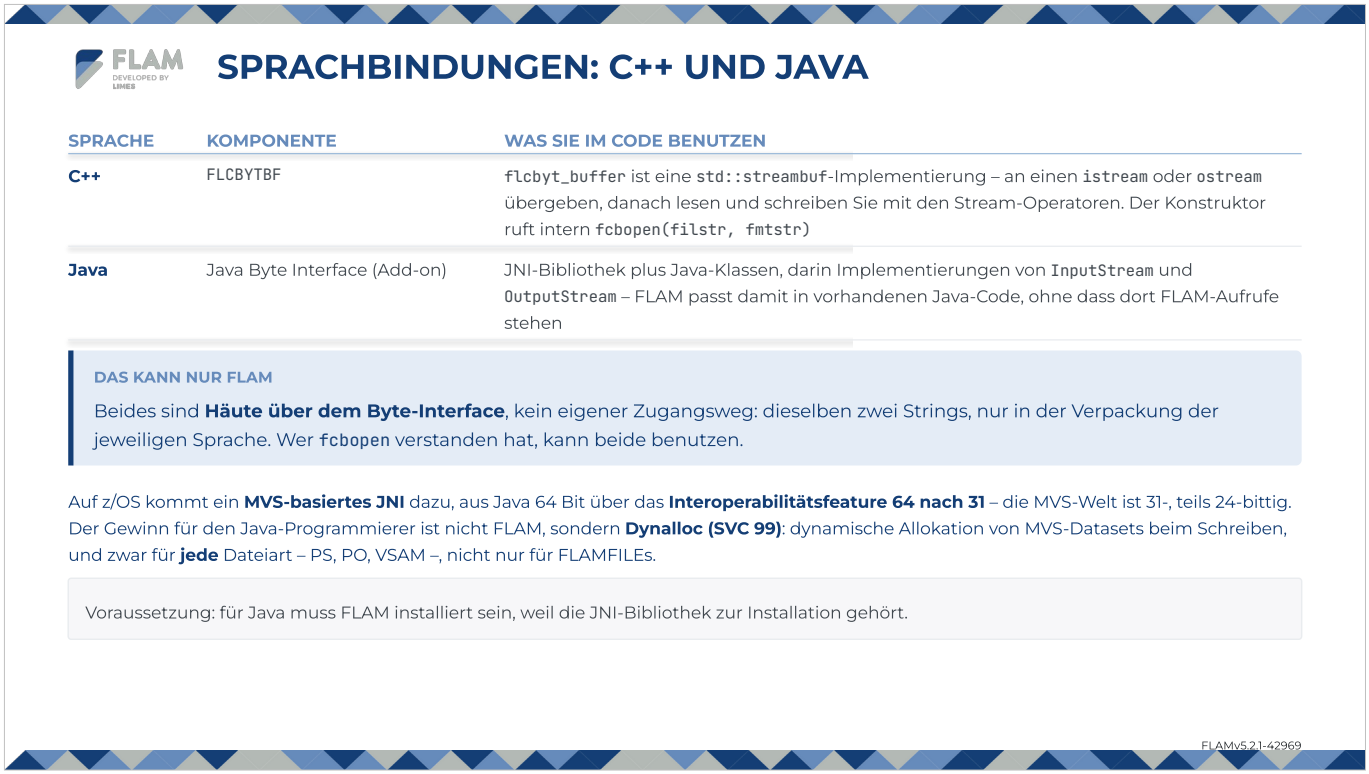
Der Kern von FLAM auf dem Mainframe. Wenige Eintrittspunkte, und das ist die ganze Schnittstelle – das überrascht, weil man von einem Zugriffsverfahren etwas Großes erwartet.
Und jetzt die Ordnung, die auf dieser Folie zählt: es gibt die Satzschnittstelle **dreimal**, für drei Ziele, und die Namen sagen, welche gemeint ist:
`**FLMxxx**` ist die alte Satzschnittstelle für die **FLAM4-Archive**. Sie hat den vollen Umfang: `FLMOPN`, `FLMCLS`, `FLMGET`, `FLMPUT`, `FLMPOS`, `FLMUPD`, `FLMDEL`.
`**FMRxxx**` ist die neue Satzschnittstelle für die **FLAM5-Archive** – FMR steht für FLAM-Record-Interface. `FMROPN`, `FMRCLS`, `FMRGET`, `FMRLOC`, `FMRPUT` und weitere. Für FLAM5 gilt im Prinzip dasselbe wie für FLAM4, nur dass hier zusätzlich eine **DLL mit einem C-Interface** die Basis stellt – das macht sie auch außerhalb der klassischen Hostsprachen benutzbar.
`**FCRxxx**` ist die Satzschnittstelle des **FLUC**, also für Originaldaten.
Und hier ist der Unterschied, der zur vorigen Folie passt: sie hat `FCROPN`, `FCRCLS`, `FCRGET`, `FCRPUT` und `FCRLOC` – aber **kein** `FND`, `INS` oder `UPD`. Weil FLUC einen Strom verarbeitet und ein Strom keinen Vollzugriff kennt.
Der Satz, der die Folie zusammenhält und den Teilnehmern die Angst nimmt:
wer eine dieser Schnittstellen kennt, kennt alle drei. Das Muster ist dasselbe – öffnen, lesen oder schreiben, positionieren, schließen. Dasselbe gilt für die Byte- und die Elementschnittstelle. Man lernt das Muster einmal.
Die Aufrufe sehen absichtlich wie File-I/O in COBOL, PL/I oder Assembler aus. Die Änderung im Programm bleibt klein – oft ist es ein Suchen-und-Ersetzen der I/O-Verben.
Sicherheitsargument nicht vergessen: nur das benötigte Segment liegt klar im Speicher, es entsteht keine temporäre Klartextdatei.

UNTERPROGRAMM: GANZE DATEIEN

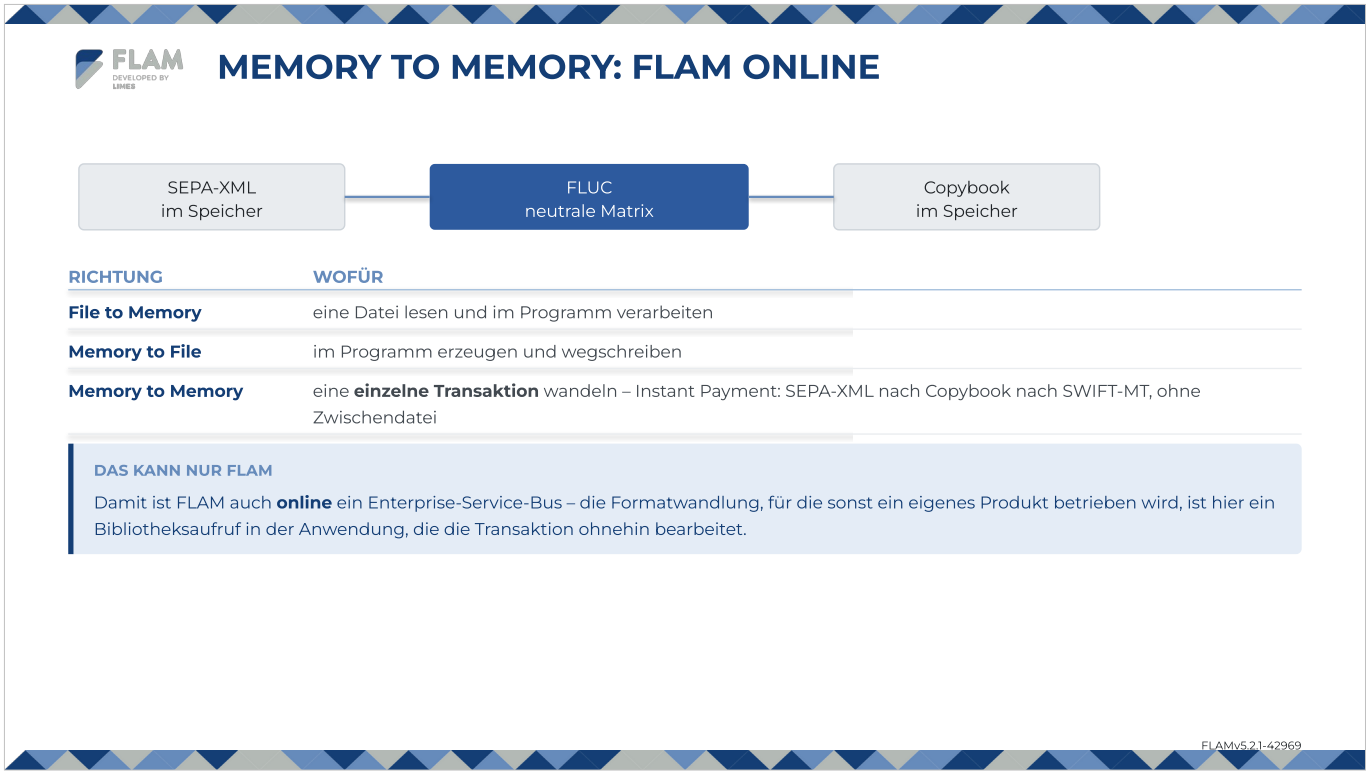
- plattformneutrales Lademodul: **FLUC-UP** für die Konvertierung, **FLAM-UP** für das Archiv
- statisch bindbar oder als Shared Library – aus jeder Sprache aufrufbar
- es ist dasselbe Modul, das auch das Utility benutzt: **was flcl kann, kann Ihr Programm auch**
- optionale **Exits** zwischen physischem I/O und Datenübergabe – der Hebel für eigene Protokolle und eigene I/O
- ein Releasewechsel erzwingt kein erneutes Binden der Schlüsselverwaltung
- es ist **keine I/O-Schnittstelle**: es ruft ein Kommando als Funktion, mit Directory-Walk – siehe Einheit *Batch*

FLAMv5.21-42969

Die Antwort auf „wir haben ein eigenes Werkzeug und wollen FLAM darin benutzen“.
Wichtig: das Unterprogramm ist dasselbe Modul, das auch das Utility benutzt.
Was flcl kann, kann Ihr Programm damit auch – es gibt keine zweite, schwächere Schnittstelle.
Die Exits sind der Hebel für eigene Protokollierung und eigene I/O; darauf hinweisen, wenn im Raum Hausstandards für Audit-Trails gelten.
Zur Einordnung, damit die Folie nicht neben den anderen steht: das Unterprogramm ist **keine I/O-Schnittstelle**. Es arbeitet nicht auf einer geöffneten Datei, sondern ruft **ein Kommando als Funktion** – mit allem, was dazugehört, einschließlich Directory-Walk. Es ist also Ganzdateiverarbeitung, und deshalb gehört es eigentlich zur Einheit *Batch*, wo FLCL, FLCC und die Lademodule mit Kommandofunktionen zusammen behandelt werden.
Es steht hier trotzdem, und zwar aus einem Grund: **es ist eine Programmierschnittstelle**, und wer diese Einheit hört, soll wissen, dass es sie gibt. Wer die Utilities in einer Anwendung integrieren will, braucht genau das.



Diese Folie stand vorher nur auf Java. Das C++-Interface fehlte ganz – es existiert als eigene Komponente `FLCBYTBF` mit eigener Doxygen-Dokumentation, und es ist der bequemere Weg für jeden, der C++ schreibt.
Die zwei Namen kurz auseinanderhalten, sonst fragt jemand nach: `FLCBYTBF` ist der Header `CPPINC/FLCBYTBF.hpp` und damit der Komponentenname in unserer 8-Zeichen-Konvention; `flcbyt_buffer` ist die Klasse darin. Ein Ding, zwei Ebenen.
C++: `flcbyt_buffer` ist eine `std::streambuf`-Implementierung. Man übergibt sie einem `istream` oder `ostream`, und danach liest und schreibt der Code ganz normal mit den Stream-Operatoren. Der Konstruktor ruft intern `fcbopen(filstr, fmtstr)` – also dieselben zwei Strings von der vorigen Folie. Nichts Neues zu lernen, nur eine andere Verpackung.
Java: dasselbe Muster, aber über das JNI. Add-on mit JNI-Bibliothek plus Java-Klassen, darin Implementierungen von `InputStream` und `OutputStream`.
Der gemeinsame Punkt, der die Folie zusammenhält: **beides sind Häute über dem Byte-Interface**, kein eigener Zugangsweg. Wer die zwei Strings verstanden hat, kann beide benutzen.
Die Voraussetzung nennen, weil sie die häufigste Rückfrage ist: für Java muss FLAM installiert sein, weil die JNI-Bibliothek Teil der Installation ist.
Und dann der Teil, der auf dem Host aus einer Sprachbindung ein Argument macht – er sollte nicht als Fußnote durchgehen:
Auf z/OS gibt es ein **MVS-basiertes JNI**, und aus einer 64-Bit-JVM heraus wird es über das **Interoperabilitätsfeature 64 nach 31** erreicht. Der Grund ist schlicht: die MVS-Welt ist 31-bittig, an manchen Stellen sogar 24.
Der Gewinn ist aber nicht die Anbindung selbst, sondern was damit erreichbar wird: der Java-Programmierer hat **Dynalloc, also SVC 99**, vollständig zur Verfügung – die dynamische Allokation von MVS-Datasets beim Schreiben. Das ist endlich volle Kontrolle über die MVS-Seite, und zwar **nicht nur für FLAMFILEs**: für jede Dateiart, die ein Java-Programm auf dem Host schreiben will, also auch PS, PO und VSAM.
Wer auf dem Host Java schreibt, kennt das Problem – und hat es bisher mit JCL-Vorarbeit oder Hilfsprozessen gelöst. Diesen Punkt sollte man im Raum stehen lassen.



Die Folie, die diese Einheit über die Batchwelt hinaushebt – und die in den Unterlagen bisher ganz fehlte.
Bei der Byte-, der Satz- und der Elementschnittstelle des FLUC kann man nicht nur **File to Memory** und **Memory to File** arbeiten, sondern auch **Memory to Memory**.
Und daraus folgt etwas, das man erst beim zweiten Hinsehen erkennt: **alle Konvertierungen des FLUC lassen sich auf einzelne Transaktionen im Speicher anwenden.** Nicht auf eine Datei mit Millionen Sätzen, sondern auf einen einzelnen Vorgang – online, im Antwortzeitfenster.
Das Beispiel, das im Zahlungsverkehr sofort sitzt: **Instant Payment.** Eine Transaktion kommt als SEPA-XML herein, muss in ein Copybook-Format für die Hostanwendung, und geht als SWIFT-MT wieder hinaus. Drei Formate, eine Transaktion, alles im Speicher, ohne eine einzige Zwischendatei.
Das ist genau das, was ein Enterprise-Service-Bus tut. Der Unterschied: ein ESB ist ein zusätzliches System mit eigener Laufzeit, eigenem Betrieb und eigener Lizenz. Hier ist es ein Bibliotheksaufruf in der Anwendung, die die Transaktion ohnehin bearbeitet.
Und der Bogen zur Grundlagen-Einheit: dort hieß es, FLAM5 sei ein Enterprise-Service-Bus für **persistente** Datenbestände. Mit Memory to Memory gilt das auch **online** – für Nachrichten im Fluss. Das ist der vollständige Satz.
Wenn im Raum jemand aus der Architektur sitzt, ist das der Punkt, an dem er zuhört: die Formatwandlung, für die er ein eigenes Produkt betreibt, ist hier eine Funktion des Zugriffsverfahrens.

FEHLER AUSWERTEN

EBENE	WOHER	WAS SIE SAGT
Returncode	Rückgabewert des Aufrufs	die Kategorie: fatal, System, Handle, Schnittstelle, nicht gefunden
Fehlernummer	`fcberrno` bzw. der Fehlerblock	die konkrete Ursache
Fehlertext	`fcberror`	die Erklärung, so wie sie auch das Utility ausgibt

Im Fehlerfall alle drei zusammen protokollieren. Genau das braucht auch ein Ticket – und es erspart die erste Rückfragerunde.

FLAMv5.21-42969

Der Teil, der in der Praxis den Unterschied macht und in Beispielen meistens fehlt.
Drei Ebenen: der Returncode sagt die Kategorie, die Fehlernummer die konkrete Ursache, der Fehlertext die Erklärung. Wer nur den Returncode loggt, hat später nichts in der Hand.
Empfehlung, die man aufschreiben sollte: im Fehlerfall immer Returncode, Fehlernummer und Fehlertext gemeinsam protokollieren – genau das braucht auch ein Ticket.

STOLPERFALLEN

! **Unterprogramm und Satzschnittstelle werden verwechselt.**
Das Unterprogramm nimmt ganze Dateien, die Satzschnittstelle einzelne Sätze. Falsche Wahl kostet Wochen.

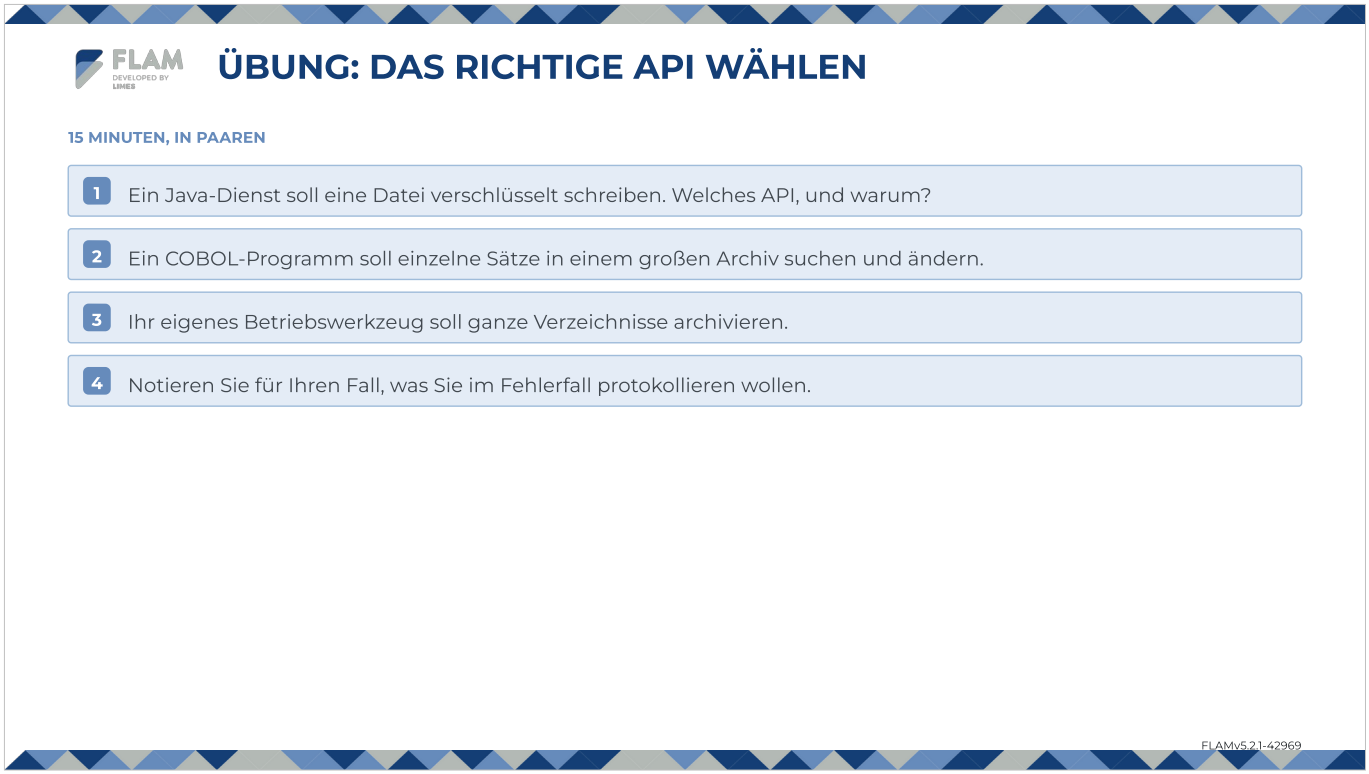
! **Der Parameterstring wird im Programm zusammengesetzt.**
Dann entstehen Quoting-Fehler zur Laufzeit. Im FLCC bauen, als Konstante ablegen, nur Werte einsetzen.

! **Nur der Returncode wird protokolliert.**
Ohne Fehlernummer und Fehlertext ist die Ursache später nicht mehr feststellbar – auch nicht von uns.

! **Java ohne FLAM-Installation.**
Die JNI-Bibliothek ist Teil der Installation. Ohne sie fällt der Fehler erst beim Laden der Klasse auf.

FLAMv5.21-42969

Alle vier aus Supportfällen der API-Nutzung.



Fünfzehn Minuten, in Paaren. Ziel ist die Begründung, nicht die Antwort – und die Begründung ist immer die Einheit, in der das Programm denkt.



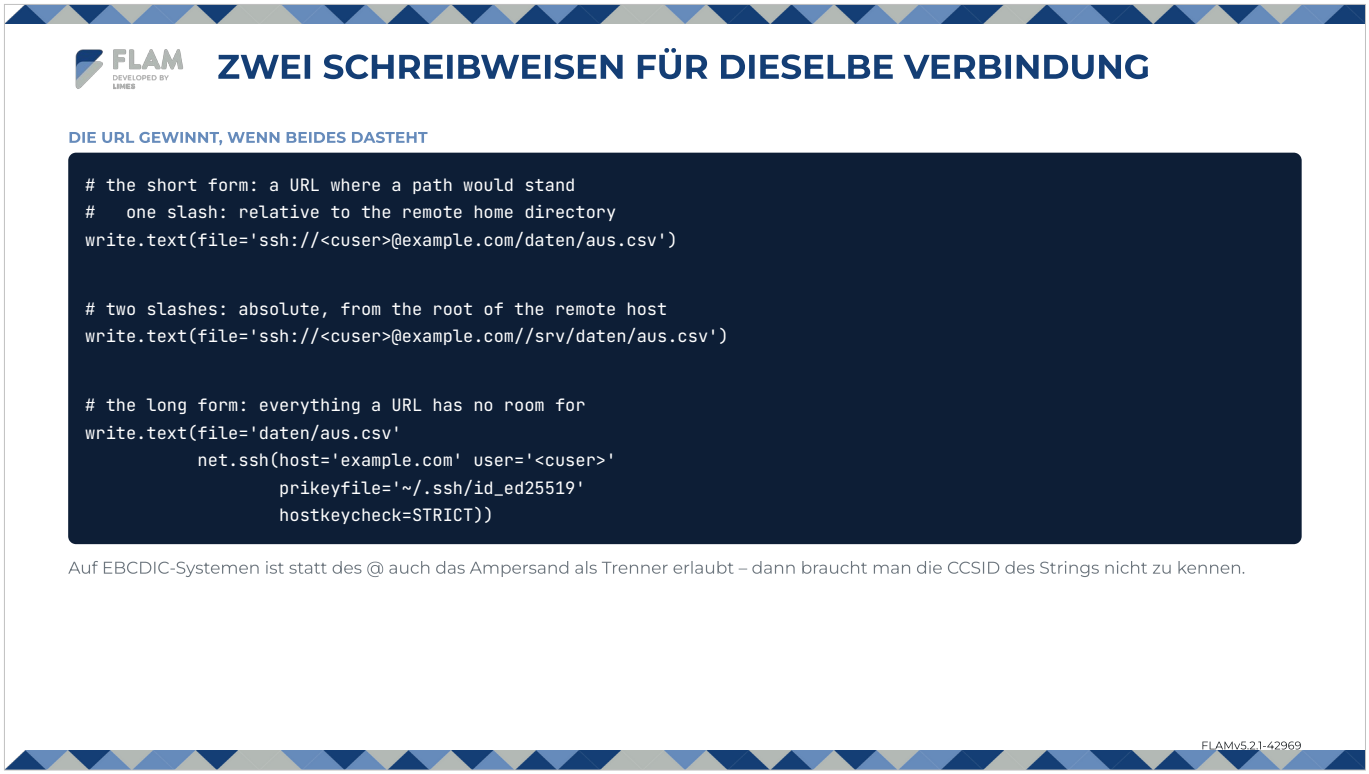
Lernziele durchgehen. Damit ist Ebene 1 abgeschlossen: alle drei Wege sind einmal am selben Parameterstring gezeigt.
Die Modulfamilien noch einmal nebeneinander, weil die Namen im Alltag begegnen: `**FMRxxx**` sind die Lademodule für die FLAM5-Archive, `**FLMxxx**` die für die FLAM4-Archive, und `**FCRxxx**` die für die FLUC-Satzschnittstelle – letztere nur mit `PUT`, `GET` und `LOC`, ohne `FND`, `INS` und `UPD`.
Es folgen die Funktionsthemen – Archiv, Konvertierung, Tabellen, Krypto.



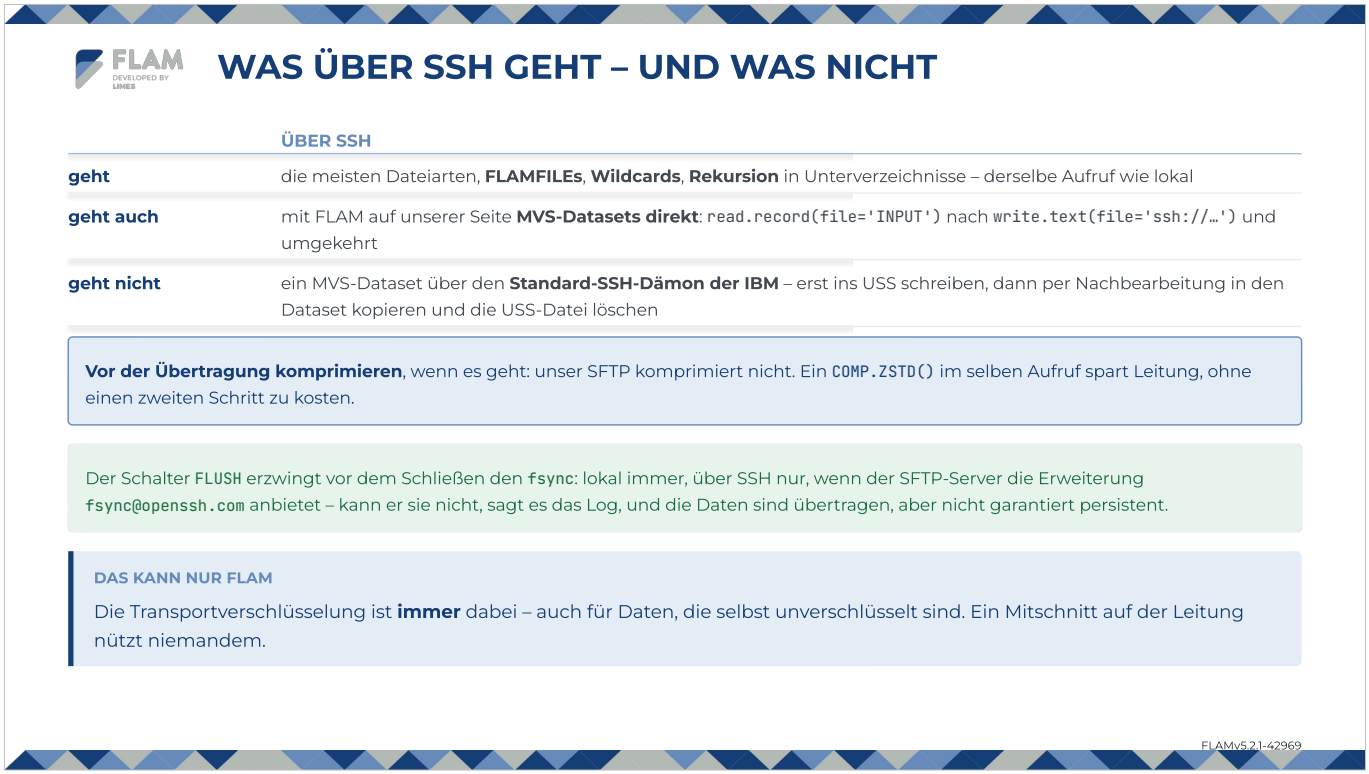
Die vierte Einheit der Ebene 1, und die kürzeste – weil der Netzzugriff absichtlich unspektakulär ist.
Der Bogen: eine URL statt eines Pfades, was dabei mitgeht und was nicht, die Authentifizierung, die Prüfung des Hostkeys, und zuletzt die Frage, was aus FLIES geworden ist.
Der Satz für den Anfang: **eine entfernte Datei ist eine Datei.** Alles, was die vorigen Einheiten über Auswahl, Konvertierung und Namensbildung gesagt haben, gilt über SSH genauso – man tauscht den Pfad gegen eine URL und sonst nichts.



Das vierte Ziel ist das, das in einem Audit gefragt wird – und das man ohne diese Folien falsch beantwortet.
Das fünfte ist der Grund, warum diese Einheit überhaupt FLIES im Titel trägt: die Frage kommt, weil die Handbuchseiten es noch beschreiben.



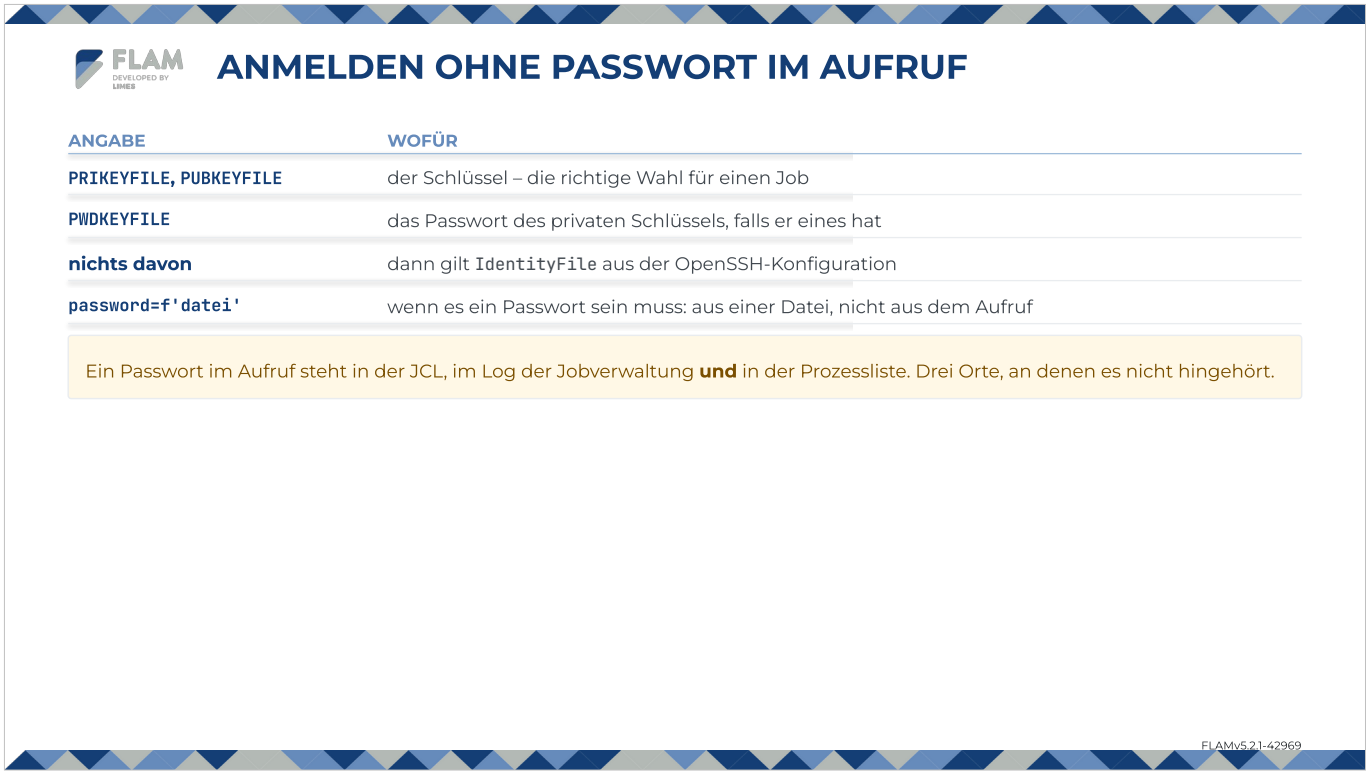
Die Folie, mit der man anfängt, weil sie die Einheit in zwei Zeilen erledigt.
Die **URL** ist die kurze Form: Benutzer, Rechner, optional Port und Pfad.
Sie steht dort, wo sonst der Pfad steht – am `READ` wie am `WRITE`, und auch in einem Archivnamen.
Das **`NET.SSH()`-Objekt** ist die lange Form. Es kann alles, was die URL kann, und zusätzlich die Dinge, für die in einer URL kein Platz ist:
Schlüsseldateien, known_hosts, die Prüfstufe des Hostkeys, Zeitlimit, Algorithmen.
Die Regel, wenn beides dasteht: **die URL gewinnt.** Das steht so im Handbuch und ist praktisch gemeint – die Verbindungsparameter stehen im Objekt, und der Job überschreibt gezielt den Rechner oder den Benutzer.
Der Pfad in der URL zählt die Schrägstriche. Ein Schrägstrich hinter dem Rechner heißt: relativ zum **Heimatverzeichnis des Benutzers**, mit dem sich FLAM anmeldet. Zwei Schrägstriche heißen: absolut, von der Wurzel des fernen Rechners. Eine Tilde gehört dort nicht hinein – sie wäre die lokale Abkürzung und auf EBCDIC-Systemen zusätzlich von der Codepage abhängig (Falk, 2026-09-16).
Und ein Detail für den Host, das in jeder z/OS-Schulung auftaucht: das Klammeraffen-Zeichen liegt in EBCDIC je Codepage anders. Deshalb ist in einer URL auch das **Ampersand** als Trenner zwischen Benutzer und Server erlaubt – dann muss man die CCSID des CLP-Strings nicht kennen.



Die Folie, die eine Enttäuschung vorwegnimmt.
Was geht: fast alles. Dieselben Dateiarten wie lokal, **Wildcards** und **Rekursion in Unterverzeichnisse**. Ein Directory-Walk über SSH ist also derselbe Aufruf wie lokal, und die Namensbildung gilt genauso.
Der Stand ist inzwischen besser als die alte Handbuchstelle: **FLAMFILEs gehen über SSH**, und über unseren Client gehen auch **MVS-Datasets direkt** – in beide Richtungen. `read.record(file='INPUT')` und `write.text(file='ssh://…')` ist ein gültiger Aufruf, und umgekehrt liest man remote und schreibt unmittelbar in einen PS oder ein PO-Member. Der Grund: auf unserer Seite steht FLAM und kennt Sätze.
Die Einschränkung liegt woanders, und sie ist wichtig, weil sie im Kundengespräch kommt: **wenn auf der Host-Seite der Standard-SSH-Dämon von IBM steht**, dann kann man damit kein MVS-Dataset schreiben – SFTP kennt Dateien als Bytesströme, keine Sätze und keine Satzformate. Man schreibt dann erst ins USS und kopiert die Datei über einen **Nachbearbeitungsschritt** in den MVS-Dataset, und löscht die USS-Datei danach. Was Kunden dafür nehmen: FLCL auf dem Host als Postprozess mit "remove at read" – dann wird der Host-Dataset vernünftig geschrieben und die Zwischendatei verschwindet.
Ein `cp` mit nachgelagertem `rm` tut es aber auch.
Und die eigentliche Nachricht dieser Folie: die Verschlüsselung der Übertragung ist **immer** dabei, auch wenn die Daten selbst unverschlüsselt sind. Ein Mitschnitt auf der Leitung nützt niemandem.
Ein praktischer Rat, der auf der Folie steht und oft vergessen wird:
komprimieren, bevor übertragen wird. Unser SFTP komprimiert nicht, und ein `COMP.ZSTD()` im selben Aufruf kostet keinen zweiten Schritt, spart aber die Leitung – bei einer Textdatei um Faktor vier.
Und ein Schalter, der genau hierher gehört, weil er über SSH etwas anderes bedeutet als lokal: **`FLUSH`**. Er steht in jedem Schreibobjekt – `write.text`, `write.binary`, `write.record` – und die Hilfe beschreibt ihn mit "Enforce flush in front of close file". Gemeint ist: vor dem Schließen wird die Datei wirklich hinausgeschrieben und nicht nur an den Puffer abgegeben. Lokal ist das `fflush` und danach `fsync`, unter Windows `FlushFileBuffers`.
Über SSH tut FLAM dasselbe – aber nur, wenn die Gegenseite es anbietet: der SFTP-Server muss die Erweiterung `fsync@openssh.com` können. Kann er sie, wird sie gerufen, und nach dem Close ist alles wirklich geschrieben. Kann er sie nicht, schreibt FLAM eine Zeile ins Log ("fsync() extension not supported by server. Output data might not be persisted, yet.") und die Daten sind übertragen, aber nicht garantiert persistent.
Der praktische Fall: ein Folgejob auf der Gegenseite, der die Datei sofort weiterverarbeitet. Da setzt man `FLUSH` und prüft einmal im Log, ob der SSHD mitspielt – sonst verlässt man sich auf eine Zusage, die niemand gegeben hat.



Die Folie, die einen langen Aufruf kurz macht – und die kaum jemand erwartet.
FLAM liest die OpenSSH-Konfigurationsdateien: die des Benutzers (`~/.ssh/config`) und die des Systems (`/etc/ssh/ssh_config`). Was dort steht, gilt auch für FLAM – Ciphers, Compression, ConnectTimeout, HostKeyAlgorithms, IdentityFile, KexAlgorithms, MACs, Port und die üblichen anderen, in OpenSSH-Syntax.
Praktisch heißt das: wenn die Verbindung mit `sftp` funktioniert, funktioniert sie auch mit FLAM, ohne dass man etwas in den Aufruf schreibt.
Die Rangfolge ist die, die man erwartet, und sie steht im Handbuch:
Aufruf, dann eine eigene Konfigurationsdatei über `CONFIGFILE`, dann die des Benutzers, dann die des Systems. Wer die beiden Voreinstellungen nicht will, schaltet sie mit `NODEFAULTCONFIG` ab – das ist der Griff für einen Produktionsjob, der sich nicht darauf verlassen soll, was im Home eines Benutzers steht.
Der Rat für den Betrieb: die Verbindungsdaten in eine eigene Konfigurationsdatei legen und mit `CONFIGFILE` darauf zeigen. Dann steht im Aufruf nur noch der Rechner, und die Betriebsmannschaft pflegt den Rest an einer Stelle.



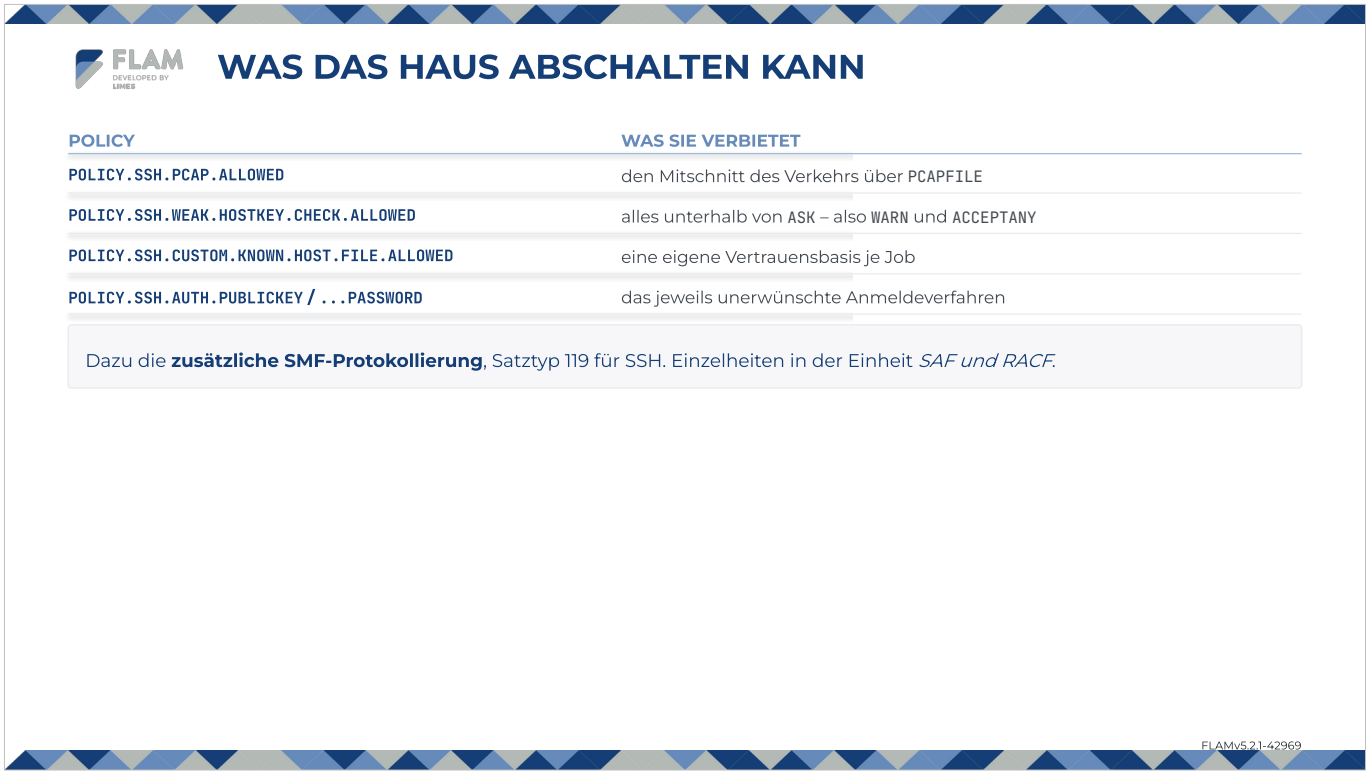
Die Folie mit der Sicherheitsaussage, die am leichtesten umzusetzen ist.
Es gibt zwei Verfahren: Passwort und öffentlicher Schlüssel. Für einen Produktionsjob ist der Schlüssel die richtige Wahl – kein Geheimnis im Aufruf, kein Ablauf, kein Wechsel bei einer Passwortregel.
Die Angaben dafür: `PRIKEYFILE` der private Schlüssel, `PUBKEYFILE` der öffentliche, `PWDKEYFILE` das Passwort des privaten Schlüssels, falls er eines hat. Oder gar nichts davon – dann nimmt FLAM, was in der OpenSSH-Konfiguration als `IdentityFile` steht.
Und wenn es doch ein Passwort sein muss: **nicht in den Parameterstring.** `password=f'/pfad/pw.txt'` liest es aus einer Datei, die man mit Dateirechten schützt; im Log erscheint `***SECRET***`. *Das ist die Angabe aus der Sprach-Einheit, und hier ist ihr wichtigster Einsatz.*
Der Satz für den Raum: ein Passwort im Aufruf steht in der JCL, im Log der Jobverwaltung und in der Prozessliste. Drei Orte, an denen es nicht hingehört.



Die wichtigste Folie der Einheit. Sie entscheidet, ob die Verbindung wirklich sicher ist – die Verschlüsselung nützt nichts, wenn man mit dem falschen Gegenüber verschlüsselt.
Der Server weist sich mit seinem öffentlichen Schlüssel aus. Bekannte Schlüssel stehen in `~/.ssh/known_hosts`, im Format von OpenSSH.
`HOSTKEYCHECK` sagt, wie streng geprüft wird, und das Handbuch listet die Stufen **in absteigender Sicherheit**:
Ganz oben steht der **Fingerabdruck** selbst. Man gibt ihn an, und er muss passen – sonst ist die Verbindung beendet. Ist der Schlüssel noch nicht in der Datei, wird er aufgenommen. Das ist die Angabe für einen **Batchjob**:
niemand kann dort eine Frage beantworten, und man akzeptiert trotzdem keinen beliebigen Schlüssel. Der Fingerabdruck steht bei jedem Verbinden im Log.
`STRICT` verlangt einen Eintrag in der Datei, `ASK` fragt beim ersten Mal nach, `WARN` verbindet auch bei unbekanntem Schlüssel und schreibt eine Warnung, und `ACCEPTANY` nimmt jeden Schlüssel auch noch in die Datei auf – womit er in künftigen strengen Läufen als vertrauenswürdig gilt.
Das Handbuch warnt ausdrücklich vor allem unterhalb von `ASK`. Für den Raum der Satz: **wer `ACCEPTANY` einmal benutzt, hat den Schutz dauerhaft aufgegeben** – nicht nur für diesen Lauf.



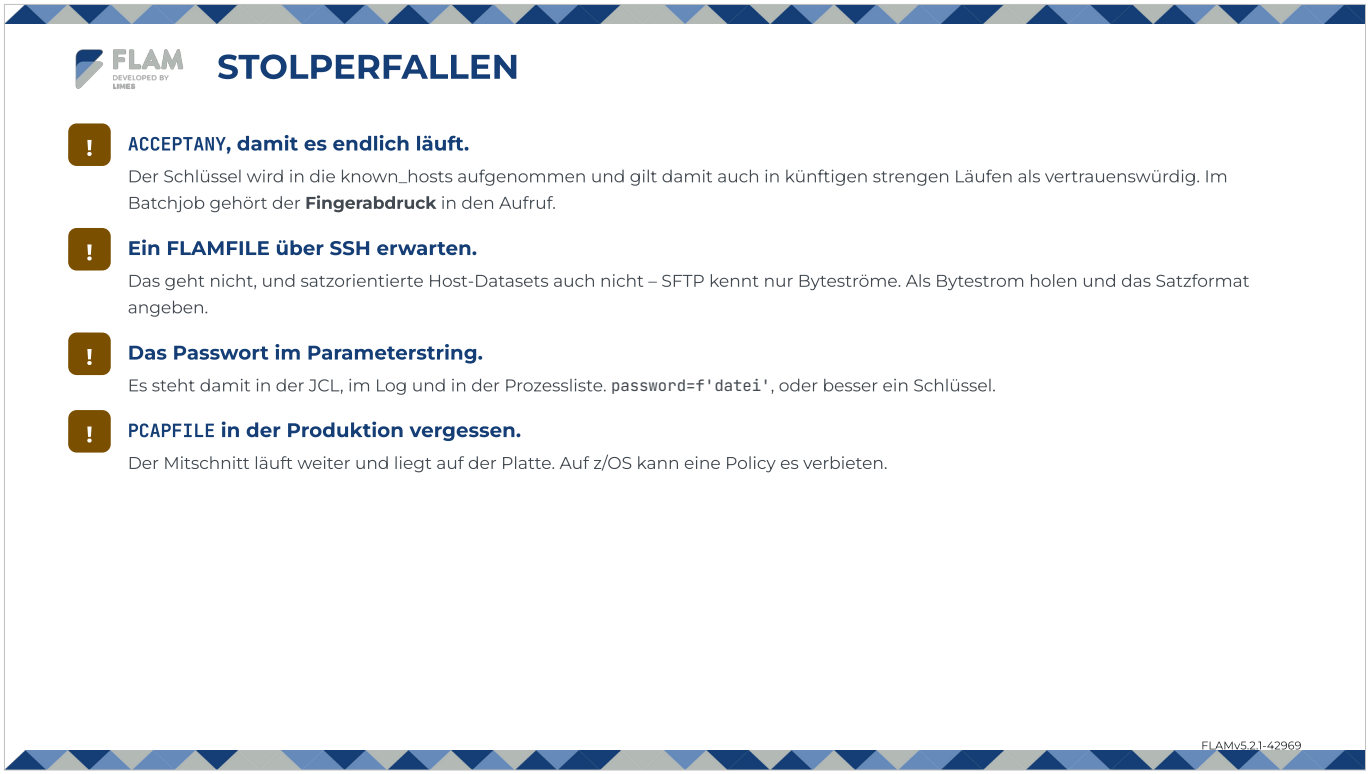
Eine kurze Folie mit einer Aussage, die Arbeit spart.
Vor- und Nachverarbeitung wird auf der Maschine ausgeführt, auf der die Datei liegt. Schreibt man über SSH, laufen die zugehörigen Kommandos **auf der Gegenseite** – man braucht also keinen zweiten Job und keine zweite SSH-Sitzung, um dort etwas anzustoßen.
Zwei Formen: bei `XCNV` gibt es die Kommandos **je Datei**, und dort steht der Dateiname über die Klammer-Muster zur Verfügung. Bei `CONV` gibt es sie global – einmal vor und einmal nach dem Lauf. Wer je Datei etwas braucht, nimmt `XCNV`.
Und die Angabe, die man dazu braucht: im `POSTPROCESS` steht ein eigenes `NET.SSH()`, damit ist ausdrücklich gesagt, wo das Kommando läuft. Die Konvertierungs-Einheit zeigt den Fall vollständig – Prüfsumme prüfen, dann auspacken.
Der Rat: das Kommando läuft **ohne Shell**. Wer eine Verkettung braucht, ruft ein Skript auf der Gegenseite.
Daraus folgt eine Kleinigkeit, die sonst Zeit kostet: **ohne Shell wird auch keine Tilde aufgelöst.** Das Kommando startet im Heimatverzeichnis des Benutzers, ein relativer Pfad wie `bin/uebernahme.sh` trifft also – eine Tilde davor wäre ein Zeichen im Dateinamen und der Aufruf schlägt fehl.



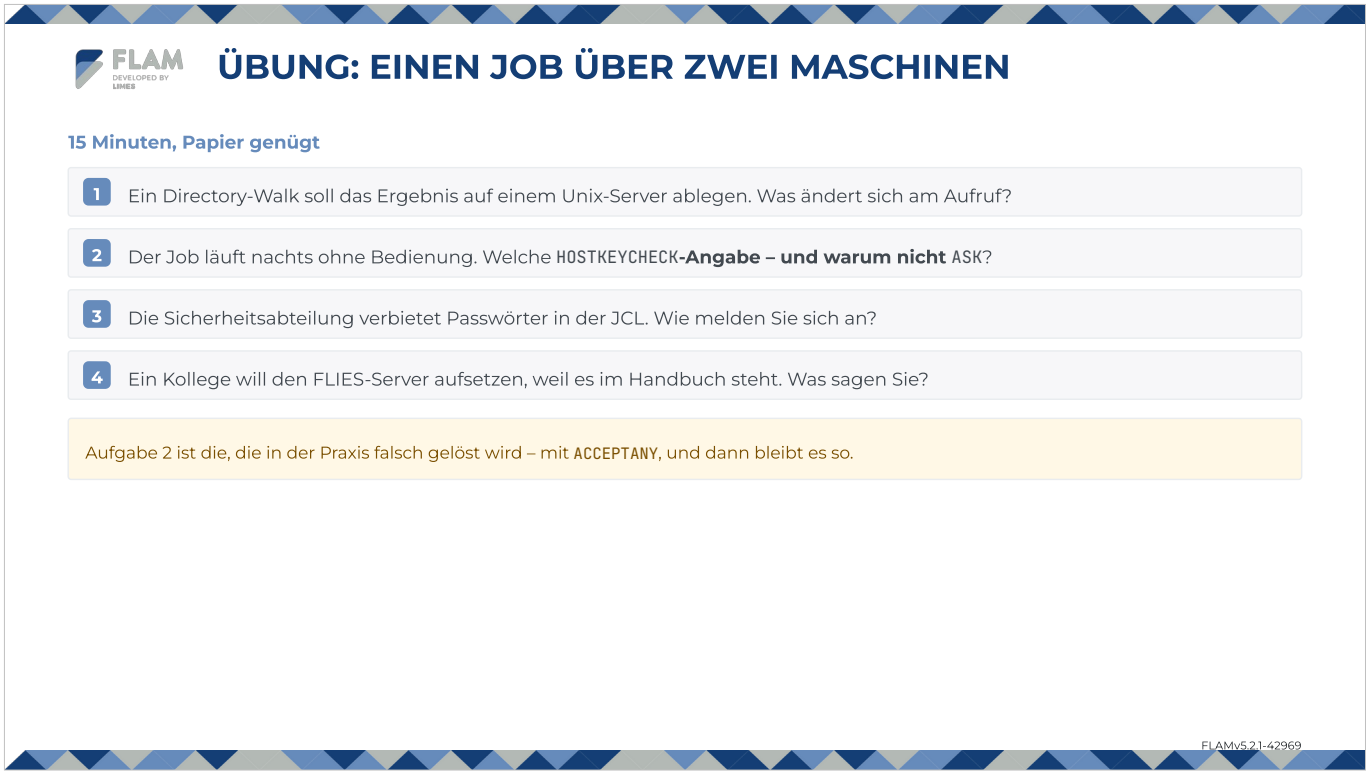
Die Folie, die diese Einheit mit der Sicherheitsgruppe verbindet – kurz halten, dort steht es ausführlich.
Auf z/OS kann ein RACF-Administrator die Schrauben festziehen, die hier gezeigt wurden:
Das **Packet Capturing** über `PCAPFILE` schreibt den Verkehr mit – für eine Fehlersuche unentbehrlich und in der Produktion eine offene Tür.
`POLICY.SSH.PCAP.ALLOWED` verbietet es.
Die **schwache Hostkey-Prüfung** – alles unterhalb von `ASK` – lässt sich über `POLICY.SSH.WEAK.HOSTKEY.CHECK.ALLOWED` verbieten. Damit kann ein Job `ACCEPTANY` schreiben, so oft er will.
Eine **eigene known_hosts-Datei** kann man über `POLICY.SSH.CUSTOM.KNOWN.HOST.FILE.ALLOWED` verbieten – sonst bringt ein Job seine eigene Vertrauensbasis mit.
Dazu kommen `POLICY.SSH.AUTH.PUBLICKEY` und `...PASSWORD`, mit denen das Haus entscheidet, welches Anmeldeverfahren erlaubt ist, und die **zusätzliche SMF-Protokollierung** mit Satztyp 119 für SSH.
Der Satz, der die Verbindung herstellt: eine Angabe im Job ist ein Wunsch, eine Policy ist eine Regel. Die SAF-Einheit sagt, wie man daraus eine Regel macht.



Die Folie, die eine Frage beantwortet, die aus den Handbuchseiten kommt – und sie beantwortet sie ehrlich.
FLIES steht für Frankenstein Limes Integrated Extended Security. In der FLAM4-Welt ist es der Server, über den entfernte Zugriffe liefen, mit drei Verfahren: **IPN** als natives IP auf Port 17996, **IPS** dasselbe mit SSL/TLS, und **MQS** über IBM MQ-Series. Für alle drei musste auf der Gegenseite ein FLIES-Server laufen.
Und heute? Nachgemessen: die drei stehen in der Parametertabelle, aber mit einem Flag, das sie **nicht in die Kommandosprache** aufnimmt. Der Parser lehnt `net.ipn(...)` ab – "Parameter nicht gültig". Was in FL5 also wirklich da ist, ist **SSH**, und für Archive im Objektspeicher die Cloud-Anbindung.
Warum das kein Verlust ist, und das sollte man so sagen: SSH braucht auf der Gegenseite **keinen FLAM-Server**, nur einen SSH-Dienst – und der läuft überall. Ein eigener Serverdienst ist Betrieb, Überwachung und eine Portfreigabe; ein SSH-Dienst ist schon da.
Wenn die Frage nach den Handbuchseiten kommt: sie beschreiben die FLAM4-Welt. Das ist ein Doku-Befund, den wir notiert haben.
Und es gibt einen zweiten Weg über die Leitung, der hier hineinspielt und den man nennen sollte, weil er die Richtung zeigt: **HTTPS**. Über **FLREST** sprechen wir mit den Cloud-Objektspeichern, und auch die Anbindung von IBM CDA läuft darüber. Wenn wir den **FLSERV** fertig machen, wird er seine Segmente ebenfalls über HTTPS mit FLREST übertragen – aus IPN, IPS und MQS wird dann also **RST**. Ein eigenes Protokoll auf einem eigenen Port kommt nicht zurück.



Alle vier aus Supportfällen.



Fünfzehn Minuten, Papier genügt.
Die Lösungen:
Aufgabe 1. `file='ssh://<cuser>@server/daten/[base].csv'` – die URL steht dort, wo sonst der Pfad steht, und die Namensbildung gilt unverändert.
Aufgabe 2. Den Fingerabdruck angeben. `ASK` kann im Batch niemand beantworten, `STRICT` scheitert beim ersten Verbinden, und `ACCEPTANY` gibt den Schutz dauerhaft auf. Der Fingerabdruck steht im Log jedes Verbindungsaufbaus.
Aufgabe 3. Einen Schlüssel benutzen: `prikeyfile=...`. Wenn es ein Passwort sein muss, `password=f'datei'` mit Dateirechten.
Aufgabe 4. Nein. FLIES ist die FLAM4-Welt; in FL5 nimmt die Kommandosprache die drei Verfahren nicht an – gemessen. Der Weg ist SSH, und der braucht dort keinen FLAM-Server.

WAS BLEIBT

- eine **URL statt eines Pfades** – Auswahl, Konvertierung und Namensbildung gelten unverändert
- **nicht** über SSH: FLAMFILEs und satzorientierte Host-Datasets
- die **OpenSSH-Konfiguration** gilt auch für FLAM – `NODEFAULTCONFIG` schaltet sie ab
- im Batch gehört der **Fingerabdruck** in den Aufruf, nicht `ACCEPTANY`
- Vor- und Nachverarbeitung läuft **dort, wo die Datei liegt**
- **FLIES** ist die FLAM4-Welt – in FL5 ist der Weg SSH, ohne Server auf der Gegenseite

FLAMv5.21-42969

Lernziele durchgehen. Der Verweis nach vorn: wo ein **Archiv** physisch liegt – Objektspeicher, Cloud – steht in der Einheit *Segment-Store und Cloud*. Diese Einheit war der Zugriff auf **Dateien**.

FLAM SCHULUNG

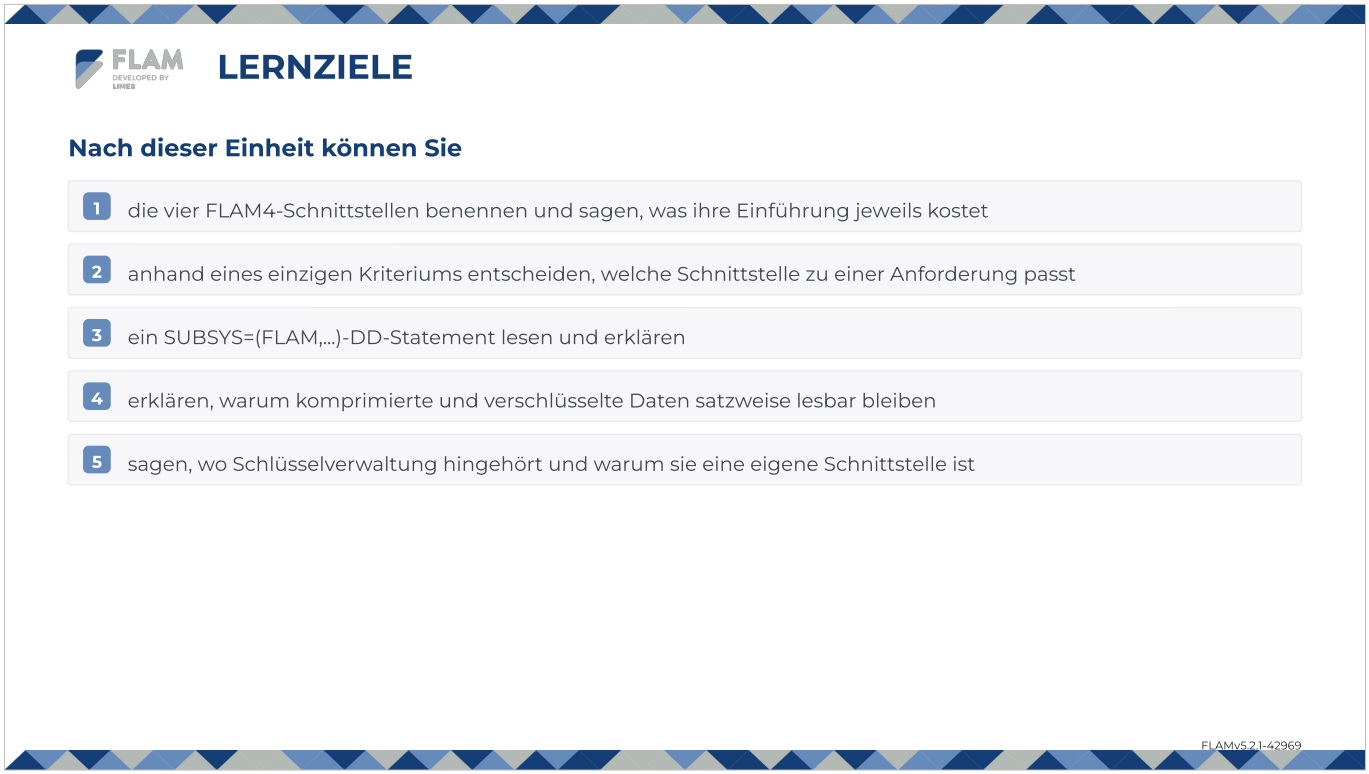
Vier Wege in dasselbe Verfahren

und wie man den richtigen wählt

Falk Reichbott · limes datentechnik gmbh

limes: leistung im grenzbereich des machbaren.

Rahmen setzen: FLAM4 ist kein Komprimierungswerkzeug mit angebautem API, sondern ein Zugriffsverfahren mit vier Eingängen. Die ganze Einheit dreht sich um die Wahl zwischen diesen vier Eingängen.
Fragen Sie zuerst, welche davon im Raum heute benutzt werden – die meisten Häuser nutzen Utility und Subsystem und wissen nicht, dass es die Satzschnittstelle überhaupt gibt.



Die fünf Ziele sind der Vertrag für die Einheit; kommen Sie in der Zusammenfassung darauf zurück. Das eine Kriterium in Ziel zwei ist die Zuständigkeit für die Änderung: wer muss etwas anfassen – Betrieb, Build oder Quellcode.

EIN ZUGRIFFSVERFAHREN, VIER EINGÄNGE

Die Schnittstelle entscheidet, **wer etwas ändern muss** – nicht, was am Ende herauskommt.

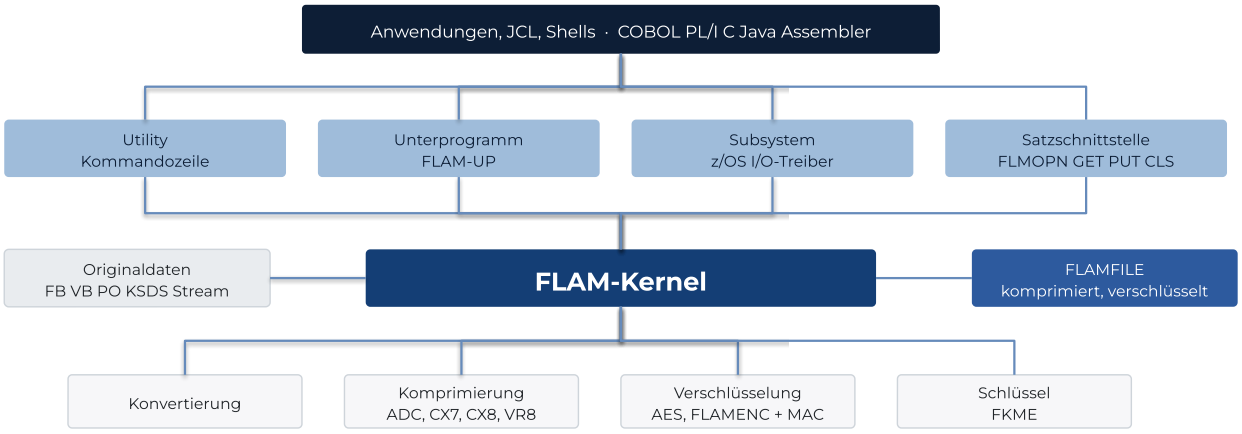
SCHNITTSTELLE	WER ÄNDERT ETWAS	TYPISCHER ANLASS
Utility	Betrieb, in JCL oder Skript	Migration, Austausch, Batch
Unterprogramm	der Build, ein Modul binden	eigenes Tool, eigene Werkzeuge
Subsystem	Betrieb, ein DD-Statement	Anwendung darf nicht angefasst werden
Satzschnittstelle	der Anwendungsquellcode	Programm arbeitet selbst satzweise

Die FLAMFILE ist in jeder Zeile dieselbe – eine Entscheidung hier ist deshalb nie endgültig.

FLAMv5.21-42969

Diese Folie trägt die ganze Einheit. Die FLAMFILE ist in allen vier Fällen identisch und untereinander austauschbar – deshalb ist die Entscheidung rein organisatorisch.
Die Altfolien haben die vier Schnittstellen als vier Produktdatenblätter präsentiert; das verdeckt die einzige Frage, die ein Kunde beantworten muss.

FLAM4 IM BILD



Jeder Pfad läuft auch rückwärts – das unterscheidet ein Zugriffsverfahren von einem Konverter.

`MODE=` wählt beides in einem Parameter: `ADC` `CX7` `CX8` `VR8` für die Kompression, `AES` und `FLAMENC` für die Verschlüsselung, `NDC` für „nicht komprimieren". Achtung: mit `CX7`, `CX8` oder `VR8` ist **keine** Verschlüsselung möglich.

FLAMv5.2.1-42969

Von oben nach unten durchgehen. Die vier Schnittstellen sitzen auf demselben Kernel; der Kernel konvertiert, komprimiert, verschlüsselt und rechnet den MAC, und er holt Schlüsselmaterial über den Key-Management-Exit.
Betonen, dass alle Wege auch rückwärts laufen: das macht FLAM zu einem Zugriffsverfahren und nicht zu einem Konverter.



Die Unterscheidung Unterprogramm gegen Satzschnittstelle wird oft verwechselt:
das Unterprogramm nimmt ganze Dateien, die Satzschnittstelle nimmt Sätze.
Wenn jemand sagt "wir wollen FLAM aus unserem eigenen Tool rufen", ist fast immer das Unterprogramm die Antwort, nicht die Satzschnittstelle.

SUBSYSTEM: DAS DD-STATEMENT MACHT DIE ARBEIT



Die Anwendung setzt weiter gewöhnliche QSAM- oder VSAM-Aufrufe ab und erfährt nie, dass die Daten komprimiert und verschlüsselt sind.

```
//DDNAME   DD  DSN=filename,
//             SUBSYS=(FLAM,'CRYPTOM=AES',
//             'KME=FKMEFILE,KMP=C''ddname1''')
```

Quelle: **doc/man_flam4_appendix_flamsub.txt** – die doppelten Hochkommata sind die Falle.

FLAMv5.21-42969

Quelle des Beispiels: doc/man_flam4_appendix_flamsub.txt, Abschnitt "Encryption using FKMEFILE". Diese Syntax nicht aus dem Kopf auf eine Folie schreiben – die doppelten Hochkommata sind der Teil, den alle falsch machen.
Die Anwendung setzt weiter gewöhnliche QSAM- oder VSAM-Aufrufe ab und erfährt nie, dass die Daten auf der Platte komprimiert und verschlüsselt sind.
Zweiter Effekt, der oft verkauft: Komprimierung heißt weniger File-I/O, der Unload wird also schneller, während er zusätzlich geschützt wird.



Die sieben Eintrittspunkte sind die ganze Schnittstelle; das ist der Punkt, denn man erwartet von einem Zugriffsverfahren etwas Großes.
Sicherheitsargument: nur das gerade benötigte Segment existiert klar, und nur im Hauptspeicher. Genau das fordern mehrere Sicherheitsstandards, und deshalb entsteht nie eine temporäre Klartextdatei.
Auch für Micro Focus COBOL verfügbar – kommt regelmäßig, wenn ein Kunde eine Hostanwendung nach Linux bewegt.

EINEN SATZ IM GROSSEN ARCHIV FINDEN

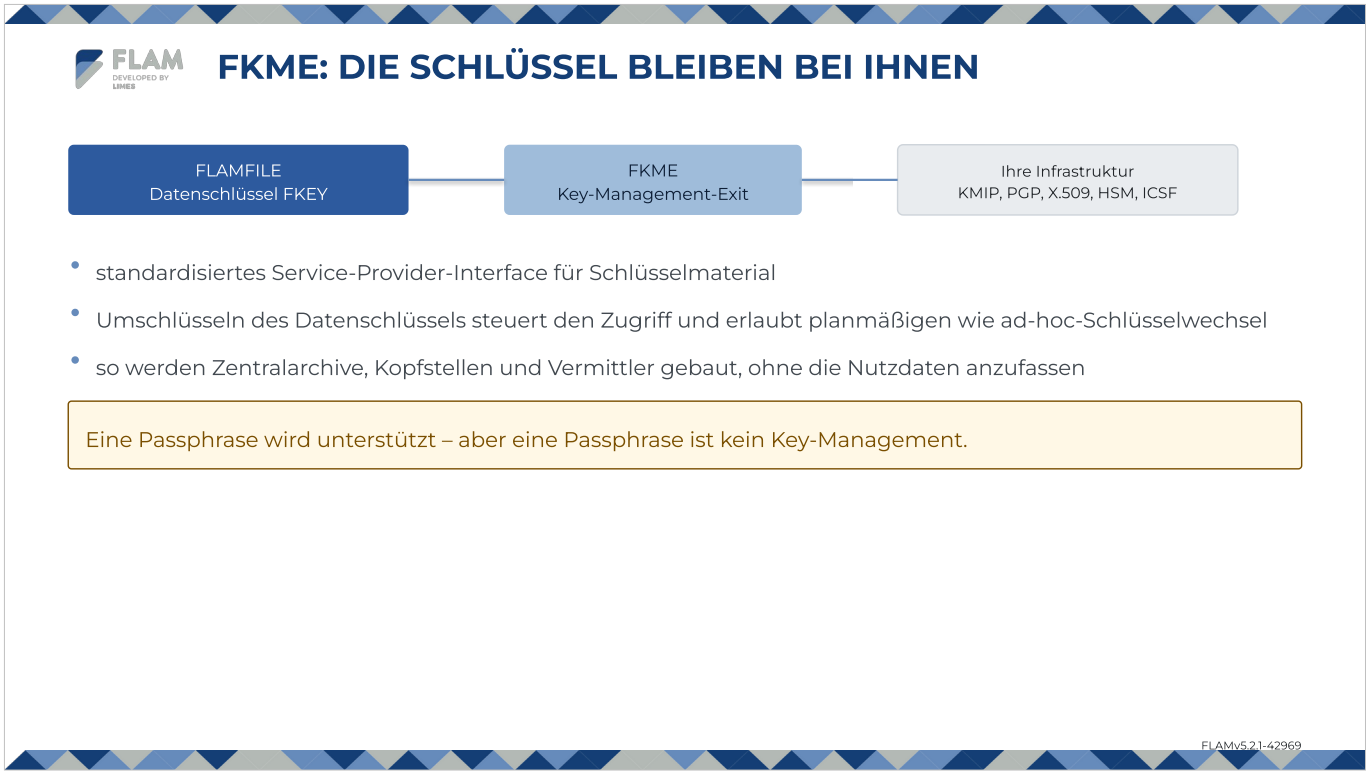
- indexsequentielle FLAMFILEs: lesen, schreiben, ändern, einfügen, löschen – über den Schlüssel des Originalsatzes
- FLAM nimmt den Schlüssel an der Satzschnittstelle an und findet damit das richtige Segment
- die Suche dekomprimiert den Rest des Archivs nicht
- auf z/OS transparent durch das Subsystem mit VSAM KSDS

Damit wird ein Archiv mit echter Suche bezahlbar: die Ergebnismenge bleibt verschlüsselt, bis jemand berechtigt ist, sie zu sehen.

Weiter im Foliensatz FLAM5-Archiv: dort ersetzen Matrix, Segmente und Bloom-Filter den Mechanismus – das Versprechen bleibt.

FLAMv5.21-42969

Der Mechanismus ist das Segment: der Schlüssel wählt ein Segment, nur das wird dekodiert. Die Kosten pro Treffer bleiben etwa konstant, wenn das Archiv wächst – diese Eigenschaft macht aus einem komprimierten Backup ein abfragbares Archiv.
Diese Folie ist die Brücke zum FLAM5-Foliensatz: FLAM5 ersetzt den Mechanismus (Matrix, Segmente, Bloom-Filter), behält aber das Versprechen.



Der Sinn der Schnittstelle ist, dass der Kunde seine eigenen Schlüsselprozesse und Berechtigungsregeln behält; FLAM wird kein zweiter Schlüsselspeicher.
Konkret: der Datenschlüssel einer FLAMFILE (FKEY) wird von einem Hauptschlüssel (FMKY) geschützt, der in der Kundeninfrastruktur liegt. Das Umschlüsseln von FKEY übergibt das Archiv an einen anderen Empfänger, ohne die Nutzdaten anzufassen – so werden Zentralarchive, Kopfstellen und Vermittler gebaut.
Der letzte Satz ist der Merksatz; schreiben Sie ihn aufs Whiteboard.

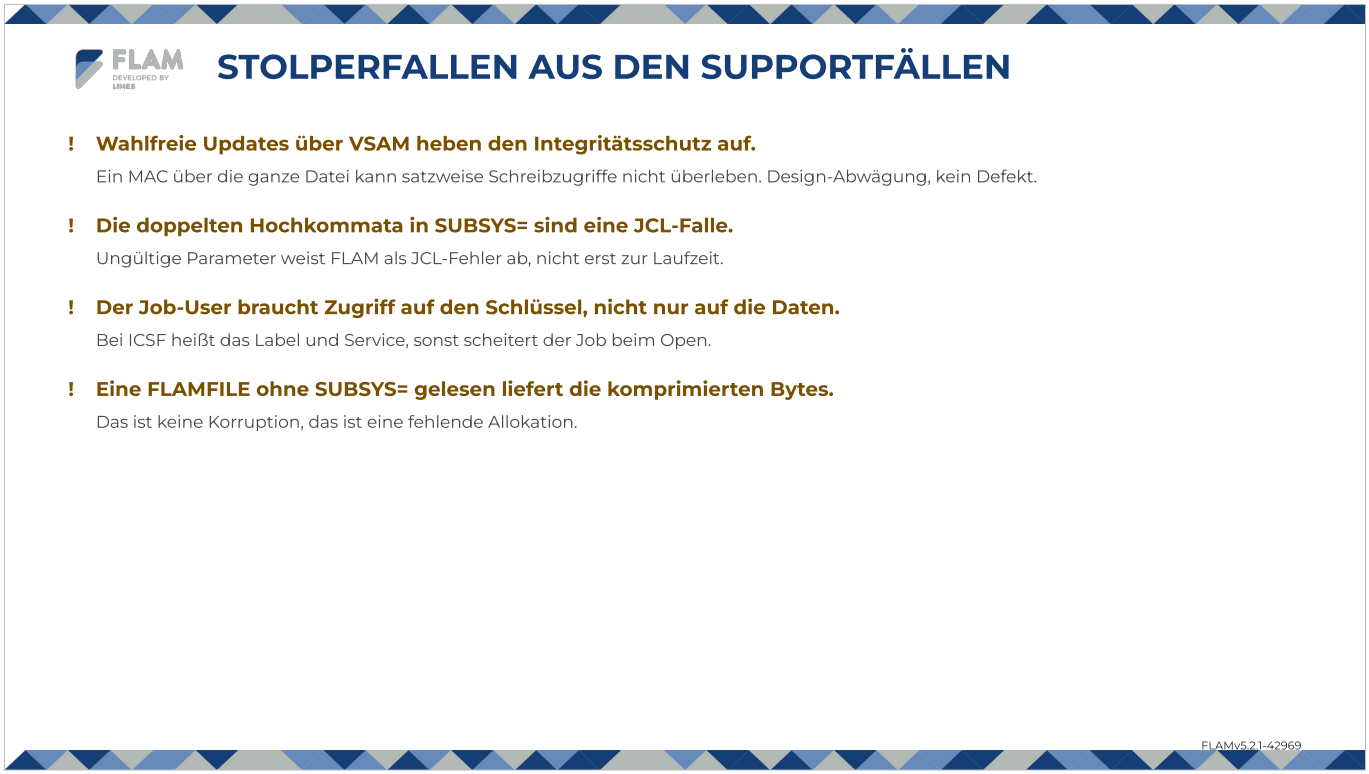
EINE FRAGE, VIER ANTWORTEN

WENN DIE ANFORDERUNG IST	DANN	WEIL
ganze Dateien austauschen oder migrieren	Utility	kein Code, kein Build, skriptfähig
unser eigenes Werkzeug soll es tun	Unterprogramm	ein Modul binden, Dateiebene
diese Anwendung darf nicht geändert werden	Subsystem	nur JCL, transparent
das Programm verarbeitet einzelne Sätze	Satzschnittstelle	Satzebene, kleine Änderung
der Schlüssel muss aus unserem HSM kommen	FKME, zu jedem der obigen	Key-Management ist orthogonal

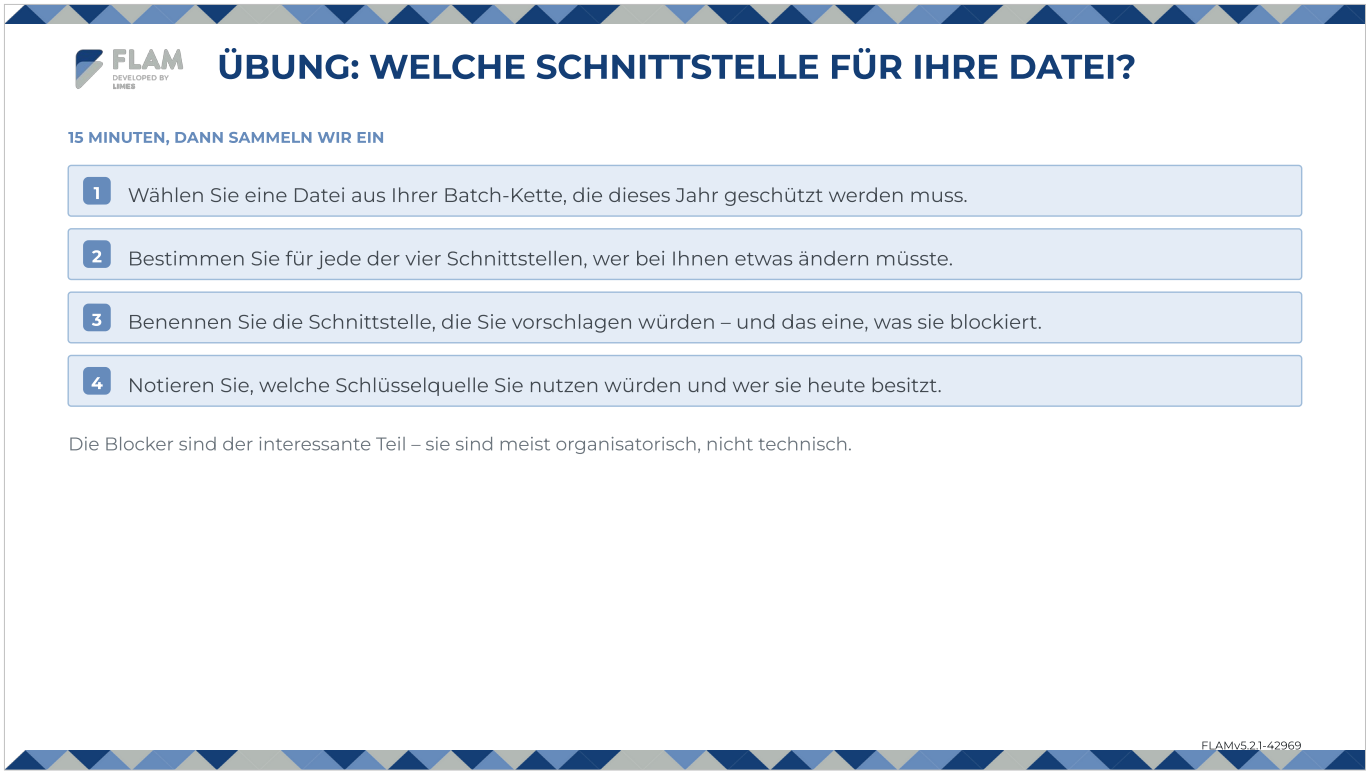
Ein Haus kann mit dem Utility anfangen und später auf das Subsystem wechseln, ohne Daten zu konvertieren.

FLAMv5.21-42969

Diese Folie austeilen. Sie ist die, die abfotografiert wird.
Die letzte Zeile betonen: weil der Container identisch ist, kann ein Haus mit dem Utility anfangen und später auf das Subsystem wechseln, ohne Daten zu konvertieren.



Alle vier kommen aus Supportfällen, und alle vier sind billig zu vermeiden.
Die erste ist eine echte Design-Abwägung, kein Defekt: der Kunde wählt zwischen satzweisem Schreibzugriff und einem Integritätsnachweis auf Dateiebene. Sagen Sie das offen, das schafft Vertrauen.



In vier Spalten sammeln, eine pro Schnittstelle. Die Blocker sind der interessante Teil – sie sind meist organisatorisch, nicht technisch, und sie sagen Ihnen, worum es im Folgetermin gehen muss.
Ergebnisse aufbewahren, sie füllen die Anforderungsliste am Ende des Workshops.
Auflösung: das Kriterium ist immer, in welcher Einheit die Anwendung denkt und wer etwas ändern darf.
- Eine Bestandsanwendung, die nicht angefasst werden darf: Subsystem. Die Anwendung merkt nichts, geändert wird die Allokation.
- Ein eigenes Werkzeug, das ganze Dateien verarbeitet: Unterprogramm.
- COBOL oder PL/I mit Satzverarbeitung: Satzschnittstelle.
- Ein Job, der eine Datei aufbereitet und weitergibt: Batch-Utility – kein Code, kein Build.
Die häufigste falsche Wahl im Raum ist das Unterprogramm, wenn eigentlich das Subsystem gemeint ist: wer nichts am Code ändern will, will das Subsystem.

WAS BLEIBT

- ein Zugriffsverfahren, vier Eingänge, eine identische FLAMFILE
- die Wahl betrifft die Zuständigkeit für die Änderung, nicht das Ergebnis
- das Subsystem ist die Antwort, wenn die Anwendung nicht angefasst werden darf
- die Satzschnittstelle ist I/O mit sieben Aufrufen, kein Framework
- Schlüsselverwaltung ist absichtlich eine eigene Schnittstelle

WEITERLESEN

Subsystem, DD-Syntax, Grenzen	doc/man_flam4_appendix_flamsub.txt
Key-Management-Exit	doc/man_flam4_appendix_kmexit.txt
User-I/O und User-Exits	doc/man_flam4_appendix_userio.txt
Was FLAM5 ändert	Foliensatz FLAM5-Archiv

FLAMv5.21-42969

Zum Schluss die Lernziele noch einmal vorlesen und fragen, ob alle fünf erreicht sind.
Der nächste Foliensatz in der empfohlenen Reihenfolge ist FLAM5-Archiv; die Gegenüberstellung FLAM4 zu FLAM5 liegt dort, nicht hier.



Erstes Funktionsthema. Die Einheit erklärt, warum es ein neues Archivformat gab – und die Antwort ist nicht "schneller", sondern "verwaltbar".
Der Bogen: was FLAM1 bis FLAM4 konnten und wo sie an die Wand liefen, was daraus als Anforderung wurde, und wie FLAM5 aufgebaut ist.



Ziel zwei ist der Kern: die Trennung von logischer Archivverwaltung und physischer Segmentablage ist die Idee, aus der alles andere folgt.

FLAM
DEVELOPED BY LIMES

RÜCKBLICK: FLAM1 BIS FLAM4

STUFE	WAS DAZUKAM	WAS BLIEB
FLAM1	autarke Segmente, Matrix im Speicher, Segmentkopf beschreibt die Struktur	eine Datei, ein Komprimat
FLAM2	Member-Header, damit das erste Archiv; FLAM-VSAM mit Signaturdatei für die Suche	Kunden verketten FLAMFILEs
FLAM3	mehrere Dateien in einem Komprimat, Memberliste auflistbar	Verketten bleibt problemlos
FLAM4	AES, Integritätsschutz über MAC und FKME für HSM-basiertes Key-Management	**der Aufbau stößt an seine Grenze**

DAS KANN NUR FLAM

Jede Stufe war rückwärtskompatibel. Ein FLAMFILE aus den frühen Neunzigern liest FLAM heute noch – und das ist der Grund, warum die großen Bestände in den Rechenzentren FLAM4-Archive sind.

FLAMv5.21-42969

Diese Folie erklärt, warum es ein neues Format gab – ohne sie klingt FLAM5 nach Selbstzweck. Jede Stufe war rückwärtskompatibel, und jede Stufe brachte genau eine neue Fähigkeit. Das ist die eigentliche Leistung dieser Zeile: 30 Jahre Format ohne Bruch.
Die letzte Zeile ist die entscheidende, und sie führt auf die nächste Folie:
mit FLAM4 kam professionelles Key-Management, und damit stieß der Aufbau an zwei Stellen an seine Grenze. Hier nur ankündigen, nicht ausführen.



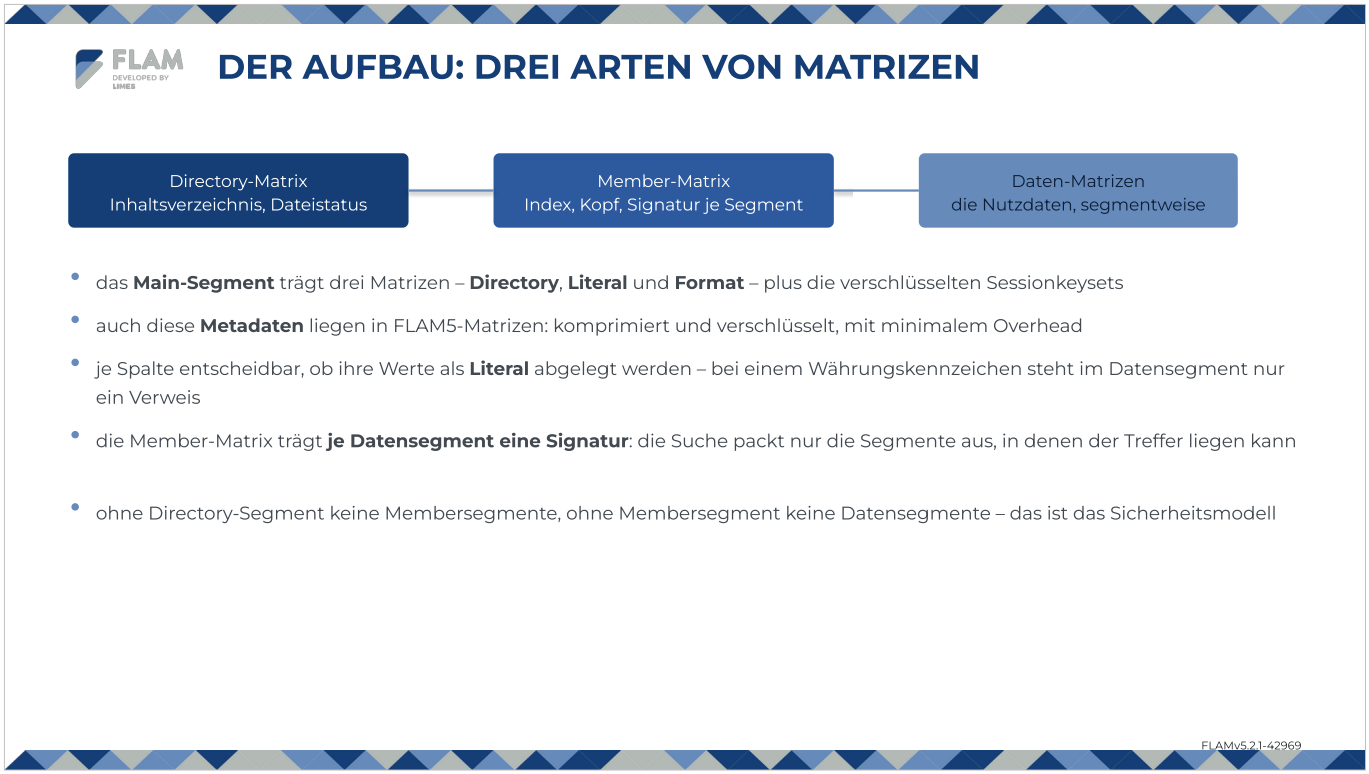
Das ist die Folie, die begründet, warum FLAM5 gebaut wurde. Beide Punkte ehrlich benennen – wer sie kennt, versteht die Architektur von FLAM5 sofort, und in einem Haus mit FLAM4-Beständen ist das eine praktisch relevante Information, keine Selbstkritik.
Erste Grenze, der Integritätsschutz. In der FLAM4-Welt macht die Zugriffsmethode selbst den MAC kaputt: Update, Insert und Delete verändern den Bestand, und der MAC über das Ganze kann nicht korrekt nachgerechnet werden. Wichtig ist die genaue Abgrenzung, weil sie die Aussage rettet: die Integrität der einzelnen Segmente bleibt über den Segment-MAC gewährleistet.
Was nach einem verändernden Zugriff auf ein altes FLAM4-Archiv nicht mehr garantiert werden kann, ist die Vollständigkeit eines Members oder des Archivs. Man kann die Prüfung dann nur abschalten – also auf die Vollständigkeitsprüfung verzichten.
Zweite Grenze, das Key-Management. Um professionelles Key-Management anbieten zu können, musste der verschlüsselte Sessionkey irgendwo im alten FLAMFILE unterkommen. Genommen wurde das Kommentarfeld des UserHeaders. Das war eine Notlösung, und wir nennen sie auch so.
Dritte Grenze, und die ist die praktisch teuerste: **Suchen und Finden zusammen mit Update und Delete gab es in FLAM4 nur als Sonderlösung.** In diesem Projekt heißt sie FLM4SEPA. Dort arbeiten **drei VSAM-Cluster** zusammen – einer für das Verzeichnis, einer für den Index, einer für die Daten. Das lief nur auf dem Host, war für **ein** bestimmtes Datenformat implementiert, und jede Erweiterung der Suchfelder war wieder Entwicklungsarbeit an dieser Lösung.
Genau das haben die FLAM5-Archive **implizit** eingebaut: Bloom-Filter, Indizierung, Vollzugriff – ohne dass jemand programmieren muss, und plattformunabhängig.
Und das ist die Aussage, die in einem Haus mit solchen Lösungen zählt: **wir lösen diese Sonderlösungen durch einen plattformunabhängigen Standard ab und holen die Kunden damit aus der VSAM-Falle.** Wer heute drei Cluster pflegt, um in seinen Archiven suchen zu können, tauscht sie gegen ein Archivformat – und nimmt es beim nächsten Plattformwechsel mit.
Der Schlusssatz: genau diese Punkte sollten die FLAM5-Archive beheben – voller Zugriff, Suche ohne Sonderlösung, und Integritäts- und Vollständigkeitsschutz bleiben dabei gewahrt.

WAS FLAM5 KÖNNEN MUSSTE

GRUPPE	ANFORDERUNG
Zugriff	voller Zugriff auf einzelne Elemente – Update, Insert, Delete, Suche – trotz Verschlüsselung und Integritätsschutz
Struktur	Verzeichnis wie bei ZIP, Indizes für die Suche, vollständiger Dateistatus je Member (CCSID, Format, DCB)
Krypto	eigene Felder für verschlüsselte Datenschlüssel und Signaturen – kein Missbrauch des Kommentarfeldes mehr
Kompression	verschiedene Standardverfahren (GZIP, BZIP, LZ4, ZSTD) bei ADC-Qualität, inklusive ZEDC
Ersatz	FLAM-VSAM und FLAM-ISAM ablösen – und so schnell sein wie zwei VSAM-Cluster

FLAMv5.21-42969

Nicht alle Punkte vorlesen; die Liste war lang und steht in den Altfolien.
Gruppieren und die vier Gruppen erklären.
Die wichtigste Zeile ist die zweite: voller Zugriff – Update, Insert, Delete, Suche – auf einzelne Elemente, bei professioneller Verschlüsselung. Das ist die Anforderung, die den ganzen Aufbau bestimmt.
Der Ersatz für FLAM-VSAM und FLAM-ISAM ist der Grund, warum es überhaupt eilte, und die Messlatte war: so schnell wie zwei VSAM-Cluster.



Von unten nach oben erklären: Datenmatrizen enthalten die Nutzdaten, Membermatrizen die Indizes und Signaturen je Segment, die Directory-Matrix das Inhaltsverzeichnis.
Jetzt genauer, weil die Rollen im Gespräch immer durcheinandergehen:
Die **Directory-Matrix** beschreibt, aus welchen **Membern** das Archiv besteht.
Die **Member-Matrizen** beschreiben, aus welchen **Datensegmenten** ein Member besteht – und sie tragen **pro Datensegment eine Signatur**. Diese Signatur ist der Grund, warum die Suche billig ist: bei einer Suche liest man nur das Member-Segment, baut daraus die Matrix, prüft, bei welchen Signaturen die gesuchten Daten überhaupt liegen **können**, und nur diese Datensegmente werden ausgepackt. Alles andere bleibt komprimiert und verschlüsselt liegen.
Die **Daten-Matrizen** enthalten die eigentlichen Nettodaten der Records.
Zwei Namen, die im Log und in Fehlermeldungen auftauchen und die man zuordnen können sollte: die Directory-Matrix heißt intern `**FL5DIR**`, die Literal-Matrix `**FL5LIT**`. Und wenn eine Textdatei ins Archiv geht, legt `read.text()` intern eine Matrix namens `**TXTRST**` an – darin stehen die Reste, die beim Trennen der Zeilen anfallen: Trennzeichen und angehängter Leerraum. Das ist der Grund, warum ein Text nach dem Rundlauf wieder **byteweise** so aussieht wie vorher, obwohl FLAM in Sätzen gerechnet hat.
Wer wissen will, warum ein Archiv eine Zeilenliste mit `TXTRST` enthält:
genau deshalb.
Und dann das **Main-Segment**, das mehr enthält, als man denkt. Es besteht aus drei Matrizen plus den verschlüsselten Sessionkeysets:
die **Directory-Matrix**, die **Literal-Matrix**, in der die Literale gehalten werden, und die **Format-Matrix**, in der die Formatbeschreibungen der Daten im Archiv stehen.
Warum das clever ist: weil die Literale **zentral** in einer Matrix und die Formatbeschreibungen **zentral** in einer weiteren gesammelt werden – und diese Matrizen wiederum komprimiert und verschlüsselt sind. Damit sind die gesamten Metadaten sicher **und** verursachen extrem wenig Overhead. Man zahlt die Struktur nicht mit Platz.
Und dann der Kniff, den man auf dem Whiteboard erklären sollte, weil er sofort einleuchtet: **der Anwender entscheidet pro Spalte, ob die Werte dieser Spalte als Literal in die Literal-Matrix gehen.** Nehmen Sie eine Spalte, die immer nur dieselben fünf Schlüsselwörter trägt – ein Währungskennzeichen, `EUR`, `USD`, `CHF` und so weiter. Dann bildet man diese ganze Spalte über die Literal-Matrix ab: im Datensegment steht nur noch ein Verweis. Bei Millionen Records ist das der Unterschied zwischen drei Zeichen pro Zeile und drei Bit.
Genau diese Kniffe zusammen machen FLAM am Ende so effektiv – nicht ein besserer Kompressor.
Und die Zugriffskette: ohne Directory-Segment keine Membersegmente, ohne Membersegment keine Datensegmente. Das ist gleichzeitig das Sicherheitsmodell.

DREI SCHLÜSSEL, DREI SICHTEN

SCHLÜSSEL	WAS ER ÖFFNET	WER IHN TYPISCHERWEISE HAT
Directory	das Inhaltsverzeichnis: welche Member existieren, mit Status	Betreiber, Archivverwaltung
Index	die Suche: welches Segment enthält einen Treffer	Suchdienst, Vermittler
Daten	die Nutzdaten im Klartext	nur der Berechtigte am Ende der Kette

Ein Betreiber kann ein Archiv verwalten und darin suchen lassen, ohne die Daten sehen zu können. Das ist die Grundlage für Zentralarchive, Kopfstellen und Cloud.

FLAMv5.21-42969

Das ist die Folie, die Cloud- und Outsourcing-Gespräche entscheidet.
Wer nur den Directory-Schlüssel hat, sieht das Inhaltsverzeichnis und sonst nichts. Wer den Index-Schlüssel hat, kann suchen, aber nicht lesen. Erst der Datenschlüssel gibt Klartext.
Praktische Folge: ein Cloud-Betreiber kann ein Archiv verwalten, ohne die Daten sehen zu können – und ohne dass er aus der Suche etwas ableiten kann.
Der Zusammenhang mit den Matrizen der vorigen Folie, weil er die Tabelle erst verständlich macht: der Directory-Schlüssel öffnet die Directory-Matrix, also die Liste der Member. Der Index-Schlüssel öffnet die Member-Matrizen – damit sieht man, aus welchen Datensegmenten ein Member besteht, und kann über die Segmentsignaturen suchen. Erst der Datenschlüssel öffnet die Daten-Matrizen mit den Nettodaten.
Die Schlüsseltrennung ist also keine Zutat, sondern fällt aus dem Aufbau heraus: jede Schicht hat ihre eigene Matrix, und damit kann sie ihren eigenen Schlüssel haben.



Die Zahl nennen: bei Tabellenunterstützung bis zu 98 Prozent Kompression bei XML. Das ist die Zahl, die ein Fachbereich versteht.
Die Versionierung ist die zweite Überraschung: jede Änderung erzeugt eine neue Version, der vorherige Stand bleibt erhalten und muss explizit gelöscht werden – wie ein inkrementelles Backup, und adressierbar wie eine GDG.
Und der Punkt, nach dem der Betrieb fragt, sobald es um Änderungen an einem verschlüsselten Archiv geht: **es gibt Commit und Rollback.** Gibt es während der Änderung einen Fehler, wird zurückgerollt. Nur wenn alles durchläuft, wird die neue Version committed. Damit kann kein halbfertiger Zustand entstehen – kein Archiv, das halb alt und halb neu ist und bei dem niemand weiß, welche Version gilt.



Das ist die Folge der Trennung von Verwaltung und Ablage: der Speicher ist austauschbar, ohne dass sich am Archiv etwas ändert.
Für den Betrieb wichtig: bei der flachen Datei ist wahlfreies I/O nötig, auf z/OS also am besten USS-Dateien. Wer ein Archiv in einen sequentiellen Datensatz legt, verliert den wahlfreien Zugriff.
Zwei Dinge, die überraschen und die man deshalb sagen sollte:
Die flache Datei geht auch remote. Wir können ein Archiv per SSH beziehungsweise SFTP entfernt lesen, schreiben und ändern – also Vollzugriff über das Netz. Dasselbe gilt für die Ordner-Variante.
Und der **Ordner ist im Grunde die lokale Abbildung eines Object-Stores.** Wer ihn versteht, versteht damit auch, wie die Cloud-Ablagen funktionieren – und kann lokal entwickeln und testen, bevor er in die Cloud geht.
Was wir **nicht** umgesetzt haben, und warum: die **Cluster-Variante**. Die Daten auf drei Datasets zu verteilen – Directory, Index, Daten – bringt gegenüber dem Flatfile keine großen Vorteile und macht alles nur komplizierter. Deshalb gibt es für die lokale Ablage nur `FILE` oder `FOLDER`. Man könnte VSAM-KSDS als Segment-Store implementieren, aber wer will sich heute noch eine solche z/OS-Abhängigkeit einbauen.
Zum Stand: neben S3 und Azure unterstützen wir **seit 5.2.1 auch IBM CDA** (Cloud Data Access). **Google Cloud Storage nativ ist geplant und kommt mit 5.2.2.** Und wer einen anderen Speicher hat, kann ihn über die SPI selbst anbinden – `SEGUSER` und `OBJUSER` sind genau dafür da.
Die Cloud-Varianten sind der Grund, warum die drei Schlüssel wichtig sind.



Diese Folie beantwortet die Frage, die nach der Ablage-Folie kommt: wenn ein Archiv aus vielen kleinen Segmenten besteht – macht das in der Cloud nicht Tausende von Zugriffen?
Die Antwort ist: nein, und zwar wegen einer Zwischenschicht. Es gibt zwei Begriffe, und sie sind nicht dasselbe:
Der **Segment-Store** ist die Schicht, die Segmente kennt. Derzeit ist er nur in Form des Flatfiles implementiert – dort liegt jedes Segment an seiner Position in einer Datei, und man springt mit Offsets hin.
Der **Object-Store** ist die Schicht darunter für alles, was keine Offsets kann: Cloud-Speicher und die Ordner-Ablage. Und er ist im Kern eine **Pufferung**. Mehrere Segmente werden zu einem Objekt zusammengefasst, und gelesen und geschrieben wird immer das ganze Objekt.
Die Voreinstellungen: **4 MiB Datensegmente in einem Objekt, 1 MiB Membersegmente in einem Objekt** – und das Main-Segment wird ein eigenes Objekt. Die Puffergrößen sind einstellbar.
Der Zweck ist genau einer: **die Anzahl der Zugriffe über das Netz zu minimieren, damit die Latenz nicht ins Gewicht fällt.** Ein Cloud-Objektspeicher ist beim Durchsatz gut und bei der Latenz schlecht.
Wer dort jedes Segment einzeln holt, bezahlt die Latenz einmal pro Segment.
Wer 4 MiB in einem Zug holt, bezahlt sie einmal pro Objekt.
Und daraus folgt eine Betriebsregel, die man mitgeben sollte: **in der Cloud lohnen größere Puffer, lokal nicht.** Lokal ist das Flatfile mit Offsets ohnehin schneller, und ein großer Puffer liest dann nur Daten, die niemand braucht.
Der Ordner ist übrigens die lokale Abbildung genau dieses Object-Stores.
Deshalb kann man ein Cloud-Layout lokal entwickeln und testen – dieselbe Aufteilung, nur ohne Netz.

WAS DER BETRIEB WISSEN MUSS

- ein bestehendes Archiv wird nicht stillschweigend überschrieben – entweder vorher löschen oder `OVERWRITE` setzen
- **gelöschte Segmente bleiben physisch liegen** und verschwinden nur aus dem Verzeichnis
- `PRUNE` **entfernt die nicht mehr benutzten Segmente – das ist der Aufräumlauf, den ein Betrieb einplanen muss**
- das ist der Preis für Versionierung: nichts geht verloren, bis es ausdrücklich entfernt wird

Ein Archiv, in dem viel gelöscht wird, wächst – bis `PRUNE` läuft. Diesen Lauf einplanen, wie einen Reorg.

FLAMv5.21-42969

Diese Folie verhindert drei Überraschungen im Produktivbetrieb.
Erstens: ein Archiv wird nicht einfach überschrieben, es muss vorher gelöscht werden – oder man setzt OVERWRITE.
Zweitens und wichtiger: gelöschte Segmente verschwinden nur aus dem Verzeichnis, physisch bleiben sie liegen. Das Archiv wächst also, obwohl gelöscht wurde. PRUNE räumt auf.
Drittens: das ist kein Defekt, sondern der Preis für die Versionierung.

DASSELBE ÜBER DIE DREI WEGE

WEG	WAS DAMIT GEHT	WAS NICHT
Batch-Utility	`flcl ARCHIVE` – anlegen, lesen, listen, aktualisieren, prüfen, aufräumen	kein satzweiser Zugriff im Programm
Subsystem, Filesystem	transparent lesen und schreiben	keine Suche, kein gezieltes Update
API	voller Zugriff: suchen, positionieren, ändern, löschen	der Quellcode muss angefasst werden

FLAMv5.21-42969

Die wiederkehrende Folie für dieses Thema, hier auf die Frage zugeschnitten, was mit einem Archiv auf welchem Weg geht.
Die Abgrenzung, die hier oft zu kurz kommt: satzweiser Zugriff **aus dem Programm heraus** ist die Domäne des APIs. Suchen und Ändern selbst kann auch `flcl archive` im Batch, und über `FLAMSUB` geht der Vollzugriff sogar transparent – bei FLAM4 unter Bedingungen. Wer die Wege gegeneinander abwägen will, findet das in der Einheit *Die drei Zugangswege*.

STOLPERFALLEN

! **Das Archiv wächst trotz Löschen.**
Gelöschte Segmente bleiben physisch liegen. `PRUNE` einplanen.

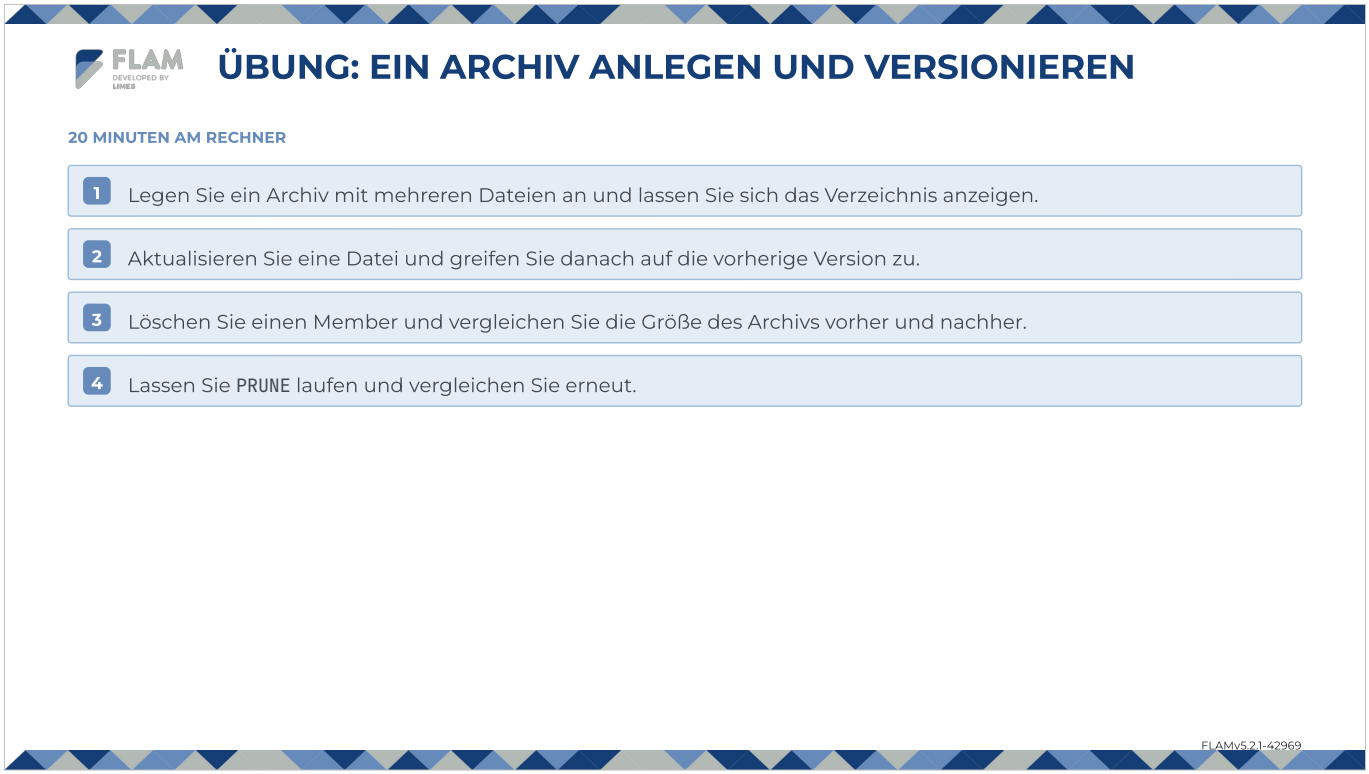
! **Ein Archiv in einer sequentiellen Datei verliert den wahlfreien Zugriff.**
Die flache Datei braucht `fseek`. Auf z/OS USS-Dateien verwenden.

! **Verketten wie bei FLAM4 funktioniert hier nicht mehr.**
FLAM5 ist ein verwaltetes Archiv mit Verzeichnis, keine Aneinanderreihung von Membern.

! **Ohne den passenden Schlüssel sieht man weniger, nicht nichts.**
Directory, Index und Daten sind getrennt verschlüsselt – das ist gewollt und muss in der Rechtevergabe abgebildet werden.

FLAMv5.21-42969

Alle vier aus der Praxis mit FLAM5-Archiven.



Zwanzig Minuten am Rechner. Der Lerneffekt liegt in Schritt drei: nach dem Löschen ist das Archiv nicht kleiner. Das muss man einmal selbst gesehen haben.
Auflösung:
1. `ARCHIVE.COMP` mit `STORE.FILE`, danach `ARCHIVE.LIST`.
2. Jede Änderung erzeugt eine neue Version. `ARCHIVE.LIST` zeigt sie; die alten Versionen bleiben lesbar – das ist gewollt und der Grund, warum ein Archiv wächst.
3. Das Archiv wird beim Löschen nicht kleiner. Erst `ARCHIVE.PRUNE` entfernt die Segmente, die zu keiner Version mehr gehören. Bei dateibasierten Archiven wird dabei das ganze Archiv neu geschrieben – also einplanen wie einen Reorg, nicht in der Mittagspause starten.
4. Die erwartete Erkenntnis: Versionierung kostet Platz, und `PRUNE` gehört in den Betriebskalender. Wer das nicht einplant, wundert sich nach einem Jahr.



Die Sammelfolie, und die, die man einem Entscheider zeigt. Fünf Punkte, und keiner davon ist bei einem gewöhnlichen Archivierer zu haben.
Erstens das Topfdesign. Jede Spalte bekommt einen Topf, und für jeden Topf ist einzeln festgelegt, wie er komprimiert, verschlüsselt und signiert wird. Man kann global arbeiten – ein Topf für alles – oder je Spalte entscheiden. Und damit minimieren: eine Spalte, die nichts Schützenswertes enthält, muss nicht verschlüsselt werden, und eine, die schon komprimiert ist, nicht noch einmal komprimiert. Das ist der Griff, mit dem man CPU spart, ohne Sicherheit aufzugeben.
Zweitens Indizierung, Primärschlüssel und Suche mit Update, Insert und Delete. Das ist zu groß für diese Einheit – es hat seinen eigenen Foliensatz, und dort steht auch, wie Bloom-Filter und Topfbildung zusammenspielen.
Drittens die Zugriffsrechte. Es gibt **limited Key Rights**: ein Schlüssel kann eingeschränkte Rechte tragen. Und wenn jemand auf eine Spalte keinen Zugriff hat, greifen die **Anonymisierungsverfahren** – er bekommt die Zeile, aber die Spalte anonymisiert oder maskiert, nicht einen Fehler.
Viertens die Trennung von Directory-, Member- und Datenschlüssel, und was daraus folgt: **Rekeying**, Rechtevergabe und die Begrenzung von Rechten. Damit richtet man **verschiedene Sichten auf dieselben Daten** ein. Zwei Beispiele, die im Raum sofort verstanden werden: die Statistikbehörde bekommt keine personenbezogenen Daten. Und für die Welt außerhalb des PCI-DSS-Geltungsbereichs sind die Kreditkartennummern maskiert – dieselben Daten, dieselbe Ablage, ein anderer Schlüssel.
Fünftens die Krypto-Infrastruktur. Heute läuft das neue Archiv über **PGP oder eine Passphrase**. Man könnte aber genauso **KMIP** sprechen oder jede andere kryptographische Infrastruktur anbinden – die Schnittstelle ist dafür gebaut.
Wenn Sie von dieser Einheit einen Satz mitnehmen: bei FLAM5 ist Sicherheit nicht eine Eigenschaft des Archivs, sondern **eine Eigenschaft jeder einzelnen Spalte**.

WAS BLEIBT

- FLAM4 hatte drei Grenzen: der MAC über das Ganze ging kaputt, der Datenschlüssel lag im Kommentarfeld, und Suchen mit Ändern brauchte drei VSAM-Cluster
- FLAM5 trennt Archivverwaltung von Segmentablage – deshalb ist der Speicher frei wählbar
- Directory, Member und Daten sind eigene Matrizen, selbst komprimiert und verschlüsselt
- drei Schlüssel, drei Sichten – Verwalten, Suchen, Lesen sind trennbar
- jede Änderung ist eine Version mit **Commit und Rollback**; `PRUNE` ist der Aufräumlauf
- je Spalte ein **Topf**: Kompression, Verschlüsselung und Signatur sind Spaltenentscheidungen

FLAMv5.21-42969

Lernziele durchgehen. Die nächste Einheit zeigt, wie der Zugriff auf einzelne Sätze in einem großen Archiv praktisch funktioniert – Bloom-Filter und Ergebnismengen.

EBENE 2 – FUNKTIONSTHEMEN: ARCHIV

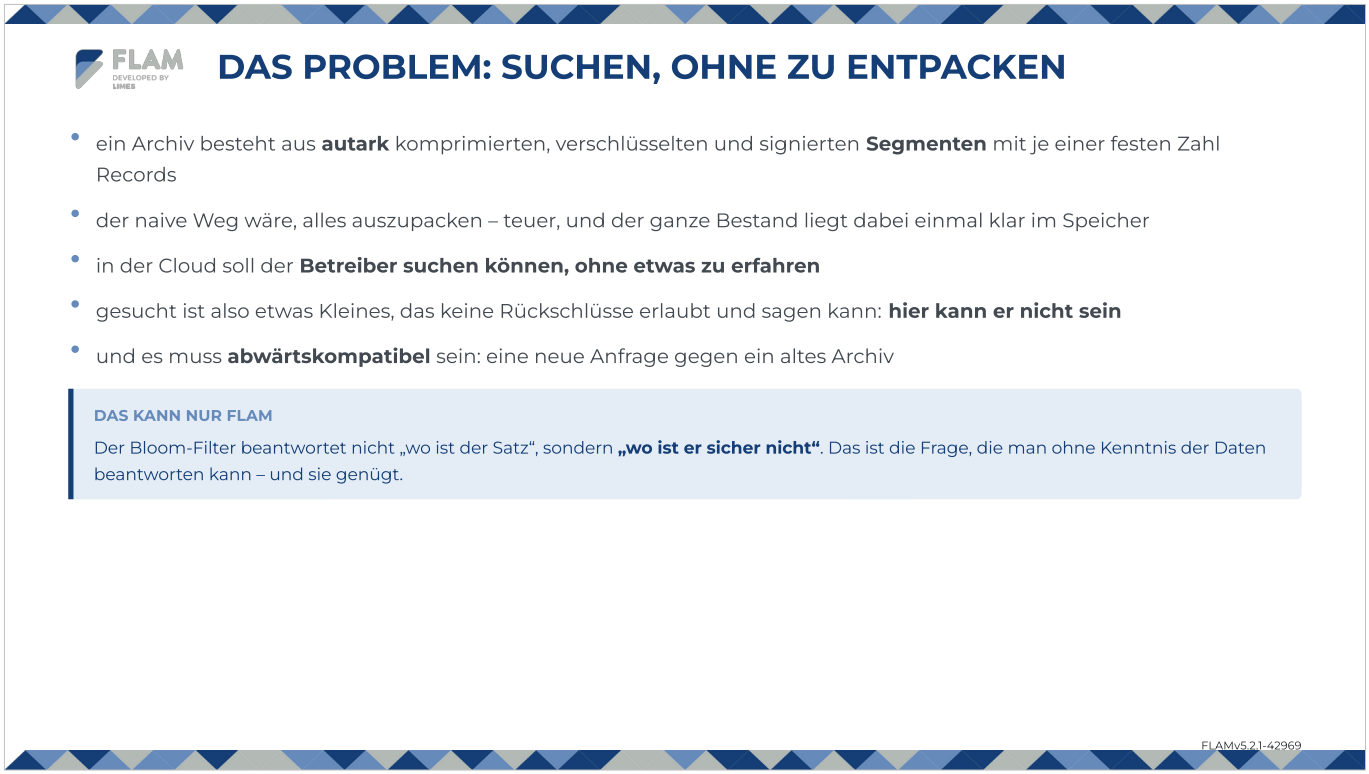
Indizierung, Filter und Suche

Wie FLAM5 in verschlüsselten Beständen sucht – und warum das wenig kostet

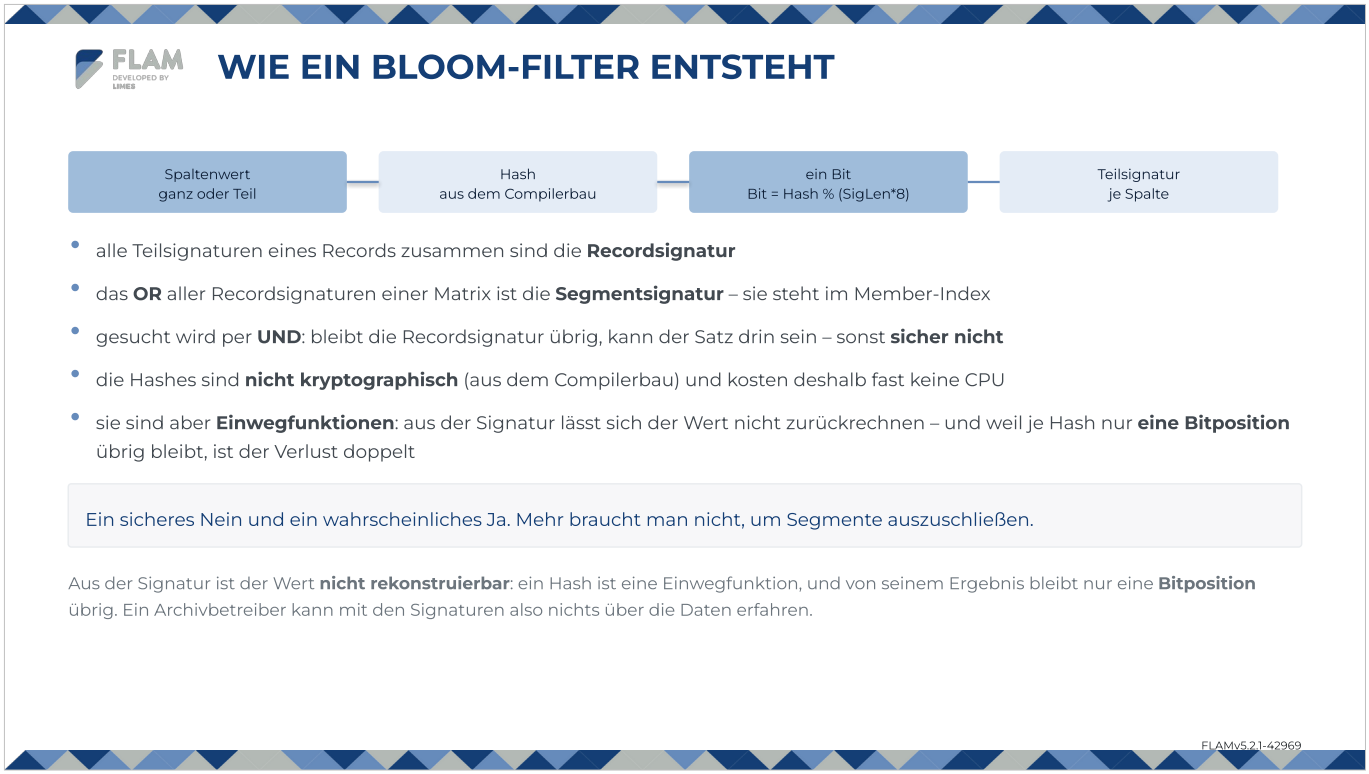
Die Einheit, die erklärt, **wie** FLAM5 das kann, was die Archiv-Einheit versprochen hat: suchen, ändern, einfügen und löschen in komprimierten, verschlüsselten und signierten Beständen – ohne die Datei klarzurechnen.
Die Antwort ist der **Bloom-Filter**: ein Index, der die Werte nicht kennt und trotzdem sagt, wo sie nicht stehen.
Wer diese Einheit verstanden hat, kann ein Archiv so entwerfen, dass die Suche billig ist. Das ist Entwurfsarbeit, keine Konfiguration – und sie zahlt sich über die Laufzeit des Bestands aus.
Was mit welcher Spalte *geschieht* – Kompression, Verschlüsselung und Signatur – ist die eigene Einheit *Topfdesign*. Hier geht es darum, wie man eine Spalte wiederfindet.
Vorausgesetzt wird die Einheit *Das FLAM5-Archiv*: Matrizen, Segmente, die drei Schlüssel.



Die Lernziele als Vertrag lesen. Am Ende der Einheit soll jeder die Entwurfsentscheidungen treffen können – welche Spalte indiziert wird und wie lang ihre Signatur sein muss.
Das vierte Ziel ist das wichtigste und wird gern unterschätzt: verstehen, warum ein Treffer im Filter noch kein Treffer im Satz ist. Wer das nicht verstanden hat, hält die false positives für einen Fehler.



Erst das Problem, dann die Lösung – sonst klingt der Bloom-Filter wie eine Spielerei.
Die Aufgabe: ein Archiv besteht aus **autark** komprimierten, verschlüsselten und signierten Segmenten. "Autark" heißt, jedes Segment lässt sich für sich allein auspacken – es trägt je eine feste Zahl von Records. Jemand sucht einen Satz. Was tut man?
Der naive Weg ist, alles auszupacken: entschlüsseln, dekomprimieren, MAC prüfen, vergleichen, wieder einpacken. Das ist bei einem Archiv mit Millionen Records nicht nur teuer – es ist auch **unsicher**, weil dabei der ganze Bestand einmal klar im Speicher lag.
Und es geht in der Cloud gar nicht: dort soll der Betreiber die Suche ausführen, ohne etwas in die Hand zu bekommen, was Rückschlüsse auf die Daten zulässt.
Also braucht man etwas, das drei Eigenschaften gleichzeitig hat: es muss klein sein, es muss **keine** Rückschlüsse auf die Daten erlauben, und es muss sagen können "in diesem Segment kann der gesuchte Satz nicht sein".
Genau das ist ein Bloom-Filter. Und die letzte Formulierung ist mit Bedacht gewählt – dazu gleich mehr.
Der zweite Grund, der in der Praxis genauso zählt: **Abwärtskompatibilität.** Eine neue Anfrage muss gegen ein altes, anders indiziertes Archiv laufen können. Sie tut es – nur mit mehr Rechenaufwand. Dazu die Folie am Ende.



Die Mechanik, und sie ist bewusst schlicht – das ist ihre Stärke.
Grundlage sind **Einweg-Hashfunktionen**, und zwar ausdrücklich **keine kryptographischen**. Es sind Hashes aus dem Compilerbau. Das ist wichtig, weil sie damit fast nichts kosten – eine kryptographische Hashfunktion über jede Spalte jedes Records wäre unbezahlbar.
Der Ablauf von unten nach oben, und den sollte man an der Tafel mitschreiben:
Für eine indizierte Spalte wird ein Hash über ihren Inhalt gerechnet – oder über einen Teil davon, dazu die nächste Folie.
Der Hashwert wird modulo der Signaturlänge in Bit gerechnet. Das Ergebnis ist eine Bitposition, und dieses **eine Bit** wird in der Teilsignatur dieser Spalte gesetzt. Man kann mehrere Hashes pro Spalte verwenden – dann werden mehrere Bits gesetzt.
Alle Teilsignaturen über alle indizierten Spalten zusammen bilden die **Signatur eines Records**.
Die Summe – ein einfaches OR – aller Recordsignaturen einer autarken Matrix bildet die **Segmentsignatur**.
Und diese Segmentsignaturen werden als Teil des **Member-Index** hinterlegt.
Das ist alles. Vier Zeilen Code, wie die nächste Folie zeigt.
Und jetzt die Suche, das ist der schöne Teil: wer nach Records sucht, bei denen eine oder mehrere Spalten einen bestimmten Wert haben, rechnet für diese Spalten die Recordsignatur – und verknüpft sie mit der Segmentsignatur per **UND**. Kommt dabei wieder die Recordsignatur heraus, ist der gesuchte Satz mit hoher Wahrscheinlichkeit in diesem Segment. Kommt etwas anderes heraus, kann er **definitiv nicht** darin sein, und das Segment wird übergangen.
Das ist die ganze Idee: ein sicheres Nein und ein wahrscheinliches Ja.

FLAM
DEVELOPED BY LIMES

VIER ZEILEN, DAS IST DER GANZE FILTER

GENERATE – EIN BIT JE HASH

```
/* generate */
h = hash(length, data);
b = h % (SigLen * 8);
Sig[b / 8] |= 0x01 << (b % 8);
```

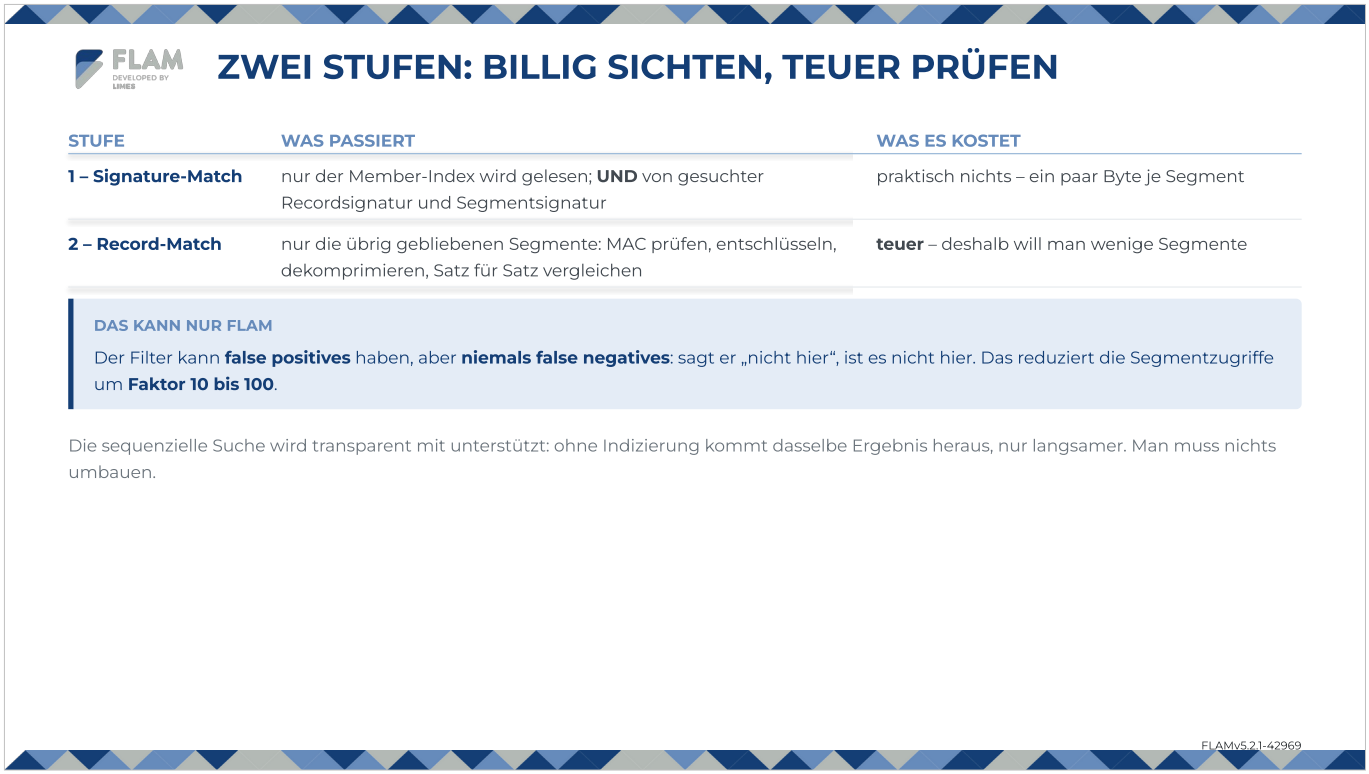
VERIFY – IST DAS BIT GESETZT?

```
/* verify */
h = hash(length, data);
b = h % (SigLen * 8);
Byte = 0x01 << (b % 8);
treffer = (Sig[b / 8] & Byte) == Byte;
```

Kein Zustand, keine Reihenfolge. Deshalb ist das Verfahren abwärtskompatibel, parallelisierbar – und Signaturen lassen sich per `OR` zusammenfassen.

FLAMv5.21-42969

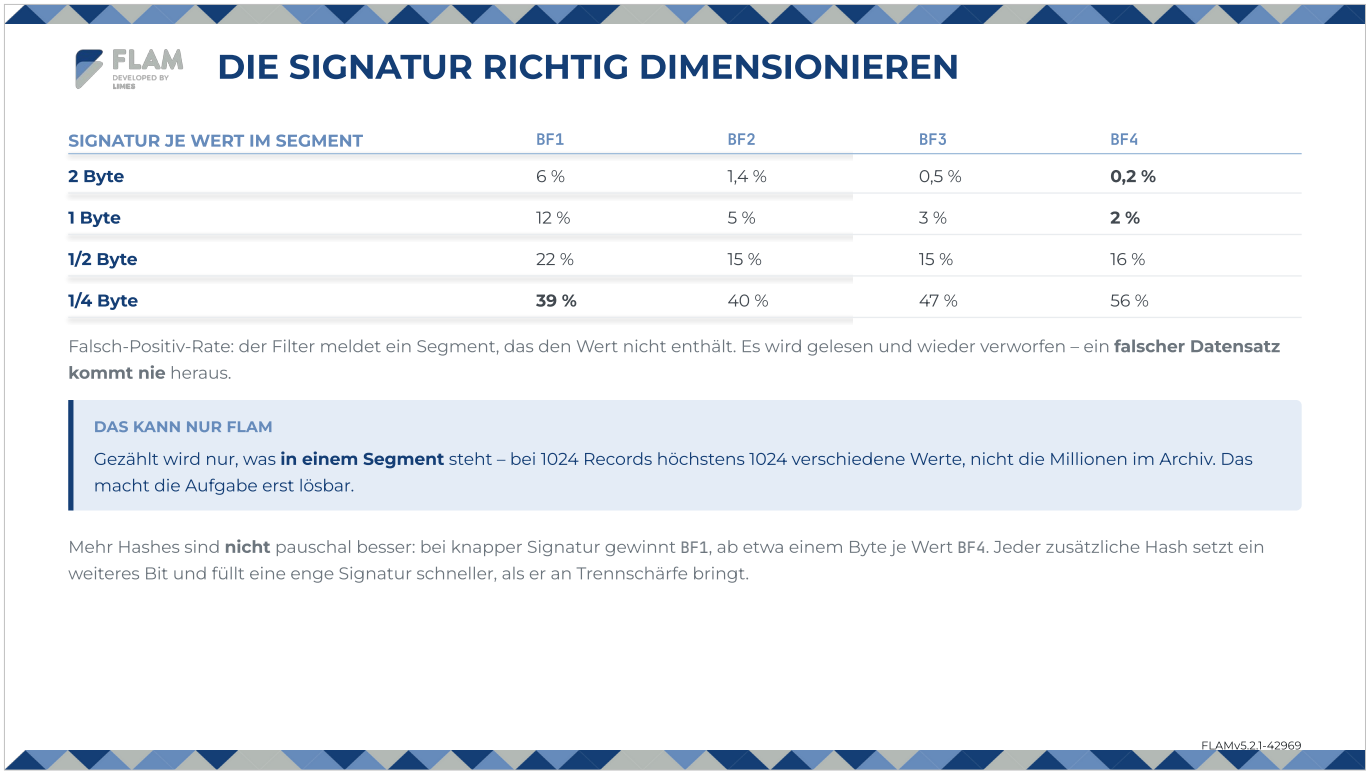
Diese Folie ist für die Entwickler im Raum, und sie wirkt: der Bloom-Filter ist kein Verfahren mit Geheimnis, er ist vier Zeilen.
Links das Erzeugen: Hash rechnen, modulo Signaturlänge in Bit, daraus Byte- und Bitoffset, und das Bit setzen. Initialisiert wird mit Nullen.
Rechts das Prüfen: derselbe Weg, nur wird am Ende gefragt, ob das Bit gesetzt ist. Wenn nicht – fertig, der Satz ist nicht in diesem Segment.
Der Punkt, den man hier machen soll: **es gibt keinen Zustand und keine Reihenfolge.** Deshalb ist das Verfahren abwärtskompatibel, deshalb lässt es sich parallelisieren, und deshalb kann man Signaturen einfach per OR zusammenfassen. Wer das einmal gesehen hat, glaubt auch, dass der Overhead klein ist.
Und weil danach immer gefragt wird: die Signaturen selbst sind gut komprimierbar. Eine Bitmaske mit wenigen gesetzten Bits komprimiert hervorragend – der Overhead im Archiv ist dadurch sehr klein.



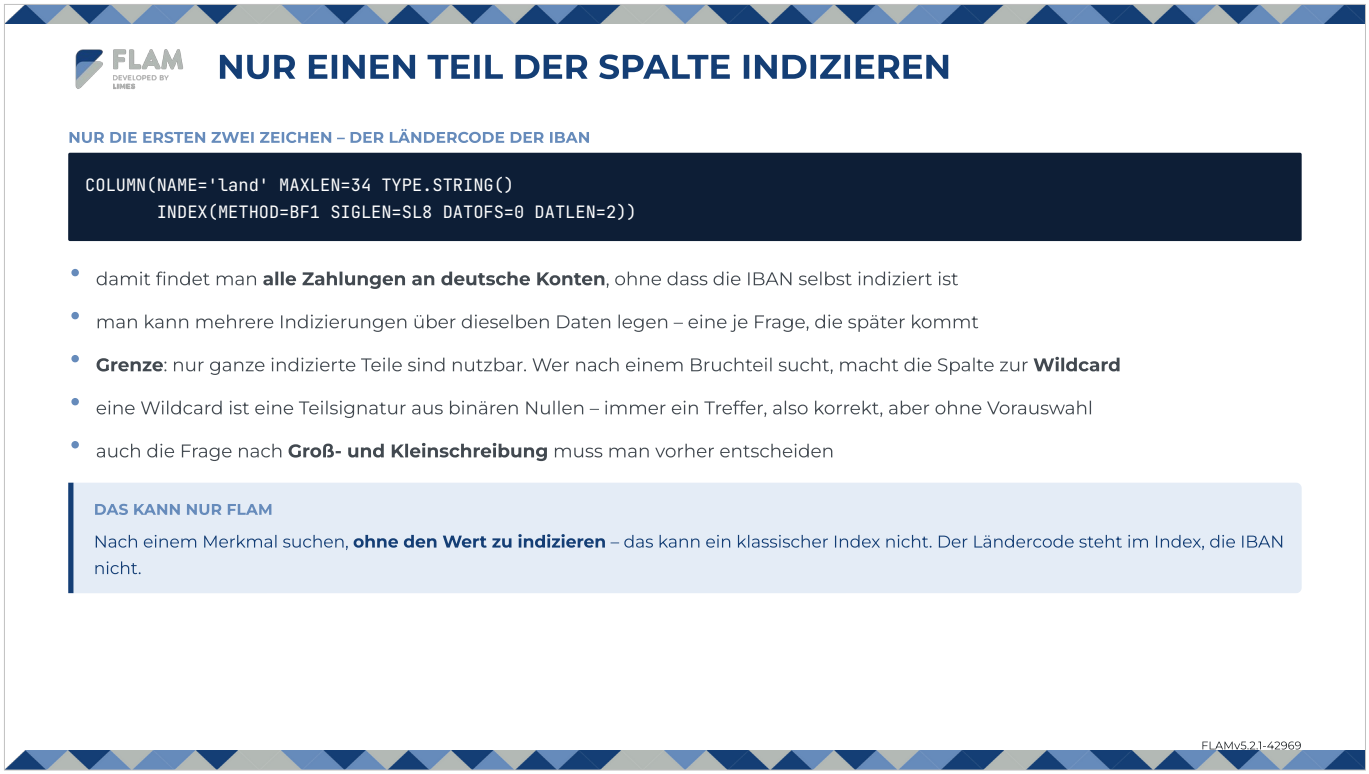
Die Folie, die den Unterschied zwischen Verstehen und Missverstehen macht.
Der Zugriff läuft in **zwei Stufen**, und sie kosten völlig unterschiedlich viel:
Stufe eins, der Signature-Match. Man liest nur den Member-Index, nimmt die Segmentsignaturen und verknüpft sie mit der gesuchten Recordsignatur.
Das ist eine UND-Operation auf ein paar Byte – praktisch kostenlos. Ergebnis ist die Liste der Segmente, die den Satz **enthalten könnten**.
Stufe zwei, der Record-Match. Nur diese Segmente werden geöffnet: MAC prüfen, entschlüsseln, dekomprimieren, und dann Satz für Satz vergleichen.
Das ist die teure Operation, und genau deshalb will man möglichst wenige Segmente dafür.
Und jetzt der Satz, den man ausdrücklich sagen muss, weil er sonst als Fehler verstanden wird: **ein Treffer in Stufe eins ist noch kein Treffer im Satz.** Ein Bloom-Filter kann false positives haben – zwei verschiedene Werte können dieselben Bits setzen. Er kann aber **keine false negatives** haben. Wenn er sagt "nicht hier", ist es nicht hier.
Das ist keine Schwäche, das ist die Bauart. Und es ist der Grund, warum Stufe zwei überhaupt existiert.
Die Wirkung in Zahlen, und die lohnt es zu nennen: **der Bloom-Filter reduziert die Segmentzugriffe um Faktor 10 bis 100.** Bei einer teuren Dekodierung je Segment ist das der Unterschied zwischen einer Suche, die läuft, und einer, die niemand bezahlt.
Und noch etwas, das die Integration einfach macht: das Verfahren unterstützt **transparent auch die sequenzielle Suche.** Wer keine Indizierung nutzt oder nutzen kann, bekommt dasselbe Ergebnis – nur langsamer. Man muss also nichts umbauen, um es einzuschalten.



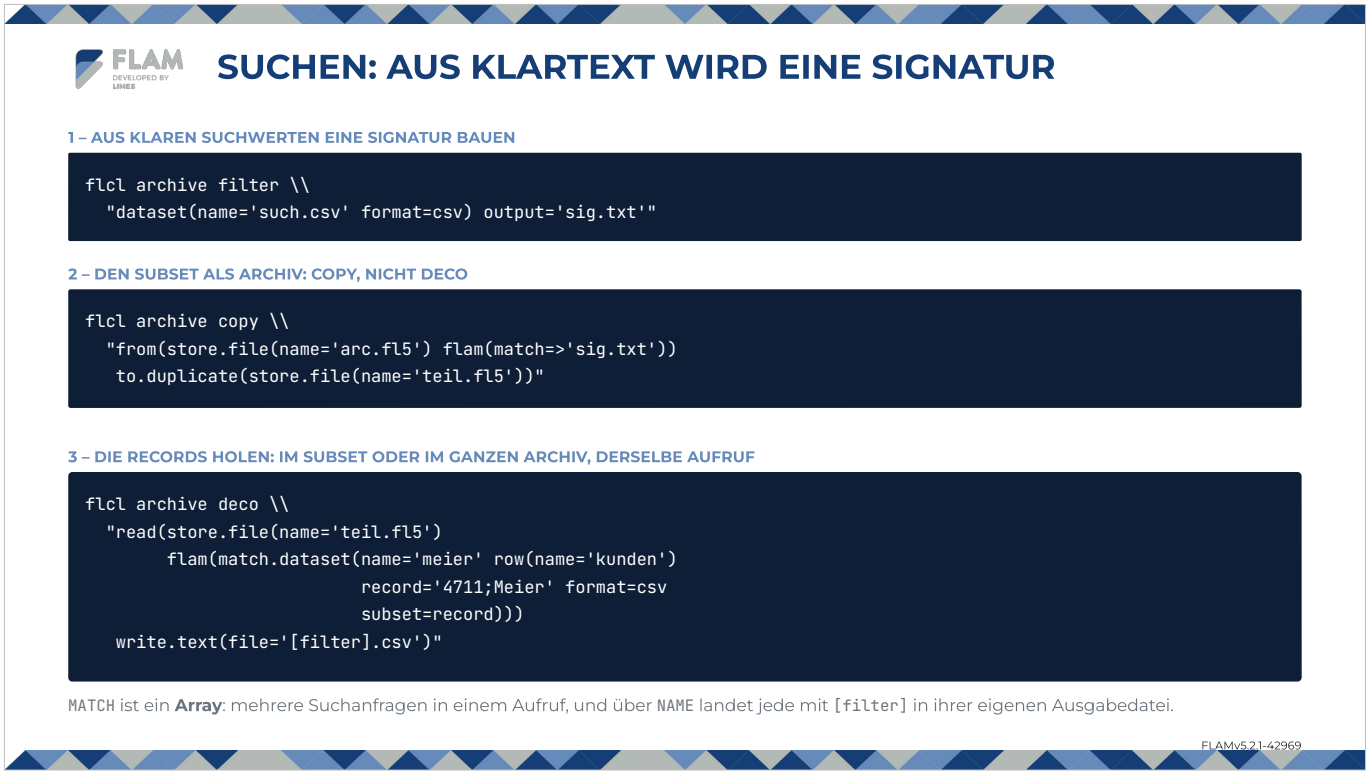
Jetzt praktisch. Die Indizierung wird **je Spalte** in der Formatbeschreibung festgelegt, im `INDEX`-Objekt der `COLUMN`. Vier Angaben, und jede hat eine klare Bedeutung:
Zuerst die gute Nachricht: **`INDEX()` allein genügt.** Seit 5.2.1 dimensioniert FLAM die Signatur selbst, wenn man sie nicht vorgibt – aus den Records je Segment, der Zahl verschiedener Werte und der zugelassenen Falsch-Positiv-Rate. Man muss also keine Bloom-Filter-Arithmetik treiben, um anzufangen.
`METHOD` ist die Anzahl der Hashes: `BF1` bis `BF4`. Gibt man nichts an, wählt FLAM `BF4` und die passende Länge dazu; gibt man nur eines von beiden an, wird das andere dazu passend gewählt. Mehr Hashes heißt mehr gesetzte Bits je Wert und damit eine bessere Trefferrate, aber auch eine schneller volllaufende Signatur.
`SIGLEN` ist die Länge der Teilsignatur für diese Spalte, **in Byte**. Es gibt vorgefertigte Werte von `SL1` bis `SL8192`, man kann aber auch eine Zahl angeben – aufgerundet auf die nächste Zweierpotenz, mindestens 4 Byte.
`DISTINCT` ist die einzige Angabe, die man in der Praxis wirklich machen sollte: die obere Grenze der verschiedenen Werte dieser Spalte. Ohne sie nimmt FLAM an, dass **jeder** Wert verschieden sein kann, und dimensioniert entsprechend groß. Gemessen mit 3000 Zahlungen, 1024 Records je Segment: die IBAN-Spalte bekommt `BF4` mit 1024 Byte, die Währungsspalte mit `DISTINCT=3` nur 4 Byte.
`FPRATE` ist die zugelassene Falsch-Positiv-Rate in Promille, Vorbelegung 50, also 5 Prozent. Ein Member mit vielen Segmenten braucht eine kleinere Rate: bei 100000 Segmenten lesen 50 Promille 5000 Segmente je gesuchtem Wert umsonst.
`DATOFS` und `DATLEN` grenzen ein, **über welchen Teil** der Spalte gehasht wird. Das ist die Teilindizierung, und sie hat eine eigene Folie.
`PRIKEY` erklärt die Spalte zum Teil des **Primärschlüssels**. Auch dazu gleich mehr.
Der Entwurfssatz für den Raum: **die Anzahl der verschiedenen Werte je Segment bestimmt, wie viele Hashes und wie lang die Teilsignatur sein sollen.** Auf das "je Segment" kommt es an: eine Signatur gilt für ein Segment, und darin stehen in der Vorbelegung 1024 Records. Millionen verschiedener IBANs im Archiv sind also nicht die Rechengrundlage – maximal 1024 verschiedene je Segment sind es.
Eine Spalte mit fünf möglichen Werten braucht ein Byte. Eine Spalte, in der jeder Record einen eigenen Wert hat, braucht das Tausendfache. Wie genau, steht auf der nächsten Folie – und das ist die Folie, die man mitnehmen sollte, denn eine zu kleine Signatur ist der häufigste Entwurfsfehler.



Die Folie, die einen Entwurfsfehler verhindert, den man nachträglich nicht mehr reparieren kann – eine zu kleine Signatur.
Die Mechanik in einem Satz: die Signatur hat **8 mal SIGLEN Bitpositionen**, und jeder Wert setzt so viele Bits, wie die Methode Hashes hat. Wenn mehr Werte hineingeschrieben werden als Bitpositionen da sind, laufen alle Bits voll und der Filter sagt zu jeder Suche "vielleicht". Er ist dann nicht falsch – er nützt nur nichts mehr, und die Suche kostet wieder so viel wie ohne Index.
Deshalb die Merkregel in der Tabelle: **ein Byte Signatur je unterschiedlichem Wert im Segment gibt rund 2 Prozent Falsch-Positive, zwei Byte rund 0,3 Prozent, ein halbes Byte schon 16 Prozent.** Das ist die ganze Dimensionierung; die Formel dahinter ist die Standardformel für Bloom-Filter und steht in unserem Handbuch am `INDEX`-Objekt.
Zwei Dinge, die dabei überraschen:
Erstens zählt nur, was im Segment steht. Nicht die Millionen IBANs im Archiv, sondern die 1024 Records eines Segments – also höchstens 1024 verschiedene Werte. Das macht die Aufgabe erst lösbar.
Zweitens sind mehr Hashes nicht pauschal besser, und das ist die Aussage, die dem Bauchgefühl widerspricht. Bei knapper Signatur ist `BF1` besser als `BF4`: ein halbes Byte je Wert gibt mit `BF1` 22 Prozent, mit `BF4` aber 16 – und bei einem Viertel Byte kippt es, da ist `BF1` mit 39 Prozent besser als `BF4` mit 56. Der Grund: jeder zusätzliche Hash setzt ein weiteres Bit, und wenn die Signatur eng ist, füllt man sie damit schneller, als man an Trennschärfe gewinnt. Ab etwa einem Byte je Wert ist `BF4` eindeutig die richtige Wahl.
Und die Kosten, damit die Zahl im Raum steht: 1024 IBANs mit `BF4` und `SL2048` sind 2 KiB Signatur je Segment gegen etwa 34 KiB Nutzdaten in dieser Spalte – rund 6 Prozent. Bei mehreren indizierten Spalten addiert sich das, denn jede Spalte hat ihre eigene Teilsignatur.
Der Satz zum Mitnehmen: **die Zahl, die man kennen muss, ist nicht die Signaturlänge, sondern die Anzahl verschiedener Werte je Segment.** Die Länge rechnet FLAM daraus; sie wird beim ersten Segment der Zeile festgelegt und steht danach im Segmentkopf – `LIST` und `INFO` zeigen sie, und man kann sie in die Formatbeschreibung übernehmen, um sie festzuschreiben. Jedes weitere Member derselben Zeile übernimmt die Indizierung des Archivs, auch mit anderer Satzzahl je Segment: eine Suche arbeitet je Formatbeschreibung mit einer Indizierung.
Und die Grenze, die man im Betrieb sieht: 8192 Byte. Reicht das für die angenommene Zahl verschiedener Werte nicht, **schreibt FLAM eine Warnung** und nennt die drei Auswege – einen Teil des Wertes indizieren (`DATOFS`/`DATLEN`), weniger Records je Segment (`MAXREC`) oder `DISTINCT` angeben. Gemessen: ohne `MAXREC` bestimmte FLAM 20343 Records je Segment, beide Spalten landeten bei 8192 Byte und es kam genau diese Warnung; mit `MAXREC=1024` und `DISTINCT=3` für die Währung waren es 1024 und 4 Byte ohne Warnung.



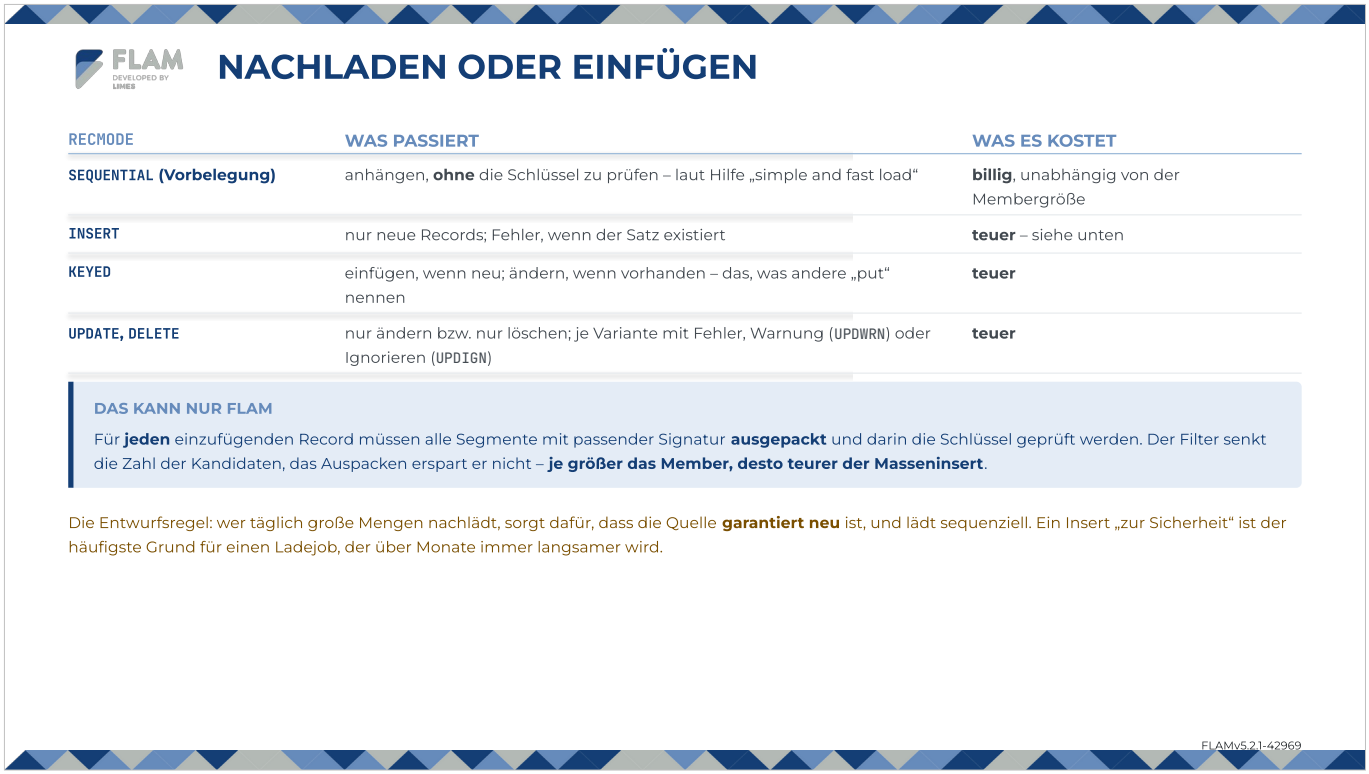
Die Folie mit dem Aha-Effekt, und sie ist zugleich das schönste Beispiel für den Datenschutz-Aspekt.
Mit `DATOFS` und `DATLEN` bestimmt man, über welchen Teil einer Spalte gehasht wird. Und damit kann man etwas, das mit einem klassischen Index nicht geht: **nach einem Merkmal suchen, ohne den Wert selbst zu indizieren.** Das Beispiel: eine IBAN. Die ersten zwei Zeichen sind der Ländercode. Wer `DATOFS=0 DATLEN=2` setzt, indiziert nur diesen Ländercode. Danach kann man alle Zahlungen an deutsche Konten finden – ohne dass die IBAN selbst in irgendeiner Form im Index steht.
Und man kann beides gleichzeitig: eine Spalte auf den Ländercode indizieren und eine zweite Indizierung derselben Daten über die ganze IBAN legen. Dann sucht man je nach Frage über die eine oder die andere.
Jetzt die Grenze, und die muss man ehrlich nennen, weil sie im Entwurf wehtut: **es lassen sich nur ganze indizierte Teile nutzen.** Wer nach einem Bruchteil einer indizierten Spalte sucht – also nach etwas, das nicht genau dem indizierten Bereich entspricht – für den wird diese Spalte zur **Wildcard**. Die Teilsignatur ist dann binär null, und binär null ist immer ein Treffer. Das ist korrekt, kostet aber die Vorauswahl.
Genau deshalb muss man beim Entwurf wissen, **wonach später gesucht wird.** Das ist die unangenehmste Eigenschaft des Verfahrens, und sie steht auf der Nachteile-Folie noch einmal.
Dasselbe gilt für Groß- und Kleinschreibung: wer später ohne Rücksicht auf Groß- und Kleinschreibung vergleichen will, muss das **vorher** wissen und die Indizierung entsprechend anlegen.



Wie eine Suche praktisch abläuft, in drei Schritten – und der erste ist der interessante.
`ARCHIVE FILTER` erzeugt aus einem klaren Datensatz – am besten CSV – eine **Signatur, die keine Rückschlüsse auf die Daten zulässt.** Das ist das Kommando, mit dem man die Suchanfrage vorbereitet. Man gibt die gesuchten Werte im Klartext ein und bekommt eine Bitmaske heraus.
Und jetzt der Punkt, warum das ein eigenes Kommando ist und nicht nur ein Parameter: **die Signatur kann man verschicken.** Man baut die Anfrage dort, wo man die Daten kennen darf, und schickt nur die Signatur an den Betreiber des Archivs. Der sucht damit, ohne zu wissen, wonach.
Damit die Anfrage zur Indizierung des Archivs passt, braucht sie deren Beschreibung. Dafür gibt es `ROWLIST`: **die Liste der Formatbeschreibungen eines Archivs lässt sich mit `LIST` erzeugen** und dann als `ROWLIST` angeben. Man muss die Indizierung also nicht kennen – man holt sie sich.
Gibt man die Indizierung stattdessen im `FILTER`-Objekt selbst an, wird sie auf Plausibilität geprüft.
Zweiter Schritt: `MATCH`. Beim Lesen gibt man die Signatur über `MATCH.FILTER` mit – am einfachsten, indem man die vom `FILTER`-Kommando erzeugte Parameterdatei einliest: `match=>'sig.txt'`. Oder man gibt gleich den klaren Suchdatensatz über `MATCH.DATASET` an; dann rechnet FLAM die Signatur selbst.
Wichtig für den zweiten Aufruf: **ein `DECO` liefert immer klare Daten.** Wer einen Subset als **Archiv** haben will, nimmt deshalb `COPY` mit `TO.DUPLICATE` – dann werden nur die getroffenen Segmente kopiert, und niemand hat die Daten gelesen.
Und `MATCH` ist ein **Array**: man kann beliebig viele Suchanfragen angeben, `FILTER` und `DATASET` gemischt. Gibt man ihnen über `NAME` verschiedene Namen, kann man diesen Namen im Input-Output-Name-Mapping mit `[filter]` verwenden – dann schreibt ein Aufruf die Treffer jeder Anfrage in ihre eigene Datei.
Dritter Schritt: `SUBSET`. Damit sagt man, was man zurückhaben will:
`SEGMENT` liefert die vollständigen Segmente, `RECORD` nur die passenden Sätze. `AUTO` entscheidet nach Kommando – Segmente beim `COPY`, Records beim `DECO`. Für eine Ergebnismenge, die weitergegeben wird, ist `SEGMENT` oft richtig: der Empfänger bekommt dann geschlossene, integritätsgeschützte Einheiten.
Und der Zusammenhang mit den Schlüsseln aus der Archiv-Einheit: für Stufe eins braucht man nur den Index-Schlüssel. Erst Stufe zwei braucht den Datenschlüssel. Deshalb kann ein Betreiber suchen, ohne lesen zu können.
Der mittlere Aufruf ist der Fall, der das Verfahren erst verständlich macht:
ein `COPY` mit dem Filter, das die getroffenen Segmente in ein neues Archiv kopiert. Ergebnis ist ein kleineres Archiv – geschlossene, integritätsgeschützte Segmente, ohne dass jemand die Daten gelesen hat.
Und jetzt der Punkt, den die Teilnehmer mitnehmen sollen: der dritte Schritt ist **derselbe Aufruf**, egal ob er auf den gefilterten Subset oder auf das ganze Archiv geht. Die Suche kennt keinen Unterschied. Der Subset ist also kein anderer Weg, sondern nur ein kleinerer Bestand – man kann



Drei Dinge, die auf der Indizierung aufsitzen und im Entwurf mitentschieden werden.
Der Primärschlüssel. Mit `PRIKEY` erklärt man eine indizierte Spalte zum Teil des Primärschlüssels. Mehrere Spalten können das gemeinsam sein. Damit hat ein Archiv einen fachlichen Schlüssel, über den `UPDATE` und `DELETE` gezielt arbeiten – nicht über eine Position, sondern über den Inhalt.
Die Eindeutigkeit. `LOOKUP` baut die Hashtabelle und stellt dabei sicher, dass ein Record eindeutig ist. Wer einen fachlichen Schlüssel hat und Duplikate ausschließen will, setzt das.
Und die Einordnung, die man dazu sagen sollte: **`LOOKUP` brauchen vor allem wir selbst** – für die Directory-Matrix. Für einen Kunden lohnt es nur, wenn er **mehrfach in derselben Matrix** sucht, und das heißt in der Praxis: der Inhalt eines ganzen Members liegt in einer Matrix. Dafür gibt es Anwendungen, aber sie sind selten. Von den Matrixgrenzen merkt ein Anwender sonst nichts, die Berechnung der Tabelle kostet, und mit der nächsten Matrix wird sie verworfen. Die Eindeutigkeitsprüfung ist der Grund, es trotzdem zu setzen – nicht die Suchgeschwindigkeit.
Die Rechte. Und hier kommt der Bogen zur Anonymisierung aus der Archiv-Einheit: mit `NOTMTD` legt man fest, **was passiert, wenn jemand auf eine Spalte keinen Zugriff hat.** Vier Verfahren: `ZERO` liefert Nullen – die Vorbelegung. `MASK` maskiert. `RAND` liefert Zufallswerte. Und `IBAN` erzeugt eine formal gültige IBAN, die aber nicht die echte ist.
Das letzte ist der interessante Fall, und man sollte ihn erklären: eine Testumgebung oder eine Statistikauswertung braucht Daten, die **die Formatprüfung überleben**. Nullen tun das nicht – eine Anwendung, die eine IBAN prüft, bricht ab. Eine erzeugte, formal gültige IBAN geht durch. Damit richtet man verschiedene Sichten auf dieselben Daten ein, ohne die Daten zu kopieren: die Statistikbehörde bekommt keinen Personenbezug, die Testumgebung bekommt plausible Werte, und die berechtigte Anwendung bekommt das Original. Ein Bestand, drei Sichten, entschieden über den Schlüssel.
Zwei Schalter stehen der Vollständigkeit halber mit in der Tabelle, weil sie zur Spaltenentscheidung gehören:
`LITERAL` verlagert die Werte in die **globale Literal-Matrix**, statt sie in den zugewiesenen Topf zu legen. Das lohnt bei Spalten mit wenigen Schlüsselwörtern. Aber Vorsicht bei der Sicherheit: die Literal-Matrix wird mit dem **Directory-Schlüssel** geschützt. Wer zwischen Verzeichnis- und Datenzugriff sicherheitstechnisch unterscheiden will, darf den Schalter **nicht** setzen – auch dann nicht, wenn die Werte Literale sind.
`NOCASE` vergleicht ohne Rücksicht auf Groß- und Kleinschreibung. Auch das ist eine Entscheidung beim **Anlegen** des Archivs, nicht bei der Suche.



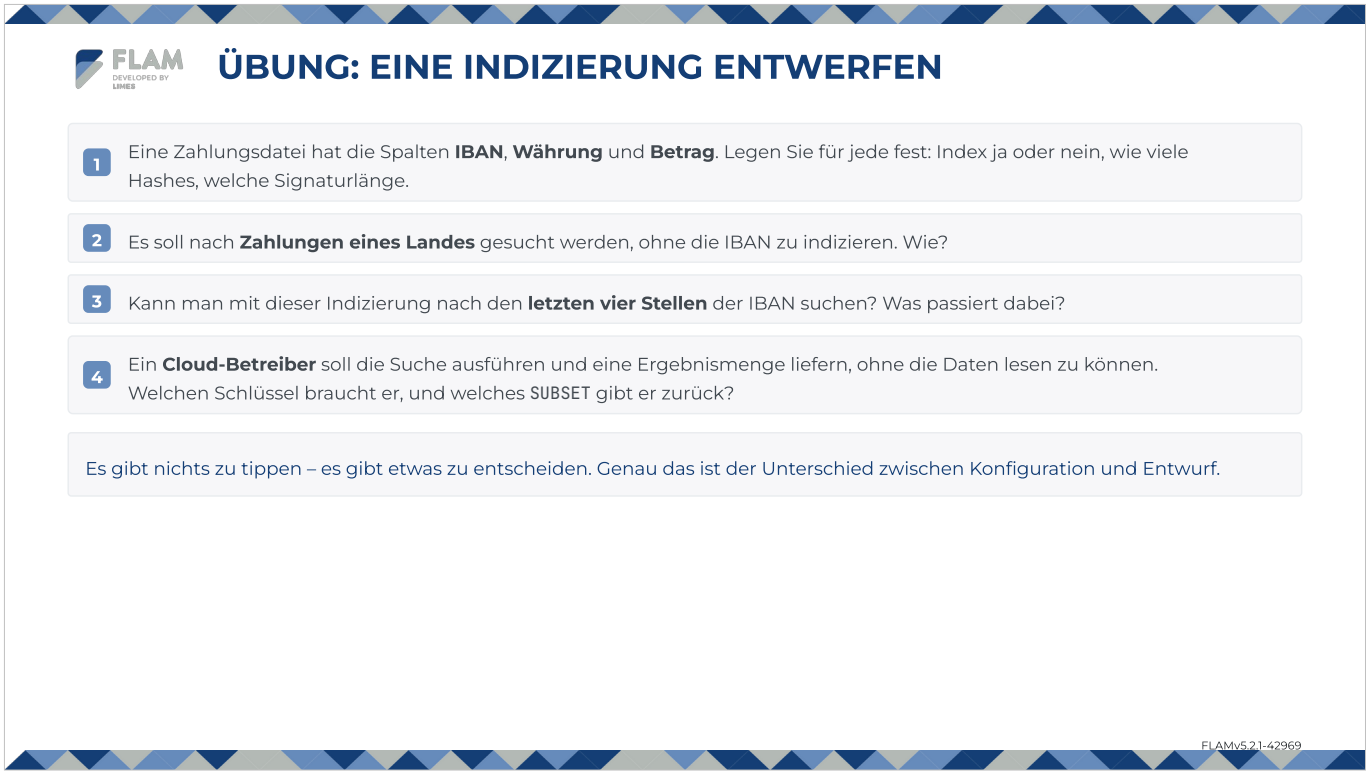
Die Folie, die einen Ladejob um Stunden verkürzen kann – und sie gehört hierher, weil auch das Einfügen über den **Primärschlüssel** geht.
Die Frage ist immer dieselbe: **weiß ich, dass diese Records im Member noch nicht vorkommen? Wenn ja, wird sequenziell nachgeladen.** `RECMODE=SEQUENTIAL` ist die Vorbelegung, und die Hilfe nennt es so, wie es ist: "no verification of keys (simple and fast load)". Es wird angehängt, nichts geprüft. Das ist der Normalfall für einen täglichen Zulauf, bei dem die Quelle schon disjunkt ist – etwa Tagesbuchungen, die es vorher nicht gab.
Wenn nein, muss man einfügen. `RECMODE=INSERT` fügt nur neue Records ein und meldet einen Fehler, wenn der Satz schon da ist. `KEYED` ist das, was andere Systeme "put" nennen: einfügen, wenn neu, ändern, wenn vorhanden.
Dazu gibt es `UPDATE` und `DELETE` sowie zu jedem die Varianten mit Warnung (`INSWRN`) oder Ignorieren (`INSIGN`) statt Fehler – die Fehlerbehandlung steckt also im Schlüsselwort selbst.
Und jetzt der Preis, und der ist der Grund für diese Folie: **ein Insert ist teuer.** Für jeden einzufügenden Record müssen alle Segmente, deren Signatur passt, **ausgepackt** und darin jeder Record geprüft werden, ob der Primärschlüssel schon vorkommt. Das heißt: **je größer das Member, desto teurer ein Masseninsert.** Der Bloom-Filter hilft, aber er reduziert nur die Zahl der Kandidaten – er erspart das Auspacken nicht.
Die Konsequenz für den Entwurf: wer täglich Millionen Sätze nachlädt, sollte dafür sorgen, dass die Quelle **garantiert neu** ist, und sequenziell laden.
Ein Masseninsert "zur Sicherheit" ist der häufigste Grund für einen Ladejob, der über Monate immer langsamer wird.



Die ehrliche Bilanz, und sie gehört dazu. Ein Verfahren, dessen Grenzen man nennt, wird geglaubt.
Die Vorteile der Reihe nach: die Signaturen komprimieren gut, der Overhead ist deshalb sehr klein. Sie lassen **keine Rückschlüsse auf die Daten** zu – wer nur die Signatur hat, erfährt nichts. Passt die Indizierung nicht zur Frage, wird die Spalte zur Wildcard und die Suche funktioniert trotzdem. Die Segmentzugriffe sinken um Faktor 10 bis 100. Die sequenzielle Suche wird transparent mitunterstützt. Und die Abwärtskompatibilität: **dieselbe Anfrage gegen unterschiedlich indizierte Archive führt immer zum gleichen Ergebnis** – nur der Rechenaufwand unterscheidet sich, und er ist um so geringer, je mehr Indizes genutzt werden können. Das ist die Eigenschaft, die Langzeitarchivierung erst möglich macht.
Die Nachteile, und die sind alle vom gleichen Typ – man muss vorher nachdenken: Die Signaturen selbst lassen sich nicht indizieren, sie werden sequenziell verglichen. Die Indizierung muss **im Vorfeld** definiert werden. Groß- und Kleinschreibung muss man vorher entscheiden. Wer später nur nach nicht indizierten Spalten sucht, muss alles auspacken. Und nur ganze indizierte Teile sind nutzbar.
Und dann die Alternative, die man kennen soll, um die Wahl zu verstehen:
die echte Datenindizierung. Ihr Vorteil ist die Freitextsuche – man kann nach allem suchen, auch nach dem, was man beim Anlegen nicht bedacht hat. Ihre Nachteile sind aber genau die, wegen der wir sie nicht genommen haben: die Datenmenge der indizierten Spalten **verdoppelt** sich, im Index stehen die **klaren Daten** – also unsicher, und bei einem Subset sind die sensiblen Nettodaten sichtbar – und der Speicherbedarf für das Halten des Index ist viel größer.
Der Abwägungssatz: **wer Freitextsuche braucht, braucht eine Suchmaschine.**
Wer einen geschützten Bestand durchsuchen will, braucht Bloom-Filter. **Und die Grenze, die im Betrieb wirklich zuschlägt, steht unten auf der Folie:** die Größe eines Members ist durch den Index begrenzt. **Die Member-Matrix mit den Segmentsignaturen muss neben dem Directory in den Speicher passen. Ein Kunde von uns arbeitet mit 4-KiB-Signaturen, hält 80 Millionen Records in einem Member und lässt nur 128 Records je Matrix zu – damit passt der Index auf z/OS im 31-Bit-Adressraum nicht mehr, und das ist der Grund, warum wir auf z/OS den 64-Bit-Support bringen.**
An dem Fall kann man das Entwerfen zeigen: 128 Records bei einer 4-KiB-Signatur sind viel zu wenig. Entweder verkleinert man die Signaturlänge, oder – besser – erhöht die Zahl der Records je Matrix auf 1024. Bei wirklich großen Membern kann es trotzdem passieren, dass der Index nicht in den Speicher geht.



Alle aus dem Entwurf, nicht aus dem Betrieb – diese Einheit macht Fehler, die man später nicht mehr billig korrigiert.
Zur ersten: eine zu kurze Teilsignatur bei einer Spalte mit vielen verschiedenen Werten läuft voll. Sind alle Bits gesetzt, ist jede Anfrage ein Treffer, und der Filter hat keine Wirkung mehr – er kostet dann nur noch. Deshalb ist die Anzahl der verschiedenen Werte die Größe, an der man die Signaturlänge ausrichtet.
Zur zweiten: umgekehrt hilft eine sehr lange Signatur bei einer Spalte mit fünf Werten nichts – sie kostet nur Platz. Für ein Währungskennzeichen ist `LITERAL` der bessere Griff als eine feine Indizierung.
Zur dritten: die Indizierung ist eine Entwurfsentscheidung. Nachträglich ändern heißt, das Archiv neu zu schreiben. Deshalb lohnt es, vorher mit den Fachleuten zu sprechen, wonach gesucht werden wird.
Zur vierten: ein Treffer ist ein Kandidat. Wer die false positives für Fehler hält, misstraut dem Verfahren zu Unrecht – und wer sie ignoriert, baut eine Anwendung, die zu viele Sätze liefert.



Zwanzig Minuten, und die Aufgabe ist bewusst eine Entwurfsaufgabe – es gibt nichts zu tippen, es gibt etwas zu entscheiden.
Zur Lösung, wenn die Gruppe soweit ist:
Aufgabe 1. Die IBAN hat Millionen verschiedene Werte – sie braucht eine lange Teilsignatur und mehr als einen Hash, also etwa `SL64` und `BF2`. Das Währungskennzeichen hat drei bis fünf Werte: `SL8` und `BF1` genügen, oder besser gleich `LITERAL` ohne Indizierung. Der Betrag wird selten gesucht – kein Index.
Aufgabe 2. Der Ländercode: `DATOFS=0 DATLEN=2` auf einer eigenen indizierten Spalte. Damit findet man alle Zahlungen eines Landes, ohne die IBAN zu indizieren.
Aufgabe 3. Nein – und das ist der Kern. Wer nach den letzten vier Stellen der IBAN sucht, trifft nicht den indizierten Bereich. Die Spalte wird zur Wildcard, die Suche funktioniert trotzdem, aber ohne Vorauswahl:
alle Segmente müssen ausgepackt werden. Die Antwort lautet also: es geht, aber es kostet den ganzen Vorteil. Wer das braucht, indiziert die letzten vier Stellen zusätzlich.
Aufgabe 4. Der Betreiber braucht den **Index-Schlüssel**, nicht den Datenschlüssel. Er kann dann Stufe eins ausführen und die Kandidaten zurückgeben – die Ergebnismenge als Segmente mit `SUBSET=SEGMENT`. Lesen kann er sie nicht. Das ist genau das Cloud-Szenario aus der Archiv-Einheit.

WAS BLEIBT

- der Bloom-Filter sagt **sicher nein** und **wahrscheinlich ja** – deshalb genügt er, ohne die Daten zu kennen
- zwei Stufen: Signature-Match ist praktisch kostenlos, Record-Match ist teuer – **Faktor 10 bis 100** weniger Segmente
- `INDEX()` genügt – FLAM dimensioniert die Signatur selbst; die Angabe, die zählt, ist `DISTINCT`
- je Spalte entworfen wird weiter: `DATOFS`/`DATLEN` für den Teil des Wertes, `PRIKEY` für den Schlüssel, `FPRATE` für die Rate
- **Teilindizierung** erlaubt die Suche nach einem Merkmal, ohne den Wert zu indizieren – der Ländercode ohne die IBAN
- `LITERAL` für Spalten mit wenigen Werten, `LOOKUP` für die Eindeutigkeit, `NOTMTD` für die Sicht ohne Zugriffsrecht
- dieselbe Anfrage liefert gegen jedes Archiv dasselbe Ergebnis – nur der Aufwand unterscheidet sich

FLAMv5.21-42969

Lernziele durchgehen. Der Satz, der hängen bleiben soll, ist der erste: ein sicheres Nein und ein wahrscheinliches Ja.
Danach folgt die Einheit *Verwaltung archivierter Daten*, in der genau diese Mechanik in `ARCHIVE`-Kommandos benutzt wird – Suche, Update, Insert, Delete und die Prüftiefen von `VERIFY`.

FLAM SCHULUNG, EBENE 2

Suchen und ändern, ohne auszupacken

Bloom-Filter, Ergebnismengen und Versionen

Falk Reichbott · limes datentechnik gmbh

Zweites Funktionsthema und die Einheit, die aus einem Archiv einen Datenbestand macht: Suchen, Einfügen, Ändern, Löschen – im komprimierten und verschlüsselten Bestand.
Der Kern ist der zweistufige Zugriff: eine billige Vorauswahl über Signaturen, und erst danach die teure Dekodierung. Wer das verstanden hat, versteht, warum das überhaupt bezahlbar ist.



Ziel zwei ist das eigentliche Lernziel. Ziel vier ist das, was in Cloud-Gesprächen zählt.

WAS AUF EINEM ARCHIV MÖGLICH IST

OPERATION	GRANULARITÄT	ANMERKUNG
Suchen	Spalten eines Satzes	liefert eine Ergebnismenge, keine Datei
Lesen	einzelner Satz	über Schlüssel oder Position
Ändern	Spalten eines Satzes	der Satz wird an seiner Stelle ersetzt
Einfügen, Löschen	ganzer Satz	Löschen entfernt aus dem Verzeichnis, nicht physisch

Alles davon **ohne die gesamte Datei klarzurechnen und wieder einzupacken** – und die Bestände bleiben über die Zugriffsmethoden ansprechbar, die Ihre Anwendung schon benutzt (auf dem Host VSAM-RRDS, KSDS).

FLAMv5.21-42969

Die vier Operationen mit ihrer jeweiligen Granularität. Der Unterschied zwischen Satz und Spalte ist wichtig: Einfügen und Löschen treffen einen Satz, Ändern und Suchen treffen Spalten.
Der Satz, der bleiben soll: alles das, ohne die gesamte Datei klarzurechnen und wieder einzupacken. Das ist der Unterschied zu jedem Archivierer.
Und der Migrationspfad für bestehende Anwendungen: die Bestände bleiben über die Zugriffsmethoden ansprechbar, die die Anwendung schon benutzt – auf dem Host VSAM-RRDS und KSDS.
Wenn jemand nach der Cloud fragt, hier die saubere Abgrenzung, weil sie oft vermischt wird. Was heute geht, ist der Segment-Store in S3, Azure und IBM CDA – die Ablage liegt in der Wolke, der Zugriff kommt von hier. Was noch nicht geht, ist der umgekehrte Weg: eine Cloud-Anwendung, die über ein Standard-API auf ein geöffnetes Member zugreift. Dafür ist **ODATA** vorgesehen, und das ist ein Ausblick, keine Zusage – Phase 3 von CloudFLAM.
Der Zusammenhang lohnt den Satz, weil er zeigt, warum die Satzschnittstelle strategisch ist: ODATA ist in der Wolke das, was VSAM auf dem Host war. Wenn der Vollzugriff auf ein geöffnetes FLAM-Member steht, müssen für ODATA nur noch die REST-Stubs gegen unsere Satzschnittstelle gebaut werden. Dann lässt sich diese Zugriffsmethode in der Cloud transparent durch uns ersetzen.



Diese Folie ist der Kern der Einheit. Von links nach rechts erklären.
Erste Stufe: aus dem gesuchten Wert wird eine Signatur gebildet und gegen die Segmentsignaturen im Member-Index geprüft. Das ist billig, weil nichts dekodiert wird.
Zweite Stufe: nur wenn die Signatur passt, wird das Segment dekodiert – MAC prüfen, entschlüsseln, dekomprimieren – und darin der Satz gesucht. Das ist teuer, passiert aber selten.
Die Unschärfe deutlich sagen: die erste Stufe kann falsch-positiv sein, nie falsch-negativ. Ein Nein ist sicher, ein Ja muss geprüft werden.

DER BLOOM-FILTER

- ein Bloom-Filter sagt: **höchstwahrscheinlich Treffer** oder **definitiv kein Treffer**
- er ist eine kleine Signatur in Form einer Bitmaske, gebildet aus Einweg-Hashfunktionen
- in FLAM beantwortet er: in welchem Segment liegt dieser Satz – **ohne dass über die klaren Daten etwas bekannt sein darf**
- der Overhead ist minimal und selbst komprimierbar; alte Archive bleiben mit neuen Signaturen lesbar

Die Unschärfe ist kein Mangel, sondern der Preis für die Günstigkeit: falsch-positiv kostet eine Dekodierung, falsch-negativ gibt es nicht.

FLAMv5.21-42969

Erst allgemein, dann für FLAM. Ein Bloom-Filter ist ein abstrakter Auswahlmechanismus: höchstwahrscheinlich Treffer oder definitiv kein Treffer.
Er steckt in vielen Betriebs-, Datenbank- und Dateisystemen, im Netzwerk-Routing und in der Rechtschreibprüfung.
Für FLAM ist er die Antwort auf die Frage: in welchem autark komprimierten, verschlüsselten und signierten Segment liegt welcher Satz – ohne dass über die klaren Daten irgendetwas bekannt sein darf.
Der Overhead ist minimal und selbst komprimierbar.

ERGEBNISMENGEN OHNE KLARTEXT

- die Auswahl der Segmente braucht nur den **Index-Schlüssel**, nicht den Datenschlüssel
- eine Ergebnismenge kann damit zusammengestellt werden, **ohne einen Satz zu lesen**
- die Ergebnismenge bleibt verschlüsselt, bis jemand mit dem Datenschlüssel sie öffnet
- ein Betreiber bekommt nichts in die Hand, was Rückschlüsse auf die Daten zulässt

Das ist die technische Grundlage für Zentralarchive, Kopfstellen und Cloud – und für die automatische Anonymisierung von Ergebnismengen.

FLAMv5.21-42969

Die Folie, die Outsourcing- und Cloud-Gespräche entscheidet, und sie folgt direkt aus den drei Schlüsseln des Archivs.
Der Suchende braucht den Index-Schlüssel, um Segmente auszuwählen. Er braucht den Datenschlüssel nicht – er kann also eine Ergebnismenge zusammenstellen, ohne einen einzigen Satz zu lesen.
Wortlaut aus den Altfolien: damit kann eine Suche und das Zusammenstellen von Subsets über die komprimierten, verschlüsselten und signierten Daten realisiert werden, ohne dass der Cloud-Betreiber etwas in die Hand bekommt, was Rückschlüsse auf die Daten zulässt.

VERSIONEN IM BETRIEB

- jede Änderung erzeugt eine neue Version – wie ein inkrementelles Backup, ohne zusätzlichen Lauf
- Zugriff über relative Version wie bei GDGs: `(0)` aktuell, `(-1)` vorletzte
- der vorherige Stand bleibt erhalten und muss **explizit** gelöscht werden
- **Löschen macht das Archiv nicht kleiner** – das tut erst `PRUNE`

Für die Kapazitätsplanung: ein Archiv mit vielen Änderungen wächst stetig. `PRUNE` gehört in den Betriebskalender, wie ein Reorg.

FLAMv5.21-42969

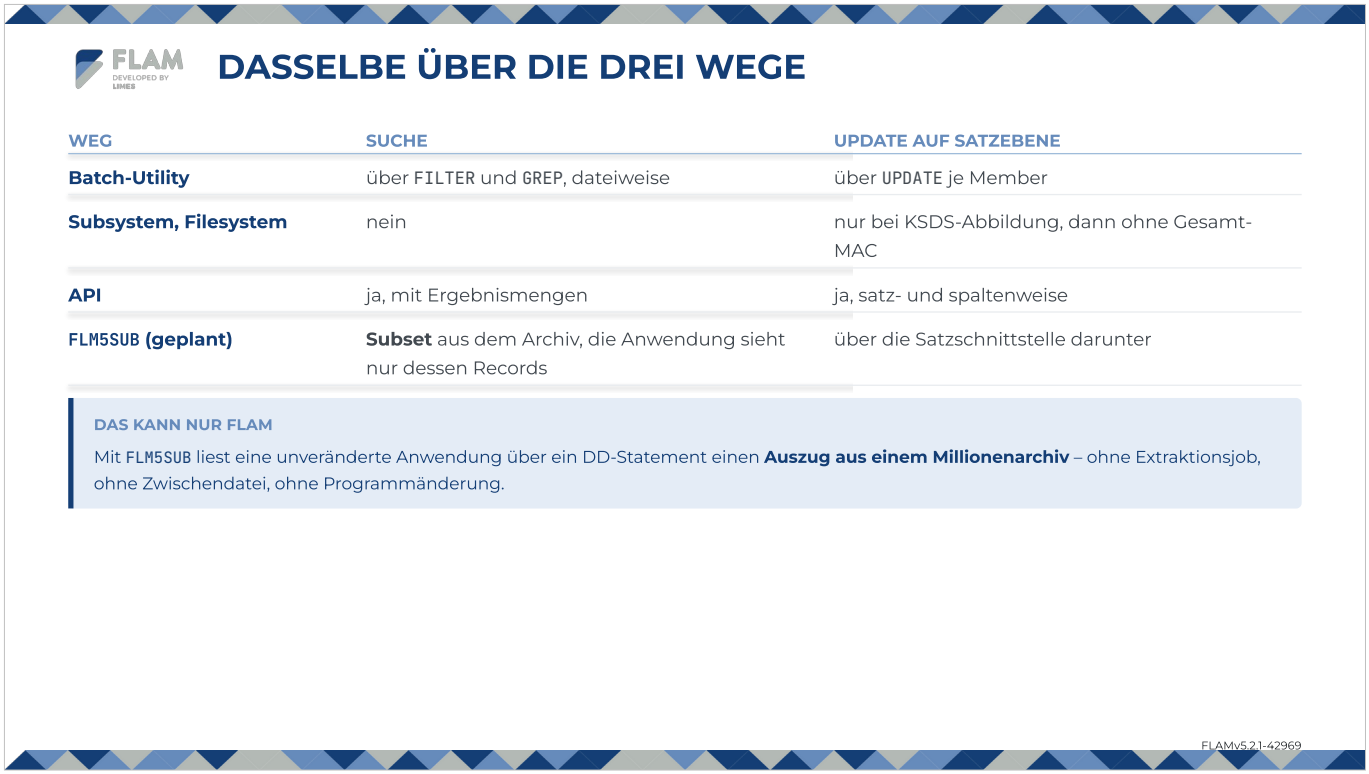
Wiederholung aus dem Archiv-Deck, hier aus der Betriebssicht – und mit dem Aufräumlauf, der eingeplant werden muss.
Der Vergleich mit GDGs zieht bei Mainframe-Leuten sofort: (0) ist der aktuelle Stand, (-1) der vorletzte.
Wichtig für die Kapazitätsplanung: Löschen macht das Archiv nicht kleiner.
Erst PRUNE tut das.



Nicht vorlesen. Die Botschaft ist: das ist eine Verwaltung, kein Packer – und alles davon ist skriptfähig.
Die drei, die im Betrieb am häufigsten gebraucht werden, sind LIST, VERIFY und PRUNE.
`COPY` steht absichtlich an zweiter Stelle, direkt hinter COMP und DECO, denn es ist beides zusammen – und es hat **zwei Varianten**, die man auseinanderhalten muss. Der Weg über die klaren Daten entpackt und packt neu; damit kann man das Ziel anders bauen als die Quelle, etwa mit anderer Kompression oder anderen Schlüsseln, braucht aber die Schlüssel. `TO.DUPLICATE` bewegt die komprimierten und verschlüsselten **Segmente** unverändert; das ist billig und geht ohne Datenschlüssel, liefert aber eine Kopie mit denselben Eigenschaften. Für einen Archivumzug ist die zweite Variante fast immer die richtige.
`LIST` gibt mehr aus, als der Name vermuten lässt: mit `ROWLIST` kommen die **Formate** mit, also die Formatbeschreibungen der Tabellen im Archiv. Damit sieht man, wie die Daten strukturiert sind, die dort liegen – und man bekommt genau die Beschreibung, die man für eine Suche braucht.
`FILTER` und `DIFF` sind zwei verschiedene Dinge und stehen deshalb getrennt:
`FILTER` baut aus Klarwerten eine Signatur und bildet damit eine Teilmenge, `DIFF` vergleicht – und zwar nicht nur Archiv gegen Archiv, sondern auch **Originaldaten gegen Archivinhalt**. Das ist der Fall, den man nach einer Migration braucht.
REKEY ist die Funktion, die im Krypto-Gespräch zählt: der Datenschlüssel wird umgeschlüsselt, ohne die Nutzdaten anzufassen.



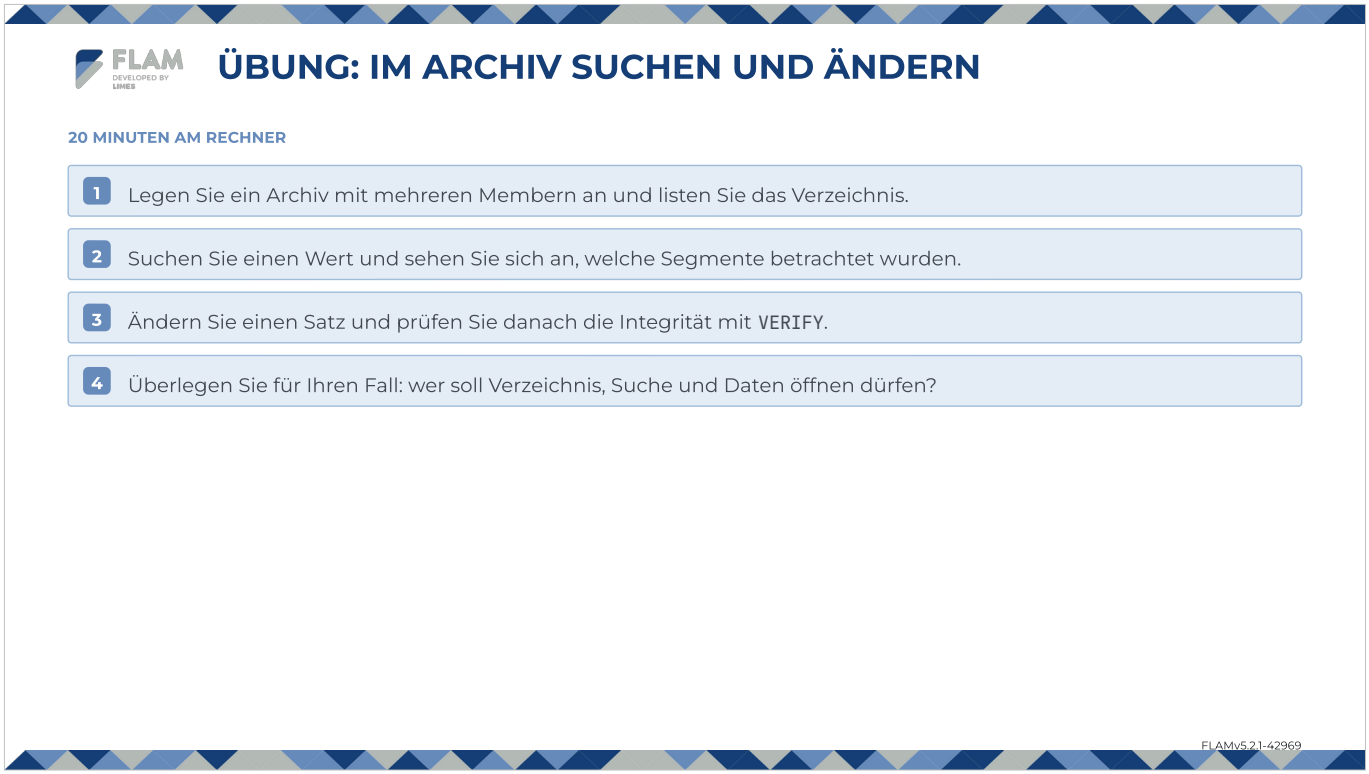
Diese Folie verbindet zwei Themen, die vorher getrennt standen: die drei Schlüssel und die Integritätsprüfung. Der Zusammenhang ist der interessante Teil und stand nirgends.
Zwei Parameter, und der Unterschied ist der ganze Punkt der Folie:
`LEVEL` sagt, wie tief in der Hierarchie geprüft wird – `MODE` sagt, was geprüft wird.
Wer die zwei verwechselt, versteht die Ausgabe nicht.
Die Hierarchie ist Verzeichnis, Member-Segmente, Datensegmente. `LEVEL` bestimmt, bis wohin kryptografisch geprüft wird; unterhalb dieser Grenze laufen nur noch die einfachen Prüfsummen.
`DIRECTORY` prüft das Verzeichnis kryptografisch, `MEMBER` zusätzlich die Member-Segmente, `DATA` auch die Datensegmente. `ALL` ist die Vorbelegung und prüft alle Segmente.
`MODE` bestimmt das Was: `CKS` nur die einfachen Prüfsummen, `KTV` zusätzlich die Schlüsseltestwerte, `MAC` zusätzlich die Message Authentication Codes, `ALL` alles.
Und jetzt der Zusammenhang mit den Schlüsseln, der nirgends steht: **ohne Schlüssel kommt man nur an die einfachen Prüfsummen.** Erst mit Schlüsselmaterial sind Schlüsseltestwerte und MACs prüfbar – und wie tief man dabei gehen kann, hängt davon ab, welche der drei Schlüssel man hat.
Nur den Directory-Schlüssel: `LEVEL=DIRECTORY`. Directory und Member: `LEVEL=MEMBER`. Alle drei: `LEVEL=DATA`.
`VERIFY` prüft also nicht "so tief wie möglich", sondern so tief, wie das Schlüsselmaterial des Prüfenden reicht. Das ist eine Eigenschaft, keine Einschränkung.
Und jetzt die Aussage, die im Gespräch mit einem Auditor zählt: damit kann der Archivbetreiber die Unversehrtheit des Bestands regelmäßig nachweisen, ohne die Daten lesen zu können. Das ist genau die Trennung von Betrieb und Dateneigentümer, die sonst organisatorisch behauptet und hier technisch erzwungen wird.



Für dieses Thema ist die Verteilung besonders deutlich: verwalten geht im Batch, transparent lesen über das Subsystem, aber Suche und gezieltes Ändern nur über das API.
Das ist auch die Antwort auf die Frage, warum es die Satzschnittstelle überhaupt gibt.
Die letzte Zeile ist ein Ausblick, und sie ist der Grund, warum die Folie nicht bei "nein" für das Subsystem stehen bleibt: **mit FLM5SUB wird dieser Weg auch für FLAM5-Archive gehen** – und dann auf eine Art, die es heute nirgends gibt.
Das Muster, das damit möglich wird, lohnt die Erklärung, weil es der eigentliche Nutzen ist: man holt aus einem riesigen Archiv einen **Subset** und gibt der Anwendung nur die Records aus diesem Subset. Die Anwendung liest über ein DD-Statement und sieht eine gewöhnliche Datei – sie merkt nicht, dass hinter ihr ein Archiv mit Millionen Records liegt, aus dem genau ihre Auswahl herausgefiltert wird. Kein Extraktionsjob, keine Zwischendatei, keine Programmänderung.
Für die Erwartungssteuerung: FLM5SUB kommt in einem der nächsten Releases.
Wer heute ein FLAM5-Archiv transparent über ein Subsystem ansprechen will, kann das noch nicht.



Alle vier aus der Praxis. Die erste ist die, die am häufigsten zu einem Ticket führt, die zweite die, die am meisten Missverständnis erzeugt.



Zwanzig Minuten. Aufgabe vier ist die interessante: sie führt zur Frage, wer im eigenen Haus welchen Schlüssel haben soll – und das ist meist eine organisatorische Diskussion, keine technische.
Auflösung:
1. `ARCHIVE.COMP` legt an, `ARCHIVE.LIST` zeigt das Verzeichnis. In der Ausgabe stehen Member, Status und Versionen; mit `VERBOSE=MAXIMAL` auch der Dateistatus je Member.
2. Im Protokoll sieht man, wie viele Segmente betrachtet wurden. Der Punkt der Aufgabe ist, dass es wenige sind – die Bloom-Filter schließen den Rest aus, ohne ihn zu entpacken.
3. Nach dem Ändern entsteht eine neue Version, und `VERIFY` bleibt grün. Genau das ist der Unterschied zu FLAM4, wo ein verändernder Zugriff die Vollständigkeitsprüfung zerstört. Wenn jemand das nachfragt: siehe die Einheit zum FLAM5-Archiv, Folie "Die drei Grenzen von FLAM4".
4. Die erwartete Antwort ist eine Rollenaufteilung, nicht eine Technikantwort:
der Archivbetreiber bekommt den Directory-Schlüssel, die suchende Anwendung den Index-Schlüssel, und nur der Dateneigentümer den Datenschlüssel. Wer alle drei in eine Hand legt, hat die Trennung aufgegeben – und sollte das bewusst tun.

WAS BLEIBT

- Suchen, Lesen, Ändern, Einfügen, Löschen – ohne das Archiv auszupacken
- zweistufig: billiger Signaturvergleich, dann teure Dekodierung nur bei Treffer
- der Bloom-Filter ist unscharf nach oben, nie nach unten – ein Nein ist sicher
- Ergebnismengen entstehen ohne Klartext: Suchen und Lesen sind trennbar
- Versionen bleiben, bis `PRUNE` läuft – das gehört in den Betriebskalender

FLAMv5.21-42969

Lernziele durchgehen. Damit ist der Archiv-Block abgeschlossen. Die nächsten Funktionsthemen sind Konvertierung und Tabellen.



Diese Einheit hat nichts mit der Indizierung zu tun, auch wenn beides Spaltenentscheidungen sind – sie stand deshalb lange im Indizierungsdeck und gehört hierher.
Worum es geht: wenn eine Tabelle in ein Segment geschrieben wird, muss für **jede Spalte** festgelegt werden, wie sie komprimiert und wie sie geschützt wird. Ein Topf ist die Sammelstelle, in die man Spalten mit gleichartigen Daten legt.
Und warum das eine eigene Einheit verdient: es entscheidet über drei Dinge gleichzeitig – **CPU-Aufwand, Sicherheit und Kompressionsrate**. Das sind genau die drei Größen, um die es bei einem großen Archiv geht.
Der Satz von unserer Homepage fasst das Ziel zusammen: es wird nur verschlüsselt, was wirklich schützenswert ist, und nur komprimiert, was sich lohnt.



Die Lernziele. Das zweite ist der Kern der Einheit – zwei Fragen je Spalte, und der Entwurf ergibt sich.
Das fünfte ist der Punkt, an dem man Geld sieht: ein Topf pro Spalte ist so falsch wie ein Topf für alles, und beides kommt in der Praxis vor.



Erst der Mechanismus, damit die Entscheidungen danach verständlich sind.
Eine Tabelle im Speicher ist eine **Matrix**: Spalten mit typisierten Werten, beliebig viele Zeilen. Zum Schreiben muss diese Matrix in ein **Segment** serialisiert werden – und dabei stellt sich je Spalte die Frage, wie sie komprimiert und wie sie geschützt wird.
Ein **Topf** ist die Antwort darauf: man definiert Töpfe mit Namen, und jeder Spalte weist man einen Topf zu. Alle Spalten eines Topfes werden gleich behandelt und **gemeinsam** komprimiert.
Und genau darin liegt der Gewinn und die Falle. Der Gewinn: Spalten mit gleichartigen Daten in einem Topf komprimieren **besser als jede für sich**, weil der Kompressor mehr vom gleichen Muster sieht. Die Falle: Spalten mit verschiedenartigen Daten im selben Topf **stören einander**.
Der Satz aus unserem Handbuch dazu, und er ist ehrlich: das Ziel sollte eine **minimale Anzahl** von Töpfen sein, weil jeder Topf Overhead kostet – aber manchmal ist mehr als einer deutlich besser.
Was die Vorbelegung macht: **alles in einen Topf.** Das ist für viele Bestände richtig und für gemischte Bestände die schlechteste Wahl. Genau darum geht es in dieser Einheit.



Die Folie erklärt, warum es die drei Ströme der nächsten Folie überhaupt gibt – sie kommen nicht aus einer Designlaune, sondern aus dem Aufbau eines Elements.
Ein Element ist nicht nur sein Wert. Es besteht aus **einem Flag-Byte, einer Länge und den Daten**. Im Flag-Byte stehen acht Bits, die mitgeschrieben werden; belegt sind heute unter anderem "der Wert ist NULL" und "der Wert ist leer", der Rest ist Reserve. Das ist der Grund, warum ein NULL-Wert in einem FLAM-Archiv **kein** Byte Daten kostet: er steht im Flag-Byte.
Der Topf sammelt diese Teile nicht elementweise, sondern **getrennt**: ein Array von Flag-Bytes, die Längen, und ein Puffer mit den Daten. Genau deshalb kann man je Teil ein eigenes Kompressionsverfahren und einen eigenen Schutz einstellen – das ist die nächste Folie.
Zwei Feinheiten, die man nicht erklären muss, aber auf Nachfrage nennen kann. Die Längen werden **byteweise** aufgeteilt: sind alle Werte kürzer als 256 Byte, wird nur ein Längenpuffer geschrieben; erst ein langer Wert kostet den zweiten. Und sind alle Flag-Bytes, alle Längen oder alle Werte identisch, steht das Ganze **einmal** im Segment statt tausendfach – eine Spalte mit einem konstanten Wert ist damit praktisch kostenlos.
Wer nach der Wirkung fragt: das ist der Unterschied zwischen einem Archiv, das eine Tabelle als Bytestrom komprimiert, und einem, das ihre Struktur kennt.



Eine Folie, die überrascht und die erklärt, warum FLAM so gute Raten erreicht.
Eine Spalte ist nicht ein Datenstrom, sondern **drei**: die Flags, die Längen und die Werte. Das ist keine Buchhaltung, sondern eine Kompressionsentscheidung – denn die drei sind völlig verschiedenartig.
Die Flags sind wenige Bits je Zeile: ist der Wert vorhanden, ist er leer, ist er der gleiche wie darüber. Über tausend Zeilen ist das eine Bitfolge mit sehr wenig Entropie – arithmetisch fast auf nichts komprimierbar.
Die Längen sind kleine Zahlen, oft gleich oder dicht beieinander. Auch das ist arithmetisch viel besser komprimierbar als mit Mustersuche.
Die Werte sind das eigentliche Material – Text, Zahlen, Binärdaten.
Und jetzt der Punkt: **jeder der drei Ströme lässt sich einzeln einstellen**, in Kompression und in Vertraulichkeit. In `POTPAR` heißen sie `FLG`, `LEN` und `DAT`, jeweils mit `...LEVEL` für die Kompression und `...CONF` für den Schutz.
Wozu man das braucht: die Längen einer Spalte sind selten geheim, die Werte oft. Man kann also die Werte verschlüsseln und die Längen klar lassen – das spart CPU. Umgekehrt gilt: wenn schon die **Länge** eines Wertes etwas verrät, verschlüsselt man sie mit.
Und unsere eigene Praxis, weil sie das Prinzip zeigt: in unseren Archiven komprimieren wir die **Längenfelder arithmetisch** und die **Daten über Mustersuche**. Das ist dasselbe, was diese Folie empfiehlt – nur intern schon so gebaut.



Die Folie, die erklärt, warum ein FLAM-Archiv bei festen Satzlängen so klein wird – und sie ist die Antwort auf die Frage "kostet die Struktur nicht Platz?".
Der Topf vergleicht beim Sammeln jedes Element mit dem ersten und merkt sich drei Dinge: sind alle Flag-Bytes gleich, sind alle Längen gleich, sind alle Werte gleich. Steht eines davon, wird der ganze Puffer **nicht geschrieben**; der eine Wert steht stattdessen im Segmentkopf. Das ist nicht selten, sondern der Normalfall: Flags sind meistens überall gleich, und bei einem festen Satzformat – ein PS mit FB1000 zum Beispiel – sind auch alle Längen gleich. Dann kostet die Länge **nichts**.
Für die Werte gilt dieselbe Regel mit zwei Grenzen: der Wert muss kürzer als 128 Byte sein, und der Topf darf nicht verschlüsselt sein – der Segmentkopf liegt unter dem Member-Schlüssel, und dort hat ein verschlüsselter Wert nichts zu suchen.
Variieren die Längen, kommt die zweite Ersparnis: die Länge ist 32 Bit und wird in vier Byte-Ebenen zerlegt. Geschrieben wird nur, was der längste Wert im Topf braucht – unter 256 Byte also eine Ebene, unter 64 KiB zwei.
Die höherwertigen Ebenen, die überall 0x00 wären, entstehen gar nicht.
Und erst wenn wirklich variiert wird, wird komprimiert: Flags und Längen mit der Vorbelegung DYNAMIC, das Verfahren dahinter kommt aus der Kompressions-Suite, und die ist ab Werk **gzip**.
Die Daten bekommen das Verfahren, das für den Topf gewählt wurde. Man kann die Stufe für Flags und Längen ändern – **man sollte es nicht**, die Vorbelegung ist für diese Art von Daten die richtige Wahl.
Letzter Punkt für die Nachfrage: bringt die Kompression nichts, wird sie verworfen und der Rohpuffer geschrieben. Ein Topf wird durch die Kompression also nie größer.



Die Kernfolie, und sie ist absichtlich so einfach gehalten, dass man sie im Kopf behält.
Je Spalte stellt man **zwei** Fragen:
Ist sie redundant? Kommen Werte mehrfach vor, gibt es Muster, lohnt Kompression überhaupt?
Ist sie sensibel? Steht dort etwas, das geschützt werden muss?
Zwei Fragen mit je zwei Antworten ergeben vier Fälle, und jeder Fall ist ein Topf:
Redundant und sensibel – komprimieren **und** verschlüsseln. Das ist der Normalfall für Namen, Adressen, Verwendungszwecke.
Sensibel, aber nicht komprimierbar – nur verschlüsseln. Das klassische Beispiel ist ein JPG oder ein PDF in einer Spalte: es ist schon komprimiert, ein zweiter Durchgang würde es nur **aufblähen** und CPU kosten.
Redundant, aber nicht sensibel – nur komprimieren. Kennzeichen, Währungen, Statuswerte, Buchungsschlüssel.
Weder noch – weder komprimieren noch verschlüsseln. Der typische Fall sind **Kryptogramme**: Daten, die schon verschlüsselt sind. Sie sind nicht komprimierbar, weil sie wie Zufall aussehen, und sie brauchen keinen Schutz, weil sie ihn schon haben.
Damit das nicht abstrakt bleibt, das Beispiel aus dem Zahlungsverkehr: der **PAC**, der Personal Authentication Code – also der verschlüsselte PIN-Block bei frei gewählter PIN, oder ein randomisierter Einweghash zur Verifikation der Kunden-PIN. So etwas steht als Spalte in der Datei, und es ist der Musterfall für "weder noch": zufällig aussehend und längst geschützt.
Der Gewinn dieser Aufteilung ist doppelt: die Kompression wird besser, weil die nicht komprimierbaren Daten die anderen nicht mehr stören, und die CPU sinkt, weil nur verschlüsselt wird, was es braucht.
Zwei Punkte gehören hier dazu, und sie sind der eigentliche Grund für die zweite Zeile der Tabelle.
Je weniger redundant die Daten sind, desto mehr Rechenzeit kostet der Kompressor. Das ist gegen die Intuition: am teuersten ist genau der Fall, in dem nichts herauskommt. Der Kompressor findet keine Treffer, muss aber trotzdem über alles suchen – ein JPG in einer Spalte ist die rechenintensivste Sache, die man einem Topf antun kann.
FLAM erkennt die Expansion und geht von selbst auf `COPY`. Wird das Komprimat größer als der Rohpuffer, schreiben wir den Rohpuffer – das gilt auch für ein einzelnes Segment und einen einzelnen Puffer, wenn also in einem Topf mit allem plötzlich ein großes JPG auftaucht. Aufgeblähte Daten gibt es deshalb nie, und das Dekodieren spart sich die Arbeit ebenfalls.
Die Rechenzeit ist aber verbrannt. Erkennen kann man es erst, wenn man es versucht hat. Genau deshalb gehört dieser Fall über das Topfdesign von vornherein auf `CMPLEVEL=COPY` – nicht weil das Ergebnis sonst falsch wäre, sondern weil man sich den Versuch spart.
Das ist das Grundmuster. **In der Regel braucht man diese vier.** Wer mit einem anfängt und die JPGs herausnimmt, hat den größten Teil des Gewinns schon.



Die Verfeinerung, und sie lohnt sich bei großen Beständen deutlich.
Die beiden Töpfe, in denen komprimiert wird, kann man **noch einmal teilen** – nach dem Verfahren, das zur Datenart passt. Denn die Verfahren arbeiten nach verschiedenen Prinzipien: **Arithmetische Kompression** ist gut für **Zahlen**: Beträge, Messwerte, Integer- und Float-Felder, und natürlich die Längen- und Flag-Ströme von Folie 4. **Mustersuche** ist gut für **Text**: Namen, Adressen, Verwendungszwecke, Referenznummern.
Wenn man also eine Tabelle hat, in der beides vorkommt – und im Zahlungsverkehr ist das immer so –, dann werden aus den zwei komprimierenden Töpfen vier: sensibel/arithmetisch, sensibel/Mustersuche, offen/arithmetisch, offen/Mustersuche. Mit den zwei nicht komprimierenden sind das sechs.
Sechs Töpfe sind aber schon viel – und damit sind wir bei der Abwägung von Folie 9. Für einen großen Bestand mit vielen Zeilen lohnt es; für eine Tabelle mit tausend Zeilen und zwanzig Spalten ist es zu fein.
Und es gibt einen zweiten Grund für mehr Töpfe, der nichts mit dem Verfahren zu tun hat: **die Zusammengehörigkeit der Daten.** Bei Messungen bei einer Zentralbank hat sich gezeigt, dass die **Kennungen** zusammen in einen Topf gehören – IBAN, BIC, Bankleitzahl, Kontonummern –, und die **Freitexte** in einen anderen: Namen, Adressen, Verwendungszwecke.
Kennungen sind untereinander ähnlich aufgebaut, Freitexte sind es auch, und gemischt stören sie sich.
Der dritte Fall ist der lohnendste und der am leichtesten zu übersehende:
eine Spalte, deren Wert in der ganzen Matrix gleich ist, bekommt einen eigenen Topf. Dann greift die Abkürzung von Folie 6 – der Wert wird in den Kopf ausgelagert und die Spalte kostet praktisch nichts. Das passiert in der Praxis, wenn die Daten **sortiert** sind: innerhalb einer Matrix ist der Mandant, das Datum oder die Währung dann konstant. Oft erledigt das auch der `LITERAL`-Schalter, aber über einen eigenen Topf hat man es in der Hand.
Der praktische Rat: **mit vier anfangen und messen.** Wenn die Beträge einen großen Teil der Daten ausmachen, ist die Teilung nach Verfahren der nächste sinnvolle Schritt. Wenn nicht, spart man sich den Overhead.



Die Nachschlagefolie. Ein Topf hat einen Namen, eine Kompressionsstufe, eine Vertraulichkeitsstufe – und optional `POTPAR` für die drei Ströme einzeln.
Bei `CMPLEVEL` ist die Reihenfolge eine Skala: `COPY` komprimiert nicht, `FAST` ist schnell mit geringerer Rate, `DYNAMIC` ist der ausgewogene Weg, `COMPACT` ist langsam mit hoher Rate. Und `ARITHMETIC` ist kein Punkt auf dieser Skala, sondern ein **anderes Verfahren** – für die Zahlen von Folie 6. Bei `CRYCONF` gibt es drei Stufen: `CLEAR` ist kein Schutz, `ENC` ist Verschlüsselung. Dazwischen liegt `VEIL`, Vertraulichkeitsschutz durch Verschleierung – die Parameterhilfe sagt genau das, und mehr ist dazu nicht dokumentiert. Wer eine Schutzanforderung hat, nimmt `ENC`; `VEIL` ist die Zwischenstufe für Daten, die nicht offen liegen sollen, aber auch keinen Schlüssel wert sind.
Und die Zuordnung: an der Spalte steht `POT='name'`. Eine Spalte ohne Angabe landet im Vorbelegungstopf.
Ein Hinweis zur Reihenfolge im Parameterstring, weil es eine häufige Rückfrage ist: die Töpfe werden **im `ROW`** definiert, neben den Spalten, nicht innerhalb einer Spalte. Die Spalte verweist nur auf den Namen.



Ein Schalter, der zum Topfdesign gehört, obwohl er nicht am Topf steht – und der eine Sicherheitsfolge hat, die man kennen muss.
Der Mechanismus: manche Spalten enthalten nur eine Handvoll verschiedener Werte – ein Statuskennzeichen, ein Buchungsschlüssel, ein Ländercode. Für solche Spalten gibt es die **globale Literal-Matrix**: mit `LITERAL` an der Spalte werden die Werte dorthin verlagert, statt in den zugewiesenen Topf zu gehen.
Der Gewinn: die Werte stehen **einmal** und werden referenziert, nicht in jedem Segment erneut. Bei einer Spalte mit zehn verschiedenen Werten über Millionen Zeilen ist das erheblich. Die Parameterhilfe nennt als Zweck ausdrücklich Selektionen – eine Auswahl über wenige Literale wird damit billig.
Und jetzt die Sicherheitsfolge, und die ist der Grund, warum diese Folie existiert: **die Literal-Matrix wird mit dem Directory-Schlüssel geschützt**, nicht mit dem Datenschlüssel.
Was daraus folgt: wer – wie in der Einheit *Das FLAM5-Archiv* beschrieben – zwischen **Verzeichniszugriff** und **Datenzugriff** sicherheitstechnisch unterscheiden will, darf den Schalter für sensible Spalten **nicht** setzen.
Sonst kann jemand mit dem Directory-Schlüssel diese Werte lesen, obwohl er die Daten nicht lesen darf.
In der Praxis sind Literale meist unkritisch – ein Buchungsschlüssel ist kein Geheimnis. Aber es ist eine Entscheidung, keine Automatik. Und sie gehört bewusst getroffen, weil sie beim Anlegen des Archivs fällt und später nicht mehr zu ändern ist.



Die Folie, die zwischen Topf und Archiv steht – und die Antwort auf die Frage, die im Sicherheitsgespräch immer kommt: "Und was ist der Integritätsschutz?" Was ein Topf einzeln festlegt, legt `DEFAULTS()` für **alle** Töpfe vor.
Zwei Angaben kennen wir schon (`CMPLEVEL`, `CRYCONF`), zwei sind neu und gehören zusammen erklärt.
`INTPROT` ist die Integritätsstufe je Segment, und sie hat vier Stufen:
`NONE` ohne Schutz, `CHKS` mit Prüfsumme, `AUTH` mit **verschlüsselter** Prüfsumme und `MAC` mit einem Message Authentication Code – konkret einem **AES-CMAC**, gerechnet über die komprimierten Daten. Die Vorbelegung ist die ehrlichste Stelle der ganzen Parameterhilfe: **`MAC`, wenn ein Schlüssel da ist, sonst `CHKS`**. Wer verschlüsselt, bekommt also automatisch den vollen Nachweis; wer nur komprimiert, bekommt eine Prüfsumme. Genau diese Stufen tauchen beim `VERIFY` als Prüftiefe wieder auf.
`SERASE` löscht Daten aus dem Speicher, sobald sie nicht mehr gebraucht werden – Vorbelegung `AUTO`, was `YES` bedeutet, wenn ein Schlüssel im Spiel ist. Für eine Schutzbedarfsanalyse ist das die Zeile, die man zeigt.
`MAXREC` und `MINSIZ` sind die Stellschrauben der Matrixgröße; sie stehen in der Performance-Einheit.
Und eine Ebene höher liegen die beiden **Suiten**: `CMPSUITE` wählt das Kompressionsverfahren (`GZIP`, `LZ4`, `BZIP2`, `ZSTD`), `CRYSUITE` die Kryptokombination – `V06` ist die Vorbelegung und heißt AES-256 für Ableitung und Verschlüsselung mit CRC32 als Prüfsumme. Die sechs Suiten unterscheiden sich in der Schlüssellänge und darin, ob CRC32 oder FNV32 gerechnet wird.



Die Folie, die vor den beiden naheliegenden Fehlern bewahrt – und beide kommen in der Praxis vor.
Das eine Extrem: alles in einen Topf. Das ist die Vorbelegung. Der Fehler dabei ist nicht die Menge, sondern die Mischung: ein JPG und ein Kryptogramm im selben Topf wie die Namen **stören die Kompression** der Namen. Der Kompressor sieht zwischen brauchbaren Mustern immer wieder Zufallsdaten und findet dadurch weniger.
Das andere Extrem: ein Topf pro Spalte. Klingt nach maximaler Optimierung und ist es nicht, denn jetzt hat **jeder Topf zu wenig Daten**.
Kompression braucht Material: ein Verfahren, das über zweihundert Byte laufen darf, findet nichts. Dazu kommt der Overhead je Topf, den unser Handbuch ausdrücklich nennt.
Zwischen beidem liegt das Optimum, und es hängt an der Datenmenge je Spalte im Segment. Die Reihenfolge, in der man sich nähert:
Erstens die nicht komprimierbaren Spalten heraustrennen – JPGs, Kryptogramme. Das ist der größte einzelne Gewinn und kostet einen Topf.
Zweitens nach Schutzbedarf trennen, damit nur verschlüsselt wird, was es braucht. Das spart CPU.
Drittens, wenn der Bestand groß genug ist, nach Verfahren trennen – Zahlen arithmetisch, Text über Muster.
Und danach **messen**. Das ist der Satz, den man mitgeben sollte: das Topfdesign ist eine Messfrage, keine Glaubensfrage. Denselben Bestand mit zwei Entwürfen schreiben und die Größen vergleichen kostet einen Nachmittag und gilt dann für Jahre.



Die Folie, an der die Teilnehmer ihren eigenen Bestand wiedererkennen sollen.
Eine Buchungstabelle, wie sie jede Bank hat. Die Spalten der Reihe nach:
IBAN und Name sind sensibel und redundant – gleiche Konten und gleiche Namen kommen wieder und wieder vor. Text, also Mustersuche, und verschlüsselt.
Der Betrag und das Datum sind sensibel und redundant, aber es sind **Zahlen**. Arithmetisch, und verschlüsselt.
Währung und Buchungsschlüssel sind nicht sensibel und sehr redundant – eine Handvoll Werte über Millionen Zeilen. Nur komprimieren. Und hier ist `LITERAL` die richtige Wahl, weil es genau der Fall ist, für den die Literal-Matrix gebaut wurde.
Der Belegscan, ein JPG oder PDF, ist sensibel und nicht komprimierbar.
Nur verschlüsseln, `CMPLEVEL=COPY`.
Das Kryptogramm – etwa ein von einer Vorstufe verschlüsseltes Feld – ist weder noch. Kein Topfaufwand.
Das sind vier Töpfe und ein Literal-Fall. Und der Vergleich, der die Einheit rechtfertigt: läge alles in einem Topf, würde der Belegscan die Kompression von Namen und Verwendungszwecken verderben, und der Betrag würde mit einem Verfahren komprimiert, das für Zahlen das falsche ist.
Wenn die Frage nach Zahlen kommt: unsere Homepage nennt Raten bis 50 zu 1 – das sind Bestände, bei denen Struktur und Topfdesign zusammenpassen. Eine konkrete Zahl für den eigenen Bestand bekommt man nur durch Messen, und genau dazu sollte man die Teilnehmer auffordern.



Vier.
Zur ersten: die Vorbelegung ist ein Topf, und für einen gemischten Bestand ist das die schlechteste Wahl. Wer nichts entscheidet, hat schlecht entschieden.
Zur zweiten: ein Topf pro Spalte klingt nach Optimierung und ist das Gegenteil.
Zur dritten: schon komprimierte Daten ein zweites Mal zu komprimieren kostet CPU und macht sie größer. Das ist kein Grenzfall, das ist die Regel bei JPG, PDF und ZIP in Spalten.
Zur vierten: `LITERAL` bei sensiblen Werten hebelt die Trennung zwischen Verzeichnis- und Datenzugriff aus. Das fällt niemandem auf, weil es funktioniert.



Zwanzig Minuten, und diese Übung sollte man mit einer echten Tabelle der Teilnehmer machen, wenn eine im Raum ist.
Die Lösungen:
Aufgabe 1. Zwei Fragen je Spalte – redundant, sensibel – und dann in die vier Felder eintragen. Die Spalten mit Binärinhalten und Kryptogrammen zuerst heraussuchen, das ist der größte Gewinn.
Aufgabe 2. `CMPLEVEL=COPY` und `CRYCONF=ENC`. Komprimieren würde es aufblähen, und geschützt werden muss es.
Aufgabe 3. Ja, wenn es nur eine Handvoll Werte sind – aber nicht, wenn diese Werte selbst schützenswert sind und das Haus zwischen Verzeichnis- und Datenzugriff trennt.
Aufgabe 4. Denselben Bestand zweimal schreiben, einmal mit einem Topf und einmal mit dem Entwurf, und die Größen und CPU-Zeiten vergleichen. Das ist die einzige belastbare Antwort – alles andere ist geschätzt.

WAS BLEIBT

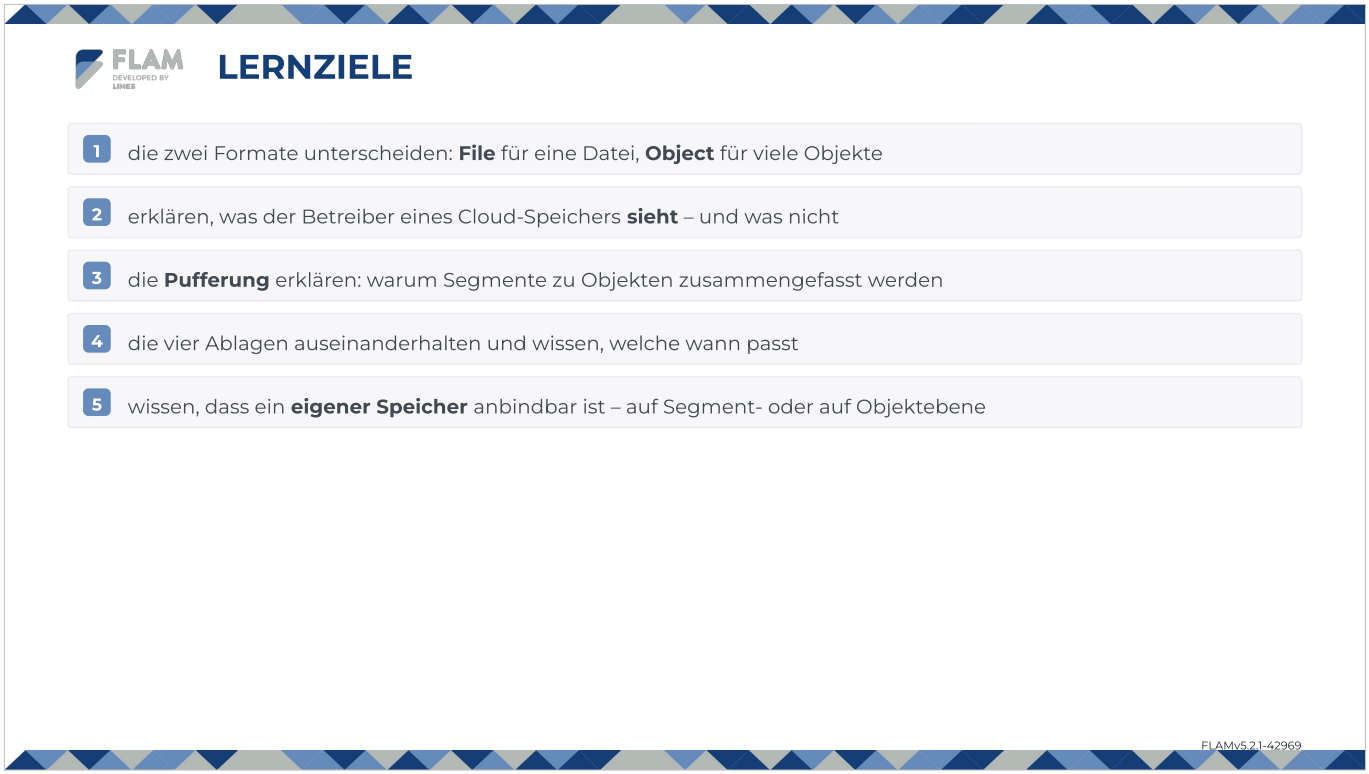
- ein **Topf** ist eine benannte Sammelstelle; Spalten darin werden gleich behandelt und **gemeinsam komprimiert**
- je Spalte **drei Ströme** – `FLG`, `LEN`, `DAT` –, jeder einzeln in Kompression und Vertraulichkeit einstellbar
- **zwei Fragen** ergeben vier Töpfe: redundant und sensibel, sensibel allein, redundant allein, weder noch
- **arithmetisch für Zahlen, Mustersuche für Text** – daraus werden sechs Töpfe, wenn der Bestand groß genug ist
- **ein Topf stört, zu viele hungern** – und `LITERAL` hängt am Directory-Schlüssel
- es ist eine **Messfrage**: zwei Läufe, ein Vergleich, und die Entscheidung gilt für Jahre

FLAMv5.21-42969

Lernziele durchgehen. Der Satz zum Schluss ist der von der Titelfolie: nur verschlüsseln, was schützenswert ist, und nur komprimieren, was sich lohnt.
Und der Hinweis auf die Nachbareinheit: die Indizierung ist die **andere** Spaltenentscheidung, und sie fällt zum gleichen Zeitpunkt – beim Anlegen des Archivs.



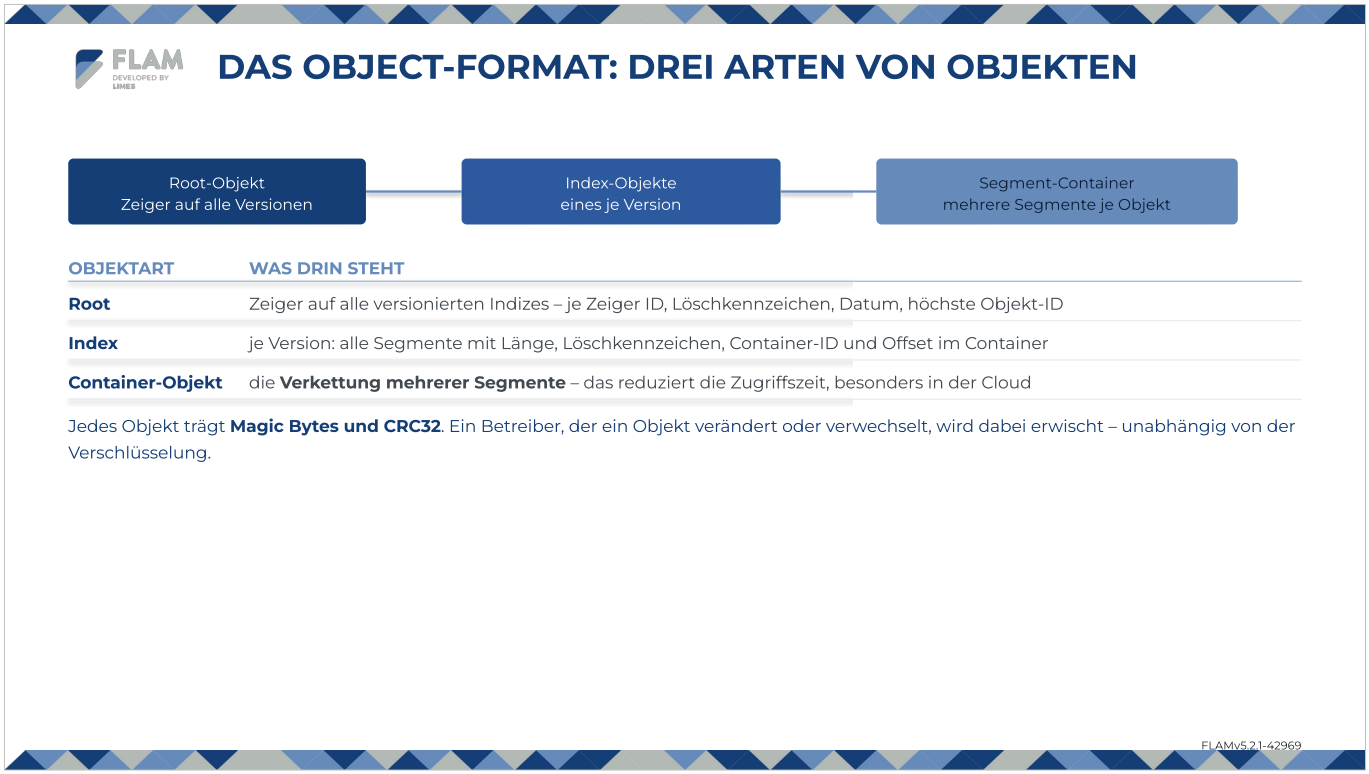
Die Einheit, die beantwortet, **wo ein FLAM5-Archiv physisch liegt** – und warum diese Frage von der Frage getrennt ist, was im Archiv steht.
Das ist die Trennung, die in der Archiv-Einheit als Architekturprinzip genannt wurde: **Archivverwaltung** und **Segmentablage** sind zwei Schichten. Diese Einheit füllt die untere aus.
Praktisch beantwortet sie drei Fragen, die in jedem Cloud-Gespräch kommen:
Was liegt eigentlich beim Betreiber? Wie oft greift FLAM übers Netz zu? Und was tun wir, wenn der Kunde einen Speicher hat, den wir nicht kennen?
Vorausgesetzt wird die Einheit *Das FLAM5-Archiv*. Die Einheit *Indizierung, Filter und Töpfe* ist nützlich, aber nicht nötig.



Die Lernziele als Vertrag. Das dritte ist das, für das man am Ende dankbar ist: verstehen, warum die Pufferung existiert. Ohne sie wäre ein Archiv in der Cloud unbenutzbar langsam, und mit der falschen Einstellung ist es das auch.
Das fünfte Ziel richtet sich an die Systemleute: dass man einen beliebigen Speicher anbinden kann, und zwar auf zwei Ebenen.



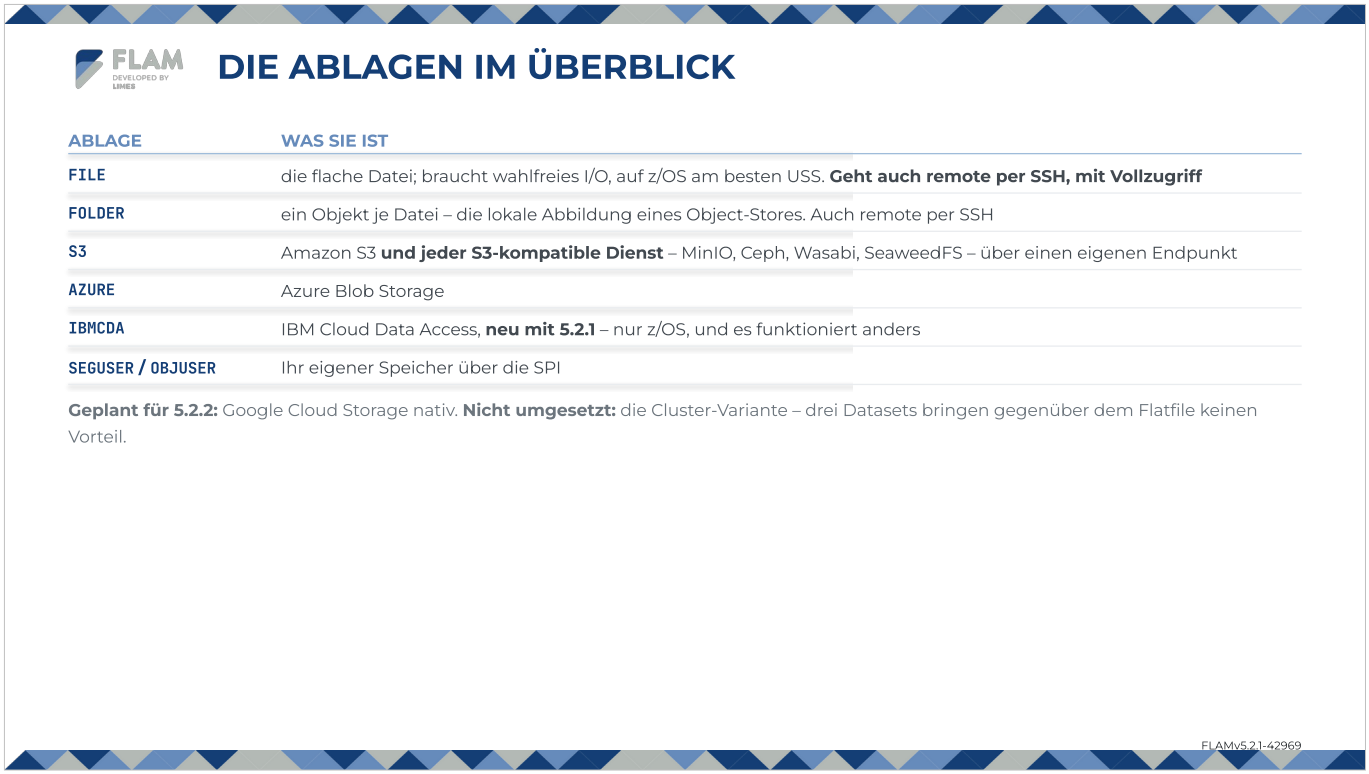
Die Grundunterscheidung, und sie ist die ganze Folie wert, weil alles Weitere daran hängt.
Der Segment-Store ist die Grundlage für die Ablage von FLAM5-Archiven. Alle archivierten Daten sind in **Segmente** zerlegt, und die Segmente werden über **versionierte Indizes** verwaltet. Jede Version des Archivs hat ihren eigenen Index, der alle Segmente mit ihren Metadaten auflistet.
Daraus folgt eine Eigenschaft, die man im Betrieb kennen muss: **wenn Segmente gelöscht werden, bleiben sie über die älteren Versionen erreichbar** – bis alle Versionen gelöscht sind, die auf sie verweisen. Das ist die Mechanik hinter dem, was in der Archiv-Einheit als Versionierung auftauchte, und der Grund, warum `PRUNE` existiert.
Und es gibt **zwei Formate**, für zwei verschiedene Welten:
Das **File-Format** legt das ganze Archiv in **eine** sequenzielle Datei.
Das ist die richtige Wahl, wenn man eine geschlossene Datei haben will – zum Verwalten, zum Kopieren, zum Übertragen. Ein Archiv, eine Datei.
Das **Object-Format** legt das Archiv in **viele kleine Objekte**. Die können lokal in einer Ordnerstruktur liegen oder in einem Cloud-Speicher.
Das ist die Wahl, wenn Skalierbarkeit zählt – und die einzige Wahl, wenn der Speicher kein Seek kennt.
Der Satz für den Raum: **es ist dasselbe Archiv.** Die Aufteilung ist eine Frage der Ablage, nicht des Inhalts. Man kann ein Archiv mit `ARCHIVE COPY` von einem Format in das andere überführen, ohne es auszupacken.



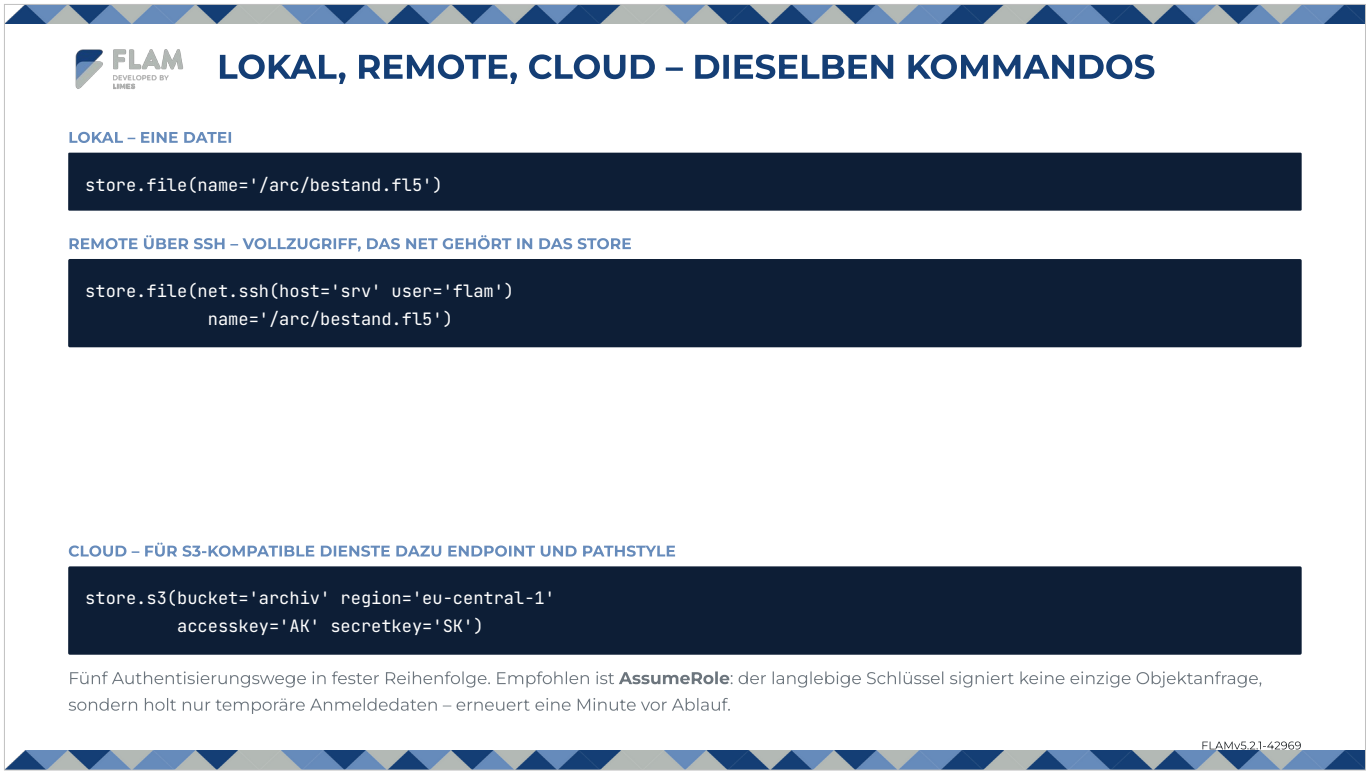
Jetzt genauer, denn wer in die Cloud geht, sollte wissen, was dort tatsächlich als Objekte ankommt. Es sind genau drei Arten:
Das **Root-Objekt** ist der Eintrittspunkt. Es enthält die Zeiger auf alle versionierten Indizes – je Zeiger eine ID, ein Löschkennzeichen, das Erstellungsdatum und die höchste benutzte Objekt-ID. Wer das Root-Objekt liest, weiß, welche Versionen existieren.
Die **Index-Objekte** sind je eines pro Version. Ein Index enthält die Liste aller Datensegmente dieser Version, und zu jedem Segment: Länge, Löschkennzeichen, die Objekt-ID des Containers, in dem es liegt, und den Offset innerhalb dieses Containers.
Die **Segment-Container** sind das Interessante: sie sind **Verkettungen mehrerer Segmente.** Und der Grund dafür steht so im Handbuch: die Zusammenfassung reduziert die Zugriffszeit, besonders bei Cloud-Speichern.
Das ist die Pufferung, die auf der nächsten Folie erklärt wird – hier ist sie erst einmal als Datenstruktur sichtbar.
Ein Detail, das für die Sicherheit zählt und im Cloud-Gespräch hilft:
jedes dieser Objekte trägt CRC32 und Magic Bytes. Ein Betreiber, der ein Objekt verändert oder verwechselt, wird dabei erwischt – unabhängig von der Verschlüsselung.
Und zur Frage nach dem Overhead, die hier meist kommt: die Metadaten der Segment- und Objektverwaltung werden **intern mit ZSTD komprimiert**. Auch die Verwaltung kostet also kaum Platz – das ist kein Nebensatz, sondern der Grund, warum man Segmente klein halten kann, ohne dass die Verwaltung ins Gewicht fällt.



Die Folie, die erklärt, warum ein Archiv in der Cloud überhaupt benutzbar ist. Und sie erklärt gleichzeitig eine Betriebsregel.
Die Ausgangslage: ein Archiv besteht aus vielen Segmenten. Ein Cloud-Objektspeicher ist beim **Durchsatz** gut und bei der **Latenz** schlecht – jeder einzelne Zugriff kostet eine Netzwerkrunde, und die bezahlt man auch für ein kleines Objekt.
Wer also jedes Segment als eigenes Objekt holt, bezahlt die Latenz **einmal pro Segment**. Bei Tausenden Segmenten ist das Archiv unbenutzbar – unabhängig davon, wie breit die Leitung ist.
Deshalb fasst der Object-Store mehrere Segmente zu einem Objekt zusammen und liest oder schreibt immer **das ganze Objekt**. Die Voreinstellungen stehen im Quelltext: **4 MiB für Datensegmente, 1 MiB für Membersegmente**, und das Main-Segment wird ein eigenes Objekt.
Damit bezahlt man die Latenz einmal pro Objekt statt einmal pro Segment.
Und jetzt die Betriebsregel, die man mitgeben soll: **in der Cloud lohnen größere Puffer, lokal nicht.** Lokal ist das File-Format mit Offsets ohnehin schneller; ein großer Puffer liest dort nur Daten, die niemand braucht.
Die Größen sind einstellbar – das ist eine Messfrage, keine Glaubensfrage.
Das Usage-Log aus der Performance-Einheit zeigt, wie viele Zugriffe wirklich stattfinden.
Und der Bogen zum Ordner: **der Ordner ist die lokale Abbildung genau dieses Object-Stores.** Dieselbe Aufteilung in Objekte, nur ohne Netz. Deshalb kann man ein Cloud-Layout lokal entwickeln und testen, bevor man in die Cloud geht – und das ist der praktische Rat für jeden, der anfängt.



Die Nachschlagefolie. Nicht vorlesen – durchgehen und die zwei Zeilen betonen, die überraschen.
`FILE` ist die flache Datei. Sie braucht wahlfreies I/O, auf z/OS also am besten USS-Dateien. Und die erste Überraschung: **sie geht auch remote.** Über `NET.SSH` lesen, schreiben und ändern wir ein Archiv auf einem entfernten System – mit Vollzugriff, über SFTP.
`FOLDER` ist die Ordnervariante, ein Objekt je Datei. Auch sie geht remote über SSH.
`S3` spricht Amazon S3 **und jeden S3-kompatiblen Dienst** – MinIO, Ceph, Wasabi, SeaweedFS. Dafür gibt man einen eigenen Endpunkt an. Das ist die zweite Überraschung für viele: man braucht keine Amazon-Cloud, um S3 zu benutzen. Viele Häuser haben einen S3-kompatiblen Objektspeicher im eigenen Rechenzentrum.
`AZURE` ist Azure Blob Storage.
`IBMCDA` ist neu mit 5.2.1 und der z/OS-Weg – dazu die nächste Folie, denn es funktioniert anders als die beiden anderen.
`SEGUSER` und `OBJUSER` sind die beiden SPI: der eigene Speicher. Auch dazu eine eigene Folie.
Und zum Stand: **Google Cloud Storage nativ ist geplant und kommt mit 5.2.2.** Bis dahin geht GCS über die S3-kompatible Schnittstelle, wenn der Kunde sie aktiviert.
Was es **nicht** gibt und warum: die **Cluster-Variante**. Die Daten auf drei Datasets zu verteilen – Directory, Index, Daten – bringt gegenüber dem Flatfile keinen Vorteil und macht alles komplizierter.



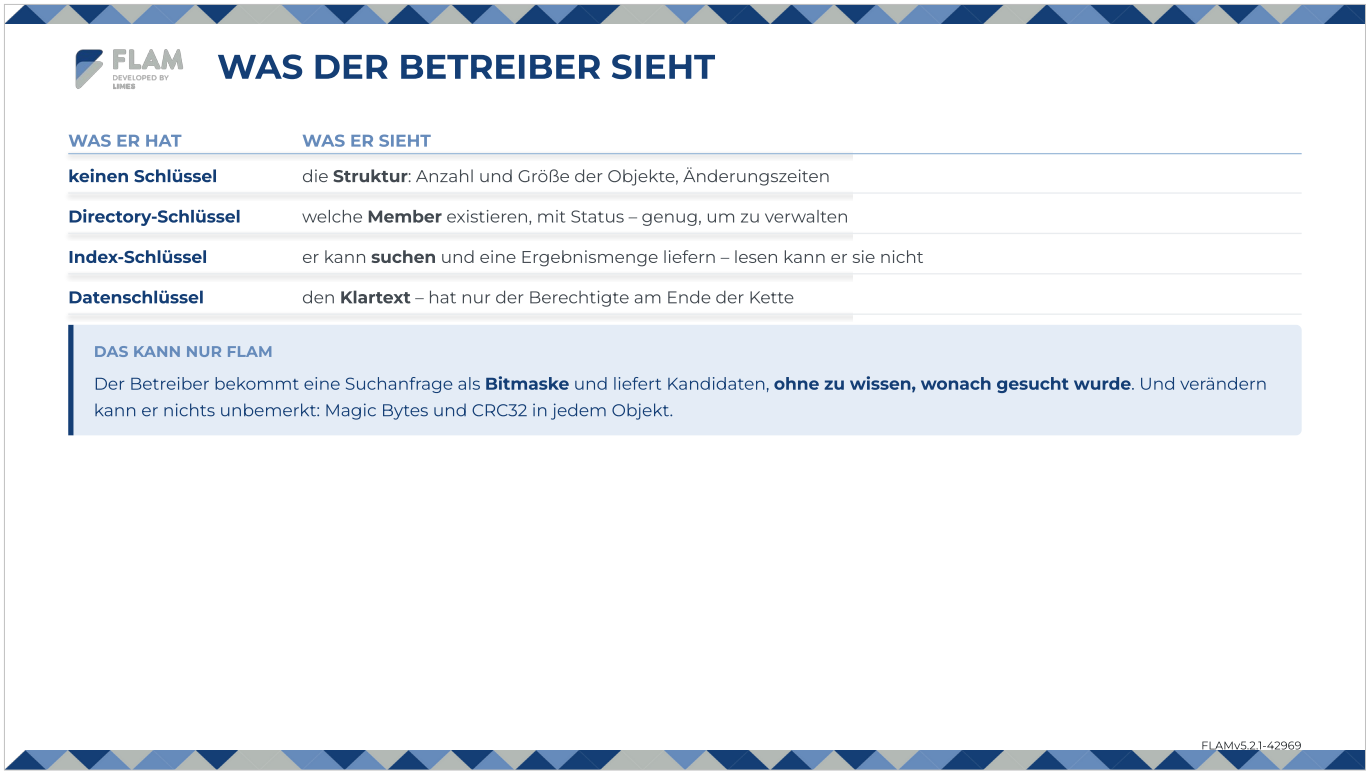
Die Folie mit den Aufrufen, und der Punkt daran ist die Gleichförmigkeit:
es ändert sich nur das `STORE`-Objekt. Das Kommando, die Member, die Schlüssel, die Suche – alles bleibt, wie es ist.
Das ist die praktische Seite der Trennung von Verwaltung und Ablage. Ein Haus, das lokal anfängt und später in die Cloud geht, ändert eine Zeile in seinen Jobs.
Zum SSH-Beispiel: das `NET`-Objekt gehört in das `STORE`-Objekt hinein, nicht daneben. Es sieht erst falsch aus und ist richtig – das Netz ist eine Eigenschaft der Ablage.
Zum S3-Beispiel: für Amazon genügen Bucket, Region und die Schlüssel; der Endpunkt wird daraus abgeleitet. Für einen S3-kompatiblen Dienst gibt man `ENDPOINT` an und schaltet `PATHSTYLE` ein.
Und zur Authentisierung ein Wort, weil es die häufigste Rückfrage ist: es gibt **fünf Wege**, und sie werden in fester Reihenfolge versucht – AssumeRole, temporäre STS-Anmeldedaten, statische IAM-Schlüssel, Web Identity über einen OIDC-Token, und Instanz-Anmeldedaten ohne jeden Schlüssel. **Empfohlen ist AssumeRole**, und die Begründung lohnt sich: der langlebige Schlüssel signiert damit **keine einzige** Objektanfrage. Er ruft nur AssumeRole, und die zurückgegebenen temporären Anmeldedaten signieren alles Weitere – erneuert eine Minute vor Ablauf. Zugriff entziehen heißt dann, die Rolle zu ändern, und nicht, ein Geheimnis in allen Jobs zu tauschen.



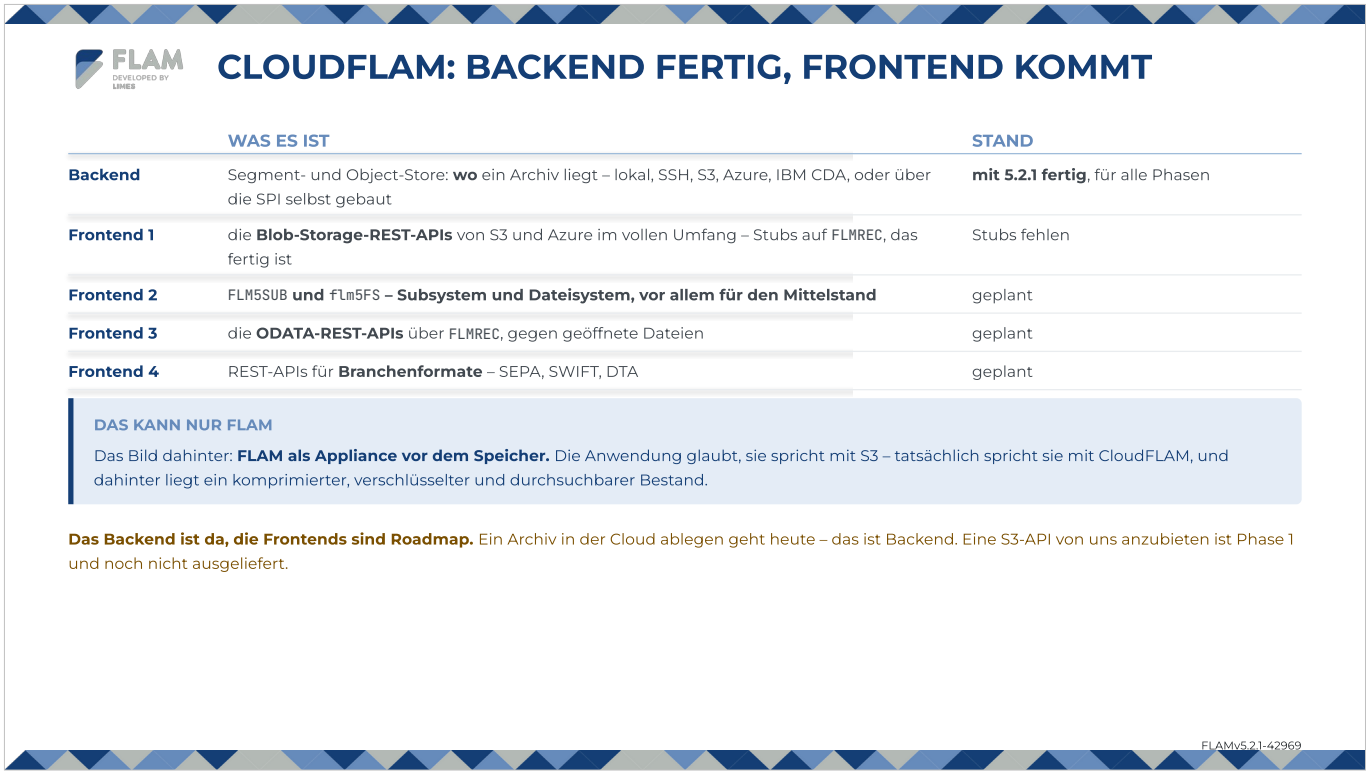
Die z/OS-Folie, und sie ist anders als die beiden nativen Cloud-Wege – deshalb eine eigene.
In einem Satz: **auf z/OS spricht FLAM nicht selbst mit der Cloud, sondern reicht die Operation an das Betriebssystem weiter.** Zuständig ist die Komponente **Cloud Data Access** von DFSMSdfp. Sie besitzt den Endpunkt, das TLS-Material und die Anmeldedaten; FLAM nennt nur den Namen einer Konfiguration und den Bucket.
Wichtig für das Verständnis: CDA ist kein Anbieter, sondern ein **Treiber**.
Welche Cloud dahinter steht, sagt die Provider-Datei – für IBM Cloud Object Storage genauso wie für S3 oder Azure. Das Kommando sieht in allen Fällen gleich aus, es ändert sich nur der Name.
Und genau das ist der Sinn: **ein Haus, das CDA eingerichtet hat, beschreibt seine Clouds einmal** – in einer Provider-Datei, die der Systemprogrammierer schreibt – und beschreibt sie nicht in jedem FLAM-Kommando noch einmal.
Deshalb hat `IBMCDA` bewusst **kein** `ENDPOINT`, `REGION`, `ACCESSKEY` oder `SECRETKEY`. Die leben in der Provider-Datei und im verschlüsselten Credential Store von CDA. Und wer sie trotzdem angibt, bekommt einen Fehler statt eines Wertes, der stillschweigend ignoriert wird – das ist Absicht.
Für die Zusammenarbeit mit dem Systemprogrammierer ein Hinweis, der viel Zeit spart, weil die Fehlermeldung nicht auf die Ursache zeigt: **die Provider-Datei muss `HEADOBJECT` unter ihren `supportedOperations` führen.** FLAM fragt über HEAD, ob ein Objekt existiert – um ein Objekt von einem Verzeichnis zu unterscheiden. Fehlt die Operation, lehnt CDA den Aufruf ab.
Von den mitgelieferten Beispielen definiert nur `IBMCOS.json` diese Operation; `S3CLOUD.json` und `AZURE.json` hören bei den sechs Datenoperationen auf. Ein S3- oder Azure-Provider wird also am besten von `IBMCOS.json` abgeleitet, oder man ergänzt die Operation in der vorhandenen Datei. Das gilt für Azure genauso wie für S3 – deshalb stehen beide auf der Folie.
Neu seit 5.2.1 und für den Betrieb die wichtigste Zahl: FLAM hält, wo der installierte CDA-Stand es anbietet, **eine Sitzung für das ganze Archiv offen** statt sich für jedes einzelne Objekt neu anzumelden. Das spart je Objekt ein Lesen der Provider-Datei, ein Lesen der Schlüsseldatei und einen TLS-Handshake. Ein älterer Stand antwortet mit X'F02', und dann steht jede Operation wieder für sich – wie bisher, ohne dass jemand etwas umstellen muss. FLAM fragt also nach und fällt zurück, statt etwas zu verlangen.
Abschalten lässt es sich mit `FL_CDA_SESSION=OFF`; das ist eine Betriebsschraube, kein Feature, über das man reden muss.
Der Satz für das Gespräch mit dem Betrieb: **wir bringen keine zweite Cloud-Konfiguration ins Haus.** Wer CDA hat, nutzt sie.



Die Folie für das Haus, das einen Speicher hat, den wir nicht kennen – oder der noch gar nicht existiert.
Es gibt **zwei** Service-Provider-Schnittstellen, und der Unterschied ist eine einzige Frage: **auf welcher Ebene setzen Sie an? `FSEG`** setzt auf der **Segmentebene** an und ist **integer-adressiert**:
FLAM vergibt einen Segmentindex, und der Provider bildet diesen Index auf einen dauerhaften Blob ab. Das passt für jeden Key-Value- oder blockadressierten Speicher. Aktiviert über `store.seguser()`.
`FOBJ` setzt auf der **Objektebene** an und ist **string-adressiert**: FLAM erzeugt für jedes Objekt einen undurchsichtigen ASCII-Schlüssel und übergibt ihn bei jedem Aufruf. Der Provider braucht nur einen flachen Key-Value-Speicher. Aktiviert über `store.objuser()`.
Der Punkt, der die Arbeit klein macht: **FLAM behält alles Schwierige.** Member-Verwaltung, Versionierung, Pufferung und Kompression bleiben bei uns. Der Provider implementiert eine **kleine Menge von Blob-Operationen**.
Und bei `FOBJ` ist die Versionierung schon erledigt, bevor der Provider sie sieht: die Version steckt als Präfix im Objektschlüssel, der Provider sieht nur einen flachen Namensraum.
Zwei Namen, nach denen im Betrieb gefragt wird: geladen wird die Bibliothek `libfseg` mit der Funktion `FSEGUSR`, beziehungsweise `libfobj` mit `FOBJUSR`. Beides lässt sich über `FL_FSEG_LIBRARY`, `FL_FSEG_FUNCTION` und die `FOBJ`-Zwillinge umstellen – und auf z/OS über SAF erlauben oder verbieten (`FSEGLIB.`, *`FSEGFUC.`*). Dazwischen sitzt je eine Brücke in unserem Code, die im Versionsbaum als `SEGSSE` und `OBJSSE` auftaucht; wer einen Provider debuggt, sieht diese Namen in den Meldungen.
Zum Anfangen gibt es beides: eine **Referenzimplementierung** (`libfsse`), die den eingebauten Flatfile- beziehungsweise Ordner-Store umhüllt, und **Vorlagen** im Quelltext – `CSRC/SAMPLE/FSEGODB.c` zeigt eine Anbindung über ODBC an PostgreSQL, MySQL, SQL Server, DB2 oder SQLite; `CSRC/SAMPLE/FOBJCLD.c` zeigt einen S3-kompatiblen Store über libcurl.
Der Satz für den Raum: **wer eine Datenbank als Archivmedium will, kann das.** Wir halten das nicht für die beste Idee, aber wir stehen nicht im Weg – und die Vorlage liegt bei.



Die Folie, die im Cloud-Gespräch entscheidet, und sie führt drei Fäden zusammen: die drei Schlüssel aus der Archiv-Einheit, die Signaturen aus der Indizierungs-Einheit und die Objektstruktur von vorhin.
Was beim Betreiber liegt, ist eine Menge von Objekten mit undurchsichtigen Schlüsseln. In ihnen stehen komprimierte und verschlüsselte Segmente.
Was er **sehen** kann, hängt davon ab, welchen Schlüssel er hat – und er hat üblicherweise keinen oder nur den Directory-Schlüssel:
Ohne Schlüssel: die Struktur. Wie viele Objekte, wie groß, wann geändert.
Mit dem Directory-Schlüssel: welche Member existieren, mit Status. Das braucht er, wenn er das Archiv verwalten soll.
Mit dem Index-Schlüssel: er kann **suchen** – also Stufe eins des zweistufigen Zugriffs ausführen und eine Ergebnismenge zusammenstellen.
Lesen kann er sie nicht.
Erst der Datenschlüssel gibt Klartext, und den hat nur der Berechtigte am Ende der Kette.
Und das Stück, das die Aussage hart macht: **die Signaturen lassen keine Rückschlüsse auf die Daten zu.** Der Betreiber bekommt eine Suchanfrage als Bitmaske und liefert Kandidaten, ohne zu wissen, wonach gesucht wurde.
Was er **nicht** kann, und das ist genauso wichtig: er kann nichts unbemerkt verändern. Magic Bytes und CRC32 in jedem Objekt, plus der Integritätsschutz der Segmente selbst.
Das ist die Folie, die man einem Datenschutzbeauftragten zeigt.



Der Ausblick – und hier muss man **zwei Seiten auseinanderhalten**, sonst entsteht genau das Missverständnis, das diese Folie vorher selbst erzeugt hat.
Das Backend ist unsere Speicherseite: der Segment- und der Object-Store, also die Frage, wo ein Archiv physisch liegt. Alles, was diese Einheit gezeigt hat, ist Backend. Und das ist **mit 5.2.1 fertig** – für die wichtigsten Cloud-Speicher, weitere kommen dazu, und über die beiden SPI kann sich ein Kunde eigene Anbindungen bauen. **Für alle vier Phasen ist das Backend damit vorhanden. CloudFLAM ist das Frontend**: die Schnittstelle, die eine **fremde Anwendung** bedient, ohne zu wissen, dass FLAM dahinter steht. Und genau diese Frontends fehlen noch.
Phase 1: die Blob-Storage-REST-APIs. Nicht "ein Archiv in S3 ablegen" – das ist Backend und geht heute. Phase 1 heißt: FLAM **bietet selbst** die REST-APIs von S3 und Azure an, und zwar im vollen Umfang. Eine Anwendung draußen glaubt, sie arbeitet gegen S3, tatsächlich ist es CloudFLAM: wir komprimieren und verschlüsseln und legen die Daten lokal, in denselben oder in einen anderen Cloud-Speicher. Technisch sind das Stubs auf unserer Satzschnittstelle `FLMREC`, die fertig ist – es fehlen die Stubs.
Phase 2: `FLM5SUB` und `flm5FS`. Subsystem und Dateisystem für FLAM5-Archive. Adressat ist ausdrücklich der **Mittelstand** auf Client-Server: eine unveränderte Anwendung liest über ein DD-Statement oder ein eingehängtes Verzeichnis.
Phase 3: die ODATA-REST-APIs. Sie nutzen ebenfalls `FLMREC` und arbeiten gegen **geöffnete Dateien**, die im Hintergrund komprimiert, verschlüsselt und in einem frei wählbaren Speicher liegen. Damit wird aus dem Archiv ein Dienst, den man abfragt.
Phase 4: REST-APIs für Branchenformate – SEPA, SWIFT, DTA für die Banken. Das ist der letzte Schritt im Frontend: die Anwendung spricht ihr Fachformat, nicht mehr eine Speicher-API.
Der Satz für die Erwartungssteuerung: **das Backend ist da, die Frontends sind Roadmap.** Wer heute in die Cloud will, kann das – als Ablage. Wer eine S3-API von uns erwartet, plant auf eine Ankündigung. Diesen Unterschied offen zu sagen kostet nichts und erspart eine Enttäuschung.



Vier, und alle drei ersten haben dieselbe Wurzel: die Cloud verhält sich nicht wie eine Platte.
Zur ersten: wer ein Archiv in einen sequenziellen Datensatz legt, verliert den wahlfreien Zugriff. Auf z/OS sind USS-Dateien die richtige Wahl – das zFS liegt physisch in einem VSAM-LDS, und ein LDS verhält sich wie eine Memory Map.
Zur zweiten: die Puffergrößen sind eine Messfrage. Wer lokal die Cloud-Einstellung fährt, liest Daten, die niemand braucht; wer in der Cloud die lokale Einstellung fährt, bezahlt Latenz pro Segment.
Zur dritten: `PRUNE` einplanen. Gelöschte Segmente bleiben über ältere Versionen erreichbar – das ist gewollt, aber ohne Aufräumlauf wächst der Speicher stetig, und in der Cloud bezahlt man ihn monatlich.
Zur vierten: die Provider-Datei bei CDA. Die Fehlermeldung zeigt nicht auf sie.



Zwanzig Minuten. Vier Fälle, und in jedem geht es darum, aus den Eigenschaften der Umgebung die Ablage abzuleiten.
Die Lösungen, wenn die Gruppe soweit ist:
Fall 1 – ein Archiv, das täglich als Ganzes zu einem Partner übertragen wird: `FILE`. Eine Datei, ein Transfer. Das Object-Format wäre hier nur lästig.
Fall 2 – ein Archiv im eigenen Rechenzentrum auf einem S3-kompatiblen Speicher: `S3` mit `ENDPOINT` und `PATHSTYLE`. Keine Amazon-Cloud nötig.
Fall 3 – z/OS, und das Haus hat CDA eingerichtet: `IBMCDA`. Der Systemprogrammierer hat die Provider-Datei; prüfen, ob `HEADOBJECT` drinsteht.
Fall 4 – der Kunde will sein Archiv in einer bestehenden Datenbank halten, weil dort seine Sicherung und sein Berechtigungskonzept schon laufen: `SEGUSER` über die FSEG-SPI, Vorlage `FSEGODB.c`. Der Hinweis gehört dazu: technisch geht es, aber man verliert die Vorteile eines Objektspeichers und sollte den Durchsatz vorher messen.

WAS BLEIBT

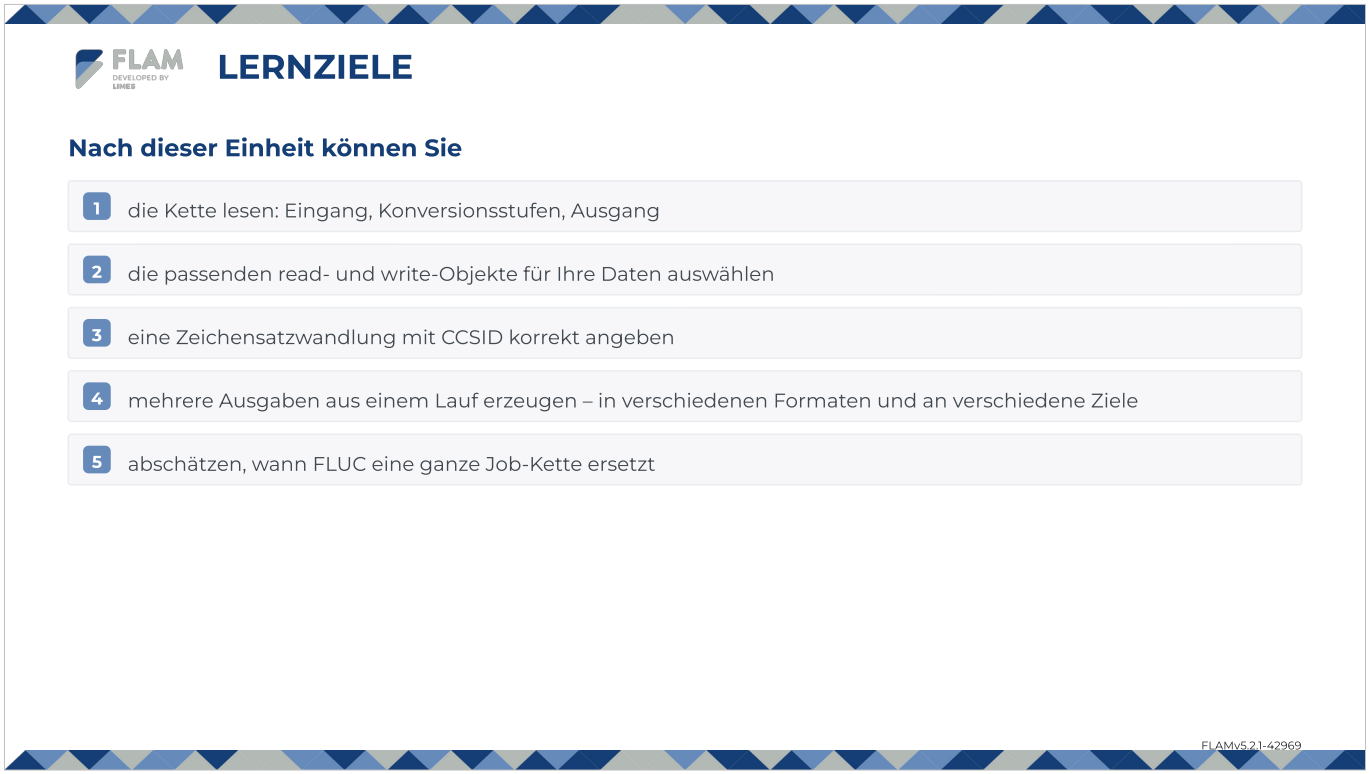
- zwei Formate: **File** für eine Datei, **Object** für viele Objekte – dasselbe Archiv, nur anders abgelegt
- das Object-Format hat **Root**, **Index je Version** und **Segment-Container**; jedes Objekt mit Magic Bytes und CRC32
- gepuffert wird gegen die **Latenz**: 4 MiB Daten-, 1 MiB Membersegmente je Objekt, einstellbar
- `FILE` und `FOLDER` gehen **auch remote per SSH**, mit Vollzugriff; `S3` spricht jeden S3-kompatiblen Dienst
- `IBMCDA` überlässt Endpunkt und Anmeldedaten der **CDA-Komponente** – das Haus beschreibt seine Cloud einmal
- ein **eigener Speicher** ist anbindbar: `FSEG` auf Segment-, `FOBJ` auf Objektebene, mit Vorlagen im Quelltext

FLAMv5.21-42969

Lernziele durchgehen. Der Satz, der hängen bleiben soll: die Ablage ist austauschbar, weil sie von der Verwaltung getrennt ist – und das heißt praktisch, dass ein Haus lokal anfangen und später in die Cloud gehen kann, ohne seine Archive anzufassen.



Drittes Funktionsthema, und das mit dem breitesten Einsatz: die meisten Kunden verdienen ihr Geld mit FLUC, nicht mit dem Archiv.
Der Kern ist eine Kette: lesen, beliebig viele Stufen konvertieren, schreiben.
Und das Besondere sind die mehrfachen Ausgänge – ein Lauf, mehrere Ziele in mehreren Formaten.



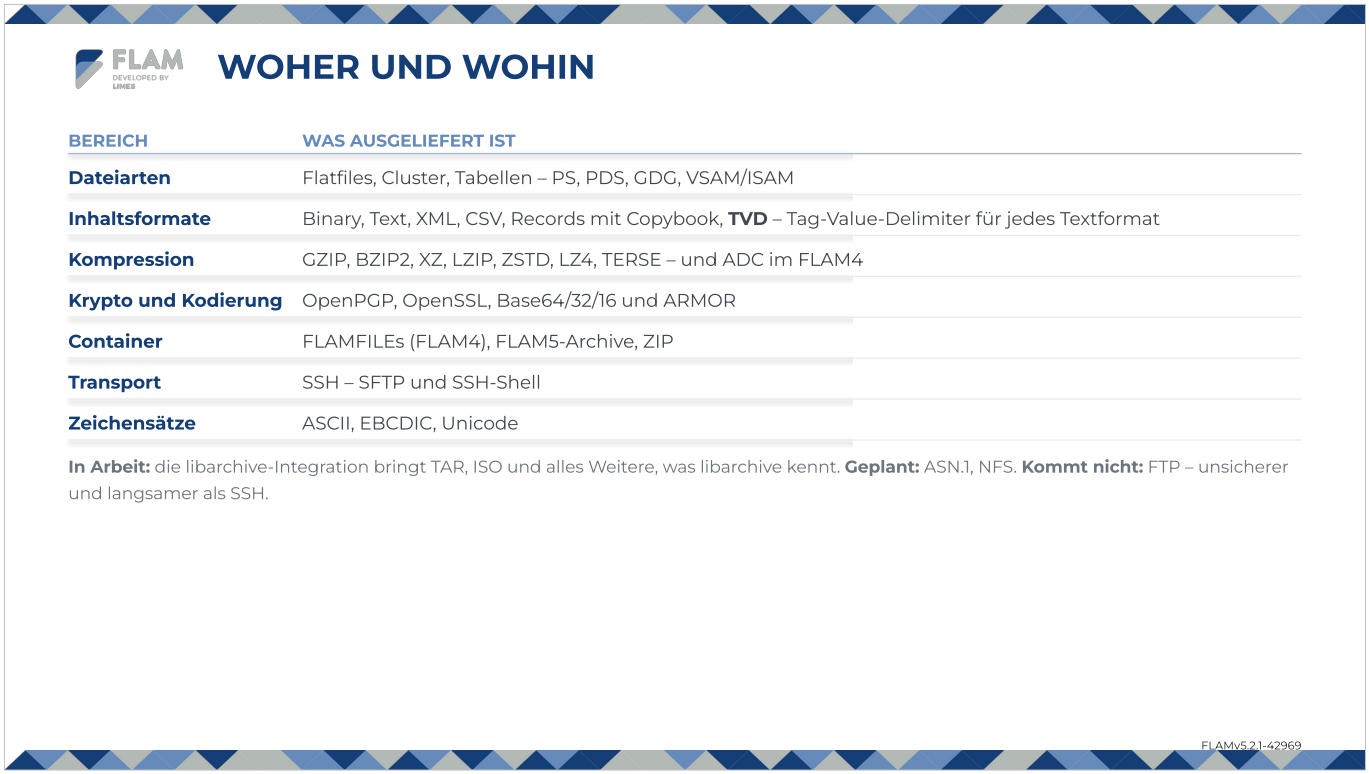
Ziel vier ist das, was die meisten Job-Ketten ersetzt und deshalb den größten Nutzen bringt.



Von links nach rechts. Der Eingang liefert einen neutralen Strom, darauf folgen die Konversionsstufen in frei wählbarer Reihenfolge, und am Ende steht ein oder mehrere Ausgänge. Wenn jemand nach einer Obergrenze fragt: das `CNV`-Array der Kommandosprache hat keine. Praktisch begrenzt es der Sinn und nicht das Werkzeug. Die eine dokumentierte Zahl in diesem Umfeld gehört zu etwas anderem und kommt auf der nächsten Folie: die Entschlüsselungen für das transparente Lesen.
Wichtig: die Stufen sind nicht vorgegeben. Komprimieren nach dem Verschlüsseln ist möglich, aber sinnlos – die Reihenfolge ist Ihre Verantwortung, und im Zweifel gilt: konvertieren, komprimieren, verschlüsseln, kodieren.



Das hier löst ein Problem, das jeder kennt, der Daten von außen bekommt: man weiß vorher nicht genau, was drin ist. Der Partner sagt "CSV", und es kommt ein ZIP mit einer BOM darin.
Zwei Mechanismen, und sie sind unterschiedlich.
Erstens `read.auto()`. FLAM erkennt das Format und liest mit den passenden Parametern. Und zwar auch verschachtelt: ein BASE64-kodiertes, PGP- verschlüsseltes, GZIP-komprimiertes FLAMFILE mit Sätzen darin wird in einem Zug aufgelöst. Dazu Textdateien mit Trennzeichen, XML, Base16/32/64, GZIP, BZIP2, XZ, satzorientierte Datasets FB/VB/VSAM, FLAMFILEs und Binärdaten.
Der praktische Fall, den man nennen sollte, weil er im Support oft auftaucht:
eine USS-Datei, die mit FILEDATA=RECORD erzeugt und jetzt mit FILEDATA=BINARY alloziert ist – `read.auto()` liest sie trotzdem richtig, weil es die 4-Byte-Längenfelder erkennt und daraus wieder Sätze baut.
Die Ehrlichkeit gehört dazu, und sie steht so in der Manpage: die Erkennung arbeitet auf Best-Effort-Basis und man soll sich nicht darauf verlassen. Sie war ursprünglich für die Anzeige gebaut – FLVIEW und FLVEDIT unter ISPF benutzen genau das. Bei sehr kleinen Dateien kann sie fehlschlagen.
Also die Regel: `read.auto()` für Anzeige, Analyse und Einmalfälle. In der Produktion sagt man dem Werkzeug, was kommt.
Zweitens, und das ist der Produktionsfall: `DECRYPT` ist ein **Array**. Man gibt mehrere Entschlüsselungsmöglichkeiten an, und FLAM nimmt die, die passt.
Gesteuert wird das über SKPKEY – überspringen, wenn der Schlüssel nicht passt – und SKPFMT – überspringen, wenn das Format nicht passt. Eine Mengengrenze gibt es nicht.
Der Fall, der die meiste Arbeit erspart und den man nennen sollte: bei public-key-verschlüsselten PGP-Dateien braucht man beim Lesen oft **gar keine DECRYPT-Angabe**. Der Schlüssel wird über die Key-ID automatisch gezogen; man sagt der Umgebung nur, welche kryptographische Infrastruktur der FKM5 nutzen soll. Auf dem Host ist das per Default ICSF – dort muss man also nichts angeben.
Dasselbe Prinzip gilt für Formate: im TABLE-Objekt kann man beliebig viele Formatbeschreibungen mitgeben, und erkannt wird, welche passt.
Warum das zählt: ein Empfangsprozess, der Dateien von zehn Partnern annimmt, braucht damit nicht zehn Jobs und keine Vorabprüfung, welcher Partner geliefert hat.



Die Liste ist absichtlich vollständig gehalten, damit die Teilnehmer die Bandbreite sehen – nicht zum Vorlesen.
Der Punkt: Quelle und Ziel sind orthogonal zur Konversion. Eine Datei vom Host kann als Member in einem ZIP auf einem entfernten Linux landen, ohne dass sich an der Konversion etwas ändert. `read.auto` ist der Einstieg für Unbekanntes: FLUC erkennt das Format selbst.
Wichtig für die Glaubwürdigkeit: **die Tabelle nennt nur, was ausgeliefert ist.** Was in Arbeit ist, steht darunter und wird als solches gesagt. Ein Kunde, der ein Feature einplant und es dann nicht findet, kostet mehr Vertrauen als eine kürzere Liste.
Zur libarchive-Integration: damit kommt auf einen Schlag alles, was libarchive kennt – TAR und ISO sind nur die zwei, nach denen am häufigsten gefragt wird.
Und zu FTP die klare Aussage, wenn danach gefragt wird: **FTP kommt nicht.** Es ist unsicherer und langsamer als SSH, und wir haben SSH. Das ist eine Entscheidung, kein Rückstand.
Ein Inhaltsformat verdient einen eigenen Satz, weil es unterschätzt wird:
TVD, Tag-Value-Delimiter. Damit lässt sich **jedes** Textformat beschreiben, das aus Marke, Wert und Trennzeichen besteht. Gedacht war es primär für **SWIFT-MT** – und genau dafür wird es auch benutzt. Aber es trägt mehr: über TVD lesen und schreiben wir auch **JSON**. Für JSON ist zusätzlich ein eigener, spezialisierter Support geplant, analog zu dem für XML; bis dahin ist TVD der Weg, und er funktioniert.
Wer also nach einem Format fragt, das nicht in der Tabelle steht, sollte als erstes gefragt werden, ob es sich als Marke-Wert-Trennzeichen beschreiben lässt. Meistens lässt es sich.



Die Nachschlagefolie, die im Handbuch als Matrix steht und die man einmal gesehen haben muss – weil zwei Kombinationen **nicht** gehen und eine stillschweigend etwas anderes tut, als der Kunde erwartet.
Zuerst die Leseseite in einem Satz je Objekt. `READ.RECORD` behält die Satzgrenzen und wandelt nur, wenn eine `CCSID` dasteht. `READ.TEXT` **wirft die Satzlänge weg**, trennt an allen üblichen Zeilentrennern und wandelt automatisch. `READ.CHAR` liest alle Zeichen, die Trenner gehören zu den Daten, Ergebnis sind Blöcke. `READ.BINARY` liest Bytes, wie sie sind.
Die Zeile, die man sich merken muss: **`READ.CHAR` oder `READ.BINARY` nach `WRITE.TEXT` ist ein Fehler** – kein stiller Umweg, sondern ein Abbruch.
Der Grund ist logisch: aus einem Block ohne Satzgrenzen kann `WRITE.TEXT` keine Zeilen machen, und raten soll es nicht.
Die zweite Zeile: **Block nach Satz ist erlaubt, aber selten sinnvoll.** Der Block wird in Sätze zerschnitten, vorbelegt in variable Sätze mit einem kürzeren Restsatz am Ende. Wer das braucht, weiß es; wer es versehentlich tut, wundert sich über die Satzlängen.
Und die Kombination, die man am häufigsten braucht, ist die unauffälligste: `READ.RECORD` nach `WRITE.TEXT` ergänzt je Satz einen Trenner – das ist der Weg vom Host in die offene Welt, und der eigene Trenner ist wählbar.
Umgekehrt (`READ.TEXT` nach `WRITE.RECORD`) entstehen üblicherweise variable Sätze; bei festen Sätzen wird aufgefüllt.

ZEICHENSÄTZE RICHTIG ANGEBEN

```
read.text(file='ein.txt' ccsid='IBM-1141')
write.text(file='aus.csv' ccsid='UTF-8')
```

- die CCSID beschreibt die **Datei**, nicht die Plattform – eine Hostdatei bleibt EBCDIC, auch wenn Sie sie unter Linux lesen
- Lesen und Schreiben haben **je eine eigene** CCSID; nur eine anzugeben ist die häufigste Ursache für kaputte Umlaute
- der deutsche Klassiker: `IBM-1141` statt `IBM-1047` – der Unterschied sind genau die Umlaute und das Euro-Zeichen
- die Wandlung macht unser eigenes Modul (`ICV`) – nicht libiconv; es kennt Normalisierung, Subsets, Best-Fit-Mapping, Reporting und Casing

Wenn nach einer Konvertierung „nur die Umlaute“ falsch sind, ist es fast immer die CCSID – und fast immer die des Eingangs.

FLAMv5.21-42969

Die häufigste Fehlerquelle in der Praxis, und sie ist trivial zu vermeiden:
die CCSID muss zur Datei passen, nicht zur Plattform.
Der Klassiker ist eine Hostdatei in 1141 (deutsche EBCDIC-Codepage mit Euro-Zeichen), die als 1047 gelesen wird – dann stimmen genau die Umlaute nicht, und das fällt erst im Fachbereich auf.
Zweiter Klassiker: nur eine Seite angeben. Lesen und Schreiben haben jeweils ihre eigene CCSID.
Wenn Sie nichts sagen, tut FLAM Folgendes – und das ist wissenswert, weil es beide Richtungen unterschiedlich behandelt:
Beim Schreiben nehmen wir immer den lokalen Zeichensatz der Maschine.
Beim Lesen detektieren wir. Das kostet CPU, und bei Singlebyte- Zeichensätzen ist es nicht eindeutig. EBCDIC lässt sich gut von ASCII und UTF-8 unterscheiden – ob es aber deutsches oder amerikanisches EBCDIC ist, steht in den Bytes nicht drin. Deshalb entscheiden wir das am **Länderkennzeichen der Maschine**: auf einem deutsch eingestellten System wird erkanntes EBCDIC zu 1141, auf einem amerikanisch eingestellten zu 1140.
Daraus folgt die Empfehlung für die Produktion: **CCSID angeben.** Dann muss nichts detektiert werden, es kostet keine CPU, und der Lauf verhält sich auf jeder Maschine gleich – unabhängig davon, wie jemand sie eingestellt hat.



Diese Folie klärt eine Verwechslung, die auf dem Host regelmäßig einen halben Tag kostet. Zeichensätze gibt es nämlich nicht nur **in den Daten**.
Es sind drei Ebenen, und sie können alle drei verschieden sein:
Erstens die **Daten**, die verarbeitet werden. Zweitens der Zeichensatz, in dem wir **bedient** werden – also in dem der Parameterstring vorliegt.
Drittens die **Parameterdateien**, die von anderen Plattformen geholt werden.
Das Beispiel ist echt und lohnt es, langsam durchzugehen: Sie sitzen auf dem Host und schreiben JCL, aber Ihr Editor steht auf 1140. Die Daten, die Sie lesen, sind in 1141. Und die Parameterdatei, die das Datenformat beschreibt, liegt auf einem entfernten Linux-Server in UTF-8.
Drei Zeichensätze in einem Lauf. Jeder braucht seine eigene Angabe:
Im JCL-Inline schreiben Sie am Anfang `&1140;` hin – damit lesen wir die Statements richtig, nämlich in dem Zeichensatz, in dem Ihr Editor sie geschrieben hat.
In der URL der entfernten Parameterdatei hängen Sie den Zeichensatz an: `ssh://user@server/format/myf.clp.txt/&UTF-8`.
Im CONV-Statement setzen Sie die CCSID der Daten – damit nichts detektiert werden muss, und damit die Ausgabe nicht im lokalen Zeichensatz landet, wenn sie das nicht soll.
Wer diese drei Ebenen einmal auseinandergehalten hat, versteht auch, warum "die Umlaute sind kaputt" fünf verschiedene Ursachen haben kann.



Das ist die Funktion, die Kunden überrascht und die die größte Einsparung bringt.
Multiple Writes heißt: mehrere Konvertierungen desselben Eingangs – einmal CSV in UTF-8 als PGP-Datei, einmal EBCDIC klar in einen FB-Datensatz.
Multiple I/O heißt: dieselbe Ausgabe an mehrere Ziele – die PGP-Datei per SSH zum Partner, dieselbe direkt auf die Platte, und zusätzlich als Member in einem ZIP für das Backup.
Beides gibt es im Batch-Utility und über die APIs.

EIN BEISPIEL AUS DER PRAXIS

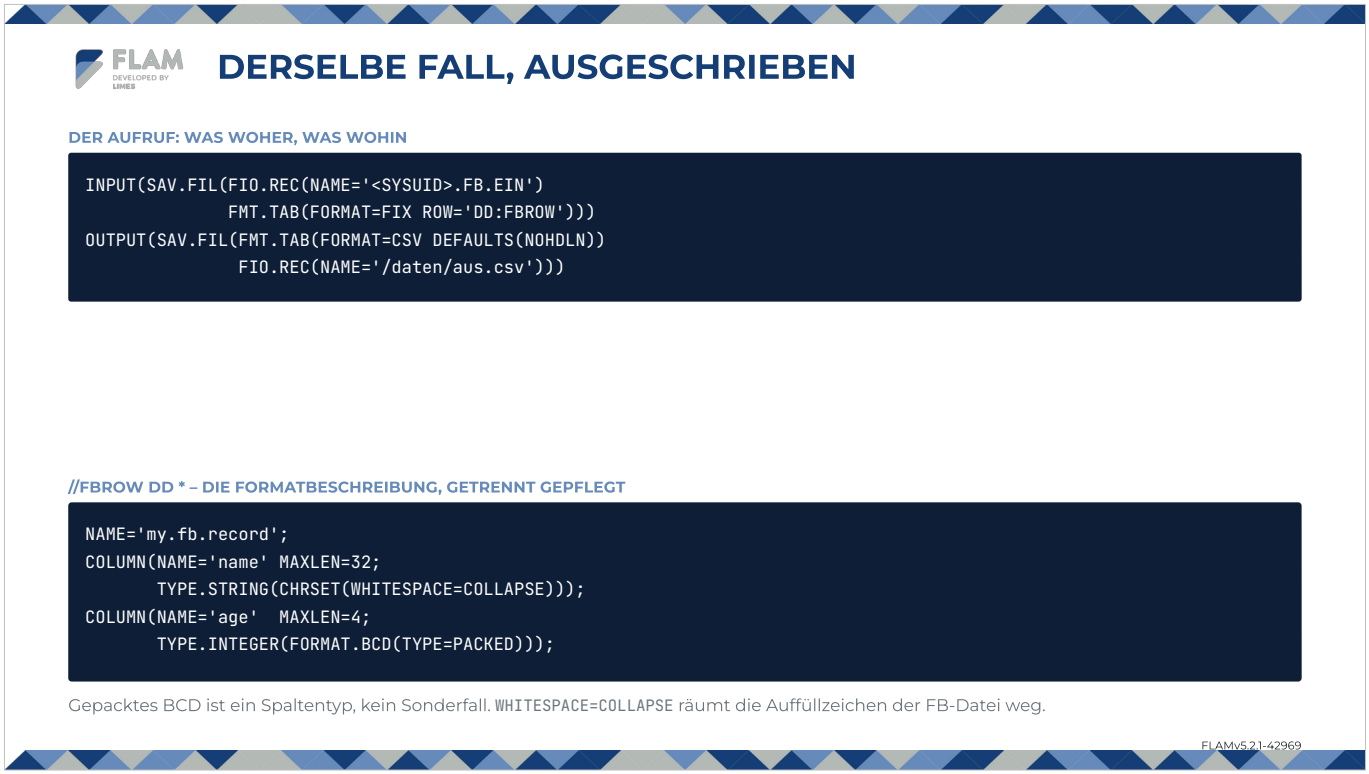
Eine Altanwendung schreibt auf dem Host FB oder VB, EBCDIC-Strings und BCD-Zahlen. Sie darf nicht angefasst werden.

- Tabellenunterstützung zerlegt den Satz in Spalten und weist ihnen Typen zu
- die Zahlen werden aus BCD in ihre externe Darstellung gewandelt, die Strings von EBCDIC nach UTF-8
- die Ausgabe entsteht als **CSV mit Unix-Zeilenenden**
- danach **OpenPGP-verschlüsselt**, per **SSH** übertragen und als **Member eines ZIP-Archivs** abgelegt

Ein Schritt. Und wenn der Weg über das FLUC-Subsystem oder flucFS läuft, merkt die Altanwendung davon nichts.

FLAMv5.21-42969

Das Beispiel steht so in den Altfolien und ist das beste Verkaufsargument, weil es eine ganze Modernisierung in einem Schritt beschreibt.
Ausgangslage: eine Altanwendung schreibt auf dem Host einen FB- oder VB-Datensatz mit EBCDIC-Strings und BCD-Zahlen. Niemand darf sie anfassen.
Ergebnis: ein CSV-Strom in UTF-8 mit Unix-Zeilenenden, OpenPGP-verschlüsselt, per SSH als Member in ein ZIP-Archiv auf einem entfernten Linux-System.
Und die Altanwendung merkt davon nichts, wenn der Weg über das Subsystem läuft.



Die Folie davor beschreibt den Fall, diese zeigt ihn vollständig. Es ist Beispiel 15 aus dem Handbuch, und es ist bewusst klein gehalten – zwei Spalten, damit man den Aufbau sieht und nicht den Umfang.

Die Aufgabe: ein FB-Dataset mit zwei Spalten wird eine CSV-Datei ohne Kopfzeile. Die erste Spalte ist ein mit Leerzeichen aufgefüllter Name in EBCDIC, bis zu 32 Byte. Die zweite ist eine **gepackte BCD-Zahl** über 4 Byte mit dem Alter. Für die CSV sollen die Leerzeichen im Namen zusammengefasst werden.

Der Aufbau, den man mitnehmen soll: **der Aufruf sagt nur, was woher kommt und wohin geht – die Formatbeschreibung steht daneben.** Das `ROW`-Objekt wird über `ROW='DD:FBROW'` aus einer eigenen DD-Karte gelesen. Damit kann dieselbe Beschreibung von mehreren Jobs benutzt werden, und die JCL ändert sich nicht, wenn sich das Format ändert.

Zwei Details, die im Raum meist eine Frage auslösen:

`TYPE.INTEGER(FORMAT.BCD(TYPE=PACKED))` – damit ist gepacktes BCD ein Spaltentyp und kein Sonderfall. Niemand muss eine Umrechnung schreiben.

`TYPE.STRING(CHRSET(WHITESPACE=COLLAPSE))` – führende, angehängte und wiederholte Leerzeichen fallen weg. Das ist der Griff, der bei FB-Daten fast immer gebraucht wird, weil dort mit Leerzeichen aufgefüllt wird.

Und die Zeichensatzfrage ist hier gar keine: geschrieben wird im lokalen Zeichensatz, also in EBCDIC, weil nichts anderes gesagt wird. Wer UTF-8 will, sagt es am `WRITE`.

DASSELBE ÜBER DIE DREI WEGE

WEG	WIE	WANN
Batch-Utility	`flcl CONV read...(...) write...(...)`	geplante Verarbeitung, Job-Ketten
Subsystem, Filesystem	FLUC-Subsystem auf z/OS, flucFS auf Linux	wenn die Anwendung nichts merken darf
API	Byte-Interface oder FLUC-UP im Programm	wenn Ihr Programm den Strom selbst verarbeitet

Bei der Konvertierung sind alle drei Wege gleichwertig – die Wahl ist rein organisatorisch.

FLAMv5.2.1-42969

Bei der Konvertierung sind alle drei Wege gleichwertig – das ist der Unterschied zum Archiv-Thema, wo nur das API den vollen Zugriff gibt.
Deshalb ist die Wahl hier rein organisatorisch: wer darf was ändern.



Das ist die Folie, bei der im Raum jemand nickt – weil er diese Kette kennt. Sie ist der Klassiker: eine FBA-Datei in EBCDIC soll als Textdatei auf einer Unix-Maschine liegen, verifiziert.
Die neun Schritte langsam vorlesen und dabei mitzählen. Jeder einzelne ist sinnvoll, keiner ist schlecht gebaut – sie sind so, weil jedes Werkzeug genau eine Sache kann:
Erst eine REXX, die die Druck-Steuerzeichen entfernt oder ersetzt. Dann `iconv` von EBCDIC nach UTF-8. Dann wieder eine REXX, die an jeden Satz ein `0x0A` anhängt. Dann die Datei nach USS übertragen. Dann eine SHA2-Prüfsumme rechnen. Dann per SSH auf die Unix-Maschine übertragen. Dann die Prüfsumme dort per SSH verifizieren. Dann die Verarbeitung dort starten. Und zuletzt die USS-Datei löschen.
Neun Steps, zwei selbstgeschriebene REXX-Programme, eine Zwischenkopie in USS und zwei SSH-Sitzungen. Und jeder der neun kann einzeln scheitern, mit je eigener Fehlerbehandlung.
Was davon in **einem** FLCL-Aufruf steckt: die Druckaufbereitung ist `PRNCONTROL`, die Zeichensatzwandlung ist `CCSID`, das Zeilenende ist `METHOD=UNIX`, das Übertragen ist die SSH-URL im `WRITE`, die Prüfsumme ist `HASH`, und das Starten der Verarbeitung ist ein `POSTPROCESS`. Die Zwischenkopie in USS entfällt vollständig, weil direkt geschrieben wird.
Der Punkt ist nicht die Tipparbeit. Der Punkt ist, dass acht Fehlerbehandlungen und eine Zwischenkopie verschwinden – und mit ihnen die Frage, in welchem der neun Schritte der Job letzte Nacht hängengeblieben ist.



Die Folie davor sagt, wo jeder Step landet – diese zeigt das Ergebnis.
Vorlesen muss man sie nicht, drei Dinge zeigen genügt.
Erstens: **oben steht kein FLAM-Wissen.** Der Step ist gewöhnliche JCL – STEPLIB, die Eingabedatei per DD, der Parameterstring in einem DD statt im PARM, weil er dort beliebig lang sein darf.
Zweitens die Leseseite: `PRNCONTROL=REPLACE` macht aus der FBA-Datei **druckaufbereitete Sätze** – der Seitenvorschub wird zur Leerzeile, das Steuerzeichen verschwindet aus den Daten. Das war die erste REXX. `CCSID` sagt, was hereinkommt. Mehr braucht die Leseseite nicht: von 1141 nach UTF-8 ist jedes Zeichen abbildbar, eine Ersetzungsregel ist also kein Thema.
Drittens die Schreibseite: `SUPTWS` nimmt das **FB-Padding** weg – ohne das wäre jede Zeile auf Satzlänge mit Leerzeichen aufgefüllt. `METHOD=UNIX` setzt das `0x0A` an jeden Satz, das war die zweite REXX. `COMP.GZIP()` komprimiert **vor** der Übertragung; das ist kein Extra, sondern der Normalfall – unkomprimiert über die Leitung ist teuer. Die `ssh://`-URL ersetzt Übertragung und Zwischenkopie in USS.
Zur URL zwei Details, nach denen oft gefragt wird: das `~` steht für das Home-Verzeichnis des Benutzers. Ein absoluter Pfad bräuchte **zwei** Schrägstriche hinter dem Server. Und `<cuser>` ist hier ein Platzhalter für die Kennung auf der Gegenseite.
Dann die Prüfsumme: `HASH` rechnet sie und legt sie über dieselbe Verbindung daneben. `MEMBER` ist die Angabe, die man leicht übersieht – sie bestimmt, welcher **Dateiname in der Prüfsummenzeile** steht. Ohne sie steht dort der Name des Ziels, also die ganze URL, und damit findet ein `sha256sum -c` auf der Gegenseite nichts. Mit einem Namen relativ zum Home passt es, denn dort läuft das Kommando.
Zuletzt die beiden `POSTPROCESS` auf der Gegenseite, nur bei Erfolg:
Prüfsumme verifizieren, dann auspacken. Es ist ein Array, sie laufen in der angegebenen Reihenfolge. Und das Kommando läuft **ohne Shell** – `cd /daten && ...` funktioniert deshalb nicht, wer eine Verkettung braucht, ruft ein Skript.
Ein Hinweis für die, die es mit 5.2.1 nachbauen: die Prüfsummendatei wurde dort erst nach dem Postprocessing beschrieben, das `sha256sum -c` im selben Aufruf fand sie leer. Ab 5.2.2 ist sie vorher herausgeschrieben.
Und das Semikolon am Ende jeder Zeile ist Absicht: es ist das Kommentarzeichen der Kommandosprache. Steht im ISPF-Editor `NUMBER=ON`, landen die Zeilennummern in den Spalten 73 bis 80 – hinter dem Semikolon sind sie ein Kommentar, und der Job läuft trotzdem. Ohne das Semikolon bricht derselbe Job mit einem Syntaxfehler in Spalte 73 ab (nachgefahren).



Die Folie, die die Frage beantwortet, die nach dem Jobstep kommt: **warum gibt es zwei Konvertierungskommandos?** Die Antwort ist eine Rangfolge, nicht eine Auswahl: **`CONV` ist eine Vereinfachung von `XCNV`.** `XCNV` stellt den FLUC eins zu eins nach außen zur Verfügung – jede Fähigkeit, die die Konvertierungsmaschine hat, ist dort erreichbar. `CONV` legt eine bequeme Schicht darüber und trifft Entscheidungen für Sie.

Was das Handbuch in Abschnitt 3.14 gegenüberstellt, und es lohnt sich, das vorzulesen: `CONV` ist "simple to use, more automatisms, fewer parameters", der Anwender "must not know how the data is formatted or where it came from", die Konvertierung läuft "on a best-fit basis" – und: "there are no possibilities to do dirty things". Bei `XCNV` steht dagegen "gives access to all capabilities of FLUC", "the user must know how the input data is formatted", "conversions are limited to a minimum of CPU usage" und "dirty things ... are possible".

Drei Dinge folgen daraus für die Produktion.

Erstens: Determinismus. `XCNV` hat weniger Automatismen, und jeden einzelnen muss man ausdrücklich einschalten. Damit hängt das Ergebnis nicht davon ab, was in der Datei von heute erkennbar war. Genau das will man in einem Nachtjob.

Zweitens: CPU. Die Automatik von `CONV` wandelt gern über einen Zwischenschritt – von X nach UTF-8 und dann nach Y. In `XCNV` schreibt man `CNV.CHR(FROM=1141 TO='UTF-8')` an die Stelle, an die es gehört, und spart eine Wandlung. Das Handbuch sagt es als Eigenschaft: minimaler CPU-Verbrauch.

Drittens: Fähigkeiten, die es nur dort gibt. Und hier muss man genau sein: **mehrere Ausgaben kann `CONV` auch** – `WRITE` ist ein Array, das entspricht den mehreren `OUTPUT` beim `XCNV`. Der Unterschied liegt eine Ebene tiefer: beim `XCNV` ist **auch das `FIO` je Ausgabe ein Array**. Eine Ausgabe kann also in mehrere Dateien zugleich schreiben – mit `CONV` ist das nicht erreichbar, dort hat jedes `WRITE` genau eine Datei. Dazu kommt **Vor- und Nachverarbeitung je Datei** statt nur global. Wer das braucht, braucht `XCNV`.

Und der Preis, den man nennen muss: man muss wissen, was man tut. "Dirty things" sind möglich, und die Parameternamen sind die internen – aus `PRNCONTROL` wird `PRNCTR`, weil das der Name in der Maschine ist.



Dieselbe Aufgabe wie zwei Folien davor, nur in `XCNV` – und man sollte die beiden Folien nebeneinander zeigen, weil man den Unterschied dann sieht, statt ihn zu erklären.
Die Struktur ist die des FLUC selbst, und dafür gibt es einen Merksatz, den man sich an die Wand hängen kann: **der FLUC ist `FIO` `CNV`…`CNV` `FMT` – `FMT` `CNV`…`CNV` `FIO`.** Beim Lesen kommt zuerst die Datei, dann die Wandlungen, dann der Formatierer, der daraus Elemente macht; beim Schreiben läuft dieselbe Kette rückwärts. Genau in dieser Reihenfolge stehen die Stufen in `SAVE.FILE` – `PRE`/`POST` kommen für die Umgebung dazu.
Lesen: `FIO.REC` mit `PRNCTR=REPLACE` – derselbe Griff wie im `CONV`, nur mit dem internen Namen. Danach `CNV.CHR(FROM=1141 TO='UTF-8')`: **eine** Wandlung, ausdrücklich angegeben, kein Umweg. Und am Ende steht `#FMT.BIN()#` – auskommentiert, denn `BIN` ist der Standardformatierer der Eingabe. Es steht trotzdem da, damit man die Kette vollständig sieht; `#…#` ist der Kommentar der Kommandosprache.
Schreiben, von oben nach unten in der Reihenfolge der Verarbeitung: `FMT.TXT(METHOD=UNIX SUPTWS)` macht aus den Elementen Text mit `0x0A` als Trenner und nimmt das FB-Padding weg – Formatierung gehört in die `FMT`-Stufe, nicht in eine Konversion. `CNV.GZP()` komprimiert.
`CNV.HSH()` rechnet die Prüfsumme und legt sie über dieselbe Verbindung daneben. `FIO.BLK` schreibt die Datei über `ssh://`. Und `POST` führt auf der Gegenseite das `sha256sum -c` aus.
Der Gewinn gegenüber der `CONV`-Fassung ist nicht die Kürze – die Fassung ist länger. Der Gewinn ist, dass **jede Stufe dasteht**: keine Erkennung, keine Vorbelegung, die morgen anders greift. Wer einen zweiten Empfänger braucht, hängt ein zweites `OUTPUT` an, und die Leseseite läuft trotzdem nur einmal.

STOLPERFALLEN

! **Nur eine CCSID angegeben.**
Lesen und Schreiben haben je eine eigene. Fehlt eine, gilt ein Standard – und der passt selten.

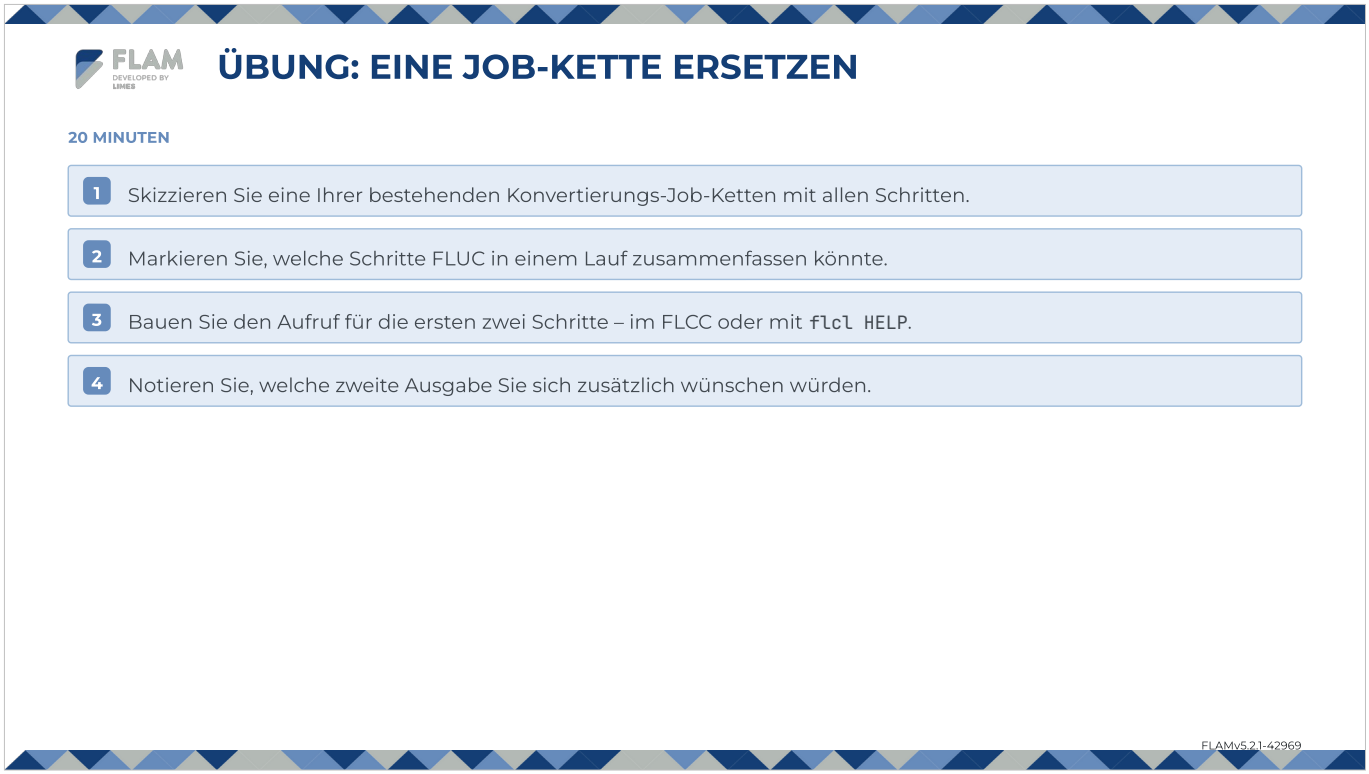
! **Komprimieren nach dem Verschlüsseln.**
Technisch erlaubt, praktisch sinnlos: verschlüsselte Daten sind nicht mehr komprimierbar. Reihenfolge prüfen.

! **Textmodus für binäre Daten.**
`read.text` interpretiert Zeilenenden. Für binäre Daten `read.binary` – sonst gehen Bytes verloren.

! **Mehrere Ausgaben, aber ein Fehlerpfad.**
Bei Multiple Writes muss klar sein, was gilt, wenn ein Ziel scheitert. Das gehört in die Kettenlogik, nicht in die Annahme.

FLAMv5.21-42969

Alle vier aus Supportfällen der Konvertierung.



Zwanzig Minuten. Ziel ist nicht die perfekte Kette, sondern die Erkenntnis, wie viele Schritte zusammenfallen.
Auflösung:
1. Der Eingang ist ein `READ`-Objekt passend zur Quelle: `read.record` für Datasets, `read.text` für Textdateien, `read.binary` für Binärdaten.
2. Die Stufen in der sinnvollen Reihenfolge: konvertieren, komprimieren, verschlüsseln, kodieren. Wer komprimiert, nachdem er verschlüsselt hat, komprimiert Zufall – das kostet CPU und bringt nichts.
3. Der Ausgang bestimmt das Zielformat, und die Zielplattform muss der Quellplattform nicht gleichen.
4. Wenn zwei Ziele gebraucht werden: zwei `WRITE`, nicht zwei Läufe. Gelesen und gemeinsam konvertiert wird dann nur einmal.
Die erwartete Erkenntnis: die alte Kette hatte einen Schritt je Werkzeug, die neue hat einen Schritt für alles – und der teuerste Posten der alten Kette war nicht die Rechenzeit, sondern der I/O für die temporären Dateien.

WAS BLEIBT

- lesen, konvertieren, schreiben – beliebig viele Stufen, alles in einem Schritt
- Quelle und Ziel sind orthogonal zur Konversion
- die CCSID beschreibt die Datei, nicht die Plattform – und es sind zwei
- Multiple Writes und Multiple I/O: einmal lesen, mehrfach ausgeben
- in der Praxis ersetzt das ganze Job-Ketten – dort liegt der Nutzen

FLAMv5.21-42969

Lernziele durchgehen. Die nächste Einheit geht eine Ebene tiefer: die Tabellenunterstützung, mit der aus einem Satz Spalten werden.



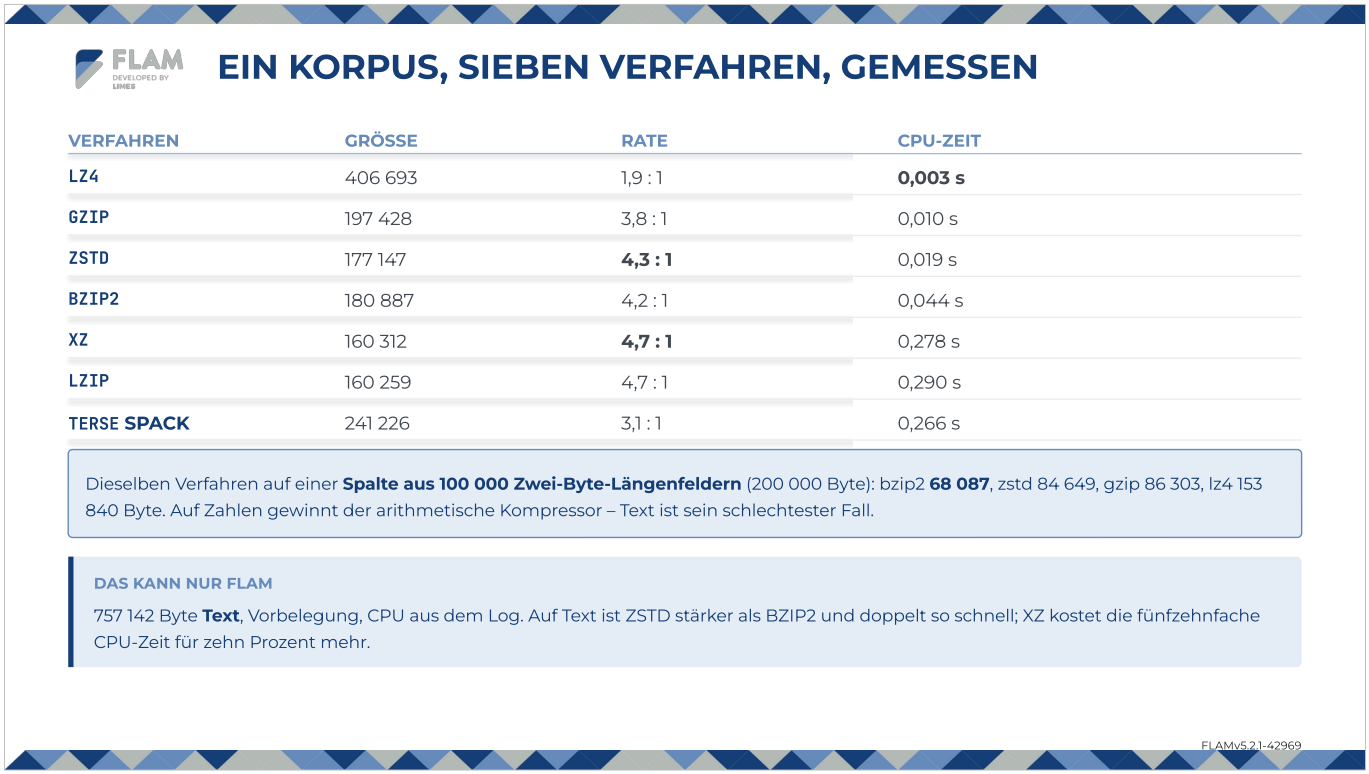
Die Einheit, in der eine Zahl mehr wert ist als eine Meinung.
Der Bogen: die sieben Verfahren, die Stufen, ein gemessener Vergleich, was ein Kopf speichert und was ohne Kopf verloren geht, TERSE für die Host-Welt, und die drei Fälle, in denen man **nicht** komprimiert.
Der Satz für den Anfang: die Frage ist nie „welches ist das beste", sondern **„was ist hier knapp"** – Platz, CPU-Zeit oder die Leitung. Für jede der drei Antworten gibt es ein anderes Verfahren, und der Unterschied ist groß: zwischen dem schnellsten und dem stärksten liegen in unserer Messung **Faktor 90 in der CPU-Zeit** und Faktor 2,5 in der Größe.



Das zweite Ziel ist das, weswegen die Einheit existiert – die Antwort hängt an der Frage, was knapp ist, und die stellt fast niemand.
Das fünfte ist der Punkt, an dem Geld verloren geht: wer schon komprimierte Daten noch einmal komprimiert, zahlt CPU für mehr Bytes.



Die Bestandsaufnahme. Die Spalte rechts ist die wichtige – sie sagt, woher das Verfahren kommt, und damit, wer es auf der Gegenseite lesen kann.
`GZIP` ist der Standard für den Austausch: jedes System kann es, und es ist das einzige mit einem **Kopf für Metadaten**.
`ZSTD` ist der Neue und in der Praxis meist die richtige Wahl: fast die Rate von XZ bei einem Fünfzehntel der CPU-Zeit. **Er hat LZ4 und LZMA überholt** – bei niedriger Stufe ist er so schnell wie LZ4 und besser, bei hoher so gut wie LZMA und braucht dafür deutlich weniger Speicher und CPU.
`BZIP2` ist der Sonderfall, und den muss man richtig einordnen: er ist **kein Mustersucher, sondern ein arithmetischer Kompressor** – der beste bekannte. Damit ist er von ZSTD **nicht** überholt, sondern für **Zahlen und binäre Daten** oft die bessere Wahl. Auf Text, dem Korpus der nächsten Folie, ist er der schlechteste seiner Klasse.
`XZ` und `LZIP` sind auf LZMA gebaut, in unserer Messung gleich stark und mit Abstand die langsamsten – heute von ZSTD überholt.
`LZ4` ist der Schnelle. Er komprimiert kaum, und ZSTD auf niedriger Stufe nimmt ihm auch diese Nische ab.
Für die Vollständigkeit, wenn jemand nach älteren Verfahren fragt: die baumbasierten mit Thesaurus – ADC, LZW – finden ebenfalls nur Wiederholungen, brauchen beim Dekodieren aber viel länger, verschlingen Speicher, und der Aufbau des Thesaurus ist langsam. Die sind den Mustersuchern unterlegen und damit überholt; der arithmetische Zweig ist es nicht.
Und `TERSE` ist die Host-Welt: das Format von TRSMAIN und AMATERSE, mit dem auf z/OS seit Jahrzehnten Datasets verschickt werden.
Der Hinweis, den man dazu geben sollte: alle sieben sind **lesend transparent**. Man muss beim Lesen nicht sagen, was es ist – FLAM erkennt es. Das steht in der Konvertierungs-Einheit.
Eine Ausnahme gehört dazu, weil sie sonst einen Supportfall erzeugt:
`read.binary()` dekomprimiert nicht von selbst. Dort ist `DECODE=NONE` die Vorbelegung, und das ist richtig so – wer binär liest, will die Bytes, wie sie sind. Mit `DECODE=DECOMP` (oder `ALWAYS`) packt auch `read.binary()` aus. `read.text()`, `read.record()` und `read.auto()` tun es von sich aus; nachgemessen an einer ZSTD-Datei.



Die Folie, die die Einheit trägt. Nicht vorlesen – die drei Gruppen zeigen und dann die zwei Zahlen nennen, die man behält.
Der Korpus: unser Handbuch als Textdatei, 757 142 Byte. Ein Lauf je Verfahren, jeweils in der Vorbelegung, die CPU-Zeit aus dem Log von `flcl`.
Die drei Gruppen liest man von oben nach unten:
LZ4 ist in einer eigenen Klasse: 1,9 zu 1 – aber in drei Millisekunden. Wer eine langsame Leitung und eine schnelle CPU hat, nimmt etwas anderes; wer eine schnelle Leitung und knappe CPU hat, nimmt das.
GZIP, ZSTD und BZIP2 sind die Mitte: **ZSTD liefert 4,3 zu 1 in 19 Millisekunden**, BZIP2 nur 4,2 zu 1 in 44 – auf **Text** ist ZSTD also besser und doppelt so schnell.
Und hier gehört der Satz dazu, der die Folie vor einem falschen Schluss bewahrt: **das gilt für diesen Korpus.** Text ist der schlechteste Fall für einen arithmetischen Kompressor. Nimmt man statt dessen eine Spalte aus Zahlen – 100 000 Zwei-Byte-Längenfelder –, dreht sich das Bild: bzip2 68 087 Byte, zstd 84 649, gzip 86 303, lz4 153 840. BZIP2 gewinnt dort um rund 20 Prozent. Genau deshalb kann man im Topfdesign je Strom ein eigenes Verfahren wählen.
XZ und LZIP holen 4,7 zu 1, brauchen dafür aber 278 bzw. 290 Millisekunden. Das ist **fünfzehnmal** die CPU-Zeit von ZSTD für zehn Prozent mehr Rate.
Und `TERSE` steht mit Absicht daneben: PACK und SPACK sind schwächer als alles andere und trotzdem richtig, wenn die Gegenseite AMATERSE erwartet.
Nicht in der Tabelle steht **ADC**, das klassische FLAMFILE-Verfahren, und das hat einen Grund: die C-Implementierung ist im CPU-Verbrauch nicht mit der Assembler-Version auf dem Host vergleichbar. Eine Zahl von hier wäre irreführend.
Die Warnung, die dazugehört: **das ist ein Korpus.** Text komprimiert anders als Zahlen, und eine Zahlungsdatei anders als ein Handbuch. Die Reihenfolge der Verfahren bleibt, die Zahlen nicht – wer es genau wissen will, misst den eigenen Bestand. Das kostet einen Nachmittag.



Die Folie, die verhindert, dass jemand blind `BEST` schreibt.
Jedes Verfahren hat Stufen, und die Schreibweisen sind absichtlich gleich:
eine Zahl, `FAST`, `LEV1` bis `LEV9` (bei ZSTD bis `LEV19`, bei LZ4 bis `LEV16`), `BEST` und `AUTO`. `AUTO` ist die Vorbelegung und bei GZIP die Stufe 6. Zwei Zahlen dazu, weil sie überall gleich gemeint sind: **Stufe 0 ist in der Regel das Kopieren** – nicht komprimieren – und **Stufe 1 der schnellste Komprimierer**. `FAST` ist genau das: die niedrigste Stufe, die überhaupt noch komprimiert.
Was man dazu wissen muss: die Stufen kosten **überproportional**. Von der Vorbelegung auf `BEST` zu gehen verdoppelt oder verdreifacht die CPU-Zeit und bringt oft ein bis drei Prozent. Wer mehr Rate braucht, wechselt besser das Verfahren als die Stufe – der Sprung von GZIP zu ZSTD bringt mehr als GZIP auf Stufe 9. Und dann der Schalter, der klug ist: **`SKIP` lässt die Kompression weg, wenn keine Redundanz gefunden wird.** Damit wird ein bereits komprimiertes JPG nicht aufgebläht, ohne dass man es vorher wissen muss. Bei GZIP geht es noch weiter: sind die Daten als redundanzfrei markiert und ist keine Stufe gesetzt, nimmt GZIP von sich aus Stufe 0, also Kopieren.
Und für den Host die Aussage, die alle Stufenüberlegungen dort beendet:
mit zEDC gibt es nur eine Stufe. Die Karte ist darauf ausgelegt, GZIP Stufe 1 zu erreichen, und liegt tatsächlich bei etwa 0,8 davon. Wer auf einem System mit aktiver zEDC eine Stufe setzt, stellt nichts ein – das Komprimat ist dasselbe.
`COPY` ist die ausdrückliche Form davon – kein Komprimieren, nur durchschreiben. Das braucht man, wenn die Kette bleiben soll, aber die Rechenzeit nicht.



Die Folie, die einen Datenaustausch rettet – und die nur bei GZIP etwas zu sagen hat.
Die Formate außer GZIP haben **keinen Kopf für Metadaten**. Das steht so in unseren Handbuchseiten: „there is no header to store meta information about the data, so all metadata is lost". Was verloren geht, ist genau das, was die Gegenseite wissen muss: der Zeichensatz, die CCSID, das Satzformat, der ursprüngliche Name.
GZIP hat einen Kopf, und FLAM kann darin Metainformationen ablegen. Der Schalter dafür ist `CHECK`: er rechnet eine Prüfsumme über den Kopf. Das ist nicht die Vorbelegung, weil **nicht jede GZIP-Implementierung** damit umgehen kann – wer den Schalter setzt, muss wissen, dass auf der Gegenseite FLAM steht.
Der Gewinn, wenn beide Seiten FLAM benutzen: die CCSID reist mit. Damit entfällt die Absprache, die sonst per E-Mail läuft und irgendwann falsch ist.
Dasselbe gilt für **ZIP** und für den **ASCII-Armor**: auch dort schreiben wir die Angaben mit, die ein Host braucht – Satzformat, Satzlänge, Zeichensatz. Bei ZIP ist das ein **privates Extra-Feld** je Member, beim Armor sind es zusätzliche Kopfzeilen. Und hier die Warnung, die aus Kundenfällen kommt: **manche fremden Werkzeuge warnen darüber**, statt sie zu ignorieren, wie es richtig wäre. Deshalb kann man die Erweiterungen abschalten – bei ZIP mit `NOXFLD`, beim Armor mit `FORMAT=STD` oder `OFF`.
Der Preis steht in der Handbuchseite: ohne das Extra-Feld kann FLAM das Originalformat nicht mehr rekonstruieren.
Und der zweite Punkt dieser Folie, der für alle Verfahren gilt:
Kompression verliert die Satzgrenzen. Das Ergebnis ist ein Byteblock.
FLAM nimmt die Satzgrenzen deshalb von sich aus mit: bei Text als Trennzeichen, bei Sätzen als **4-Byte-Längenfeld vor jedem Satz** – so steht es in `man_compress_gzip.txt`, und mit `write.record()` wandern zusätzlich alle Dateiattribute in den GZIP-Kopf. Beim Lesen erkennen wir das automatisch wieder. **Die meisten fremden Werkzeuge erkennen die Längenfelder nicht** und können mit den Daten dann nichts anfangen – das ist der häufigste Anfängerfehler auf dem Weg vom Host nach Unix.



Eine kurze Folie für den Host – die Beschleuniger-Einheit hat die Einzelheiten, hier steht die eine Aussage, die man beim Komprimieren wissen muss.
Steht auf dem Rechner eine zEDC-Karte zur Verfügung, benutzt FLAM sie für GZIP und spart damit CPU-Zeit. Das ist der Normalfall auf einem modernen z-System, und es ist der Grund, warum GZIP auf dem Host oft die richtige Wahl ist, obwohl ZSTD in Software besser abschneidet.
Und jetzt der Satz, den man kennen muss, weil er in einem Audit oder bei einem Partner auftaucht: **die Kompression entspricht dann nicht RFC 1951.** Die komprimierten Daten sind nach RFC 1950 kodiert und das Dateiformat nach RFC 1952 – lesbar also für jeden GZIP-Leser, aber wer die Bitfolge gegen RFC 1951 prüft, findet einen Unterschied. Das steht so in unserer Handbuchseite.
Praktisch heißt das: austauschbar ja, bitidentisch mit einem Software-GZIP nein. Wer Prüfsummen über das **Komprimat** vergleicht, statt über die Daten, sieht einen Unterschied, der keiner ist. Die Prüfsumme gehört über die klaren Daten.
Und der Punkt, der in die Wirtschaftlichkeitsrechnung gehört, weil er oft fehlt: **die Karte komprimiert schlechter als Software.** Sie ist auf GZIP Stufe 1 ausgelegt und erreicht etwa 0,8 davon; die Stufe kann man nicht beeinflussen. Bei statisch strukturierten Daten kann das Komprimat damit **bis zu Faktor 10 größer** sein als mit ZSTD oder ADC. Die gesparten MSU sind also nicht kostenlos – sie werden in Plattenplatz, Bändern und Leitungsbandbreite bezahlt. Die Frage, die ein Haus beantworten muss, lautet deshalb nicht "spart zEDC CPU" (das tut sie), sondern **ab welchem Faktor mehr Speicher sich die gesparten MSU rechnen.**



Die Folie für die Frage „und was ist mit TRSMAIN?" – sie kommt in jeder Host-Schulung.
TERSE ist das Format, mit dem auf z/OS seit Jahrzehnten Datasets transportiert werden: PTFs von IBM, Dumps zum Hersteller, Kundendaten zum Dienstleister. Erzeugt wird es dort von TRSMAIN oder AMATERSE.
Was FLAM damit kann, und das ist die gute Nachricht: **entpacken kann es alle vier Varianten**, die IBM im Lauf der Zeit ausgeliefert hat.
Erzeugen kann es die zwei Verfahren, die der IBM-Entpacker annimmt – `PACK` und `SPACK`. Dazu kommen zwei Kopfformate, `NATIVE` und `COMPAT`.
Was es **nicht** tut, und das ist wichtig: es wandelt keine Zeichensätze.
Wer ein EBCDIC-Dataset getersed bekommt und Text braucht, entpackt zuerst und wandelt danach mit der Zeichensatzstufe. Umgekehrt genauso.
Die Zahlen aus unserer Messung ordnen es ein: SPACK bringt 3,1 zu 1, PACK 2,4 zu 1 – also weniger als GZIP, bei mehr CPU-Zeit. Man nimmt TERSE nicht, weil es gut komprimiert, sondern **weil die Gegenseite es erwartet.**



Die Folie, die Geld spart – und die man in der Praxis am häufigsten braucht.
Erstens: schon komprimierte Daten. Ein JPG, ein PDF, ein ZIP, ein Kryptogramm. Ein zweiter Durchgang macht sie **größer** und kostet CPU.
Das ist kein Grenzfall, das ist die Regel. Der Griff dagegen ist `SKIP` oder `CMPLEVEL=COPY`.
Zweitens: verschlüsselte Daten. Sie sehen wie Zufall aus, also findet der Kompressor nichts. Und daraus folgt die Reihenfolge, die man sich merken muss: **erst komprimieren, dann verschlüsseln.** Umgekehrt ist technisch erlaubt und praktisch sinnlos.
Drittens: wenn die CPU knapper ist als der Platz. Auf einem Host, wo die CPU-Sekunde abgerechnet wird, kann es billiger sein, ein paar Megabyte mehr zu schreiben. Das ist eine Rechnung, keine Meinung – und sie fällt mit einer zEDC-Karte anders aus als ohne.
Der Satz für den Raum: **Kompression ist kein Selbstzweck.** Sie kostet CPU, und bei den falschen Daten bringt sie nichts. Die Topfdesign-Einheit zeigt, wie man das je Spalte entscheidet, statt für die ganze Datei.



Alle vier aus Supportfällen.



Fünfzehn Minuten. Die Übung ist absichtlich eine Entscheidungsübung – es gibt nichts zu tippen. Die Lösungen:
Aufgabe 1. ZSTD. Die Leitung ist der Engpass, die CPU nicht – und ZSTD liefert fast die Rate von XZ bei einem Fünfzehntel der Zeit. GZIP wäre die Antwort, wenn die Gegenseite kein ZSTD kann.
Aufgabe 2. LZ4 oder gar nicht. Wenn die Leitung schneller ist als der Kompressor, kostet jede Rate mehr Zeit, als sie spart.
Aufgabe 3. GZIP – wegen der zEDC-Karte. In Software wäre ZSTD besser, in Hardware gewinnt GZIP, weil die Karte die CPU-Zeit übernimmt.
Aufgabe 4. `SKIP` setzen, oder besser: die Spalte mit dem Belegscan in einen eigenen Topf mit `CMPLEVEL=COPY`. Dann bleiben die anderen Spalten komprimierbar – das ist die Topfdesign-Einheit.

WAS BLEIBT

- die Frage ist **was ist knapp** – Platz, CPU oder Leitung
- **ZSTD** ist meist die richtige Wahl; **GZIP** wenn es austauschbar sein muss; **LZ4** wenn die CPU knapp ist
- `XZ` und `LZIP` holen zehn Prozent mehr für die **fünfzehnfache** CPU-Zeit
- nur **GZIP** hat einen Kopf für Metadaten – sonst ist die CCSID Absprache
- **Kompression verliert Satzgrenzen** – Trennzeichen, Längenfeld oder feste Länge
- **erst komprimieren, dann verschlüsseln** – und schon Komprimiertes nicht noch einmal

FLAMv5.21-42969

Lernziele durchgehen. Der Verweis nach vorn: wie man je Spalte entscheidet, statt für die ganze Datei, steht in *Topfdesign* – und was die Hardware auf dem Host übernimmt, in *Beschleuniger auf z/OS*.

FLAM SCHULUNG, EBENE 2 – DATEN

Was ein Satz ist – und wer es weiß

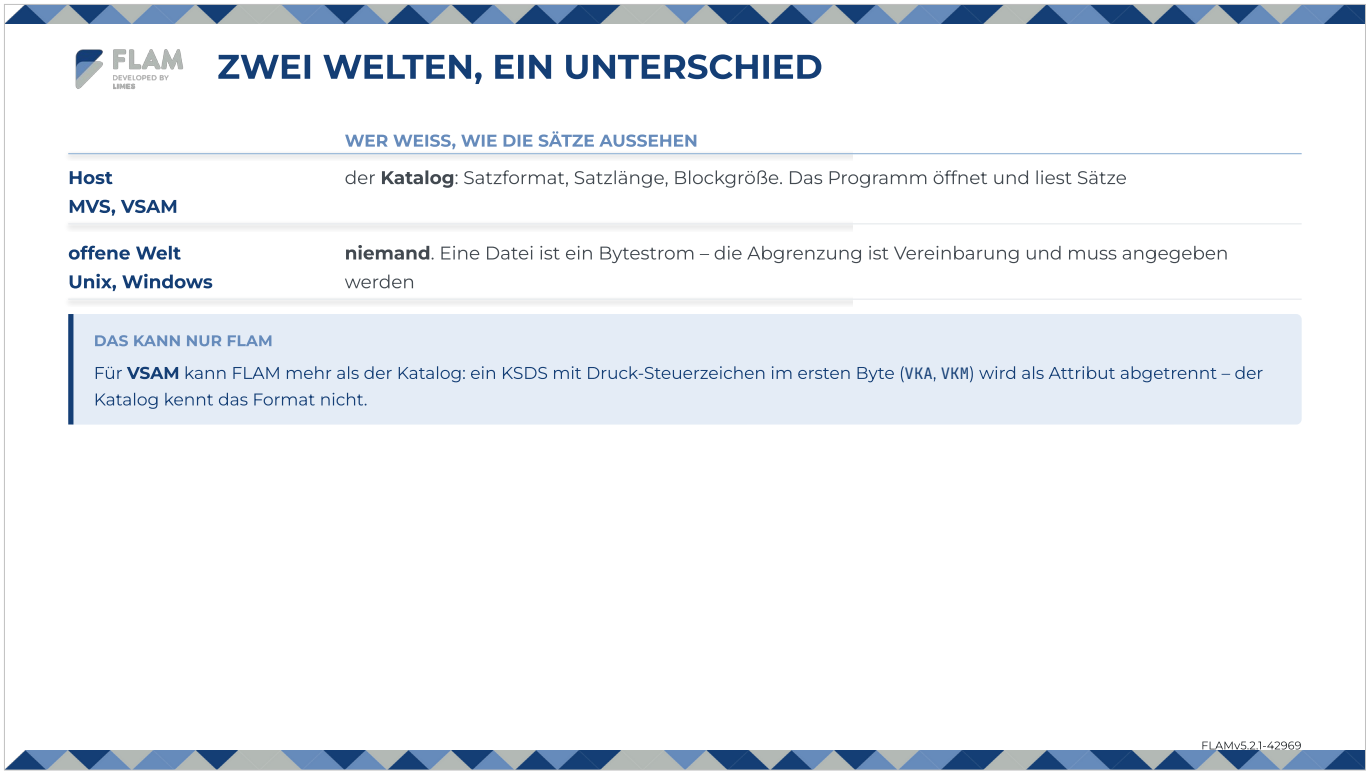
Satzformate, Längenfelder, Trennzeichen – und die automatische Erkennung

Falk Reichbott · limes datentechnik gmbh

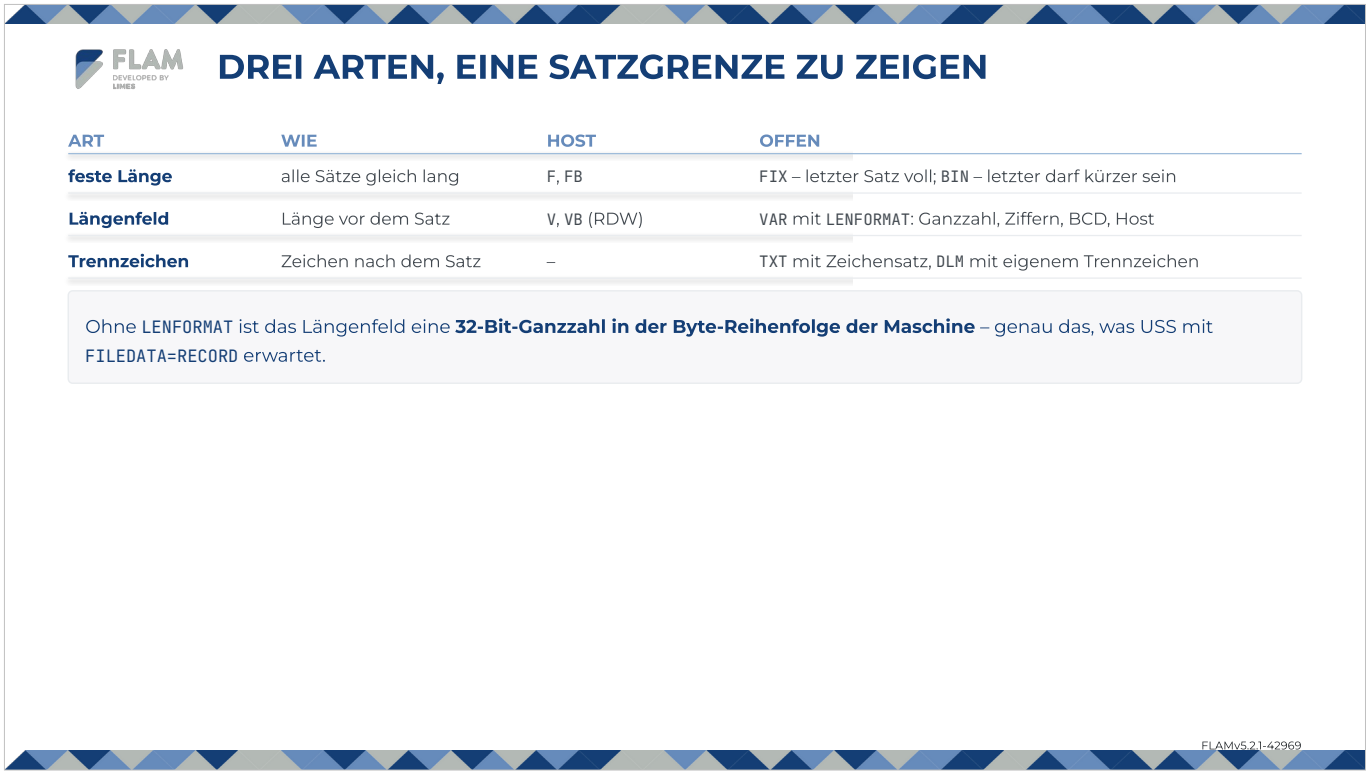
Die Einheit, die den Unterschied zwischen zwei Welten erklärt – und sie ist die, nach der die meisten Supportfälle aufhören.
Der Bogen: warum ein Satz auf dem Host etwas anderes ist als auf Unix, die drei Arten, eine Satzgrenze auszudrücken, die Formatnamen, die Angaben drumherum, die automatische Erkennung, und zuletzt die Stolperfallen.
Der Satz für den Anfang: **auf dem Host weiß das Dateisystem, wie die Sätze aussehen – auf Unix und Windows weiß es das nicht.** Deshalb muss man es dort sagen. Diese Einheit sagt, wie.



Das dritte Ziel ist das, das die Supportfälle erzeugt: die Satzlänge ist die **Netto**-Länge. Wer die Länge aus dem Katalog übernimmt, liegt bei Druckdateien um eins daneben.
Das fünfte ist der Griff, mit dem man sich die halbe Angabe spart.



Die Folie, die alles Weitere erklärt.
Auf dem Host steht im Katalog, wie die Datei aufgebaut ist: Satzformat, Satzlänge, Blockgröße. Ein Programm öffnet die Datei und bekommt Sätze – es muss nichts wissen und nichts angeben.
Auf Unix und Windows gibt es keinen Katalog. Eine Datei ist ein Bytestrom. Wer Sätze darin sehen will, muss sagen, wie sie abgegrenzt sind – sonst gibt es nur Bytes.
Daraus folgt die ganze Einheit: FLAM muss beim Lesen wissen, wie die Sätze aussehen, und beim Schreiben, wie sie aussehen sollen. Auf dem Host holt es sich das aus dem Katalog, auf der offenen Seite aus dem Parameterstring.
Und es gibt einen Fall, in dem FLAM **mehr** kann als der Katalog: für VSAM sind Satzformate mit Druck-Steuerzeichen nicht vorgesehen. Wenn man weiß, dass in einem KSDS das erste Byte ein Steuerzeichen ist, kann man das Format `VKA` oder `VKM` angeben – dann wird das Zeichen als Attribut abgetrennt. Das steht so in unserer Handbuchseite, und es ist ein Beispiel dafür, dass die Formatangabe auch auf dem Host nützlich sein kann.



Die Kernfolie. Mehr Möglichkeiten gibt es nicht, und jede hat ihre Welt.
Feste Länge. Jeder Satz ist gleich lang, das Ende ergibt sich aus dem Zählen. Das ist die Host-Welt (`F`, `FB`) und die portable Form heißt `FIX`. Der letzte Satz ist bei `FIX` ebenfalls voll lang – bei `BIN`, das einen Bytestrom in Sätze schneidet, darf er kürzer sein. Das ist der Unterschied zwischen den beiden, und er ist leicht zu übersehen.
Längenfeld. Vor jedem Satz steht seine Länge. Das ist die Host-Welt der variablen Sätze (`V`, `VB`) und auf der offenen Seite die Angabe `LENFORMAT`. Vier Formen: als Ganzzahl, als Ziffernfolge, als BCD oder im Host-Format – also als RDW, wie MVS es schreibt.
Trennzeichen. Nach jedem Satz steht ein Zeichen oder eine Folge. Das ist die Unix-Welt: `TXT` mit einem Zeichensatz, oder `DLM` für ein eigenes Trennzeichen in Binärdaten.
Der wichtige Satz für die offene Seite: ohne `LENFORMAT` wird ein Längenfeld als **32-Bit-Ganzzahl in der Byte-Reihenfolge der Maschine** geschrieben. Genau das erwartet USS bei einer Allokierung mit `FILEDATA=RECORD` – damit kann eine MVS-Anwendung die Datei lesen, die FLAM auf USS geschrieben hat, ohne eine Angabe. Das ist bewusst so gewählt.

DIE SATZLÄNGE IST DIE NETTO-LÄNGE

GEMESSEN: DIE NETTO-LÄNGE, NICHT DIE DES KATALOGS

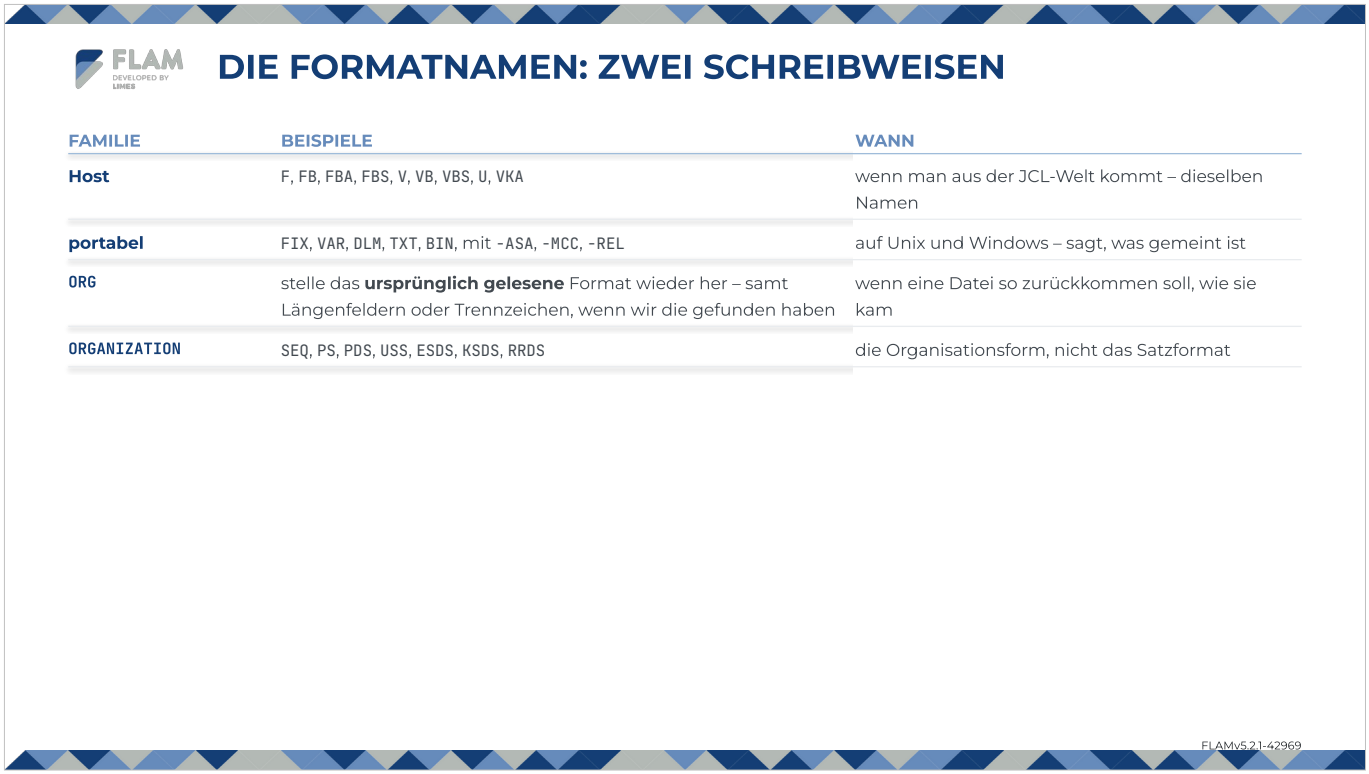
```
# 81 bytes per record on disk: 1 control character + 80 data
read.record(file='<SYSUID>.LISTE.FBA' recformat=FIX-ASA
            reclength=80
            prncontrol=REPLACE ccsid=1141)


# reclength=81 stops with:
#   Record length ... don't match the expected record
#   length (81+1) at read
```

Im Katalog steht die **physische** Länge, in FLAM die **fachliche**. Bei `ASA` und `MCC` also eins abziehen – das Steuerzeichen ist ein Attribut.

FLAMv5.21-42969

Die Folie, die man zeigt, bevor jemand eine Druckdatei anfasst – sie erspart einen Supportfall.
Die Angabe `RECLENGTH` ist bei FLAM **immer die reine Datenlänge, nach allen Abzügen**.
Das steht so im Handbuch, und es ist die Stelle, an der sich die Host-Erfahrung und FLAM unterscheiden.
Das Beispiel aus dem Handbuch: eine FBA-Datei mit LRECL 121 hat bei FLAM die Satzlänge **120**.
Das Druck-Steuerzeichen ist ein Attribut, kein Teil des Satzes.
Und weil es so leicht zu verwechseln ist, hier gemessen: eine Datei mit 81 Byte je Satz – ein Steuerzeichen und 80 Datenbytes – braucht `RECFORMAT=FIX-ASA RECLENGTH=80`. Schreibt man 81, sagt der Leser "expected record length (81+1)" und bricht ab. Die Fehlermeldung nennt also beide Zahlen, was die Suche kurz macht.
Der Merksatz für den Raum: **im Katalog steht die physische Länge, in FLAM die fachliche.**
Wer die Zahl aus einer Listing-Ausgabe übernimmt, muss bei ASA und MCC eins abziehen.



Die Nachschlagefolie. Nicht vorlesen – den Aufbau erklären, dann wissen die Teilnehmer, wie sie jeden Namen lesen.
Es gibt zwei Familien. Die **Host-Schreibweise** ist die aus der JCL-Welt: `F` fest, `V` variabel, `U` undefiniert, dazu `B` für blocked, `S` für spanned, und als Suffix `A` für ASA-Steuerzeichen oder `M` für Maschinen-Steuerzeichen. `FBA` ist also fest, blocked, mit ASA-Steuerzeichen. Wer JCL kennt, kennt diese Namen.
Die **portable Schreibweise** beschreibt dasselbe ohne Host-Begriffe:
`FIX`, `VAR`, `DLM`, `TXT`, `BIN`, jeweils optional mit `-ASA` oder `-MCC`, und `-REL` für die Satznummer bei RRDS.
Dazu kommt `ORG`, und das ist keine Katalogabfrage, sondern eine Zusage:
stelle das Format wieder her, das wir beim Lesen vorgefunden haben. Haben wir dort echte Sätze gelesen und weder Längenfelder noch Trennzeichen gefunden, ist das genau das ursprüngliche Satzformat. Haben wir dagegen Längenfelder oder Trennzeichen gefunden, dann **baut `ORG` sie wieder ein** – auch das ist das Original.
Der Katalog ist eine andere Sache: er ist für **uns** interessant, um das Satzformat einer Datei zu kennen, die auf dem Host auf Platte liegt. Er sagt, was das System über die Datei weiß – nicht, was `ORG` zurückgibt.
Und die Organisationsform ist eine eigene Angabe: `ORGANIZATION` sagt, ob es sequenziell, ein PDS, ein USS-Pfad oder VSAM in einer seiner Formen ist – `ESDS`, `KSDS`, `RRDS`. Auf der offenen Seite braucht man sie nicht.
Der praktische Rat: auf dem Host **nichts** angeben und das System arbeiten lassen – es kennt das Format der Datei. Auf der offenen Seite die portable Schreibweise nehmen, weil sie sagt, was gemeint ist, statt was es früher einmal hieß.



Die zweite Nachschlagefolie – und die vier Angaben darauf deckt man in neunzig Prozent der Fälle ab.
`PRNCONTROL` entscheidet, was mit einem Druck-Steuerzeichen passiert:
`DETACH` behandelt es als Attribut, `RETAIN` lässt es im Satz, `ERASE` löscht es.
`REPLACE` ist mehr als "Leerzeile": es **bereitet den Druck vollständig auf** und führt die Steuerzeichen auf den Daten im Speicher aus. Steht dort eine ASA-`3` (`0xF3`), werden aus den drei Zeilenvorschüben zwei zusätzliche Leerzeilen dahinter. Für den Seitenvorschub füllt FLAM die Seite auf – mit **60 Zeilen je Seite** als Vorbelegung, einstellbar über `RPLFFD` (dort bedeutet `0` wieder 60 und `1`: Seitenvorschübe ignorieren).
Was dabei nicht funktioniert, muss man sagen, sonst fragt es der Teilnehmer: **Überdrucken lässt sich in einer Datei nicht darstellen.** Unterstreichen wird zu einer Zeile aus Bindestrichen, und ein fett gedruckter Text steht einfach zweimal da – dicker wird er davon nicht. Das Ergebnis sieht aus wie vom Typenraddrucker, mit den Grenzen eines Typenraddruckers.
Und die Kleinigkeit, die im Support Zeit kostet: **setzt man `RPLFFD` und `PRNCONTROL` nicht, ist `REPLACE` die Vorbelegung** – nicht mehr `DETACH`.
So steht es im Code, und das ist auch sinnvoll: wer die Zeilen je Seite angibt, will drucken.
`SUPPADDING` entfernt die Auffüllzeichen am Satzende, und `PADCHARACTER` sagt, welches Zeichen das ist. Gemessen: `x'40'` wird als `EBCDIC-BLANK` erkannt – man darf also das Byte schreiben, statt den Namen zu suchen.
`RECMODE` entscheidet, was bei einem **zu langen** Satz passiert: `STOP` ist die Vorbelegung und bricht ab, `CUT` schneidet ab, `WRAP` macht einen zweiten Satz daraus, `SKIP` überspringt ihn.
Die Wahl ist eine fachliche: ein abgeschnittener Betrag ist schlimmer als ein Abbruch.
Eine Grenze dabei, die man kennen muss: sobald ein **Satztrenner** (`RECDELIM`), eine Schlüsselbehandlung oder erhaltene Druckaufbereitung im Spiel ist, muss der Satz zwischengepuffert werden, und der Modus wirkt davor. Dann helfen nur noch `SKIP` und `CUT` – `CUT` kürzt so, dass der Trenner noch in den Satz passt. `WRAP` und `STOP` werden mit einer Meldung abgelehnt statt abzubrechen. Ohne Trenner, Schlüssel und Druckaufbereitung arbeitet jeder Modus für jede Satzlänge.
`RECCOUNT` und `BLKSIZE` sind Mengenangaben für die Verarbeitung, keine Formatangaben – sie bestimmen, wie viel auf einmal gelesen wird.



Die Folie, die eine Frage beantwortet, die vorher keine gute Antwort hatte:
was ist, wenn die Sätze anders aussehen sollen als die Datei, in die sie geschrieben werden?
Seit 5.2.1 sind das **zwei getrennte Angaben**. `RECFORMAT` und `RECLENGTH` am Schreibobjekt selbst sagen, wie die Daten in Sätze geschnitten werden, bevor sie an die Dateischicht gehen – das ist die **logische** Seite.
Dieselben zwei Parameter im `FALLOC()` sagen, wie die Datei oder das Dataset **angelegt** wird – die physische Seite. Beide Längen sind netto, also Daten ohne Druckaufbereitungszeichen und ohne Längenfelder.
Der wichtige Satz für den Kunden: **beide dürfen abweichen, das ist der Zweck.** Eine Textmatrix darf als Textsätze in ein FB-Dataset gehen, und Fixsätze dürfen in ein VB-Dataset.
Und jetzt der Teil, der in der Praxis die Supportfälle auslöst – das **Vorbelegungsverhalten beim Schreiben**: Sagt man gar nichts, versucht FLAM, **das Original wiederherzustellen**. Das ist bei einer dynamischen Allokation richtig und unauffällig. Bei einer **statischen** Allokation über ein DD-Statement wird es zur Falle: Das Dataset gibt den DCB vor, FLAM bekommt keine Vorgabe, wie es die Daten formen soll, und stellt die Form des Originals wieder her. War das Original eine gewöhnliche Textdatei – also ein Bytestrom mit Zeilentrennern –, dann steht genau dieser Bytestrom im FB-Dataset: die Trenner stehen mit in den Daten, jeder Satz ist bis auf den letzten randvoll. **Wer Sätze erwartet hat, findet eine Bytewurst.** Der Ausweg ist genau diese Folie: **das gewünschte Satzformat logisch angeben**, am Schreibobjekt. `recformat=FIX reclength=80` sagt, wie die Daten in Sätze geschnitten werden sollen – unabhängig davon, dass es bei einer statischen Allokation gar nichts zu allokieren gibt. War das Original selbst ein Satz-Dataset, passt es ohnehin, und man muss nichts angeben.
Und der Fall, der die Änderung ausgelöst hat: bei einer statischen Allokation über ein DD-Statement gibt es gar keine Allokation zu beschreiben. Dort gehört die Angabe an das Objekt – das `FALLOC()` läuft ins Leere. Bis 5.2.0 war es der einzige Weg und wirkte ersatzweise mit; das bleibt gültig, ist aber nicht mehr die richtige Formulierung.
Was nicht zusammengeht, wird abgelehnt, und die Meldung nennt den Parameter, der es auflöst – `PRNCONTROL`, `RECMODE` oder `FALLOC(ORGANIZATION=)`. Man muss also nicht raten.

DAS FORMAT KANN IM NAMEN STEHEN

PARSEEXT UND ADDEXT – DER NAME TRÄGT DAS FORMAT

```
# writing: the qualifier is appended, with the NET length
write.record(file='/daten/liste' addext='.dat'
             falloc(recformat=FBA reclength=80))
#   ADDEXT: appended '.FBA80.dat' -- 81 bytes per record on disk


# reading the same name back, no format parameters needed
read.record(file='/daten/liste.FBA80.dat' parseext ccsid=1141)
#   AUTODET: record length 80 applied from filename qualifier
```

Die Ziffern sind die Netto-Länge. Auf z/OS gewinnt der Katalog; dort wird der Name nicht ausgewertet. Und der Schalter ist nicht die Vorbelegung.

FLAMv5.21-42969

Die Folie mit dem Griff, den fast niemand kennt und der eine Kette von Jobs vereinfacht.
Auf der offenen Seite kann das Satzformat **im Dateinamen** stehen, als Abschnitt zwischen Punkten oder Unterstrichen. Mit dem Schalter `PARSEEXT` liest FLAM ihn und wendet Format und Länge an, ohne dass man sie im Aufruf wiederholt. Gemessen an einer Datei `plain.FB80.bin`: "AUTODET: record length 80 applied from filename qualifier".
Die Ziffern sind die **Netto-Länge**, nicht die des Katalogs. `.FB80.` und `.FBA80.` bedeuten beide 80 Byte Nutzdaten und unterscheiden sich nur im Format – bei `.FBA80.` liegen 81 Byte je Satz auf der Platte, weil das Steuerzeichen dazukommt. Wer das Steuerzeichen in die Daten nimmt und 81 Byte schreibt, bekommt `.FB81.`.
Wozu das gut ist: wenn eine Datei vom Host auf ein Unix-System wandert und dort von mehreren Jobs weiterverarbeitet wird, trägt sie ihr Format selbst mit. Kein Job muss wissen, was in ihr steht – und wenn sich das Format ändert, ändert sich der Name, nicht die JCL.
Die Gegenrichtung gibt es auch: beim Schreiben kann FLAM den Qualifier an den Namen **anhängen**, damit die nächste Stufe ihn lesen kann.
Zwei Grenzen, die man dazu sagen muss: auf z/OS gewinnt immer der Katalog – der Name wird dort nicht ausgewertet. Und der Schalter ist **nicht** die Vorbelegung; ohne `PARSEEXT` bleibt der Name ein Name.



Diese Folie hat zwei Hälften, und die zu trennen ist der halbe Verstehens- gewinn: **oben die Sätze, unten die Spalten.** Es sind zwei verschiedene Erkennungen, und nur die untere heißt ROWDET.
Oben: das physische Satzformat. Beim Lesen sucht FLAM zuerst in den Daten – gibt es **Längenfelder** davor oder **Trennzeichen** dahinter?
Findet es keine, nimmt es die **Satzlängen vom System**, also das, was der Katalog oder die Dateiorganisation hergibt. Und hat es auch die nicht, bleibt die **feste Satzlänge**, die man angeben kann. Damit liest FLAM jedes der vier physischen Satzformate transparent ein, ohne dass man es ansagen muss – und danach hat man Sätze.
Unten: die Spalten. Erst jetzt kommt **ROWDET** ins Spiel: `READ.AUTO()` zerlegt die Sätze anhand der Auffüllgrenzen und der Datenarten in Spalten und macht daraus eine **Tabelle**. Das ist die Erkennung, die neu ist und die das Verhalten ändert – sie steht deshalb als ausdrücklicher Hinweis in den Release Notes.
Was das bringt: eine Datei, deren Aufbau niemand mehr kennt, wird lesbar, ohne dass man ein Copybook hat. Das ist der Fall, der in jeder Modernisierung vorkommt.
Was man wissen muss, weil es Bestehendes ändert: wer die alte Behandlung will, schaltet sie mit `DETECT(DISABLE=ALL)` oder `NOROWDET` ab. Und bei `WRITE.TEXT(DUMP)` erzeugt eine erkannte Tabelle **CSV** statt eines Hex-Dumps – wer den Dump will, setzt `NOROWDET` innerhalb von `READ.AUTO()`.
Der Rat für die Praxis: die Erkennung ist ein Werkzeug für die Analyse und für den Einzelfall. In einem Produktionsjob, der jede Nacht dasselbe Format liest, gibt man das Format an – dann ist das Ergebnis nicht davon abhängig, was in der Datei von heute erkennbar ist.



Zwei Randfälle, die je einen Supportfall im Jahr erzeugen – und beide stehen im Handbuch, aber in keiner Schulung.

Der leere Satz. Auf z/OS schreibt das Language Environment einen leeren Satz variabler Länge **gar nicht**. Damit er nicht verschwindet, schreibt FLAM ein Füllzeichen: `0x00` bei binären Daten, das Leerzeichen `0x40` bei Text. Beim Lesen wird ein Satz der Länge 1 mit genau diesem Zeichen wieder zum leeren Satz – vorausgesetzt, `_EDC_ZERO_RECLEN` steht auf `Y`, was die Vorbelegung ist. Wer diese Variable im Haus abschaltet, bekommt aus einem leeren Satz ein Blank, und niemand weiß mehr, warum.

Der Satz zum Mitnehmen: **ein leerer Satz überlebt den Rundlauf, aber nur, weil FLAM ein Zeichen unterschiebt. Der große Satz.** FLAM5 verarbeitet **spanned records** – die Grenze von 1 MiB ist keine technische, sondern eine Speicherentscheidung: wir belegen `RECCOUNT` mal Satzlänge im Voraus, und mehr als 1 MiB je Satz würde das unvertretbar groß machen. **Auf Anfrage lässt sich die Länge erweitern**, so steht es auch im Handbuch.

Die Beschränkung liegt bei **FLAM4**, nicht bei FLAM5. Wer einen großen Satz in eine alte FLAMFILE schreiben will, wandelt die Matrix in einen Block mit Längenfeldern – FLAM5 umgeht die FLAM4-Grenze also durch **Partitionierung**, und damit passen lange Sätze auch in ein altes FLAMFILE. Die FLAM4-Komponenten sehen dabei einen undefinierten Bytestrom.

Beim Lesen muss dieselbe Methode angesagt werden, sonst kommen die Sätze nicht zurück. Im `CONV` gibt es dafür den Schalter **`LARGER`** an `READ.FLAM4()` und `WRITE.FLAM4()`; er benutzt die Methode `B4X`. Automatisch geht das nicht, und das ist eine bewusste Entscheidung.

Zwei Grenzen dazu, die man kennen sollte: die **Satzlängenerkennung arbeitet nicht mehr ab 32768 Byte** – dort muss das Format angesagt werden. Und die FLAM5-I/O liest große V(B)S-Sätze von sich aus; mit `NOSPAN` schaltet man das ab.



Alle vier aus Supportfällen, und die erste ist die häufigste überhaupt.



Zwanzig Minuten, Papier genügt. Die Aufgaben sind echte Fälle.
Die Lösungen:
Aufgabe 1. `recformat=FIX-ASA reclength=132` – die LRECL ist 133, das Steuerzeichen geht ab. Dazu `prncontrol=REPLACE`, wenn daraus Text werden soll.
Aufgabe 2. `recformat=VAR lenformat.host()` – das RDW, wie MVS es schreibt. Ohne Angabe wäre es eine 32-Bit-Ganzzahl in der Byte-Reihenfolge der Maschine, was auf einer Intel-Maschine die falsche Reihenfolge ist.
Aufgabe 3. `recformat=TXT` mit der CCSID der Datei, oder `DLM` mit einem eigenen Trennzeichen, wenn es kein Text ist.
Aufgabe 4. `STOP`. Ein abgeschnittener oder übersprungener Satz ist ein stiller Verlust, und in einer Zahlungsdatei ist das nicht verhandelbar.
Wer die langen Sätze sehen will, fährt einen Lauf mit `SKIP` und schaut sich das Log an – aber nicht als Dauerzustand.



Lernziele durchgehen. Der Verweis nach vorn: wie aus Sätzen **Spalten** werden – Typen, Copybooks, CSV – steht in *Tabellen und CSV*. Diese Einheit endete am Satz.

FLAM SCHULUNG, EBENE 2

Aus einem Satz werden Spalten

Formate erkennen, Spalten umbauen, typgerecht ausgeben

Falk Reichbott · limes datentechnik gmbh

Viertes Funktionsthema und das mächtigste. Bis hierher war eine Datei ein Strom von Bytes oder Sätzen – jetzt wird sie eine Tabelle mit typisierten Spalten.
Der Ursprung ist der Zahlungsverkehr: DTA, SWIFT und SEPA einheitlich in COBOL oder PL/I als Sätze und in Java oder C++ als Strom verarbeiten. Aber es hilft bei fast jedem kommerziellen Datenformat.



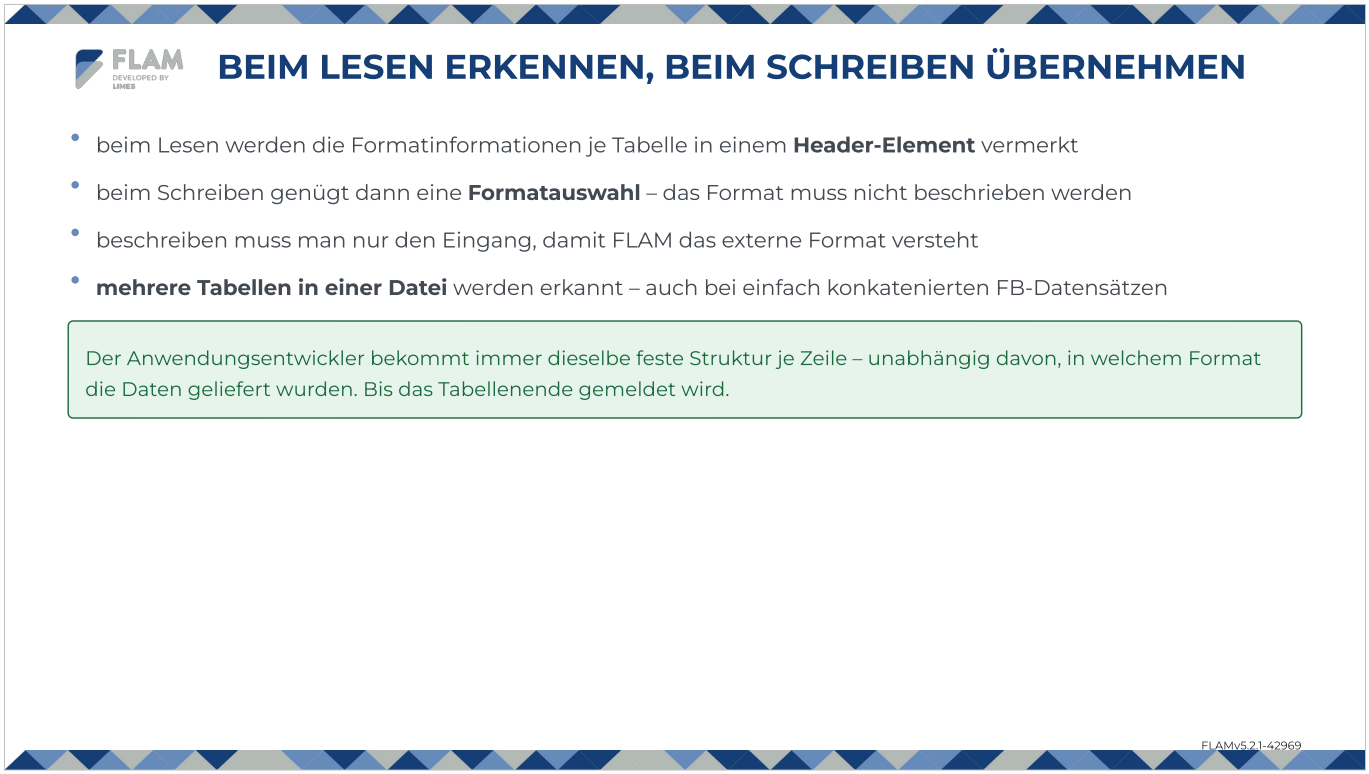
Ziel drei ist das, was den Aufwand halbiert: beim Schreiben muss man das Format meist nicht beschreiben.



Fünf Stufen, und sie sind symmetrisch: beim Lesen von links nach rechts, beim Schreiben rückwärts.
Der Kern ist die Mitte: das neutralisierte Element mit einem **Typ**. Erst der Typ macht aus Bytes eine Zahl oder eine Zeichenkette – und erst dadurch kann FLAM zwischen BCD, Binär und Text umrechnen.
Die Pre- und Post-Prozessoren sind der Platz für Spaltenverschlüsselung und Maskierung. Sie liegen bewusst zwischen Zerlegung und Typisierung.
Zum Stand der Typen, weil hier zuletzt etwas dazugekommen ist: neben `BINARY`, `STRING`, `INTEGER` und `FLOAT` gibt es **seit 5.2.1 `DATE`** für Datum und Zeitstempel, und **`INT64`**, der primär intern für die FLAM5-Archive genutzt wird.
Und ein Typ, der eigentlich keiner ist: **`TYPE.DETECT()`**. Damit erkennen wir selbst, welche String-Repräsentation vorliegt – Zahl, String, Datum oder HexBinary. Über `LOCALE` sagt man, in welcher Schreibweise zu erwarten ist: `DE` und `US` unterscheiden sich beim Dezimaltrenner und beim Datumsformat, und das ist genau die Stelle, an der eine automatische Erkennung sonst falsch abbiegt.



Die Liste einmal durchgehen, aber ohne Details. Wichtig ist die Zweiteilung:
flache Formate und hierarchische.
Bei den flachen kommt die Struktur aus Satzlaenge oder Trennzeichen, bei den hierarchischen aus einem Pfad. Der Pfad ist der Grund, warum XML und SWIFT im selben Mechanismus liegen wie eine CSV-Datei.
Headline, Header und Trailer sind die Kennzeichen, an denen ein Tabellenwechsel in Unload- und Backup-Dateien erkennbar ist – aber auch ohne sie wird der Wechsel erkannt.
Ein Format fehlt in der Tabelle, weil es keine Datei beschreibt, sondern den Speicher: **die Zeigerformate.** Wer in den Speicher schreibt, kann die Spalten als **Array aus Zeiger und Länge** bekommen – oder als Zeiger, Länge und Flag, wenn man eine Null-Indikation braucht.
Das gibt es in zwei Varianten: **für jede Spalte** oder **nur für die belegten**. Bei der zweiten kommt ein weiterer Zeiger dazu, der auf den nullterminierten Spaltennamen zeigt – sonst wüsste der Aufrufer nicht, welcher Wert zu welcher Spalte gehört.
Wichtig für die Erwartung: **das geht nur auf der Format-Seite**, also im Speicher, nicht auf der File-Seite. Eine Datei aus Zeigern gibt es naturgemäß nicht.
Für Entwickler ist das der bequemste Weg, eine Struktur zu füllen, ohne Offsets zu rechnen – und genau der Punkt, an dem die Satzschnittstelle gegenüber einer selbstgeschriebenen Zerlegung gewinnt.



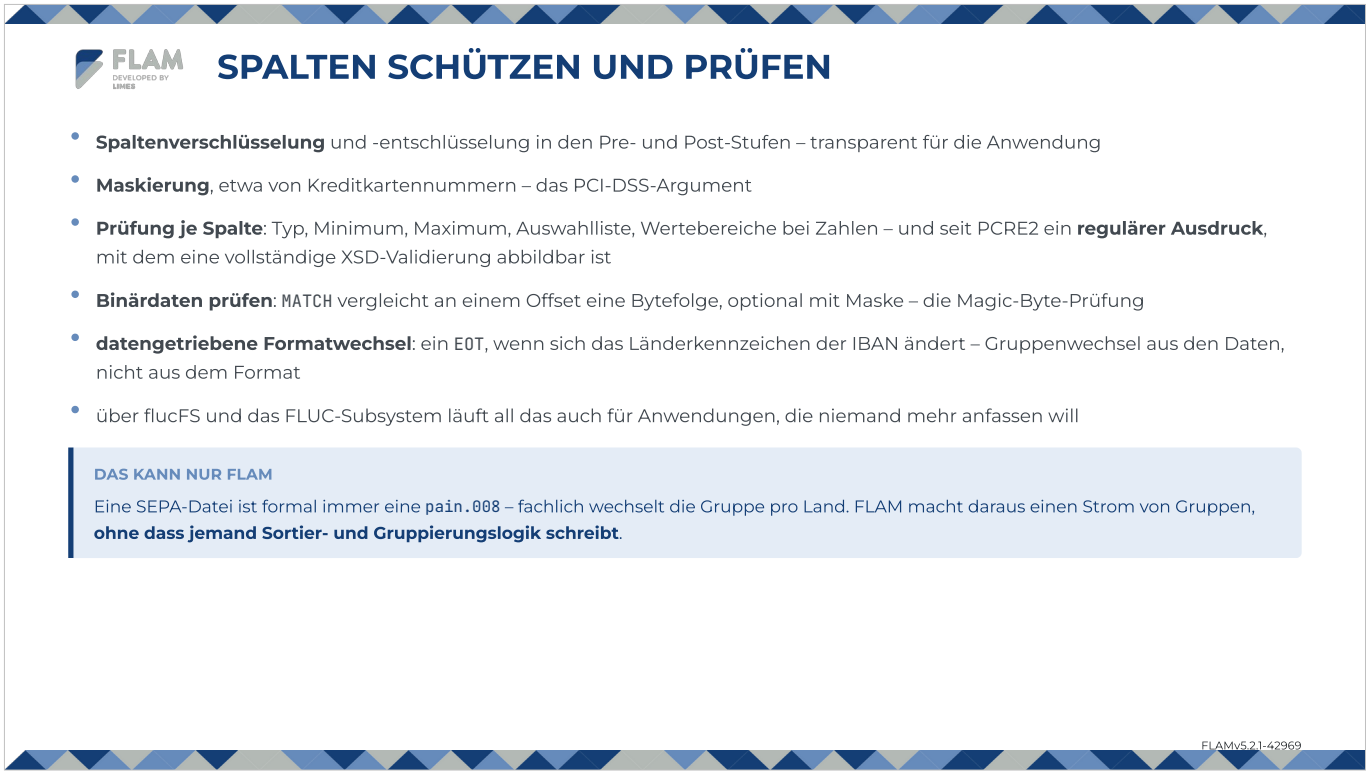
Das ist die Folie, die den Aufwand halbiert, und sie wird gerne übersehen.
Beim Lesen sammelt FLUC die Formatinformationen je Tabelle in einem Header-Element. Beim Schreiben genügt dann eine Formatauswahl – das Format muss nicht noch einmal beschrieben werden.
Beschreiben muss man nur den **Eingang**, damit FLAM das externe Format versteht. Und selbst das nur, wenn Spalten gelöscht oder umsortiert werden sollen.
Der praktische Effekt: verschiedene Plattformen liefern verschiedene Formate, und die Anwendung bekommt immer dieselbe feste Struktur je Zeile.

SPALTEN UMBAUEN

- Spalten **löschen, hinzufügen und umsortieren** – über ihren Namen
- je Spalte ein **Alignment**, damit keine Dummy-Spalten für die Luft in Strukturen nötig sind
- damit lassen sich C-Strukturen und COBOL-Copybooks mit **einem Aufruf pro Zeile** befüllen
- Tabellen und Spalten brauchen **eindeutige Namen** – sie sind der Bezug für alles Weitere
- eine Datei kann beim Schreiben nach Tabellen **auf mehrere Ziele aufgeteilt** werden

FLAMv5.21-42969

Die vier Operationen und das Alignment. Das Alignment ist der Punkt, den Entwickler schätzen: ein Compiler legt jedes Strukturelement auf seine **natürliche Grenze** – ein 4-Byte-Integer auf eine durch 4 teilbare Adresse, ein 8-Byte-Wert auf eine durch 8 teilbare. Zwischen den Feldern entstehen dadurch Füllbytes, die im Copybook nicht stehen. Ohne Alignment-Angabe müsste man für diese Luft Dummy-Spalten anlegen.
Wie groß die Grenze ist, hängt vom Compiler und von der Plattform ab – und genau deshalb ist es eine Angabe und keine feste Regel.
Damit lassen sich C-Strukturen und COBOL-Copybooks auf verschiedenen Plattformen und gegen verschiedene Compiler mit **einem Aufruf pro Zeile** befüllen. Das ist das Argument für die Satzschnittstelle.
Namen sind Pflicht – ohne eindeutige Tabellen- und Spaltennamen kann man nichts umsortieren und nichts abfragen.



Der Teil, der in Compliance-Gesprächen zählt, weil er unter der Anwendung liegt: Spaltenverschlüsselung und Maskierung passieren in den Pre- und Post-Stufen, also ohne dass die Anwendung etwas davon weiß.
Das Beispiel, das immer zieht: Kreditkartennummern maskieren. Wer PCI DSS im Haus hat, versteht sofort, warum das unter der Anwendung passieren muss.
Die Prüfung je Spalte kam mit RegEx dazu – Typ, Minima, Maxima und Auswahllisten gab es schon, das Muster ist neu. Damit ist eine XSD-Musterprüfung ohne separaten XML-Validator möglich; mit regulären Ausdrücken lässt sich eine **vollständige XSD-Validierung** abbilden.
Drei Dinge, die dabei oft übersehen werden:
Bei **Zahlen** kann man Wertebereiche einschränken – nicht nur Länge, sondern der zulässige Bereich.
Bei **Binärdaten** kann man validieren: `MATCH` prüft an einem Offset eine Bytefolge, optional mit Maske. Das ist die Magic-Byte-Prüfung – damit erkennt man, ob in einer Spalte wirklich ein JPEG steht und nicht etwas anderes mit passender Länge.
Und der Kniff, der in der Praxis am meisten Zeit spart: **man kann anhand der Daten künstliche Formatwechsel erzwingen.** Also einen `EOT` bekommen, wenn sich das Länderkennzeichen der IBAN ändert. Eine SEPA-Datei ist formal immer eine `pain.008`, aber fachlich findet ein Gruppenwechsel pro Land statt – und der steht nirgends im Format, der steht in den Daten. Damit wird aus einer Datei ein Strom von Gruppen, ohne dass jemand eine Sortier- und Gruppierungslogik schreibt.

FLAM
DEVELOPED BY LIMES

DIE KOMMANDOS FÜR CSV

KOMMANDO	AUSGANGSLAGE	ERGEBNIS
BCSV	binäre Datei **mit** Beschreibung: COBOL-Copybook, PL/I, HLASM	deterministisch, produktionstauglich
ACSV	binäre Datei **ohne** Beschreibung	automatisch erkannt, Schema optional speicherbar
FCSV	vorhandene CSV-Datei	normalisiert: Trennzeichen, Kodierung, Struktur

Wer eine Beschreibung hat, nimmt BCSV – das Ergebnis ist reproduzierbar. ACSV ist für Erkundung und für Fälle, in denen es keine Beschreibung gibt.

FLAMv5.21-42969

Drei Kommandos für drei Ausgangslagen. Die Unterscheidung ist wichtig, weil Kunden sonst das falsche wählen.
BCSV braucht eine Beschreibung – ein Copybook, eine PL/I-DECLARE, ein HLASM-DSECT. ACSV braucht keine und erkennt selbst. FCSV normalisiert vorhandene CSV-Dateien.
Wer eine Beschreibung hat, soll BCSV nehmen: das Ergebnis ist deterministisch.
ACSV ist für Erkundung und für Fälle ohne Beschreibung.



Zwei Punkte, die auf keiner Folie standen und beide in Gesprächen regelmäßig gebraucht werden. Erstens: die drei Kommandos gibt es **auch als eigenständige Programme** – `flbcsv`, `flacsv` und `flfcsv`, jedes mit eigenem Handbuch. Wer nur diese eine Aufgabe hat, braucht nicht das ganze FLAM zu lizenzieren. Für ein Gespräch mit einer Fachabteilung, die genau ein Problem hat, ist das oft der Einstieg – und der Umstieg auf das volle FLAM kostet danach keine Änderung an den Aufrufen, weil es dieselbe Sprache ist.
Zweitens `UNLOAD`, und das ist der Punkt für ein Rechenzentrum mit IMS oder DB2. Ein Datenbank-Unload ist keine flache FB-Datei: jeder logische Satz hat einen Präfixblock, und am Ende steht ein Sentinel-Satz. Mit `UNLOAD=` wird jedes FORMAT-Schema in genau diese Hülle gepackt – ein Präfixblock vor jede Datenzeile und eine Auffangzeile am Ende für den Sentinel. Ohne das müsste jemand die Präfixe selbst wegrechnen.
Die Formate benennen, wenn jemand aus dem Datenbankumfeld im Raum ist: DFSURGU0 in seinen Varianten LONG, SHORT und HD-Unload, jeweils auch in den Spielarten mit und ohne Längenfeld, dazu die DB2-Unload-Formate. Sonst nur sagen, dass es sie gibt.



Die konkreten Beispiele, nach denen im Anschluss immer gefragt wird – beide klein gehalten, damit man den Aufbau sieht und nicht den Umfang. Beide sind so nachgefahren, das ist kein Pseudocode.
Links: ein Record in Spalten zerlegen. Das ist die Copybook-Frage. Das `ROW`-Objekt beschreibt die Zeile, jede `COLUMN` eine Spalte mit Namen, Typ und Länge. Gepacktes BCD, eine Zeichenkette mit `WHITESPACE=COLLAPSE` gegen die Auffüllzeichen, eine Fließkommazahl – das sind die drei Fälle, die in einem Hostdataset zu 90 Prozent vorkommen.
Wichtig ist die Trennung: das `ROW` steht in einer **eigenen Datei** und wird mit `row='...'` geholt. Damit benutzen mehrere Jobs dieselbe Beschreibung, und die Beschreibung ändert sich, ohne dass ein Job angefasst wird.
Rechts: XML über Pfadangaben lesen. Dasselbe `ROW`-Objekt, aber die Spalten tragen jetzt ein `PATH`. Damit wird aus einem XML-Dokument eine Tabelle – ohne XSLT, ohne Parser im eigenen Code.
Und zwei Fallen dabei, die ich beim Nachfahren selbst gefunden habe, also bitte weitergeben:
Erstens: **der Name des `ROW` ist Teil des Pfades.** Wer zusätzlich ein `ROOT` angibt, bekommt beides hintereinander – aus `NAME='auftrag'` und `ROOT='/auftrag'` wird `auftrag/auftrag`, und dann findet er nichts. Also entweder das eine oder das andere.
Zweitens: bei XML muss das `ROW` ein **`MAXLEN`** tragen, das für das ganze Element reicht. Der Default ist zu klein, und die Meldung sagt es genau – "row is longer than the provided maximum length".

DASSELBE ÜBER DIE DREI WEGE

WEG	WAS SIE DAVON HABEN
Batch-Utility	`flcl BCSV`, `ACSV`, `FCSV` oder `CONV` mit Tabellenangabe – geplante Umwandlung ganzer Dateien
Subsystem, Filesystem	Spaltenverschlüsselung und Maskierung für Anwendungen, die nichts davon merken dürfen
API	eine feste Struktur je Zeile, ein Aufruf – unabhängig davon, in welchem Format geliefert wurde

FLAMv5.21-42969

Bei den Tabellen ist der API-Weg besonders attraktiv, weil die Anwendung dieselbe feste Struktur je Zeile bekommt, unabhängig vom externen Format.
Und der Subsystem-Weg ist der, mit dem man Standardsoftware absichert, die man nicht anfassen darf.



Das Batch-Beispiel, nach dem am häufigsten gefragt wird: ein Unload mit **vielen Tabellen** soll ein ZIP werden, mit **einer CSV je Tabelle**, in UTF-8 und mit Unix-Zeilenenden – also fertig für die Linux-Seite.
Der Schlüssel ist ein einziges Token im Ausgabenamen: `**[table]**`. Es wird zum Namen der aktuellen Tabelle ausgewertet, sobald die End-of-Table-Behandlung aktiv ist (`ENDOFT`). Damit entsteht je Tabelle automatisch ein eigenes Member – ohne Schleife, ohne Vorabanalyse, ohne zu wissen, wie viele Tabellen drin sind.
Wenn dieselbe Row-Spezifikation mehrfach in dasselbe ZIP schreibt, gibt es zusätzlich `**[midN]**`: einen archivlokalen Zähler, damit die Membernamen eindeutig bleiben und das ZIP **ISO/IEC 21320-1**-konform ist. Das ist der Standard, den Behörden und Banken für Langzeitablage verlangen.
Und jetzt die Gegenüberstellung, weil sie die Werkzeugwahl erklärt:
`**flcl CONV**` nimmt eine **CLP-Beschreibung** der Zeile – das `ROW`-Objekt von der vorigen Folie. Das ist der Weg, wenn man die Struktur selbst beschreibt oder sie aus mehreren Quellen zusammensetzt.
`**flbcsv**` nimmt **das Copybook selbst** und übersetzt es. Dazu kennt es die Unload-Formate direkt: `UNLOAD=IMSL`, `IMSS`, `IHDU` für IMS, `DB2U` und `DB2S` für DB2. Wer einen IMS-Unload und ein COBOL-Copybook hat, braucht genau eine Zeile und muss nichts beschreiben.
Die Regel für den Raum: **hat man ein Copybook, nimmt man `flbcsv`.**
Beschreibt man die Struktur selbst, nimmt man `CONV`. **Beide schreiben dasselbe ZIP.**
Der `flbcsv`-Aufruf ist bewusst so kurz – er zeigt, wie wenig nötig ist.
Ein `ZIP()`-Objekt braucht es nicht: die Vorbelegungen passen hier, und dass ein ZIP entstehen soll, sagt schon der Ausgabename. Und eine `CCSID` gehört hier **nicht** hin, aus einem Grund, den man kennen sollte: `CCSID` bei `flbcsv` ist der Zeichensatz der **Eingabe**, also der Binärdaten aus dem Unload – bei einem Host-Unload wäre `UTF-8` dort schlicht falsch. Die Ausgabeseite braucht die Angabe nicht, weil ein ISO-konformes CSV-Member in einem ISO-konformen ZIP ohnehin in UTF-8 geschrieben wird.

STOLPERFALLEN

! **Fehlende Namen machen alles Weitere unmöglich.**

Tabellen und Spalten brauchen eindeutige Namen – sonst gibt es keinen Bezug für Umsortieren oder Abfragen.

! **Struktur ohne Alignment ergibt Versatz.**

Compiler legen jedes Element auf seine natürliche Grenze (1, 2, 4 oder 8 Byte). Ohne Alignment-Angabe verschiebt sich alles ab dem ersten Füllbyte.

! **Die Autoerkennung wird beim Schreiben erwartet.**

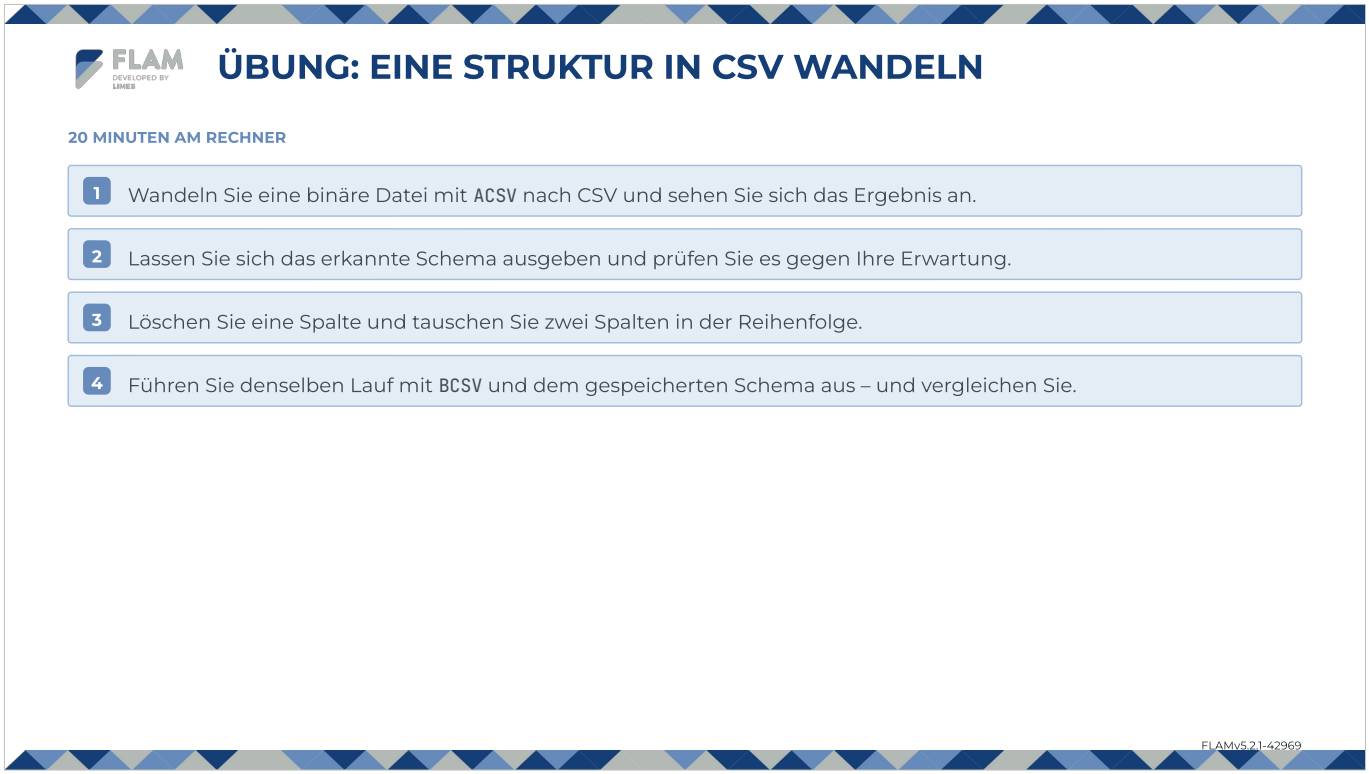
Erkannt wird beim **Lesen**. Beim Schreiben wird das gesammelte Format übernommen – oder man gibt es vor und schaltet die Erkennung ab.

! **Bei mehreren Tabellen fehlt der Tabellenname im Dateinamen.**

Sonst schreiben alle Tabellen in dieselbe Datei. Der logische Name kann in den Dateinamen einbezogen werden.

FLAMv5.21-42969

Alle vier aus der Praxis mit der Tabellenunterstützung.



Zwanzig Minuten am Rechner. Schritt vier ist der Lerneffekt: das Schema speichern und mit BCSV reproduzierbar machen.
Auflösung:
1. `flcl ACSV` erkennt selbst und braucht keine Beschreibung. Das Ergebnis ist gut genug, um zu sehen, was in der Datei steckt – aber es ist Erkundung, nicht Produktion.
2. Das erkannte Schema lässt sich über `ROWOUT` herausschreiben. Damit hat man eine Beschreibung, die man prüfen und versionieren kann.
3. `flcl BCSV` mit diesem Schema über `FORMAT` liefert dasselbe Ergebnis – aber reproduzierbar, weil nichts mehr geraten wird.
Die erwartete Erkenntnis, und sie ist die Botschaft der Einheit: `ACSV` zum Verstehen, `BCSV` zum Produzieren. Wer `ACSV` in die Produktion stellt, hat eine Erkennung im kritischen Pfad – und die ändert ihr Urteil, wenn sich die Daten ändern.

WAS BLEIBT

- fünf Stufen je Spalte: zerlegen, vorverarbeiten, typisieren, nachverarbeiten, zusammenfügen
- flache Formate aus Länge oder Trennzeichen, hierarchische aus einem Pfad
- erkannt wird beim Lesen, übernommen beim Schreiben – das halbiert den Aufwand
- Spaltenverschlüsselung, Maskierung und Musterprüfung liegen unter der Anwendung
- BCSV mit Beschreibung, ACSV ohne, FCSV für vorhandene CSV-Dateien

FLAMv5.21-42969

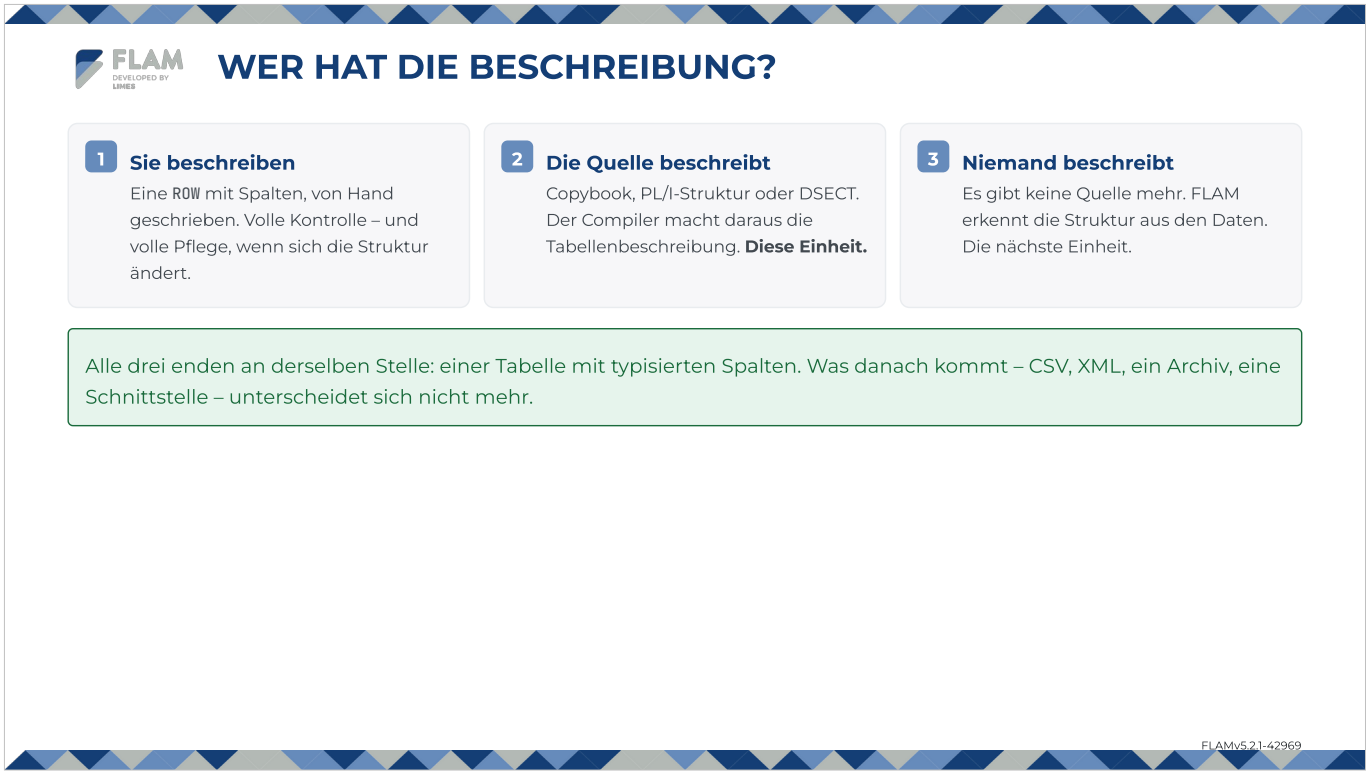
Lernziele durchgehen. Damit ist der Konvertierungsblock abgeschlossen; es folgen die Hilfsfunktionen.



Diese Einheit und die nächste gehören zusammen und beantworten dieselbe Frage von zwei Seiten: wer hat die Beschreibung der Daten?
Hier ist die Antwort: Sie haben sie, in Ihren eigenen Quellen. Ein Copybook liegt in jedem Hostbetrieb, eine DSECT auch – nur hat sie bisher niemand als Datenbeschreibung benutzt, sondern als Übersetzungseinheit.
Genau das dreht diese Einheit um.



Ziel zwei ist das, was die Teilnehmer am meisten überrascht: der Compiler bringt Bytes in die Beschreibung, die im Copybook nicht stehen.



Die Einstiegsfrage, und sie hat genau drei Antworten. Die mittlere ist diese Einheit, die rechte ist die nächste.
Die linke – jemand schreibt die Beschreibung von Hand – ist nicht falsch, aber sie ist Arbeit, und sie veraltet. Wenn eine Spalte im Copybook länger wird, ist die handgeschriebene Beschreibung still falsch.
Deshalb ist die Reihenfolge auf der Folie auch eine Empfehlung: nehmen Sie die Quelle, wenn es eine gibt. Erkennen lassen, wenn es keine gibt. Von Hand nur dort, wo beides nicht geht.

DREI SPRACHEN, EIN ERGEBNIS

SPRACHE	QUELLE	WORAN DER ÜBERSETZER SIE ERKENNT
COBOL	Copybook, fest oder frei formatiert	an den Stufennummern und PIC-Klauseln
PL/I	DCL/DECLARE, Ebene 1 bis 15	an der Deklaration selbst
Assembler	HLASM-DSECT mit benannten DS-Feldern	an DSECT und den DS-Feldern

- **der Dateiname spielt keine Rolle** – die Übersetzer werden durchprobiert, bis einer die Datei annimmt
- auf dem Host sind die Beschreibungen Member einer PDS-Bibliothek und haben gar keine Endung
- **C-Strukturen sind vorgesehen, aber noch nicht enthalten** – das ist heute die ehrliche Antwort

EINE DSECT ALS DATENBESCHREIBUNG – MEHR STEHT NICHT IM AUFRUF

```
flcl BCSV "input='KUNDEN.DAT' format='CUSTREC.asm' \
           reclen=37 output='kunden.zip'"
```

FLAMv5.21-42969

Die Tabelle ist die Faktenfolie, und der wichtigste Satz steht unter ihr:
der Dateiname spielt keine Rolle. FLAM probiert die Übersetzer einfach der Reihe nach durch und fragt jeden:
erkennst du diese Datei? Der erste, der ja sagt, liefert die Spalten.
Das ist keine Bequemlichkeit, sondern die Rücksicht auf den Host. Dort liegen Copybooks und Makros als Member in einer PDS-Bibliothek, und ein Member hat keine Endung. Eine Erkennung, die an einer Endung hinge, wäre auf dem Host unbrauchbar.
Die Reihenfolge ist COBOL, PL/I, Assembler – COBOL zuerst, weil es der häufigste Fall ist. Wer welche Datei nimmt, merkt man im Normalfall nicht.
Zu C: Strukturen aus C-Headern sind vorgesehen, aber nicht da. Wenn jemand danach fragt, ist das die ehrliche Antwort – und die nächste Folie zum Ausblick sagt, warum ein weiterer Übersetzer kein großes Vorhaben ist.

AUS FELDERN WERDEN SPALTEN

DIE DSECT, WIE SIE IN DER MAKRO-BIBLIOTHEK LIEGT

```
CUSTREC  DSECT
CUSTID   DS    CL8
CUSTNAM  DS    CL20
CUSTBAL  DS    PL5
CUSTCNT  DS    H
         DS    XL1
```

WAS DER COMPILER DARAUS MACHT

```
col(name='CUSTID'  format.fix(length=8)  type.string())
col(name='CUSTNAM' format.fix(length=20) type.string())
col(name='CUSTBAL' format.fix(length=5)
    type.integer(format.bcd(signed type=PACKED)))
col(name='_ALIGN_33_' format.fix(length=1) drain skip)
col(name='CUSTCNT' format.fix(length=2)
    type.integer(format.bin(signed endian=BIG)))
col(name='_FIL_36_'  format.fix(length=1) drain skip)
```

`_ALIGN_33_` steht nicht in der DSECT: der Assembler legt das Halbwort auf eine gerade Adresse. Die Satzlänge ist deshalb 37 und nicht 36.

FLAMv5.21-42969

Das ist der Kern der Einheit, und es ist bewusst ein echter Lauf: links die DSECT, rechts die Beschreibung, die FLAM daraus erzeugt hat.
Durchgehen, Feld für Feld: CL8 wird eine Zeichenkette, PL5 eine gepackte Dezimalzahl mit Vorzeichen, H eine zwei Byte große Binärzahl in Big-Endian – so, wie der Host sie schreibt.
Und dann der Punkt, der hängen bleibt: zwischen Betrag und Zähler steht eine Spalte, die in der DSECT nicht vorkommt. Der Assembler legt ein Halbwort auf eine gerade Adresse, also entsteht ein Füllbyte. FLAM kennt diese Regel und trägt es als übersprungene Spalte ein.
Wer die Beschreibung von Hand schreibt, muss an dieses Byte denken. Hier denkt der Compiler daran.



Die Verallgemeinerung der vorigen Folie, und die Folie, auf die man später zurückkommt, wenn eine Satzlänge nicht passt.
Die Ausrichtung ist bei allen drei Sprachen ein Thema: in PL/I heißt sie ALIGNED, in C wäre es das Gleiche, im Assembler steht sie gar nicht da, sondern folgt aus dem Typ.
Füllfelder: ein unbenanntes DS, ein FILLER im Copybook. Sie bleiben in der Beschreibung, damit die Offsets stimmen – aber sie stehen nicht in der Ausgabe. Bei COBOL kann man das umstellen, FILLER=COLUMN, ALIGN oder SKIP.
Und die Satzlänge: sie fällt aus der Beschreibung heraus. Das ist die Zahl, die man bei einer flachen Datei als RECLEN mitgibt. Stimmt sie nicht, läuft der erste Satz über – und genau so sieht dann auch die Fehlermeldung aus, sie nennt beide Längen.

COMPILE: DIE BESCHREIBUNG IM AUFRUF

FLCL CONV – DIESELBE BESCHREIBUNG IM KONVERTIERUNGSWEG

```
read.record(file='KUNDEN.DAT' recformat=FIX reclength=37
            table(compile(format='CUSTREC.asm' ccsid='IBM-1141')))
write.text(file='kunden.csv' table(format=csv))
```

ANGABE	WOFÜR SIE DA IST
CCSID	Zeichensatz der Daten; EBCDIC-Quelle, UTF-8-Ausgabe
ENDIAN	Byte-Reihenfolge für COMP-5 – BIG liest Host-Daten auf x86
MAXOCCURS	Obergrenze beim Auffalten von Wiederholungen
COLPATH / COLSEP	Spaltennamen: ganzer Pfad oder nur das Blatt
TRIM / FILLER	Füllzeichen und Füllfelder in der Ausgabe
DETECT	Zeichenspalten nicht festlegen, sondern erkennen lassen

FLAMv5.21-42969

Jetzt die Einordnung: COMPILE ist ein Unterobjekt von TABLE und steht dort neben den handgeschriebenen ROW-Einträgen. Beides zusammen ist erlaubt, und die Reihenfolge ist festgelegt: erst die eigenen Zeilen, dann die übersetzten.
Mehrere COMPILE-Objekte sind erlaubt, jedes mit eigenen Einstellungen. Das braucht man, wenn zwei Copybooks aus verschiedenen Zeichensätzen kommen.
Die Parameter in der Liste sind die, nach denen in der Praxis gefragt wird.
CCSID ist der häufigste: die Daten kommen in EBCDIC, die Ausgabe soll UTF-8 sein – das ist die eine Angabe, die fast immer nötig ist.
ENDIAN betrifft nur COMP-5. Normale Binärfelder sind nach COBOL-Standard immer Big-Endian, da ändert der Parameter nichts.

EIN ZEICHENFELD IST NICHT IMMER TEXT

- `CL20`, `PIC X(20)`, `CHAR(20)` sind **Bytefelder** – die Quelle sagt die Breite, nicht den Inhalt
- ohne Angabe wird daraus `TYPE.STRING()`: sicher, aber eine Zahl bleibt dann Text
- mit dem Schalter wird daraus `TYPE.DETECT()` **– Zahl, String, Datum oder HexBinary, je Wert entschieden**
- `LOCALE` sagt die Schreibweise: DE und US trennen Dezimalstellen und Datum verschieden

IM CONV EIN SCHALTER IM COMPILE, IM BCSV DAS DETECT-OBJEKT

```
read.record(... table(compile(format='CUSTREC.asm' detect)))


flcl BCSV "input='KUNDEN.DAT' format='CUSTREC.asm' \
           reclen=37 detect() output='kunden.zip'"
```

Beim BCSV genügt `detect()` ohne Inhalt; die Hinweise im Objekt steuern danach, was als Zahl, Datum oder Binäres gelesen werden darf.

FLAMv5.21-42969

Die Folie, die in der Praxis am meisten Ärger erspart, und sie steht bewusst vor den Anmerkungen – weil sie die Hälfte davon überflüssig macht.
Ein `CL20` im Assembler, ein `PIC X(20)` im Copybook, ein `CHAR(20)` in PL/I: das sind **Bytefelder**, keine Textfelder. Die Quelle sagt nur, wie breit das Feld ist. Was darin steht, sagt sie nicht – und in gewachsenen Beständen steht dort alles: eine Zahl als Ziffernfolge, ein Datum, eine Kennung, manchmal auch wirklich Binäres.
Der Übersetzer muss sich ohne weitere Angabe entscheiden und nimmt `TYPE.STRING()`. Das ist die sichere Wahl und oft die schlechtere: eine Zahl, die als Text ankommt, bleibt Text, und wer danach rechnen oder suchen will, fängt von vorne an.
Mit dem Schalter wird daraus `TYPE.DETECT()`: FLAM sieht sich den **Wert** an und entscheidet je Spalte, ob eine Zahl, ein String, ein Datum oder HexBinary vorliegt. `LOCALE` sagt, in welcher Schreibweise zu erwarten ist – DE und US unterscheiden sich beim Dezimaltrenner und beim Datum, und das ist genau die Stelle, an der eine Erkennung sonst falsch abbiegt.
Zum Aufruf: im Konvertierungsweg ist es ein Schalter im COMPILE. Beim BCSV schaltet die **Angabe des DETECT-Objektes selbst** um – `detect()` allein genügt, und die Hinweise darin steuern dann die Erkennung.



Die Folie, die im Vortrag die meisten Rückfragen bringt – weil sie ein Problem löst, das jeder kennt. Ein Copybook sagt PIC 9(8) oder PL4. Es sagt nicht, dass darin ein Datum steht. Es sagt auch nicht, dass dieses eine Textfeld aus einem anderen Zeichensatz kommt als der Rest des Satzes. Diese Information steht im Kopf des Entwicklers – oder im Kommentar daneben.
Genau da liest FLAM sie. Im Assembler ist es der Text hinter dem Operanden, im Copybook der Kommentar. Die Quelle bleibt übersetzbar: für den Assembler ist das eine Bemerkung, und Bemerkungen ignoriert er.
OMIT ist der unauffällige Vierte und der nützlichste: das Feld bleibt in der Längenrechnung, verschwindet aber aus der Ausgabe. So nimmt man ein Feld heraus, das nicht herausgegeben werden darf, ohne die Satzstruktur anzufassen.
Und der Hinweis, der die Folie entlastet: wer `TYPE.DETECT()` einschaltet, braucht DATE und CCSID in den meisten Fällen nicht mehr – die Erkennung sieht dem Wert selbst an, dass er ein Datum ist. Die Anmerkungen bleiben für die Fälle, in denen es nicht am Wert zu sehen ist, und für OMIT, das eine Entscheidung ist und keine Erkennung.



Die Datenbankentladung ist der Fall, der die Sache lohnend macht, denn hier hat niemand eine Beschreibung der Datei – aber jeder hat die Beschreibung der Segmente.
Eine IMS-Entladung ist nicht eine Struktur, sondern viele: je Segmenttyp eine, alle in derselben Datei, jede mit einem Umschlag davor. UNLOAD sagt, welcher Umschlag es ist.
Für die beiden DFSURGU0-Formate steht der Segmentname im Umschlag, also findet FLAM selbst, welches Copybook zu welchem Satz gehört. Bei der HD-Entladung steht dort kein Name, sondern ein Code – deshalb die SEG-CODE-Anmerkung über dem 01-Level.
Der letzte Punkt ist der, der in gemischten Beständen zählt: der Umschlag kann je Copybook überschrieben werden. Eine Datei, in der ein Teil aus der Datenbank kommt und ein Teil flach daneben liegt, geht in einem Lauf.
Und die Abschlusszeile hängt FLAM selbst an: ein Auffangeintrag, damit die Datei sauber zu Ende gelesen wird.

EINMAL ÜBERSETZEN, OFT BENUTZEN

ERSTER LAUF ÜBERSETZT, JEDER WEITERE LIEST DIE BESCHREIBUNG

```
flcl BCSV "input='KUNDEN.DAT' format='CUSTREC.asm' \
           rowout='custrec.row' output='kunden.zip'"

flcl BCSV "input='KUNDEN.DAT' rowin='custrec.row' \
           output='kunden.zip'"
```

- ist `ROWIN` gesetzt, wird `FORMAT` nicht angeschaut – der Compiler läuft nicht
- die Datei ist lesbarer Text: **ansehen, bevor zehn Millionen Sätze darauf treffen**
- sie gehört zur Verarbeitung und kann versioniert werden, nicht zum einzelnen Lauf
- **sie altert still** – ändert sich das Copybook, muss sie neu erzeugt werden

FLAMv5.21-42969

Die Betriebsfolie. Das Übersetzen kostet Zeit, und bei einer täglichen Verarbeitung kostet es sie jeden Tag neu, obwohl sich das Copybook im Jahr vielleicht zweimal ändert.
ROWOUT schreibt die übersetzte Beschreibung weg, ROWIN liest sie wieder ein. Ist ROWIN gesetzt, wird FORMAT gar nicht angeschaut.
Der zweite Nutzen ist der wichtigere und steht unten: die Datei ist lesbar.
Man kann hineinschauen, bevor man zehn Millionen Sätze darauf loslässt – und man kann sie anfassen, wenn der Compiler etwas anders aufgelöst hat, als man es wollte.
Damit ist die Beschreibung auch etwas, das man in die Versionsverwaltung legen kann. Sie gehört zur Verarbeitung, nicht zum Lauf.
Der Haken, der in die Stolperfallen gehört: die Datei veraltet still, wenn das Copybook sich ändert. Deshalb der letzte Punkt.

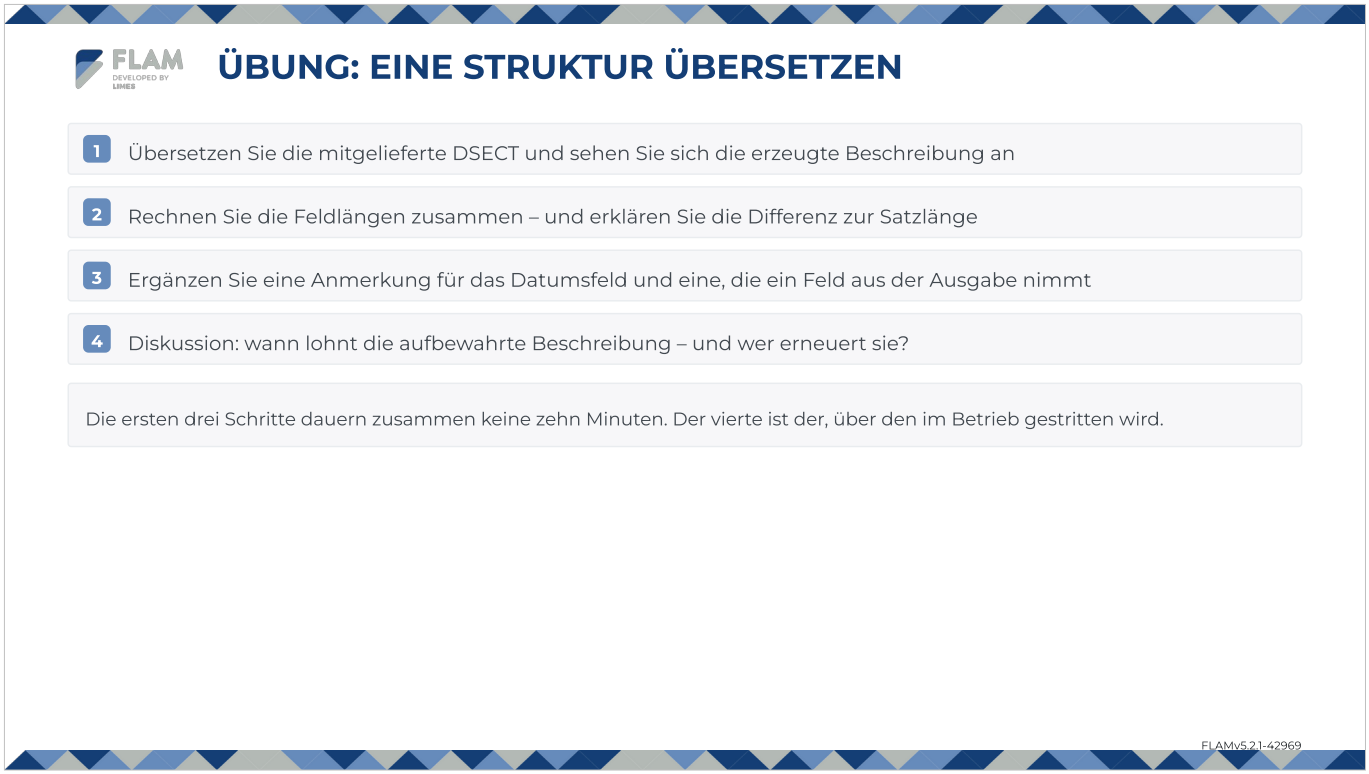
STOLPERFALLEN

FEHLERBILD	URSACHE UND ABHILFE
Satz zu lang, beim ersten Satz	Satzlänge ohne Ausrichtungsbytes gerechnet – die Meldung nennt beide Zahlen
Syntaxfehler, obwohl das Kommando stimmt	kein Übersetzer nimmt die Datei an – die Beschreibung bleibt leer
Umlaute und Vorzeichen falsch	`CCSID` fehlt – ohne Angabe gilt der Zeichensatz der Umgebung
Zahlen stimmen nicht auf x86	COMP-5 ist plattformabhängig; `ENDIAN=BIG` für Host-Daten
DSECT geht nicht durch	Makros, bedingte Assemblierung, COPY und DC werden nicht aufgelöst

Der Compiler **liest** die Quelle, er übersetzt sie nicht: was erst der Assembler oder der COBOL-Übersetzer erzeugen würde, steht ihm nicht zur Verfügung.

FLAMv5.21-42969

Die fünf, die in Supportfällen wirklich vorkommen.
Die Satzlänge ist die häufigste: die Beschreibung ergibt 37, die Datei wird mit 36 gelesen, und der erste Satz läuft über. Die Meldung nennt beide Zahlen – das ist die schnellste Diagnose im ganzen Umfeld.
Die zweite ist die, bei der man am längsten sucht: nimmt kein Übersetzer die Datei an, dann bleibt die Beschreibung leer, und der Fehler liest sich wie ein Syntaxfehler im Kommando. Die Frage ist dann nicht das Kommando, sondern die Datei – meist eine Quelle, die erst ein Makro oder ein COPY vollständig machen würde.
Der Zeichensatz ist der stille Fehler: ohne CCSID wird der Zeichensatz der Umgebung angenommen. Auf dem Host passt das zufällig, auf dem Linux-Rechner daneben nicht mehr.
Und das Letzte, weil es bei HLASM anders ist als erwartet: der Compiler liest die Quelle, er assembliert sie nicht. Bedingte Assemblierung, COPY, USING und DC-Werte werden nicht aufgelöst. Eine DSECT, die ihre Felder über Makros zusammensetzt, geht nicht durch.



Lösung:
Erstens: `flcl BCSV "input='KUNDEN.DAT' format='CUSTREC.asm' rowout='custrec.row' output='kunden.zip'"` – und dann in custrec.row hineinsehen.
Zweitens: die Summe der Feldlängen ist 36, in der Beschreibung steht 37.
Die Differenz ist das Ausrichtungsbyte vor dem Halbwort; es taucht als _ALIGN_33_ auf.
Drittens: hinter das Datumsfeld `DATE YYYYMMDD` schreiben, hinter das interne Feld `OMIT`.
Danach denselben Lauf noch einmal und die beiden CSV-Dateien vergleichen – eine Spalte weniger, eine Spalte als Datum.
Viertens, die Diskussionsfrage: ROWIN, wenn die Struktur stabil ist und täglich verarbeitet wird.
Dann aber mit einer Regel, wer die Datei neu erzeugt, wenn das Copybook sich ändert. Ohne diese Regel ist der Zwischenstand gefährlicher als der verlorene Lauf.



Die Folie, die zeigt, dass hier kein Einzelfall gebaut wurde, sondern eine Stelle, an der man andocken kann.
Technisch ist es eine Liste von Übersetzern, die der Reihe nach gefragt werden: "erkennst du diese Datei?" Wer ja sagt, liefert Zeilen und Spalten.
Alles danach – Typen, Umbau, CSV, XML, Archiv – ist für alle gleich. Ein neuer Übersetzer ist deshalb genau eine Funktion und kein neues Produkt.
Deshalb die Statusspalte: C und C++ ist der nächste, der Platz dafür steht schon in der Liste. XSD ist beschlossen und ist mehr als eine Strukturbeschreibung: ein XML-Schema beschreibt auch die **zulässigen Werte**, bis hin zum regulären Ausdruck. Gedacht ist es vor allem für den Zahlungsverkehr – SEPA, SWIFT-MX und was sonst in dieser Form kommt. Damit ist es nicht nur Compiler, sondern auch Prüfung.
DFDL und ASN.1 sind erwogen, nicht zugesagt. DFDL ist interessant, weil es genau für diesen Zweck erfunden wurde: eine Sprache, um Fremdformate zu beschreiben. ASN.1 kommt aus der Telekommunikation und dem Zertifikatswesen und ist dort die Beschreibung schlechthin.
Wenn im Vortrag jemand nach einem Format fragt, das hier nicht steht: das ist die richtige Frage, und die ehrliche Antwort ist, dass es eine Frage von Aufwand ist und nicht von Architektur.

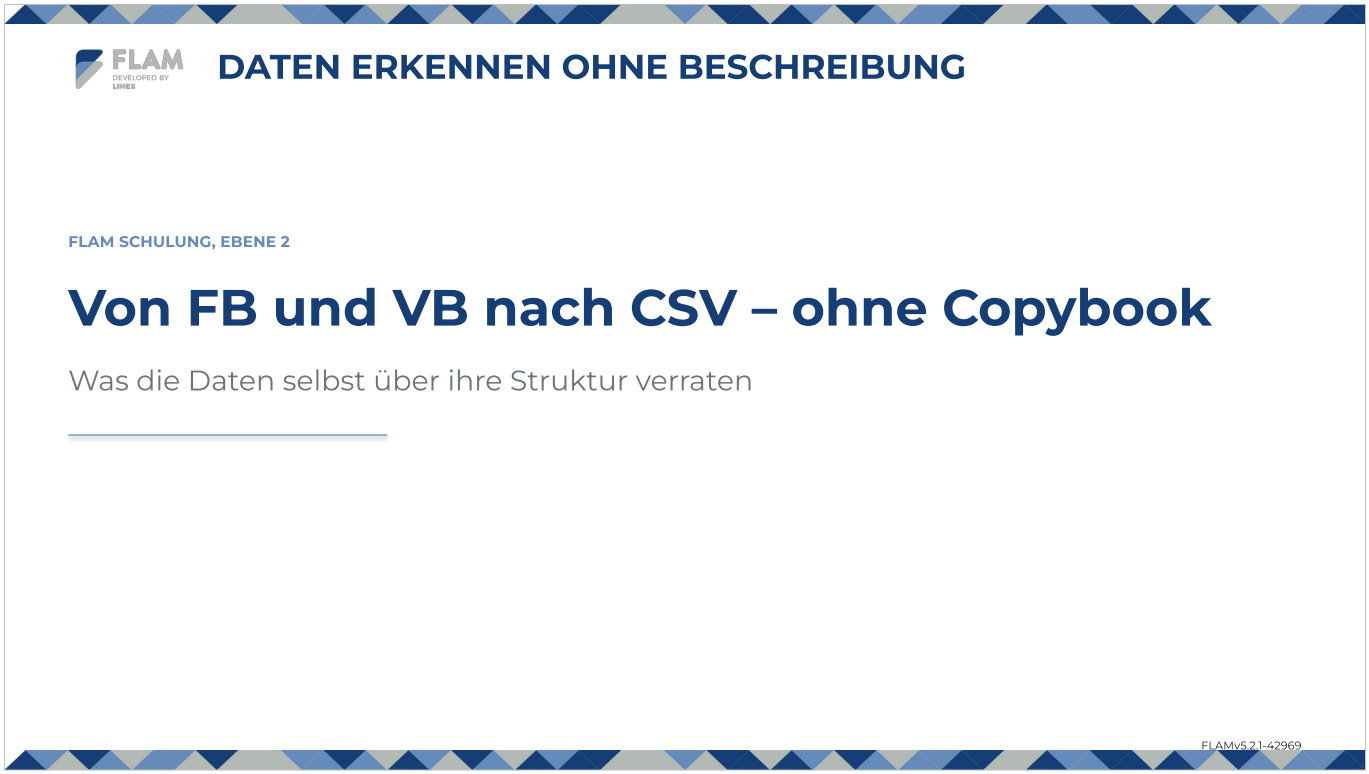
WAS BLEIBT

- drei Sprachen, der Reihe nach durchprobiert – der Dateiname spielt keine Rolle
- Zeichenfelder sind Bytefelder: mit `TYPE.DETECT()` entscheidet der Wert, nicht die Vorgabe
- der Compiler bringt mit, was in der Quelle nicht steht: Ausrichtung, Füllfelder, Satzlänge
- vier Anmerkungen am Feld steuern ihn, ohne die Quelle unübersetzbar zu machen
- IMS-Entladungen bekommen ein Copybook je Segment, der Umschlag ist überschreibbar
- einmal übersetzt, aufbewahrt und wiederverwendet – mit einer Regel für das Erneuern

Und wenn es keine Quelle mehr gibt? Dann erkennt FLAM die Struktur aus den Daten – die nächste Einheit.

FLAMv5.21-42969

Zusammenfassung und Übergang. Der letzte Punkt ist die Brücke zur nächsten Einheit: dort gibt es keine Quelle mehr, und trotzdem entsteht eine Tabelle.



Die Fortsetzung der vorigen Einheit, mit der dritten Antwort auf dieselbe Frage: niemand hat die Beschreibung.
Das ist kein Randfall. Die Anwendung ist abgeschaltet, die Bibliothek mit den Copybooks ist nicht mitgewandert, der Entwickler ist in Rente – und die Datei liegt noch da, weil sie zehn Jahre aufbewahrt werden muss.
Wichtig für die Erwartungshaltung, und das sage ich lieber gleich am Anfang als auf Nachfrage: hier wird geschätzt, nicht gewusst. Die Einheit zeigt genauso, wo die Schätzung endet.



Ziel drei ist das ehrliche: wer die Grenze kennt, verkauft das Ergebnis richtig. Wer sie nicht kennt, wird von ihr eingeholt.



Die Problemfolie. Drei Fälle, und alle drei kommen aus echten Kundengesprächen – ohne Namen zu nennen.
Der erste ist der häufigste und der ärgerlichste: die Daten müssen aufbewahrt werden, das Programm, das sie lesen konnte, nicht.
Der zweite ist der Übernahmefall. Es gibt sie noch, aber niemand weiß, welche zu welcher Datei gehört – und das Ausprobieren ist teurer als das Erkennen.
Der dritte ist der, der am Anfang eines Projektes steht: man will überhaupt erst wissen, was in der Datei steht, bevor man entscheidet, was damit geschieht.
Der Punkt unten ist der, den man festhalten sollte: die Erkennung ersetzt das Copybook nicht. Sie macht die Datei zugänglich.

EIN KOMMANDO, KEINE BESCHREIBUNG

MEHR BRAUCHT ES IM EINFACHSTEN FALL NICHT

```
flcl ACSV "input='KUNDEN.DAT' output='kunden.zip'"
```

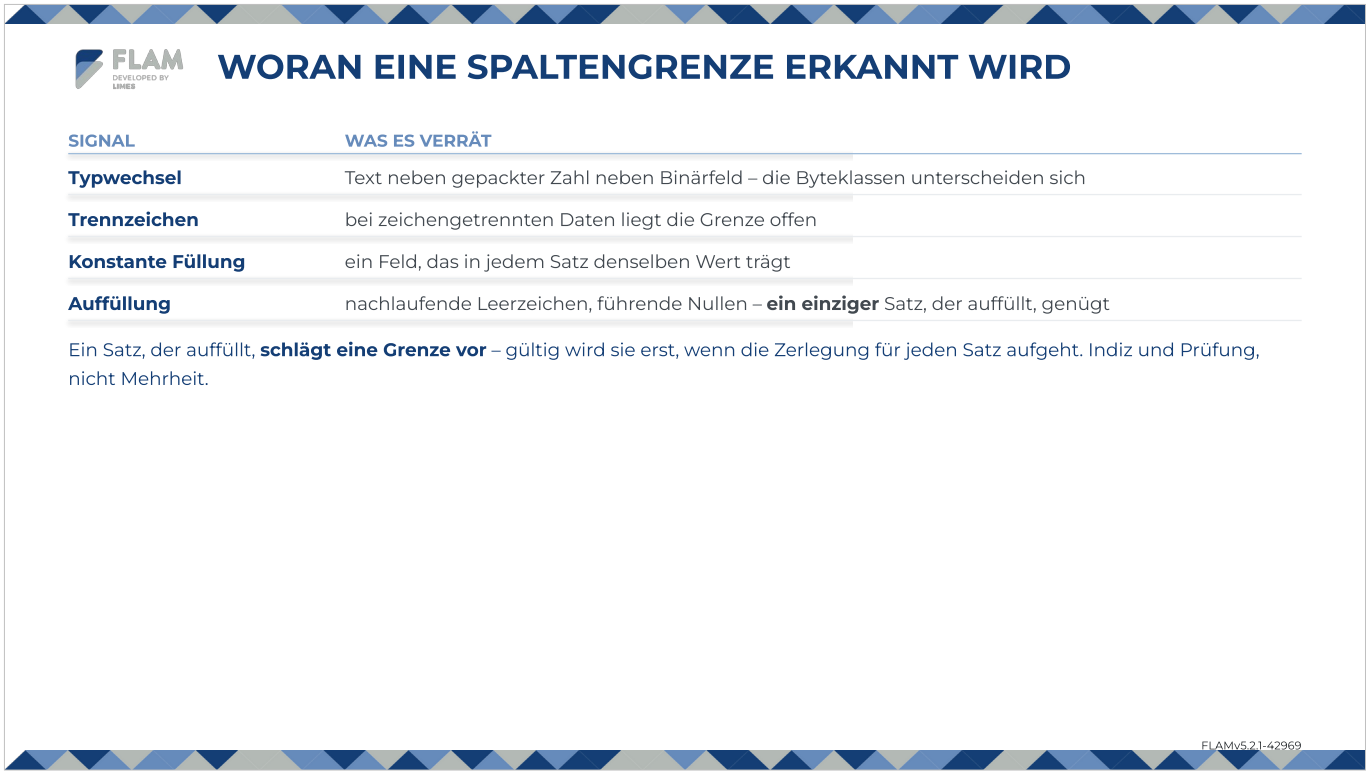
- das Ziel ist ein **ZIP-Archiv**: je erkannter Struktur ein Member, Muster `[name].[table].csv`
- die Ausgabe ist **UTF-8-CSV**, unabhängig davon, in welchem Zeichensatz die Quelle liegt
- gepackte Dateien (gzip, bzip2, xz, zstd, lz4) werden an den Kennbytes erkannt und ausgepackt
- ohne Ausgabeangabe geht es nach stdout – der schnelle Blick in einen fremden Bestand

> **DAS KANN NUR FLAM**
>
> Eine Datei, zu der es **keine Beschreibung mehr gibt**, wird mit einem Aufruf zu typisierten Spalten – ohne dass jemand eine Struktur rekonstruiert.

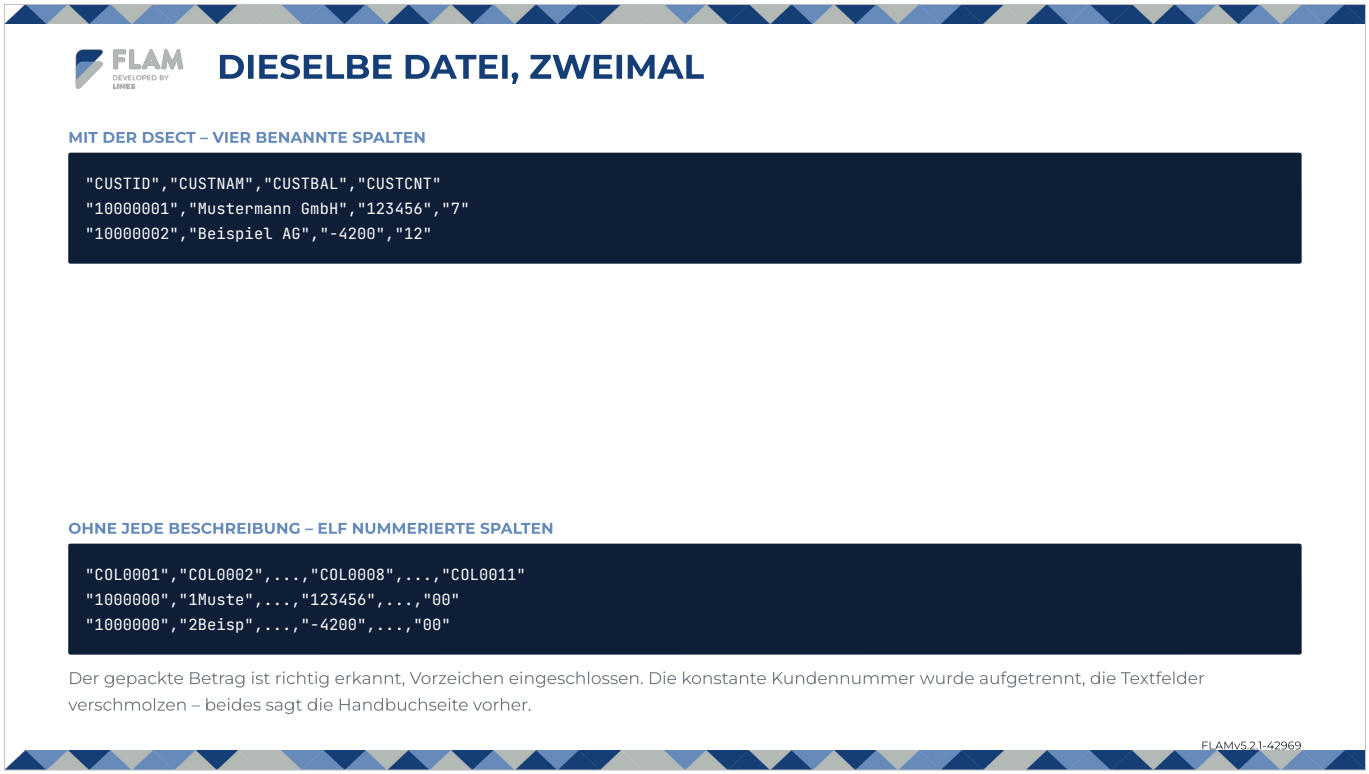
Die einfachste Folie der Einheit, und sie wirkt im Vortrag am besten, wenn man sie wirklich kurz hält: Eingabe, Ausgabe, fertig.
Das Ziel ist ein ZIP-Archiv, weil in einer Datei mehrere Tabellen stecken können – je erkannter Struktur ein Member. Das Muster für den Membernamen ist einstellbar.
Dass ohne Ausgabeangabe nach stdout geschrieben wird, ist für das schnelle Hineinsehen praktisch: einmal durch weniger Sätze laufen lassen und anschauen, bevor man den ganzen Bestand anfasst.



Die Folie, die den Umfang klarstellt, bevor es in die Tiefe geht – sonst denkt der halbe Saal, es gehe nur um Hostdateien mit fester Satzlänge.
Es sind drei Erkenner unter einem Kommando. Welcher greift, entscheiden die Daten; man kann es aber auch sagen, wenn man es weiß.
Der feste ist der schwierigste und steht deshalb im Rest der Einheit im Vordergrund. Er liefert auch mehrere Satzarten aus einer Datei – je Satzart eine eigene Zeilenbeschreibung, also genau das, was eine Überlagerung im Copybook wäre.
Der zeichengetrennte ist der angenehmste, und das zeigt das Beispiel: hier liegt die Spaltengrenze offen, und wenn eine Kopfzeile da ist, stehen sogar die **Namen** in der Datei. Dazu kommt der Typ je Spalte – im Beispiel hat er das Semikolon als Trenner, das Komma als Dezimaltrenner und das Datumsmuster selbst gefunden.
Der dritte liest XML, wo die Struktur im Pfad steckt.
Wenn jemand fragt, warum das überhaupt getrennt ist: weil die Signale verschieden sind. Im zeichengetrennten Fall ist die Grenze ein Zeichen, im festen Fall muss sie aus den Bytes erschlossen werden. Das ist ein anderer Beruf, auch wenn die Ausgabe dieselbe ist.



Jetzt das Wie, und das ist die Folie, nach der die Einheit verstanden ist oder nicht.
Der Erkenner liest Bytes. Eine Spaltengrenze kann er nur dort setzen, wo die Daten selbst etwas verraten. Vier Dinge verraten etwas, und sie stehen in der Tabelle.
Der Typwechsel ist der stärkste: zwischen Text und gepackter Zahl liegt eine Grenze, weil die Byteklassen sich unterscheiden.
Die Füllzeichen sind der subtile Fall und der häufigste Gewinn: ein Textfeld, das nicht ganz gefüllt ist, endet in Leerzeichen, eine Ziffernfolge beginnt mit führenden Nullen.
Und jetzt der Punkt, auf den es ankommt und der oft falsch erwartet wird:
dafür genügt **ein einziger** Satz. Sieben Sätze, die jedes Feld randvoll füllen, sagen nichts darüber, wo die Grenze liegt – sie widersprechen aber auch nicht. Der eine Satz, der auffüllt, weiß mehr als alle anderen zusammen, und wenn die Zerlegung, die er vorschlägt, für jeden Satz aufgeht, dann gilt sie.
Das ist eine Entscheidung über Indizien und keine Abstimmung. Eine Mehrheit würde hier die falsche Frage beantworten.
Der Zeichensatz ist der vierte, und er wirkt doppelt: er entscheidet, wie der Inhalt gelesen wird, und bei festen Sätzen auch, wo die Grenze liegt – denn der Erkenner trennt Text von Nicht-Text Byte für Byte.



Die Gegenüberstellung, und sie ist ein echter Lauf über dieselbe 37-Byte-Datei – links mit der DSECT aus der vorigen Einheit, rechts ohne jede Beschreibung.
Was gut gegangen ist: der gepackte Betrag wurde als gepackte Dezimalzahl mit Vorzeichen erkannt, mit richtigem Wert und richtigem Vorzeichen. Die Satzlänge stimmt.
Was nicht gehen konnte: die Namen. Sie stehen nirgends in der Datei. Und die Kundennummer wurde zerlegt, weil ihre ersten sieben Stellen in allen betrachteten Sätzen gleich waren – aus einer Konstanten wird eine eigene Spalte.
Das ist der Punkt, an dem man im Vortrag innehalten sollte: beides ist richtig. Die rechte Seite ist keine schlechtere Lösung, sie ist eine Lösung für einen anderen Fall. Wer die DSECT hat, nimmt die DSECT.

WO DIE ERKENNUNG ENDET

- mehrere **Zahlenfelder** nebeneinander, gleicher Zeichensatz, kein Trenner – werden eine breite Zahl
- mehrere **Textfelder** nebeneinander, in **jedem** Satz voll gefüllt – werden ein breites Textfeld
- **gepackte oder binäre Felder** ohne eigenes Kennzeichen – nicht von einem breiteren Feld desselben Typs zu unterscheiden
- **Spaltennamen** gibt es nicht: sie stehen in keiner Satzdatei, nur in der Beschreibung

Das ist nicht die Grenze des Verfahrens, sondern die Grenze der Information in der Datei. Für eine garantiert richtige Spaltenstruktur führt der Weg über die Beschreibung – siehe die vorige Einheit.

FLAMv5.21-42969

Die Grenzfolie, und sie steht bewusst früh – nicht am Ende, wo sie nach Kleingedrucktem aussieht. Drei Fälle, und alle drei haben dieselbe Ursache: zwei benachbarte Felder gleichen Typs ohne Trenner sind in den Bytes nicht unterscheidbar. Fünf Zahlenfelder hintereinander werden eine breite Zahl, zwei Textfelder werden ein breites Textfeld.
Wichtig ist das Wort **jedem** in der zweiten Zeile: es genügt ein einziger Satz, der ein Feld nicht ganz füllt, damit die Grenze sichtbar wird. Erst wenn kein einziger Satz auffüllt, ist sie wirklich nicht da.
Das ist keine Schwäche der Umsetzung, das ist die Information, die in der Datei fehlt. Ein Mensch mit denselben Bytes und ohne Copybook kann es auch nicht entscheiden.
Der Nutzen der Folie im Verkaufsgespräch ist genau dieser Satz: wir sagen vorher, was nicht geht. Das macht die Zusage glaubwürdig, die wir für den Rest geben.
Und die Konsequenz steht unten: für eine garantiert richtige Spaltenstruktur führt der Weg über die Beschreibung. Die Erkennung ist der Weg dorthin, nicht der Ersatz dafür.



Die Stellschraube, die am meisten bewirkt und am wenigsten bekannt ist.
Vorgabe sind acht Sätze. Das ist für eine gleichförmige Datei richtig und kostet nichts. Für eine Datei mit mehreren Satzarten ist es zu wenig – und das merkt man daran, dass eine Satzart gar nicht auftaucht.
Die zweite Zahl ist die Vertrauensschwelle: wie viele der angesehenen Sätze sich einig sein müssen. Vorgabe zwei. Das ist der Grund, warum ein Kopfsatz, der anders aussieht als der Rest, die Erkennung nicht kaputt macht.
Das Beispiel aus dem Handbuch ist gut zum Erzählen: acht Sätze angesehen, sieben mit Länge 80, einer mit Länge 20. Erkannt wird 80 – der Kopfsatz stört nicht.
Und der Zusammenhang zur vorigen Folie: mehr Sätze schaden der Auffüll-Erkennung nicht. Im Gegenteil – je mehr Sätze angesehen werden, desto wahrscheinlicher ist einer darunter, der ein Feld nicht ganz füllt und damit seine Grenze zeigt.

FLAM
DEVELOPED BY LIMES

HINWEISE STATT RATEN

DIE DREI HINWEISE, DIE FAST IMMER REICHEN

```
detect(hints(chrset=EBCDIC endian=BIG locale=DE))
```

- **CHRSET** ist der wichtigste: er entscheidet den Inhalt – und bei festen Sätzen auch die Grenzen
- ohne Angabe wird er aus der Byteverteilung geschätzt; bei textarmen EBCDIC-Daten geht das schief
- **ENDIAN** und **LOCALE** für Binärzahlen, Dezimaltrenner und Datumsschreibweise
- feiner je Typ: Jahresbereiche für Datumsfelder, Vorzeichenkonvention, Nullwert-Liste
- ein ausdrücklicher Wert wird **nie** überstimmt – Hinweise sind Vorgaben, keine Vorschläge

Die Folie für den, der schon etwas über seine Daten weiß – und das ist fast immer jemand im Raum.
Der Zeichensatz ist der wichtigste Hinweis, und zwar aus einem Grund, den man erklären sollte: er entscheidet nicht nur, wie der Inhalt gelesen wird, sondern bei festen Sätzen auch, wo die Grenzen liegen. Der Erkenner trennt Text von Nicht-Text Byte für Byte, und welche Bytes Text sind, hängt vom Zeichensatz ab.
Ohne Angabe wird er aus der Byteverteilung geschätzt, und das geht bei EBCDIC-Daten mit wenig Text gern daneben – weil in EBCDIC fast alles ab 0x40 druckbar aussieht und Binärfelder dann wie Text zerfallen.
Deshalb die Regel: wenn Sie es wissen, sagen Sie es. Ein ausdrücklicher Wert wird nie überstimmt.
Die übrigen Hinweise sind feiner – je Typ lässt sich einschränken, was überhaupt in Frage kommt, bis hin zu Jahresbereichen für Datumsfelder und der Liste der Zeichenketten, die als Nullwert gelten.

VARIABLE SÄTZE UND IMS-ENTLADUNGEN

HINWEIS	WAS DAMIT GELESEN WIRD
NONE	Vorgabe – der Umschlag wird selbst erkannt
RDW	einfache Sätze variabler Länge mit Längenfeld
IMSL / IMSS	IMS DFSURGU0, langer bzw. kurzer Umschlag
IHDU	IMS HD-Entladung, DFSURGL0 – das physische Abbild

EINE IMS-ENTLADUNG OHNE EINE EINZIGE SEGMENTBESCHREIBUNG

```
flcl ACSV "input='IMS.UNLOAD' output='segmente.zip' \
           detect(format=IMSS hints(chrset=EBCDIC))"
```

Ohne den Hinweis wird der Umschlag meist selbst erkannt. Der kurze DFSURGU0-Umschlag ist der Fall, in dem das Kennzeichen oft nicht in die Stichprobe fällt.

FLAMv5.21-42969

Der Teil, der im Hostumfeld den Unterschied macht.
Eine Datei mit variablen Sätzen trägt vor jedem Satz ein Längenfeld, und eine IMS-Entladung trägt zusätzlich einen Umschlag mit Segmentnamen oder Segmentcode. Wer das nicht wegrechnet, erkennt Unsinn: der Umschlag sieht aus wie die ersten Spalten.
FLAM erkennt diese Umschläge normalerweise selbst. Der Hinweis ist für die Fälle, in denen die Stichprobe zu dünn ist – der Handbuchtext nennt den kurzen DFSURGU0-Umschlag, dessen Kennzeichen selten im ersten Block auftaucht.
Praktischer Rat für den Betrieb: wenn man weiß, welches Entladeformat vorliegt, schreibt man es hin. Die Erkennung ist dann schneller und sicher, und man diskutiert hinterher nicht über einen Lauf, den man nicht wiederholen kann.

ERKENNEN, ANSEHEN, FESTSCHREIBEN

EINMAL ERKENNEN – DANACH MIT DER GEPRÜFTEN BESCHREIBUNG ARBEITEN

```
flcl ACSV "input='KUNDEN.DAT' output='kunden.zip' \
           detect(maxrec=FULL rowout='kunden.row')"

flcl BCSV "input='KUNDEN.DAT' rowin='kunden.row' \
           output='kunden.zip'"
```

1. erkennen lassen und die gefundene Beschreibung herausschreiben
2. hineinsehen und korrigieren: Spalten benennen, Verschmolzenes trennen, Typen richtigstellen
3. ab dann die festgeschriebene Beschreibung benutzen – es wird nicht mehr geschätzt

In der Datei steht **nicht nur eine** Beschreibung: was sonst noch in Frage kam, steht als Kommentar darunter, jede Variante mit ihrer Bewertung. Korrigieren heißt deshalb oft nur auswählen.

FLAMv5.21-42969

Die Folie, die die beiden Einheiten zusammenbindet, und der Rat, den man im Betrieb geben sollte.
Die Erkennung schreibt auf Wunsch die gefundene Beschreibung weg. Diese Datei ist lesbarer Text und genau das Format, das die vorige Einheit als aufbewahrte Beschreibung kennt.
Damit wird aus dem Schätzen ein Verfahren mit drei Schritten: einmal erkennen lassen, hineinschauen und korrigieren – Spalten benennen, zwei verschmolzene Felder trennen, einen Typ richtigstellen – und ab dann mit der festgeschriebenen Beschreibung arbeiten.
Der Gewinn ist doppelt. Die tägliche Verarbeitung schätzt nicht mehr, sondern liest eine feste Beschreibung. Und man hat wieder eine Beschreibung der Daten, die man in die Versionsverwaltung legen kann – für einen Bestand, der keine mehr hatte.
Das ist der Satz, mit dem man diese Einheit verkauft.
Und der Punkt, der die Korrektur überhaupt erst praktikabel macht: in der Datei steht nicht nur eine Beschreibung. Was sonst noch in Frage kam, steht als Kommentar darunter, jede Variante mit ihrer Bewertung. Wer korrigiert, rät also nicht, sondern wählt – und sieht, wie knapp die Entscheidung war. Bei einer Datei, deren Struktur eindeutig ist, steht dort nur eine Zeile.

STOLPERFALLEN

FEHLERBILD	URSACHE UND ABHILFE
Struktur wirkt stimmig, Werte sind Unsinn	Satzlänge falsch angenommen – die Sätze liegen gegeneinander verschoben
Binärfelder zerfallen in viele Textspalten	Zeichensatz als EBCDIC geschätzt oder umgekehrt – `HINTS.CHRSET` setzen
Eine Satzart fehlt in der Ausgabe	Stichprobe zu klein – `MAXREC` erhöhen, notfalls `FULL`
Spalten mit festem Wert und Bedingung	konstantes Feld in zu kleiner Stichprobe – mehr Sätze ansehen
Jeden Tag eine andere Spaltenzahl	täglich neu erkannt – einmal erkennen, prüfen, festschreiben

FLAMv5.21-42969

Die fünf aus der Praxis.
Die Satzlänge ist die häufigste und die unauffälligste: wird sie falsch angenommen, ist die erkannte Struktur in sich stimmig und trotzdem vollständig falsch, weil die Sätze gegeneinander verschoben liegen. Das sieht man nur, wenn man in die Ausgabe schaut.
Der Zeichensatz ist die zweite und wirkt doppelt, wie besprochen.
Die dritte ist die unterschätzte: mehrere Satzarten in einer Datei, und die Stichprobe sieht nur die erste. Erkennbar daran, dass ein Member fehlt.
Die vierte kennen wir von der Gegenüberstellung: konstante Felder werden zu eigenen Spalten mit einer Bedingung darauf. In einer kleinen Stichprobe ist jeder Wert leicht konstant.
Und die fünfte ist die Verfahrensfalle: wer jeden Tag neu erkennen lässt, bekommt möglicherweise jeden Tag eine andere Spaltenzahl – weil die Daten des Tages andere Signale tragen. Tägliche Verarbeitung gehört auf eine festgeschriebene Beschreibung.

ÜBUNG: EINEN FREMDEN BESTAND ÖFFNEN

1. Lassen Sie eine unbekannte Satzdatei ohne jede Angabe erkennen und sehen Sie sich das Ergebnis an
2. Wiederholen Sie es mit allen Sätzen und schreiben Sie die Beschreibung heraus
3. Geben Sie den Zeichensatz als Hinweis mit – und vergleichen Sie die Spaltenzahl
4. Benennen Sie zwei Spalten in der Beschreibung und fahren Sie den Lauf damit erneut

Diskussion: ab welchem dieser Schritte würden Sie das Ergebnis einem Fachbereich zeigen?

Lösung:
Erstens: `flcl ACSV "input='UNBEKANNT.DAT' output='x.zip'"` und in die Member sehen. Die Spaltenzahl notieren.
Zweitens: derselbe Lauf mit `detect(maxrec=FULL rowout='x.row')`. In der Regel ändert sich die Spaltenzahl, weil mehr Sätze mehr Fälle zeigen.
Beide Beschreibungen nebeneinanderlegen.
Drittens: mit `detect(hints(chrset=EBCDIC))` wiederholen. Bei EBCDIC-Daten fällt die Zahl der Spalten meist deutlich, weil Binärzonen nicht mehr als Text zerfallen.
Viertens: in x.row zwei Spalten benennen, die man wiedererkennt, und den Lauf mit `rowin` wiederholen. Jetzt stehen Namen in der Kopfzeile.
Die Diskussionsfrage am Ende: ab wann würden Sie dieses Ergebnis einem Fachbereich zeigen? Die erwartete Antwort ist – nach Schritt vier, nicht nach Schritt eins.



Die Zukunftsfolie, und sie knüpft genau an das an, was zwei Folien vorher schon da war: die Alternativen im ROWOUT.
Heute ist es ein rechnerisches Urteil. Der Erkenner bewertet jede Variante nach Struktur, Typsicherheit und danach, wie viele Sätze sie fehlerfrei durchlaufen – und stellt sie der Reihe nach hin. Die Entscheidung trifft ein Mensch.
Das ist genau die Stelle, an der ein Sprachmodell etwas kann, was eine Formel nicht kann: Es sieht nicht nur die Struktur, sondern die **Werte**.
Ob in Spalte drei Kontonummern stehen oder Mengen, ob die Zahl in Spalte fünf ein Datum sein könnte, ob eine Aufteilung fachlich Sinn ergibt – das ist Lesen und nicht Rechnen.
Deshalb die zwei geplanten Schritte. Erstens: das Modell bekommt alle Kandidaten samt einiger Sätze als Tabelle und sagt, welcher der wahrscheinlichste ist. Zweitens, und im Alltag vielleicht noch wertvoller:
es schlägt **sprechende Namen** vor. Aus COL0003 wird KONTONUMMER, aus dem Satz ein Name wie BUCHUNG.
Was dabei wichtig ist, und das gehört in den Vortrag: der Vorschlag geht in dieselbe lesbare Beschreibungsdatei wie alles andere. Es entscheidet also niemand hinter dem Rücken des Betriebs – man sieht den Vorschlag, man prüft ihn, man nimmt ihn an oder nicht. Die Verarbeitung selbst läuft danach ohne Modell.
Der Anschluss ist bewusst anbieterneutral gehalten und kann auch gegen ein lokal betriebenes Modell laufen. Bei Kundendaten ist das regelmäßig die Frage, die als erste kommt.

WAS BLEIBT

- ein Aufruf bringt eine unbeschriebene Satzdatei nach CSV – auch aus einer IMS-Entladung
- drei Erkenner unter einem Kommando: feste Sätze, zeichengetrennte Daten, XML
- Grenzen entstehen dort, wo die Daten ein Signal tragen: Typwechsel, Trenner, Füllung, Auffüllung
- was kein Signal trägt, verschmilzt – das ist die Grenze der Information, nicht des Verfahrens
- Hinweise verengen den Suchraum; der Zeichensatz ist der wirksamste
- Stichprobengröße und Einigungsschwelle entscheiden, was überhaupt gefunden wird
- mehrdeutige Fälle liefern mehrere bewertete Lesarten – künftig schlägt ein Modell die wahrscheinlichste und sprechende Namen vor

Einmal erkennen, prüfen, festschreiben – danach wird nicht mehr geschätzt. Ein Bestand ohne Beschreibung hat am Ende wieder eine.

FLAMv5.21-42969

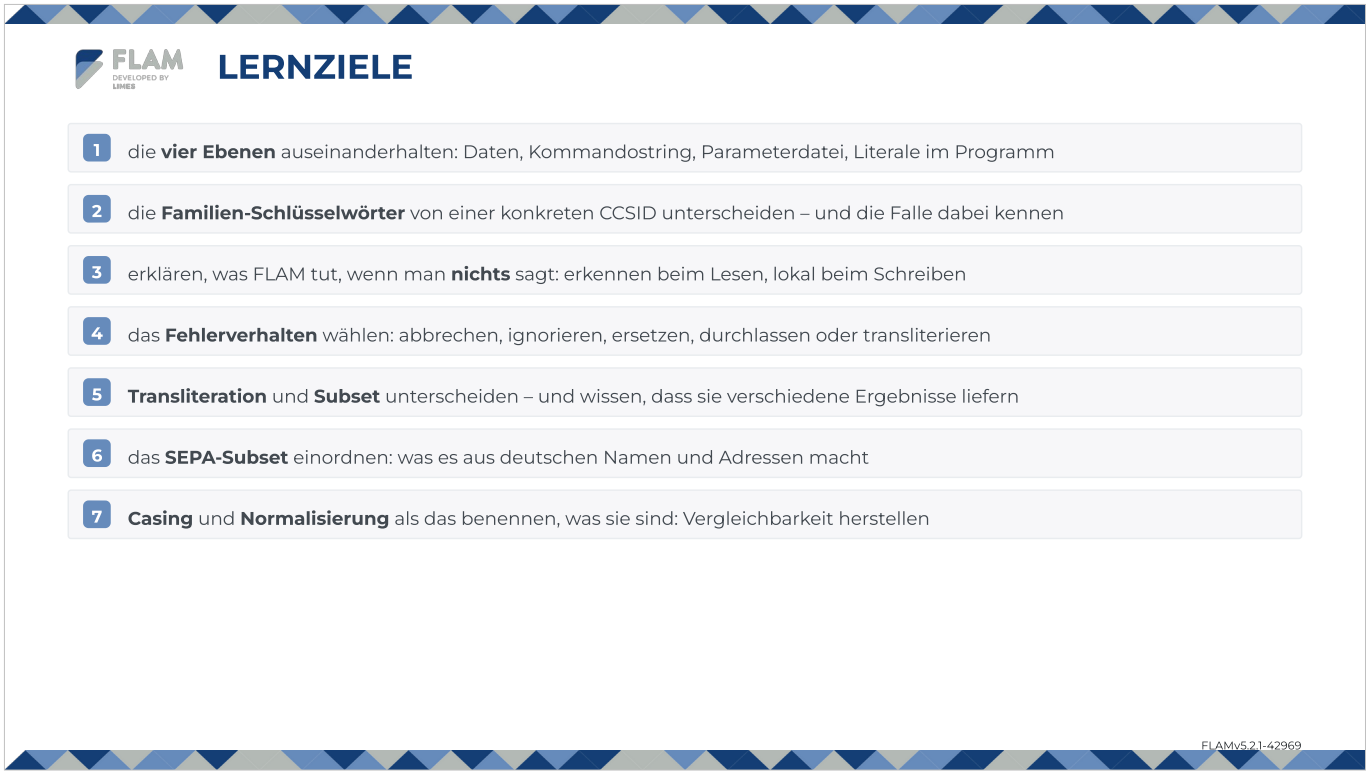
Zusammenfassung. Der letzte Punkt ist der, der die beiden Einheiten verbindet und im Gedächtnis bleiben soll: die Erkennung ist der Weg zu einer Beschreibung, nicht der Ersatz dafür.

EBENE 2 – FUNKTIONSTHEMEN: DATEN

Zeichensätze

Vier Ebenen, 186 Codepages – und was passiert, wenn ein Zeichen im Ziel nicht vorkommt

Das Thema, das in der Praxis die meisten Datenfehler verursacht – und das einzige, bei dem ein Fehler oft erst beim Empfänger auffällt.
Diese Einheit geht deutlich über das hinaus, was in der Konvertierungs-Einheit stand. Dort ging es um die CCSID als Angabe: welche Datei ist in welchem Zeichensatz. Hier geht es um zwei Dinge mehr: um das, was **dazwischen** passiert, wenn ein Zeichen im Ziel nicht vorkommt – und darum, dass der Zeichensatz nicht an **einer** Stelle steht, sondern an **vier**.
Wichtig gleich am Anfang: **wir nutzen seit langem nicht mehr libiconv, sondern unser eigenes Modul.** Das ist kein Detail, denn daraus folgen die Fähigkeiten dieser Einheit: Normalisierung, Subsets, Best-Fit-Mapping, eigene Mappings, Reporting und Casing. Eine Bibliothek, die nur umkodiert, kann das nicht.
Vorausgesetzt wird die Einheit *Konvertierung mit FLUC*.



Die Lernziele als Vertrag. Das erste ist das, was die Einheit zusammenhält: wer die vier Ebenen auseinanderhält, stellt die richtige Frage, wenn etwas schiefgeht – nämlich **welche** Ebene betroffen ist.
Das vierte ist das, worauf der Datenteil hinausläuft: es gibt nicht **eine** richtige Umkodierung, es gibt eine fachliche Entscheidung, und das Werkzeug hat für jede davon einen Schalter.
Das sechste ist für ein Haus im Zahlungsverkehr das wichtigste – und es wird auf einer eigenen Folie gemessen, nicht behauptet.



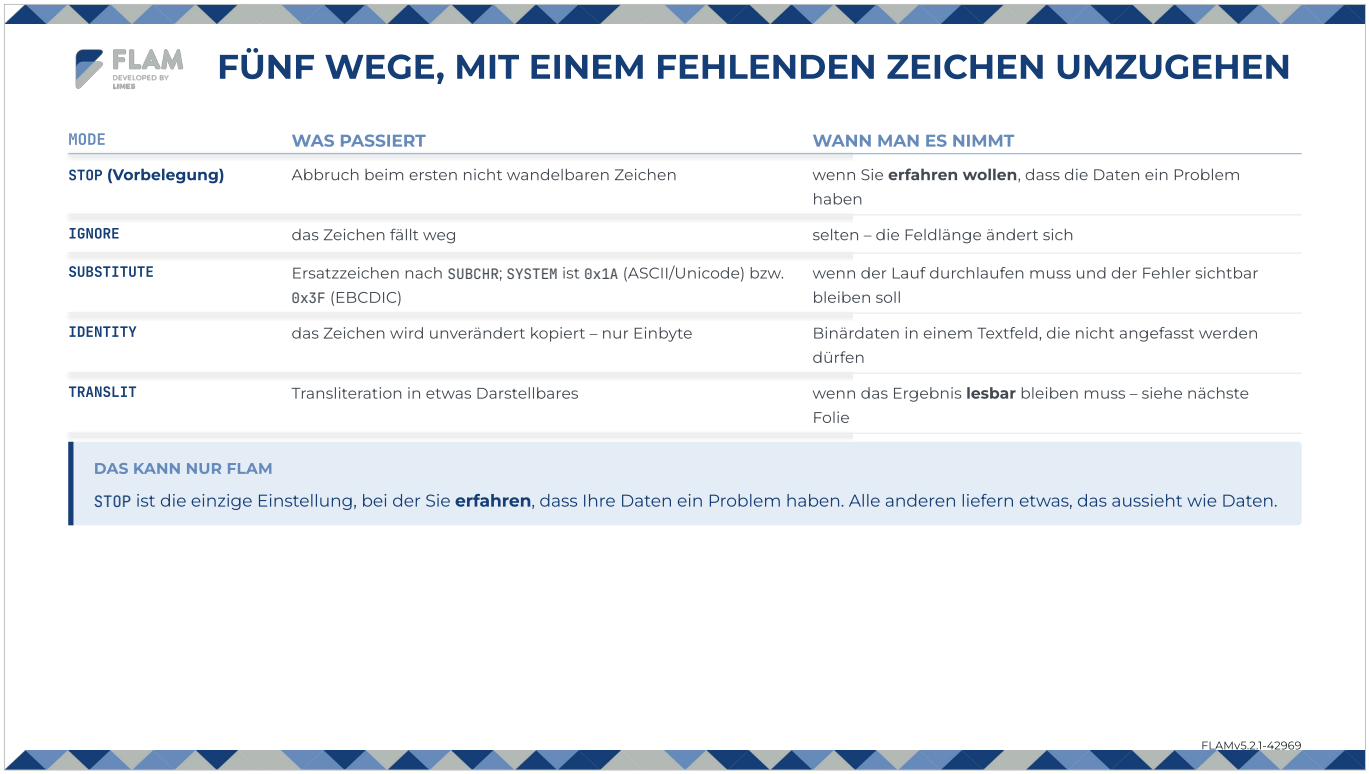
Die Landkarte für die ganze Einheit, und die Folie, die man aufschlagen sollte, wenn im Betrieb etwas mit Zeichen schiefgeht. Die erste Frage ist nie "welche CCSID", sondern **welche Ebene**.
Ebene 1, die Daten. Der Inhalt der Dateien, die wir verarbeiten. Das ist das, woran man zuerst denkt, und der größte Teil dieser Einheit dreht sich darum: Erkennung, Fehlerverhalten, Transliteration, Subsets, Casing, Normalisierung.
Ebene 2, der Kommandostring. In welchem Zeichensatz ist das `flcl`-Kommando selbst geschrieben? Auf ASCII- und UTF-8-Plattformen ist das keine Frage.
Auf EBCDIC ist es eine, weil Klammern, Tilde und Dollar je nach Codepage an verschiedenen Stellen liegen.
Ebene 3, die Parameterdatei. Dieselbe Frage für eine Datei, die den Kommandostring enthält – und die schärfer wird, sobald die Datei von einer **anderen Plattform** kommt, etwa per SSH vom Host geholt.
Ebene 4, die Literale im Programm. Wer gegen unsere APIs programmiert, hat einen Zeichensatz, in dem der **Compiler** seine Literale ablegt – und der kann ein anderer sein als der, in dem die Variablen zur Laufzeit stehen.
Diese vier sind unabhängig voneinander, und genau das macht das Thema komplex. Die gute Nachricht: für jede Ebene gibt es genau einen Ort, an dem man es sagt. Die stehen auf den folgenden Folien.



Ab hier die Daten. Erst der Bestand, dann die Falle.
FLAM kennt **186 Codepages**, und man kann sie sich selbst ansehen: `flcl INFO "GET.CCSIDS"` listet sie mit Nummer, Zeichensatzfamilie und Encoding-String. Die Verteilung ist aufschlussreich: **82 ASCII, 67 EBCDIC**, der Rest UTF-8, UTF-16, UTF-32 und die UCS-Varianten in beiden Byte-Reihenfolgen.
Dass zwei Drittel der Codepages Einbyte-Zeichensätze sind, erklärt, warum dieses Thema überhaupt so groß ist: in der Unicode-Welt gibt es das Problem "Zeichen kommt im Ziel nicht vor" kaum, in der Einbyte-Welt ständig.
Jetzt die Falle, und die kostet regelmäßig einen halben Tag: neben den konkreten Codepages gibt es **Familien-Schlüsselwörter** – `ASCII`, `EBCDIC`, `SYSTEM` und `LOCAL`. Die sind bequem, aber sie sind **nicht** eine Codepage, sondern lösen zur Laufzeit auf die Voreinstellung der Plattform auf.
Konkret: auf einer UTF-8-Linux-Box löst `to=ASCII` zu **UTF-8** auf – nicht zu 7-Bit-ASCII. Das habe ich selbst nachgemessen, und ich habe zuerst gedacht, das Werkzeug ignoriere den Parameter. Es tut das Richtige, es bedeutet nur etwas anderes, als man annimmt.
Die Regel daraus: **in der Produktion eine konkrete CCSID angeben.** Wer 7-Bit will, sagt `IBM-367` und nicht `ASCII`. Wer 1141 will, sagt `1141` und nicht `EBCDIC`.



Die Folie, die eine häufige Rückfrage vorwegnimmt: "ich habe keine CCSID angegeben, und es hat funktioniert – warum soll ich sie dann angeben?" **Beim Schreiben** nimmt FLAM den lokalen Zeichensatz der Maschine. Das ist meist richtig und immer vorhersagbar.
Beim Lesen wird erkannt. Und die Erkennung ist gut, aber sie ist nicht kostenlos und nicht eindeutig:
Sie kostet CPU – jedes Byte wird angesehen.
Sie ist bei Einbyte-Zeichensätzen **nicht eindeutig**. EBCDIC lässt sich gut von ASCII und UTF-8 unterscheiden, weil die Bytebereiche verschieden sind.
Aber ob es deutsches oder amerikanisches EBCDIC ist, steht in den Bytes nicht drin – 1141 und 1140 unterscheiden sich nur in wenigen Positionen.
Deshalb entscheidet FLAM das am **Länderkennzeichen der Maschine**: auf einem deutsch eingestellten System wird erkanntes EBCDIC zu 1141, auf einem amerikanisch eingestellten zu 1140.
Und das ist der Punkt, an dem es gefährlich wird: **derselbe Job liefert auf zwei Maschinen zwei Ergebnisse**, wenn sie verschieden eingestellt sind. Das findet man nicht im Test, das findet man in der Produktion.
Ein Detail, das man vorführen kann: die Erkennung meldet eine **Konfidenz**. Im Log steht etwa "DETECTED CHARSET FOR CHARACTER CONVERSION: UTF-8 TXT (100.00%)". Eine Datei mit Umlauten macht UTF-8 eindeutig; eine Datei aus reinem 7-Bit-Text kann alles sein, und dann steht dort ein niedrigerer Wert. Wer den Wert sieht, versteht sofort, warum Angeben besser ist.



Der Kern des Datenteils. Ein Zeichen kommt im Ziel nicht vor – und jetzt gibt es fünf Antworten. Keine davon ist die richtige; es ist eine **fachliche** Entscheidung.
`STOP` bricht beim ersten nicht wandelbaren Zeichen ab. Das ist die Vorbelegung, und sie ist teuer, weil jedes Zeichen geprüft werden muss – aber sie **findet falsch kodierte Daten**. Das ist unser Alleinstellungsmerkmal gegenüber Werkzeugen, die still ersetzen.
`IGNORE` lässt das Zeichen weg. Die Länge des Feldes ändert sich damit, und bei festen Satzlängen ist das ein Problem.
`SUBSTITUTE` ersetzt durch ein Ersatzzeichen. Welches, sagt `SUBCHR` – `SYSTEM` nimmt die Plattformkonvention, `0x1A` für ASCII und Unicode, `0x3F` für EBCDIC. Man kann auch eine Liste angeben.
`IDENTITY` kopiert das Zeichen unverändert. Das geht nur bei Einbyte-Wandlungen, und es ist der Weg, wenn man Binärdaten in einem Textfeld hat und sie nicht anfassen will.
`TRANSLIT` schaltet die Transliteration ein – aus `ü` wird etwas, das im Ziel darstellbar ist. Das ist die nächste Folie, denn dort liegt die eigentliche Entscheidung.
Der Satz, den man mitgeben sollte: **`STOP` ist die einzige Einstellung, bei der Sie erfahren, dass Ihre Daten ein Problem haben.** Alle anderen liefern etwas, das aussieht wie Daten. Wer aus CPU-Gründen umstellt, sollte `REPORT` einschalten – dazu die Folie über das Reporting.



Die wichtigste Folie der Einheit, und die Zahlen darauf sind **gemessen**, keine Beispiele. Derselbe Eingabetext, dieselbe Zielcodepage (`IBM-367`, also 7-Bit-US-ASCII), einmal mit der Systemtabelle und einmal mit dem SEPA-Subset.

Die Systemtabelle ist die klassische ICONV-Transliteration. Sie will **verlustfrei lesbar** sein und setzt dafür Markierungszeichen ein: aus `ü` wird `"u`, aus `é` wird `'e`. Und sie kennt Regeln: `ß` wird `ss`, das Eurozeichen wird `EUR`. Für einen Menschen bleibt der Text lesbar.

Das SEPA-Subset macht etwas anderes. Sein Vorrat ist **US-ASCII** – die Einbyte-Zeichen von UTF-8, also alles unter Codepunkt 128. Und die Abbildung dorthin ist **ein Zeichen auf ein Zeichen**: aus `ü` wird `u`, aus `Ä` wird `A`, aus `é` wird `e`. Kein Markierungszeichen, keine Längenänderung.

Und jetzt die zwei Ergebnisse, bei denen man im Raum kurz innehalten sollte, weil sie für deutsche Namen und Adressen zählen:

`Straße` wird `Strase` – mit einem `s`. Die Systemtabelle macht `Strasse`. `Strase` ist kein Wort, und in einem Adressfeld ist das ein Datenqualitätsproblem.

`Müller` wird `Muller` – nicht `Mueller`. Das ist ein anderer Name.

Das Eurozeichen wird `E` – nicht `EUR`.

Das ist keine Kritik am Subset, und das sollte man auch so sagen: das Subset definiert einen **Zeichenvorrat**, nicht eine Schreibkonvention. Es bildet auf US-ASCII ab, und ein Ein-zu-eins-Mapping hält die Feldlängen stabil – was in einem Format mit festen Längen genau richtig ist. Die deutsche Konvention `ue`/`ss` ist eine fachliche Regel, die daneben liegt. Aber man muss es **wissen**, bevor man es auf einen Namensbestand anwendet.

Die praktische Empfehlung: wer Namen und Adressen nach SEPA überführt, sollte vorher entscheiden, ob er die deutsche Konvention (`ue`, `ss`) haben will. Dann braucht er eine **eigene Tabelle** über `USRTAB` mit einem Dateinamen statt eines der eingebauten Schlüsselwörter – das ist genau der Zweck der Möglichkeit, eine eigene anzugeben.



Vier Subsets sind eingebaut, und sie sind nicht willkürlich gewählt – jedes entspricht einer Vorgabe, die ein deutsches Haus kennt.
`SEPA` bildet auf **US-ASCII** ab: alles unter Codepunkt 128, also die Einbyte-Zeichen von UTF-8. Das ist der, um den es im Zahlungsverkehr geht.
Und hier ein wichtiger Hinweis, damit niemand mehr verspricht, als das Werkzeug hält: das Subset prüft den **Zeichenvorrat**, nicht die Feldregeln eines Zahlungsformats. Nachgemessen: mit `USRTAB=SEPA` und `MODE=STOP` läuft der gesamte druckbare ASCII-Bereich unverändert durch – einschließlich Raute, At-Zeichen, geschweifter Klammern und Tilde, die in vielen Zahlungsverkehrsfeldern nicht zugelassen sind. Was das Subset leistet, ist die Zusicherung: **hier ist kein Mehrbyte-Zeichen mehr drin.** Wer auf einen engeren Feldvorrat prüfen will, gibt bei `USRTAB` eine eigene Tabelle an.
`NPAS` ist der Vorrat für das gesetzliche Meldewesen: neuer Pass, XOEF, String.Latin. Wer Meldedaten liefert, braucht diesen.
`DELA` ist deutsches Latin als Schnittmenge von IBM-1141, ISO8859-15 und CP1252. Das ist der praktische Vorrat für einen Bestand, der zwischen Host, Unix und Windows wandert – alles, was in allen drei darstellbar ist.
`DLAX` ist `DELA` erweitert um XOEF.
Der Nutzen ist nicht nur die Wandlung. Man kann ein Subset auch als **Prüfung** benutzen: mit `MODE=STOP` und einem Subset bricht der Lauf ab, sobald ein Zeichen auftaucht, das die Vorgabe nicht erlaubt. Damit prüft man einen Bestand **vor** der Lieferung, statt die Ablehnung des Empfängers abzuwarten.
Das ist der Satz, den ein Compliance-Mensch hören will: die Vorgabe steht nicht in einer Prüfroutine, die jemand geschrieben hat und pflegen muss, sondern ist ein Parameter.



Die Folie, die aus einem Parameter ein Werkzeug macht – und die etwas zeigt, das es sonst nicht gibt. `USRTAB` nimmt nicht nur die vier eingebauten Schlüsselwörter. Es nimmt **eine eigene Tabelle**, und darin definiert der Kunde vier verschiedene Dinge.
Die vier Arten unterscheidet ein Vorzeichen, und der Unterschied ist wichtig:
Ohne Vorzeichen oder mit `+` ist es eine Transliteration – sie greift **nur**, wenn das Zeichen im Ziel nicht vorkommt. Das ist das Best-Fit-Mapping: für den Normalfall ändert sich nichts.
Mit `` *ist es ein Mapping* – es wird **immer** ausgeführt, ob das Zeichen im Ziel vorkommt oder nicht.
Mit `^` ist es ein Casing – eigene Groß-, Klein- oder Faltungsregeln, und sie greifen nur, wenn `CASE=USRTAB` gesetzt ist.
Mit `-` wird ein Codepunkt abgeschaltet – und genau damit definiert man ein **eigenes Subset**. Auch als Bereich: `(-00-7F)` schaltet ganz US-ASCII ab, `(+39-30)` schaltet die Ziffern wieder ein.
Jetzt der Punkt, an dem man langsam sprechen sollte, denn er ist der eigentliche Grund für diese Folie: beim **Mapping** darf auch **links** eine Liste stehen. Transliteration ist ein Zeichen auf bis zu acht; ein Mapping ist **mehrere auf mehrere**. Und damit wird aus einer Umkodierungstabelle eine **Eingabehilfe**.
Der Fall aus der Praxis: ein Haus arbeitet auf der 3270 in EBCDIC, muss aber NPAS unterstützen – also Zeichen liefern, die auf der Tastatur gar nicht vorkommen. Statt eines anderen Terminals definiert es sich Kürzel: `&OM` wird das griechische große Omega. Man kann die Kürzel immer zweistellig halten oder mit einem Semikolon abschließen, wie es ins eigene Schema passt.
Der Sachbearbeiter tippt `&OM`, im Archiv steht UTF-8 mit dem richtigen Zeichen. **Das kann sonst niemand** – und es kostet eine Zeile in einer Tabelle.
Zwei Dinge noch, die die Tabelle praktisch machen: sie **ergänzt oder überschreibt** eine Systemtabelle, man muss also nicht alles selbst definieren – `SYSTAB=ICONV` plus eigene Zeilen für die eigenen Fälle. Und die Transliteration arbeitet **rekursiv**: ist ein Ersatzzeichen selbst nicht im Ziel darstellbar, wird dessen Ersetzung angewandt.
Die Tabelle darf eine Datei sein – mit allen Regeln der Dateinamen, also auch `ssh://` von einem anderen System – oder ein String im Parameter, der mit einem Doppelpunkt beginnt. Für kurze Tabellen und für Tabellen, die eine Anwendung selbst erzeugt, ist der String der bequemere Weg.
Zur Vertiefung gibt es eine ausführliche Manpage; sie steht unter `doc/man_cnvchr_usrtab.txt` und im Handbuch am Parameter.



Casing sieht nach einer Kleinigkeit aus und ist eine, solange man nur Englisch verarbeitet. Sechs Einstellungen, und drei davon braucht man nur außerhalb des Englischen:
`UPPER` und `LOWER` sind das Erwartete.
`FOLD` ist **Case Folding für den Vergleich**. Das ist nicht dasselbe wie Kleinschreibung: Folding bildet auf eine Form ab, in der zwei Schreibweisen desselben Wortes gleich sind. Wer Namen vergleichen will, nimmt `FOLD` und nicht `LOWER` – das ist der Unterschied zwischen "sieht gleich aus" und "ist gleich".
`SUPPER` und `SLOWER` sind das **spezielle** Casing. Das braucht man, weil die einfache Regel in manchen Sprachen falsch ist: das deutsche `ß` wird in Großschreibung zu `SS` – ein Zeichen wird zwei. Und im Türkischen hat das `i` ein eigenes Großbuchstaben-Verhalten. Eine Ein-zu-eins-Tabelle kann das nicht, deshalb gibt es die spezielle Variante.
`USRTAB` schließlich casing nach eigener Tabelle – für Hausregeln.
Der praktische Hinweis: **Casing gehört zur Vergleichbarkeit, nicht zur Darstellung.** Wer einen Bestand durchsuchbar machen will, normalisiert Groß- und Kleinschreibung einmal beim Einlesen und vergleicht danach. Wer es bei jedem Vergleich macht, bezahlt es bei jedem Vergleich.
Und der Bogen zur Indizierungs-Einheit: wer später ohne Rücksicht auf Groß- und Kleinschreibung suchen will, muss das **vor** dem Anlegen des Archivs entscheiden. Die Indizierung kennt keine nachträgliche Änderung.



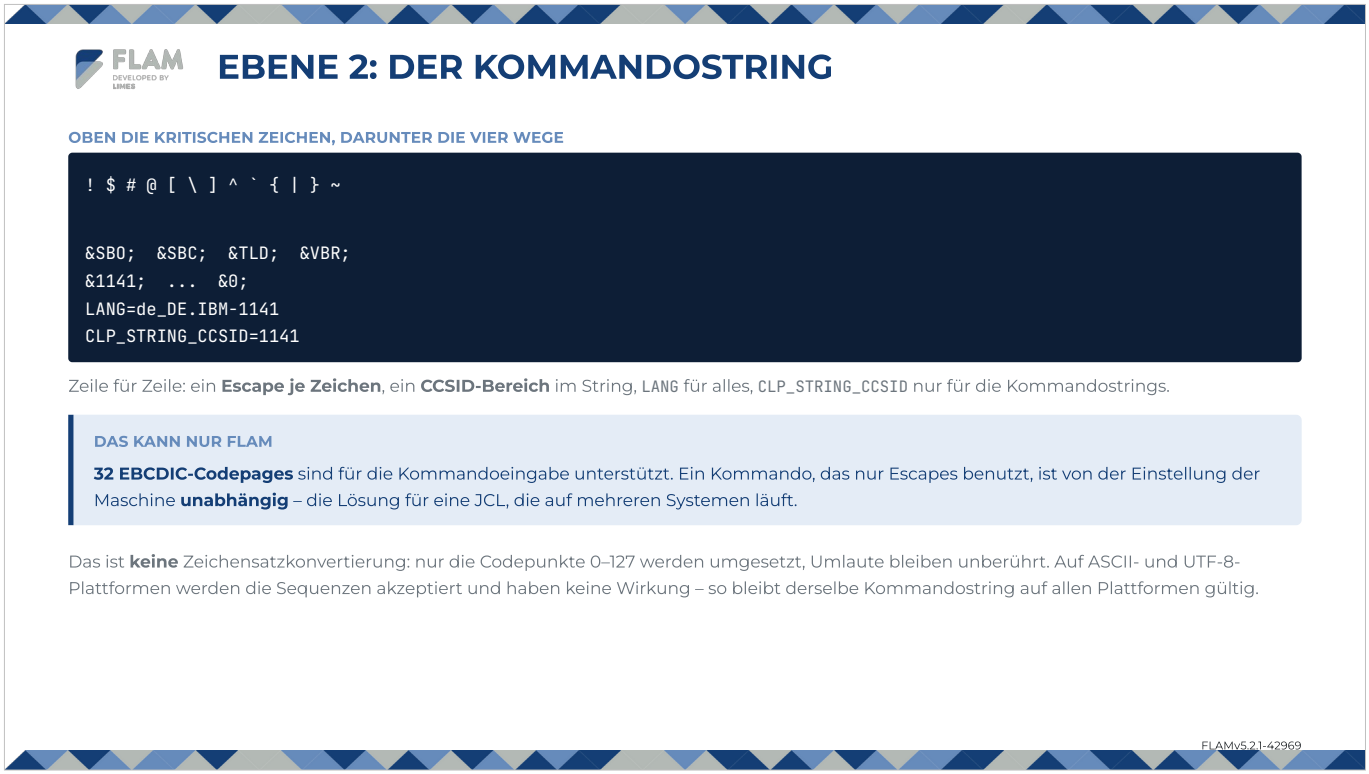
Das Thema, das die meisten überrascht, und das in einer Unicode-Welt unvermeidlich ist.
Der Sachverhalt: `ü` kann in Unicode **zwei verschiedene Byte-Folgen** sein. Einmal als eigenes Zeichen, einmal als `u` plus ein kombinierendes Trema. Beide sehen gleich aus. Beide sind korrekt. Und ein Byte-Vergleich sagt "verschieden".
Das ist kein akademisches Problem: macOS liefert traditionell die zerlegte Form, Windows und der Host die zusammengesetzte. Wer Dateien aus beiden Welten in einen Bestand legt, hat beides darin – und findet dann seine Datensätze nicht.
Die Antwort ist Normalisierung, und `CMBFRM` hat sie:
`NFD` ist die kanonische **Zerlegung** – alles wird in Grundzeichen plus Kombinationszeichen zerlegt.
`NFC` ist Zerlegung **und** anschließende kanonische Zusammensetzung – alles wird zur kompakten Form. Das ist die Form, die man für einen Bestand will.
`AUTO` erkennt kombinierte Zeichen und setzt sie zusammen, wenn es sinnvoll ist.
`ON` schaltet die Unterstützung für kombinierte Zeichen ein, und die Hilfe sagt ausdrücklich "slow" – das ist eine ehrliche Angabe und man sollte sie weitergeben.
`OFF` schaltet sie ab.
Die Empfehlung für einen Bestand, der durchsuchbar sein soll: **beim Einlesen auf `NFC` normalisieren.** Dann ist jedes Zeichen einmal dargestellt, und der Vergleich funktioniert. Danach kostet es nichts mehr.



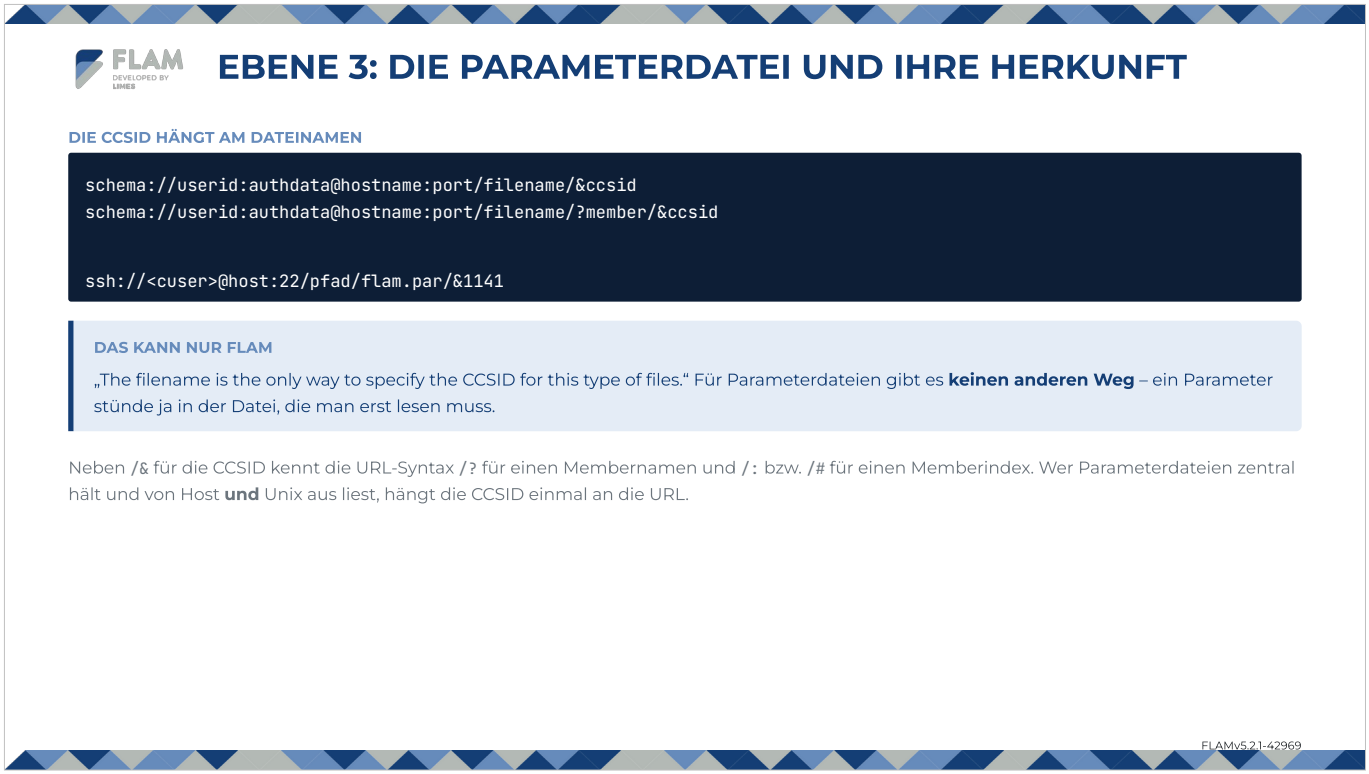
Die Folie, die aus einer Einstellung eine nachvollziehbare Verarbeitung macht – und die, die man braucht, sobald man `STOP` verlässt.
Die Lage: `SUBSTITUTE` oder `TRANSLIT` liefern eine Datei, die aussieht wie eine gute Datei. Der Empfänger sieht nicht, dass drei Zeichen ersetzt wurden. Man selbst auch nicht.
`REPORT` löst das: es protokolliert die **ignorierten und ersetzten Zeichen**. Ziel ist `STDOUT`, `STDERR` oder eine Datei – also auch eine Datei, die man archiviert.
Damit hat man drei Betriebsarten, und die sollte man bewusst wählen:
`MODE=STOP` ohne Report – für alles, wo ein Zeichenfehler ein Datenfehler ist. Der Lauf bricht ab, man sieht es sofort.
`MODE=SUBSTITUTE` mit Report – für alles, wo der Lauf durchlaufen muss.
Man bekommt das Ergebnis **und** die Liste dessen, was angefasst wurde. Das ist der Kompromiss, den die meisten Häuser wollen.
`MODE=SUBSTITUTE` ohne Report – das ist die Einstellung, die man nicht haben will, und genau die ist am weitesten verbreitet, weil sie nie auffällt.
Der Satz für den Raum: **ein Report macht aus einer stillen Ersetzung eine Entscheidung.**
Und für einen Auditor ist der Unterschied genau der, um den es geht.



Zwei Dinge, die nur beim Übergang zwischen den Welten vorkommen.
Die Zeilenenden. EBCDIC hat historisch zwei Zeichen, die als Zeilenende auftreten: `0x15` (New Line) und `0x25` (Line Feed). Welches ein Programm erzeugt, hängt vom Compiler und vom Alter des Codes ab. `ENL2LF` ersetzt beim Lesen `0x15` durch `0x25`, `ELF2NL` beim Schreiben `0x25` durch `0x15`.
Wer eine Datei bekommt, in der die Zeilenenden nicht erkannt werden, braucht meist genau einen dieser beiden Schalter.
Das BOM. Beim Wechsel zwischen Unicode-Formen steht am Anfang eine Byte-Order-Mark. `BOM` schreibt eine, `KEPBOM` behält eine vorhandene. Wer eine Datei an ein Programm gibt, das kein BOM erwartet, muss das ausschalten – und wer eine an Windows gibt, will es meist haben.
Und die Leseseite dazu, weil sie genauso oft gebraucht wird: als `CCSID` kann man **`BOMUTF`** oder **`BOMUCS`** angeben. Dann entscheidet die Byte-Order-Mark der Datei, welche Unicode-Form gelesen wird – `BOMUTF` nimmt UTF-8 als Rückfallebene, `BOMUCS` UTF-16. Das ist die Antwort auf "ich weiß nicht, in welcher Unicode-Form die Datei kommt".
Damit ist Ebene 1 abgeschlossen. Ab der nächsten Folie geht es nicht mehr um die Daten, sondern um das Kommando, die Parameterdatei und das Programm.



Jetzt die Ebene, die niemand erwartet – und die auf dem Host regelmäßig zuschlägt.
Der Sachverhalt: vierzehn Satzzeichen liegen in den EBCDIC-Codepages **nicht an festen Positionen**. Ausrufezeichen, Dollar, Raute, At, die eckigen und geschweiften Klammern, Backslash, Zirkumflex, Backtick, Pipe und Tilde. Genau diese Zeichen sind aber die Syntax unserer Kommandosprache.
Ein Kommando mit einer eckigen Klammer, das auf einer 1141-Maschine getippt und auf einer 1047-Maschine gelesen wird, ist ein anderes Kommando.
Wie FLAM das löst – und das ist eine Menge Arbeit, die im Produkt steckt:
Erstens über `LANG`. Der Parser wertet die Umgebungsvariable aus. `LANG=de_DE.IBM-1141` ist explizit, `LANG=de_DE` reicht auch – dann wird die CCSID aus dem Sprachcode abgeleitet: DE ist 1141, US ist 1140, C ist 1047. Ohne `LANG` gilt 1047. **32 EBCDIC-Codepages** sind für die Kommandoeingabe unterstützt.
Zweitens über Escapes. Man kann jedes kritische Zeichen als Sequenz schreiben: `&SBO;` für die eckige Klammer auf, `&TLD;` für die Tilde, `&VBR;` für den senkrechten Strich. Ein Kommando, das nur Escapes benutzt, ist von der Einstellung der Maschine **unabhängig**. Das ist die Lösung für eine JCL, die auf mehreren Systemen laufen soll.
Drittens über CCSID-Sequenzen im String. `&1141;` schaltet die Interpretation mitten im String auf 1141 um, `&0;` zurück auf die Systemcodepage. Das ist Ebene 4 und kommt gleich.
Viertens über `CLP_STRING_CCSID` – die CCSID nur für die CLP-Strings, unabhängig von `LANG`.
Zwei Dinge noch, die man sagen muss: das ist **keine Zeichensatzkonvertierung**, es werden nur die Codepunkte 0 bis 127 umgesetzt. Umlaute werden nicht angefasst. Und auf ASCII- und UTF-8-Plattformen gibt es das Problem nicht – die Escapes werden dort trotzdem akzeptiert und haben keine Wirkung. Das ist Absicht: **man kann plattformunabhängige Kommandostrings schreiben.**



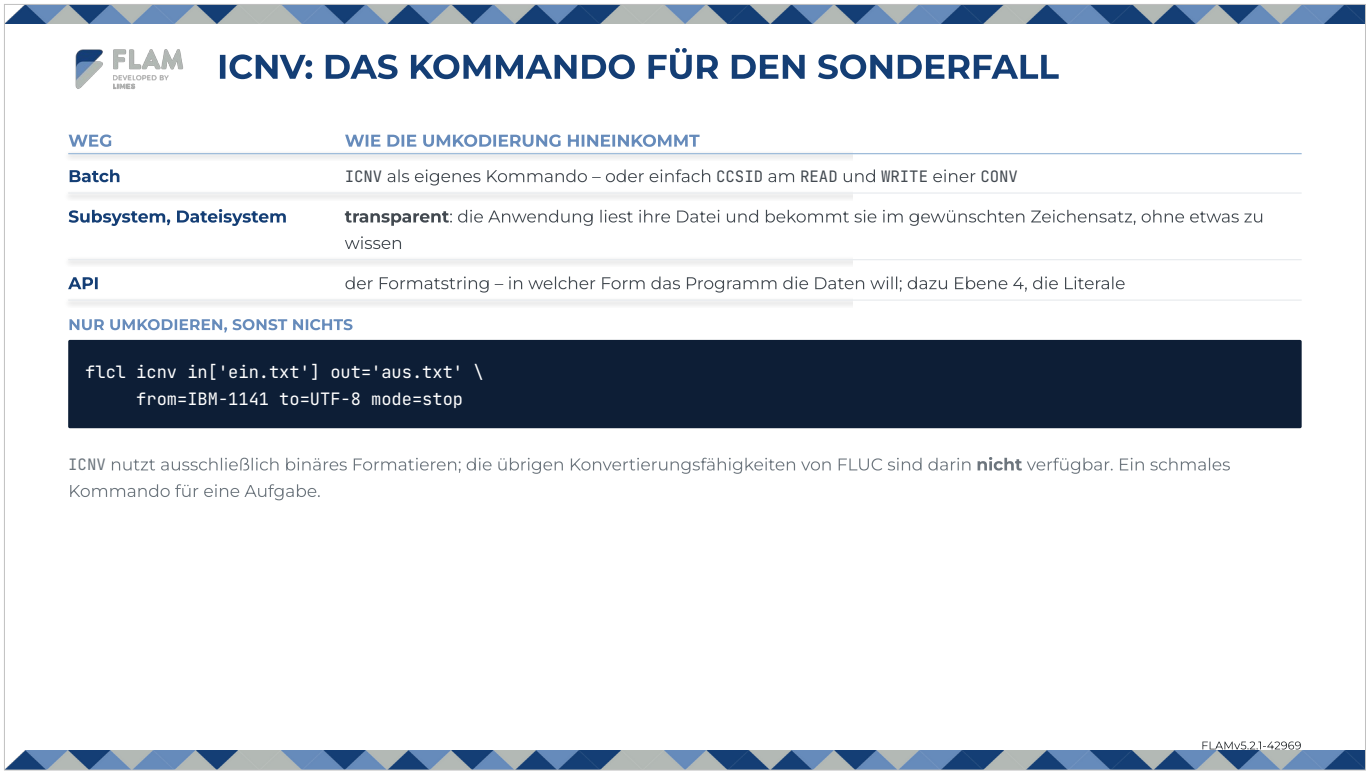
Dieselbe Frage eine Ebene weiter, und hier wird sie praktisch.
Ein Kommando muss nicht in der JCL oder auf der Kommandozeile stehen. Für Objekte, Overlays und Arrays kann man eine **Parameterdatei** angeben, mit `=` oder `=>`, für normale Argumente mit `=>`. Der Inhalt ist ein Kommandostring in derselben Syntax.
Jetzt der entscheidende Punkt: eine Datei trägt ihren Zeichensatz nicht in sich. Solange sie neben dem Programm auf derselben Maschine liegt, ist das egal – sie ist im lokalen Zeichensatz. Aber die Datei kann per URL von **einer anderen Plattform** kommen: `ssh://` holt sie von einem entfernten System, und ein Linux-Server liefert UTF-8 an einen z/OS-Job, der EBCDIC erwartet.
Dafür gibt es die CCSID am Dateinamen. Die URL-Syntax kennt drei Zusätze:
`/?` für einen Membernamen, `/:` oder `/#` für einen Memberindex – und `/&` für die **CCSID, in der der Inhalt der Datei kodiert ist**.
Und dann der Satz aus unserem eigenen Handbuch, den man vorlesen sollte:
"The filename is the only way to specify the CCSID for this type of files." Für Parameterdateien gibt es **keinen anderen Weg**. Es gibt keinen Parameter dafür, denn der Parameter stünde ja in der Datei, die man erst lesen muss. Das ist kein Mangel, das ist die Auflösung eines Henne-Ei-Problems – aber man muss es wissen, sonst sucht man an der falschen Stelle.
Praktisch heißt das: wer Parameterdateien zentral auf einem Server hält und von Host und Unix aus liest, hängt einmal die CCSID an die URL und hat das Thema erledigt.



Die vierte Ebene, und die betrifft jeden, der gegen unsere APIs programmiert – also die Teilnehmer aus der Anwendungsentwicklung.
Die Lage auf dem Host: ein Programm baut sich seinen Kommandostring zusammen. Ein Teil davon steht als **Literal** im Quelltext, ein anderer Teil kommt zur Laufzeit aus einer Variablen – ein Dateiname zum Beispiel.
Und jetzt der Punkt: der Zeichensatz der Literale wird **beim Übersetzen** festgelegt, der der Variablen gilt **zur Laufzeit**. Das können zwei verschiedene sein.
Konkret: C-Quelltext liegt normalerweise in 1047, und wenn nichts anderes eingestellt ist, liegen die Literale auch in 1047. In COBOL ist die Vorbelegung für Literale 1140. Der Dateiname aber, den das Programm bekommt, ist im lokalen Zeichensatz – also zum Beispiel 1141.
Ein Kommandostring, der aus beiden zusammengesetzt ist, ist damit **in zwei Zeichensätzen gleichzeitig**. Kein Parser kann das erraten.
Die Lösung ist die CCSID-Sequenz mitten im String, und das Beispiel steht so in unserem Handbuch: der Literalteil wird mit `&1047;` markiert, vor der Variablen schaltet `&0;` auf die Systemcodepage zurück, danach wieder auf 1047. Jeder Teil des Strings ist dann in dem Zeichensatz, in dem er tatsächlich vorliegt.
Und für den Fall, dass man nicht neu übersetzen kann – alter Code, fremdes Programm: dann setzt man `CLP_STRING_CCSID` von außen. Das ist genau dafür da.
Ein Detail, das Ruhe gibt: auf ASCII- und UTF-8-Plattformen sind diese Sequenzen wirkungslos. Man kann sie also im gemeinsamen Quelltext stehen lassen und muss den Code nicht nach Plattform verzweigen.



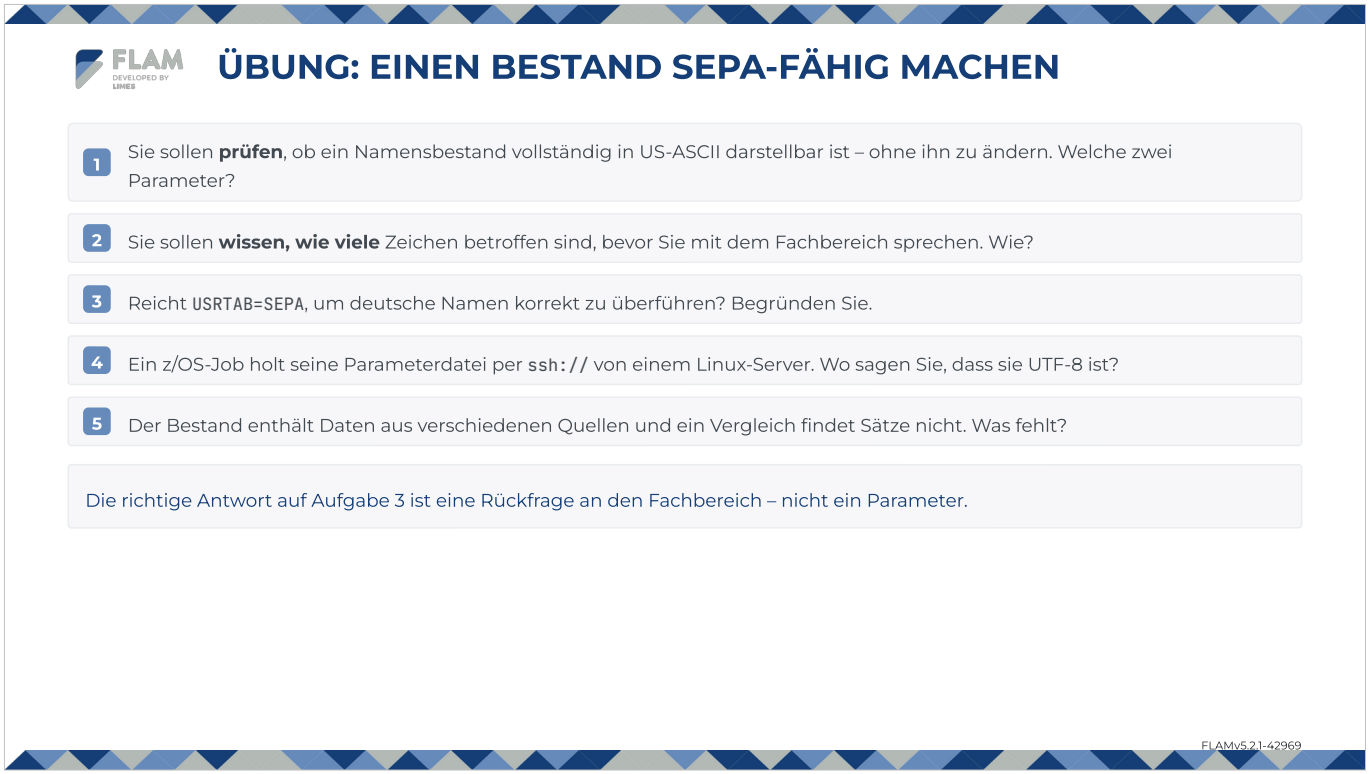
Eine Folie für den Betrieb auf dem Host, und eine, bei der wir eine ehrliche Abwägung liefern statt einer Empfehlung.
Seit Version 5.1.27 kann FLAM auf System Z die **IBM Unicode Services** benutzen statt der eigenen Umsetzung. Das spart CPU und Speicher. Aber nur für **einfache** Wandlungen: sobald eine eigene Tabelle, Transliteration, Casing, ein Subset, Whitespace-Behandlung, Reporting oder Normalisierung im Spiel ist, arbeitet ohnehin unser eigenes Modul. Alles, was in dieser Einheit interessant war, läuft also weiter bei uns.
Gesteuert wird es über `FL_IBM_UNICODE_SERVICE`, auf z/OS auch als Systemsymbol `&FLIBMUS`. Drei Werte:
`AUTO` ist die Vorbelegung und der Kompromiss: die Services werden benutzt, wo sie sicher sind.
`ON` nimmt sie auch für `MODE=STOP` über eine Round-Trip-Technik. Das ist die schnellste Einstellung, und wir empfehlen sie ausdrücklich, **wenn** die Verhaltensunterschiede akzeptabel sind.
`OFF` schaltet sie ab und kostet mehr CPU.
Die Unterschiede, und die muss man nennen, weil sie in der Praxis auffallen:
Ein unvollständiges Zeichen liefert bei den Services einen Fehler, wo unser Modul es ignorieren oder ersetzen würde.
Bei UTF-8 akzeptieren die Services nur die **kürzeste** Form einer Kodierung. Unser Modul dekodiert erst und prüft dann den Codepunkt – es akzeptiert also auch überlange Formen.
Beim BOM hängt das Verhalten von der APAR-Version ab.
Und bei **gemischter Byte-Reihenfolge** in einer Datei sollte man die Services abschalten – ein Wechsel von LE nach BE mitten in den Daten wird nicht erkannt.
Der Satz für den Raum: das ist eine Einstellung, die man **einmal bewusst trifft** und dann dokumentiert. Nicht eine, die man ausprobiert.



Zur Einordnung, denn diese Einheit hat viel über `ICNV` gesprochen: die Zeichensatzwandlung ist **normalerweise Teil einer Konvertierung**, nicht ein eigener Lauf. Wer eine Datei wandelt und dabei umkodiert, gibt einfach `CCSID` am `READ` und am `WRITE` an – fertig.
`ICNV` ist für den Fall, dass man **nur** umkodieren will. Es läuft als Teil des Lesevorgangs von FLUC, arbeitet blockorientiert – mit `RECORD` satzorientiert – und nutzt ausschließlich binäres Formatieren. Alle anderen Konvertierungsfähigkeiten von FLUC sind in diesem Kommando **nicht** verfügbar. Das ist Absicht: ein schmales Kommando für eine Aufgabe.
Und die Einordnung in die drei Wege, weil sie hier besonders klar ist:
Im **Batch** ist `ICNV` das Kommando, oder `CCSID` an einer `CONV`-Anweisung.
Im **Subsystem und Dateisystem** passiert die Umkodierung transparent – die Anwendung liest ihre Datei und bekommt sie im gewünschten Zeichensatz, ohne etwas zu wissen. Das ist der Weg für Standardsoftware.
Über die **API** ist es der Formatstring: man sagt, in welcher Form das Programm die Daten will – und beachtet dabei Ebene 4. Der Satz zum Schluss: **die Wahl des Zeichensatzes ist dieselbe Entscheidung, egal über welchen Weg.** Nur die Verpackung unterscheidet sich – wie bei allem in FLAM.



Fünf, und die erste hat mich selbst erwischt.
Zur ersten: ich habe `to=ASCII` gesetzt und mich gewundert, dass die Umlaute bleiben. Im Log stand "Character conversion to = UTF-8" – und ich hielt es erst für einen Defekt. `ASCII` ist ein Familien-Schlüsselwort, und auf einer UTF-8-Linux-Box löst es zu UTF-8 auf. Das Werkzeug tat das Richtige.
Zur zweiten: die Erkennung hängt am Länderkennzeichen der Maschine. Zwei Maschinen, zwei Ergebnisse, und im Test fällt es nicht auf, weil dort dieselbe Einstellung gilt.
Zur dritten: das SEPA-Subset ist ein Ein-zu-eins-Mapping. Für Namen und Adressen ist das Ergebnis nicht die deutsche Konvention.
Zur vierten: wer `STOP` verlässt, sollte `REPORT` einschalten. Sonst verschwindet die Information, dass überhaupt etwas ersetzt wurde.
Zur fünften, und das ist die Ebenen-Falle: wenn auf dem Host ein Kommando nicht angenommen wird, obwohl es richtig aussieht, liegt es fast nie an den Daten. Dann ist es Ebene 2 – eine Klammer oder eine Tilde, die auf dieser Maschine woanders liegt. Wer zuerst die Daten prüft, sucht am falschen Ende.



Zwanzig Minuten, und die Aufgaben sind so gewählt, dass drei davon die Ebenen auseinanderhalten verlangen.
Die Lösungen:
Aufgabe 1. `MODE=STOP` zusammen mit `USRTAB=SEPA`. Der Lauf bricht beim ersten Zeichen ab, das nicht in US-ASCII liegt – ohne eine Zeile Prüfcode.
Und hier bitte genau bleiben, wenn die Frage aus dem Raum kommt: das ist die Prüfung auf den **Zeichenvorrat**, nicht auf die Feldregeln eines Zahlungsformats. Raute, At-Zeichen und Tilde sind US-ASCII und kommen durch.
Aufgabe 2. Mit `MODE=SUBSTITUTE` oder `TRANSLIT` und `REPORT` auf eine Datei. Dann läuft es durch und man bekommt die Liste der betroffenen Zeichen. Das ist die Zahl, mit der man in das Gespräch mit dem Fachbereich geht.
Aufgabe 3. Nein – und das ist der Kern. Das SEPA-Subset macht `Muller` und `Strase`. Wer `Mueller` und `Strasse` braucht, braucht eine eigene Tabelle über `USRTAB` mit Dateinamen. Die Frage, welche Variante fachlich richtig ist, kann die IT nicht entscheiden: `Muller` ist ein anderer Name, und ob der Empfänger das akzeptiert, steht in seiner Spezifikation.
Aufgabe 4. Die CCSID an den Dateinamen hängen: `/&1141` an der `ssh://`-URL. Der Dateiname ist für Parameterdateien der **einzige** Weg – es gibt keinen Parameter dafür.
Aufgabe 5. Beim Einlesen auf `NFC` normalisieren. Sonst liegen zwei Darstellungen desselben Zeichens im Bestand und der Vergleich schlägt fehl – besonders bei Daten, die über macOS gelaufen sind.

WAS BLEIBT

- **vier Ebenen**: die Daten, der Kommandostring, die Parameterdatei, die Literale im Programm – die erste Frage ist, **welche**
- **186 Codepages**, zwei Drittel davon Einbyte; `ASCII` und `EBCDIC` sind **Familien-Schlüsselwörter**, keine Codepages
- ohne Angabe: **lokal beim Schreiben, erkannt beim Lesen** – und die Erkennung hängt an der Maschineneinstellung
- fünf Fehlerverhalten; `STOP` ist das einzige, bei dem Sie erfahren, dass die Daten ein Problem haben
- **Transliteration** setzt Markierungszeichen und kennt Regeln (`ß` zu `ss`); ein **Subset** ist ein Zeichenvorrat – `SEPA` ist US-ASCII – und mappt ein Zeichen auf ein Zeichen
- `FOLD` für den Vergleich, `NFC` für den Bestand, `REPORT` für den Nachweis, `/&ccsid` für die Parameterdatei

FLAMv5.21-42969

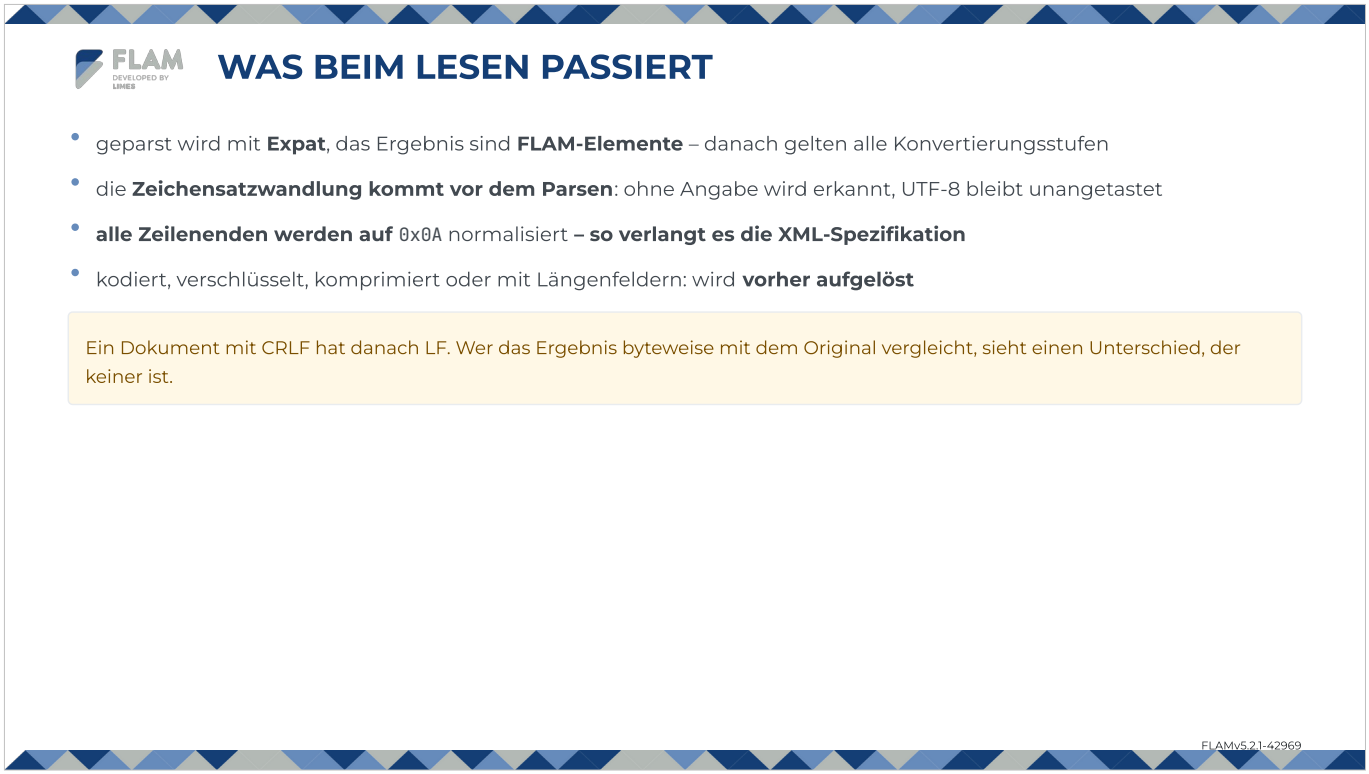
Lernziele durchgehen. Zwei Sätze sollen hängen bleiben: der Zeichensatz steht an vier Stellen, nicht an einer – und die Umkodierung der Daten ist eine fachliche Entscheidung, für die das Werkzeug jeden Schalter hat, die es aber nicht für Sie trifft.



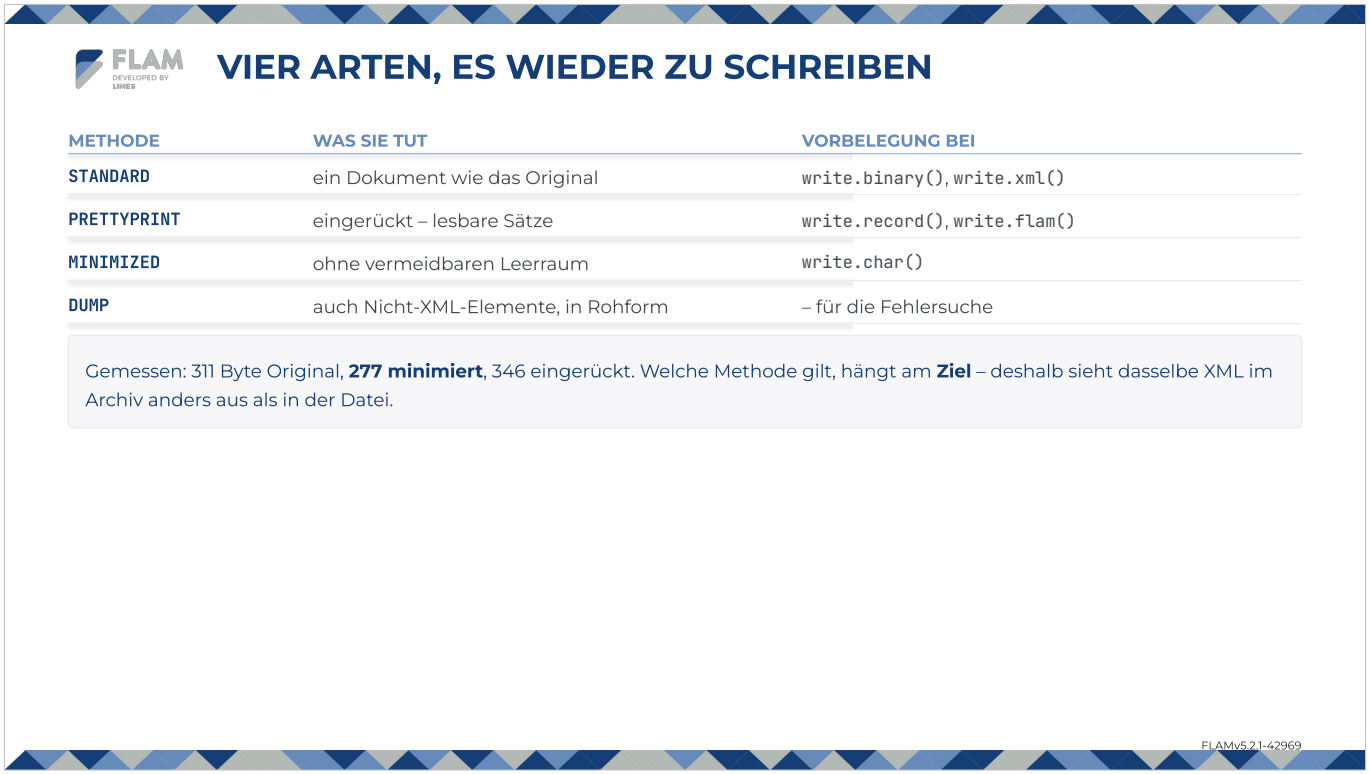
Die Einheit, die zwei Welten verbindet: die hierarchische von XML und die flache von Sätzen und Spalten.
Der Bogen: was FLAM beim Lesen aus XML macht, die vier Arten, es wieder zu schreiben, die Zeichensatzfragen, und dann der eigentliche Grund für diese Einheit – XML als Tabelle, also der Weg nach CSV und in ein Archiv.
Der Satz für den Anfang, und er ist eine Warnung und ein Versprechen zugleich: **XML ist mächtiger als eine Tabelle.** Es geht also nicht jedes XML in eine Tabelle – aber das, was im Zahlungsverkehr vorkommt, geht, und dann in einem Aufruf.



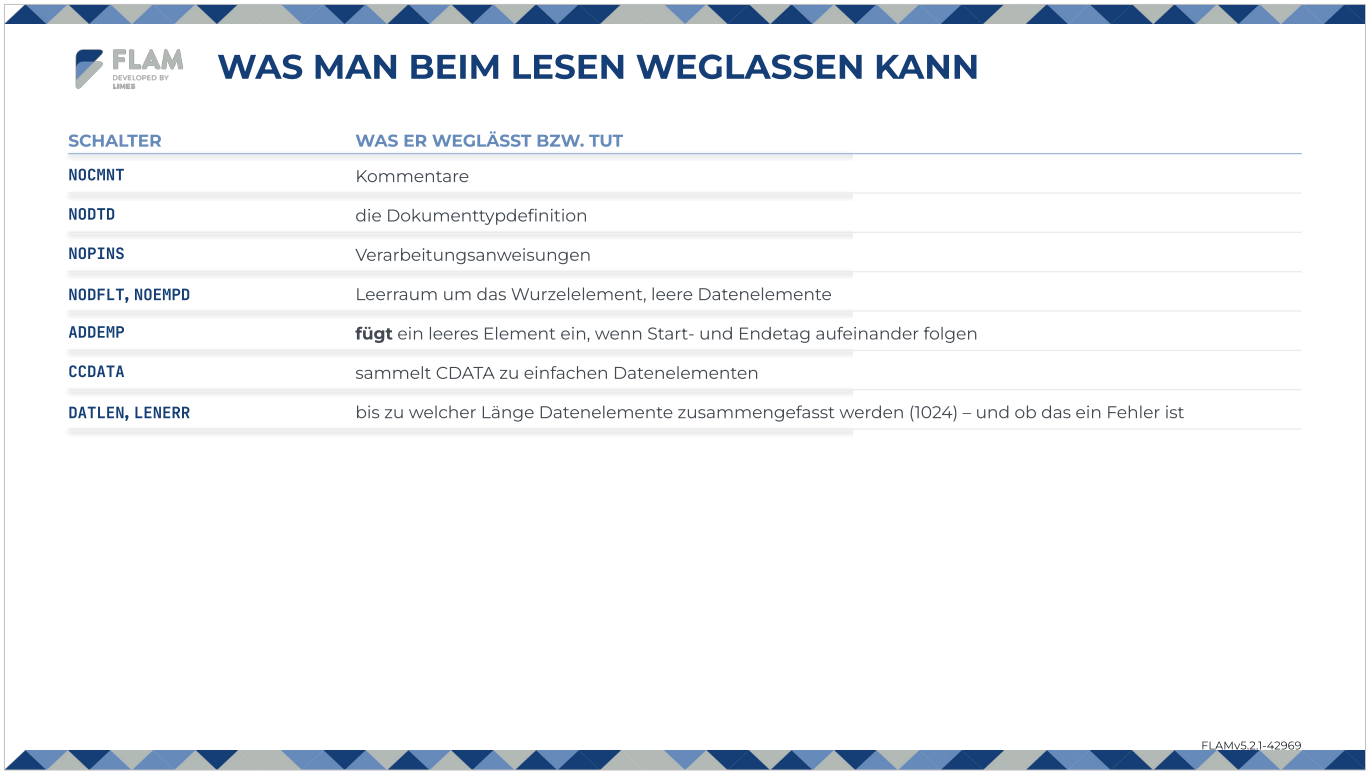
Das vierte Ziel ist das, weswegen die Einheit existiert.
Das fünfte ist das, das ein Projekt vor einer falschen Zusage bewahrt: die Tabellenform hat eine klare Grenze, und die kennt man besser vorher.



Die Folie, die den Mechanismus erklärt – und sie räumt eine Fehlvorstellung aus.
FLAM parst das Dokument mit **Expat** und macht daraus **FLAM-Elemente**.
Das ist der Punkt: es entsteht keine Textkopie und kein DOM im Anwendungssinn, sondern die übliche interne Form – dieselbe, mit der auch Sätze und Spalten arbeiten. Deshalb lassen sich danach alle Konvertierungsstufen anwenden.
Was vorher passiert, ist wichtig für das Ergebnis: **die Zeichensatzwandlung kommt vor dem Parsen.** Ohne Angabe wird die Kodierung erkannt; ist es UTF-8, wird nichts gewandelt. Mit `CCSID` wird von dort nach UTF-8 gewandelt, und die Angabe gewinnt gegen die Deklaration im Dokument.
Und es passiert noch etwas, das man wissen muss: **alle Zeilenenden werden auf 0x0A normalisiert**. Das ist keine Freiheit von FLAM, das verlangt die XML-Spezifikation. Ein Dokument, das mit CRLF hereinkommt, hat danach LF – wer byteweise vergleicht, findet einen Unterschied, der keiner ist.
Und die Kette vorher gilt auch hier: ist die Eingabe kodiert, verschlüsselt, komprimiert oder mit Längenfeldern versehen, wird das automatisch aufgelöst, bevor geparst wird.



Die Folie, nach der niemand mehr fragt, warum das Ergebnis anders aussieht als die Eingabe.
`STANDARD` schreibt ein Dokument, das dem Original entspricht – die Vorbelegung bei `write.binary()` und `write.xml()`.
`PRETTYPRINT` rückt ein. Das ist die Vorbelegung bei `write.record()` und `write.flam()`, und das hat einen Grund: in einem FLAM-Archiv ist XML satzorientiert, und ein eingerücktes Dokument hat Sätze, die man lesen kann.
`MINIMIZED` wirft jeden vermeidbaren Leerraum weg – die Vorbelegung bei `write.char()`. Für die Übertragung ist das die richtige Wahl.
`DUMP` schreibt auch Nicht-XML-Elemente in ein XML-Dokument, in Rohform.
Das ist das Werkzeug für die Fehlersuche.
Gemessen an einer kleinen Zahlungsdatei: 311 Byte im Original werden 277 Byte minimiert und 346 Byte eingerückt. Der Unterschied ist bei echten Dokumenten größer, weil dort mehr Struktur und mehr Einrückung ist.
Der Satz, der dazugehört: **welche Methode gilt, hängt am Ziel.** Wer das nicht weiß, wundert sich, dass dasselbe XML im Archiv anders aussieht als in der Datei.



Eine Nachschlagefolie mit einem praktischen Kern.
Ein XML-Dokument enthält vieles, was für die Verarbeitung nicht gebraucht wird, und jedes Stück davon kostet Speicher und Zeit. Die Schalter sagen, was man nicht sehen will: `NOCMNT` die Kommentare, `NODTD` die Dokumenttypdefinition, `NOPINS` die Verarbeitungsanweisungen, `NODFLT` den Leerraum vor und nach dem Wurzelelement, `NOEMPD` leere Datenelemente.
`ADDEMP` ist die Gegenrichtung: es fügt ein leeres Datenelement ein, wenn ein Endetag direkt auf ein Starttag folgt. Das braucht man, wenn die Weiterverarbeitung für jedes Tag einen Wert erwartet – sonst fehlt die Spalte.
`CCDATA` sammelt CDATA-Abschnitte zu einfachen Datenelementen, damit man sie wie normalen Text behandeln kann.
Und `DATLEN` mit `LENERR` ist die Stelle für große Textknoten: `DATLEN` sagt, bis zu welcher Länge Datenelemente zusammengefasst werden – die Vorbelegung ist 1024 –, und `LENERR` macht daraus einen Fehler statt einer stillen Teilung.
Der Rat: beim Lesen für die Tabellenwandlung ruhig **alles** wegwerfen, was keine Daten ist. Die Tabellenwandlung ignoriert Formatierung, Kommentare und DTD ohnehin.

XML ALS TABELLE: ROOT UND PATH

EIN AUFRUF: XML NACH CSV

```
# row spec in a file of its own -- ROW=> reads it
NAME='zahlung'
COLUMN(NAME='id'   FORMAT.XML(ROOT='<zahlungen/zahlung>'
                              PATH='<@id>')  TYPE.STRING())
COLUMN(NAME='iban' FORMAT.XML(ROOT='<zahlungen/zahlung>'
                              PATH='<iban>') TYPE.STRING())


flcl XCNV "input(sav.fil(fio.blk(file='zahlung.xml')
                         fmt.tab(format=XML row=>'zrow.txt')))
           output(sav.fil(fmt.tab(format=CSV)
                         fio.blk(file='zahlung.csv')))"


# measured result:
# "id","iban","betrag","währung"
# "1","DE02120300000000202051","1234.56","EUR"
```

FLAMv5.21-42969

Die Kernfolie der Einheit, und das Beispiel ist ein echter Lauf.
Zwei Angaben genügen: `**ROOT**` ist der Pfad zu einer Zeile, `**PATH**` der Pfad innerhalb der Zeile. Beide stehen in spitzen Klammern, beide sind groß/klein-empfindlich, und die Tiefe darf bis zu 64 Ebenen gehen.
Ein Attribut erreicht man mit einem `@` vor dem letzten Element – und weil das Klammeraffen-Zeichen in EBCDIC je Codepage anders liegt, geht auch ein `&`. Das ist dieselbe Vorsorge wie in der SSH-URL.
Im Beispiel: `ROOT='<zahlungen/zahlung>'` macht jedes `zahlung`-Element zu einer Zeile, und die vier Spalten holen die Attribut- und Textknoten. Die Satzbeschreibung steht in einer eigenen Datei und wird mit `ROW=>` eingelesen – genau der Griff aus der Sprach-Einheit, und hier ist er besonders nützlich, weil eine Satzbeschreibung lang wird.
Gelaufen ist das mit `XCNV`: am Eingang `FMT.TAB(FORMAT=XML)`, am Ausgang `FMT.TAB(FORMAT=CSV)`. Heraus kommen die drei Zeilen unten, mit Kopfzeile.
Der Satz für den Raum: **das ist ein Aufruf.** Kein XSLT, kein Programm, keine Zwischendatei. Und weil es eine Konvertierungskette ist, kann am Ausgang genauso ein FLAM-Archiv stehen wie eine CSV-Datei.



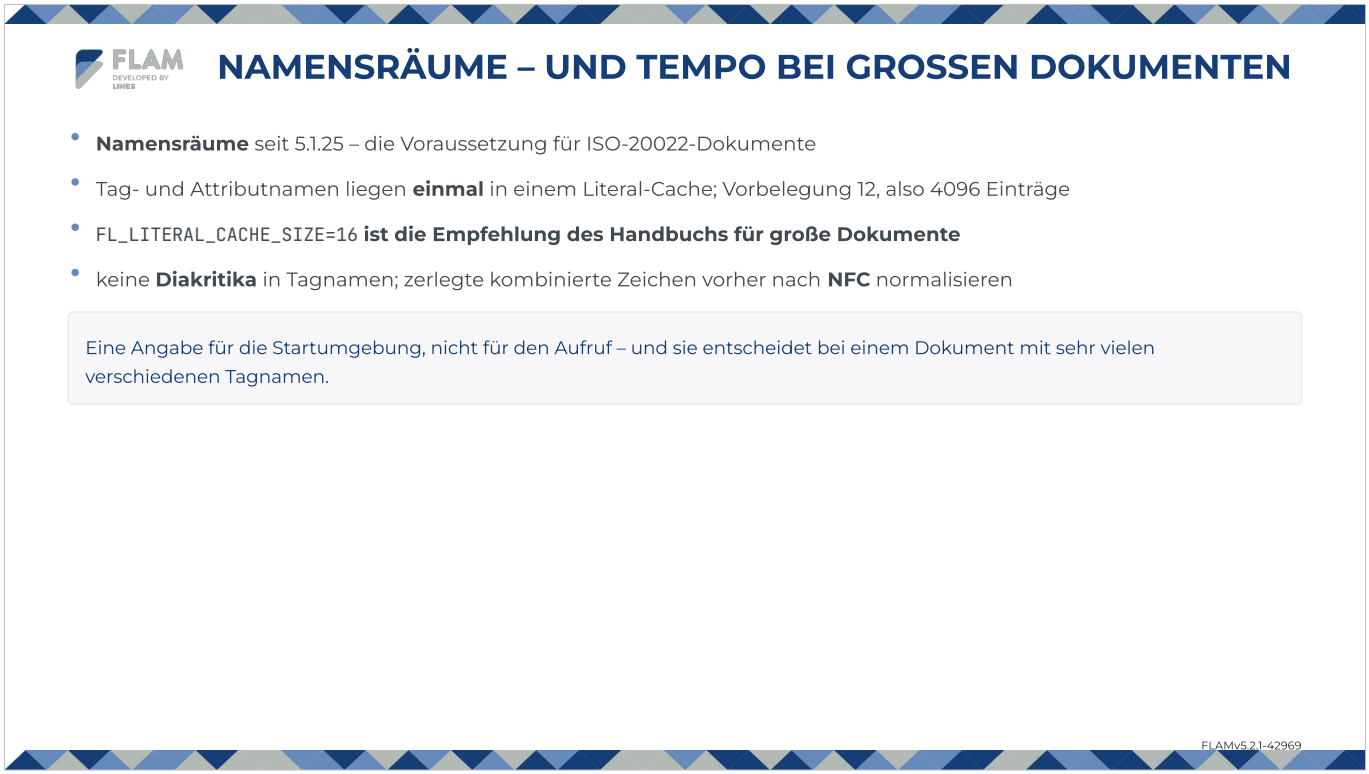
Die Folie, die ein Projekt vor einer falschen Zusage bewahrt. Sie steht absichtlich direkt hinter dem Erfolgsbeispiel.
Unser Handbuch sagt es selbst: XML ist ein sehr mächtiges Datenformat, und es wird hier in eine **weniger mächtige** Form übersetzt. Nicht jedes XML lässt sich in eine Tabelle wandeln.
Die Regel, an der man es vorher erkennt: die Tabellenform verträgt genau **eine unbegrenzte Wiederholung**. Jede dieser Wiederholungen wird eine Zeile, und ihr Beginn ist der Wurzelpfad. Kommt darunter eine **zweite** unbegrenzte Wiederholung, geht es nicht.
Weitere Arrays sind möglich, aber nur mit einer **festen Obergrenze**: man gibt n Spalten mit demselben Pfad an, und damit ist die Zahl der Einträge gedeckelt. Eine unbegrenzte Zahl von Spalten je Zeile gibt es nicht.
Und eine Zahl, die man kennen muss: die Vorbelegung für die maximale Datenlänge einer Spalte ist **1024 Byte**. Wer längere Texte hat, setzt `MAXLEN` – sonst ist der Wert abgeschnitten.
Der praktische Rat: bei einem neuen Format zuerst die Struktur ansehen und die unbegrenzten Wiederholungen zählen. Das ist eine Frage von fünf Minuten und entscheidet, ob der Weg über die Tabelle führt oder über einzelne Dokumente.

ZWEI SCHEMATA

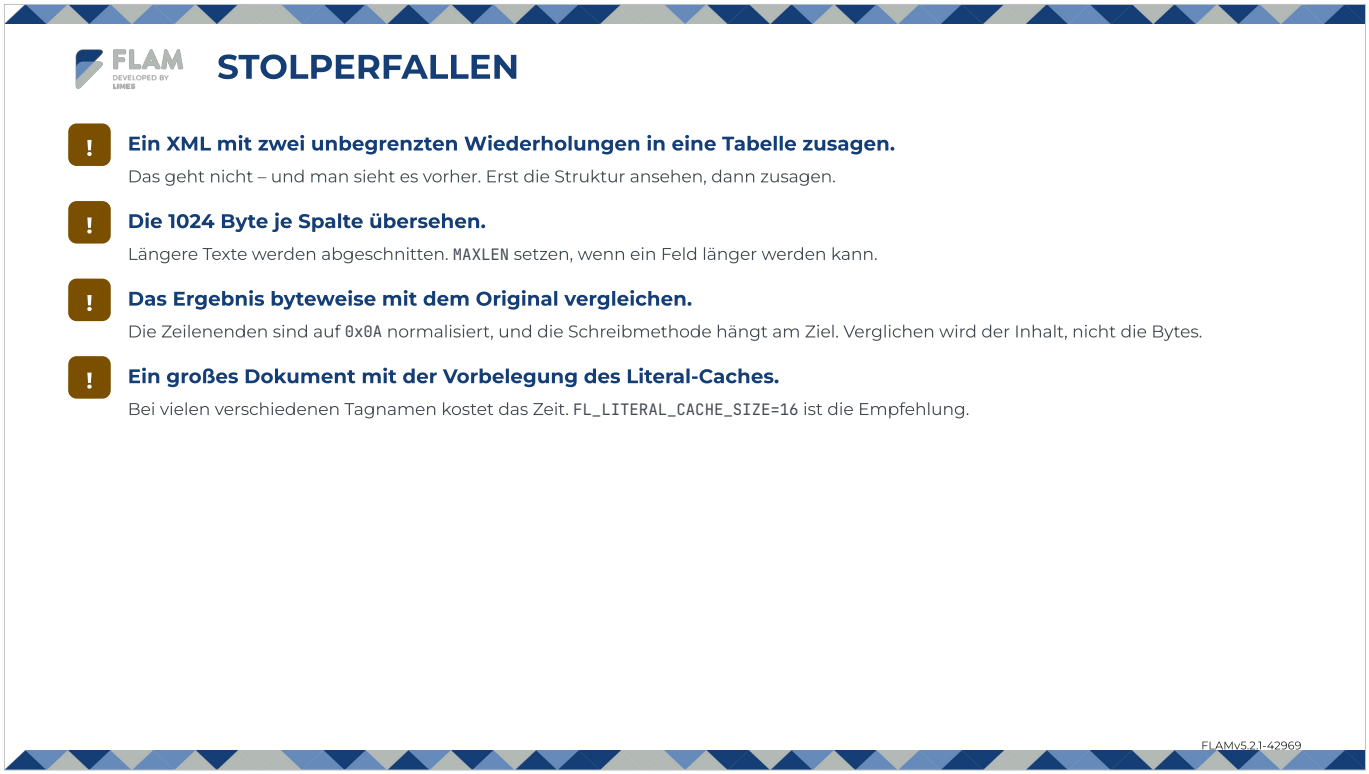
	ZEILE BEGINNT	WAS MAN DAFÜR BEKOMMT
strukturiert (Vorbelegung)	mit einem umschließenden Tag	Vorbelegungen für fehlende Einträge, Kopfdaten, optionale Zweige
unstrukturiert	wenn eine Spalte **wiederkehrt**	beliebige Spaltenreihenfolge – aber **keine optionalen Spalten**

FLAMv5.21-42969

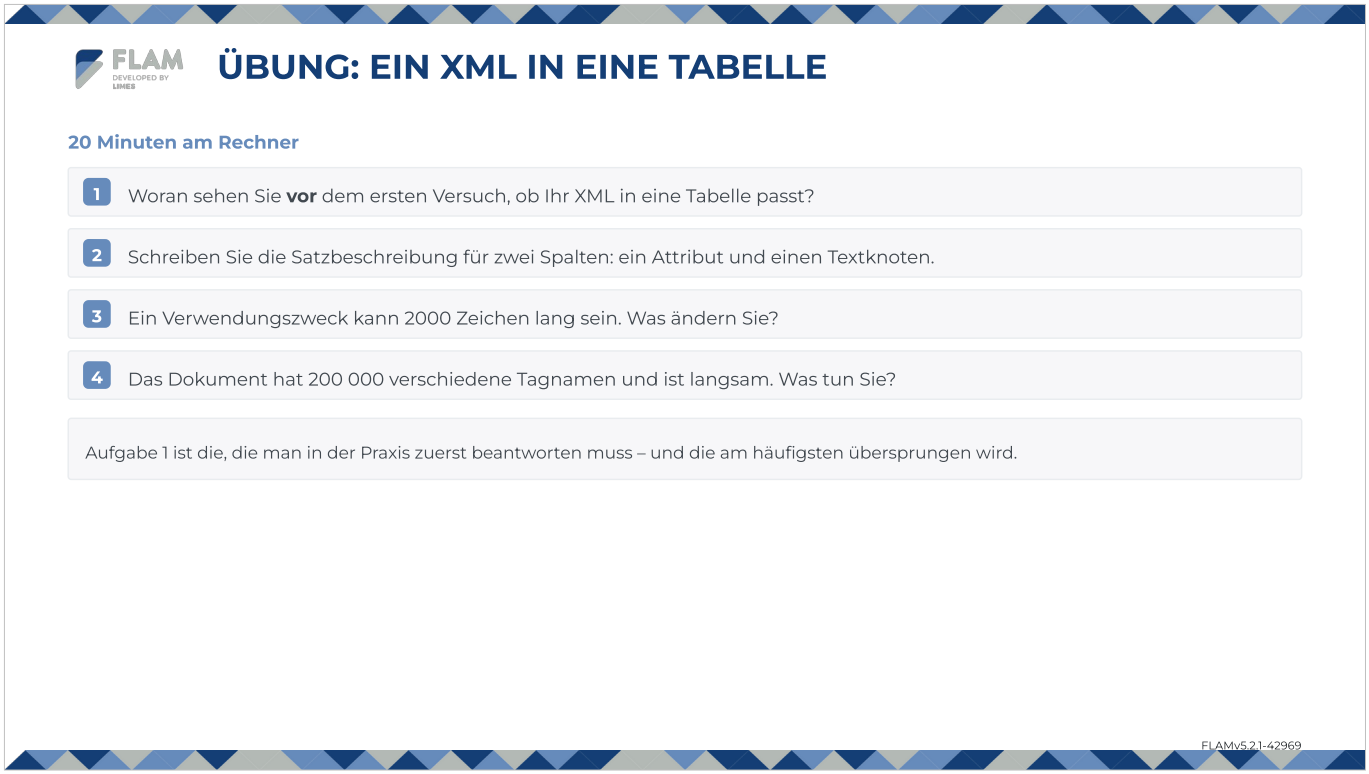
Die Folie für den Fall, dass das Dokument nicht so gebaut ist, wie man es erwartet.
Das **strukturierte** Schema ist die Vorbelegung: jede Zeile ist von einem Tag umschlossen. Das ist der Normalfall, und nur so gibt es Vorbelegungen für fehlende Einträge, Kopfdaten, optionale Zweige und die Prüfung darauf.
Das **unstrukturierte** Schema braucht keine umschließende Klammer: eine neue Zeile beginnt, wenn eine Spalte auftritt, die für die laufende Zeile schon gelesen wurde. Damit ist die Reihenfolge der Spalten innerhalb einer Zeile beliebig – aber es gibt **keine optionalen Spalten**: jede Spalte muss in jeder Zeile vorkommen.
Seit 5.1.25 verträgt das strukturierte Schema auch Dokumente mit nur einer Ebene – dann ist das Ganze eine Tabelle mit einer Zeile. Das klingt akademisch, ist aber der Fall, wenn man Kopfdaten eines Dokuments in eine Zeile holen will.
Der Rat: wenn man wählen kann, das strukturierte nehmen. Das unstrukturierte ist für Dokumente, die man nicht ändern kann.



Zwei Dinge, die bei echten Dokumenten auftauchen.
Namensräume werden seit 5.1.25 unterstützt. Das ist die Voraussetzung dafür, dass ein ISO-20022-Dokument überhaupt verarbeitbar ist – dort steht an jedem Tag ein Präfix.
Und mit derselben Änderung kam eine Optimierung, die man kennen muss, weil sie eine Stellschraube hat: Tag- und Attributnamen werden **einmal** in einem Literal-Cache abgelegt und danach wiederverwendet. Das spart Speicher und Zeit – aber nur, wenn die Hashtabelle groß genug ist.
Die Vorbelegung ist 12, also 4096 Einträge. Über die Umgebungsvariable `FL_LITERAL_CACHE_SIZE` lässt sich das zwischen 8 (256) und 20 (1 048 576) verstellen, und unser Handbuch **empfiehlt für große Dokumente den Wert 16**.
Das ist die Art Angabe, die man einmal in die Startumgebung schreibt und danach vergisst – und die den Unterschied macht, wenn ein Dokument hunderttausend verschiedene Tagnamen hat.
Zwei Kleinigkeiten zum Schluss: in Tagnamen keine Diakritika verwenden, und wenn das UTF-8 zerlegte kombinierte Zeichen enthält, vorher nach NFC normalisieren. Die Zeichensatz-Einheit sagt, wie.



Alle vier aus Projekten.



Zwanzig Minuten am Rechner, mit einem eigenen Dokument, wenn eines da ist.
Die Lösungen:
Aufgabe 1. Die unbegrenzten Wiederholungen zählen. Genau eine: es geht.
Zwei: es geht nicht, und man braucht einen anderen Weg – etwa je Wiederholung ein eigenes Dokument.
Aufgabe 2. `ROOT` ist der Pfad zur Zeile, `PATH` der Pfad in der Zeile; ein Attribut mit `@` davor.
Und die Satzbeschreibung in eine eigene Datei, weil sie lang wird.
Aufgabe 3. `MAXLEN` auf die erwartete Länge setzen. Ohne die Angabe sind es 1024 Byte, und der Rest fehlt still.
Aufgabe 4. Nichts am Aufruf – `FL_LITERAL_CACHE_SIZE=16` in die Startumgebung. Das ist eine Betriebsangabe, keine Verarbeitungsangabe.

WAS BLEIBT

- geparst wird mit **Expat**, danach sind es FLAM-Elemente – alle Konvertierungsstufen gelten
- **vier Schreibmethoden**, und die Vorbelegung hängt am **Ziel**
- Zeilenenden werden auf `0x0A` **normalisiert** – die Spezifikation verlangt es
- `ROOT` **ist die Zeile,** `PATH` **die Spalte,** `@` ein Attribut – ein Aufruf nach CSV oder ins Archiv
- die Tabellenform verträgt **eine** unbegrenzte Wiederholung, und **1024 Byte** je Spalte
- für große Dokumente `FL_LITERAL_CACHE_SIZE=16`

FLAMv5.21-42969

Lernziele durchgehen. Der Verweis nach vorn: was mit den Spalten danach passiert – Typen, Prüfungen, CSV-Feinheiten – steht in *Tabellen und CSV*. Und die Formate anderer Hersteller stehen in *Fremdformate und Austausch*.

FLAM SCHULUNG, EBENE 2 – DATEN

Was die Gegenseite lesen kann

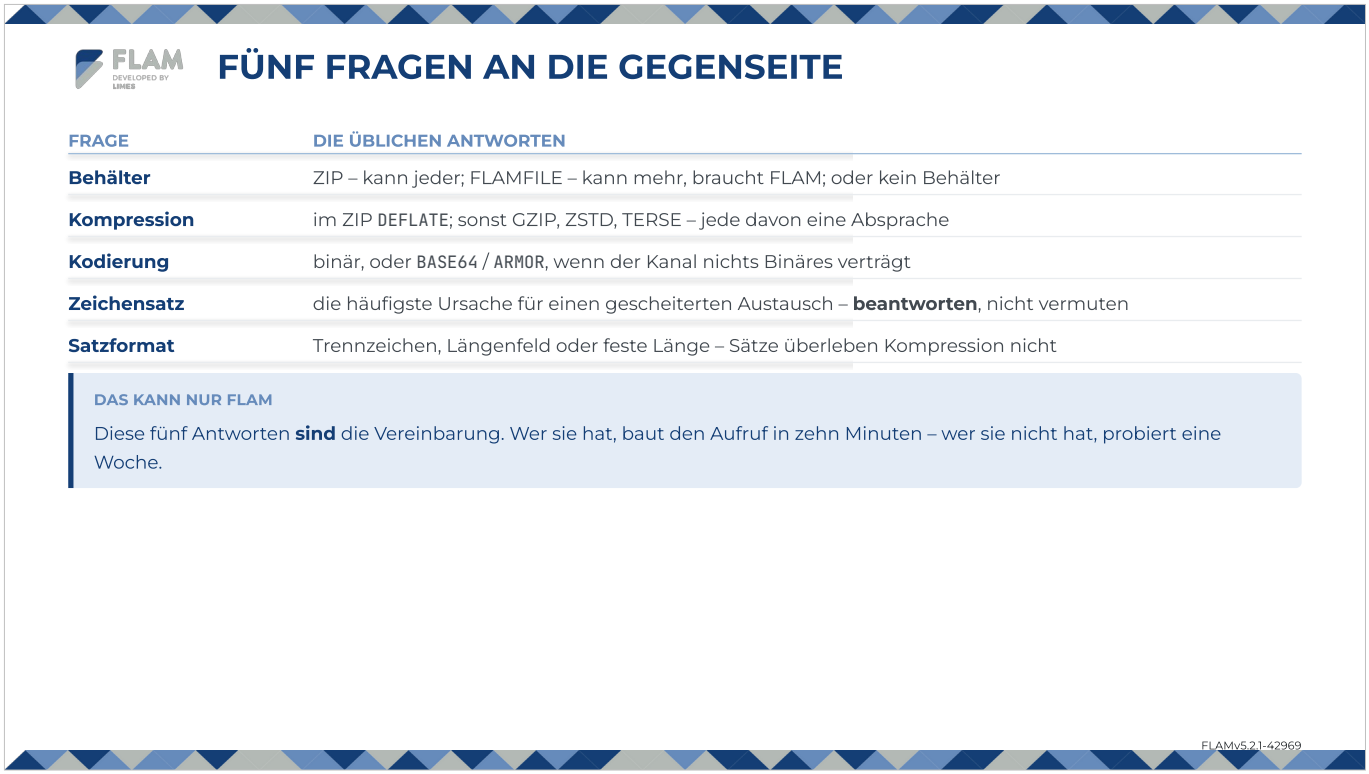
ZIP, Base64, FLAMFILE – und das transparente Lesen

Falk Reichbott · limes datentechnik gmbh

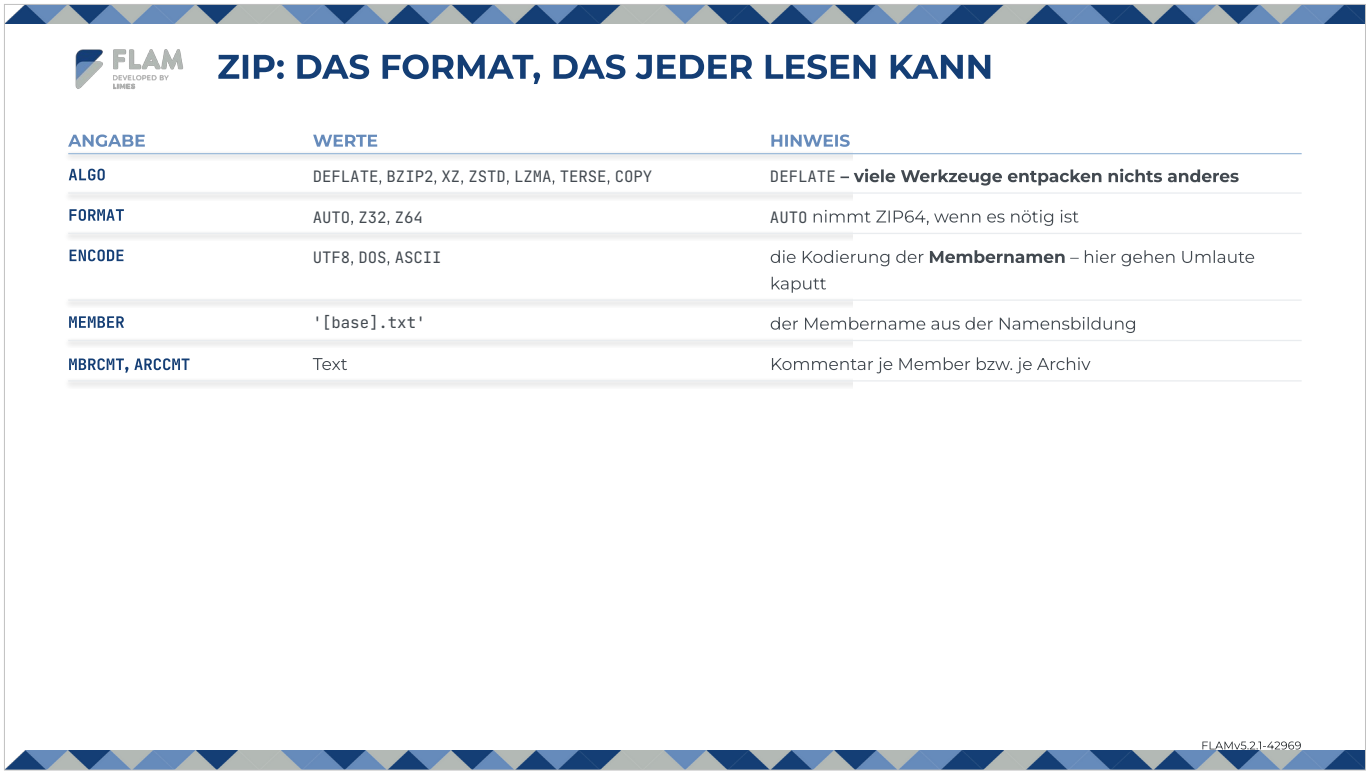
Die letzte Einheit der Gruppe Daten, und die mit dem meisten Bezug zum Alltag: fast jeder Datenaustausch geht an jemanden, der kein FLAM hat.
Der Bogen: die Frage, die man zuerst stellen muss, dann ZIP als das Format, das jeder lesen kann, die FLAM-eigenen Formate, die Kodierungen für Kanäle, die nichts Binäres vertragen, und das transparente Lesen.
Der Satz für den Anfang: **die Frage ist nicht, was wir können, sondern was die Gegenseite lesen kann.** Alles andere ergibt sich daraus – und diese Einheit gibt die Liste, die man mit dem Partner durchgeht.



Das erste Ziel ist die Checkliste, die man in ein Partnergespräch mitnimmt. Die anderen sind die Antworten darauf.
Das fünfte ist die Warnung: transparentes Lesen ist bequem und ausdrücklich nicht verlässlich – das steht so in unserer Handbuchseite.



Die Folie, die man ausdruckt und ins Gespräch mitnimmt. Jede Zeile hat eine falsche Antwort, die teuer ist.
Der Behälter. ZIP kann jeder. Ein FLAMFILE kann nur, wer FLAM hat – dafür kann es mehr. Ein einzelner Strom braucht keinen Behälter.
Die Kompression. Innerhalb eines ZIP ist DEFLATE die Antwort, die jedes Werkzeug versteht. Alles andere ist eine Absprache.
Die Kodierung. Verträgt der Kanal Binärdaten? E-Mail, Formularfelder und manche Schnittstellen nicht – dann Base64 oder ASCII-Armor, und man zahlt ein Drittel mehr Platz.
Der Zeichensatz. Die Frage, an der die meisten Austausche scheitern, und sie hat eine eigene Einheit. Hier zählt nur: sie muss **beantwortet** sein, nicht vermutet.
Das Satzformat. Sätze überleben weder Kompression noch Verschlüsselung. Wenn die Gegenseite Sätze braucht, muss vereinbart sein, wie sie abgegrenzt sind – Trennzeichen, Längenfeld oder feste Länge.
Der Satz für den Raum: **diese fünf Antworten sind die Vereinbarung.** Wer sie hat, baut den Aufruf in zehn Minuten. Wer sie nicht hat, probiert eine Woche.



Die Folie zum Arbeitspferd des Austauschs.
Ein ZIP entsteht als Teil der Kette: `ARCHIVE.ZIP()` am `WRITE`, und der Membername kommt aus der Namensbildung – `member='[base].txt'` schreibt jede Eingabedatei unter ihrem Basisnamen hinein. Gemessen: ein Directory-Walk über drei Dateien ergibt ein ZIP mit `f1.txt` bis `f3.txt`, lesbar mit dem gewöhnlichen `unzip`.
Die Angaben, die man wirklich braucht:
`ALGO` – `**DEFLATE` ist die Vorbelegung und die richtige Wahl**, weil viele Werkzeuge nichts anderes entpacken. `BZIP2`, `XZ`, `ZSTD`, `LZMA` und `TERSE` gehen auch, aber dann muss die Gegenseite mitspielen. `COPY` legt unkomprimiert ab.
`FORMAT` entscheidet über ZIP32 oder ZIP64. `AUTO` wählt, was nötig ist – ZIP64 ab vier Gigabyte oder 65 535 Membern.
`ENCODE` bestimmt die **Kodierung der Membernamen**: `UTF8`, `DOS` oder `ASCII`. Das ist die Stelle, an der Umlaute in Dateinamen kaputtgehen, und sie wird fast immer übersehen.
Und Kommentare: `MBRCMT` je Member, `ARCCMT` für das Archiv. FLAM schreibt von sich aus einen Membervermerk mit Version – man sieht ihn in der `unzip`-Liste.

DIE ZIP-FALLE: ZWEIMAL KOMPRIMIERT

DIE KOMPRESSION GEHÖRT IN DAS ZIP-OBJEKT

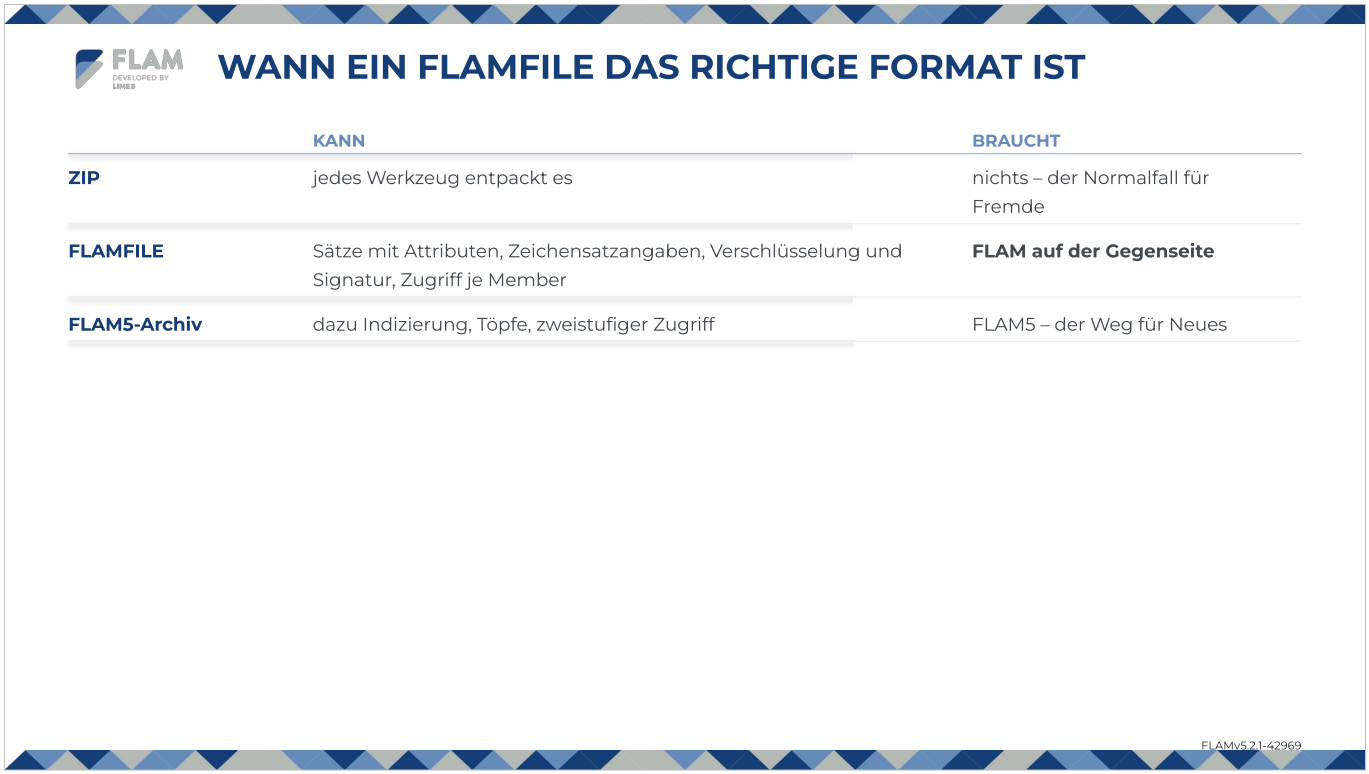
```
# right: the member is compressed the way a ZIP expects it
write.text(file='aus.zip' archive.zip(member='[base].txt'
                                      algo=DEFLATE))


# wrong: a .gz inside a ZIP that says 'stored'
write.text(file='aus.zip' comp.gzip()
           archive.zip(member='[base].txt'))
```

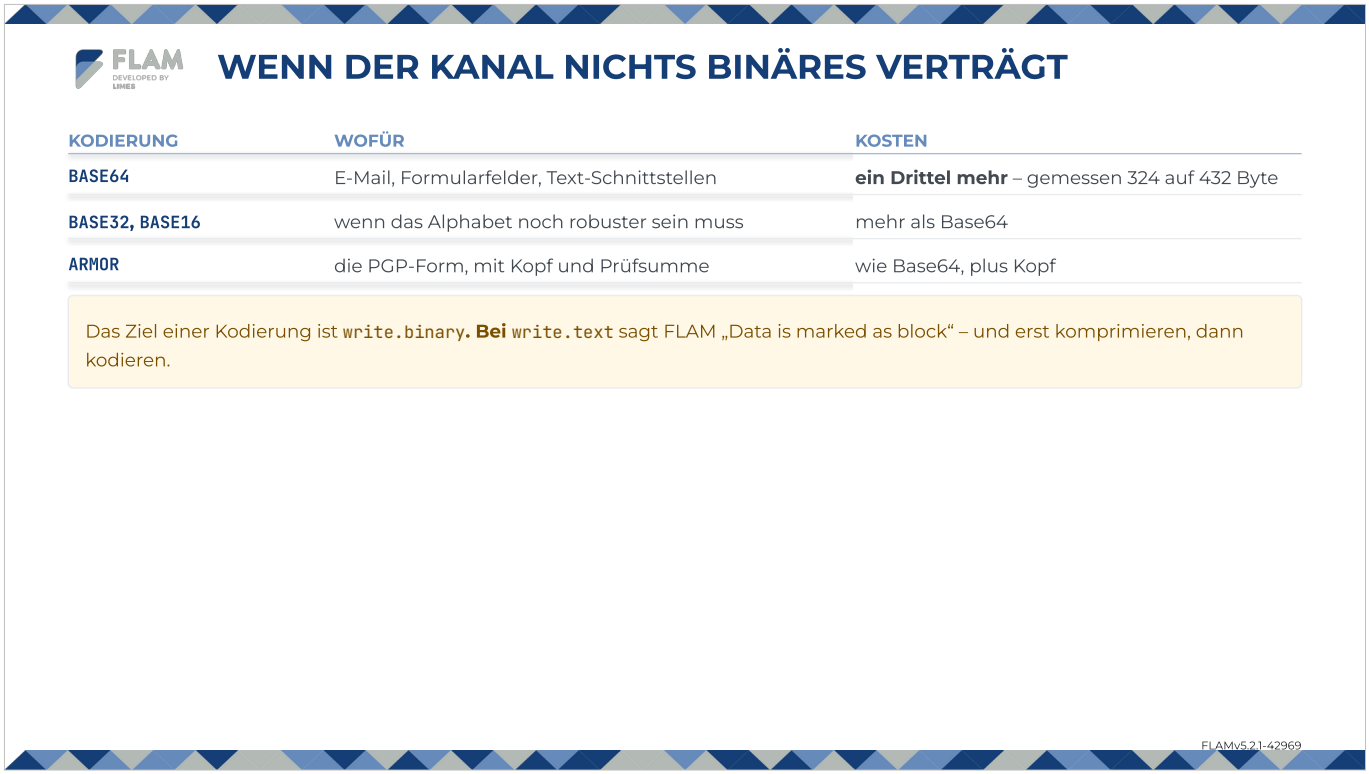
Beim Lesen werden GZIP-, BZIP2- und XZ-Member automatisch entpackt – mit SKPORG lässt sich das abschalten.

FLAMv5.21-42969

Eine kurze Folie über einen Fehler, der ein ZIP unbrauchbar macht, ohne einen Fehler zu melden. Der Unterschied ist feiner, als er klingt: `**ARCHIVE.ZIP(ALGO=...)**` **komprimiert die Member so, wie ein ZIP es vorsieht.** Ein anderes Werkzeug entpackt sie damit ganz normal.
Nimmt man statt dessen eine **eigene Kompressionsstufe** in der Kette – also `COMP.GZIP()` und danach das ZIP –, dann liegt im ZIP eine Datei, die aus ZIP-Sicht **unkomprimiert** ist und in Wahrheit ein GZIP-Strom.
Das steht so in unserer Handbuchseite. Das Ergebnis: das ZIP lässt sich öffnen, der Member lässt sich entpacken – und was herauskommt, ist ein `.gz`, das niemand erwartet hat.
Die Gegenrichtung ist freundlicher: beim **Lesen** aus einem ZIP werden Member, die mit GZIP, BZIP2 oder XZ komprimiert sind, automatisch entpackt. Wer das nicht will, setzt `SKPORG`.
Der Merksatz: **die Kompression eines ZIP gehört in das ZIP-Objekt**, nicht in die Kette davor.



Die Folie, die die eigene Welt einordnet – ehrlich, mit beiden Seiten.
Ein **FLAMFILE** ist das FLAM4-Austauschformat und über `ARCHIVE.FL4()` bzw. `FLAM4()` erreichbar. Es kann Dinge, die ein ZIP nicht kann:
satzorientierte Daten mit ihren Attributen, Zeichensatzangaben, Verschlüsselung und Signatur in einem Behälter, und den Zugriff auf einzelne Member ohne Auspacken.
Und es hat eine Bedingung, die man aussprechen muss: **die Gegenseite braucht FLAM.** Damit ist es das richtige Format für den Austausch innerhalb eines Hauses, zwischen zwei Standorten oder mit einem Dienstleister, der ebenfalls FLAM hat – und das falsche für einen beliebigen Empfänger.
Für neue Vorhaben gilt der Hinweis aus der Archiv-Einheit: das **FLAM5-Archiv** ist der Nachfolger, mit Indizierung, Töpfen und dem zweistufigen Zugriff. Ein FLAMFILE nimmt man, wenn die Gegenseite bei FLAM4 ist.
Der Satz für den Raum: die Wahl ist keine Geschmacksfrage, sondern eine Frage danach, was auf der anderen Seite installiert ist.



Die Folie für die Fälle, in denen man Daten durch ein Nadelöhr schiebt.
Es gibt Kanäle, die kein Binär vertragen: eine E-Mail, ein Formularfeld, eine Schnittstelle, die Text erwartet, oder ein Weg mit Zeichensatzwandlung. Dafür gibt es die Kodierungen: `BASE64` ist die übliche, `BASE32` und `BASE16` sind die Varianten mit noch kleinerem Alphabet, und `ARMOR` ist die PGP-Form mit Kopf und Prüfsumme.
Der Preis ist bekannt und gemessen: **Base64 kostet genau ein Drittel mehr Platz** – aus 324 Byte werden 432. Base32 und Base16 kosten mehr, dafür ist ihr Alphabet noch robuster.
Und eine Kleinigkeit, die Zeit kostet, wenn man sie nicht weiß: das Ziel einer Kodierung ist `write.binary`, nicht `write.text`. Gemessen sagt FLAM sonst "Data is marked as block, text formatting is not possible" – die Meldung ist präzise, aber man muss sie einmal gesehen haben.
Der Rat: erst komprimieren, dann kodieren. Die Kodierung bläht auf, also gehört sie an das Ende der Kette – und dann bläht sie das Komprimat auf, nicht die Rohdaten.

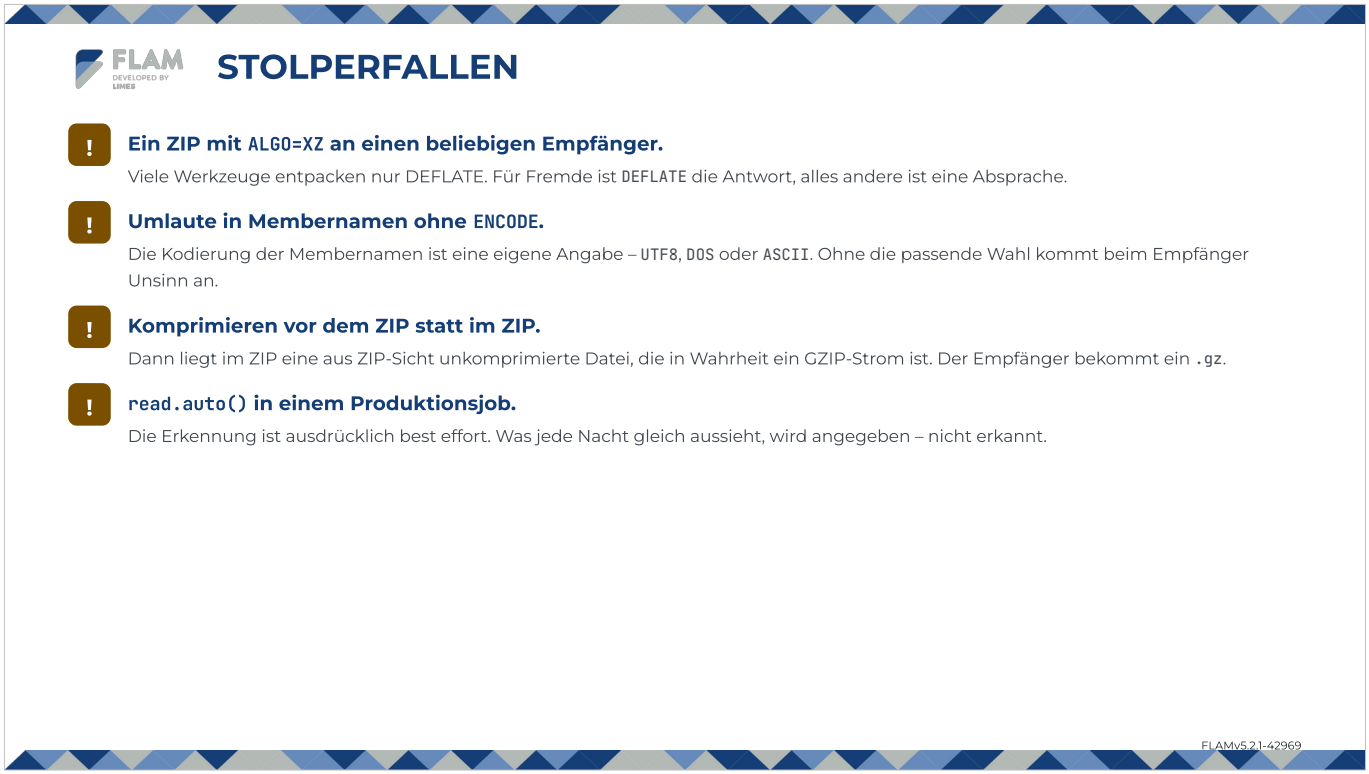
TRANSPARENT LESEN – UND SEINE GRENZE

- erkennt Text mit Trennzeichen, XML, `BASE16/32/64`, verschlüsselt, komprimiert – **auch verschachtelt** –, Sätze, VSAM, FLAMFILEs, Binärdaten
- `read.auto()` `write.record()` wandelt auf dem Host fast jedes Format in Sätze – so arbeiten **FLVIEW und FLVEDIT**
- man kann Satzbeschreibungen mitgeben: passt eine, wird die Tabellenform benutzt
- **„best effort basis and should not be relied on“** – so steht es in unserer Handbuchseite

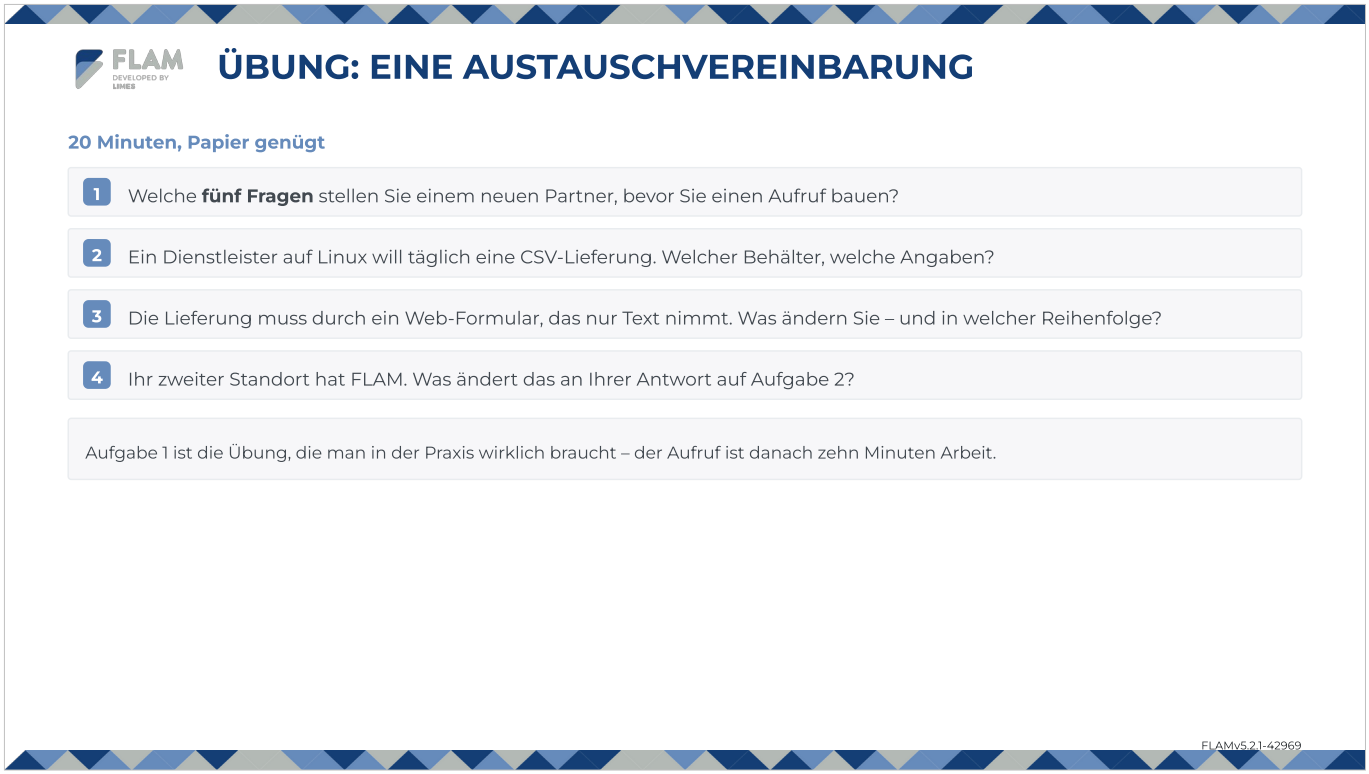
Zum **Hineinsehen** und für den Einzelfall: ja. Im Produktionsjob das Format **angeben** – dann hängt das Ergebnis nicht an den Daten von heute.

FLAMv5.21-42969

Die Folie mit dem bequemsten Werkzeug der ganzen Schulung und der deutlichsten Warnung. `READ.AUTO()` versucht, das Format selbst zu erkennen, und es kann dabei viel: Text mit Trennzeichen, XML, kodierte Daten (Base16/32/64), verschlüsselte oder komprimierte Dateien – **auch verschachtelt** –, satzorientierte Datasets, VSAM, FLAMFILEs und Binärdaten. Damit wandelt `read.auto() write.record()` auf dem Host praktisch jedes unterstützte Format in Sätze; genau das benutzen die ISPF-Kommandos FLVIEW und FLVEDIT.
Und jetzt der Satz aus unserer eigenen Handbuchseite, wörtlich: **"Detection works on a best effort basis and should not be relied on."** Die Erkennung kann scheitern, besonders bei sehr kleinen Dateien.
Daraus folgt die Regel, die man mitgeben muss: `read.auto()` ist das Werkzeug für den **Blick hinein** und für den Einzelfall. Im Produktionsjob, der jede Nacht dasselbe Format liest, gibt man das Format an – dann hängt das Ergebnis nicht daran, was heute erkennbar war.
Ein Zwischenweg steht auch in der Handbuchseite: man kann `read.auto()` eine Liste von Satzbeschreibungen mitgeben. Passt eine, wird die Tabellenform benutzt; passt keine, läuft die normale Erkennung.



Alle vier aus echten Austauschen.



Zwanzig Minuten. Die Übung ist ein Gespräch, nicht ein Aufruf – und genau das ist in der Praxis die Arbeit.
Die Lösungen:
Aufgabe 1. Die fünf Fragen: Behälter, Kompression, Kodierung, Zeichensatz, Satzformat. Wer alle fünf beantwortet hat, hat die Vereinbarung.
Aufgabe 2. ZIP mit `ALGO=DEFLATE`, `FORMAT=AUTO`, `ENCODE=UTF8` für die Membernamen – und im Zweifel ohne Umlaute in den Namen. Die Daten selbst in UTF-8 mit Unix-Zeilenenden, weil der Empfänger auf Linux sitzt.
Aufgabe 3. Base64, und das Ziel ist `write.binary`. Vorher komprimieren, damit das Drittel Aufschlag auf das Komprimat geht und nicht auf die Rohdaten.
Aufgabe 4. Ein FLAMFILE oder ein FLAM5-Archiv – aber nur, weil auf beiden Seiten FLAM steht. Der Gewinn ist der Zugriff je Member und die Verschlüsselung im Behälter; bei einem Fremden wäre es die falsche Antwort.

WAS BLEIBT

- **fünf Fragen** sind die Vereinbarung: Behälter, Kompression, Kodierung, Zeichensatz, Satzformat
- **ZIP mit** `DEFLATE` **für Fremde – und** `ENCODE` für die Membernamen
- die Kompression gehört **in** das ZIP-Objekt, nicht in die Kette davor
- ein **FLAMFILE** kann mehr, braucht aber FLAM auf der Gegenseite – für Neues das FLAM5-Archiv
- `BASE64` kostet **ein Drittel** und gehört an das **Ende** der Kette; Ziel ist `write.binary`
- `READ.AUTO()` ist **best effort** – zum Hineinsehen, nicht für die Produktion

FLAMv5.21-42969

Lernziele durchgehen. Und damit endet die Gruppe Daten: Konvertierung, Tabellen, Zeichensätze, Kompression, Satzformate, XML und der Austausch.
Der Bogen war von der Kette über die Struktur bis zu der Frage, wer es am Ende lesen kann.

EBENE 2 – FUNKTIONSTHEMEN: SICHERHEIT

Krypto und Schlüssel

Die Algorithmen sind der leichte Teil – die Frage ist, wo der Schlüssel herkommt

Erste Einheit der Gruppe Sicherheit. Und der erste Satz sollte die Erwartung zurechtrücken: **die Algorithmen sind der leichte Teil.** AES, SHA, RSA – darüber wird viel gesprochen, und es ist selten das Problem.
Das Problem ist der Schlüssel: wo kommt er her, wer hat ihn, wie wird er gewechselt, und was passiert, wenn er wegkommt. Diese Einheit beantwortet genau das, und zwar in der Reihenfolge, in der ein Auditor fragt.
Was hier **nicht** behandelt wird: die drei Schlüssel eines FLAM5-Archivs und die Töpfe – das steht in den Einheiten *Das FLAM5-Archiv* und *Topfdesign*.
Hier geht es um die Krypto selbst.



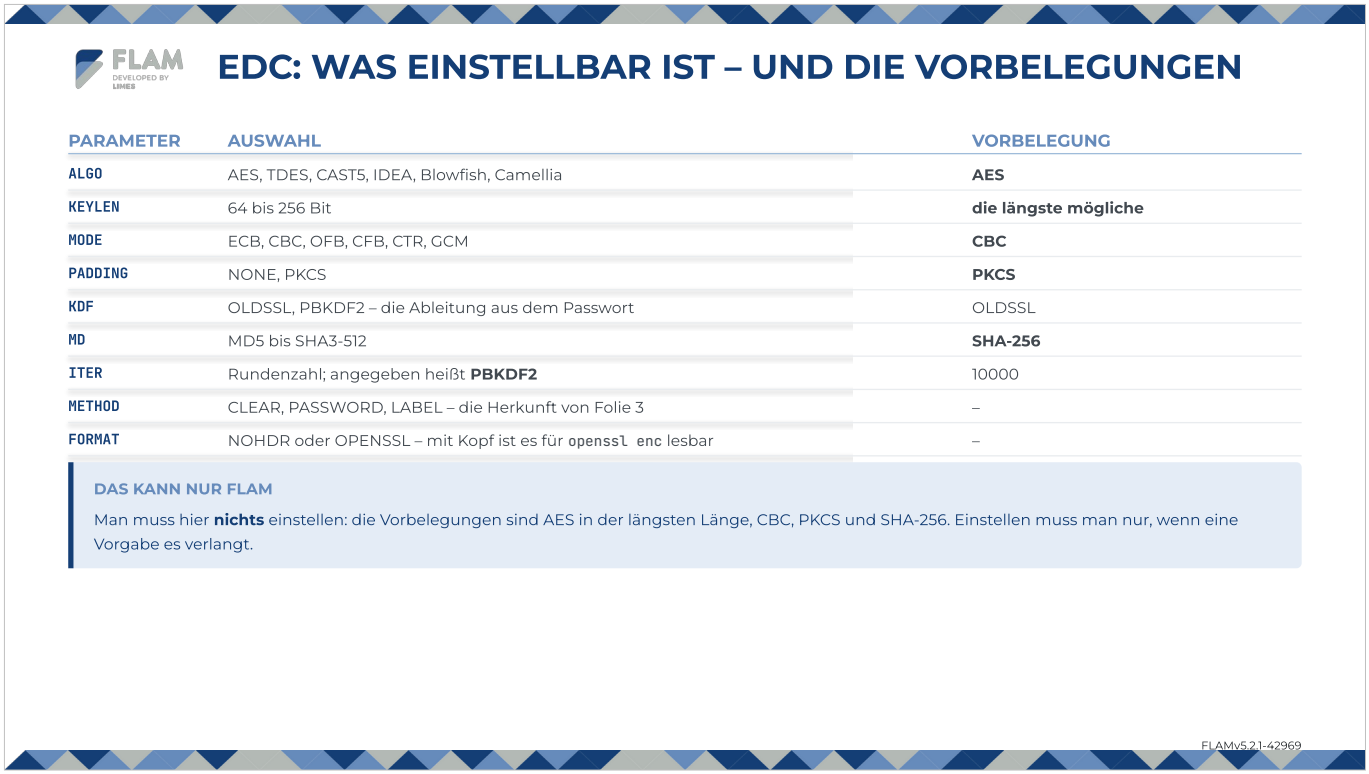
Die Lernziele als Vertrag. Das zweite ist das, worauf die Einheit hinausläuft: ein Passwort im Parameterstring ist der schlechteste von drei Wegen, und die anderen zwei sind da.
Das vierte ist für einen Auditor das wichtigste – er will wissen, dass die Implementierung geprüft ist, nicht dass sie existiert.



Die Landkarte der Einheit, und die Reihenfolge ist eine Rangfolge – von unten nach oben wird es besser.
Die interne Konstante. FLAM kann Daten mit einem eingebauten Schlüssel schützen. Das ist kein Geheimnis und soll keines sein: es schützt gegen Draufschauen, nicht gegen einen Angreifer. Der ehrliche Zweck ist Verschleierung – etwa damit eine Datei nicht versehentlich lesbar ist.
Das Passwort oder die Passphrase. Funktioniert überall, braucht keine Infrastruktur, und hat zwei Schwächen: es steht irgendwo, und es wird selten gewechselt. Wenn Passwort, dann in einer Parameterdatei, nicht im Kommandostring – dazu die Einheit *Sicherheit im Betrieb*.
Die Schlüsselstelle. Das ist der Weg, den wir empfehlen: der Schlüssel liegt in der Infrastruktur, die das Haus schon hat – ein HSM über PKCS#11, IBM CCA oder auf z/OS ICSF -, und FLAM benennt ihn nur. Der Vorteil ist nicht die Hardware, sondern dass **die Prozesse schon existieren**: Erzeugung, Berechtigung, Wechsel, Protokollierung. Das ist der Satz aus unserem Handbuch, und er ist das eigentliche Argument: "no downstream costs caused by encryption (reuse of key management)".
Der Punkt für den Raum: Verschlüsselung scheitert in der Praxis fast nie an der Mathematik. Sie scheitert daran, dass niemand den Schlüsselwechsel organisiert hat.



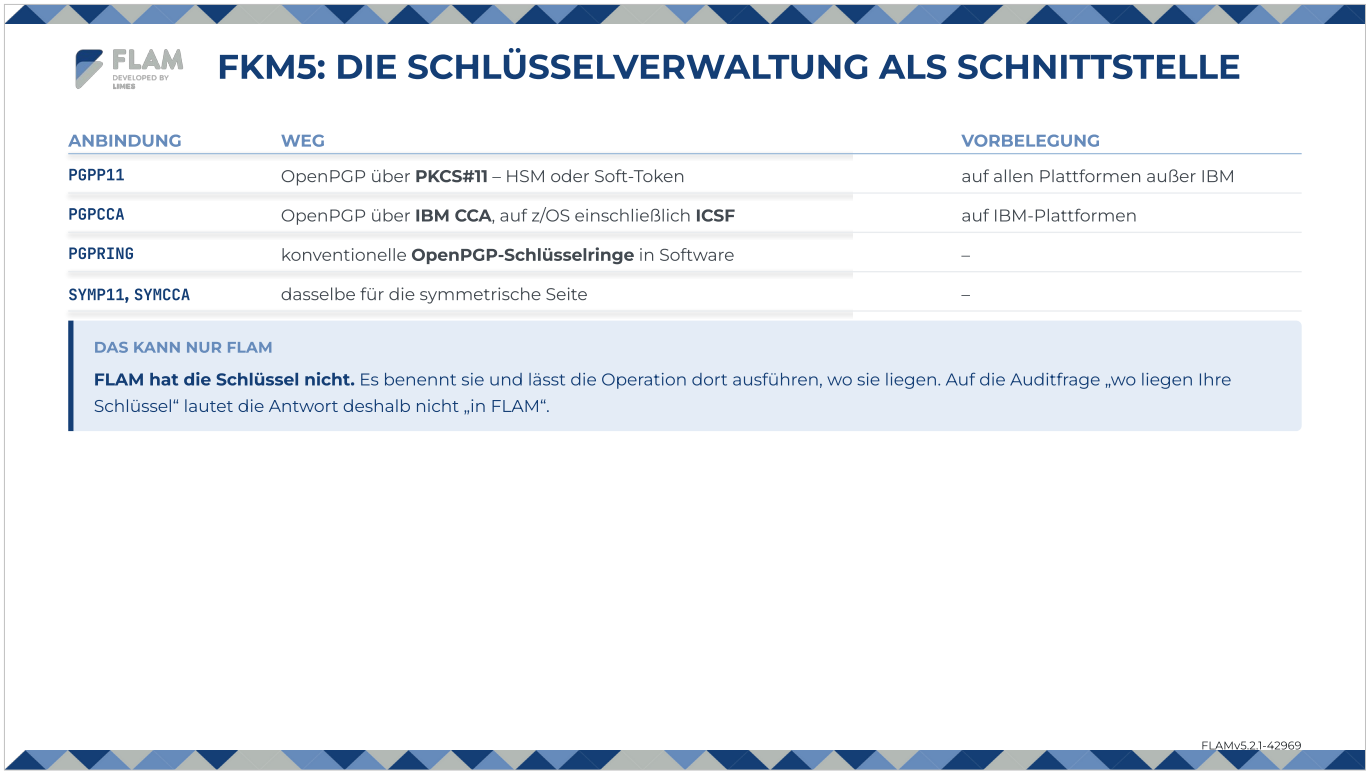
Jetzt die Werkzeuge. An `ENCRYPT` hängen genau drei Objekte, und man wählt nicht nach Geschmack, sondern nach dem Anlass.
`**PGP**` ist OpenPGP nach RFC 4880 – der Weg, wenn Daten **das Haus verlassen**. Der Empfänger braucht nichts von FLAM zu wissen, er braucht nur einen OpenPGP-Client. Dazu gibt es eine eigene Einheit.
`**PWD**` ist die passwortbasierte Verschlüsselung im FLAM-Format. Der Weg, wenn Daten im Haus bleiben und es schnell gehen soll.
`**EDC**` ist die rohe Verschlüsselung – "Raw encryption" sagt die Hilfe.
Hier stellt man alles selbst ein: Verfahren, Schlüssellänge, Betriebsart, Padding, Ableitungsfunktion. Der Anlass ist fast immer eine **Vorgabe von außen**: ein Partner oder ein Standard schreibt genau eine Kombination vor, und man muss sie treffen.
Die Regel dazu: `**EDC**` **nimmt man, wenn man muss, nicht wenn man kann.** Wer alles selbst einstellt, kann alles selbst falsch einstellen – ECB ohne Padding ist syntaktisch erlaubt und kryptografisch ein Fehler.



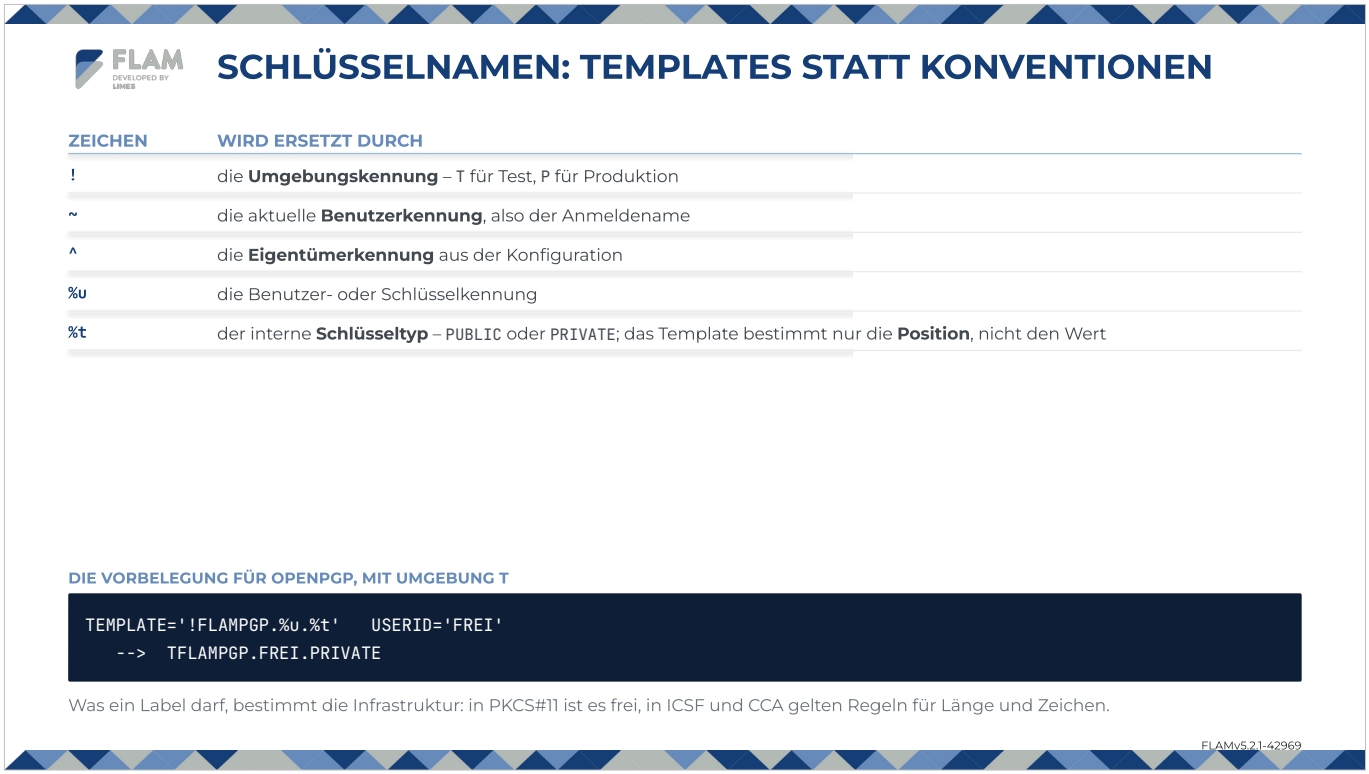
Diese Folie ist die Nachschlagefolie, und die rechte Spalte ist die wichtige: **die Vorbelegungen sind vernünftig.** Wer nichts angibt, bekommt AES in der längsten möglichen Schlüssellänge, CBC, PKCS-Padding und SHA-256 für die Ableitung. Das ist genau die Kombination, die man auch von Hand wählen würde.
Deshalb der Satz, mit dem man die Folie einführen sollte: **man muss hier nichts einstellen.** Man kann, wenn eine Vorgabe es verlangt.
Zu den Werten einzeln, soweit gefragt wird: `METHOD` sagt, woher der Schlüssel kommt – `CLEAR` für die interne Konstante, `PASSWORD` für die Passphrase, `LABEL` für die Schlüsselstelle. Das ist die Folie 3 in einem Parameter.
`FORMAT` entscheidet, ob ein Kopf geschrieben wird: `OPENSSL` erzeugt das Format, das `openssl enc` liest – praktisch, wenn die Gegenseite kein FLAM hat. `NOHDR` schreibt nur das Kryptogramm, dann muss die Gegenseite alle Parameter kennen.
Bei den Algorithmen stehen auch alte drin – TDES, CAST5, IDEA, Blowfish.
Die sind nicht als Empfehlung da, sondern weil ein Partner sie vorschreiben kann. Wer neu baut, nimmt AES.



Die Folie, die einen häufigen Denkfehler ausräumt: ein Passwort **ist** kein Schlüssel. Es muss erst zu einem gemacht werden, und wie das passiert, entscheidet über die Angreifbarkeit.
`KDF=OLDSSL` ist das alte OpenSSL-Verfahren: ein Hash über Passwort und Salz, ein paar Runden. Schnell – und genau das ist das Problem, denn für einen Angreifer ist es auch schnell.
`KDF=PBKDF2` macht daraus eine Rechenaufgabe: `ITER` Runden, standardmäßig zehntausend. Für den berechtigten Benutzer kostet das einmal ein paar Millisekunden. Für einen Angreifer, der Millionen Passwörter durchprobiert, kostet es das Zehntausendfache.
Und der bequeme Teil: **wer `ITER` angibt, bekommt automatisch PBKDF2.** Man muss nicht beides setzen.
Der Satz für den Raum: **ein langes Passwort mit einer schnellen Ableitung ist schlechter als ein kurzes mit einer langsamen.** Wenn im Haus Passwörter benutzt werden, ist `ITER` die eine Angabe, die man nicht weglassen sollte.
Wer die Diskussion ganz vermeiden will, nimmt die Schlüsselstelle – dann gibt es kein Passwort, aus dem etwas abgeleitet werden muss.



Jetzt die Schlüsselstelle, und das ist die Folie, die im Gespräch mit der Sicherheitsabteilung zählt. **FKM5 ist eine Schnittstelle, keine Implementierung.** FLAM bringt Anbindungen mit, aber die Schnittstelle ist offen – ein Haus kann seine eigene Schlüsselverwaltung anbinden, und dann ist FLAM nur noch der Aufrufer.
Mitgeliefert sind drei Wege, und die Namensgebung ist einfach: der Funktionsname sagt, welcher Weg. `PGPP11` ist OpenPGP über PKCS#11, `PGPCCA` ist OpenPGP über IBM CCA einschließlich ICSF auf z/OS, `PGPRING` ist der Softwareweg über konventionelle OpenPGP-Schlüsselringe. Für die symmetrische Seite gibt es `SYMP11` und `SYMCCA`.
Die Vorbelegung ist plattformabhängig und vernünftig: auf IBM-Plattformen `PGPCCA`, sonst `PGPP11`. Wer nichts angibt, bekommt den Weg, der zur Maschine passt.
Der Nutzen, und das ist die Liste aus unserem Handbuch, die man vorlesen kann: vorhandene Infrastruktur nutzen, vorhandene Prozesse für Schlüsselverwaltung und Berechtigung wiederverwenden, Hardwaremodule einsetzen – und **keine Folgekosten durch die Verschlüsselung**, weil die Schlüsselverwaltung schon bezahlt ist. Dazu die Konformität mit PCI DSS.
Ein Satz noch zur Abgrenzung, weil er Missverständnisse vermeidet: FLAM **hat** die Schlüssel nicht. Es benennt sie und lässt die Operation dort ausführen, wo der Schlüssel liegt. Wer im Audit gefragt wird, wo die Schlüssel liegen, antwortet also nicht "in FLAM".



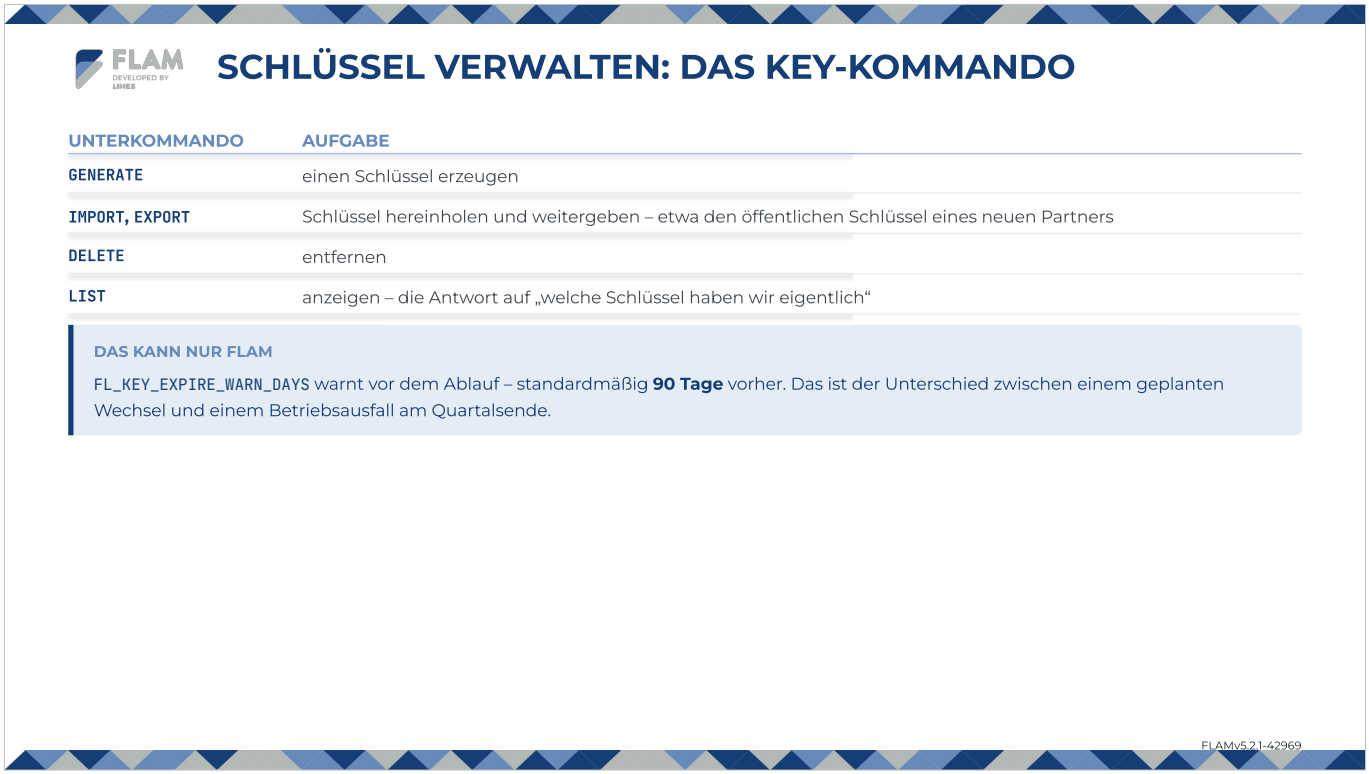
Diese Folie löst ein Problem, von dem die meisten nicht wissen, dass es eines ist – bis sie FLAM in ein Haus mit Namenskonventionen einbauen.
Die Lage: ein Schlüssel in einem HSM wird über einen **Namen** angesprochen, das Key Label. Und jedes Haus hat dafür eine eigene Konvention – mit Umgebung, Eigentümer, Anwendung, Typ, in eigener Reihenfolge.
FLAM löst das nicht mit einer eigenen Konvention, sondern mit einem **Template**. Drei Kürzel für die Umgebung und die Identität:
`!` ist die Umgebungskennung – T für Test, P für Produktion.
`~` ist die aktuelle Benutzerkennung, also der Anmeldename.
`^` ist die Eigentümerkennung aus der Konfiguration.
Dazu zwei Platzhalter, die die Anwendung füllt: `%u` wird die Benutzer- oder Schlüsselkennung, `%t` der interne Schlüsseltyp – PUBLIC oder PRIVATE.
Wichtig für die Erwartung: über das Template bestimmt man nur die **Position** des internen Typs, nicht seinen Wert. Den gibt FLAM vor.
Die Vorbelegung für OpenPGP ist `!FLAMPGP.%u.%t`. Mit Umgebung T und dem Benutzer FREI wird daraus `TFLAMPGP.FREI.PRIVATE`.
Und die Grenze, die man nennen sollte: was ein Label darf, bestimmt die Infrastruktur, nicht FLAM. In PKCS#11 ist ein Label frei wählbar, in ICSF und CCA gibt es Regeln für Länge und Zeichen. Das Template kann also nicht alles – es kann sich nur an die Konvention des Hauses anpassen.



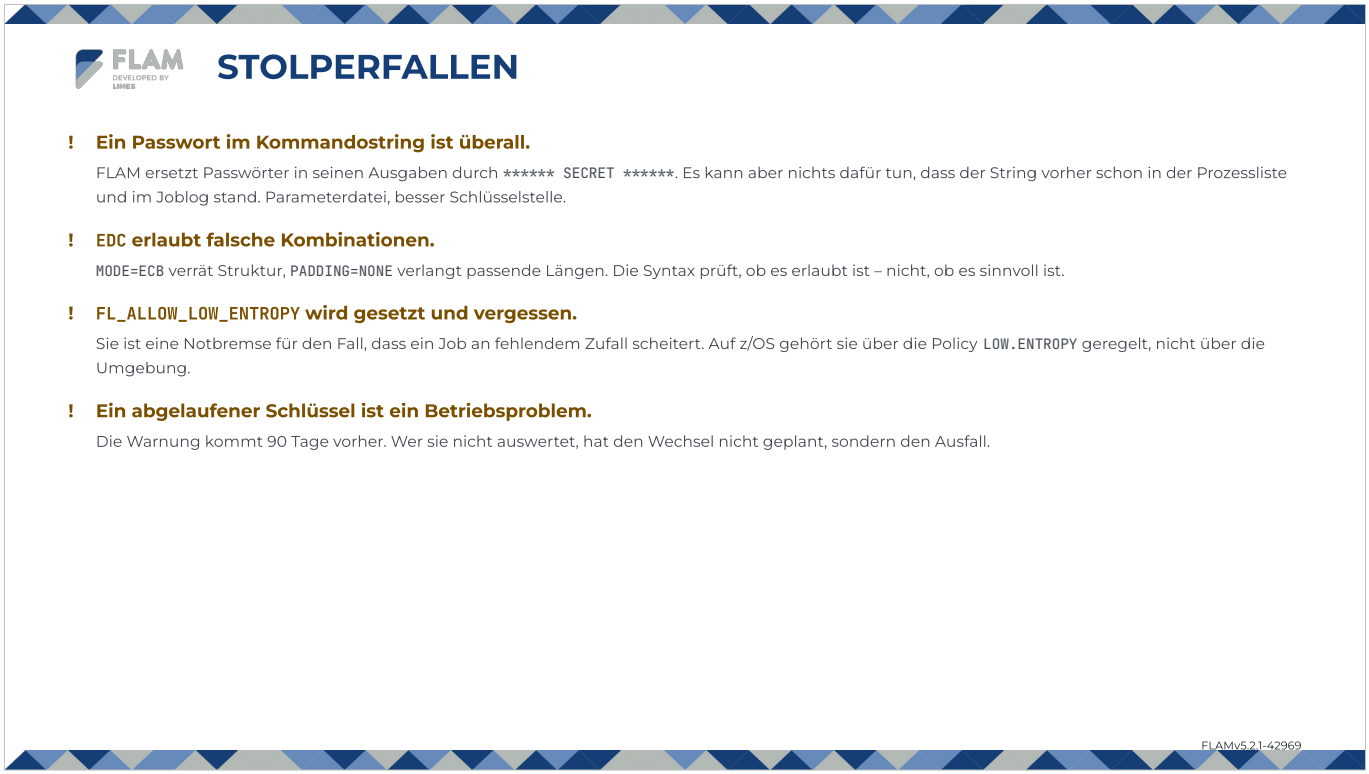
Die Folie für den Auditor, und sie enthält eine Aussage, die in dieser Form selten ist.
Alle Softwarekrypto läuft über **einen** zentralen Kern, CryCore. Der ist ein Singleton und wählt die beste Implementierung für die Maschine, auf der er läuft – mit Hardwarebeschleunigung, wo es sie gibt. Auf IBM-Mainframes sind das im Wesentlichen die CPACF-Funktionen, auf allen anderen Plattformen die `libcrypto` von OpenSSL.
Und jetzt der Punkt: **bevor ein Algorithmus das erste Mal benutzt wird, läuft ein Selbsttest gegen die NIST-Testvektoren.** Es wird also nicht geglaubt, dass die geladene Funktion AES richtig implementiert – es wird geprüft. Das ist der Unterschied zwischen "wir benutzen AES" und "wir haben nachgesehen, dass dieses AES AES ist".
Warum das mehr ist als Sorgfalt: die Implementierung kann je Maschine eine andere sein. Eine falsch erkannte Hardwarefähigkeit oder eine fremde `libcrypto` würde sonst still falsch rechnen – und falsch verschlüsselte Daten merkt man erst beim Entschlüsseln, im schlimmsten Fall Jahre später.
Wenn ein Auditor nach der Krypto-Qualifizierung fragt, ist das die Antwort.
Die Umgebungsvariable `FL_DEACTIVATE_HW_CRYPTO_ACCELERATION` schaltet die Hardware ab und erzwingt die Softwareimplementierung. Das ist ein Diagnosewerkzeug, kein Betriebsschalter.



Ein kleines Thema mit einer Entscheidung darin, die man bewusst treffen sollte.
Ein Schlüssel ist nur so gut wie der Zufall, aus dem er entsteht. FLAM verlangt deshalb standardmäßig **hohe Entropie** – und bricht ab, wenn nur schwacher Zufall verfügbar ist. Das kommt in der Praxis vor: eine frisch gestartete virtuelle Maschine, ein Container ohne Entropiequelle, ein Batchsystem ohne Benutzerinteraktion.
Der Abbruch ist die richtige Vorbelegung, denn ein Schlüssel aus schwachem Zufall ist ein schwacher Schlüssel – und das merkt niemand.
`FL_ALLOW_LOW_ENTROPY` erlaubt den schwachen Fall. Und hier ist die Abwägung, die man selbst treffen muss: es gibt Zusammenhänge, in denen die Qualität des Zufalls kaum zählt – dann ist die Variable bequem. Und es gibt Zusammenhänge, in denen sie alles ist – dann ist sie gefährlich.
Deshalb gibt es dazu auch eine **SAF-Policy** `LOW.ENTROPY`: auf z/OS kann der Sicherheitsverantwortliche verhindern, dass ein Anwender den schwachen Fall einschaltet. Das ist das Muster, das in der Einheit *SAF und RACF* wiederkehrt: eine Umgebungsvariable ist ein Wunsch, eine Policy ist eine Regel.
Und der Nachbar-Punkt, weil er hierher gehört: FLAM **löscht sensible Daten sicher** – Schlüssel, Passwörter, PINs werden so früh wie möglich überschrieben, und das Überschreiben wird vom Optimierer nicht weggeräumt. Das ist eine Kleinigkeit, nach der in Audits regelmäßig gefragt wird.



Kurz, aber es gehört dazu: FLAM kann Schlüssel nicht nur benutzen, sondern auch verwalten – über `flcl KEY` mit fünf Unterkommandos.
`GENERATE` erzeugt, `IMPORT` und `EXPORT` bewegen, `DELETE` entfernt, `LIST` zeigt an. Alles skriptfähig, also automatisierbar.
Der Sinn ist nicht, die Schlüsselverwaltung des Hauses zu ersetzen – sie soll ja gerade wiederverwendet werden. Der Sinn ist, dass man die wiederkehrenden Handgriffe in eine Jobkette packen kann: einen Schlüssel für einen neuen Partner anlegen, einen öffentlichen Schlüssel importieren, ein Inventar ziehen.
Und der Punkt, der im Betrieb zählt: **`LIST` ist die Antwort auf "welche Schlüssel haben wir eigentlich".** Diese Frage kommt in jedem Audit, und ohne ein Inventar ist sie nicht beantwortbar.
Dazu passt `FL_KEY_EXPIRE_WARN_DAYS`: FLAM warnt, bevor ein Schlüssel abläuft – standardmäßig **90 Tage** vorher. Das ist der Unterschied zwischen einem geplanten Wechsel und einem Betriebsausfall am Quartalsende.



Vier, alle aus der Praxis.
Zur ersten: ein Passwort im Kommandostring landet in der Prozessliste und oft im Joblog. FLAM ersetzt es in seinen eigenen Ausgaben, aber es kann nichts dafür tun, dass es vorher schon irgendwo stand.
Zur zweiten: `EDC` erlaubt Kombinationen, die kryptografisch falsch sind.
ECB verrät Struktur, `PADDING=NONE` verlangt passende Längen. Wer `EDC` benutzt, sollte eine Vorgabe haben, an die er sich hält.
Zur dritten: `FL_ALLOW_LOW_ENTROPY` ist eine Notbremse, die man gern setzt, wenn ein Job abbricht – und dann vergisst. Auf z/OS gehört sie über die Policy geregelt.
Zur vierten: ein abgelaufener Schlüssel ist kein Krypto-Problem, sondern ein Betriebsproblem. Die Warnfrist ist da, damit man sie nutzt.



Fünfzehn Minuten. Die Aufgaben sind so gebaut, dass die Antwort jedes Mal eine **Begründung** verlangt, nicht einen Parameter.
Die Lösungen:
Aufgabe 1. Die Schlüsselstelle, also `METHOD=LABEL` mit FKM5 – auf z/OS über ICSF.
Begründung: die Prozesse für Erzeugung, Berechtigung und Wechsel existieren schon, und der Schlüssel verlässt das HSM nicht.
Aufgabe 2. `KDF=PBKDF2` mit einer hohen Rundenzahl, und das Passwort gehört in eine Parameterdatei mit engen Rechten. Besser wäre die Schlüsselstelle – wenn das Argument "wir haben kein HSM" kommt: PKCS#11 gibt es auch als Soft-Token.
Aufgabe 3. Nein. Die interne Konstante ist Verschleierung, kein Schutz.
Für eine Datei, die das Haus verlässt, ist sie ungeeignet – dort gehört OpenPGP hin, weil der Empfänger einen eigenen Schlüssel hat.
Aufgabe 4. Der Selbsttest gegen die NIST-Vektoren vor der ersten Benutzung, der zentrale Kryptokern und – auf z/OS – CPACF. Und die Antwort auf "wo liegen die Schlüssel": nicht in FLAM.

WAS BLEIBT

- drei Herkünfte: **interne Konstante** (Verschleierung), **Passwort** (überall), **Schlüsselstelle** (empfohlen)
- der Gewinn eines HSM sind die **vorhandenen Prozesse** – Verschlüsselung erzeugt dann keine Folgekosten
- `PGP` fürs Haus verlassen, `PWD` fürs Haus, `EDC` nur wenn eine **Vorgabe** es verlangt
- ein Passwort ist kein Schlüssel: `ITER` ist die Angabe, die man nicht weglässt
- ein **Selbsttest gegen die NIST-Vektoren** läuft vor der ersten Benutzung jedes Algorithmus
- **FLAM hat die Schlüssel nicht** – es benennt sie über ein Template und lässt dort rechnen, wo sie liegen

FLAMv5.21-42969

Lernziele durchgehen. Der Satz zum Schluss ist der erste Satz der Einheit, und er soll jetzt belegt sein: die Algorithmen sind der leichte Teil.



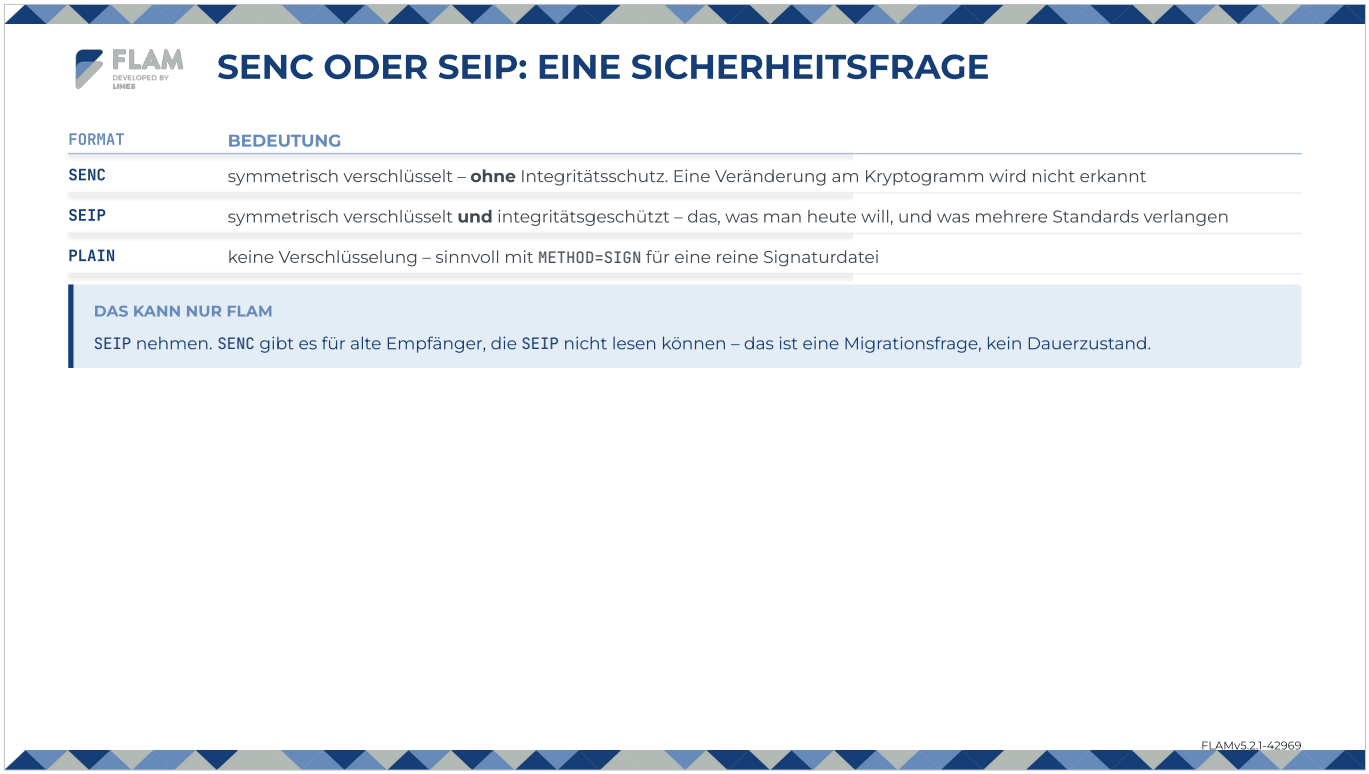
Zweite Einheit der Gruppe Sicherheit, und sie setzt die vorige fort: dort ging es um Schlüssel, hier um das Format, **mit dem Daten das Haus verlassen**.
Der Rahmen in einem Satz: OpenPGP ist der einzige Weg, bei dem der Empfänger nichts von FLAM wissen muss. Er braucht einen OpenPGP-Client, und die gibt es überall. Deshalb ist PGP im Datenaustausch die Voreinstellung, nicht das FLAM-eigene Format.
Und die zweite Hälfte der Einheit, die oft übersehen wird: **Signaturen ohne Verschlüsselung.** Die Frage "sind die Daten unterwegs verändert worden" ist eine andere als "hat sie jemand gelesen", und sie wird viel häufiger gestellt.



Die Lernziele. Das dritte ist das, was diese Einheit von einer Produktvorstellung unterscheidet: auf der **Leseseite** kann man erzwingen, was eine Datei erfüllen muss. Das ist der Unterschied zwischen "wir akzeptieren PGP" und einer Richtlinie.
Das fünfte ist der Punkt, den kaum jemand erwartet – man kann den Sitzungsschlüssel für einen weiteren Empfänger umschlüsseln, ohne die Daten anzufassen.



Die erste Entscheidung, und sie wird oft nicht getroffen, sondern geerbt – jemand hat vor Jahren `ENCSIG` geschrieben, und so läuft es seither.
`METHOD=CRYPT` verschlüsselt nur. Die Frage "wer darf es lesen" ist beantwortet, die Frage "ist es unverändert und von wem" nicht.
`METHOD=SIGN` signiert nur. Der Inhalt bleibt lesbar – und das ist oft genau richtig: eine Preisliste, ein Kursblatt, ein öffentlicher Datensatz muss nicht geheim sein, aber er muss echt sein.
`METHOD=ENCSIG` macht beides. Der Normalfall im Zahlungsverkehr.
Der Satz, den man mitgeben sollte: **Vertraulichkeit und Echtheit sind zwei verschiedene Anforderungen**, und man sollte wissen, welche man gerade erfüllt. Die häufigste Lücke in der Praxis ist eine verschlüsselte Datei ohne Signatur – niemand kann beweisen, von wem sie kommt.
Zum Signieren gehört `SIGHASH`, der Hash über den Inhalt. Die Auswahl reicht von MD5 bis SHA-512; MD5 und SHA-1 stehen drin, weil Partner sie vorschreiben können, nicht weil man sie nehmen soll.
Und `COMPRESS` ist hier ein Wort wert: OpenPGP komprimiert **innerhalb** des Formats. Wer die Daten vorher schon komprimiert hat, sollte hier `COPY` setzen – doppelte Kompression kostet CPU und bringt nichts. `AUTO` ist die bequeme Wahl.



Eine kleine Folie mit einer Aussage, die man kennen muss, weil sie in Standards auftaucht.
`FORMAT` hat drei Werte, und zwei davon verschlüsseln:
`SENC` ist symmetrisch verschlüsselt. Der Inhalt ist geheim – aber es gibt **keinen Integritätsschutz**. Ein Angreifer, der das Kryptogramm verändert, wird nicht dabei erwischt; das Entschlüsseln liefert dann Müll oder, in unglücklichen Fällen, veränderte Daten.
`SEIP` ist symmetrisch verschlüsselt **und integritätsgeschützt**. Das ist das, was man heute will, und mehrere Standards verlangen es ausdrücklich.
`PLAIN` verschlüsselt nicht – nützlich in Verbindung mit `METHOD=SIGN`, wenn eine reine Signaturdatei im PGP-Format entstehen soll.
Der praktische Rat: **SEIP nehmen**, und auf der Leseseite erzwingen, dass eingehende Dateien integritätsgeschützt sind. Dazu die nächste Folie.
Wenn die Frage kommt, warum es `SENC` überhaupt noch gibt: alte Empfänger.
Ein Partner mit einem sehr alten Client kann SEIP nicht lesen. Das ist eine Migrationsfrage, kein Dauerzustand.



Das ist die stärkste Folie der Einheit, und sie beantwortet eine Frage, die in jedem Compliance-Gespräch kommt: **wie stellen Sie sicher, dass eine eingehende Datei die Anforderungen erfüllt?** Die Lage: eine PGP-Datei kann verschlüsselt sein oder nicht, signiert oder nicht, integritätsgeschützt oder nicht. Wer sie einfach entschlüsselt, akzeptiert alles – auch eine Datei, die nur signiert und nicht verschlüsselt war, oder eine ohne Signatur.
Am `DECRYPT.PGP` gibt es dafür drei Schalter, und sie sind der ganze Punkt:
`ENCRYPED` erzwingt einen Fehler, wenn die Datei **nicht verschlüsselt** war.
`INTEGRITY` erzwingt einen Fehler, wenn sie **nicht integritätsgeschützt** war – also SENC statt SEIP.
`SIGNING` erzwingt einen Fehler, wenn sie **nicht signiert** war.
Alle drei sind standardmäßig **aus**. Und das ist die Falle: die Vorbelegung ist "nimm, was kommt". Wer eine Richtlinie hat, muss sie einschalten – dann ist sie nicht mehr eine Absprache, sondern eine technische Bedingung.
Zwei weitere Schalter, damit die Folie vollständig ist: `NOSIGVERIFY` schaltet die Signaturprüfung ab – das will man fast nie, und wenn, dann begründet. `INTERIM` erlaubt Zwischenergebnisse, die noch kein Klartext sind.
Und zur Schreibweise, weil sie auffällt: der Schalter heißt tatsächlich `ENCRYPED`, ohne T. Das ist der Parametername; wir zitieren ihn, wie er ist.



Die Nachschlagefolie für den Aufruf.
`USERID` beziehungsweise `RECEIVER` nennt die Empfänger – und es ist ein **Array**. Mehrere Empfänger heißt: der Sitzungsschlüssel wird für jeden einzeln verschlüsselt, die Daten nur einmal. Das ist der Grund, warum eine Datei für zehn Empfänger nicht zehnmal so groß ist.
`SIGNID` beziehungsweise `SIGNER` nennt, wer signiert.
Die Schlüssel selbst kommen über `FKM5()` – Keyring, PKCS#11 oder CCA, siehe die vorige Einheit. Ohne `FKM5` gilt die plattformabhängige Vorbelegung.
`ARMOR()` schaltet die ASCII-Armor-Kodierung ein: aus dem Binärformat wird Text. Nötig ist das, wenn die Übertragung kein Binär verträgt – eine E-Mail, ein Formularfeld, ein Kanal mit Zeichensatzwandlung. **Es kostet etwa ein Drittel mehr Platz**, also nur einschalten, wenn es gebraucht wird.
Der häufigste Fehler dazu: `ARMOR` steht **innerhalb** von `PGP`, nicht daneben. Das ist die Struktur der Sprache – was zu PGP gehört, steht in PGP.



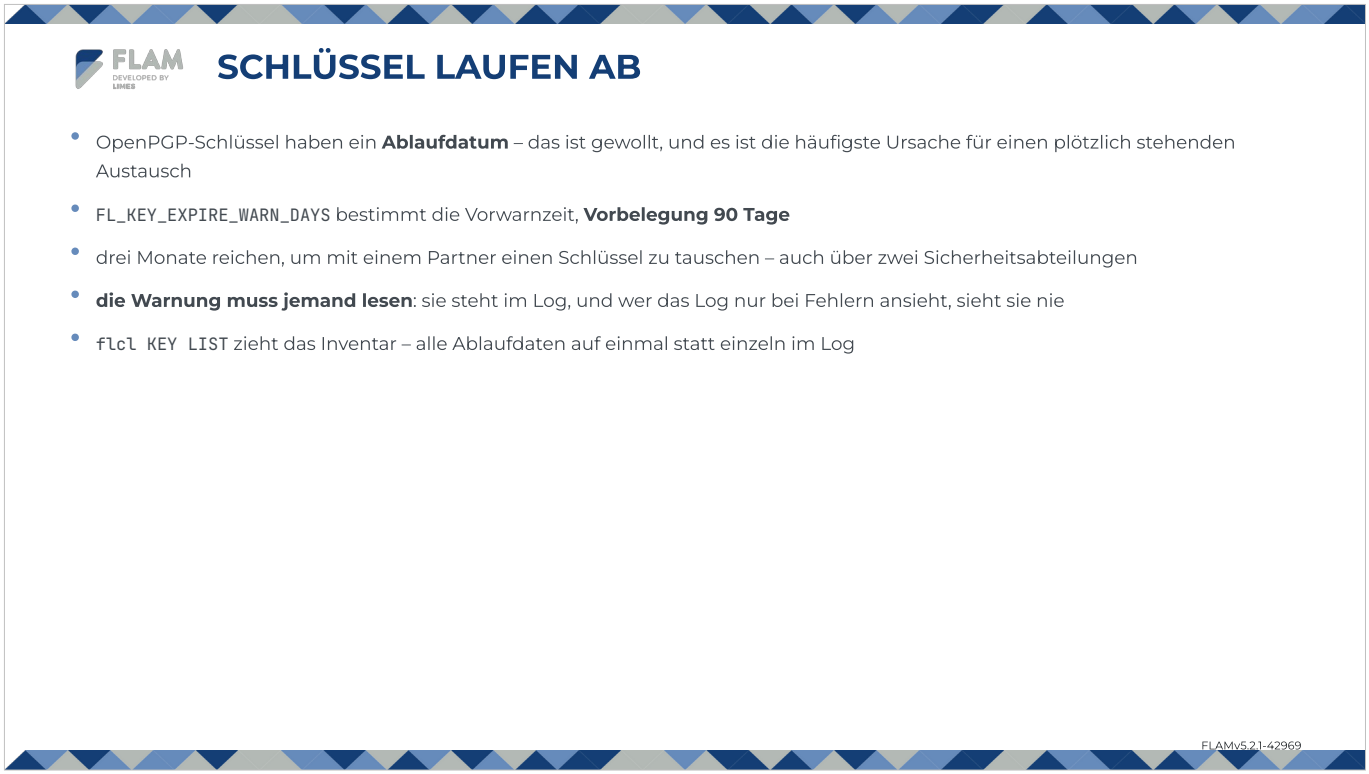
Die Folie, die überrascht – und die einen realen Betriebsfall löst.
Der Fall: eine verschlüsselte Datei liegt vor, und ein **weiterer** Empfänger soll sie lesen können. Die naive Lösung ist: entschlüsseln, neu verschlüsseln. Dabei liegt der Klartext einmal vor, und genau das will man in einer Weitergabestelle nicht.
`REENCYPT` löst das anders. Die Daten sind mit einem **Sitzungsschlüssel** verschlüsselt, und dieser Sitzungsschlüssel ist es, der je Empfänger verschlüsselt im Kopf steht. Um einen Empfänger hinzuzufügen, muss man also nur den **Sitzungsschlüssel** für ihn verschlüsseln – die Nutzdaten bleiben unangetastet.
`MODE=ADD` fügt Empfänger hinzu, `MODE=SET` ersetzt die Liste. Letzteres ist der Weg, wenn ein Empfänger die Berechtigung verlieren soll.
Was es **nicht** tut, und das muss man sagen: die Stelle, die umschlüsselt, braucht Zugriff auf den Sitzungsschlüssel – also einen privaten Schlüssel, für den die Datei verschlüsselt war. Es ist kein Zaubertrick, mit dem ein Unbeteiligter Empfänger hinzufügen kann. Aber der **Klartext der Nutzdaten entsteht nie**.
Der praktische Nutzen: eine Verteilstelle kann eine Datei an wechselnde Empfänger weitergeben, ohne die Daten je im Klartext zu sehen. Für ein Rechenzentrum, das für mehrere Mandanten arbeitet, ist das genau die Trennung, die verlangt wird.



Die zweite Hälfte der Einheit, und sie ist unabhängig von PGP als Containerformat.
Der Sachverhalt aus unserem Handbuch, wörtlich: eine Signaturdatei enthält **nur die Signatur und nicht die Datei, die sie signiert**. Und: die signierten Dokumente werden bei Erzeugung und Prüfung **nicht verändert**.
Warum das zählt: die Datei bleibt genau die, die sie war. Kein Kopf, keine Hülle, kein anderes Format. Ein Empfänger, der die Signatur nicht prüfen will oder kann, verarbeitet die Datei wie immer. Und eine Prüfung ist nachträglich möglich – auch Jahre später, auf einem Archivmedium.
Das ist der Unterschied zu einer PGP-signierten Datei: dort ist die Signatur **in** der Datei, und die Datei ist damit eine PGP-Datei.
Der Aufruf ist symmetrisch: `SIGNATURE.GENERATION` am `WRITE`, `SIGNATURE.VERIFICATION` am `READ`. In beiden steht ein Dateiname für die Signaturdatei, ein Format und die Schlüsselherkunft über `FKM5()`.
Das Format ist derzeit **ausschließlich OpenPGP** – das sollte man sagen, damit niemand nach CMS oder XMLDSig fragt und enttäuscht wird. Die Struktur ist als Überlagerung gebaut, also erweiterbar.
Der typische Einsatz: eine Datei wird übertragen, und daneben liegt die Signatur. Der Empfänger prüft sie und hat damit den Nachweis über Unversehrtheit und Herkunft – ohne dass die Datei selbst angefasst wurde.



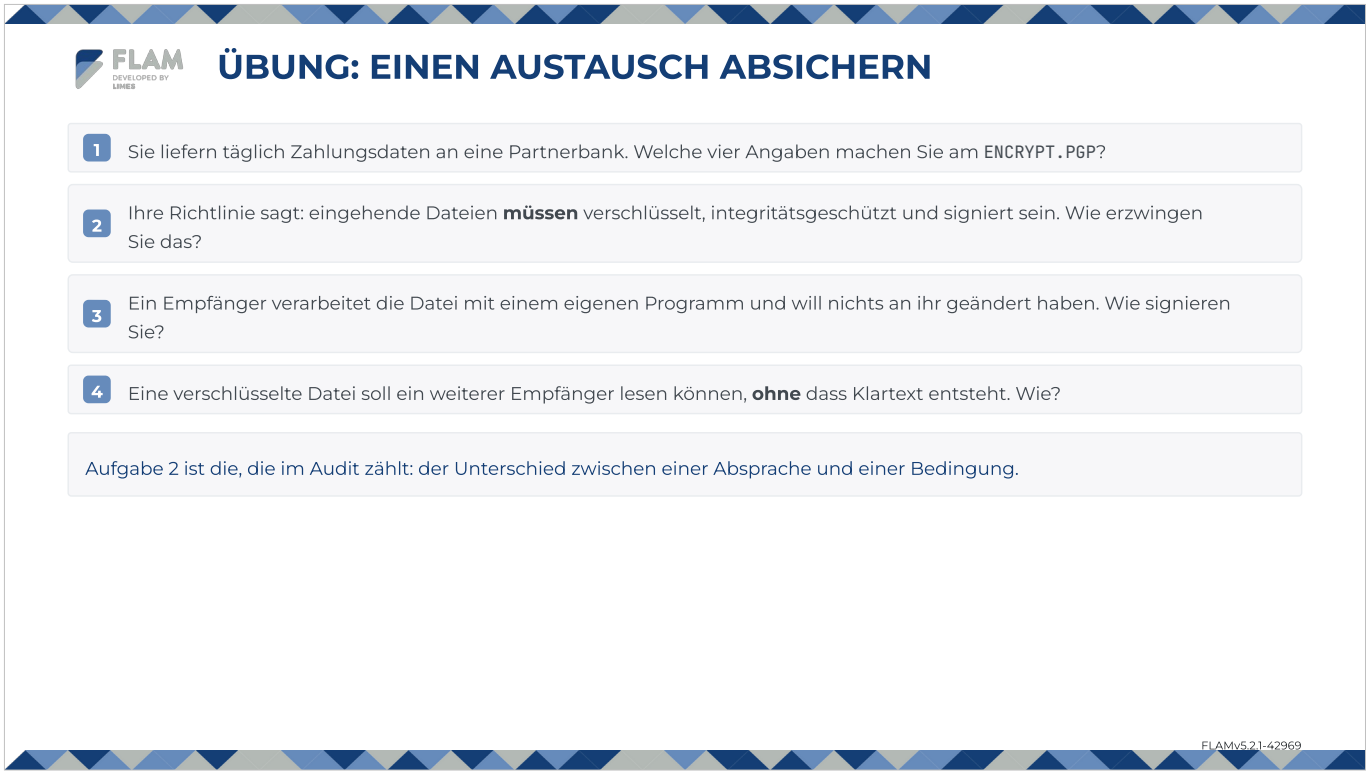
Die kleine Schwester der Signatur, und in der Praxis oft das, was wirklich gebraucht wird.
Eine Signatur beantwortet zwei Fragen: unverändert **und von wem**. Eine Prüfsumme beantwortet nur die erste – aber sie braucht keinen Schlüssel, keine Infrastruktur und keine Absprache über Formate.
Das `HASH`-Objekt am `WRITE` erzeugt sie mit, im selben Lauf. Die Auswahl ist breit: von MD5 und SHA-1 über die SHA-2- und SHA-3-Familie bis zu SHAKE, und dazu die CRC-Varianten. Die CRCs sind keine Krypto – sie finden Übertragungsfehler, nicht Manipulation. Man sollte den Unterschied im Raum klarmachen, weil er regelmäßig verwischt wird.
`FORMAT` bestimmt, wie die Prüfsumme geschrieben wird: `GNU` und `BSD` sind die Formate der Unix-Werkzeuge – damit kann der Empfänger mit `sha256sum -c` prüfen, ohne irgendetwas von FLAM zu wissen. `HEX` und `BIN` sind für eigene Weiterverarbeitung.
Und `CHECK` ist die andere Richtung: beim Lesen gegen eine mitgelieferte Prüfsumme prüfen.
Der Satz für den Raum: **wenn Sie nur wissen wollen, ob die Übertragung sauber war, ist eine Prüfsumme das richtige Werkzeug** – und `FORMAT=GNU` macht sie für jeden prüfbar.



Eine Betriebsfolie, kurz, aber sie verhindert einen Ausfall.
OpenPGP-Schlüssel haben ein Ablaufdatum – das ist gewollt und gut. In der Praxis ist es die häufigste Ursache für einen plötzlich stehenden Datenaustausch: der Schlüssel des Partners ist abgelaufen, und niemand hat darauf geachtet.
FLAM warnt vorher. `FL_KEY_EXPIRE_WARN_DAYS` legt fest, wie früh – standardmäßig **90 Tage**. Drei Monate sind genug, um mit einem Partner einen neuen Schlüssel auszutauschen, auch wenn es über zwei Sicherheitsabteilungen läuft.
Der Punkt, der daraus folgt: **die Warnung muss jemand lesen.** Sie steht im Log. Wer das Log nur bei Fehlern ansieht, sieht sie nie – und erlebt stattdessen den Abbruch am Tag des Ablaufs.
Praktischer Rat: die Warnung in die Auswertung aufnehmen, die ohnehin läuft. Und das Schlüsselinventar über `flcl KEY LIST` regelmäßig ziehen – dann sieht man die Ablaufdaten aller Schlüssel auf einmal statt einzeln im Log.



Vier, alle aus Supportfällen.
Zur ersten: die Erzwingungsschalter sind aus. Wer glaubt, eine Richtlinie umgesetzt zu haben, weil sie in einem Dokument steht, hat sie nicht umgesetzt.
Zur zweiten: `ARMOR` ohne Not kostet ein Drittel Platz und bei großen Dateien entsprechend Übertragungszeit.
Zur dritten: doppelte Kompression. PGP komprimiert selbst; wer vorher schon komprimiert hat, sollte es abschalten.
Zur vierten: eine verschlüsselte Datei ohne Signatur beweist nichts über die Herkunft. Das ist die Lücke, die im Audit am häufigsten auffällt.



Fünfzehn Minuten.
Die Lösungen:
Aufgabe 1. `METHOD=ENCSIG`, `FORMAT=SEIP`, die Empfänger als Array, der Signierer benannt, `SIGHASH=SHA256`. `ARMOR` nur, wenn der Kanal es verlangt.
Aufgabe 2. Auf der Leseseite `ENCRYPED`, `INTEGRITY` und `SIGNING` setzen. Damit ist die Anforderung technisch erzwungen; eine Datei, die sie nicht erfüllt, führt zum Abbruch und nicht zur Verarbeitung.
Aufgabe 3. Eine abgetrennte Signatur: die Datei bleibt unverändert, der Empfänger kann sie ohne Kenntnis der Signatur verarbeiten, und die Prüfung ist später noch möglich. Eine PGP-signierte Datei wäre eine PGP-Datei und damit ein anderes Format.
Aufgabe 4. `REENCYPT(MODE=ADD)` – den Sitzungsschlüssel für den neuen Empfänger verschlüsseln. Die Nutzdaten werden nicht angefasst, und der Klartext entsteht nicht. Voraussetzung: die Stelle hat einen privaten Schlüssel der Datei.

WAS BLEIBT

- **OpenPGP ist der Weg nach draußen** – der Empfänger muss nichts von FLAM wissen
- drei Betriebsarten: `CRYPT`, `SIGN`, `ENCSIG` – **Vertraulichkeit und Echtheit sind zwei Anforderungen**
- `SEIP` statt `SENC`**: mit Integritätsschutz;** `SENC` nur für alte Empfänger
- auf der Leseseite **erzwingen**: `ENCRYPED`, `INTEGRITY`, `SIGNING` – standardmäßig aus
- **abgetrennte Signatur**: die Datei bleibt unverändert, die Prüfung ist Jahre später möglich
- **Umschlüsseln** fügt Empfänger hinzu, ohne dass Klartext entsteht

FLAMv5.21-42969

Lernziele durchgehen. Der Satz zum Schluss: OpenPGP ist der Weg nach draußen, und die Erzwingungsschalter sind der Unterschied zwischen einer Absprache und einer Richtlinie.



Dritte Einheit der Gruppe Sicherheit, und die unspektakulärste – aber die, nach der ein Auditor zuerst fragt. Nicht "welche Algorithmen", sondern **wie wird es betrieben**.
Der Rahmen in einem Satz, und er sollte deutlich gesagt werden: FLAM kann helfen, viele Anforderungen zu erfüllen – PCI DSS, ISO 27001 und andere.
Aber es gibt Punkte, die der Betreiber beachten muss, sonst ist die Verschlüsselung nur Dekoration. Diese Punkte sind der Inhalt der Einheit.
Ein zweiter Satz zur Ehrlichkeit: einige Fähigkeiten von FLAM sind **absichtlich Hintertüren** – der Virenscan sieht Klartext, die Vorverarbeitung führt Programme aus, das Packet Capturing schreibt Netzverkehr mit. Das ist alles gewollt und nützlich. Man muss nur wissen, dass es da ist, und es schließen, wo man es nicht braucht.



Die Lernziele. Das zweite ist der Kern: es gibt vier Stellen, an denen Klartext oder ein Kommando FLAM verlässt, und alle vier sind nützlich. Wer sie kennt, kann sie schließen.
Das vierte ist der praktische Nutzen für den Betrieb – eine Installationsprüfung, die zwei Minuten kostet und eine Fehlerklasse ausschließt.



Die Folie, die die Verantwortungsgrenze zieht, und man sollte sie nicht überspringen: sie verhindert das Missverständnis "wir haben FLAM, also sind wir sicher".
Links, was FLAM mitbringt: Verschlüsselung über OpenPGP und SSH, Hardwaremodule über PKCS#11 und CCA beziehungsweise ICSF für Schlüsseltausch und Signatur, strenge Zugriffskontrolle über SAF, Prüfsummen, Virenscan.
Rechts, was der Betreiber leisten muss – und das ist die längere Spalte:
die Schlüssel verwalten und wechseln, die Umgebungsvariablen prüfen, die Hintertüren schließen, den Fehlertrace schützen, die Kohärenz der Installation sicherstellen, und die Warnungen lesen.
Der Satz, der hängen bleiben soll: **FLAM stellt Werkzeuge bereit, keine Zustände.** Eine Datei ist nicht verschlüsselt, weil FLAM installiert ist, sondern weil jemand es angeordnet und geprüft hat.



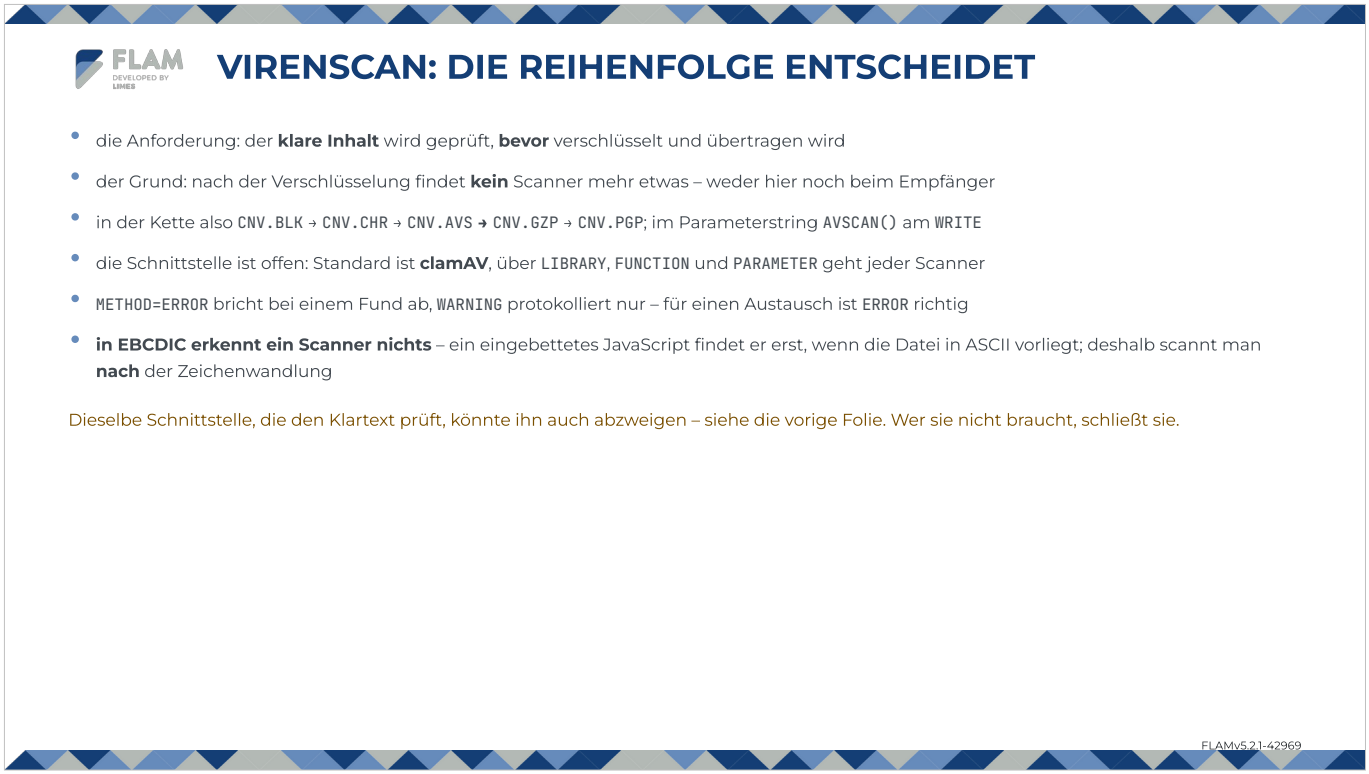
Eine Betriebsfolie, die zwei Minuten kostet und eine ganze Fehlerklasse ausschließt.
Der Sachverhalt: FLAM besteht aus vielen Komponenten, und einige sind dynamisch gebunden. Nach einer unvollständigen Installation oder einem halben Update können Komponenten verschiedener Stände zusammenlaufen. Das äußert sich in Fehlern, die keiner Ursache zuzuordnen sind.
`flcl version` schreibt zu **jeder** Komponente eine Version. Und die entscheidende Zahl ist nicht die Versionsnummer, sondern die **Build-Nummer** – die letzte Zahl hinter dem Bindestrich. Bei einer kohärenten Installation ist sie **überall gleich**.
Im Beispiel unten ist sie es: 41973 in allen Zeilen. Die Versionsnummern dürfen dagegen abweichen – eine Komponente, die sich seit 5.2.0 nicht geändert hat, trägt noch 5.2.0. Das ist normal und kein Befund.
Die Empfehlung aus unserem Handbuch, und sie ist konkret: **einen Installationsprüfprozess einrichten**, der `VERSION` aufruft und prüft, dass alle Build-Nummern gleich sind. Das ist eine Zeile in einem Skript und gehört hinter jede Installation und jedes Update.
Dazu passt: wir signieren Module und Installationspakete, wo die Plattform es zulässt. Wo nicht, liefern wir sichere Prüfsummen, damit man das Download-Paket prüfen kann.



Die Folie zum häufigsten Fehler überhaupt.
Zuerst, was FLAM selbst tut: Passwörter, PINs und Schlüssel, die über den Kommandostring kommen, werden als **geschützt markiert** und nicht im Klartext ausgegeben – in den Ausgaben steht ` **SECRET** `. Das Parsen erkennt sie, danach weiß FLAM, welche Werte kritisch sind.
Und jetzt die Einschränkung, die man kennen muss: **wenn die Eingabeprotokollierung eingeschaltet ist, wird der komplette Kommandostring geschrieben** – vor dem Parsen, also mit allem darin. Das ist genau die Hintertür von Folie 6. Wichtiger noch: FLAM kann nichts dafür tun, dass ein Passwort im Kommandostring schon **vorher** überall stand – in der Prozessliste, im Joblog, in der History der Shell, im Änderungsprotokoll der JCL.
Deshalb die vier Stufen, von schlecht nach gut:
Im Kommandostring – funktioniert, und ist die schlechteste Stufe.
In einer Parameterdatei – die Datei kann man mit Rechten schützen, und sie steht nicht in der Prozessliste. Das ist die Stufe, die jedes Haus sofort erreichen kann.
In einem verschlüsselten Member – neu, und die Stufe, die man ohne zusätzliche Hardware erreichen kann: die Parameterdatei liegt als Member in einem verschlüsselten FLAM5-Archiv, das Kommando nennt nur den Membernamen (`cred.fl5/?s3.provider`), und entschlüsselt wird über `FL_ARCHIVE_DECRYPT`. Jetzt schützt nicht mehr das Dateisystem, sondern Kryptographie – und das Archiv darf sogar in der Versionsverwaltung liegen.
Die Einheit zur Kommandosprache zeigt die Mechanik.
Im Hardwaremodul – dann gibt es kein Passwort, das schützenswert wäre.
Das ist die Empfehlung aus unserem Handbuch, und sie ist eindeutig: HSM statt klarer Passwörter für Schlüsselspeicher.
Für z/OS zwei Hinweise, die man im Betrieb braucht: Es gibt dort **keine Pipe**, und in-stream in der JCL ist die **schlechteste** Wahl, nicht die gleichwertige – das Credential steht dann im Joblog und im Spool. Nehmen Sie stattdessen ein Dataset, das RACF schützt, oder verschlüsseln Sie das Archiv für einen OpenPGP-Schlüssel: dann liegt der private Schlüssel in ICSF und es gibt überhaupt kein Passwort. Und für Cloud Data Access gilt:
dort gehört gar nichts hinein, CDA ist selbst ein Credential-Store.
Und der Satz, der im Raum bleiben soll: die Frage ist nicht, ob das Passwort stark ist. Die Frage ist, **wie viele Stellen es kennen**.



Die wichtigste Folie der Einheit, und sie ist ungewöhnlich für einen Hersteller: wir zählen selbst auf, wo Klartext oder ausführbarer Code das System verlässt.
Alle vier sind nützlich und alle vier sind gewollt. Der Punkt ist nicht, dass sie schlecht sind, sondern dass man sie **kennen und gezielt schließen** muss.
Und der Satz, der vor die Liste gehört: **jede dieser Türen ist ab Werk zu.** Die Eingabeprotokollierung ist aus, das Datenlogging ist aus, und im Fehlertrace werden die Daten ausgesternt, solange nicht ausdrücklich eine Umgebungsvariable etwas anderes sagt. Der Grund ist nicht Vorsicht um ihrer selbst willen, sondern ein konkreter Fall: **damit ein Kunde uns nicht versehentlich sensible Daten in den Support schickt.** In einem Parameterstring steht auch mal eine Passphrase oder eine PIN, und im Trace eines Zeichensatzfehlers stehen die Bytes davor und danach.
Die Eingabeprotokollierung schreibt den kompletten Parameterstring ins Log – vor dem Parsen, also mit Passwörtern. Sie ist standardmäßig aus und nur für die Fehlersuche gedacht. Die SAF-Policy `LOG.INPUT` verhindert, dass jemand sie einschaltet.
Das Packet Capturing schreibt Netzverkehr mit. Unentbehrlich, wenn eine SSH-Verbindung nicht funktioniert – und die Pakete enthalten die Daten.
Policy: `SSH.PCAP.ALLOWED`.
Die Vor- und Nachverarbeitung führt Kommandos aus, und zwar auch **auf dem entfernten System**. Sie ist für Prozessautomatisierung gebaut und ist mächtig. Sie ist damit auch der direkteste Weg, fremden Code auszuführen.
Es gibt einen kompletten Satz SAF-Kontrollen dafür.
Die Virenscan-Schnittstelle bekommt die Daten **im Klartext** – das muss sie, sonst könnte sie nichts finden. Damit ist sie gleichzeitig ein Weg, Klartext abzuzweigen. Unser Handbuch sagt das wörtlich: diese Schnittstelle ist gleichzeitig eine Hintertür und sollte für alle anderen Fälle geschlossen werden.
Der Satz für den Raum: **auf z/OS kann man alle vier über SAF schließen.** Auf anderen Plattformen gibt es diesen Schutz nicht – dort muss man über Dateirechte und Konfiguration arbeiten. Das ist eine ehrliche Auskunft und sie gehört dazu.



Eine eigene Folie, weil die Reihenfolge der einzige Punkt ist, der hier zählt – und weil sie gern falsch gemacht wird.
Die Anforderung, die mehrere Sicherheitsstandards stellen: der **klare Inhalt** einer Datei muss geprüft werden, **bevor** sie verschlüsselt und auf ein entferntes System geschrieben wird.
Der Grund ist einfach und wird trotzdem übersehen: nach der Verschlüsselung kann **kein** Scanner mehr etwas finden. Weder bei uns noch beim Empfänger.
Wer erst verschlüsselt und dann scannt, hat einen Scanner, der Zufallsbytes prüft.
Deshalb sitzt der Scan in der Konvertierungskette **vor** der Verschlüsselung – in der Einheit *Batch-Einbettung* ist das die Kette `CNV.BLK`, `CNV.CHR`, **`CNV.AVS`**, dann `CNV.GZP` und `CNV.PGP`.
Im Parameterstring ist es `AVSCAN()` am `WRITE`.
Die Schnittstelle ist offen: unsere Standardimplementierung benutzt clamAV, aber man kann jeden Scanner anbinden – über `LIBRARY`, `FUNCTION` und `PARAMETER`.
`METHOD` entscheidet, was ein Fund bedeutet: `WARNING` protokolliert und läuft weiter, `ERROR` bricht ab. Für einen Datenaustausch ist `ERROR` richtig – eine infizierte Datei soll nicht beim Partner ankommen.
Der zweite Grund für die Reihenfolge ist der, den niemand erwartet, und er gehört gesagt: **ein Virenscanner erkennt in EBCDIC nichts.** Ein eingebettetes JavaScript wird nur erkannt, wenn die Datei in ASCII vorliegt – dieselbe Datei in EBCDIC geht durch. Der Scan muss also **nach** der Zeichenwandlung laufen, und das heißt auch: über die Leitung muss klares ASCII gehen, damit die Prüfung überhaupt greifen kann.
Und die ehrliche Einordnung, die man geben soll, wenn jemand nach dem Wert fragt: **Virenscannen macht nichts besser.** Erkannt wird nur, was schon bekannt ist, und dafür eine Hintertür in ein Verschlüsselungsprodukt zu bauen, ist eine Farce. Was wir daraus gemacht haben: die Schnittstelle ist offen, damit es auch in der offenen Welt eine Lösung gibt – und man kann **FLAC/FAVE ausschalten** und damit dieses Hintertürchen zumachen.
Der Hinweis von Folie 6 gehört dazu: dieselbe Schnittstelle, die den Klartext prüft, könnte ihn auch abzweigen. Wer sie nicht braucht, sollte sie über SAF schließen.



Eine kurze Folie mit einer Aussage, die man nur einmal hören muss und dann nie vergisst.
Der Fehlertrace ist eines der nützlichsten Werkzeuge von FLAM. Für eine brauchbare Ursachenanalyse **könnte** er die beteiligten Daten mitschreiben – und deshalb tut er es ab Werk **nicht**: die Daten werden **ausgesternt**, solange nicht `FL_WRITE_DATA_TO_ERRTRACE` ausdrücklich etwas anderes sagt.
Die Hilfe zu der Variablen sagt es selbst: "must be OFF if error trace is to be part of the support ticket".
In unserer eigenen Testumgebung setzt das FTINIT sie **an**, weil man dort die Bytes an der Fehlerstelle sehen will. Die Regel für ein Haus ist damit einfach: **in der Produktion aus, im Test an.** Und die praktische Folge für ein Ticket: entweder die Variable war aus – dann ist der Trace verschickbar – oder der Fall wird mit Testdaten nachgestellt.
Damit hat er denselben Charakter wie ein Dump. Ein Dump wird auch nicht weitergegeben, weil er praktisch ist, sondern geschützt, weil er alles enthält.
Die praktische Folge für ein Ticket: ein Trace darf zu uns – aber jemand muss entschieden haben, dass er das darf. Bei produktiven Zahlungsdaten heißt die Antwort meist: den Fall mit Testdaten nachstellen.
Dazu passt aus der Einheit *Doku und Support* die Regel, was ins Ticket gehört. Und aus unserem eigenen QA-Vortrag der Satz, der es zusammenfasst:
wer sensible Daten verschickt, hat sie beim Losschicken kompromittiert.
Zwei Nachbarn gehören auf dieselbe Folie, weil sie dieselbe Logik haben:
flucFS sollte nur unter dem eigenen Benutzer laufen – sonst kann man über das Dateisystem Rechte umgehen. Und die **FLAM Started Task** auf z/OS ist eine aktive Komponente im Hintergrund; sie implementiert keine eigenen Supervisor Calls, und ihre PC-Routinen werden in unserer QA mit zACS angegriffen. Der einzige Befund dort sind reguläre Beendigungen mit 0C4.



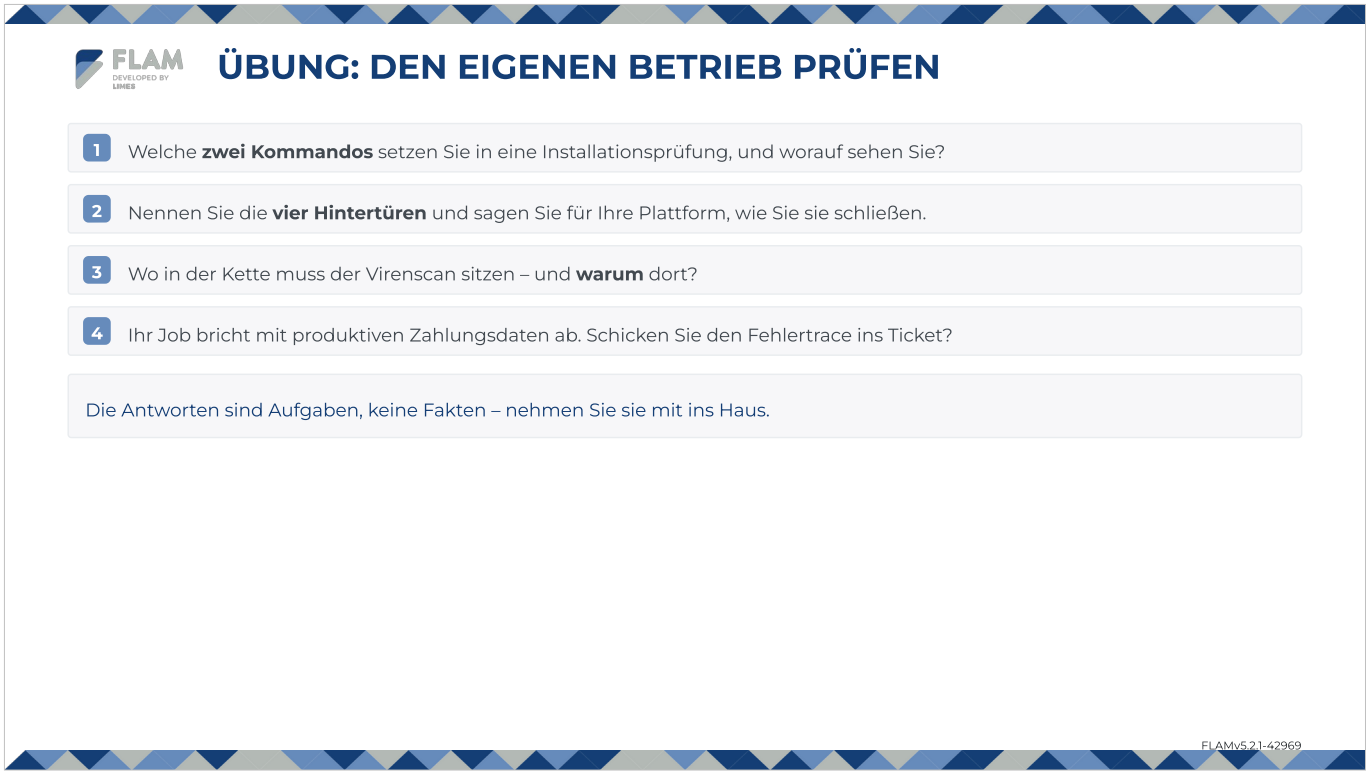
Die Nachschlagefolie, und die Liste ist ausdrücklich **nicht vollständig** – das sagt unser Handbuch selbst. Sie enthält die, die man kennen sollte.
`FL_ALLOW_LOW_ENTROPY` erlaubt schwachen Zufall. Bequem, wenn ein Job abbricht – und damit ein Weg zu schwachen Schlüsseln.
`FL_DEACTIVATE_HW_CRYPTO_ACCELERATION` schaltet die Hardwarebeschleunigung ab und erzwingt Software. Ein Diagnosewerkzeug.
`FL_PGP_PREF_SYMALG_AES128` dreht die Reihenfolge der bevorzugten symmetrischen Algorithmen – statt AES-256 zuerst dann AES-128. Es gibt Gründe dafür (ein Partner, der 256 nicht kann), aber es ist eine Abschwächung.
`FL_SSH_HOME_DIR` bestimmt, wo die SSH-Schlüssel und die `known_hosts` liegen. Wer das umlenkt, lenkt die Vertrauensbasis der Verbindungen um.
`FL_KEY_EXPIRE_WARN_DAYS` ist die Vorwarnzeit für ablaufende Schlüssel, standardmäßig 90 Tage.
Vier gehören dazu, die wir beim Durchgehen von `asFl5EnvVarTab` ergänzt haben (Falk, Review 2026-09-18):
`FL_WRITE_DATA_TO_ERRTRACE` ist die Variable von der vorigen Folie – aus in der Produktion, an im Test.
`FL_CA_BUNDLE` bestimmt, **wem wir bei TLS vertrauen**: eine PEM-Datei, auf z/OS ein SAF-Keyring oder eine Key-Database. Wer das umlenkt, lenkt die Vertrauensbasis um – wie `FL_SSH_HOME_DIR`, nur für HTTPS.
`FL_FAVE_LIBRARY`, `FL_FOBJ_LIBRARY` und `FL_FSEG_LIBRARY` sagen, **welche Bibliothek geladen wird**. Das ist fremder Code im eigenen Prozess, und damit die wirksamste Variable dieser Liste.
`FL_DEFAULT_P11PIN` und `FL_DEFAULT_RNGPASSWD` legen Geheimnisse in die Umgebung. Das Handbuch rät selbst davon ab, und die Umgebung eines Jobs ist kein Geheimnisspeicher.
Die vollständige Liste bekommt man übrigens ohne Suchen: `flcl HLPENV` nennt jede Variable, die das Produkt beachtet, mit ihrer Bedeutung.
Und der praktische Teil: **man muss nicht raten, was gesetzt ist.** `flcl info get.system` zeigt die aktuelle Umgebung. Das gehört in die Installationsprüfung von Folie 4 – nicht nur "sind die Build-Nummern gleich", sondern auch "ist etwas gesetzt, das nicht gesetzt sein sollte".
Auf z/OS gibt es zu einigen davon SAF-Policies. Das ist der Unterschied, den die nächste Einheit ausführt: eine Umgebungsvariable ist ein Wunsch des Anwenders, eine Policy ist eine Regel des Hauses.



Die Folie für das Security Operations Center, und sie beantwortet eine Frage, die jedes regulierte Haus stellt: **woher erfahren wir, dass es uns betrifft?** Erstens: es gibt einen **Newsletter zu Sicherheitslücken**. Mit einem Wartungsvertrag kann man sich auf flam.de dafür registrieren. Er wird verschickt, wenn eine Schwachstelle FLAM oder eine der benutzten Bibliotheken betrifft. Das ist ausdrücklich als SOC-Unterstützung gedacht.
Zweitens, und das ist der Teil, der im Ernstfall zählt: **man kann selbst feststellen, ob eine Meldung relevant ist.** Zwei Stellen:
In der `install.txt` der Plattform stehen unter Voraussetzungen die Software, auf der FLAM aufsetzt. Steht dort nichts, ist nur das Betriebssystem mit Language Environment und der zugehörigen libc betroffen.
In der `about.txt` stehen **alle externen Bibliotheken mit ihren Versionen**, die FLAM einbindet. Damit ist die Frage "ist diese CVE für uns relevant" in zwei Minuten beantwortbar – ohne uns zu fragen.
Der Satz für das Gespräch: Sie müssen nicht warten, bis wir etwas sagen. Sie können nachsehen. Die vollständige Behandlung – die zwei CVE-Listen mit Begründung je Eintrag – steht in der Einheit *Betrieb und Wartung*. Hier gehört nur hin, wo man nachsieht.



Vier, alle aus der Praxis.
Zur ersten: die Eingabeprotokollierung wird für eine Fehlersuche eingeschaltet und dann vergessen. Ab dann steht jedes Passwort im Log.
Zur zweiten: der Virenscan nach der Verschlüsselung. Er läuft, er meldet nichts, und alle sind beruhigt – weil er Zufallsbytes prüft.
Zur dritten: ein Trace im Ticket. Gut gemeint, und bei produktiven Daten ein Vorfall.
Zur vierten: die Installationsprüfung, die niemand macht. Uneinheitliche Build-Nummern erzeugen Fehler, die man an der falschen Stelle sucht.

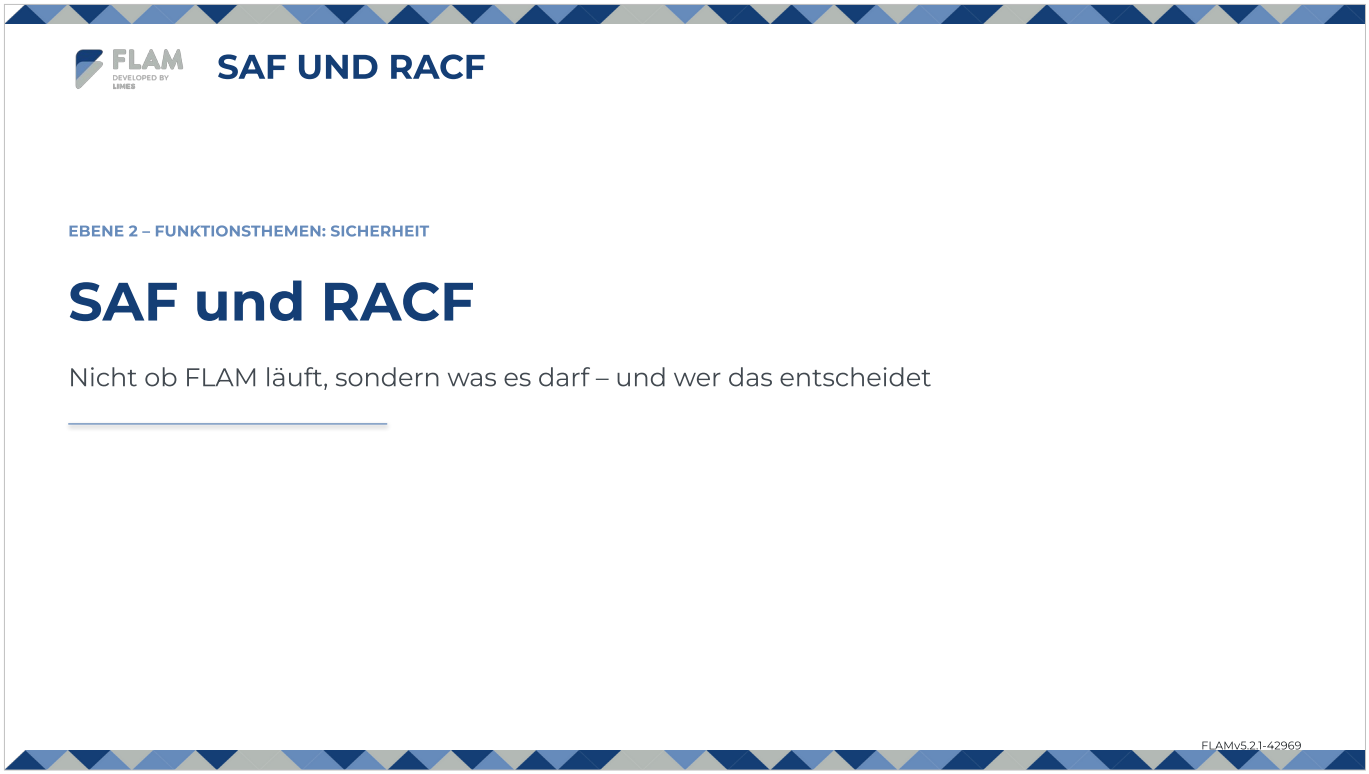


Fünfzehn Minuten, und diese Übung ist absichtlich auf das eigene Haus gerichtet – die Antworten sind Aufgaben, die die Teilnehmer mitnehmen.
Die Lösungen:
Aufgabe 1. `flcl version`, alle Build-Nummern vergleichen; dazu `flcl info get.system` und nachsehen, ob eine der Variablen von Folie 9 gesetzt ist. Beides in ein Skript, das nach jeder Installation läuft.
Aufgabe 2. Die vier: Eingabeprotokollierung, Packet Capturing, Vor- und Nachverarbeitung, Virenscan-Schnittstelle. Auf z/OS über SAF schließen, auf anderen Plattformen über Rechte und Konfiguration.
Aufgabe 3. Vor der Verschlüsselung – weil danach niemand mehr etwas findet. Praktisch: `AVSCAN()` am `WRITE`, `METHOD=ERROR` für einen Austausch.
Aufgabe 4. Nein. Der Trace enthält beteiligte Daten. Entweder mit Testdaten nachstellen, oder jemand mit Befugnis entscheidet und der Trace wird geschützt übertragen.

WAS BLEIBT

- **FLAM stellt Werkzeuge bereit, keine Zustände** – die Anordnung und die Prüfung macht der Betrieb
- **vier absichtliche Hintertüren**: Eingabeprotokollierung, Packet Capturing, Vor-/Nachverarbeitung, Virenscan – auf z/OS über SAF schließbar
- der **Virenscan muss vor die Verschlüsselung**, danach findet kein Scanner mehr etwas
- Credentials in drei Stufen: Kommandostring **schlecht**, Parameterdatei sofort erreichbar, **HSM empfohlen**
- der **Fehlertrace ist ein Dump** – mit Testdaten nachstellen statt weitergeben
- **gleiche Build-Nummern** in `flcl version` sind die kürzeste Installationsprüfung

Lernziele durchgehen. Der Satz zum Schluss ist der von Folie 3: FLAM stellt Werkzeuge bereit, keine Zustände.



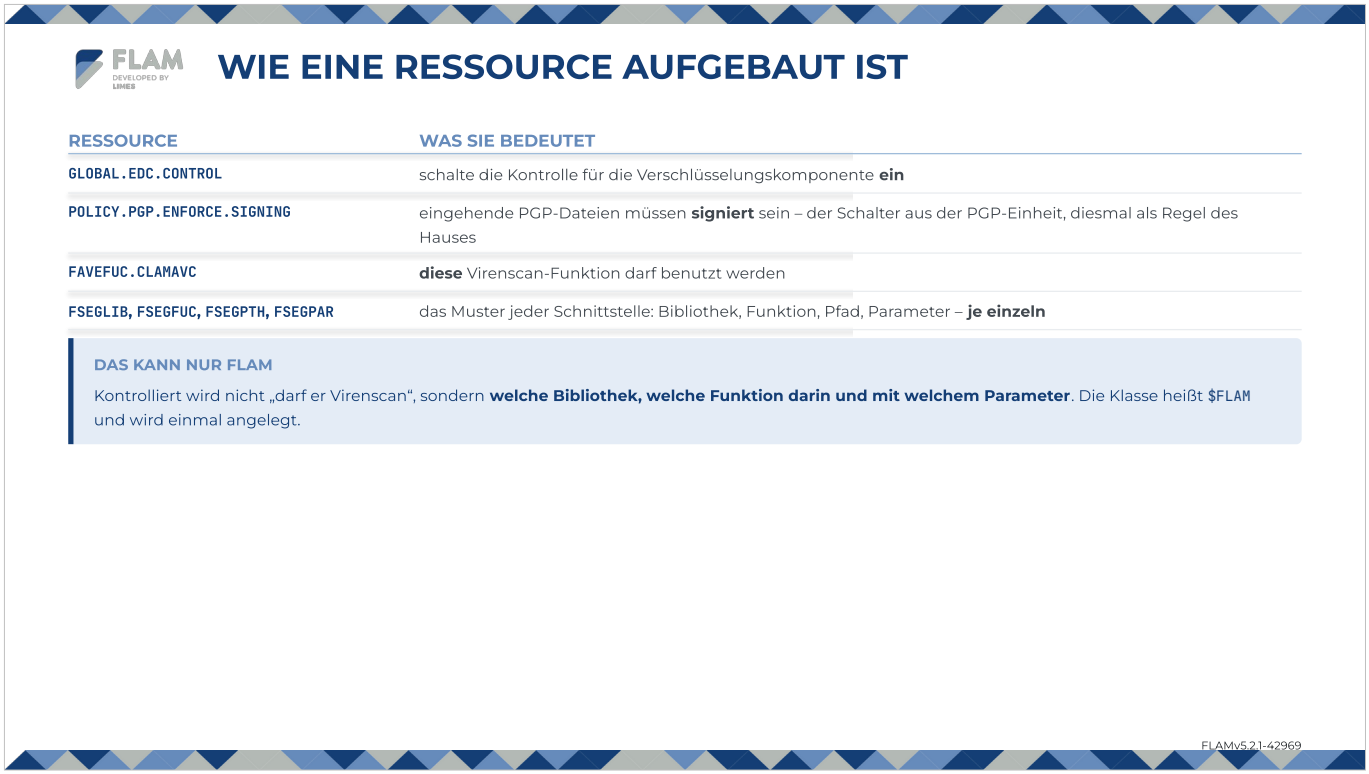
Letzte Einheit der Gruppe Sicherheit, und die einzige, die an eine Plattform gebunden ist. Adressat ist der **Sicherheitsverantwortliche**, nicht der Anwender – das sollte man am Anfang sagen, damit die Entwickler im Raum wissen, warum sie zuhören: sie erfahren, was ihnen verboten werden kann.
Der Rahmen in einem Satz: seit Version 5.1.21 kann ein RACF-Administrator auf z/OS Sicherheitsrichtlinien aktivieren und einen großen Teil dessen kontrollieren, was FLAM tut. **Nicht ob FLAM läuft, sondern was es darf.** Und der Satz, der die Einheit zusammenhält – er kam in den vorigen Einheiten schon vor: eine Umgebungsvariable ist ein **Wunsch des Anwenders**, eine Policy ist eine **Regel des Hauses**. Hier lernen wir, wie man Regeln macht.



Die Lernziele. Das dritte ist das, weswegen die Einheit existiert: die Mechanik über `GLOBAL.MUSTDEF` ist nicht zu erraten, und wer sie falsch versteht, richtet einen Schutz ein, der nichts tut.
Das vierte ist der Satz, der einem RACF-Administrator die Arbeit erspart – und der Fehler, den er sonst macht.



Die Bestandsaufnahme, und sie ist länger, als die meisten erwarten. Nicht vorlesen – zwei oder drei Punkte herausgreifen, die im Raum Bedeutung haben.
Die Liste aus unserem Handbuch, gruppiert:
Richtlinien: PCAP abschalten, Fingerabdrücke erzwingen, Mindestlängen für Schlüssel und Passwörter.
Die Parameterstrings selbst – das ist der weitreichendste Punkt. Über `CLPAR` lässt sich kontrollieren, welche Teile der Kommandosprache ein Anwender überhaupt benutzen darf. Damit ist die Funktionalität selbst steuerbar, nicht nur ihr Ergebnis.
Vor- und Nachverarbeitung: welche Kommandos ausgeführt werden dürfen, lokal und entfernt.
Alle Schnittstellen: FAVE (Virenscan), FCPE (Vor-/Nachverarbeitung), FKME (Schlüssel), FSEG und FOBJ (Speicher). Einschließlich der Frage, **an welchen Virenscanner** die klaren Daten übergeben werden dürfen.
Komplette URLs – Verbindung, Pfad, Dateiname und Member. Damit kann man sogar einzelne Member eines PDS/E, eines ZIP oder eines FLAM-Archivs schützen.
Verbindungen: SSH mit Benutzer, Server und Port; HTTP/REST mit Methodenebene und Schutz gegen TLS-Umgehung.
PGP-Schlüsselverwaltung: welche Benutzerkennungen importiert, exportiert, erzeugt oder gelöscht werden dürfen – und welche User- und Key-IDs zum Signieren und für den Schlüsseltausch benutzt werden dürfen.
Zusätzliche SMF-Protokollierung, etwa Satztyp 119 für SSH.
Alle Feature-Codes des Lizenzmoduls. Der Satz für den Raum: das ist keine Zugriffskontrolle auf Dateien – die hat z/OS schon. Das ist Kontrolle über **Funktionen eines Programms**, und das ist ungewöhnlich.



Die Lesehilfe, kurz und nötig, weil alles Weitere darauf aufbaut.
Es gibt eine **eigene Ressourcenklasse**, `$FLAM`. Sie wird einmal über `RDEFINE CDT` angelegt; das steht im Muster-Setup im Handbuch.
Darin hat jede Ressource die Form **Gruppe.Entität**. Die Gruppe sagt, worum es geht, die Entität, welches konkrete Ding.
Drei Beispiele, an denen das Muster sichtbar wird:
`GLOBAL.EDC.CONTROL` – Gruppe `GLOBAL`, und die Entität sagt: schalte die Kontrolle für die Verschlüsselungskomponente ein.
`POLICY.PGP.ENFORCE.SIGNING` – Gruppe `POLICY`, und die Entität ist die Richtlinie: eingehende PGP-Dateien müssen signiert sein. Das ist genau der Schalter aus der PGP-Einheit, nur diesmal als Regel des Hauses.
`FAVEFUC.CLAMAVC` – Gruppe `FAVEFUC`, Entität `CLAMAVC`: diese Virenscan-Funktion darf benutzt werden.
Die Schnittstellen haben je drei Gruppen, und das Muster ist bei allen gleich: `...LIB` für die Bibliothek, `...FUC` für die Funktion, `...PAR` für den Parameter. Bei FSEG und FOBJ kommt `...PTH` für den Pfad dazu.
Man kontrolliert also nicht "darf er Virenscan", sondern **welche Bibliothek, welche Funktion darin und mit welchem Parameter**.



Die zentrale Folie der Einheit. Wer sie versteht, versteht das ganze Verfahren; wer sie überspringt, richtet einen Schutz ein, der nicht das tut, was er glaubt.
Der Ausgangspunkt ist eine Frage, die jedes Berechtigungssystem beantworten muss: **was passiert mit einer Ressource, die nicht definiert ist?** RACF liefert dafür drei Antworten: 0 heißt berechtigt, 4 heißt **Ressource nicht definiert**, 8 heißt nicht berechtigt. Die 4 ist der interessante Fall, denn in einem großen Setup ist immer etwas nicht definiert.
FLAM entscheidet das nicht selbst, sondern lässt es sich sagen – über **eine** Ressource: `GLOBAL.MUSTDEF`. Und der Weg ist elegant: die **Zugriffsklasse**, die dem FLAM-Benutzer darauf gegeben wird, bestimmt die Schwelle für **alle** weiteren Prüfungen.
Die Tabelle liest man so: mit `CONTROL` – das ist die Empfehlung im Muster-Setup – werden undefinierte Ressourcen akzeptiert. Das ist der freundliche Betrieb: man definiert, was man kontrollieren will, und alles andere läuft.
Mit `UPDATE`, `READ` oder `NONE` werden undefinierte Ressourcen **abgelehnt**. Das ist der strenge Betrieb: was nicht ausdrücklich erlaubt ist, ist verboten. Wer das einschaltet, muss vollständig definieren – sonst steht der Betrieb.
Und `ALTER` schaltet SAF **komplett ab**.
Ein technisches Detail für die, die es genau wissen wollen: unterhalb von `CONTROL` setzt FLAM zusätzlich ein Strong-Flag, das den Speicherschnittstellen mitgeteilt wird – ein Provider erfährt also, dass er in einem streng kontrollierten Betrieb läuft.



Diese Folie ist die wichtigste der Einheit, und sie widerspricht dem Bauchgefühl jedes RACF-Administrators.
Die Erwartung: mehr Berechtigung heißt mehr Möglichkeiten für den Benutzer.
Also gibt man großzügig `ALTER`, um niemanden zu behindern.
Bei den `GLOBAL..*CONTROL*`-Ressourcen ist es **umgekehrt**: die Kontrolle für eine Komponente ist genau dann **aktiv**, wenn der Benutzer **kein** `ALTER` hat. Wer `ALTER` vergibt, sagt damit: für diesen Benutzer wird diese Komponente nicht geprüft.
Das steht so auch im Muster-Setup unseres Handbuchs, und man kann es dort sehen: hinter den `PERMIT`-Zeilen mit `ACC(ALTER)` steht als Kommentar `/*INACTIV*/`, und die auskommentierten Zeilen sind mit `ACTIVE` gekennzeichnet. Die Datei sagt es selbst.
Warum das so gebaut ist: damit ein Haus die Kontrolle **einzeln einschalten** kann, ohne alles definieren zu müssen. Im Grundzustand hat jeder `ALTER`, nichts wird geprüft, FLAM läuft wie ohne SAF. Dann nimmt man Komponente für Komponente das `ALTER` weg und definiert dafür die Ressourcen.
Der Satz für den Raum, und er sollte langsam gesagt werden: **eine großzügige Berechtigung schaltet den Schutz aus, sie erweitert ihn nicht.** Wer nach dem Einrichten prüft, ob es noch läuft, und bei einem Problem `ALTER` gibt, hat den Schutz wieder entfernt – und niemand merkt es, weil nichts abbricht.



Jetzt der Nutzen, und diese Folie ist die Antwort auf "was haben wir davon".
Die Auswahl ist so gewählt, dass jede Zeile an eine der vorigen Einheiten anknüpft. `POLICY.PGP.ENFORCE.ENCRYPTION`, `...INTEGRITY` und `...SIGNING` sind genau die drei Schalter von der PGP-Folie – nur eben nicht als Wunsch eines Jobs, sondern als Regel des Hauses. **Das ist der Unterschied, um den es in dieser Gruppe geht**: ein Job kann sie setzen, aber er kann sie auch weglassen. Eine Policy kann er nicht weglassen.
`POLICY.SYM.KEY.LEN.ENC` und `POLICY.SYM.PWD.LEN.ENC` erzwingen **Mindestlängen** für Schlüssel und Passwörter beim Verschlüsseln – und es gibt sie in der `.DEC`-Variante auch fürs Entschlüsseln, damit man alte Daten nicht versehentlich mit schwachen Schlüsseln liest.
`POLICY.LOW.ENTROPY` und `POLICY.LOG.INPUT` sind die Gegenstücke zu den Umgebungsvariablen aus der vorigen Einheit.
`POLICY.SSH.PCAP.ALLOWED` schließt das Packet Capturing, `POLICY.SSH.WEAK.HOSTKEY.CHECK.ALLOWED` und `POLICY.SSH.CUSTOM.KNOWN.HOST.FILE.ALLOWED` verhindern, dass sich jemand an der Vertrauensbasis der SSH-Verbindungen vorbeiarbeitet.
`POLICY.KEY.GENERATE.PGP.EXPIRD.REQIRED` verlangt, dass neu erzeugte Schlüssel ein Ablaufdatum haben. Und `POLICY.KEY.GENERATE.PGP.ALGO.RSA.KEYLEN` erzwingt eine Mindestlänge beim Erzeugen.
Ein Detail zur Bedienung: bei den Policies trägt die **Zugriffsklasse den Wert**. Im Muster-Setup steht etwa `POLICY.SSH.AUTH.PUBLICKEY` mit `READ` und `POLICY.SSH.AUTH.PASSWORD` mit `UPDATE` – die Klasse sagt, wie weit etwas erlaubt ist, nicht nur ob.
Und die Schreibweise `EXPIRD.REQIRED` ist die des Produkts. Wir zitieren sie, wie sie ist.



Die Folie mit den Kommandos, die das Handbuch vorgibt – und ohne die alles andere nicht greift. Sie ist die Antwort auf die Rückfrage "und wie fängt der Systemprogrammierer an?".
`$FLAM` ist **keine** vorhandene RACF-Klasse. Sie wird im **Class Descriptor Table** angelegt, und dabei sind vier Angaben entscheidend:
`POSIT(202)` liegt im Anwenderbereich 128 bis 527 – die Nummer ist frei wählbar, solange sie dort liegt und nicht belegt ist.
`MAXLENGTH(246)`, weil unsere Ressourcennamen lang werden: Verbindung, Pfad, Dateiname und Member stehen darin.
`CASE(ASIS)` und `OTHER(ALPHA,NUMERIC,NATIONAL,SPECIAL)`, weil in einem Pfad Klein- und Sonderzeichen vorkommen. Ohne `ASIS` passt kein USS-Pfadname mehr.
`RACLIST(ALLOWED)`, damit die Profile in den Speicher gelegt werden dürfen – das ist die Voraussetzung für die nächsten zwei Zeilen und für die Laufzeit.
Dann `SETROPTS CLASSACT` (Klasse aktivieren), `SETROPTS RACLIST` (Profile in den Speicher) und nach jeder Änderung `REFRESH`. **Das `REFRESH` ist die Zeile, die im Betrieb vergessen wird**: die Regel steht in RACF, wirkt aber erst nach dem Auffrischen.
Und noch einmal der Hinweis von der Folie davor: das Muster stammt aus unserem Handbuch, gemacht werden sollte es von jemandem, der RACF beruflich macht.



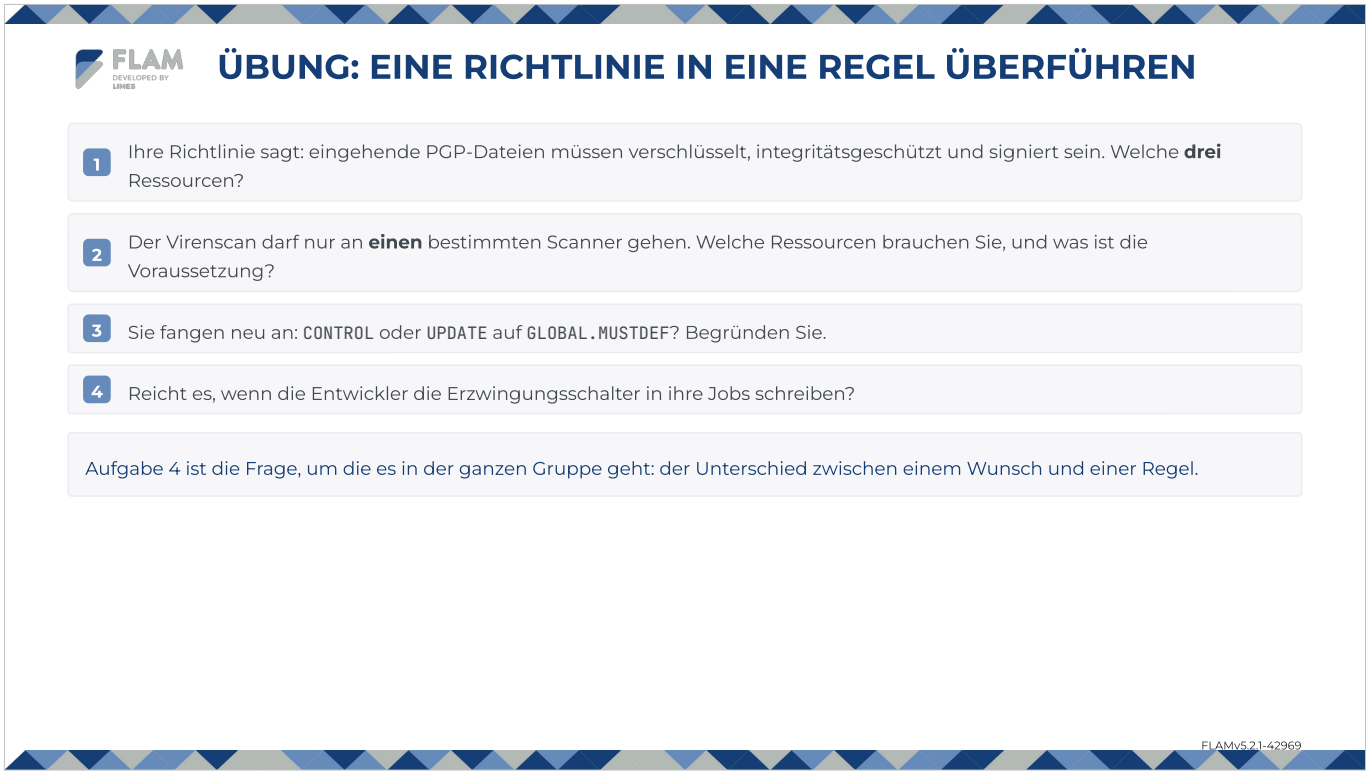
Die praktische Folie: wie geht ein Haus das an, ohne den Betrieb anzuhalten.
Der Weg, den unser Handbuch vorschlägt und der sich bewährt hat, hat vier Schritte:
Erstens die Klasse `$FLAM` anlegen und generische Profile einschalten.
Das ist einmalige Arbeit und ändert noch nichts.
Zweitens alle Ressourcen definieren – mit `` **als Auffangprofil und `UACC(NONE)`. Auch das ändert noch nichts, solange die Berechtigungen großzügig sind.**
Drittens `GLOBAL.MUSTDEF` mit `CONTROL` vergeben. Damit ist SAF an, undefinierte Ressourcen werden aber akzeptiert – der Betrieb läuft weiter.
Viertens, und das ist die eigentliche Arbeit: **Komponente für Komponente** das `ALTER` wegnehmen und die zugehörigen Ressourcen definieren. Erst den Virenscan, dann die Verbindungen, dann die Parameterstrings. Nach jedem Schritt beobachten.
Der Rat, den man dazu geben sollte: **nicht mit `CLPAR` anfangen.** Die Kontrolle über die Parameterstrings ist die mächtigste und die, bei der sich am leichtesten ein Job stilllegt. Sie gehört an das Ende, wenn man die eingesetzten Kommandos kennt.
Und ein Hinweis zur Ehrlichkeit, der auch in unserem Handbuch steht: das Beispiel dort ist ausdrücklich mit Vorsicht zu behandeln – **wir sind keine RACF-Spezialisten.** Das Setup gehört von jemandem gemacht, der es ist.



Die Folie, die eine unangenehme Wahrheit sagt, statt sie zu umgehen – und damit Vertrauen schafft.
Unser Handbuch formuliert es selbst so: es gibt **keinen** vergleichbaren Schutz anwendungsspezifischer Ressourcen auf anderen Betriebssystemen.
Linux, Unix und Windows haben keine SAF-Schnittstelle, in die sich ein Programm einhängen kann, um seine eigenen Funktionen kontrollieren zu lassen.
Was dort an die Stelle tritt, ist schwächer, und man sollte das so sagen:
Dateirechte auf die Konfiguration, die Parameterdateien und die Schlüsselspeicher. Wer die Parameterdatei nicht schreiben darf, kann den Aufruf nicht ändern.
Getrennte Benutzer für getrennte Aufgaben – und flucFS nur unter dem eigenen Benutzer, wie in der vorigen Einheit.
Die Umgebung kontrollieren: wer die Umgebungsvariablen setzen kann, kann Schutz abschalten. Also gehören sie in eine kontrollierte Startumgebung, nicht in das Profil des Anwenders.
Und nicht installieren, was nicht gebraucht wird: die Anbindung an einen Virenscanner, die niemand benutzt, ist eine Hintertür ohne Nutzen.
Der Satz für den Raum: auf z/OS kann der Sicherheitsverantwortliche es **erzwingen**. Auf den anderen Plattformen kann er es nur **organisieren**.
Das ist ein echter Unterschied, und er ist ein Argument für den Host, wo es um Zahlungsverkehr geht.



Vier, und die erste ist die, die in der Praxis wirklich passiert.
Zur ersten: jemand richtet SAF ein, ein Job bricht ab, und um ihn wieder zum Laufen zu bringen wird `ALTER` vergeben. Damit ist der Schutz weg – und es fällt nicht auf, weil nichts abbricht. Genau deshalb steht die Umkehrung auf einer eigenen Folie.
Zur zweiten: der strenge Modus verlangt Vollständigkeit. Wer `UPDATE` auf `MUSTDEF` gibt und dann eine Ressource vergisst, hält den Betrieb an einer Stelle an, an die niemand gedacht hat.
Zur dritten: `CLPAR` zuerst einzuschalten ist der schnellste Weg, einen produktiven Job stillzulegen.
Zur vierten: eine Policy ist nur dann eine Regel, wenn sie auch definiert ist. Im freundlichen Modus wird eine **nicht definierte** Policy akzeptiert – sie tut also nichts. Wer eine Richtlinie einführt, muss sie definieren, nicht nur benennen.

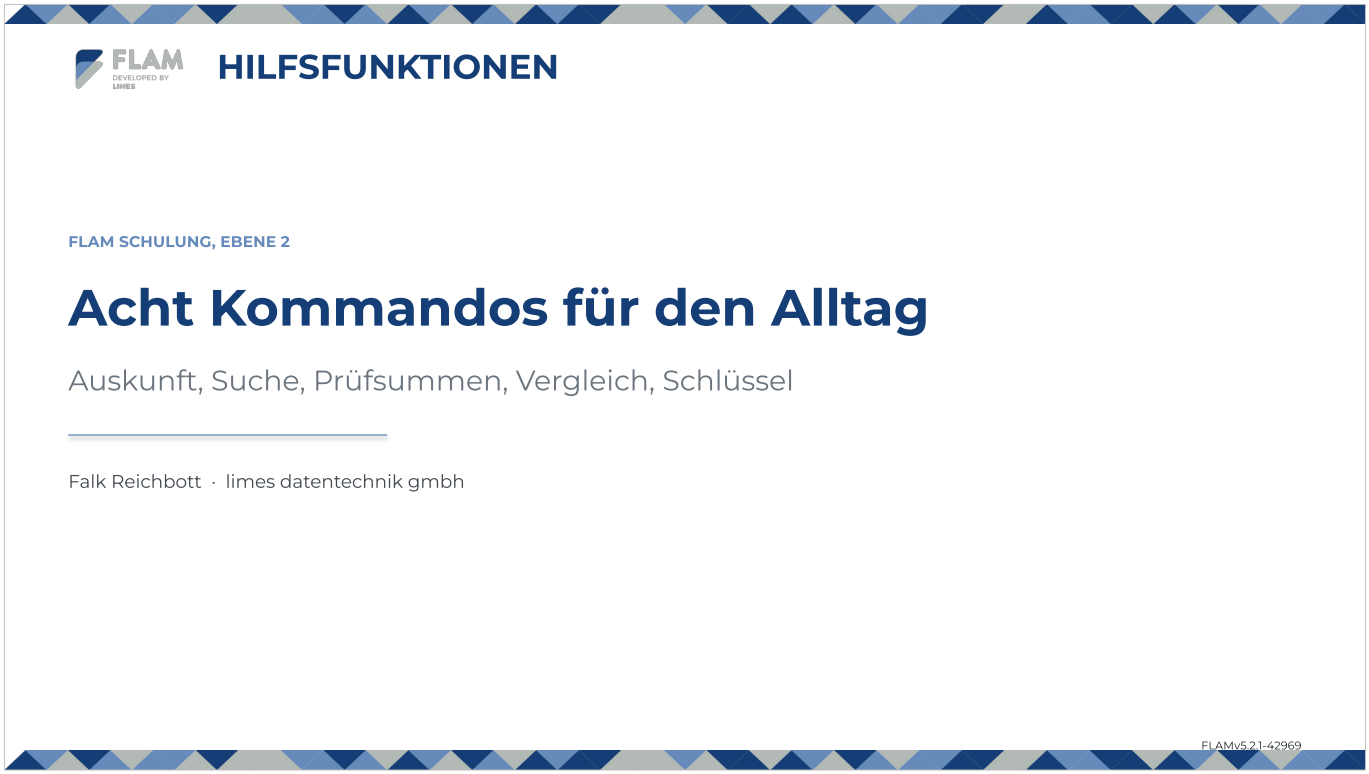


Fünfzehn Minuten, und die Übung ist absichtlich die Brücke zu den vorigen Einheiten.
Die Lösungen:
Aufgabe 1. `POLICY.PGP.ENFORCE.ENCRYPTION`, `...INTEGRITY` und `...SIGNING` definieren und so berechtigen, dass sie greifen. Damit ist die Richtlinie nicht mehr abhängig davon, ob ein Job die Schalter setzt.
Aufgabe 2. `GLOBAL.AVE.CONTROL` aktivieren – also **kein** `ALTER` darauf – und dann `FAVELIB`, `FAVEFUC` und `FAVEPAR` nur für die Bibliothek, die Funktion und den Scanner definieren, die benutzt werden sollen. Alles andere ist damit ausgeschlossen.
Aufgabe 3. Mit `CONTROL` anfangen. Der strenge Modus verlangt ein vollständiges Setup, und das hat man am Anfang nicht. Umstellen kann man später – und sollte man, wenn das Setup steht.
Aufgabe 4. Nein, und das ist der Kern der Einheit: der Schalter im Job ist ein Wunsch. Er kann weggelassen werden, und niemand merkt es. Die Policy ist die Regel.

WAS BLEIBT

- SAF kontrolliert **Funktionen eines Programms**, nicht Dateien – das ist ungewöhnlich und der eigentliche Wert
- eine Ressource ist **Gruppe.Entität** in der eigenen Klasse `$FLAM`
- `GLOBAL.MUSTDEF` bestimmt die Strenge aller Prüfungen – `CONTROL` ist freundlich, `UPDATE` und weniger ist streng
- `ALTER` schaltet ab, es erweitert nicht **– eine großzügige Berechtigung entfernt den Schutz**
- Policies machen aus den Schaltern der vorigen Einheiten **Regeln des Hauses**
- auf anderen Plattformen kann man es nur **organisieren**, nicht erzwingen

Lernziele durchgehen. Und damit endet die Gruppe Sicherheit: Krypto, OpenPGP, Betrieb, Durchsetzung. Der Bogen war vom Algorithmus über den Schlüssel zum Betrieb und zur Regel – und die Regel ist der einzige Teil, der nicht weggelassen werden kann.



Fünftes Funktionsthema, und das mit dem besten Verhältnis von Aufwand zu Nutzen: acht Kommandos, die im Alltag Zeit sparen und die kaum jemand kennt.
Nicht alle gleich tief behandeln. GREP und INFO sind die, die jeder braucht.

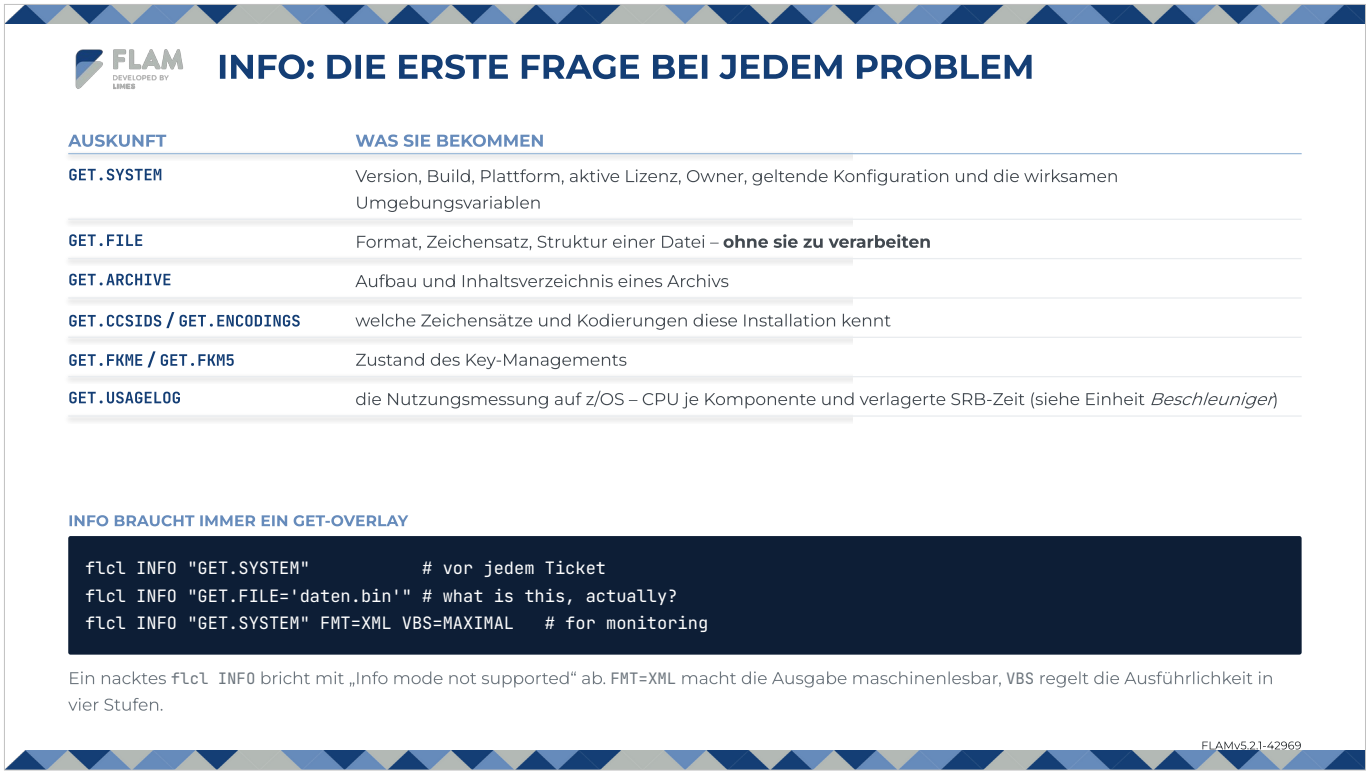


Ziel zwei ist das, was auf dem Host am meisten Zeit spart – und was viele nicht wissen, weil sie GREP unter z/OS nicht erwarten.

DIE ACHT IM ÜBERBLICK

KOMMANDO	WOFÜR
INFO	Auskunft über Installation, Lizenz, Konfiguration, Umgebung – und über eine Datei oder ein Archiv
GREP	Suche in Dateien und Archiven, mit Muster oder regulärem Ausdruck
HASH	Prüfsummen bilden und prüfen
DIFF	zwei Dateien oder Archive vergleichen
XCHK	Struktur prüfen – passt der Inhalt zur Formatbeschreibung
ICNV	Zeichensatzwandlung als eigenständiges Kommando
KEY	Schlüssel und Keysets verwalten
UTIL	Hilfsfunktionen rund um Dateien und Umgebung

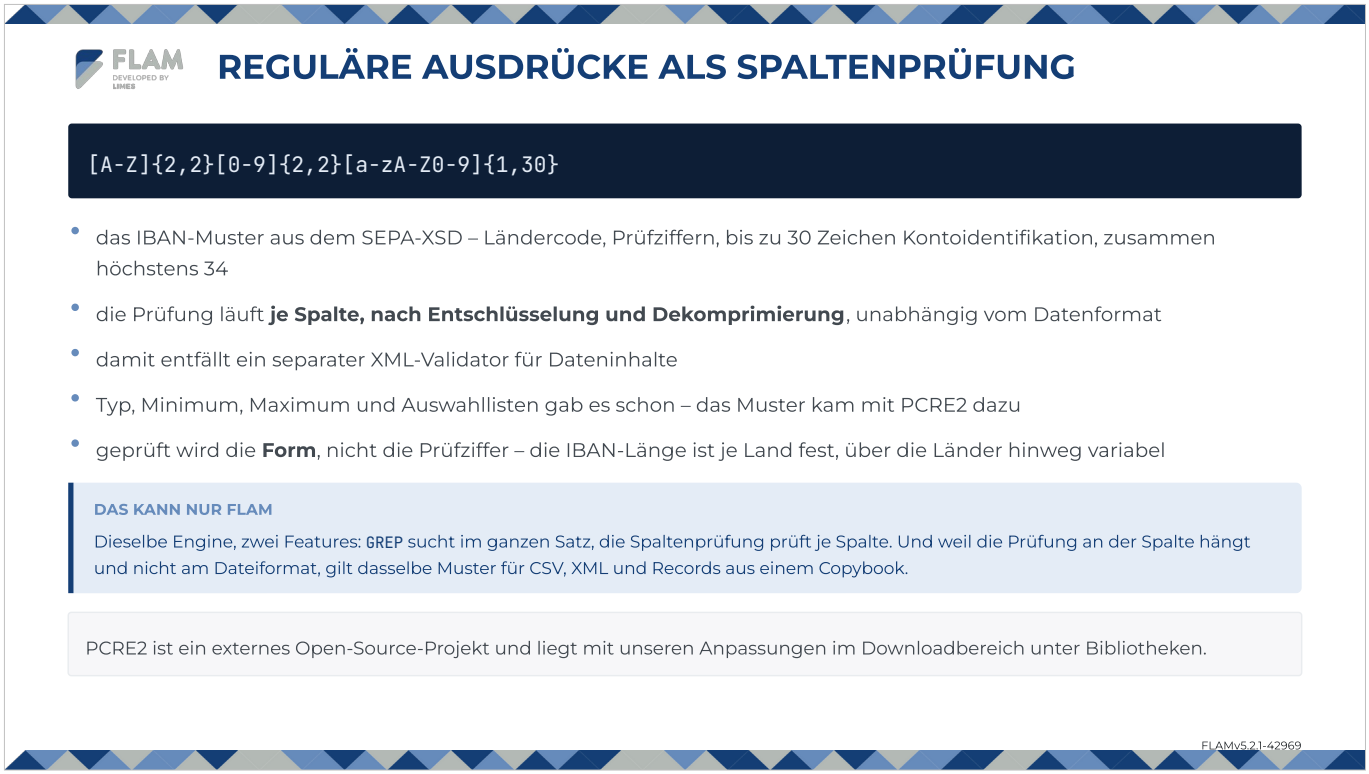
Einmal durchgehen, nicht vorlesen. Der Zweck der Folie ist, dass die Teilnehmer wissen, dass es diese Kommandos gibt – die Details holen sie sich mit `flcl HELP`.
Wenn die Zeit knapp ist: nur INFO und GREP behandeln, die anderen als Liste stehen lassen.



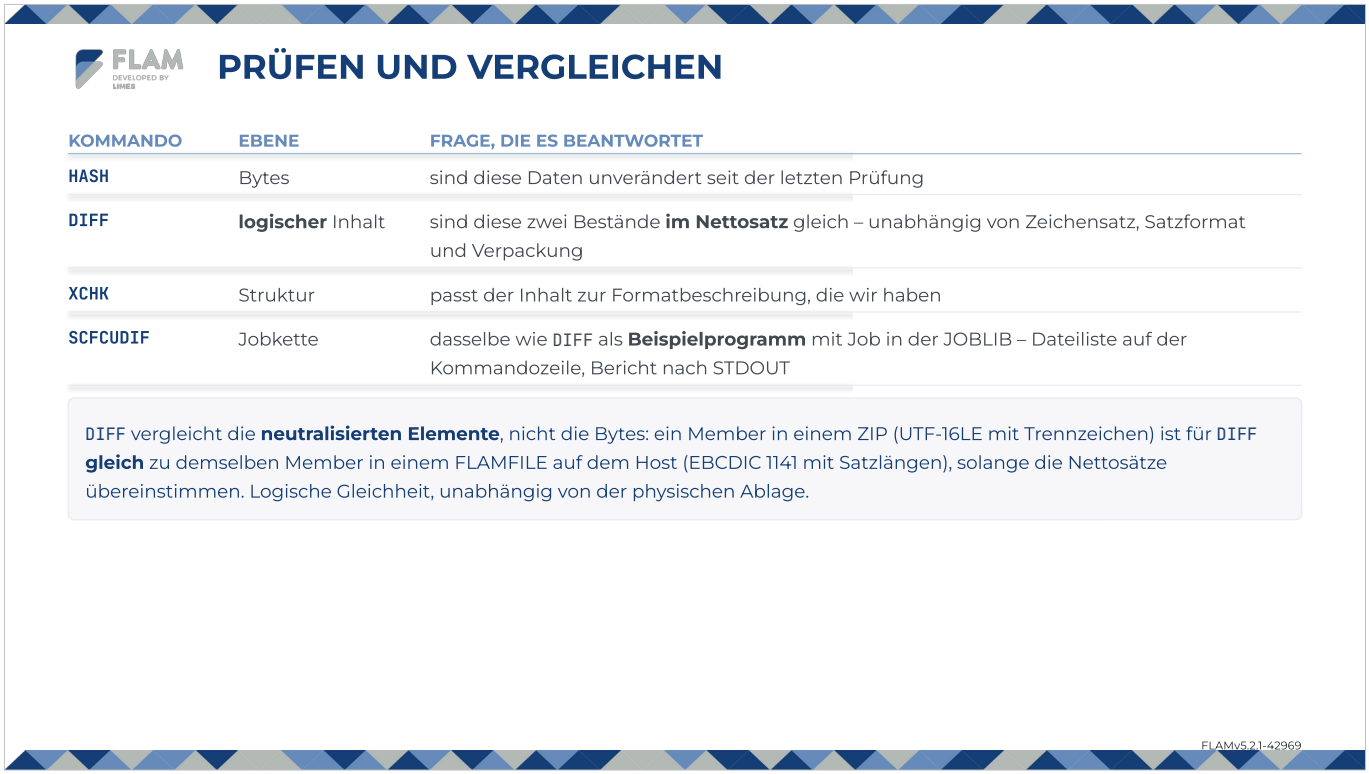
Das Kommando, das die erste Rückfragerunde in jedem Ticket erspart.
Wichtig zur Bedienung, weil es sonst gleich schiefgeht: `INFO` braucht immer ein `GET`-Overlay. Ein nacktes `flcl INFO` bricht mit "Info mode not supported" ab. Das ist kein Defekt, sondern die Sprache: `GET` ist ein Overlay, und ein Overlay muss ausgewählt werden.
Die sieben Auskünfte einmal durchgehen. Die zwei, die im Alltag überraschen, sind `GET.FILE` und `GET.ARCHIVE` – man kann eine Datei oder ein Archiv befragen, ohne sie zu verarbeiten.
`GET.USAGELOG` nur auf dem Host erwähnen und den Bogen zur Einheit über die Beschleuniger schlagen: dort liegen die Messwerte, mit denen man entscheidet, ob eine zIIP-Verlagerung etwas bringt.
Die Ausgabe steuert man über zwei Parameter, die man kennen sollte: `FMT=XML` macht sie maschinenlesbar, `VBS` regelt die Ausführlichkeit in vier Stufen.
Damit taugt `INFO` auch für die Überwachung.
Und der Satz für den Betrieb: vor jedem Ticket `flcl INFO GET.SYSTEM` ausführen und die Ausgabe mitschicken.



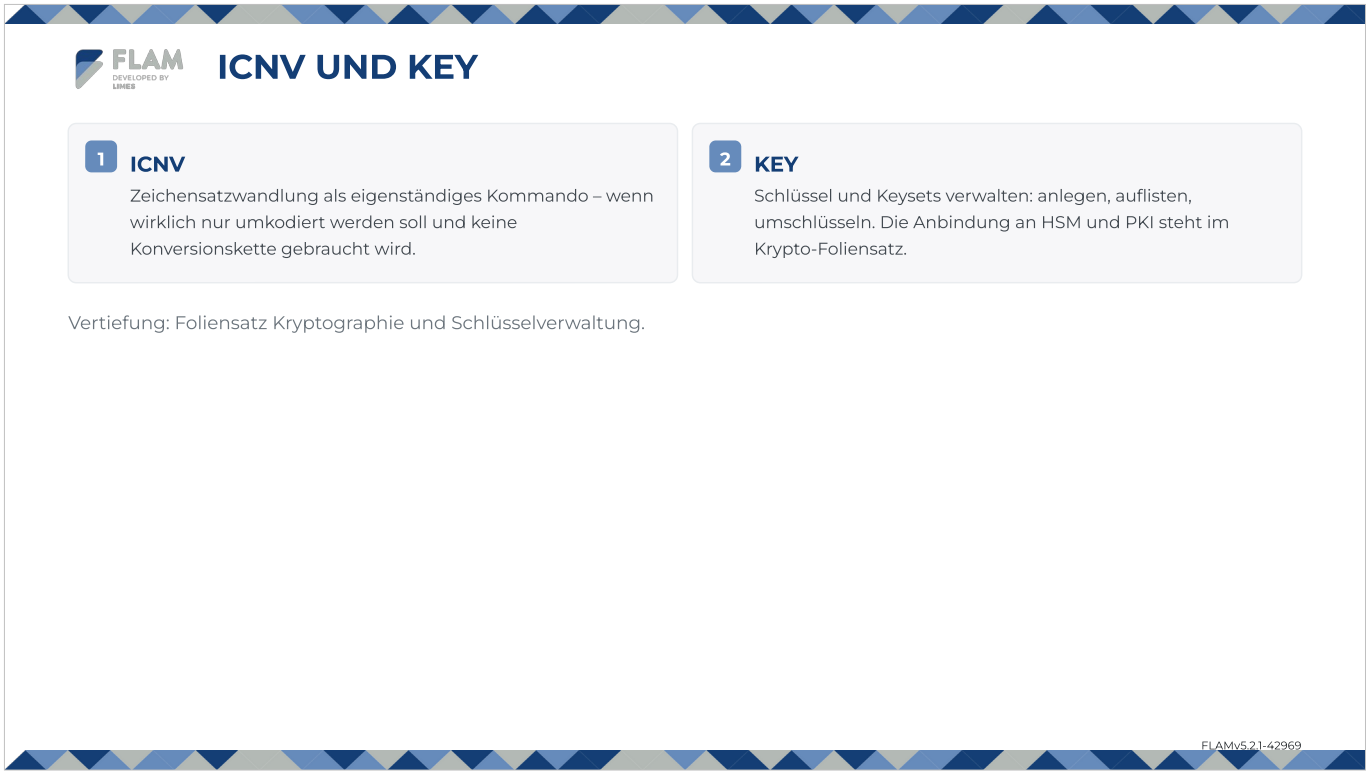
Die Überraschung dieser Einheit: es gibt grep unter z/OS, und es sucht auch in Archiven.
Der Mechanismus dahinter erklärt alles andere, und er lohnt sich: `GREP` ist eine Vereinfachung des Konvertierungsablaufs. Gelesen wird mit `READ.AUTO()` – deshalb werden komprimierte Dateien entpackt, verschlüsselte entschlüsselt, und Member von ZIP- und FLAM-Archiven sind über die Schalter im `DIR`-Objekt durchsuchbar. Nichts davon musste extra gebaut werden; es fällt aus der Architektur.
Das Zweite, was fast niemand kennt und was `GREP` von `grep` unterscheidet:
je Treffer entsteht eine **Zeile mit fünf Spalten** – PATTERN, FILE, LINE, COLUMN, DATA – und geschrieben wird mit `WRITE.TEXT()`. Deshalb gibt es `FORMAT=CSV`, `XML` und `JSON`: das Suchergebnis ist maschinenlesbar, ohne dass jemand die Ausgabe parsen muss. Genau das kann `grep` nicht.
Die Muster sind Perl-kompatibel, gesteuert über `NOCASE` und `EXTENDED`.
Und jetzt die Einschränkung, die auf die Folie gehört, weil sie im Handbuch ausdrücklich steht: gelesen wird **ohne** Tabelleninterpretation. Das Muster wird immer gegen den ganzen Satz geprüft, eine Row-Spec kann man hier nicht angeben, und einzelne Spalten werden **nicht** getrennt durchsucht. Die spaltenweise Prüfung ist die nächste Folie – ein anderes Feature, nur dieselbe Engine.
Kleinigkeiten, die im Support auftauchen: verglichen wird auf Basis von UTF-8 im neutralen Stringformat, nachlaufende Leerzeichen werden entfernt, und das Trennzeichen ist nicht Teil der verglichenen Daten.
Auf z/OS unter ISPF heißt das Kommando `FLGREP pattern dsn`.



Zuerst die Abgrenzung zur vorigen Folie, sonst verschwimmt es: `GREP` sucht, und zwar im ganzen Satz. **Hier** geht es um Prüfung, und zwar **je Spalte**.
Dieselbe Engine, zwei verschiedene Features – wer das verwechselt, sucht das Muster an der falschen Stelle.
Der Grund, warum PCRE2 überhaupt eingebaut wurde, war nicht die Suche, sondern genau diese Validierung: XML-Dokumente im Zahlungsverkehr, SEPA und SWIFT-MX, sollten ohne separaten XML-Validator prüfbar werden.
Das Beispiel ist das IBAN-Muster aus dem SEPA-XSD – Ländercode, zwei Prüfziffern, dann bis zu dreißig Zeichen Kontoidentifikation, zusammen also höchstens 34.
Hier lohnt eine Nachfrage im Raum, weil fast jeder es falsch erinnert: **die IBAN hat keine feste Länge.** Sie ist je Land fest – Deutschland 22, Malta 31 – aber über die Länder hinweg variabel. Genau deshalb steht im XSD eine Obergrenze und keine feste Zahl.
Und gleich die Grenze der Methode mitnennen, sonst verspricht die Folie zu viel: das Muster prüft die **Form**, nicht die Prüfziffern. Eine IBAN mit falscher Mod-97-Prüfziffer passt auf das Muster. Für die Formprüfung im Datenstrom ist das genau richtig – die fachliche Prüfung bleibt Aufgabe der Anwendung.
Der wichtige Punkt für die Sicherheit: die Prüfung passiert je Spalte, nach Entschlüsselung und Dekomprimierung, und unabhängig vom Datenformat. Sie greift also da, wo die Daten klar vorliegen – im Speicher. Und weil sie an der Spalte hängt und nicht am Dateiformat, gilt dieselbe Prüfung für CSV, für XML und für Records aus einem Copybook.



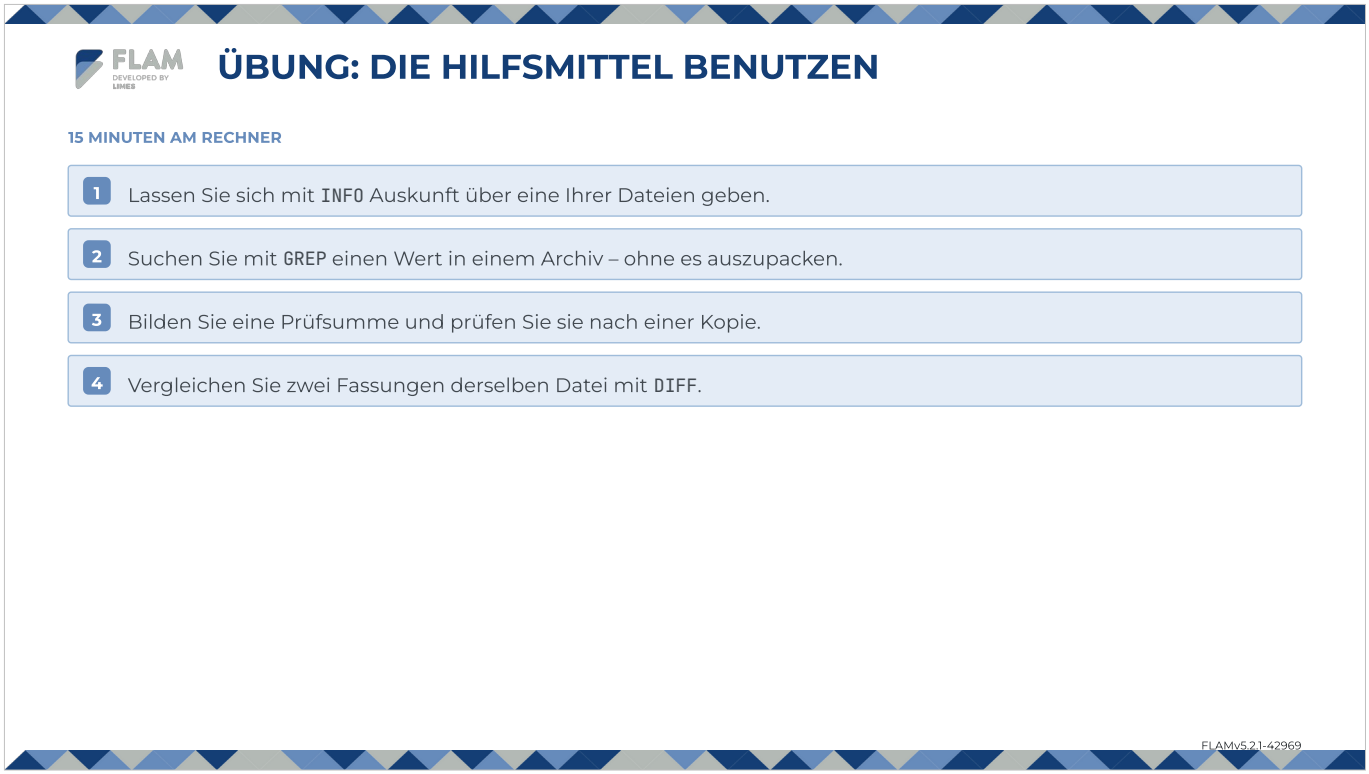
Drei Kommandos, die zusammengehören, weil sie alle eine Frage nach Gleichheit beantworten – auf verschiedenen Ebenen.
HASH ist die Byte-Ebene, DIFF die Inhaltsebene, XCHK die Strukturebene.
Und hier ist die Aussage, die `DIFF` erst wertvoll macht: **wir vergleichen die neutralisierten Elemente, nicht die Bytes.** Die Frage, die `DIFF` beantwortet, lautet also: sind diese beiden Bestände **logisch gleich**?
Das Beispiel, an dem es klickt: ein Member in einem ZIP-Archiv, dort in UTF-16LE mit Trennzeichen abgelegt, und dasselbe Member in einem FLAMFILE auf dem Host, dort in EBCDIC 1141 mit Satzlängen. Byteweise haben die zwei nichts miteinander zu tun. Für `DIFF` sind sie **gleich**, solange die Nettosätze übereinstimmen – die physische Repräsentation ist ihm gleichgültig. Genau das kann `HASH` nicht, und genau deshalb sind es drei Kommandos und nicht eines.
XCHK ist das, was in Migrationen zählt: passt der Inhalt überhaupt zur Beschreibung, die wir haben.
Und ein Hinweis für den Host, der in der Handbuchseite steht und selten bekannt ist: es gibt **`SCFCUDIF`** als Beispielprogramm in der Auslieferung, mit einem Job in der JOBLIB. Es nimmt eine Liste von Dateinamen von der Kommandozeile, vergleicht die erste gegen alle weiteren und schreibt den Bericht nach STDOUT – verglichen werden die **Nettodaten** nach Dekodierung, Entschlüsselung, Dekompression und Zeichenwandlung, nicht die Bytes. Vier Optionen: `-f` vollständig vergleichen, `-o` Byteoffsets in den Bericht, `-m` weniger Protokoll, `-r` der erste Name ist die Berichtsdatei. Für ein Haus, das einen SuperC-artigen Vergleich in einer Jobkette braucht, ist das die Antwort, ohne ein Kommando zu lernen.



Zwei Kommandos für Sonderfälle, aber beide werden regelmäßig gebraucht.
ICNV ist die Zeichensatzwandlung als eigenes Kommando – für die Fälle, in denen man wirklich nur umkodieren will und keine Kette braucht.
KEY ist das Werkzeug für die Schlüsselverwaltung außerhalb eines Laufs.
Details stehen im Krypto-Deck; hier nur, dass es das Kommando gibt.



Drei aus Supportfällen, eine aus der Erfahrung mit regulären Ausdrücken.



Fünfzehn Minuten. Aufgabe zwei ist die mit dem größten Aha-Effekt: in einem Archiv suchen, ohne es auszupacken.
Auflösung:
1. `flcl INFO GET.FILE='...'` sagt, was eine Datei ist, ohne sie zu verarbeiten – Format, Zeichensatz, Struktur.
2. `flcl GREP input='...' pattern='...'` findet die Treffer. Mit `FORMAT=CSV` wird das Ergebnis maschinenlesbar; das ist die Antwort auf "und wie werte ich das aus".
3. `flcl HASH` mit `ALGO=SHA256` erzeugt die Prüfsumme, `CHECK` prüft sie gegen eine gespeicherte. `FORMAT=GNU` ergibt eine Datei, die `sha256sum -c` liest – praktisch, wenn die Gegenseite kein FLAM hat.
4. Die Ebenen auseinanderhalten: `HASH` beantwortet "sind es dieselben Bytes", `DIFF` beantwortet "wo unterscheiden sich die Inhalte", `XCHK` beantwortet "passt der Inhalt zur Formatbeschreibung". Zwei Dateien können inhaltlich gleich und byteweise verschieden sein – etwa nach einer Zeichensatzwandlung. Genau das ist die Falle, in die man ohne diese Unterscheidung läuft.

WAS BLEIBT

- `INFO` ist die erste Frage bei jedem Problem – und die erste Zeile jedes Tickets
- `GREP` sucht auch auf dem Host und auch in Archiven, mit PCRE2
- reguläre Ausdrücke prüfen Spalten nach der Entschlüsselung – ohne separaten Validator
- HASH, DIFF und XCHK beantworten Gleichheit auf drei verschiedenen Ebenen
- ICNV und KEY sind die Werkzeuge für Umkodierung und Schlüsselverwaltung

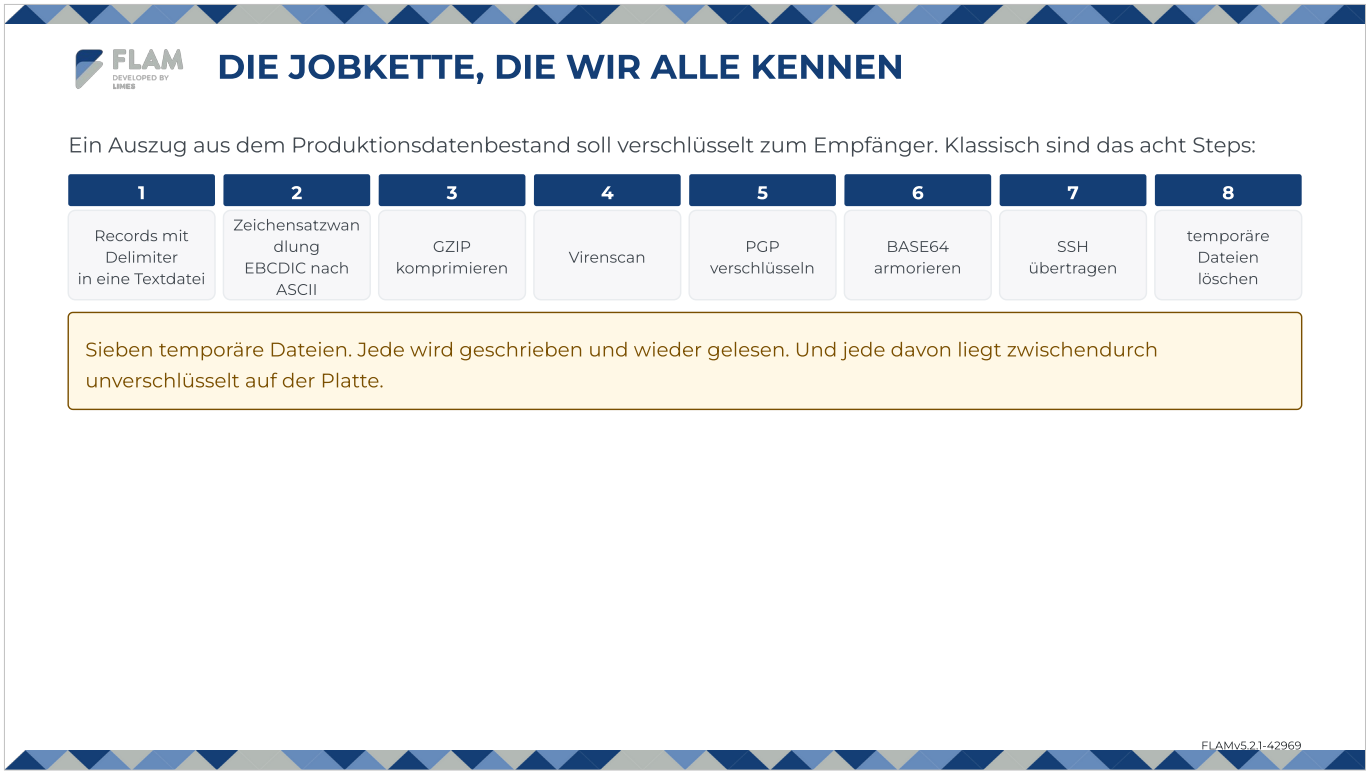
Lernziele durchgehen. Damit sind die Funktionsthemen für diese Schulung abgedeckt.



Erste Einheit am Donnerstag. Das ist die Einheit, in der die Teilnehmer den größten unmittelbaren Nutzen mitnehmen, weil jeder von ihnen Jobketten betreut, die genau so aussehen wie die auf Folie 3. Einstieg: nicht mit FLAM anfangen, sondern mit ihrer Jobkette. Fragen, wie viele Steps ihre längste Kette hat – die Antwort ist die Motivation für die ganze Einheit.



Ziel 3 ist das eigentliche Aha-Erlebnis dieser Einheit. Ziel 5 ist das, was im Betrieb den Unterschied zwischen einem sauberen und einem störenden Job macht.



Diese Kette langsam vorlesen und dabei mitzählen. Acht Steps, sieben temporäre Dateien.
Genau diese Kette ist der Ausgangspunkt für alles, was danach kommt. Wer sie im eigenen Haus wiedererkennt – und das tun alle – hört den Rest der Einheit anders zu.
Wichtig: das ist keine schlecht gebaute Kette. Sie ist so gebaut, weil jedes Werkzeug genau eine Sache kann.

WAS DIESE KETTE KOSTET

POSTEN	WARUM ER ANFÄLLT
I/O (EXCP)	sieben temporäre Dateien, je einmal geschrieben und einmal gelesen – der größte einzelne Posten der Kette
Werkzeug-Overhead	acht Programme werden geladen, initialisiert und beendet; jedes liest und schreibt seine eigenen Puffer
interpretierter Code	Delimiter und Feldumformung als REXX-Routine – gegen Massendaten die teuerste Variante
Klartext auf Platte	die Zwischenstände liegen unverschlüsselt im Dateisystem, bis Step 8 sie löscht – falls er läuft
Betriebsaufwand	acht Returncodes, acht Fehlerquellen, acht Stellen für den Restart

FLAMv5.21-42969

Der Hauptposten ist der I/O, nicht die Rechenzeit der einzelnen Werkzeuge.
Jede temporäre Datei wird einmal geschrieben und einmal gelesen – das sind EXCPs, und die sind auf dem Host das teuerste an der Kette.
Der zweite Posten sind die REXX-Routinen. Delimiter setzen, Felder umformen, Satzarten unterscheiden: das ist typischerweise selbstgeschriebenes REXX, und interpretierter Code gegen Massendaten ist die teuerste Form, das zu tun.
Der dritte Posten steht nicht in der Rechnung, sondern im Auditbericht.



Das ist die Kernfolie. Nicht über die Syntax reden, sondern über die Struktur: ein READ, ein WRITE, und alles dazwischen sind Konvertierungsstufen im Speicher.
Der wichtige Satz: es gibt keine temporäre Datei mehr. Die Daten verlassen den Speicher genau einmal – am Ende, auf der Leitung.
Auf die Rückwärtsverbindung zur Einheit Konvertierung verweisen: das sind dieselben Konvertierungsstufen, nur diesmal in JCL.
Zwei Parameter im Beispiel verdienen einen Satz, weil sie oft vergessen werden und der Empfänger dann eine unbrauchbare Datei bekommt.
`CCSID=1208` ist UTF-8. Und `METHOD=UNIX` setzt `0x0A` als Trennzeichen – ohne das schreibt FLAM das Trennzeichen der Quellplattform, und auf dem Host ist das nicht `0x0A`. Wer nach Unix liefert und `METHOD` vergisst, liefert Zeilen, die dort niemand trennen kann.
Auf der Leseseite steht bewusst keine `CCSID`: dort arbeitet der Detektor.
Er entscheidet ASCII oder EBCDIC, und welche Codepage das dann konkret ist, sagt die Umgebung – die Default-CCSID für ASCII beziehungsweise EBCDIC.
Gesetzt wird sie entweder direkt oder über `LANG`; aus `de_DE` wird auf dem Host IBM-1141. Das ist der Grund, warum `LANG` auf Folie 10 in der Umgebung des Steps steht.
Die Verschlüsselung steckt in `ENCRYPT.PGP()`, und `ARMOR()` ist ein Objekt **darin** – nicht daneben. Das ist die Struktur der Sprache: was zu PGP gehört, steht in PGP.
Ein Parameter darin ist Pflicht, und in Beispielen fehlt er gern:
`USERID` – die User-ID des Empfängers. PGP verschlüsselt für einen Empfänger, und der ist nicht vorbelegt; die eigene User-ID ist ausdrücklich **nicht** der Standard. Wer symmetrisch arbeiten will, gibt statt dessen `PASSWORD` an, und wer zusätzlich signieren will, `SIGNID` – das ist dann die eigene User-ID.
Die Kette oben ist absichtlich vollständig, damit die Reihenfolge sichtbar wird. `CNV.BLK` macht aus den Sätzen Blöcke mit Trennzeichen – das muss **vor** der Zeichenwandlung passieren. Dann `CNV.CHR` für den Zeichensatz.
Dann `CNV.AVS`, der Virenscan: er läuft auf dem **lesbaren** Text, also bevor komprimiert und verschlüsselt wird. Danach kann kein Scanner mehr etwas finden, weder bei uns noch beim Empfänger. Im Parameterstring ist das `AVSCAN()` am `WRITE`. Erst danach `CNV.GZP`, `CNV.PGP` und `CNV.BAS`.
Was man auf Nachfrage noch zeigen kann, ohne die Folie zu überladen: der Lauf kann zusätzlich eine **Prüfsumme** oder eine **abgetrennte Signatur** erzeugen, und über das Nachbearbeiten (`POST`) lässt sich auf der Gegenseite etwas anstoßen – entweder direkt die Verarbeitung, oder einfach ein `RENAME`, damit ein Cronjob dort die Datei aufgreift. Damit endet die Kette nicht bei der Übertragung, sondern beim Verarbeitungsstart auf dem Zielsystem.



Das ist der Punkt, an dem die Rechnung endgültig zugunsten von FLAM ausgeht – und der Punkt, den fast niemand kennt.
Der Lesevorgang und alles, was allen Ausgaben gemeinsam ist, passiert genau einmal. Erst danach verzweigt es sich. Die Rechnung ist einfach und man kann sie im Raum vorrechnen: drei getrennte Läufe lesen dreimal, ein Lauf mit drei Writes liest einmal – das sind zwei Drittel der Lesearbeit, die wegfallen.
Wichtig ist die Einschränkung, sonst verspricht man zu viel: geschrieben wird weiterhin dreimal, und die Konvertierung fällt nur insoweit weg, wie sie allen Ausgaben gemeinsam ist. Wieviel das an Elapsed Time bringt, hängt vom Verhältnis von Lese- zu Schreibarbeit ab – auf dem Host, wo der EXCP der teure Posten ist, ist es viel.
Das Beispiel ist echt: derselbe Datenbestand geht als UTF-8 an den Empfänger, als CP1252 mit CRLF in ein ZIP für die Windows-Auswertung, und als verschlüsseltes FLAMFILE auf die Platte, damit die Host-Anwendung die Records später wieder eins zu eins lesen kann. Letzteres ist der PCI-DSS-Punkt.
Verfügbar in allen drei Zugangswegen: Batch, API und Subsystem.

FLAM
DEVELOPED BY LIMES

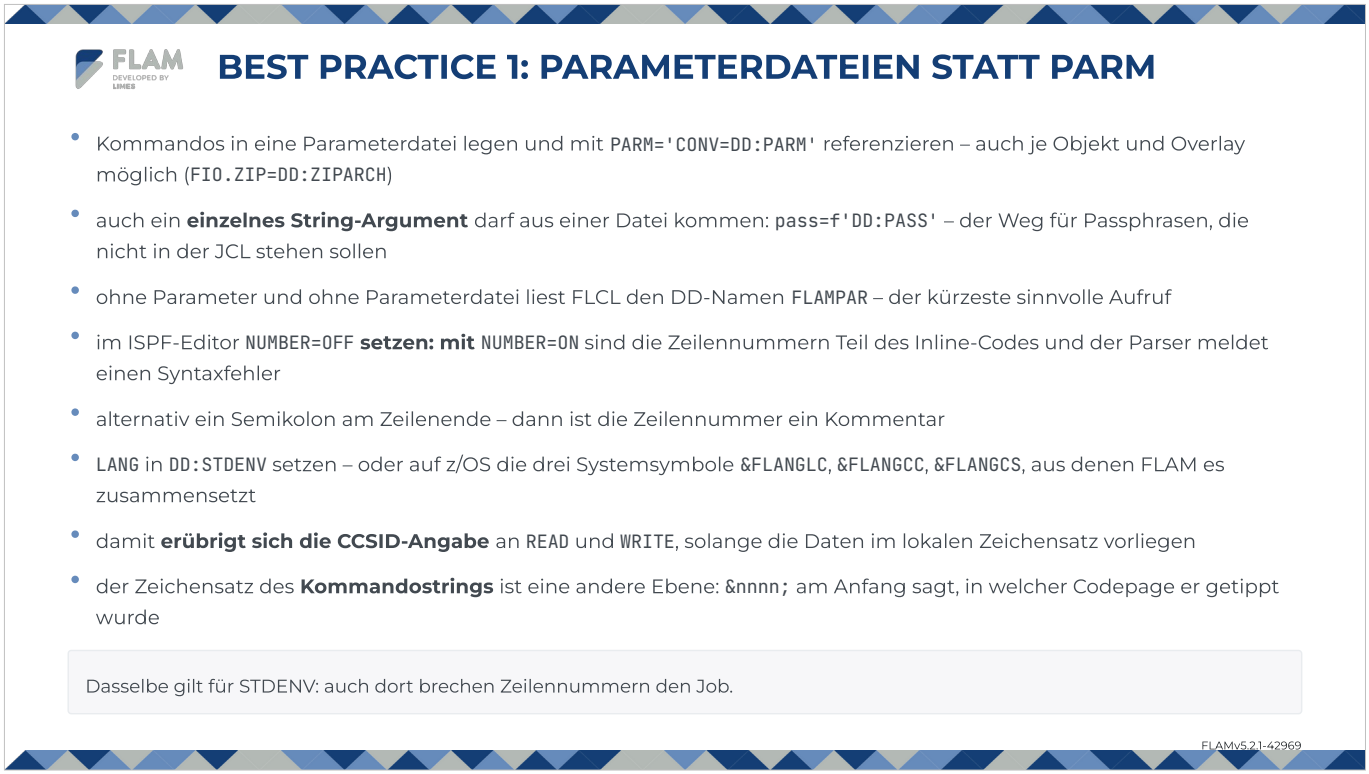
MULTIPLE I/O: EIN LAUF, MEHRERE ZIELE

	MULTIPLE WRITES	MULTIPLE I/O
was sich unterscheidet	die Konvertierung je Ausgabe	nur das Ziel
wie oft konvertiert wird	je Ausgabe einmal	genau einmal
typischer Fall	UTF-8 für den Partner, CP1252 für die Auswertung	dieselbe Lieferung an zwei Empfänger, oder Platte plus Bandkopie

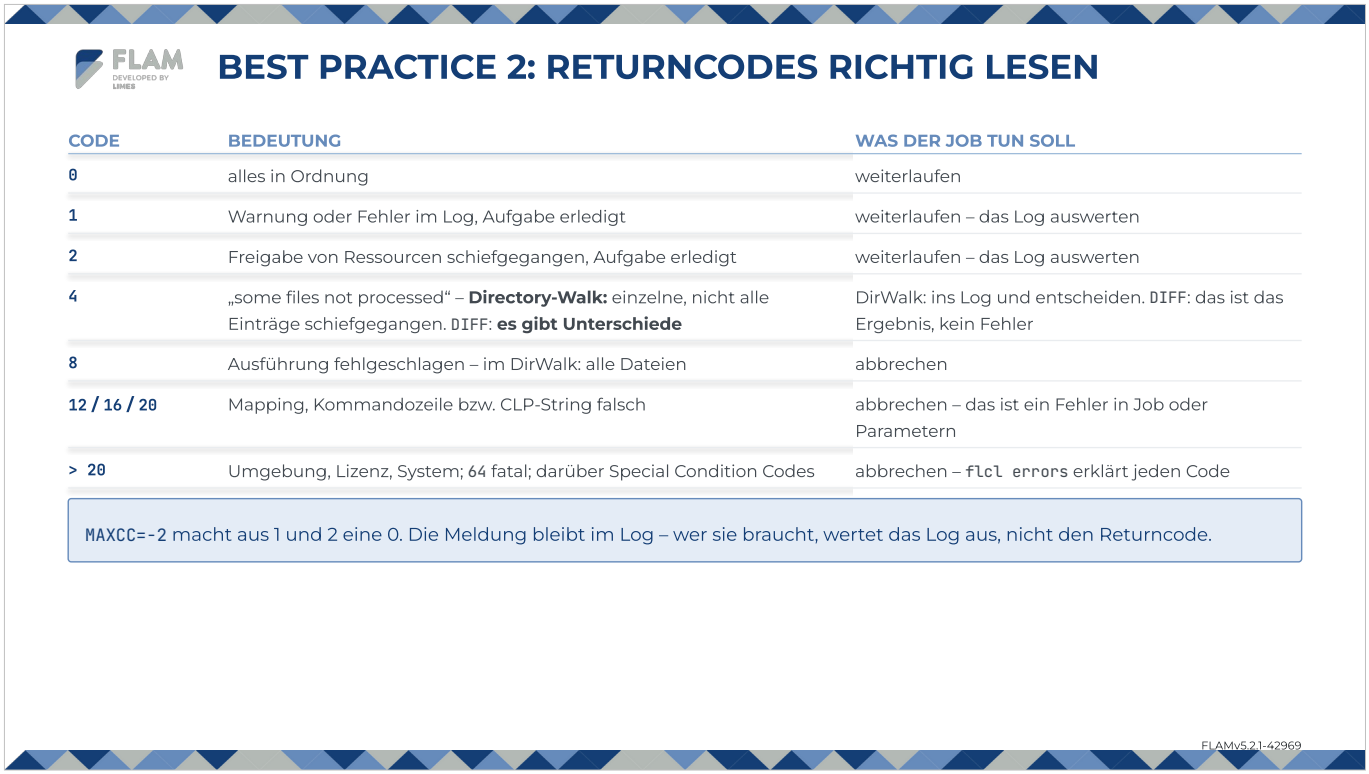
Beides ist kombinierbar: mehrere Writes, und einzelne davon schreiben ihr Ergebnis an mehrere Ziele. Wer alle Möglichkeiten nutzt, spart am meisten.

FLAMv5.21-42969

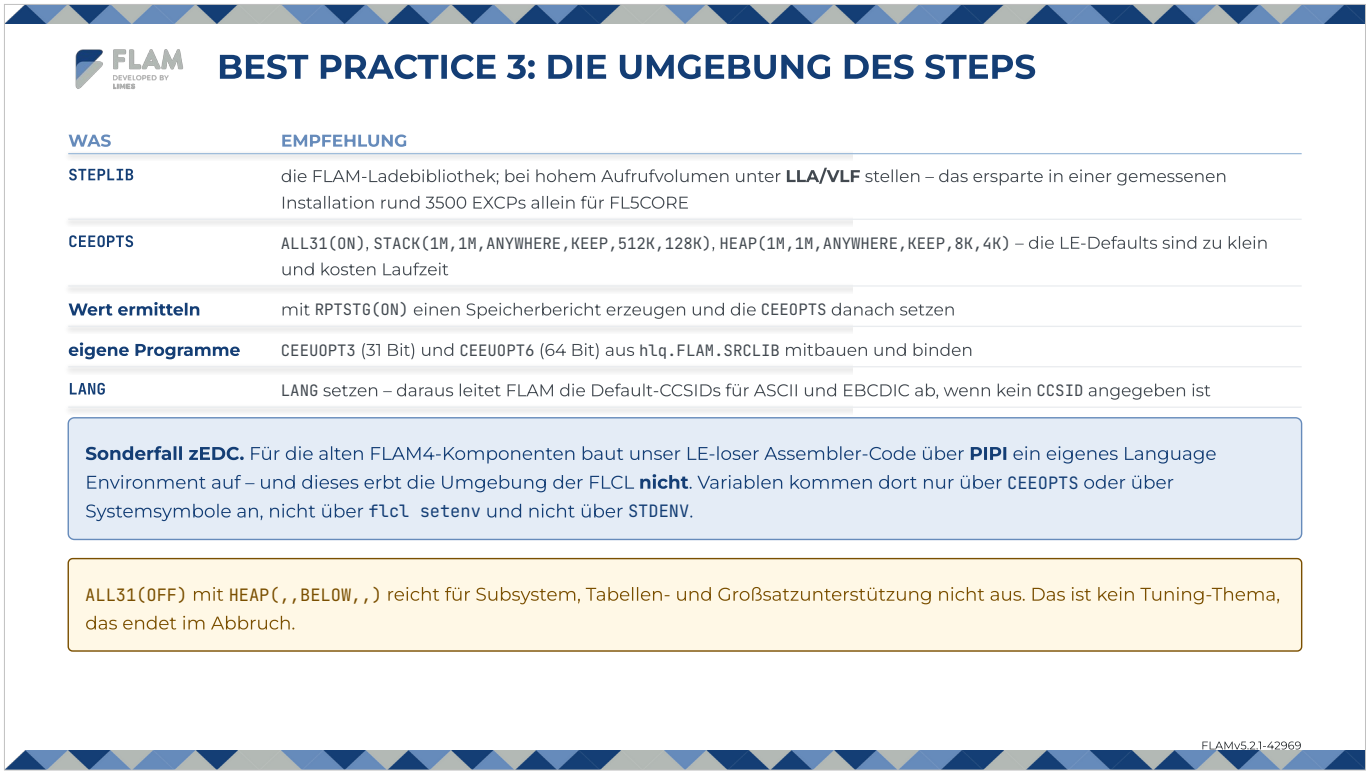
Der Unterschied zu Multiple Writes klar herausarbeiten – das ist die häufige Verwechslung.
Multiple Writes = verschiedene Ergebnisse, jedes mit eigener Konvertierung.
Multiple I/O = ein Ergebnis, mehrere Ziele; konvertiert wird genau einmal.
Und man kann beides kombinieren: drei Writes, und der zweite davon geht an zwei Ziele.



Praktischer Teil, und zuerst eine Klarstellung, weil die Frage nach der Länge des Parameterstrings regelmäßig kommt: **von unserer Seite gibt es keine Grenze.** FLAM5 verarbeitet einen beliebig langen Parameterstring, FLAM4 nimmt, was z/OS maximal übergibt. Die alte Grenze von 100 Zeichen ist Geschichte, und mit `PARMDD` gibt es einen zweiten Weg, den String aus einer DD-Karte zu lesen.
Der Grund für eine Parameterdatei ist deshalb nicht die Länge, sondern:
Lesbarkeit – ein Aufruf über mehrere Zeilen mit Einrückung ist im Review lesbar, ein PARM-String nicht. Wiederverwendung – dieselbe Datei aus mehreren Jobs. Und die Hochkommata, die im PARM verdoppelt werden müssen.
Die drei Fallen in der Reihenfolge, in der sie auftreten: NUMBER=ON im ISPF- Editor, fehlendes LANG, und das Vergessen von MAXCC.
NUMBER=ON ist die häufigste Supportanfrage zu diesem Thema überhaupt. Die Zeilennummern stehen dann mit im Inline-Code und der Parser meldet einen Syntaxfehler an einer Stelle, die im Editor völlig korrekt aussieht.
Zu `LANG` gehören zwei Dinge, die man auseinanderhalten muss – dieselbe Ebenentrennung wie in der Einheit *Zeichensätze*:
`LANG` gilt für die Daten. Es sagt, in welchem Zeichensatz die Massendaten lokal vorliegen, und erspart damit die CCSID-Angabe an `READ` und `WRITE`. Auf z/OS setzt man es über `DD:STDENV` oder über die drei Systemsymbole `&FLANGLC`, `&FLANGCC` und `&FLANGCS`, aus denen FLAM den Wert zusammenbaut.
Der Kommandostring ist eine andere Ebene. Die Falschinterpretation von Tilde und eckigen Klammern betrifft den Parameterstring, den wir einlesen – nicht die Daten. Wer in einer 3270 editiert, kann mit `&nnnn;` am Anfang sagen, in welcher Codepage getippt wurde; ohne diese Angabe gilt `LANG`.
Und der Grund, warum das auf dem Host überhaupt auseinanderfallen kann:
JCL muss eigentlich in 1140 geschrieben sein, weil JES nur 1140 versteht – die Daten auf einem deutschen System sind aber oft 1141.
In der Praxis geht das meist gut, und man sollte auch sagen warum: die meisten deutschen Arbeitsplätze haben 1141 in der 3270, und solange die JCL selbst ohne kritische Zeichen auskommt – was normale JCL tut –, klappt der Submit, der Inline-Code ist in 1141 geschrieben und wird mit 1141 gelesen.
Es geht also nur deshalb gut, weil JCL die diakritischen Zeichen nicht braucht. Sobald im Parameterstring eine Klammer oder eine Tilde steht, ist `&nnnn;` der saubere Weg.



Das ist die Folie, die einen unnötigen Nachtalarm verhindert.
Returncode 1 heißt: es steht eine Warnung oder ein Fehler im Log, aber die Aufgabe ist erledigt.
Returncode 2 heißt: beim Freigeben von Ressourcen ist etwas schiefgegangen, die Aufgabe ist erledigt. In beiden Fällen darf die Folgeverarbeitung laufen.
Zur `2` gehört eine Einordnung, falls sie doch einmal kommt: sie bedeutet, dass wir im Abschluss – `FLUC`/`FLAMFIN` – nicht fehlerfrei durchgekommen sind, also möglicherweise **nicht alle Ressourcen freigegeben** wurden, etwa eine Datei nicht geschlossen. Im Log ist dazu wenig zu erwarten, weil die Protokollierung im Abschluss nicht mehr sicher zur Verfügung steht. Das Ergebnis der Aufgabe ist davon nicht betroffen, und in der Praxis ist die `2` bisher **nicht aufgetreten**.
MAXCC=-2 macht daraus eine 0. Das ist keine Schönfärberei – das Log enthält die Meldung weiterhin, und wer sie auswerten will, wertet das Log aus, nicht den Returncode.
Jetzt der Punkt, der in der Praxis am meisten Ärger macht – die `4`. Sie bedeutet laut `flcl errors` wörtlich "some files not processed", und das setzt mehr als eine Datei voraus. Im Code gibt es genau zwei Stellen, die sie erzeugen:
Erstens der Directory-Walk beziehungsweise eine Memberliste: einige, aber nicht alle Einträge sind schiefgegangen – "There was an error at n of m members". Typisch sind fehlende Zugriffsrechte auf eine einzelne Datei.
Sind **alle** schiefgegangen, kommt eine `8`. Dazu zählt auch der Fall, dass der Verzeichnisdurchlauf selbst warnt, etwa bei einem nicht lesbaren Unterverzeichnis.
Zweitens `DIFF`: findet der Vergleich Unterschiede, ist das Ergebnis eine `4`. Hier ist die `4` kein Mangel, sondern **die Antwort** – ein Job, der 4 pauschal als Fehler behandelt, kann mit `DIFF` nicht arbeiten.
Bei der Verarbeitung **einer** Datei kommt keine `4`: eine Warnung im Log ergibt `1`, ein echter Fehler `8`. Wer bei einer Einzeldatei eine `4` sieht, hat entweder einen Dirwalk (Wildcards im Namen) oder ein `DIFF`.
Darüber wird es systematisch: `12` heißt, das Mapping der Parameter ist gescheitert – da war etwas im System nicht da. `16` heißt Syntaxfehler in der Kommandozeile, `20` Syntaxfehler im CLP-String selbst. Ab `24` sind es Umgebungs- und Systemfehler, `40` ist Zugriffskontrolle oder Lizenz, `64` ist der fatale Fehler. Über `64` liegen die Special Condition Codes, die ein Kommando selbst für die Jobsteuerung setzt.
Die vollständige Liste holt sich jeder selbst mit `flcl errors` – sie steht auch im Handbuch. Auf der Folie stehen die, die in einer Jobkette wirklich vorkommen.



Das ist die Folie für den Systemprogrammierer im Raum.
CEEOPTS: die LE-Standardinkremente sind für FLAM zu klein. Bei 128 Byte Inkrement steigen Laufzeit und CPU-Zeit erheblich, weil das Language Environment andauernd nachlaedt. 1 MiB für Stack und Heap sind der Ausgangswert; den wirklich passenden Wert liefert RPTSTG(ON).
Seit 5.1.25 bringt FLAM eigene Defaults über CEEUOPT mit – CEEUOPT3 für 31 Bit, CEEUOPT6 für 64 Bit, beide in hlq.FLAM.SRCLIB. Wer eigene Anwendungen gegen unsere APIs baut, sollte die Beispiele mitbauen und binden.
ALL31(OFF) mit HEAP BELOW reicht für Subsystem, Tabellen- und Großsatzunterstützung nicht – das ist kein Tuning, das ist ein Abbruch.
Ein Sonderfall, der Zeit kostet, wenn man ihn nicht kennt: **die alten FLAM4-Komponenten mit zEDC.** Unser LE-loser Assembler-Code stellt für die zEDC beziehungsweise die IBM-zlib über **PIPI** eigens ein Language Environment hin – und die Umgebung der FLCL wird an diese Umgebung **nicht vererbt**. Gelesen wird dort nur `CEEOPTS`. Wer also Variablen für die alten FLAM4-Komponenten setzen will, solange diese über PIPI mit der LE arbeiten (derzeit nur bei zEDC), kommt nur über `CEEOPTS` oder über Systemsymbole dorthin – nicht über `flcl setenv` und nicht über `STDENV`.
Das Laden der Module ist der eine Posten, den ein System vollständig entfernen kann: die FLAM-Ladebibliotheken unter LLA/VLF. In einer gemessenen Installation waren das rund 3500 EXCPs allein für FL5CORE. Aber: eine mit FREEZE aufgenommene Bibliothek liefert die von LLA gecachte Kopie – vor der Wartung muss sie aus der LLA-Kontrolle heraus und danach wieder hinein.

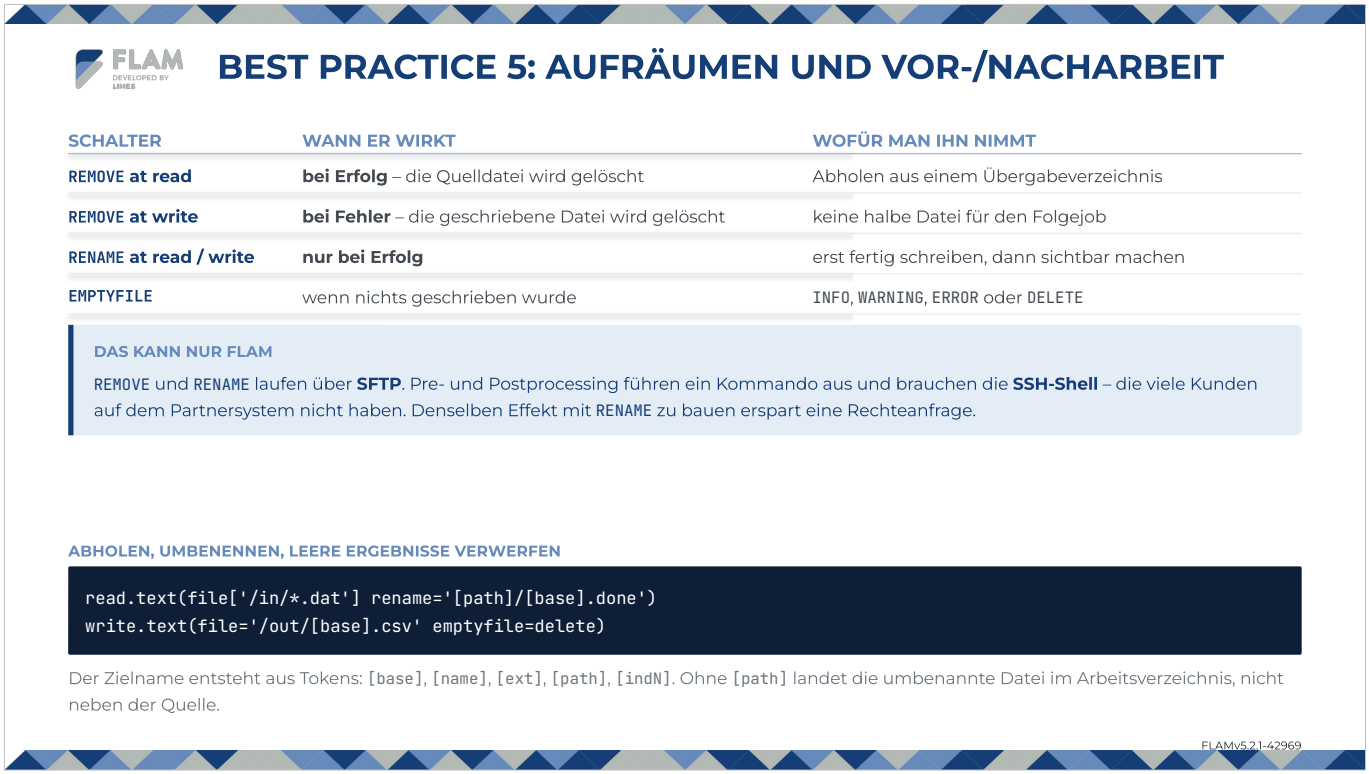
BEST PRACTICE 4: JCL-SYMBOLE IN DEN KOMMANDOS

```
//E0       EXPORT SYMLIST=(CHAROUT,HASHALGO)
//S1       SET    CHAROUT=IBM1141
//CONV     EXEC   PGM=FLCL,PARM='CONV=DD:PARM'
//PARM     DD     *,SYMBOLS=JCLONLY
   WRITE.TEXT(FILE='/u/<cuser>/export.gz' COMP.GZIP();
              HASH(CCSID=<CHAROUT> ALGO=&HASHALGO.));
```

- `SYMLIST=*` exportiert alle Symbole – damit werden JCL-Prozeduren wirklich wiederverwendbar
- Priorität: **Umgebungsvariable** vor **JCL-Symbol** vor **Systemsymbol** (`<&LDAY>`)
- in einer **externen** Parameterdatei gilt nur `<name>`; `&name.` ist eine JES-Ersetzung und geht nur inline
- steht das Language Environment nicht in der `STEPLIB`, ist `CEEGTJS` nicht verfügbar und die Symbole werden nicht ersetzt

FLAMv5.21-42969

Der Weg zu wiederverwendbaren Prozeduren. Seit z/OS 2.1 kann FLAM über CEEGTJS die Werte exportierter JCL-Symbole ermitteln und wie Umgebungsvariablen einsetzen. Das Language Environment muss dafür in der STEPLIB-Verkettung stehen.
Die Priorität ist wichtig, weil sie beim Debuggen den Unterschied macht: Umgebungsvariable schlägt JCL-Symbol schlägt Systemsymbol.
Und die Syntaxfalle: in einer externen Parameterdatei gilt nur die spitze-Klammer-Form. Das kaufmännische Und mit Punkt ist eine Ersetzung von JES und funktioniert nur bei Inline-Coding.



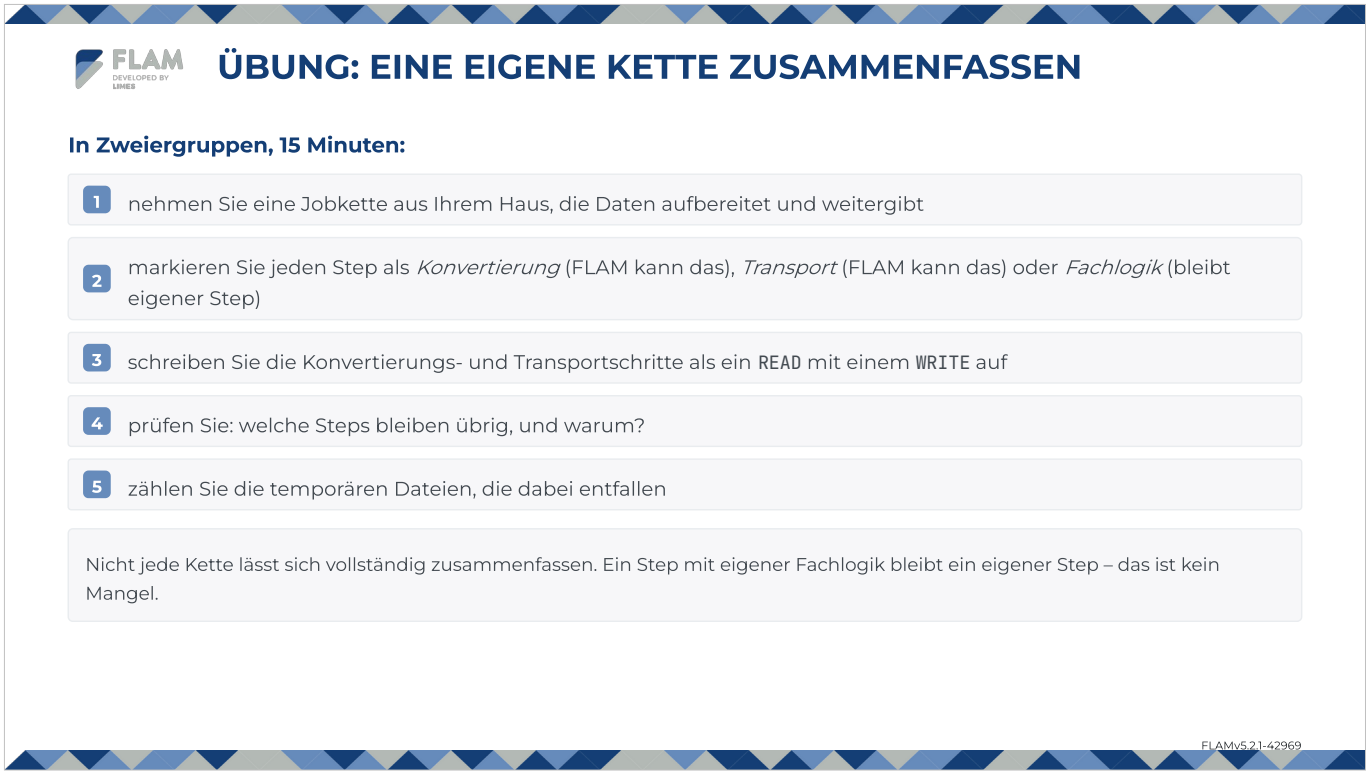
Diese Folie beantwortet eine Frage, die in jeder Einführung kommt: wer räumt die Zwischen- und Quelldateien weg?
Die vier Schalter sehen ähnlich aus, tun aber Verschiedenes – und einer von ihnen ist gefährlich, wenn man den Directory-Walk nicht mitdenkt.
`REMOVE` **auf der Leseseite** löscht die Quelldatei, wenn sie erfolgreich verarbeitet wurde. Genau das will man beim Abholen von einem Übergabeverzeichnis: was geholt ist, ist weg, und ein Wiederanlauf holt es nicht doppelt.
Und hier die Falle: **im Directory-Walk ist bei einem Returncode 4 das Original der erfolgreichen Dateien schon gelöscht.** Wer den Lauf dann wiederholt, überführt genau die nicht nochmal – was richtig ist, aber man muss es wissen, sonst sucht man die Dateien. Der Wiederanlauf arbeitet also den Rest ab, nicht alles.
`REMOVE` **auf der Schreibseite** ist das Gegenteil: es löscht die geschriebene Datei, **wenn es einen Fehler gab** – damit keine unvollständige Datei stehen bleibt, die ein Folgejob für fertig hält.
`RENAME` gibt es auf beiden Seiten und wirkt immer nur bei Erfolg. Das ist das übliche Muster für Übergabeverzeichnisse: erst unter einem Arbeitsnamen schreiben, dann umbenennen – der Empfänger sieht die Datei erst, wenn sie vollständig ist.
`EMPTYFILE` regelt, was mit leeren Ergebnissen passiert: nur ins Log (`INFO`), als Warnung (`WARNING`), als Fehler mit eigenem Condition Code (`ERROR`) oder gar nicht anlegen (`DELETE`).
Der letzte Punkt ist der wichtigste für die Praxis und der Grund, warum die Folie existiert: **`REMOVE` und `RENAME` gehen über SFTP.** Pre- und Postprocessing führen ein Kommando aus und brauchen deshalb die SSH-Shell.
Viele Kunden haben auf dem Partnersystem SFTP-Rechte, aber keine Shell-Rechte. Wer das vorher weiß, baut den Ablauf mit `RENAME` statt mit einem Postprocessing-Skript – und spart sich eine Rechteanfrage, die oft Wochen dauert und manchmal abgelehnt wird.
Noch eine Sache zum Zielnamen, weil sie beim ersten Versuch schiefgeht: der Name entsteht aus **Tokens in eckigen Klammern** – `[base]` ist der Name ohne Endung, `[name]` mit, `[ext]` nur die Endung, `[path]` das Verzeichnis, `[indN]` ein laufender Index. Dazu gibt es eine eigene Handbuchseite.
Und die Falle: `rename='[base].done'` legt die Datei im **Arbeitsverzeichnis** ab, nicht neben der Quelle – dafür braucht es `rename='[path]/[base].done'`.



Hier die Zahlen, aber mit der richtigen Begründung. Wer die Zahl ohne die Begründung nennt, wird zu Recht nicht geglaubt.
Der Hauptgrund ist der eingesparte I/O: die temporären Dateien entfallen, also entfallen ihre EXCPs. Darauf kommt, dass jede eingesparte Komponente noch ein paar Prozent bringt, und dass ersetzte REXX-Routinen für Delimiter deutlich mehr bringen als ein paar Prozent.
Der Sonderfall Subsystem: dort entfällt sogar die erste Datei, weil wir die Daten direkt aus dem Speicher der Anwendung lesen.
Die Zahl selbst: ein großer IT-Dienstleister hat seine Jobketten auf FLAM umgestellt und im Schnitt Faktor 3 bis 5 an MSU eingespart. Keinen Kundennamen nennen.
Und der Punkt, der nicht in MSU gemessen wird, aber im Audit zählt: die Zwischenstände gibt es nicht mehr.



Kurz halten, aber nicht weglassen. In einer Bank ist das oft das stärkere Argument als die MSU. Die Kette auf Folie 3 legt sieben Zwischenstände unverschlüsselt auf die Platte. Wer Zugriff auf das Dateisystem hat, hat Zugriff auf die Klartexte – unabhängig davon, wie gut die Verschlüsselung am Ende ist. Und Step 8, der aufraeumt, läuft nach einem Abbruch gerade nicht. Beim Subsystem gibt es nicht einmal die Eingabedatei als Kopie.



15 Minuten in Zweiergruppen, dann zwei Ketten im Plenum durchsprechen.
Wichtig ist Schritt 4: nicht jede Kette lässt sich vollständig zusammenfassen. Ein Schritt, der ein Fremdprogramm mit eigener Logik aufruft, bleibt ein eigener Step – und das ist in Ordnung. Wer das sagt, wird geglaubt.
Auflösung:
2. Erfahrungswert für die Einordnung: Delimiter setzen, Zeichensatz wandeln, komprimieren, verschlüsseln, armorieren, übertragen und aufräumen sind alle Konvertierung oder Transport – also FLAM. Fachlogik ist alles, was Werte prüft, ableitet oder aus mehreren Quellen zusammenführt.
3. Das Ergebnis ist ein `READ` der Eingabe plus ein `WRITE` mit der Kette der Konvertierungsstufen. Wer dabei zwei verschiedene Zielformate braucht, schreibt zwei `WRITE` – das ist der Multiple-Write-Fall.
4. Übrig bleiben typischerweise: der Fachlogik-Step, ein Scheduler-Step und eventuell ein Freigabe- oder Prüfschritt. Drei von acht ist ein gutes Ergebnis, nicht ein schlechtes.
5. Die Zahl der entfallenden temporären Dateien ist immer die Zahl der zusammengefassten Steps minus eins.

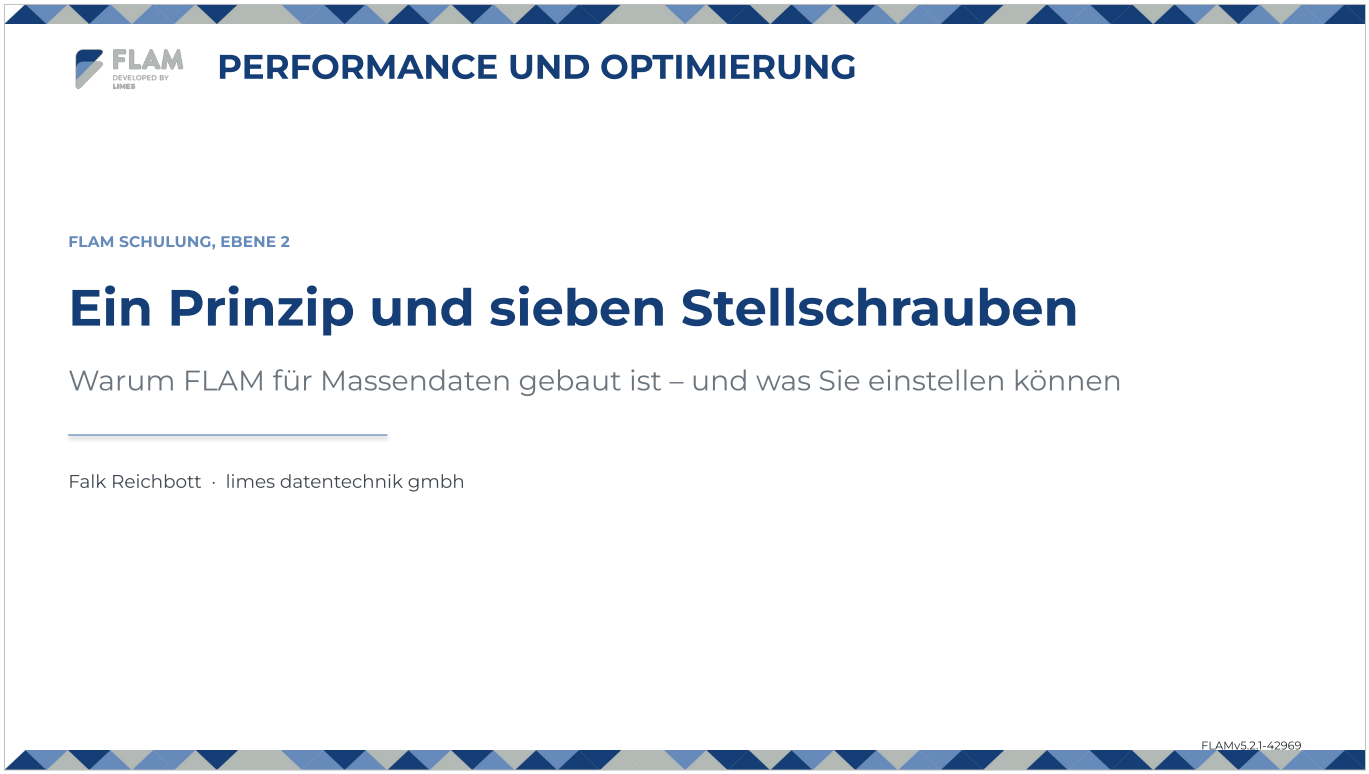
WAS BLEIBT

- eine Jobkette wird zu **einem Step**, weil die Konvertierungsstufen im Speicher hintereinander laufen
- der größte Posten ist der **gesparte I/O** – die temporären Dateien entfallen
- **Multiple Writes** und **Multiple I/O** liefern mehrere Ergebnisse aus einem Lesevorgang
- `MAXCC=-2` für die Codes 1 und 2, `NUMBER=OFF` im Editor, `LANG` in `STDENV` – die drei häufigsten Stolpersteine
- `CEEOPTS` und `LLA/VLF` sind die zwei Stellschrauben außerhalb von FLAM

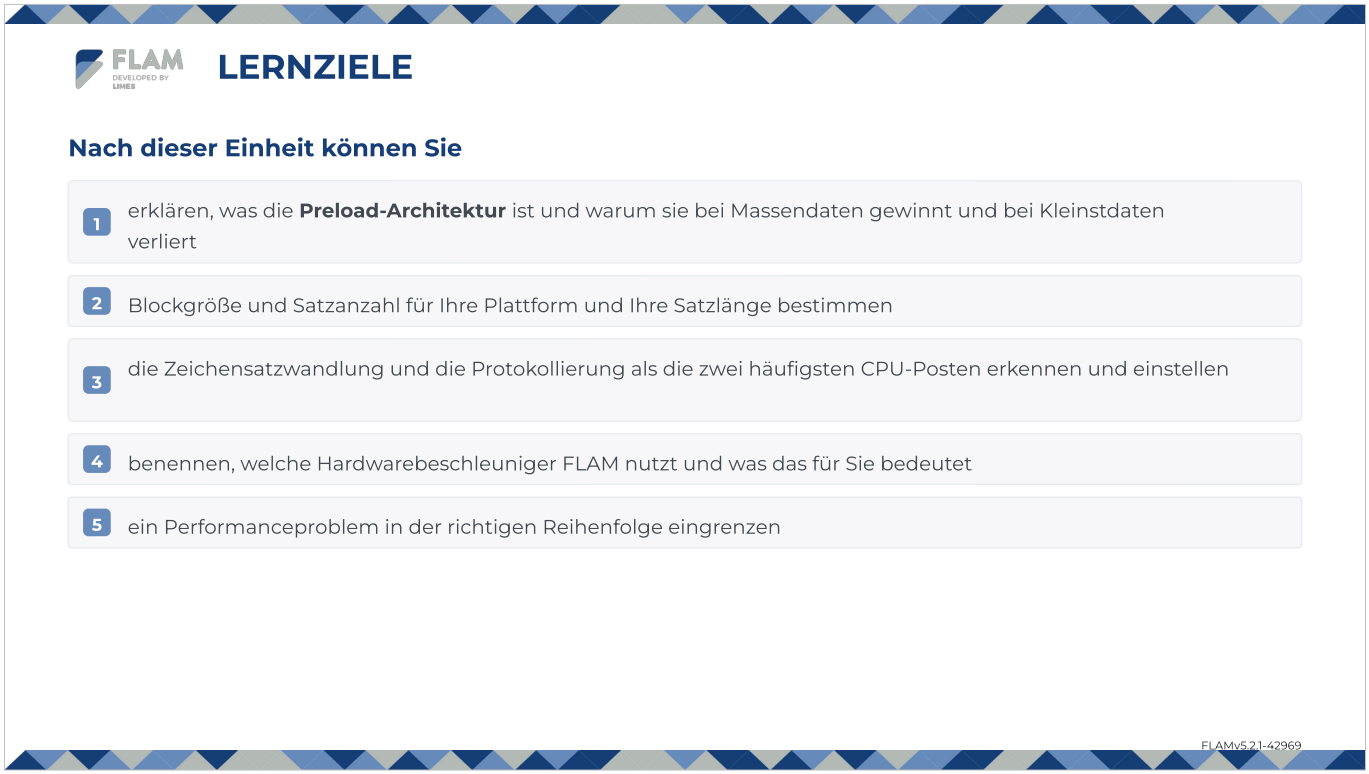
Nächste Einheit: was Sie an diesem einen Step noch einstellen können – Blockgröße, Satzanzahl, Zeichensatzwandlung, Protokollierung.

FLAMv5.21-42969

Der letzte Punkt ist die Brücke zur nächsten Einheit: warum es sich lohnt, den Step danach noch zu optimieren.



Zweite Einheit am Donnerstag, direkt nach der Batch-Einbettung. Die Reihenfolge ist Absicht: erst die Kette zu einem Step machen, dann diesen einen Step einstellen.
Einstieg mit der Frage: wer von Ihnen hat FLAM schon einmal als langsam erlebt? Die Antworten sind fast immer derselbe Fall – viele kleine Dateien.
Genau darauf läuft Folie 3 zu.



Ziel 1 ist das wichtigste. Wer die Preload-Architektur versteht, kann selbstständig entscheiden, ob FLAM für einen Anwendungsfall das richtige Werkzeug ist – und braucht uns dafür nicht zu fragen.



Das ist die zentrale Folie der Einheit. Langsam machen.
Der Kern: FLAM rechnet im Open so viel wie möglich vor – Tabellen, Konvertierungsketten, Puffer, Formatbeschreibungen. Das kostet Speicher und es kostet einmalig CPU. Danach ist der Run über die Massendaten so günstig wie möglich.
Die Formulierung, die hängen bleibt: wir bezahlen einmal, damit wir eine Million Mal billig sind.
Die Gegenrichtung nicht verschweigen – das ist der Grund, warum diese Folie Vertrauen schafft: wer 10000 Dateien mit je ein paar Byte verarbeitet, zahlt die Vorberechnung 10000 Mal und bekommt sie nie zurück. Da geht der CPU-Verbrauch durch die Decke, und das ist kein Defekt, sondern die Bauart.
Die Caches erwähnen, aber nicht als Lösung verkaufen: sie dämpfen den Effekt, sie heben ihn nicht auf.

WOFÜR FLAM GEBAUT IST – UND WOFÜR NICHT

	GUT FÜR FLAM	SCHLECHT FÜR FLAM
Datenmenge	Massendaten, GiB bis TiB	Dateien mit wenigen Byte
Aufrufmuster	ein Lauf über viele Daten	ein Prozess je Datei
Kosten pro MiB	konstant	der Vorberechnungsanteil dominiert
Was hilft	Blockgröße und Satzanzahl passend setzen	viele Dateien in **einem** Lauf verarbeiten, damit die Caches greifen

VIELE EINGABEN, EIN LAUF – DIE VORBERECHNUNG FÄLLT EINMAL AN

```
flcl conv "read.record(file['<SYSUID>.EIN1' '<SYSUID>.EIN2'
                           '<SYSUID>.EIN3'])
           write.text(file='/daten/aus.gz' comp.gzip())"
```

Die Caches leben im Prozess: ein Prozess je Datei gibt sie jedes Mal wieder auf.

FLAMv5.21-42969

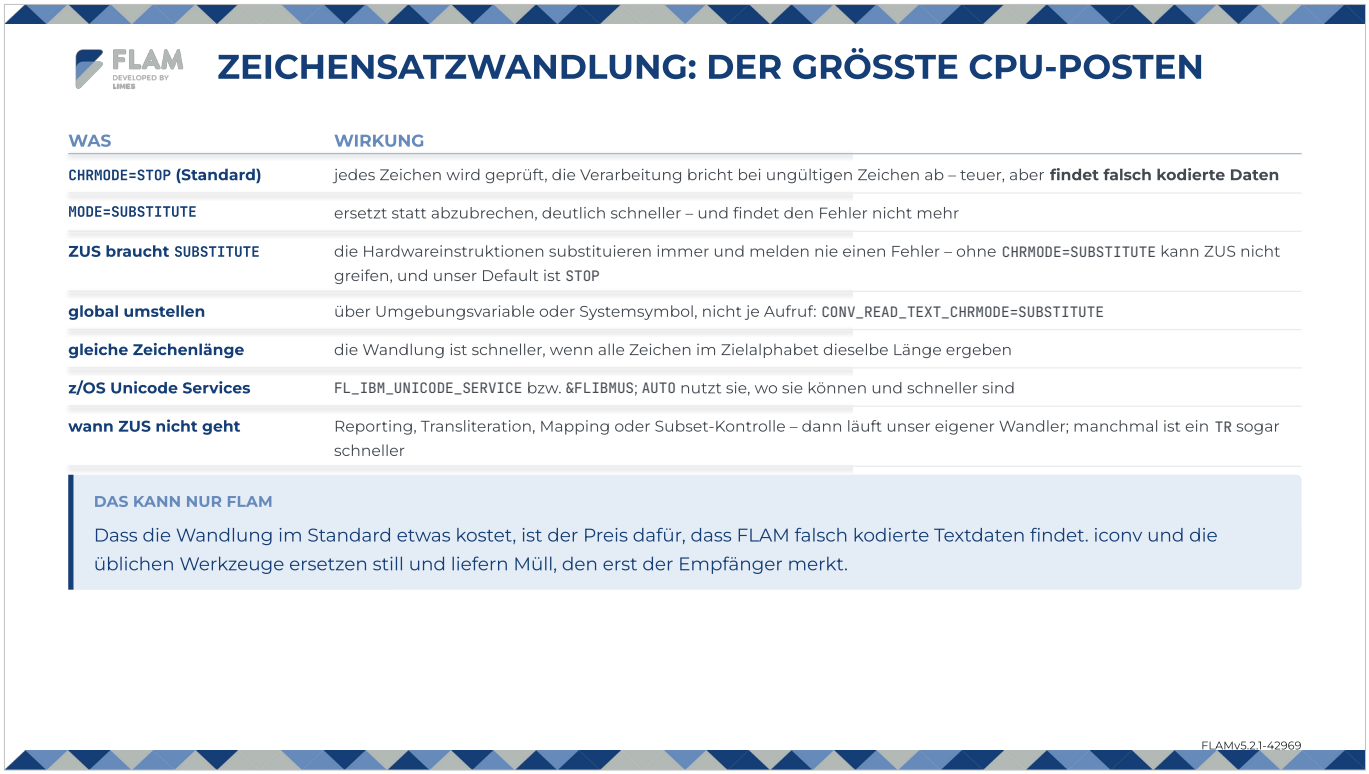
Diese Folie beantwortet eine Frage, die im Betrieb immer wieder auftaucht, und sie beantwortet sie ehrlich. Das ist ein Vertrauensgewinn, kein Eingeständnis.
Die Caches konkret nennen, weil sie zeigen, dass wir das Problem kennen und adressiert haben: die Zeichensatzwandlungstabellen werden gecacht, die Literale werden gecacht. Wer in einem Lauf viele kleine Dateien verarbeitet, profitiert davon. Wer je Datei einen eigenen Prozess startet, profitiert nicht – der Cache lebt im Prozess.
Daraus folgt die praktische Empfehlung: viele kleine Dateien in einem Lauf verarbeiten, nicht in vielen Läufen. Das Dateiarray und der Verzeichnis-Durchlauf sind genau dafür da.



Diese Folie ist die Landkarte für den Rest der Einheit und gleichzeitig die Folie, die abfotografiert wird. In der Reihenfolge der Wirkung, nicht der Bequemlichkeit.
Die letzten zwei Zeilen sind die, die nicht in FLAM eingestellt werden, sondern im System – das ist der Punkt für den Systemprogrammierer.



Die Mechanik erst, dann die Zahlen – und die Zahlen sind bei FLAM5 andere als die, die man aus der FLAM4-Zeit kennt.
FLAM verarbeitet in eigenständigen Segmenten. Jedes Segment hat einen Overhead, also sollen die Segmente nicht zu klein sein. Ist zu viel in einem Segment, läuft die Verarbeitung dauernd außerhalb der Caches – und der Durchsatz sinkt wieder. Es gibt ein Optimum, und es hängt an der Plattform.
Zu den **64 KiB**, weil die Zahl herumgeht: das ist ein **historischer Wert**, den wir aus Abwärtskompatibilitätsgründen für die Blockung immer noch nehmen. Er ist kein Optimum.
Für die FLAM5-Archive haben wir umfangreiche **Messreihen** gefahren und die Defaults danach neu ausgerichtet: **eine Matrix soll rund 1 MiB tragen**, damit das Komprimat bei 100 bis 200 KiB landet. Das ist die Größe, in der sich Kompressionsrate und Cacheverhalten am besten treffen.
Auf der Leseseite richtet sich FLUC nach dem Medium: **von einer SSD liest er per Default 1 MiB, von alten Platten mit 4-KiB-Sektoren 64 KiB.** Auf dem Host unter MVS wird selbstverständlich blockweise gelesen.
Bei der Satzanzahl je Matrix ist der Unterschied am größten: **FLAM4 hat einen festen Record Count von 128. FLAM5 macht ihn dynamisch** – Minimum 4, Mittel 1024, und bei vielen kleinen Records auch mehr.
Und jetzt der Punkt, der erklärt, warum das überhaupt eine Rolle spielt.
Die Originaldatenverarbeitung richtet sich nach dem System aus. Die Archivierung läuft dagegen **völlig autark und asynchron**. Ein Block Originaldaten kann also zu mehreren Matrizen führen oder nur einen Teil einer Matrix füllen – die beiden Seiten sind entkoppelt.
Wichtig ist dabei nur eines: **die Anzahl der Records pro Matrix muss definiert sein, damit man positionieren kann.** Position geteilt durch Record Count ergibt den Matrixindex. Ist die Anzahl je Matrix nicht konstant, dauert der Zugriff auf einen bestimmten Record länger, weil man sich herantasten muss.
Daraus folgt eine Design-Entscheidung, die im Betrieb regelmäßig auffällt:
gelöschte Records werden nur als gelöscht markiert. Erst wenn die Segmente neu geschrieben werden, werden die Matrizen reorganisiert. Wer sich fragt, warum das Archiv nach dem Löschen nicht kleiner wird – das ist der Grund, und `PRUNE` ist die Antwort.



Hier liegt in der Praxis das meiste Potenzial, und hier liegt gleichzeitig die gefährlichste Abkürzung. Beides ehrlich nebeneinander stellen.
Standard ist MODE=STOP: jedes Zeichen wird auf Gültigkeit geprüft und die Verarbeitung bricht bei einem ungültigen oder nicht abbildbaren Zeichen ab.
Das kostet CPU und es ist der Grund, warum FLAM falsch kodierte Textdaten findet, wo andere Werkzeuge stillschweigend Müll erzeugen. Das ist ein Alleinstellungsmerkmal, kein Overhead.
Wer SUBSTITUTE einstellt, wird schneller und verliert genau diese Erkennung.
Das ist eine fachliche Entscheidung, keine technische.
Zweiter Punkt: die Wandlung ist schneller, wenn alle Zeichen des Eingabealphabets im Ausgabealphabet dieselbe Länge ergeben.
Dritter Punkt, nur z/OS, und der ist der wichtigste für die CPU-Rechnung:
die **z/OS Unicode Services**, kurz ZUS. Wir nutzen sie, solange sie die verlangte Wandlung können und schneller sind als unsere Implementierung.
Hier gibt es aber eine Bedingung, die man kennen muss, weil sie sonst stillschweigend verhindert, dass ZUS überhaupt zum Einsatz kommt: **die Hardwareinstruktionen substituieren immer und melden nie einen Fehler.** Damit ZUS genutzt werden kann, muss also `CHRMODE=SUBSTITUTE` gesetzt sein.
Und unser Default ist aus historischen Gründen `STOP`.
Daraus folgt unsere Empfehlung für Kunden, die auf z/OS CPU sparen wollen:
den `CHRMODE` per Default auf `SUBSTITUTE` stellen – über die Umgebungsvariable oder das Systemsymbol, nicht in jedem einzelnen Aufruf.
Dann kann ZUS immer greifen.
Und die Kehrseite ehrlich dazu: wer **Reporting** aller Änderungen braucht, oder **Transliteration**, oder ein **Mapping**, oder die Kontrolle eines **Subsets** – für den können wir ZUS nicht nutzen. Dann läuft unser eigener Wandler. Es gibt außerdem Fälle, in denen ein einfaches `TR` schneller ist als ZUS; dann nehmen wir auch das und nicht die Spezialinstruktionen.
Die Entscheidung ist damit fachlich, nicht technisch: Fehlererkennung gegen CPU. Wir treffen sie nicht für den Kunden.

PROTOKOLLIERUNG: DER VERGESSENE POSTEN

- FLAM schreibt im Standard **viele** Meldungen – das kostet CPU und I/O
- `QUIET` und `SILENT` nehmen die Gesprächigkeit zurück
- das `MESSAGE`-Objekt regelt die Menge fein – in beide Richtungen
- der häufigste Fall: beim Übergang von Test nach Produktion wird das Zurückstellen vergessen
- `MESSAGE` ist die **Auswahl** der Meldungen, `LOGGING` das **Ziel** – zwei getrennte Objekte, die oft verwechselt werden

IN PRODUKTION LEISE, IM FEHLERFALL GEZIELT LAUTER

```
//CONV     EXEC PGM=FLCL,PARM='CONV MAXCC=-2 SILENT'
//FLAMPAR  DD *
  read.record(file=DD:IN);
  write.text(file='/daten/aus.txt');
  message(minimal);
  logging.file(name='/pfad/flam.log');
```

`MESSAGE` ist eine **Liste von Schaltern** – manche sind Gruppen (`MINIMAL`, `DEFAULT`, `ERRONLY`), manche einzelne Meldungstypen (`INFO`, `WARNING`, `DEBUG`, `NOTICE`). `LOGGING` bestimmt getrennt davon, **wohin** geschrieben wird.

FLAMv5.21-42969

Die Folie mit dem besten Verhältnis von Aufwand zu Nutzen der ganzen Einheit, weil die Umstellung fünf Minuten dauert.
FLAM schreibt im Standard viele Meldungen. Das kostet CPU und I/O. Beim Übergang von Test nach Produktion wird regelmäßig vergessen, die Gesprächigkeit zurückzunehmen – und dann kostet es in der Produktion Plattenplatz und Rechenzeit.
Die Gegenrichtung auch nennen: im Fehlerfall will man mehr, nicht weniger.
Das MESSAGE-Objekt kann in beide Richtungen.

LITERAL-CACHE

- ein **globaler** Literal-Cache spart Speicher und CPU beim Vergleich von Zeichenketten
- dahinter steht eine Hashtabelle; ihre Größe wird als **Zweierpotenz** angegeben
- `FL_LITERAL_CACHE_SIZE` bzw. `&FLITCS`, Werte 8 bis 20 – also 256 bis rund 1.048.576 Einträge
- im **Debug-Log** stehen die Statistiken dazu: man muss nicht raten, ob die Tabelle zu klein ist

SETZEN UND IM DEBUG-LOG KONTROLLIEREN

```
export FL_LITERAL_CACHE_SIZE=16     # 65536 entries
flcl info get.system
```

Auf z/OS als Systemvariable `&FLITCS`. Der Cache lebt im Prozess – er wirkt in einem langen Lauf, nicht über viele kurze.

FLAMv5.21-42969

Kurz halten – das ist eine Feineinstellung, kein Hebel für den Alltag.
FLAM nutzt einen globalen Literal-Cache, um Speicher und CPU beim Zeichenkettenvergleich zu sparen. Dahinter steht eine Hashtabelle. Ihre Größe wird als Zweierpotenz angegeben, 8 bis 20, also 256 bis rund eine Million Einträge.
Der wichtige Satz: im Debug-Log stehen die Statistiken dazu. Man muss also nicht raten, ob die Tabelle zu klein ist.



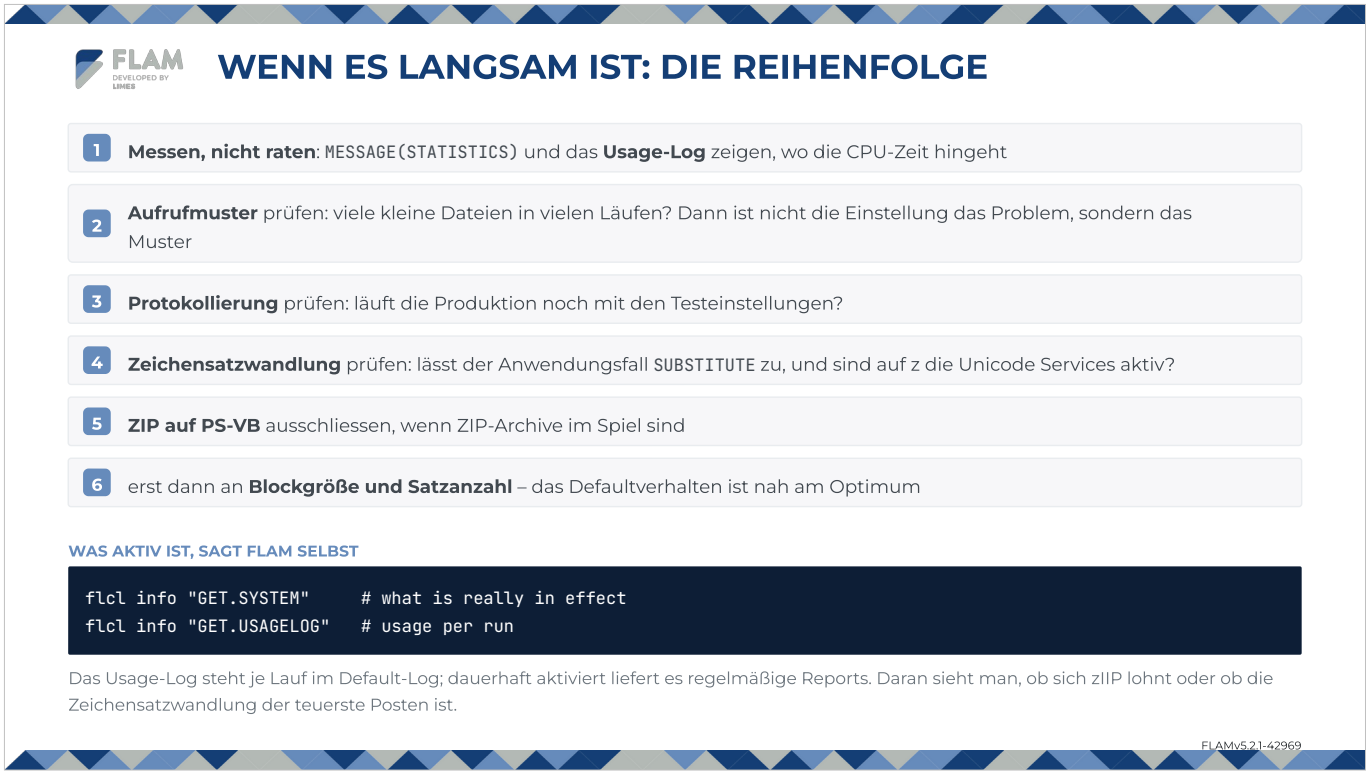
Diese Folie ist der Punkt, der auf unserer Homepage bewusst nicht steht – sie ist ein echtes Argument und gleichzeitig ein Überblick, den man sonst nirgends bekommt.
Wichtig: das meiste davon nutzt FLAM automatisch, wenn es verfügbar und lizenziert ist. Es ist keine Konfigurationsaufgabe.
CPACF für Verschlüsselung und Hashing, zEDC dort, wo die zlib zum Einsatz kommt, spezielle Instruktionen einschließlich Vektorrechnung auf allen Plattformen – nicht nur auf dem Host.
Und der Punkt, den kaum jemand erwartet: wir binden dort, wo es geht, die schnellsten verfügbaren Bibliotheken. zlib-ng statt zlib, expat als XML-Parser. Das ist eine bewusste Entscheidung und wird bei jedem Bibliotheks-Update überprüft.
Zum zEDC noch ein Punkt, der eine Rückfrage vorwegnimmt: **zEDC-Support gibt es auch für die alten FLAMFILEs.** Dort wird der ADC durch DEFLATE ersetzt.
Man muss aber die ganze Rechnung aufmachen, sonst empfiehlt man etwas Falsches: DEFLATE liefert **wesentlich größere Komprimate** als der ADC. Man bezahlt die Kompression dann nicht mehr in CPU – aber in Plattenplatz und in zusätzlichen EXCPs, also I/O-Operationen. Über die Kette hinweg rechnet sich das häufig **nicht**.
Also: bei FLAM5 ist zEDC dort, wo die zlib arbeitet, eine gute Sache. Beim FLAM4-ADC ist es eine Abwägung, und die geht oft gegen zEDC aus.
Auf die nächste Einheit verweisen: zIIP steht hier absichtlich nur in einer Zeile, weil es kein Performance-Thema ist, sondern ein Abrechnungsthema.



Das ist die Folie, die den Unterschied zu ZIP und GZIP am klarsten macht – und sie kommt spät, weil sie erst nach der Mechanik wirkt.
Erster Punkt: der Tabellen-Support reduziert die Nettodaten, bevor überhaupt komprimiert wird. Bei XML in der Regel um Faktor 3, weil die Struktur einmal beschrieben und nicht je Zeile wiederholt wird. Genau das kann ein ZIP prinzipiell nicht: es sieht einen Bytestrom, keine Spalten.
Zweiter Punkt: der segmentweise Zugriff. Wer in einem Archiv nach einer Transaktion sucht, entpackt nicht das Archiv, sondern die Segmente, in denen die Treffer liegen. In der Praxis sind das oft 1 bis 5 Segmente mit je 128 Records – statt Hunderttausenden. Das ist die größte Einsparung, die es hier gibt, und sie ist mit ZIP oder GZIP grundsätzlich nicht erreichbar.
Keinen Kundennamen nennen, auch nicht die Tagesmengen. Die Mechanik reicht.
Dritter Punkt: die Zahlen, die wir öffentlich sagen. 100 MiB auf 2 bis 10 MiB, bei SEPA-XML bis 98 Prozent. Das steht auf der Homepage und ist damit zitierbar.



Sehr konkret, sehr host-spezifisch, und ein Fall, der im Support regelmäßig als "FLAM ist langsam" ankommt.
Der Mechanismus: beim Erzeugen eines ZIP-Archivs müssen am Ende Offsets und Statistikwerte in die Member-Header geschrieben werden. Das braucht ein Zurueckpositionieren in der Datei. Unter z/OS geht das über RBA-Zugriff. In einer PS-VB-Datei führt das Betriebssystem dafür im Hintergrund einen Baum, der sich merkt, welche relative Byteadresse in welchem Block, in welchem Record, an welchem Offset liegt. Das ist extrem aufwendig, und bei vielen Membern steigt die Laufzeit erheblich. Die Pflege dieses Baums kostet außerdem CPU, die der Anwendung nicht angerechnet wird – man sieht sie also in der Jobstatistik nicht.
Die drei Auswege in dieser Reihenfolge: einen für RBA-Zugriff geeigneten Dateityp nehmen – und das sind **USS-Dateien** –, oder das STREAM-Schlagwort nutzen, oder PS-FB.
Warum USS-Dateien: das zFS wird physisch in einem VSAM-LDS abgebildet. Ein LDS ist der Vorläufer dessen, was man heute Memory Map nennt – ein flacher, unstrukturierter Adressraum, auf den man mit Offsets zugreift. Deshalb ist Seeken dort billig, und deshalb ist es genau die Ausprägung von Flatfile, die wir brauchen. Die Datei selbst ist kein LDS, das Dateisystem darunter ist eines – diesen Unterschied fragt erfahrungsgemäß jemand nach.
Bei STREAM und bei PS-FB jeweils die Einschränkung ehrlich nennen – das ist der Grund, warum die Folie glaubwürdig ist.



Die praktische Folie. In dieser Reihenfolge vorgehen, nicht in der Reihenfolge der Vermutungen. Punkt 1 zuerst, weil er die häufigste Ursache ist und weil die Antwort dann lautet: das ist die Bauart, ändere das Aufrufmuster.
Punkt 2, weil Protokollierung in fünf Minuten umgestellt ist.
Punkt 6 ist der, den man am liebsten zuerst macht und am besten zuletzt.
Die zwei Aufrufe unten gehören vor jede Messung, und zwar aus einem Grund:
sie trennen Vermutung von Tatsache. `INFO GET.SYSTEM` zeigt die wirksamen Umgebungsvariablen – also ob `SUBSTITUTE` und die Unicode Services wirklich gesetzt sind und nicht nur gesetzt sein sollten. `LICENSE` zeigt die aktiven Feature, darunter die Beschleuniger: wer CPACF erwartet und keine Lizenz dafür hat, misst umsonst.
Der wichtigste Rat vorweg, und er steht deshalb an Punkt eins: **messen, nicht raten.** Das Werkzeug der Wahl dafür ist das **Usage-Log** zusammen mit den Statistikinformationen. Man kann es sich je Lauf im Default-Log ansehen, oder das Usage-Logging dauerhaft aktivieren und regelmäßige Reports ziehen. Dann sieht man schwarz auf weiß, ob sich zIIP lohnen würde, oder ob die Zeichensatzwandlung der teuerste Posten ist – und muss nicht darüber diskutieren.
Und ein Satz zur Erwartungssteuerung, den wir ruhig sagen können, weil er stimmt: **unser Defaultverhalten ist nah am Optimum.** Nur in speziellen Fällen muss man an Blockgrößen und Recordanzahlen schrauben – gerade bei sehr großen Dateien. Wer bei jedem Job an den Schrauben dreht, verschwendet seine Zeit.
Ein Werkzeug für den Sonderfall noch, weil es kaum bekannt ist: die **Force-Switches**. Per Default erhalten wir Recordlängen und lesen transparent. Wenn man aber von vornherein weiß, dass das nichts bringt – etwa weil in einem satzorientierten Hostdataset ohnehin auf Delimiter gelesen werden muss und die Recordlängen keine Rolle spielen – dann schaltet `FRCBLK` die Behandlung der Recordlängen ab und erzwingt Blockorientierung.
Man verzichtet damit auf die Universalität des transparenten Lesens und gewinnt Geschwindigkeit.



Die Frage kommt in jeder Schulung, und sie hat keine Antwort in einer Zahl – aber sie hat eine Reihenfolge, in der man sie beantwortet.
Die wichtigsten Gegenspieler sind vier:
GZIP ist der Standard und die richtige Antwort, solange nichts dagegen spricht. Überall lesbar, gut untersucht, und wir binden zlib-ng statt zlib, also die schnellere Implementierung.
GZIP gegen zEDC: auf z/OS macht zEDC dasselbe Verfahren in Hardware. Das kostet keine CPU. Aber die Hardware sucht Muster nur in einem begrenzten Fenster – das entspricht etwa DEFLATE-Level 1. Bei strukturierten Daten kann das Komprimat deshalb **bis zu Faktor 10 größer** werden als mit unserer Tabellenunterstützung. Man tauscht CPU gegen Platz und I/O.
GZIP gegen BZIP2: BZIP2 komprimiert stärker, kostet aber deutlich mehr CPU – und zwar in beide Richtungen. Für ein Archiv, das einmal geschrieben und selten gelesen wird, kann das richtig sein. Für einen täglichen Austausch fast nie.
Und jetzt der Unterschied, der die Wahl **fachlich** macht statt nach Gefühl, denn die Verfahren arbeiten nicht nach demselben Prinzip: **BZIP2 ist ein arithmetisches Verfahren, GZIP und ZSTD machen Mustersuche.** Daraus folgt eine Regel, die man sich merken kann: **BZIP2 ist gut für Zahlenkolonnen und binäre Formate** – Beträge, Messwerte, Integer- und Float-Felder. **Mustersuche ist gut für Text** – Namen, Adressen, Verwendungszwecke. Wir machen es in unseren eigenen Archiven genau so: die **Längenfelder** komprimieren wir mit BZIP2, die **Daten** mit Mustersuche.
Das ist auch die Brücke zum Topfdesign: sobald man Spalten in verschiedene Töpfe legt, kann man je Topf das passende Verfahren wählen – Beträge arithmetisch, Texte über Muster. Wer alles in einen Topf legt, muss sich für eins entscheiden und verliert auf der anderen Seite.
GZIP gegen ZSTD: ZSTD ist der modernere Kompromiss – bei vergleichbarer Rate schneller, und über den Level breit einstellbar. Wenn beide Seiten ZSTD können, ist es meist die bessere Wahl. Wenn der Empfänger nur GZIP kann, ist die Diskussion vorbei.
Dann der Level, und dazu eine praktische Regel: **die oberen Level kosten viel CPU für wenige Prozent.** Wer den Level erhöht, sollte vorher messen, was er dafür bekommt.
`XZ` beziehungsweise LZMA, `LZ4` und `TERSE` unterstützen wir auch, aber das sind unübliche Fälle – `LZ4` wenn es nur um Geschwindigkeit geht, `TERSE` wenn ein Host-Partner es verlangt.
Und der Punkt, der die ganze Diskussion relativiert: **weil wir die Komprimate in der Regel verschlüsseln und die Verschlüsselung über CPACF läuft, fallen Verschlüsselung und Hashing kaum ins Gewicht.** Bei den FLAM5-Archiven kommt dazu, dass man das **pro Topf und damit pro Spalte** entscheidet: verschlüsselt wird nur, was sensibel ist. Ob eine Spalte verschlüsselt oder klar gehalten wird, spielt für die Laufzeit dann kaum eine Rolle – die Entscheidung ist fachlich, nicht technisch.

ÜBUNG: EINE BLOCKGRÖSSE BESTIMMEN

Einzeln oder in Zweiergruppen, 10 Minuten:

1. Ihre Records sind im Mittel 200 Byte lang. Welche Satzanzahl gehört zu einer Blockgröße von 256 KiB?
2. Was passiert, wenn Sie die Blockgröße erhöhen, aber die Satzanzahl bei 512 lassen?
3. Nehmen Sie einen eigenen Job und schätzen Sie, welcher der sechs Punkte von Folie 13 bei ihm greifen würde

DIE RECHNUNG ALS AUFRUF

```
flcl conv "read.record(file='<SYSUID>.EIN')
           write.fl5(file='<SYSUID>.ARCHIV.FL5'
                     fmt.blk(blksiz=256k reccnt=1300))" \
     SILENT MAXCC=-2
```

Messen, nicht schätzen: denselben Job mit den Defaults und mit Ihren Werten laufen lassen und die CPU-Zeiten vergleichen.

FLAMv5.21-42969

10 Minuten. Rechenaufgabe, absichtlich einfach, damit die Formel hängen bleibt.
Die Auflösung: 256 KiB durch 200 Byte sind rund 1300 Records. Wer 512 stehen lässt, füllt das Segment nur zu gut einem Drittel – knapp zwei Drittel bleiben ungenutzt.
Beim zweiten Teil geht es darum, dass sie den Effekt selbst messen – nicht darum, dass die Zahl stimmt.

WAS BLEIBT

- **Preload**: teuer im Open, günstig im Run – deshalb konstante Kosten pro MiB bei Massendaten
- **viele kleine Dateien** sind der eine Fall, in dem das gegen Sie arbeitet – dann das Aufrufmuster ändern, nicht die Einstellung
- der wirksamste Hebel ist **weniger Daten**: Tabellen-Support und segmentweiser Zugriff
- die zwei häufigsten CPU-Posten sind **Zeichensatzwandlung** und **Protokollierung**
- **Beschleuniger** nutzt FLAM automatisch – `flcl LICENSE` zeigt, welche lizenziert sind

Nächste Einheit: zIIP, zEDC und CPACF im Zusammenspiel – und warum zIIP kein Performance-Thema ist, sondern ein Abrechnungsthema.

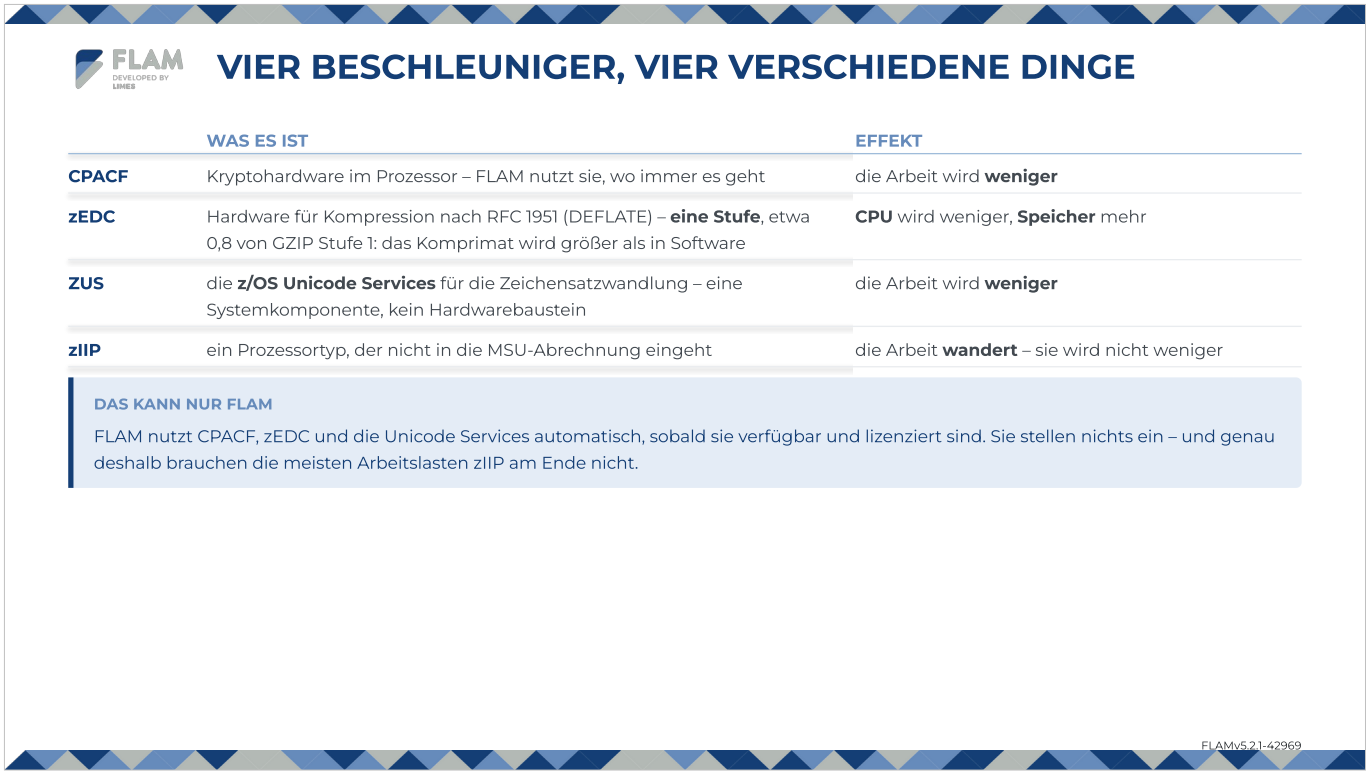
Der erste und der letzte Punkt sind die, die hängen bleiben sollen.
Überleitung zur nächsten Einheit: dort geht es um zIIP, und der erste Satz dort ist, dass zIIP kein Performance-Thema ist.



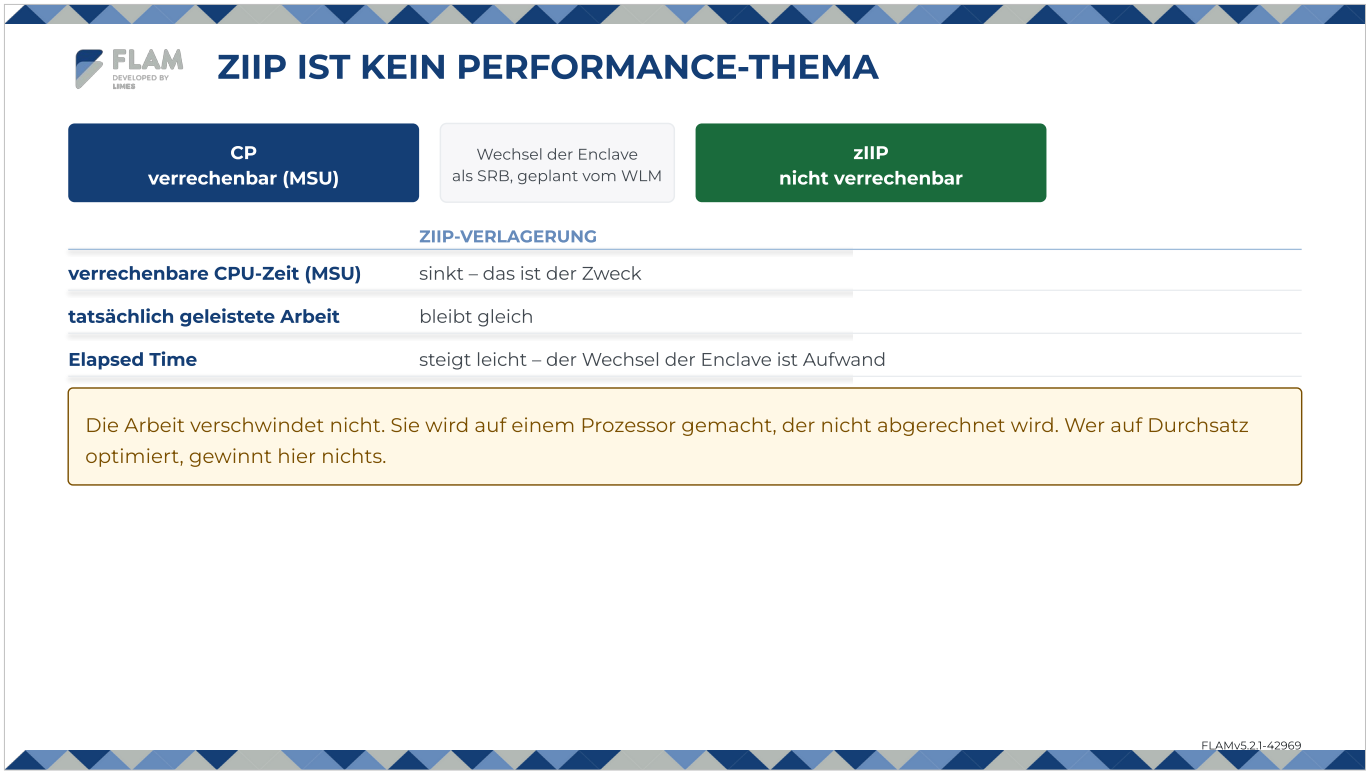
Letzte Einheit am Donnerstag. Das ist die Einheit, in der die konkrete Frage aus dem Haus beantwortet wird – deshalb bewusst allgemeingültig gehalten und ohne Kundennamen. Den konkreten Fall mündlich erzählen.
Der erste Satz der Einheit soll sitzen: zIIP ist kein Performance-Thema.
Danach ist das Gespräch ein anderes.



Ziel 1 ist die Kernbotschaft, Ziel 4 ist die Antwort auf die Frage, mit der die Teilnehmer gekommen sind, und Ziel 5 ist das, was sie danach selbst tun können.



Diese Folie sortiert die Begriffe, die im Gespräch regelmäßig durcheinandergehen – und das Durcheinander ist die eigentliche Ursache der meisten Missverständnisse zu diesem Thema. CPACF, zEDC und die z/OS Unicode Services sind echte Beschleuniger: sie machen die Arbeit schneller und günstiger. zIIP ist kein Beschleuniger, sondern ein anderer Prozessortyp, der nicht in die MSU-Abrechnung eingeht.
Die letzte Spalte ist die wichtige: bei den ersten drei sinkt die Arbeit, bei zIIP wandert sie nur.
Die **z/OS Unicode Services** sind der, den man am leichtesten übersieht, und sie sind auf einem Host mit viel Zeichensatzwandlung der wirksamste:
FLAM nutzt sie für einfache Wandlungen statt der eigenen Umsetzung. Das ist kein Hardware-Beschleuniger, sondern eine Systemkomponente – gesteuert über `FL_IBM_UNICODE_SERVICE` beziehungsweise das Systemsymbol `&FLIBMUS`.
Details und die Verhaltensunterschiede stehen in der Einheit *Zeichensätze*; hier gehört nur hin, dass es sie gibt und dass sie zählen.
CPACF und zEDC hängen am Accelerator-Feature der Lizenz – das ist der Punkt, auf den Folie 10 zurückkommt.
Eine Einschränkung gehört bei der zEDC gleich dazu, damit niemand sie für kostenlos hält: **die Karte komprimiert schlechter als Software.** Sie ist darauf ausgelegt, GZIP Stufe 1 zu erreichen, und liegt tatsächlich bei etwa 0,8 davon; die Stufe kann man nicht beeinflussen. ZSTD komprimiert in Software rund doppelt so gut wie DEFLATE, ADC bei strukturierten Daten noch besser – wer auf zEDC umstellt, braucht also unter Umständen **deutlich mehr Plattenplatz oder mehr Bänder** als mit GZIP in Software, ZSTD oder ADC. Der Effekt ist echt und er ist bezahlt: gespart wird CPU, ausgegeben wird Speicher und Bandbreite.



Die Kernfolie. Wenn nur eine Folie dieser Einheit hängen bleibt, dann diese.
Der Mechanismus: eine memory-to-memory-Funktion wird als SRB auf einen zIIP gegeben. Der Workload Manager plant das ein. Das kostet einen Wechsel der Enclave, und dieser Wechsel ist Aufwand – kein Gewinn.
Die Rechnung ist also: die verrechenbare CPU-Zeit sinkt, die Elapsed Time steigt leicht. Wer auf Durchsatz optimiert, gewinnt hier nichts. Wer auf die Rechnung optimiert, gewinnt.
Das ist keine Kritik an zIIP. Es ist die Voraussetzung dafür, die richtige Frage zu stellen: was will ich eigentlich optimieren?
Wichtig gegen ein häufiges Missverständnis: die Arbeit verschwindet nicht.
Sie wird auf einem Prozessor gemacht, der nicht abgerechnet wird.



Jetzt die technische Grenze, und die ist hart. Drei Bedingungen, alle drei müssen erfüllt sein.
Erstens memory-to-memory: die Funktion darf nur Speicher lesen und Speicher schreiben. Kein Dateizugriff, kein Datasetzugriff.
Zweitens metal, also ohne Language Environment. Ein SRB läuft nicht im LE.
Drittens kein Protokoll. Eine Funktion, die eine Meldung ins Log schreibt, macht I/O – und damit ist sie disqualifiziert.
Daraus folgt der Satz, der die Struktur erklärt: verlagerbar sind immer nur die RUN-Funktionen.
OPN und CLS schreiben ins Log, dort geht es prinzipiell nicht.
Das ist auch der Grund, warum das kein Schalter ist, den man umlegt: die Module müssen dafür gebaut sein.



Hier kommt der Punkt, der oft überrascht: verlagert wird gerade das Alte.
ADC, CX7, CX8 und VR8 sind die klassischen FLAM4-Kompressionsalgorithmen.
Diese Module waren schon immer metal – deshalb erfüllen sie die drei Bedingungen von Folie 5, ohne dass etwas umgebaut werden musste.
Und das ist keine Randnotiz: genau diese Verfahren sind das, was bei Kunden massenhaft im Einsatz ist. Die großen Bestände sind FLAM4-Archive.
Die zweite Hälfte der Folie ist die ehrliche Aussage, was nicht verlagert wird und warum. Nicht als Mangel formulieren – als Konsequenz aus Folie 5.

WAS ZUSÄTZLICH MÖGLICH WÄRE

KANDIDAT	WARUM ER IN FRAGE KOMMT
CNV-Module	die Konvertierungsstufen arbeiten memory-to-memory – Zeichensatzwandlung, Kompression, Verschlüsselung
FMT-Module	Formatierung und Satzaufbereitung, ebenfalls reine Speicherarbeit
Archiv außer FLMSEG	nur `FLMSEG` liest und schreibt die Segmente wirklich; alles darüber rechnet im Speicher
ZSTD und moderne Kompressoren	dieselbe Bauart wie ADC, nur neuer
Tabellen-Support (FMTTAB)	spaltenweise Aufbereitung, rechenintensiv und ohne I/O

Ausblick, keine Zusage: diese Module müssten dafür metal gebaut werden – und auch dann gilt weiterhin, dass nur die `RUN`-Funktionen in Frage kommen.

Diese Folie ist ein Ausblick und als solcher zu kennzeichnen. Nicht als Zusage formulieren.
Der Gedanke: alle unsere CNV- und FMT-Module arbeiten memory-to-memory. Im Archiv gilt das für alles außer FLMSEG, weil FLMSEG die Segmente tatsächlich liest und schreibt. Diese Module könnten metal gebaut und damit verlagert werden.
Konkret wären das die Zeichensatzwandlung, ZSTD und die anderen modernen Kompressoren, und der Tabellen-Support.
Immer mit derselben Einschränkung: nur die RUN-Funktionen.
Wer fragt, warum das nicht schon so ist: es ist Aufwand, und der Nutzen hängt daran, ob beim Kunden überhaupt zIIP-Kapazität frei ist. Das ist die Brücke zu Folie 11.



Das konkrete Beispiel, an dem die Bedingung von Folie 5 sichtbar wird – und es ist ein gutes Beispiel, weil es zeigt, dass die Grenze nicht am Modul hängt, sondern an der Konfiguration. Dieselbe Zeichensatzwandlung ist verlagerbar oder nicht, je nachdem, ob eine Reportdatei geschrieben wird. Mit Report: I/O, also kein zIIP. Ohne Report:
reine Speicherarbeit, also grundsätzlich möglich.
Der Ausweg, den man bauen könnte: den Report je Block in den Speicher legen und erst später herausschreiben. Auch das als Überlegung kennzeichnen.
Und dann der Punkt, der die ganze Überlegung für die Zeichensatzwandlung relativiert: auf z/OS verlagern wir sie schon – nämlich auf die Spezialhardware hinter den Unicode Services. Das spart echte CPU, nicht nur abgerechnete. Damit ist zIIP dort gar nicht mehr die interessante Frage.
Das ist die Überleitung zur nächsten Folie.



Die Zusammenfassung des vorigen Gedankens, jetzt für alle drei Bereiche.
Verschlüsselung: wir nehmen immer CPACF, wenn es möglich ist. Damit gibt es keinen Grund, Verschlüsselung auf zIIP zu geben.
Zeichensatzwandlung: die Unicode Services nutzen die Spezialhardware.
Kompression: zEDC macht RFC 1951 in Hardware; in FLAM5 nutzen wir das automatisch dort, wo die zlib zum Einsatz kommt.
Der Satz, der bleiben soll: die Reihenfolge ist Hardware zuerst, zIIP zuletzt. zIIP ist die Antwort für das, was keine Hardwareunterstützung hat.



Das ist die Folie, die die eigentliche Frage beantwortet – und sie beantwortet sie mit einem Mechanismus, nicht mit einer Meinung.
Der Ablauf: das Accelerator-Feature der Lizenz aktiviert zEDC und zIIP zusammen. In FLAM5 ist zEDC unproblematisch, dort wird es dort genutzt, wo die zlib arbeitet. Für FLAM4 haben Kunden sich zEDC transparent für den ADC gewünscht – und da beginnt das Problem.
zEDC wird dem zIIP vorgezogen. Und zEDC komprimiert nach RFC 1951, was für strukturierte Daten deutlich schlechter ist als der klassische ADC: die Komprimate können bis zu Faktor 10 größer werden. Wer das nicht ausschaltet, hat also mehr Plattenplatz, mehr I/O und kein zIIP – und dann "komische Effekte", die niemand zuordnen kann.
Die Lösung stehen lassen und langsam vorlesen: FL_ZEDC_SUPPORT oder &FLZEDC.
OFF, oder ein Schwellwert in Prozent.
Die Einschränkung beim Schwellwert unbedingt nennen, sonst ist die Empfehlung gefährlich: nur das erste Segment eines FLAMFILEs wird analysiert. Wenn sich die Datencharakteristik danach ändert, greift der Automatismus ins Leere.

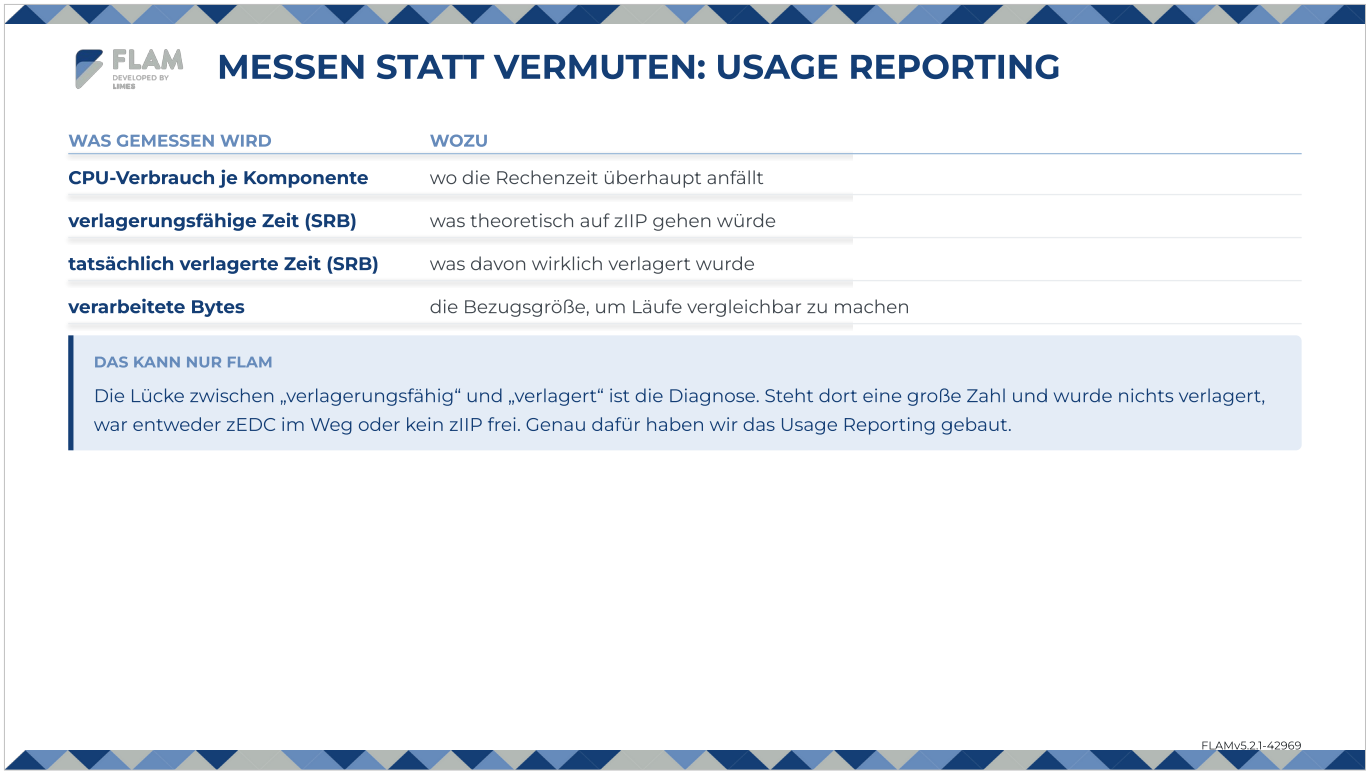
UND SELBST DANN: IST EIN ZIIP FREI?

- eine Verlagerung braucht **freie zIIP-Kapazität** – der WLM plant den SRB ein
- ist kein zIIP verfügbar, läuft die Arbeit auf dem CP – also genau dort, wo sie ohnehin gelaufen wäre
- auf heutigen Hosts ist zIIP-Kapazität oft knapp, weil **Java** sie belegt
- das ist keine FLAM-Frage und keine Konfigurationsfrage, sondern eine **Kapazitätsfrage**

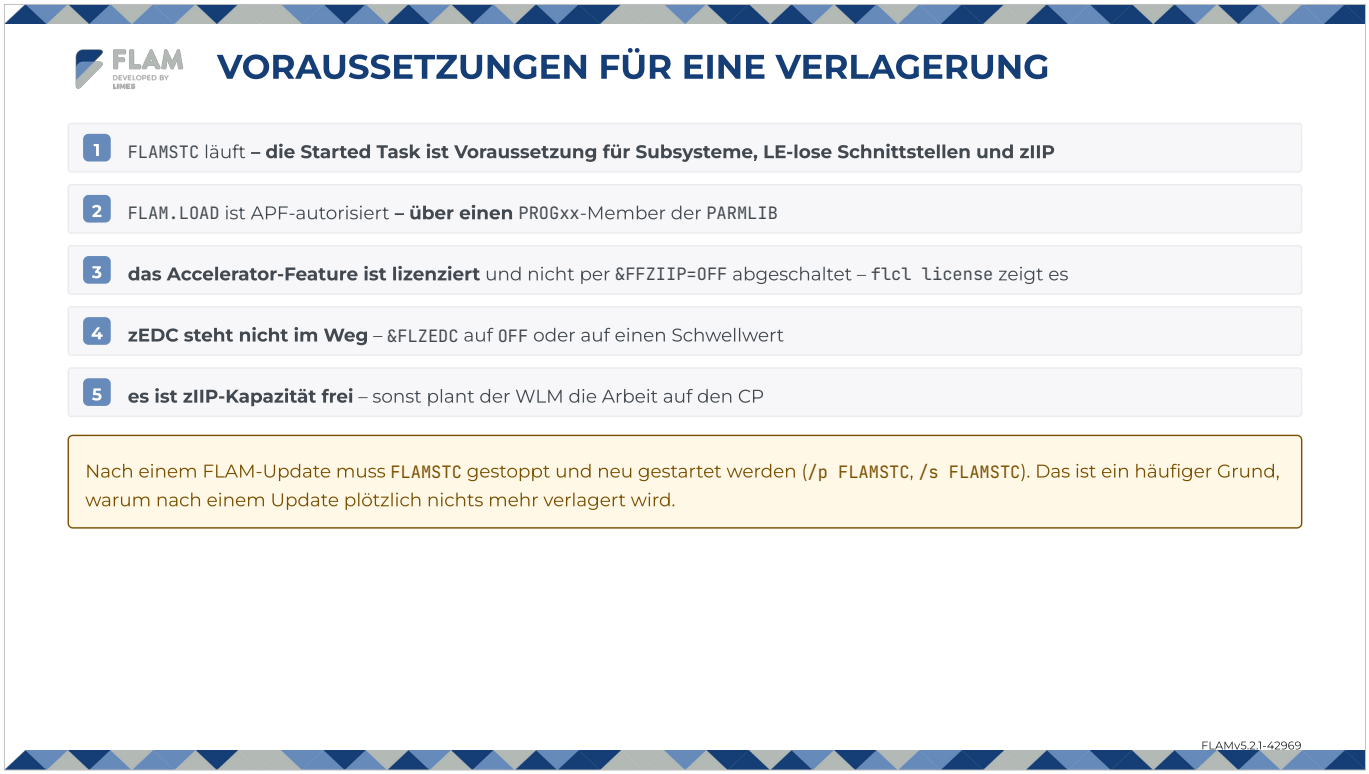
Die ehrliche Antwort auf „warum bei uns kein zIIP-Offloading?“ lautet fast immer: weil es in dieser Konstellation nichts gebracht hätte. Das lässt sich messen, statt es zu vermuten.

FLAMv5.21-42969

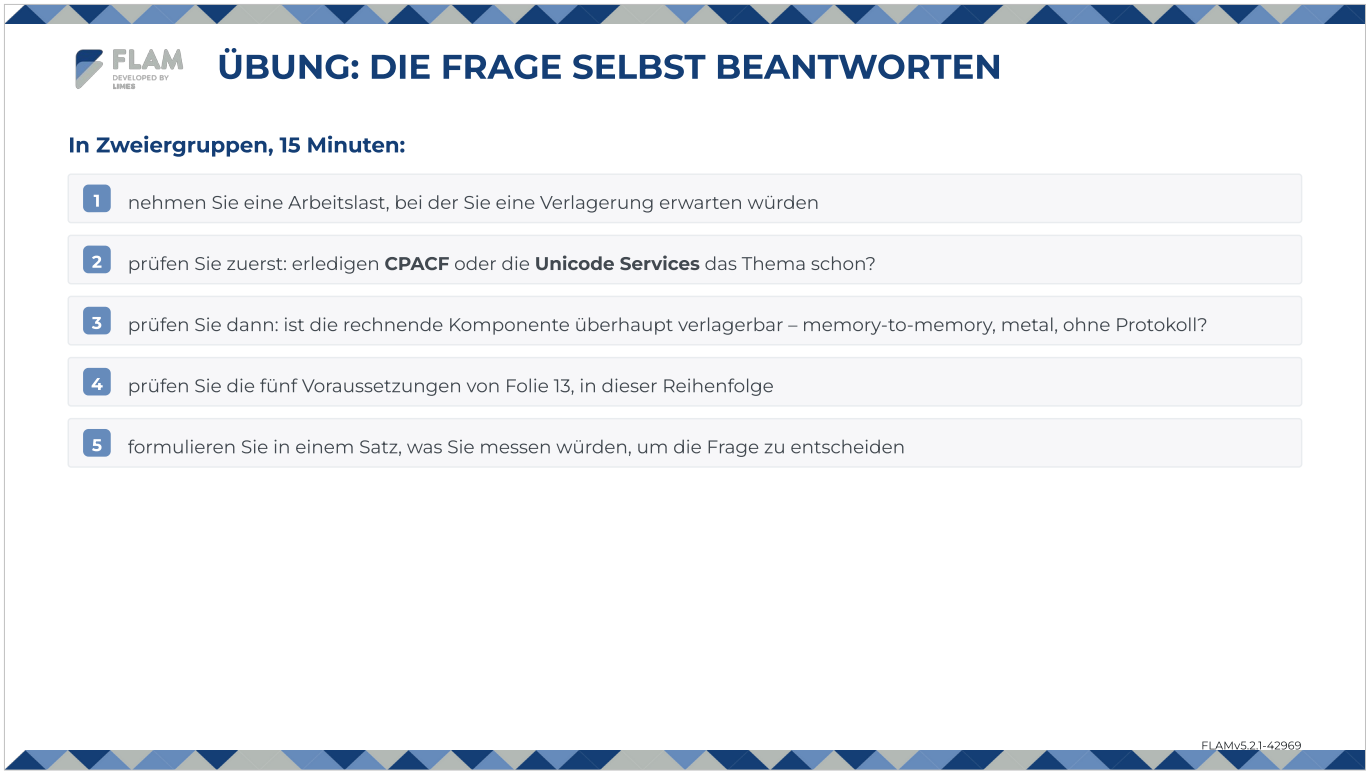
Der letzte Baustein der Antwort, und der, den man nicht in FLAM einstellen kann.
Eine Verlagerung braucht freie zIIP-Kapazität. Der Workload Manager plant den SRB ein; ist kein zIIP verfügbar, läuft die Arbeit auf dem CP – also genau dort, wo sie ohnehin gelaufen wäre.
Auf heutigen Hosts ist zIIP-Kapazität oft knapp, weil Java sie belegt. Das ist keine FLAM-Frage und keine Konfigurationsfrage, sondern eine Kapazitätsfrage für den Kapazitätsplaner.
Deshalb endet die ehrliche Antwort auf "warum kein zIIP bei uns" fast immer mit: weil es in dieser Konstellation nichts gebracht hätte. Und deshalb kommt jetzt die Folie zum Messen.



Die konstruktive Folie, und der Grund, warum diese Einheit nicht mit einem "kommt drauf an" endet.
Seit 5.1.13 misst FLAM den CPU-Verbrauch je Komponente und protokolliert dazu drei Werte: die verlagerungsfaehige Zeit als SRB, die tatsächlich verlagerte Zeit als SRB, und die verarbeiteten Bytes.
Das ist genau die Antwort auf die Frage. Die erste Zahl sagt, was theoretisch gehen würde. Die zweite sagt, was tatsächlich passiert ist. Die Lücke zwischen beiden ist die Diagnose – und in dieser Lücke stecken entweder zEDC oder ein belegter zIIP.
Der Aufwand ehrlich nennen: es braucht die Systemvariablen FLOGQL1 bis 3, einen Log Stream für SYSPLEX oder LPAR, eine laufende Started Task LOGR und die passenden RACF-Einstellungen. Das ist eine Aufgabe für den Systemprogrammierer, keine Kleinigkeit.
Aber es ist die einzige Grundlage, auf der man diese Diskussion mit Zahlen statt mit Vermutungen führen kann. Genau dafür haben wir es gebaut.



Die Checkliste zum Mitnehmen. Wer eine Verlagerung erwartet und keine sieht, geht diese fünf Punkte von oben nach unten durch.
Punkt 1 und 2 sind Installationsfragen, Punkt 3 ist die Lizenz, Punkt 4 ist die zEDC-Falle von Folie 10, Punkt 5 ist die Kapazitätsfrage von Folie 11.
Nach dem Umstellen von FLAM muss FLAMSTC gestoppt und neu gestartet werden – das ist ein häufiger Grund, warum nach einem Update nichts mehr verlagert wird.



15 Minuten, Zweiergruppen. Die Übung ist absichtlich die Originalfrage – damit die Teilnehmer merken, dass sie sie jetzt selbst beantworten können.
Bei der Auflösung darauf achten, dass die Reihenfolge stimmt: erst klären, ob überhaupt etwas verlagerbar ist, dann ob es lizenziert ist, dann ob zEDC im Weg steht, dann ob Kapazität frei ist. Und ganz am Anfang die Frage, ob CPACF oder ZUS das Thema nicht schon erledigt haben.

WAS BLEIBT

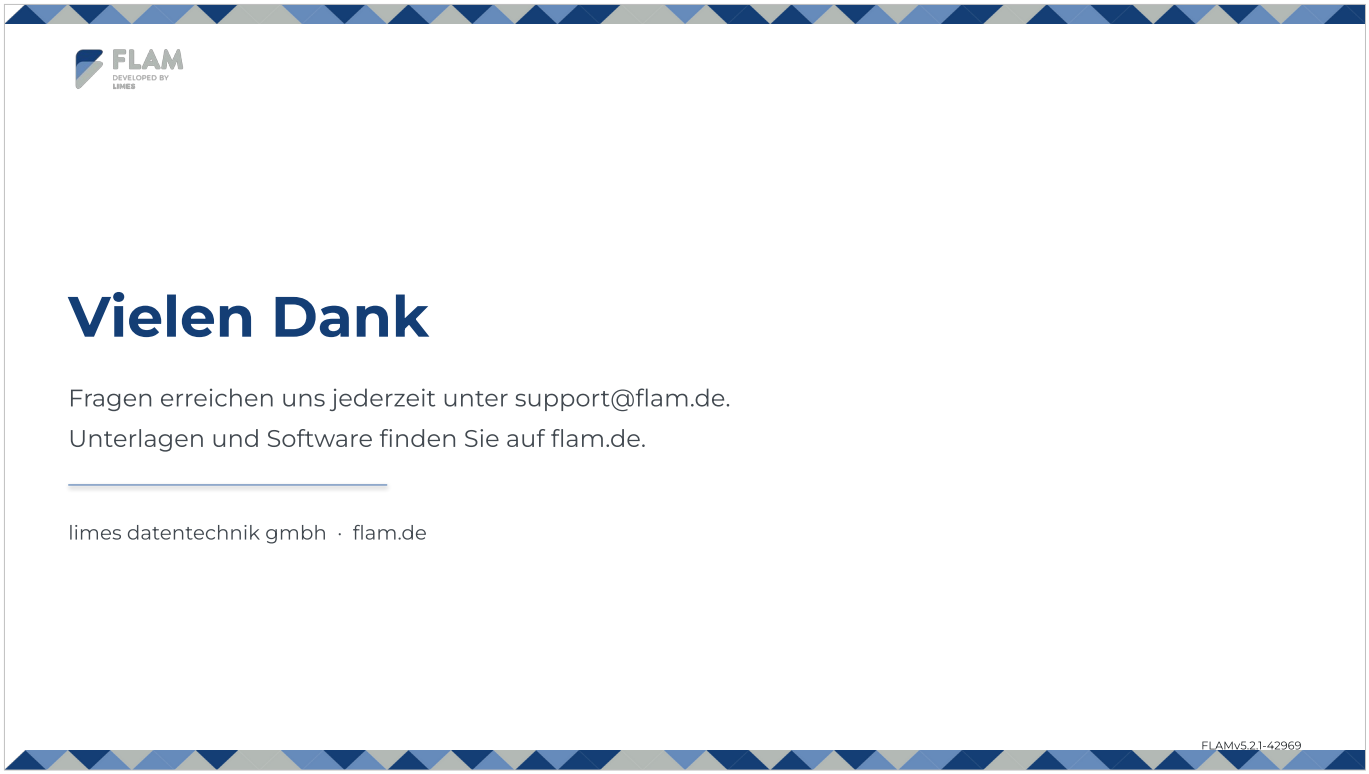
- **zIIP verlagert, es beschleunigt nicht** – die MSU sinken, die Elapsed Time steigt leicht
- verlagerbar ist nur, was **memory-to-memory**, **metal** und **ohne Protokoll** arbeitet – also die `RUN`-Funktionen
- heute sind das die klassischen FLAM4-Algorithmen **ADC, CX7, CX8, VR8** – genau das, was massenhaft im Einsatz ist
- wo **CPACF**, **zEDC** oder die **z/OS Unicode Services (ZUS)** greifen, erübrigt sich zIIP – dort sinkt die echte Arbeit
- die häufigste Ursache für „es verlagert nichts“ ist **zEDC vor zIIP** oder ein belegter zIIP

DAS KANN NUR FLAM

Und der Satz für die Diskussion im eigenen Haus: das ist messbar. Das **Usage Reporting** sagt, was verlagerungsfähig war und was verlagert wurde – die Lücke dazwischen ist die Antwort.

FLAMv5.21-42969

Der erste und der letzte Punkt sind die, die hängen bleiben sollen. Damit endet der Donnerstag.



Abschluss.
Der Support beantwortet auch Fragen, die erst Wochen später beim Umsetzen auftauchen – dafür ist er da, und dafür zahlen Sie ihn bereits. Wenn Sie in Ihrer Umgebung auf einen Fall stoßen, den wir hier nicht behandelt haben, schicken Sie ihn uns: daraus wird in aller Regel ein Beispiel für die nächste Schulung.