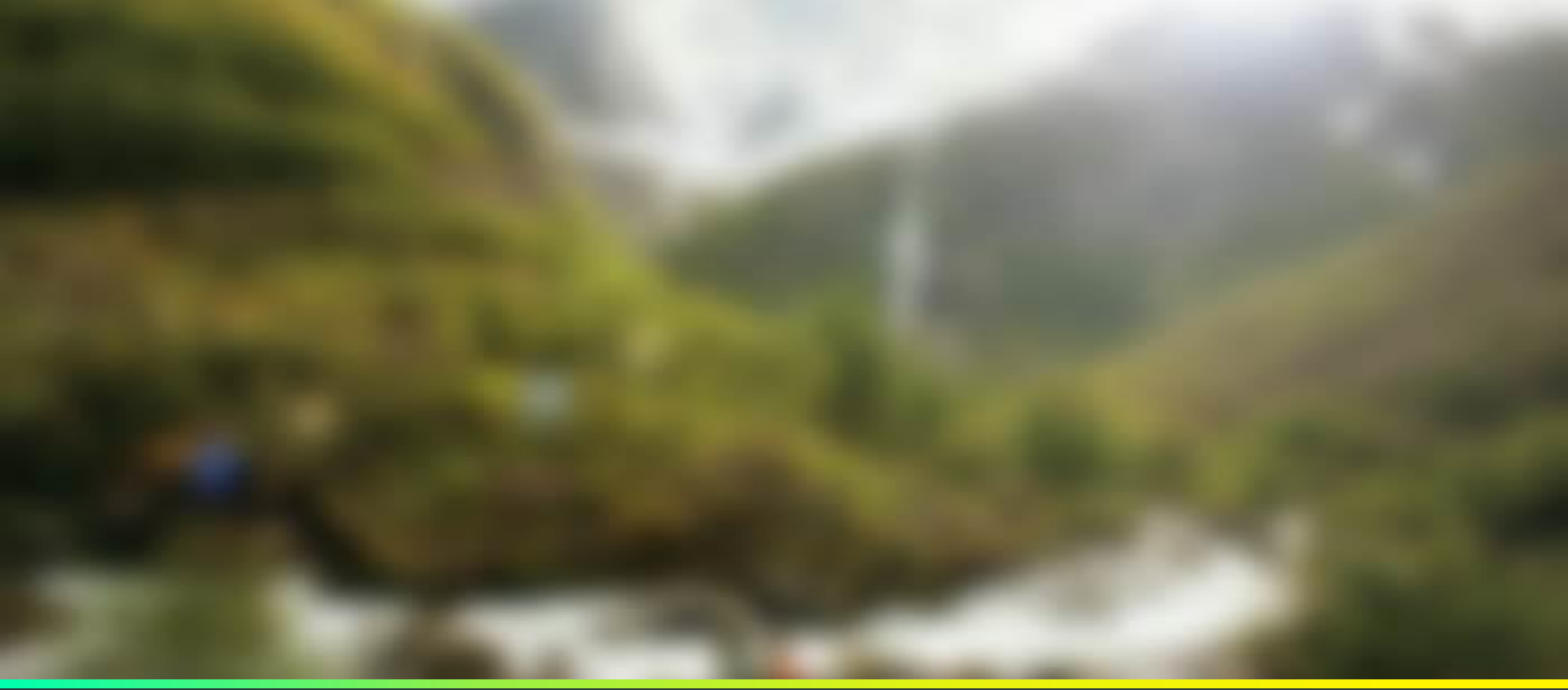


Refactoring at heart

Technischen Schulden begegnen und diese beseitigen

Dresden, 22. Dezember 2018

Dr.-Ing. Claas Wilke, Anwendungsentwicklung

comdirect

Über Mich

- Baujahr 1983
- Studium an der TU Dresden (Dipl.-Medieninf)
- Promotion an der TU Dresden

- Seit 2014 in der „industriellen Praxis“
  - Consulting, Softwareentwicklung
  - Seit 2016 in-house bei comdirect

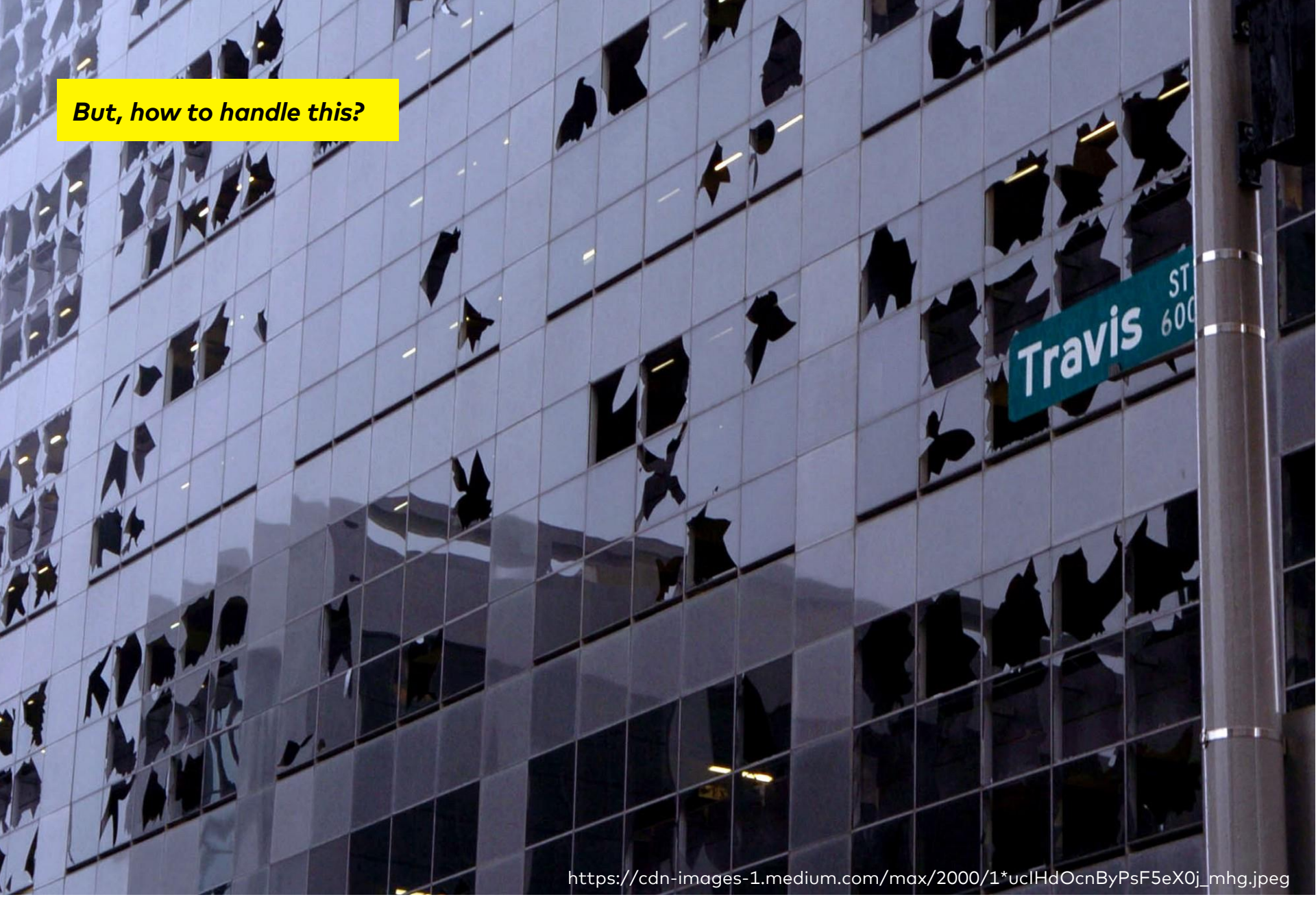
- **Kontakt:**
  claas.wilke@comdirect.de

Das sind wir

- Direktbank seit 1994
- Standorte in Rostock und Quickborn (bei Hamburg)
- über 200 Kolleginnen und Kollegen im IT-Bereich
- Über 1000 Kolleginnen und Kollegen insgesamt
- Keine typische Bank: Innovationsführer im Bankenbereich, Start Up-Mentalität

„One broken window, left unrepaired for any substantial length of time, […] In a relatively short space of time, the building becomes damaged beyond the owner's desires to fix it, […]" **[HT00]**

https://fair.org/wp-content/uploads/2016/07/BrokenWindow.jpg


But, how to handle this?
Travis
https://cdn-images-1.medium.com/max/2000/1*uclHdOcnByPsF5eX0j_mhg.jpeg

Agenda

1. Technologien & Prozesse bei der comdirect
2. Migration von Altsystemen
3. Technische Schulden
4. Refactoring at Heart – Technischen Schulden begegnen

Eingesetzte Technologien

Java EE

- Java 8
- JPA/Hibernate
- JBoss und Tomcat

Spring Framework

- Spring Beans
- Spring Web Flow
- Einige comdirect-Spezifika

Java Server Faces

- Mit Spring Web Flow verheiratet
- Zunehmend Responsive Ansätze

Comdirect Frameworks

- Code-Generatoren für RMI-Aufrufe
- Aspekte / Request Filter
- Eigenes Batch-Framework
- Eigener JSF-Dialekt

Oracle 12 DB

- Jede Menge PLSQL Procedures

C / ProC

- Jede Menge Legacy C-Batches

Automic Job Control

- Workflow Engine für Batch und Prozess-Steuerung

Eingesetzte Tools

Eclipse IDE

- Java Editor, Debugger
- JUnit Runner, Mockito, Emma
- Einige comdirect Plugins

Git

- Sechs zentrale Repositories

Maven, Jenkins

- Ca. 1300 Maven Module
- Buildprozess mit Durchlaufzeiten zwischen 1 und 4 Stunden

Atlassion Jira, Confluence, Bitbucket

- Anforderungsmanagement
- Ticket- / Projektverwaltung
- Code Reviews

Assyst & Splunk

- Vorfallbearbeitung
- Log-Recherche

Outlook

- Warum sprechen, wenn man eine Email schreiben kann...

Software-Entwicklung einer Bank?

Globalisierung des Finanzwesens

- Euro-Einführung
- SEPA 3/4/5 (Umstellung auf IBAN/BIC)

Neue Geschäftsmodelle und Prozesse

- Mobile Payment
- Robo Advising
- Umstellung auf responsive Website

Regulatorische Anforderungen

- EU-DSGVO
- PSD-2

Wie komplex kann das sein?

Sehr komplex!

- 1.300 Maven Module (ohne Framework)
- > 2 Mio. Lines of Code (Java)
- 34.000 Unit-Tests
- 650 Batch-Programme (ca. 50% Java, 50% C)
- 1.400 Datenbanktabellen
- 380 PLSQL-Packages (nicht Procedures!)
- 350 verschiedene Fachlichkeiten
  (nach Domänenbeschreibung)

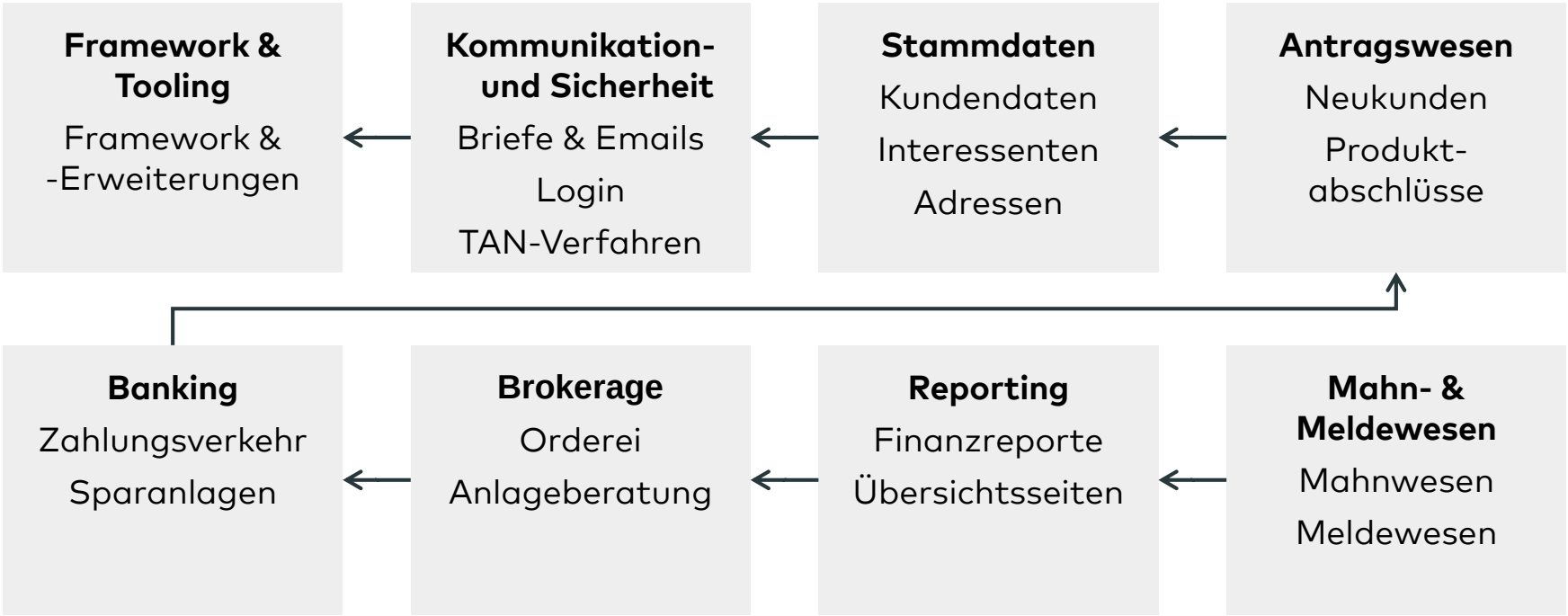
Vertikalisierung der Anwendung und der Entwicklerteams

- Devide and Conquer
- Anwendung wird nach Fachlichkeiten zerteilt
  - Dies betrifft alle Schichten (Frontend, Middletier, Backend)
  - Entwicklerteams übernehmen Verantwortung für eine oder mehrere Komponenten

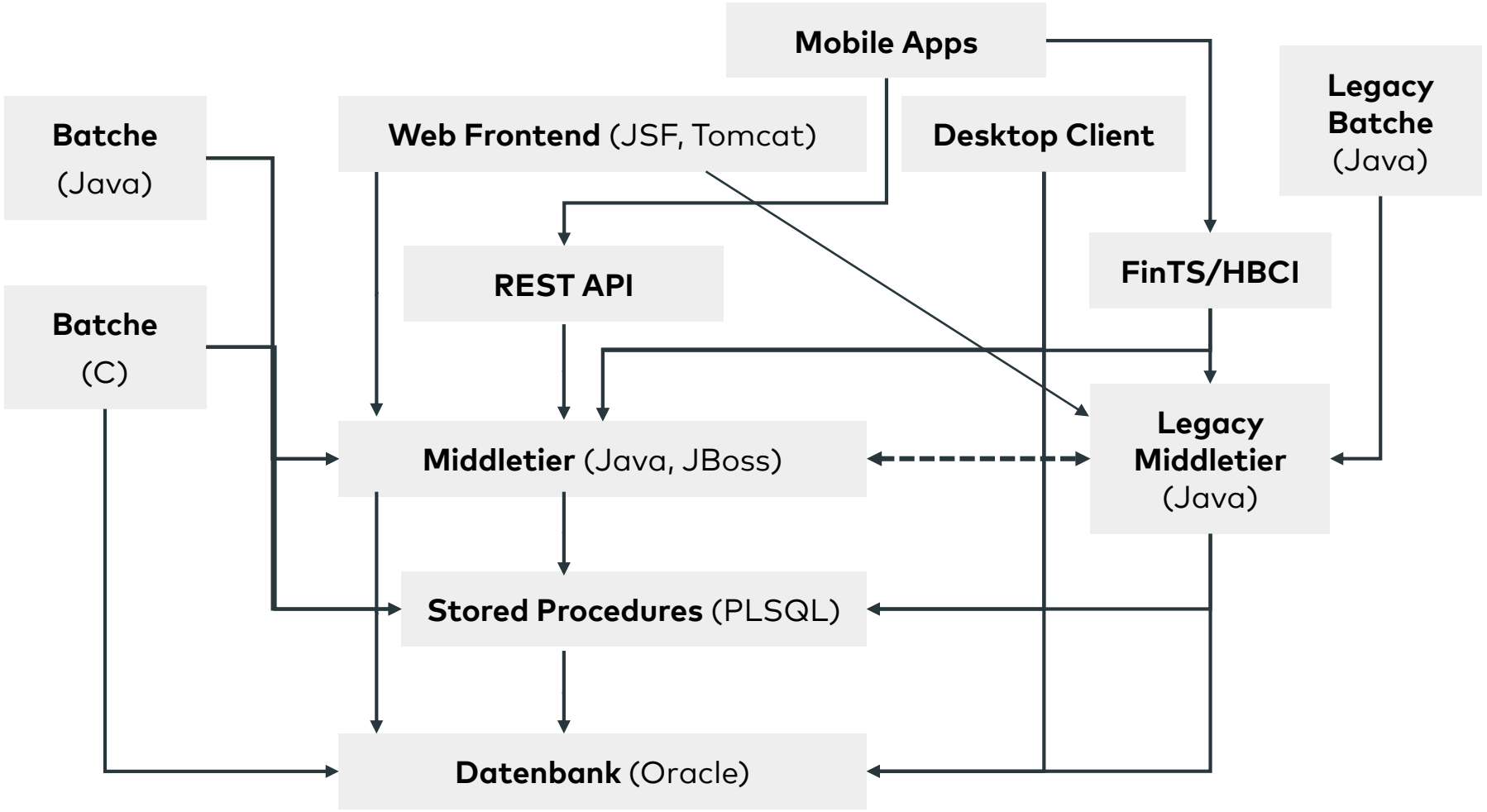
Vorteile:

- Schnittmengen zwischen Teams klein (wenig Konflikte)
- Spezialisierung der Teams auf bestimmte Fachlichkeiten
- Komponentisierung (Micro-Service Architektur) möglich
- Definition von Zugriffsrichtungen möglich

Vertikalisierung der Anwendung bei der comdirect



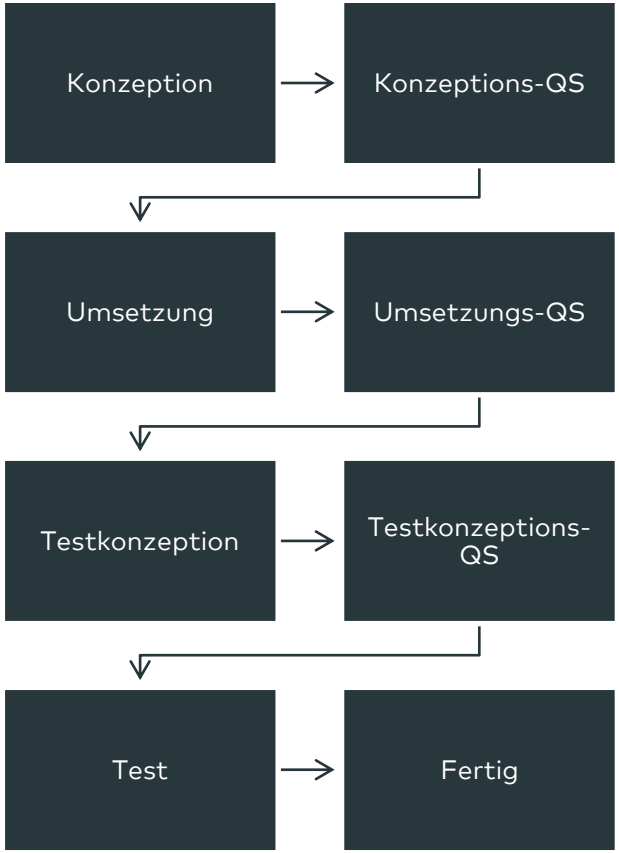
comdirect Top-Level Architektur



comdirect

Entwicklungsprozess und QS für einzelne Anforderungen

- In jedem Entwicklungsschritt gilt das Vier-Augen-Prinzip
- Lösungsskizzen beschreiben Ist-Zustand und Zielbild (Prosa oder Quellcode möglich)
- Code Review bei jeder Änderung (Pull-Request)
- Abnahmetest wird durch Testkonzept beschrieben
- Vollständigkeit wird durch zweiten Tester geprüft



Qualitätssicherung durch Test

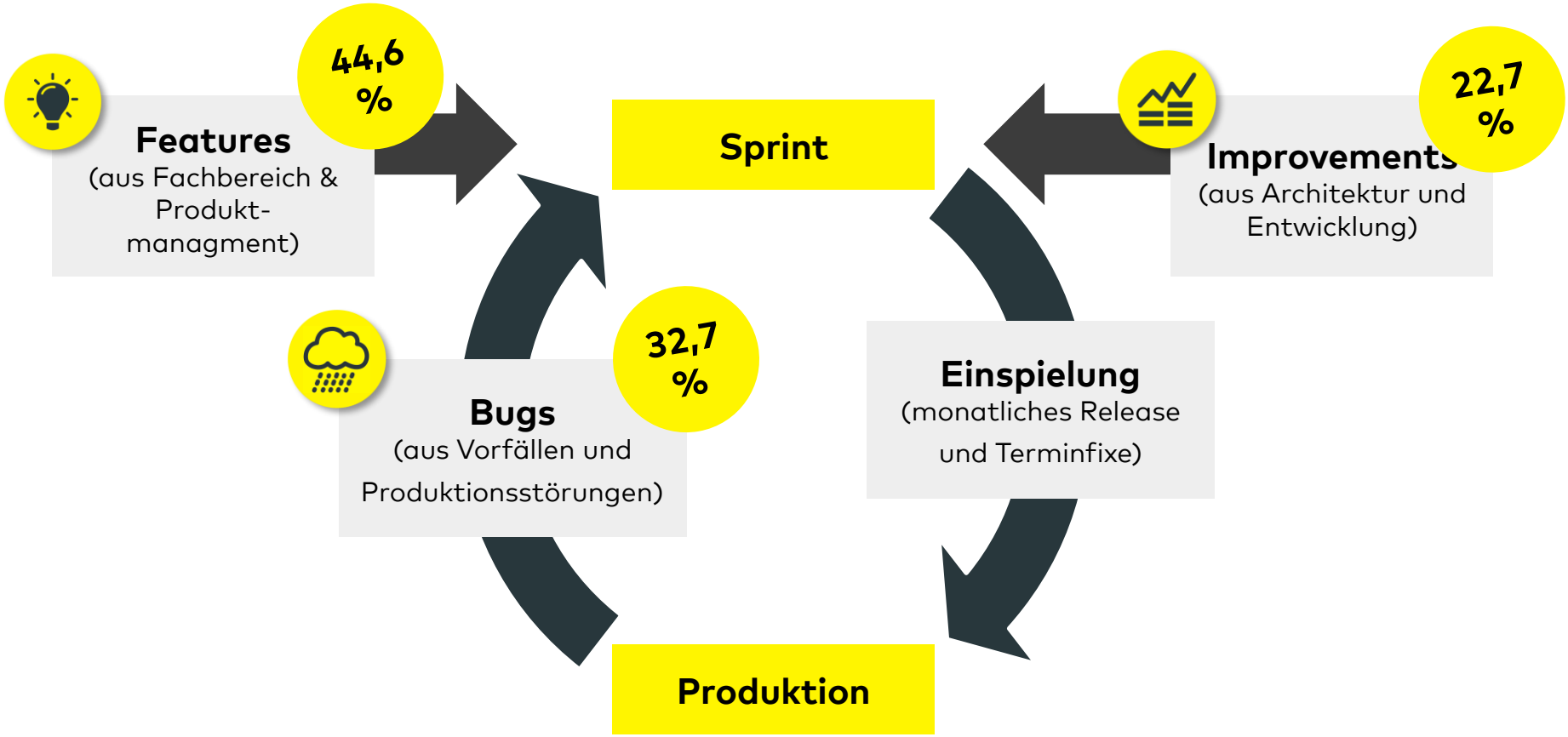
- **Fail early:**
  Je eher Fehler auftreten, desto günstiger sind Korrekturen

→ **Tests in allen Entwicklungsstufen:**

1. Unit-Tests und automatisierte Integrationstests
2. Oberflächentests durch Entwickler
3. Automatisierte Oberflächentests
4. Abnahmetest (durch Tester != Entwickler)
5. Standardtest (vor jedem Major Release)
6. Test durch Kunden in Produktion

- **Jede kundenwirksame Änderung muss getestet werden!**

Drei Arten von Anforderungen



comdirect

Zwischenfazit

- Banken- & Finanzsektor ist durchaus komplex und schnelllebig
- Komplexe, stark verwobene Anwendungsarchitektur
- Vertikalisierung fördert Spezialisierung der Teams und reduziert Abhängigkeiten
- Entwicklungsprozess beinhaltet viele Qualitätssicherungsmaßnahmen
- Neben neuen Fachanforderungen spielen Refactorings und Bugfixes eine große Rolle in der täglichen Entwicklungsarbeit

Agenda

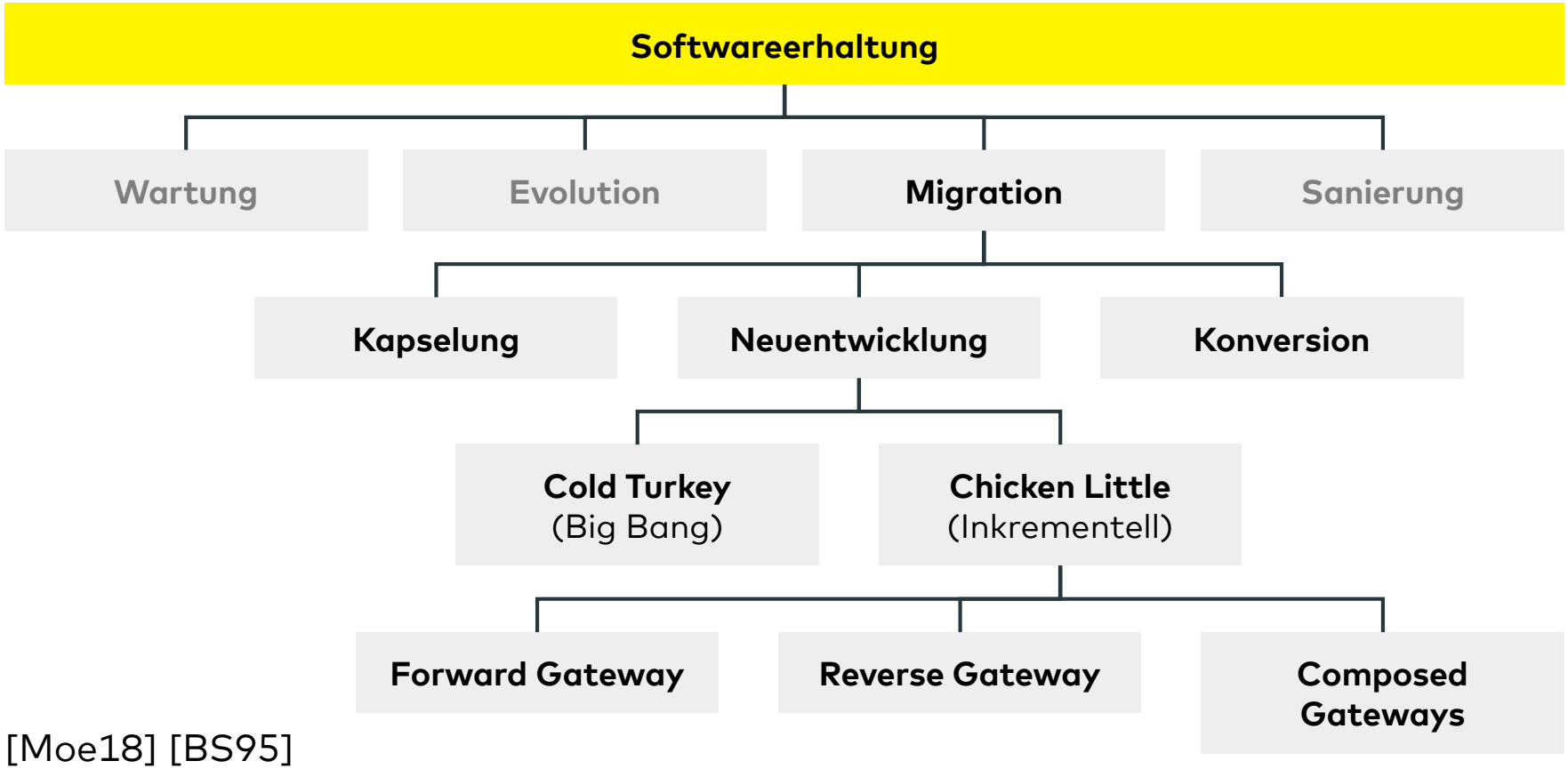
1. Technologien & Prozesse bei der comdirect
2. Migration von Altsystemen
3. Technische Schulden
4. Refactoring at Heart – Technischen Schulden begegnen

Warum migrieren?

1. **Technologiestack veraltet**
   - Business Logik in Stored Procedures
   - C-Batche
2. **Keine Mitarbeiter mit (veraltetem) Knowhow verfügbar**
   - C-Batche
   - PowerBuilder Desktop-Anwendung
3. **Fehlender Support bei Kaufprodukten**
4. **Neuentwicklung günstiger als Weiterentwicklung**
   - Degenerierte, komplexe Anwendung (vgl. Crap Cycle)

[Moe18]

Migrationsstrategien



[Moe18] [BS95]

Cold Turkey / Big Bang

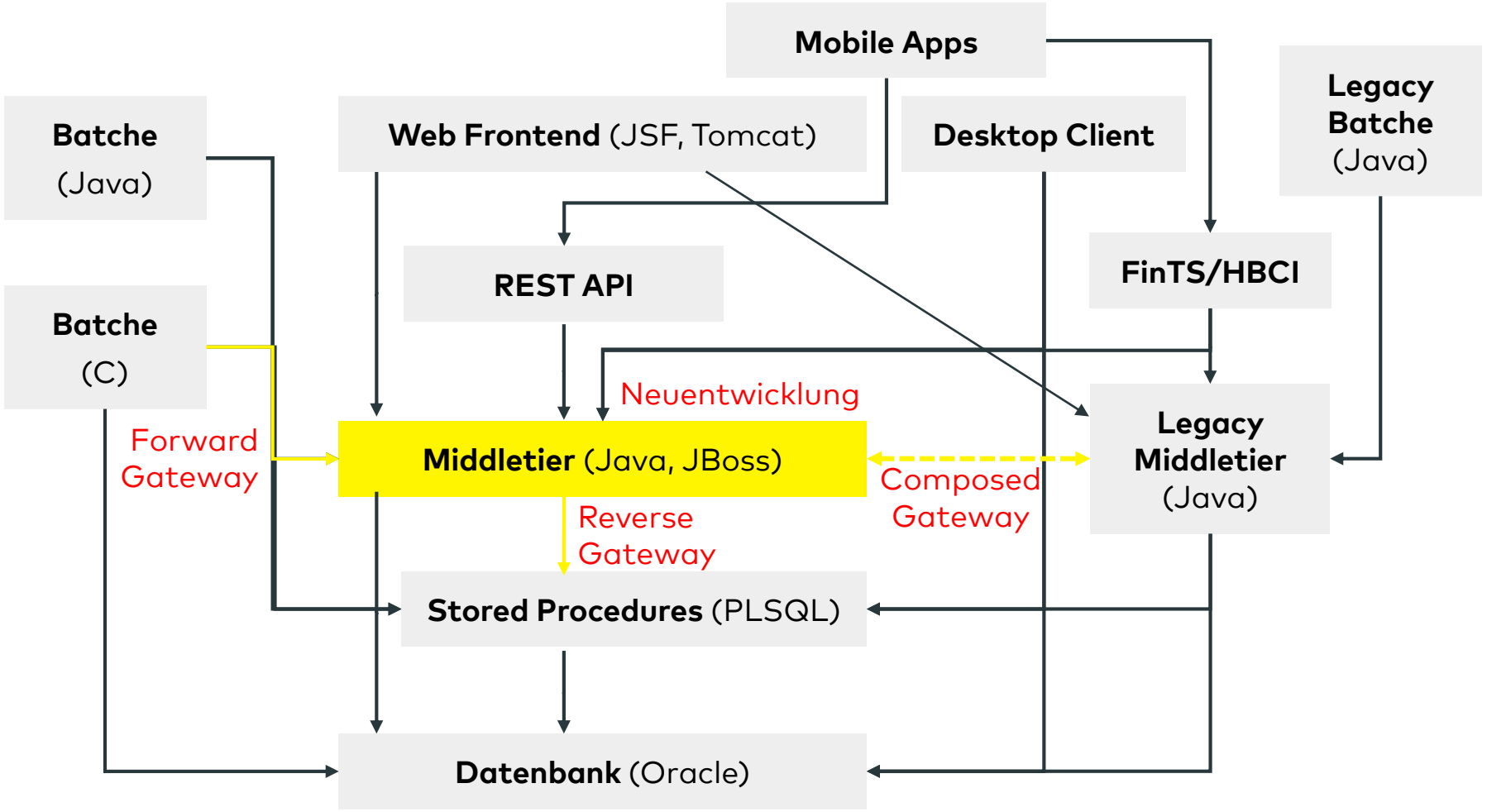
- Anwendung wird im Big Bang migriert
- **Hohe Risiken:** [Moe18]
  - Funktionale Vollständigkeit
  - Verfügbarkeit / Ausfallzeiten
  - Datenqualität
  - Hohes Risiko des Scheiterns
  - Unbekannte Abhängigkeiten
  - Unbekannte Geschäftsregeln
  - Mangelnde Spezifikation

→ **Keine Anwendung bei comdirect**

Chicken Little bei comdirect

- **Softwaremigration erfolgt inkrementell**
  - Cold Turkey zu riskant
  - Alttechnologien werden weiter eingesetzt
  - Ablösung nur schleppend, in Linientätigkeit
- Hochfrequenter Einsatz von Reverse Gateways
- Einsatz von Forward und Composed Gateways

comdirect Top-Level Architektur



comdirect

Chicken Little - Fazit

Pro:

- Migration erfolgt schrittweise
- Geringeres Risiko
- Fallback per Schalter möglich

Contra:

- Migration erfolgt schleppend (nur dann wenn Zeit ist)
- Redundanzen und Duplikate von Business Logik
- Wartungsaufwand erhöht sich
- Abhängigkeiten zwischen Teilsystemen behindern Migration

Agenda

1. Technologien & Prozesse bei der comdirect
2. Migration von Altsystemen
3. Technische Schulden
4. Refactoring at Heart – Technischen Schulden begegnen

Technische Schulden

„Technische Schulden entstehen, wenn bewusst oder unbewusst falsche oder suboptimale Entscheidungen getroffen werden." [Lil17]

Folgen:

- Wartung und Erweiterung werden immer teurer
- Frustration und Demotivation der Entwickler steigt
- Zukunftsfähigkeit sinkt
- Entwicklungsgeschwindigkeit sinkt
- Fehleranfälligkeit steigt

→ **Das System wird zu teuer**

Arten technischer Schulden (1/2)

1. **Implementationsschulden**
   - Anti-Patterns
   - Code-Duplikate

2. **Design- und Architekturschulden**
   - Fachlich
     Beispiel: nur ein Depot pro Kundenverbindung
   - Technisch
     Beispiel: Inline-SQL in Desktop Client

[Lil17]

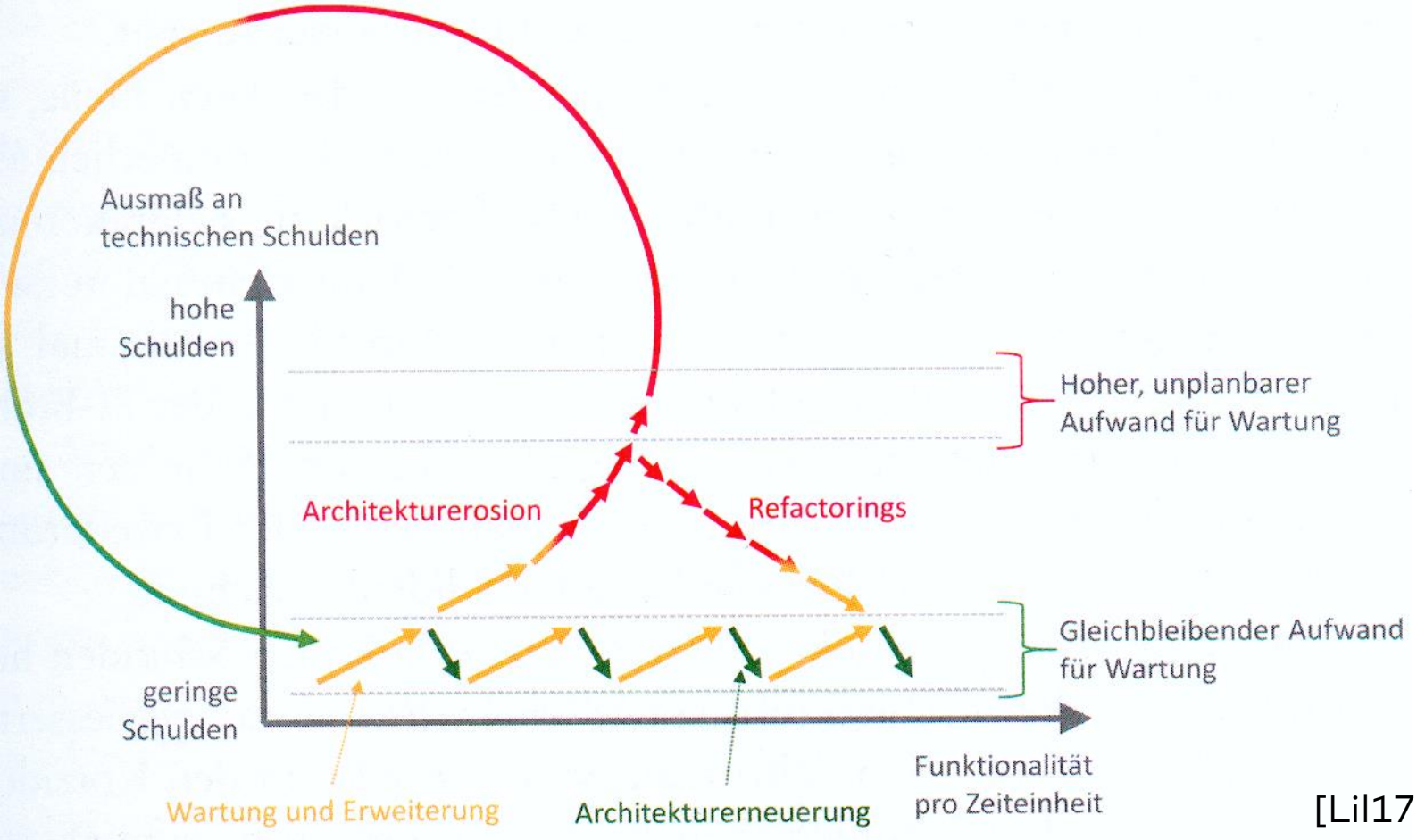
Arten technischer Schulden (2/2)

3. **Testschulden**
   - Schlechte Coverage
   - Edge Cases nicht betrachtet
   - Keine Tests

4. **Dokumentationsschulden**
   - Fachlogik nicht dokumentiert
   - Designentscheidungen nicht dokumentiert

[Lil17]

CRAP-Cycle



[Lil17, Vog15]

Ursachen technischer Schulden

1. „Programmieren kann jeder“
2. Architekturerosion steigt unbemerkt
3. Komplexität und Größe
4. Unverständnis des Managements

[Lil17]

Technische Schulden der comdirect

„Programmieren kann jeder“

- Code-Duplikate
- If-then-else-Höllen

Architekturerosion steigt unbemerkt

- Chicken-Little fördert dieses Problem

Komplexität und Größe

- Mehrfachprüfung derselben Geschäftsregeln

Unverständnis des Managements

- Features, Features, Features …

Agenda

1. Technologien & Prozesse
2. Migration von Altsystemen
3. Technische Schulden
4. Refactoring at Heart – Technischen Schulden begegnen

Technische Schulden sind überall (1/2)

Aus dem Quellcode:

- *„Ganz böser Hack. Sollte unbedingt wieder ausgebaut werden!“* (Alter > 4 Jahre)
- ```
  if (enum == A) return true;
  else {
      switch (enum) {
          default: return false;
      }
  }
  ```
- *„Das gehört eigentlich in XYZ, aber vorher müssen (1), (2) und (3) gemacht werden.“*
  ```

Technische Schulden sind überall (2/2)

Wer sucht der findet

- „Analyseparalyse“ [Moe18] vermeiden
- Nicht jedes Refactoring muss sofort gemacht werden

Refactorings sind teuer

- Qualitäts-/Strukturverbesserung ohne Änderung des Programmverhaltens
- Wer stellt sicher, dass sich das Verhalten nicht geändert hat?

→ Abnahmetest

→ Je größer das Refactoring, desto größer der (Test-)Aufwand!

Der Preis von Refactorings

Managementsicht:

- Refactorings sind teuer
- Es wird nichts geändert
- Die Entwickler beschäftigen sich mit sich selbst
- Kein Change
- Nach dem Refactoring ist vor dem Refactoring

Entwicklersicht:

- Refactorings sind notwendig, unumgänglich
- Reduzieren den Aufwand für neue Anforderungen (CRAP-Cycle)
- Verbessern das Wissen über die Anwendung

Refactorings managen (1/2)

Was du heut nicht kannst besorgen …

- Nicht jedes Refactoring sollte sofort umgesetzt werden
- Aber: es sollte protokolliert und bewertet werden

Improvement Backlog (bspw. in Jira)

- Klassifikation:
  - Fachlichkeit (Synergien erkennen)
  - Testaufwand (ja/nein)
  - Abhängigkeiten (was muss vorher getan werden?)
- Bewertung: Nur was sich lohnt, sollte umgesetzt werden [Her17]

Refactorings managen (2/2)

Gegen das vergessen …

1. **Improvement Backlog regelmäßig prüfen**
   - Zufällig erledigte Tasks schließen
   - Ggf. an neuen Ist-Zustand anpassen

2. **Im Quellcode auf Tasks im Backlog verweisen**
   - „TODO: XYZ-123: Das sollte unbedingt wieder ausgebaut werden."

3. **Veraltete Stellen kennzeichnen und auf Kollegen hoffen:**
   - „@deprecated: Es sollte XYZ genutzt werden."

Refactorings umsetzen (1/2)

1. **Synergien senken Testaufwände/Kosten**
   - Drive-By-Refactoring: gemeinsam mit Feature umsetzen
   - Klassifikation hilft Synergien zu erkennen
   - TODOs im Code helfen Dinge mitzuerledigen

2. **Refactoring als Voraussetzung verkaufen**
   - „Das schaffen wir nur, wenn wir vorher …"
   - Return of Investment benennen

Refactorings umsetzen (2/2)

3. **Refactoring als Feature verkaufen (IT-Fachanforderung)**
   - Nichtfunktionale Anforderungen
     - Performanceaspekte
     - Sicherheitsaspekte

4. **Nutze die Gunst der Stunde**
   - Prio-Hoch-Vorfall, Prod-Störung
   - Problem ansprechen, wenn die „Management Attention“ da ist
   - Kenne deine Baustellen (Baglog)

Do good things and tell about them (1/2)

1. **Neue Schnittstellen und Alternativen kommunizieren**
   - Eine Schnittstelle die niemand kennt wird auch nicht benutzt
   - Beispiel: Neuimplementierung als Alternative für Reverse Gateway

2. **Regelmäßig Metriken auswerten und Ergebnisse kommunizieren**
   - „Das haben wir im letzten Sprint geschafft“
   - „BTW: das sind unsere schlimmsten Anti-Patterns“
   - Gamification fördert die Motivation

Do good things and tell about them (2/2)

Fr 21.09.2018 13:14
Wilke, Claas Dr.
QS-Report für den Team-Branch Oktober
An ⊞ DL_IT_BK_dev

Hallo,

anbei die aktuellen QS-Zahlen für den Oktober-Team-Branch:

Fazit: Der Oktober wird QS-technisch ein ziemlich gutes Release. Die Refactorings haben sich gelohnt. Aus der IVR-Richtung kommt allerdings wieder etwas Ungemach …

team/BK/release/18.10:

- Ausbau von Zugriffen auf den WaehrungDomainAdapter und die STP EUROUTIL.Waehrkonv: **5 Ausbauten**
- Ausbau von Repository-Zugriffen in der Application-Schicht und in Batches: **7 Ausbauten / 3 Einbauten**
  - Zwei neue Zugriffe auf das BatchRepository (FW). Ich habe dafür jetzt einen DomainService beantragt (vgl. BK-6494)
  - Ein neuer Zugriff aus IVR (nicht von uns ☹)
- Ausbau von DomainAdapter-Zugriffen in der Application-Schicht und in Batches: **4 Ausbauten**
- Ausbau von StpAdapter-Zugriffen in der Application-Schicht und in Batches: keine Ausbauten
- Ausbau von LegacyStpAdaptern: **8 Ausbauten**
- Ausbau von Zugriffen aus Batches auf Application-Schicht: **1 Ausbau**
- Ersetzen von org.apache.commons.lang durch org.apache.commons.lang3: **7 Ausbauten**
- Ersetzen von ?-Platzhaltern in ORMs durch sprechende Variablennamen: keine Ausbauten
- Ersetzen von String-referenzierten Usertypes in Hibernate-Mappings durch Konstanten: keine Ausbauten
- Ersetzen von String-referenzierten Beans in Spring durch Konstanten: **26 Ausbauten**
- Ersetzen falscher toString()-Methoden (ToStringBuilder-Aufruf ohne ShortPraefixStyle): **8 Ausbauten**
- Ablösen von TODOs im Code: **8 Ausbauten/12 Einbauten**
- Ausbau von Mobile Flows und Views im Web: **11 Ausbauten**
  - Das entspricht einem Modul

Neue technische Schulden vermeiden

- **Hydra-Symptom:**
  Während wir ein Refactoring umsetzen, schaffen Kollegen die Basis für zwei neue.

- Software ist nie fertig und wird auch nie frei von technischen Schulden sein

- Wir müssen nur schneller alte Schulden beseitigen, als neue Schulden aufgebaut werden

Neue Schulden frühzeitig erkennen (1/3)

1. **Metriken und Anti-Patterns definieren und tracken**
   - PMD- und Checkstyle-Regeln werden bei Pull-Requests automatisiert geprüft
2. **Kritische Bereiche gesondert überwachen**
   - Datenbank-QS bei Anpassungen an Stored Procedures oder HQL-Queries
3. **Skriptbasiert Antipatterns erkennen und mit dem Vorgänger-Release vergleichen**
   - Neue Code Smells so schnell identifizieren, dass sie noch vor dem Release wieder ausgebaut werden können (kein zusätzlicher Testaufwand)

Neue Schulden frühzeitig erkennen (2/3)

4. **Falsche Design-Entscheidungen frühzeitig erkennen**
   - Lösungsskizzen als erstes Entwurf
   - QS der Lösungsskizzen erfolgt nur von wenigen Personen, die die Zielarchitektur im Blick behalten

5. **Bekannte „Problembären“ im Auge behalten**
   - Bei manchen Kollegen ist die Wahrscheinlichkeit neuer technischer Schulden höher als bei anderen …
   - Jira-Task beobachten
   - Gezielt QS-Aufgaben übernehmen oder überprüfen

Neue Schulden frühzeitig erkennen (3/3)

6. **Falsche Entwicklungen ansprechen und erklären**
   - Kein Finger-Pointing!
   - Diplomatisch vorgehen und auch Positivbeispiele belohnen

7. **Schulen, schulen, schulen …**

Success Stories (1/2)

Nichts ist wichtiger, als den Fortschritt zu messen und sich selbst zu belohnen (Gamification)

1. **Reimplementierung Umsatzanzeige**
   (Stored Procedures, Middletier und Frontend)
   - Dauer: 18 Monate
   - Erfolg: Reduzierung der DB-Last für diese Anfragen um 80%

2. **Refactoring Erfassung von SEPA-Überweisungen**
   - Dauer: 4 Monate
   - Erfolg: Code-Reduktion um ca. 50%

Success Stories (2/2)

3. **Skript-basiertes Tracken von Antipatterns**
   und schnelles Eingreifen bei neuen Treffern
   - Dauer: bisher 11 Monate
   - Erfolg: bisher 500 Antipatterns beseitigt

4. **„Instabile" Integrationstests systematisch migrieren**
   - Dauer: bisher ca. 2 Jahre
   - Erfolg: 75% der Tests migriert, Laufzeitreduktion ca. 50%

Konklusion

- Es ist einfacher ein Softwaresystem zu bauen als es zu warten
- Jedes Softwaresystem verrottet im Laufe seiner Lebenszeit
- Gegenmaßnahmen
  - Refactorings systematisch tracken und sammeln
  - Bei Gelegenheit Refactorings umsetzen
  - Neue Schulden vermeiden
- Refactorings gibt es nicht umsonst
- **Aber: sie zahlen sich auch aus**

Literatur (1/2)

[BS95] Michael L.Brodie, Michael Stonebraker: *Migration Legacy Systems*, Morgan Kaufmann, 1995.

[Her17] Corinna Hertweck: *aim24 in der Praxis – Softwarebauwerke restaurieren*.
In: JavaMagazin 12/2017, Software & Support Media GmbH, Frankfurt am Main, 2017.

[HT00] Andrew Hunt, David Thomas: The Pragmatic Programmer: From journeyman to master. Addison-Wesley, Boston, 2000.

Literatur (2/2)

[Lil17] Carola Lilienthal: *Langlebige Software-Architekturen*, 2. Auflage. dpunkt.verlag, Heidelberg, 2017.

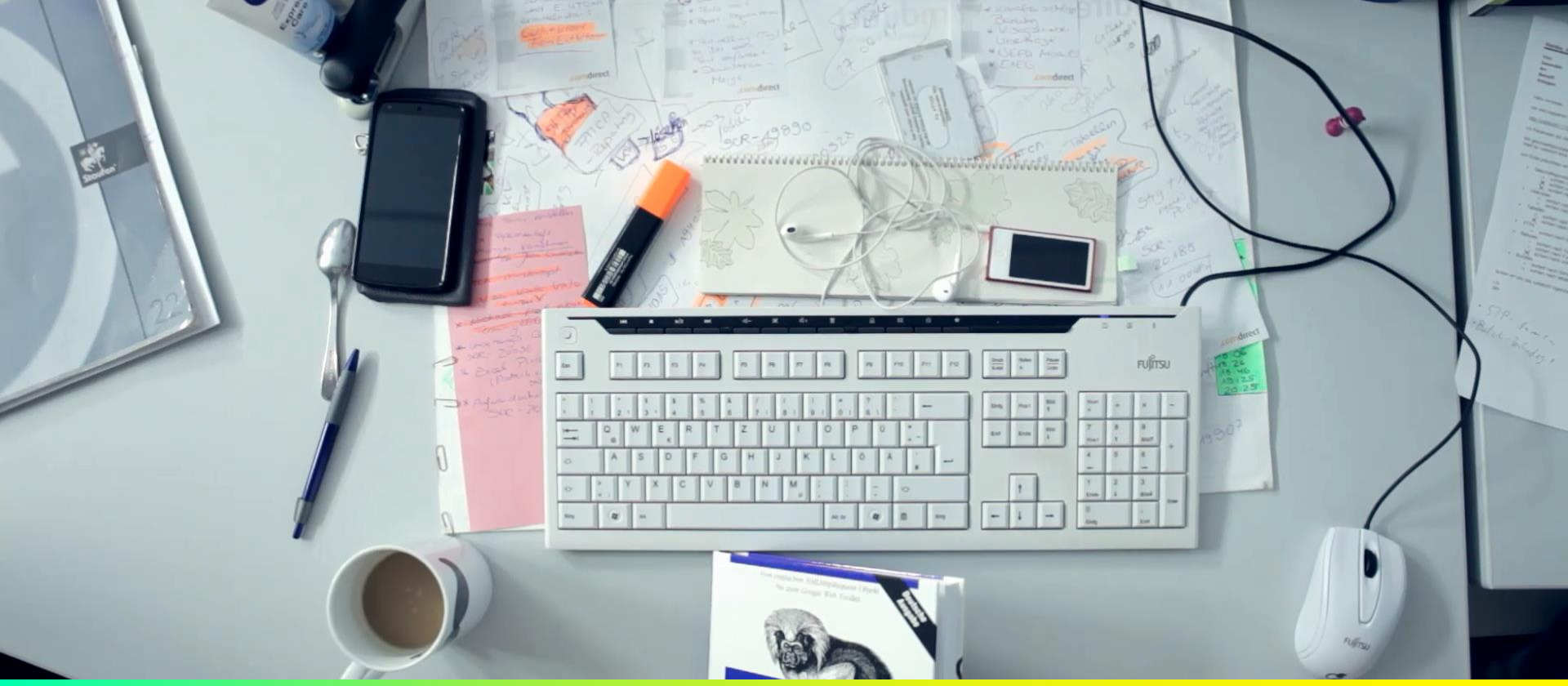
[Moe18] Matthias Möser: *Abschied nehmen vom Legacy-System*. In: JavaMagazin 3/2018, Software & Support Media GmbH, Frankfurt am Main, 2018.

[Vog15] Peter Vogel: *Domain-Driven Design: Everything You Believe Is Wrong!* In: Visual Studio Magzine, 2015: https://visualstudiomagazine.com/articles/2015/07/01/domain-driven-design.aspx

You are wanted!

- Java-Entwickler (m/w)
- App-Entwickler (m/w)
- KI-Entwickler (m/w)
- Softwaretester (m/w)
- Praktikanten (m/w)

comdirect



Wir bieten

- Anspruchsvolle Aufgaben statt „Schema F“
- Customer on Site: enge Abstimmung mit dem Auftraggeber
- Flexible Arbeitszeiten, Fahrtkosten- und Essenszuschuss, Forschungszeit, Sabbatical, Altersvorsorge, Massage
- Zeit für Qualität in der Software-Entwicklung
- Weiterbildung, Konferenzen, Coding-Events

comdirect

Keep in touch:

www.comdirect.de/karriere

comdirect