Software-Entwicklungsumgebungen
Datentypen, Fehlerbehandlung

Vorlesung 9: Software-Engineering für große Informationssysteme

TU-Wien, Sommersemester 2002

Wolfgang Keller

Motto dieser Vorlesung

Der beste Code ist der, den Sie nicht schreiben ...

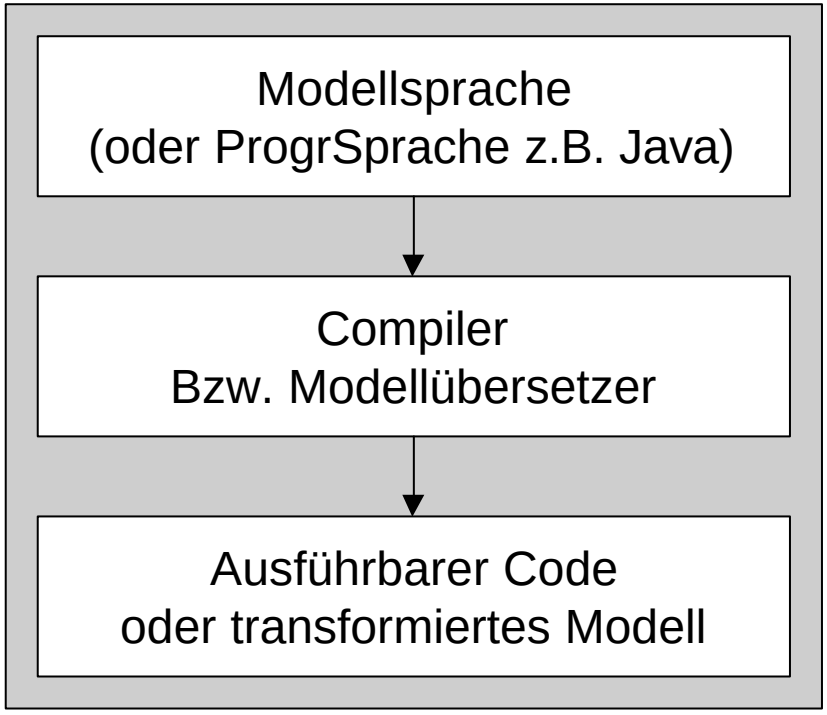
Überblick

- Idee hinter Softwareentwicklungsumgebungen
  - MDA – Model Driven Architecture
- Elemente einer SEU
- Datentypen
- Fehlerbehandlung
- Zugriffsschichten

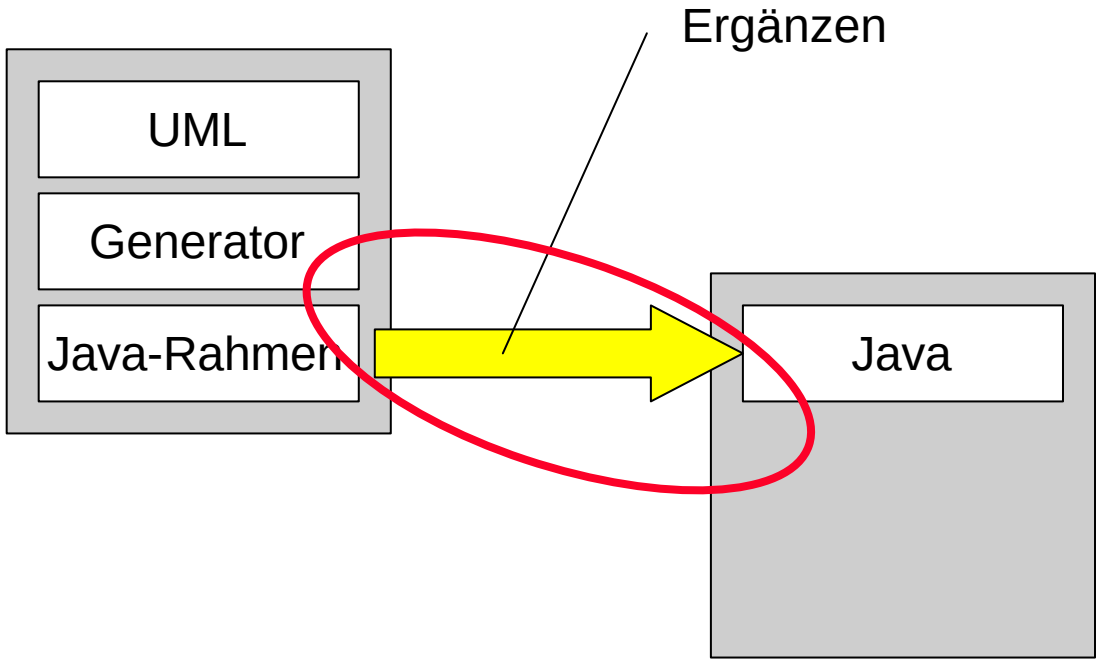
Der Traum hinter Software-Entwicklungsmugebungen

- Abstraktionsniveau höher ziehen
- Am liebsten Modellieren statt Programmieren
- Unabhängig von Plattformen arbeiten
- Möglichst viel Code nicht selbst schreiben
  - sondern sicher generieren lassen
  - oder noch besser eine entsprechende Meta Maschine laufen lassen .. Und nur dann compilieren, wenn Interpretation zu langsam ist ..

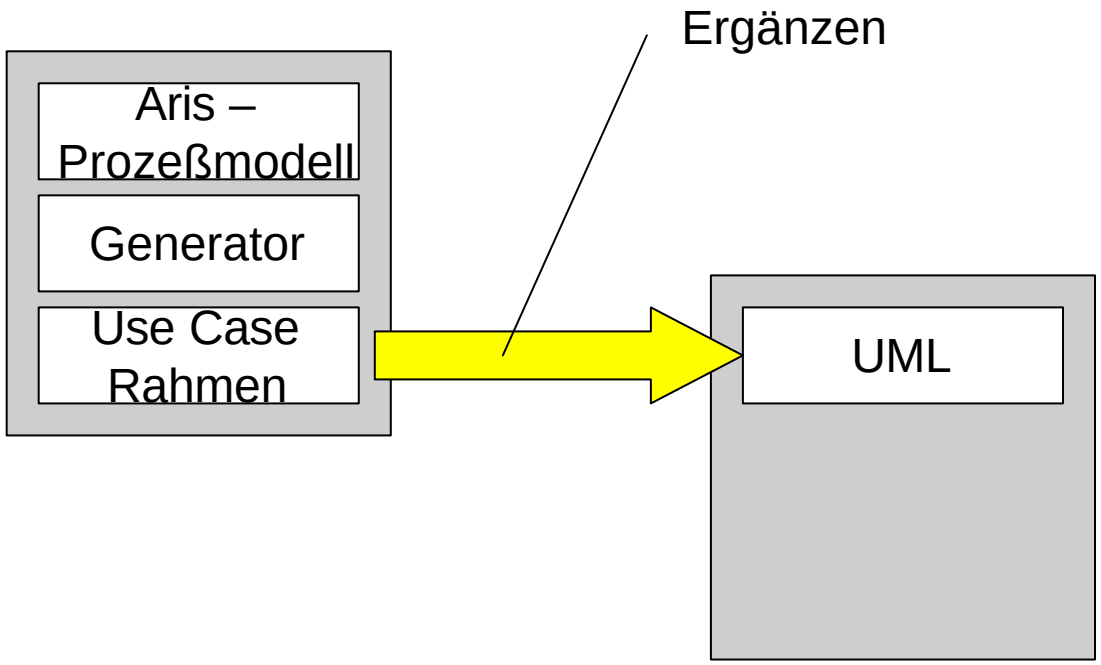
Begriff eines Programmierstacks (Pstacks)



Nächstes Beispiel
2 PStacks



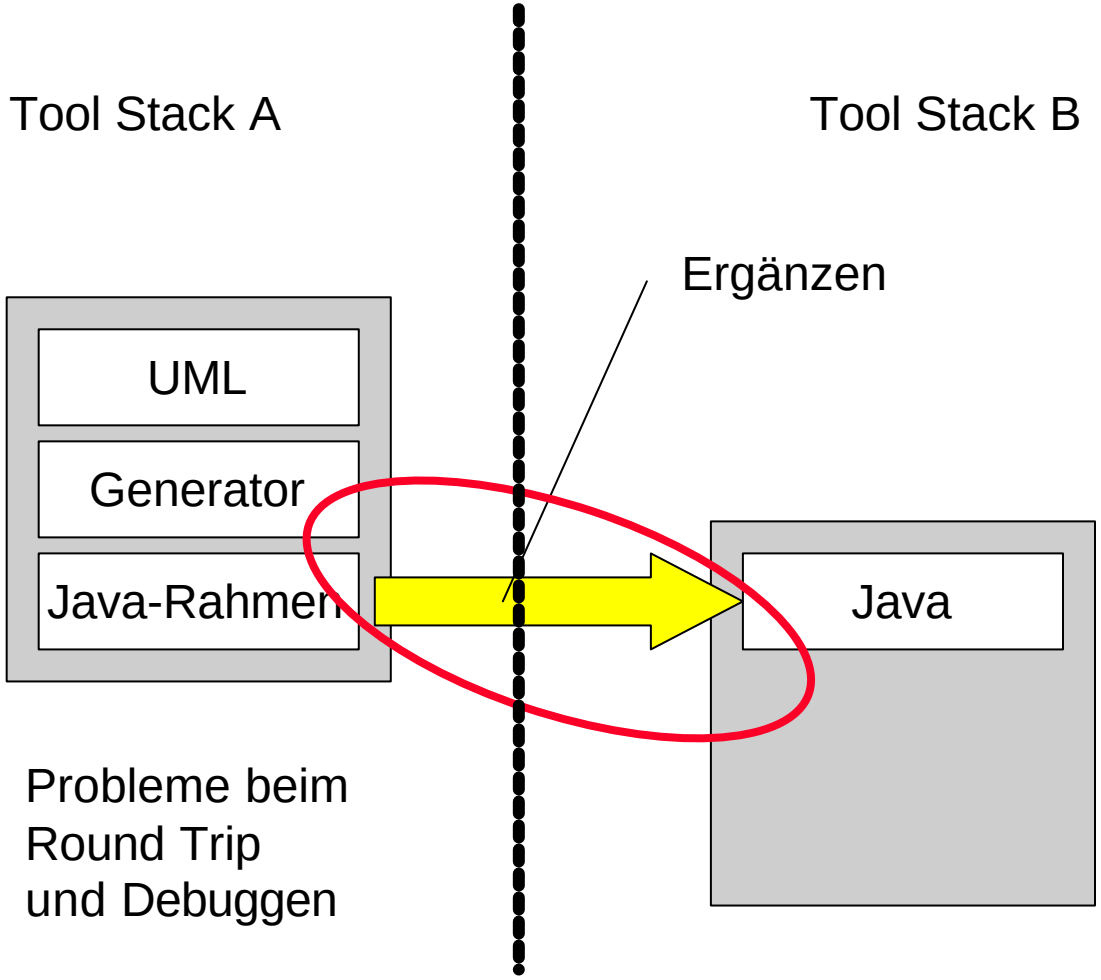
Und noch einer der dritte PStack



Wo funktioniert so etwas gut?

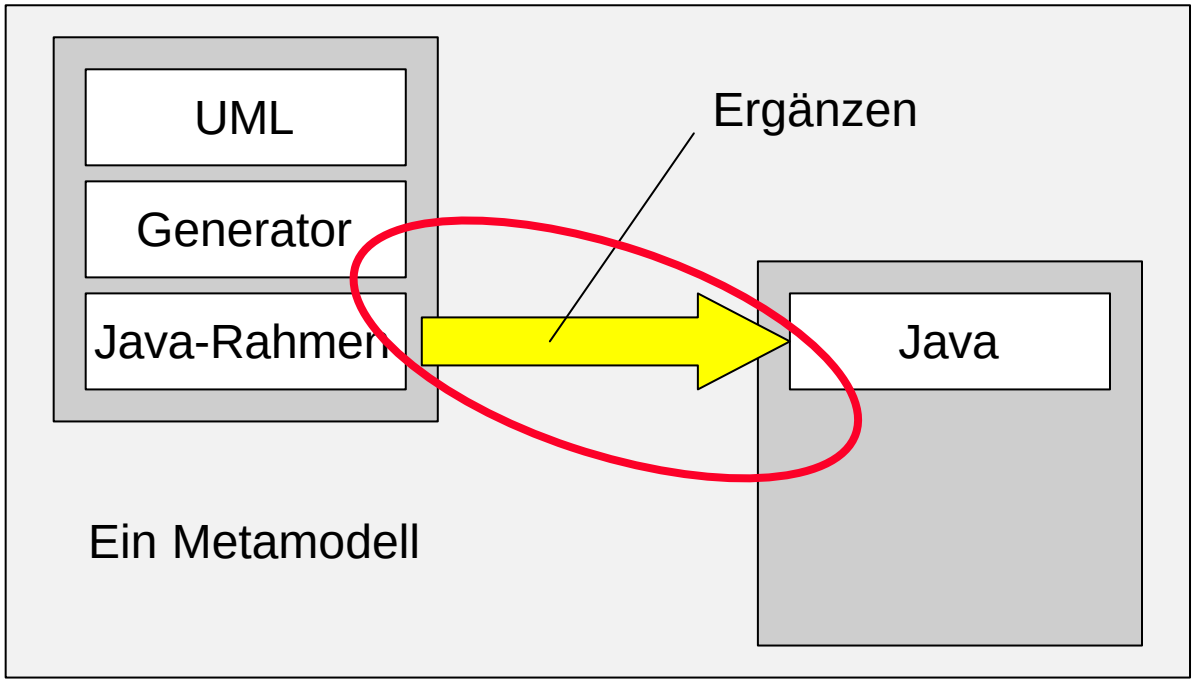
- Funktioniert gut, wenn man manuell überhaupt nichts mehr ergänzen muss
  - Compiler
  - Das war auch bei Compilern nicht immer so
    - Handoptimierter Code
- Funktioniert dann nicht gut, wenn man generiert und keinen „perfekten Roundtrip“ hat,
  - also die Ergänzungen nach jeder Generierung neu machen muß ..

Heute üblich ...

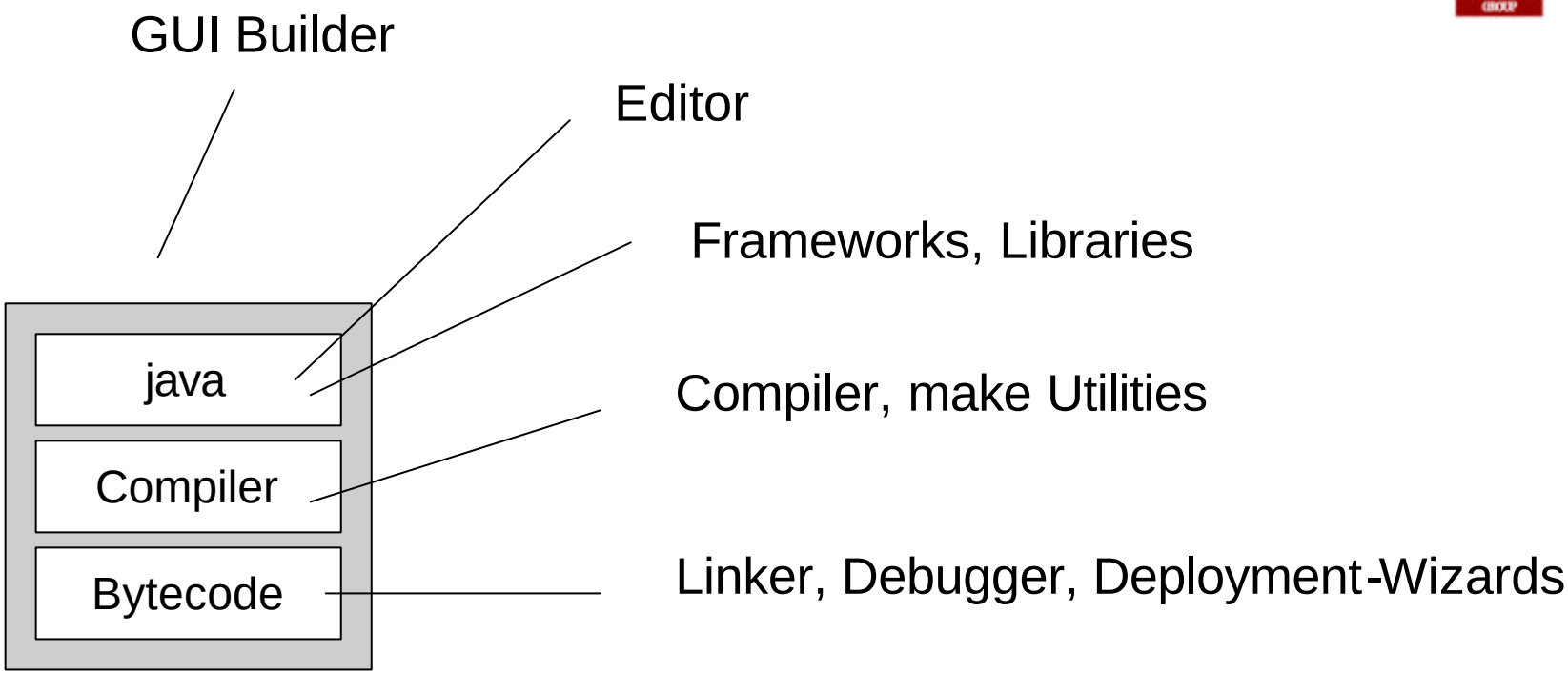


Besser Modellintegration

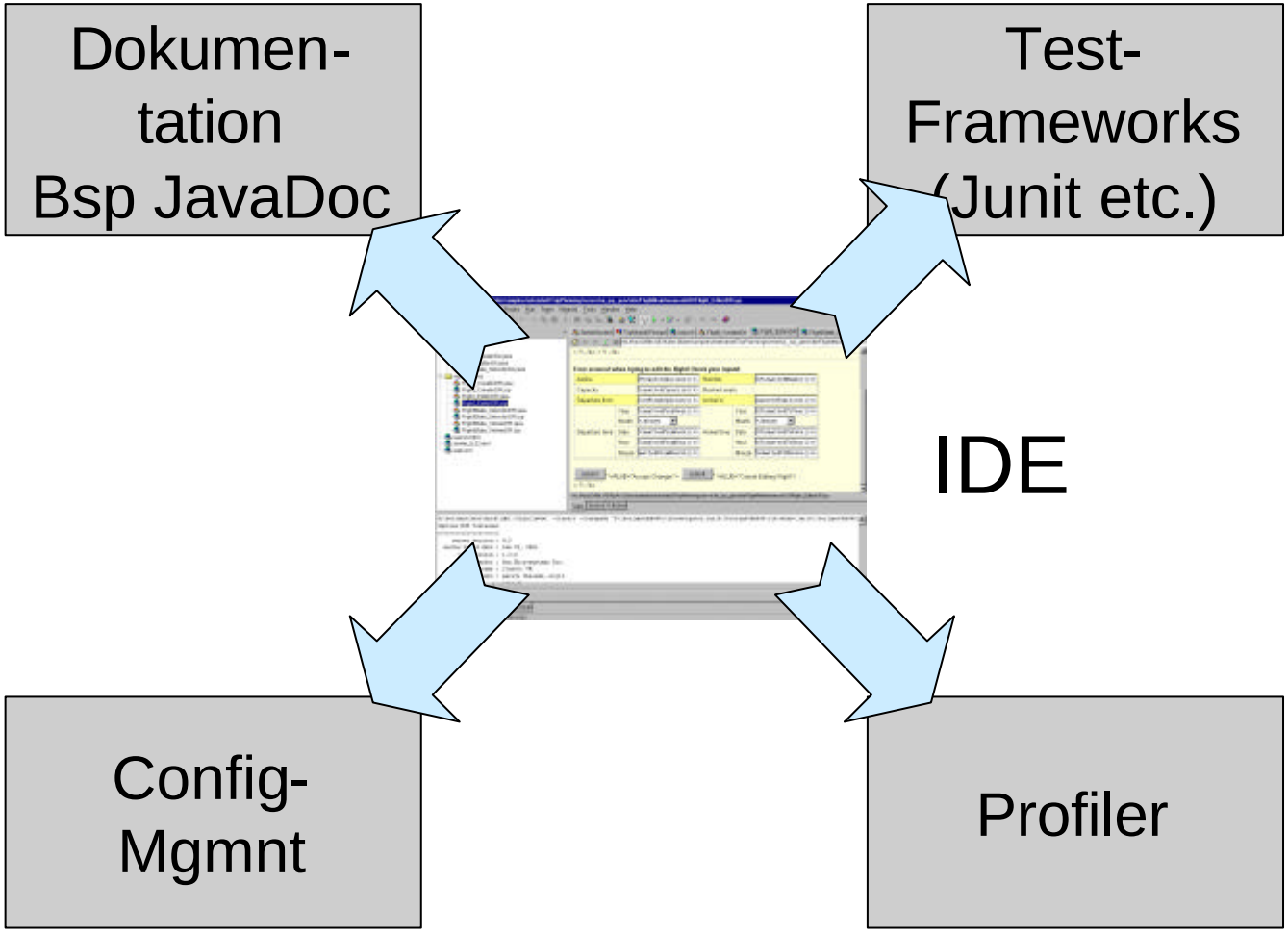
Keine Grenze, nur ein Tool, der beide Meta-Modelle beider PStacks in sich vereint



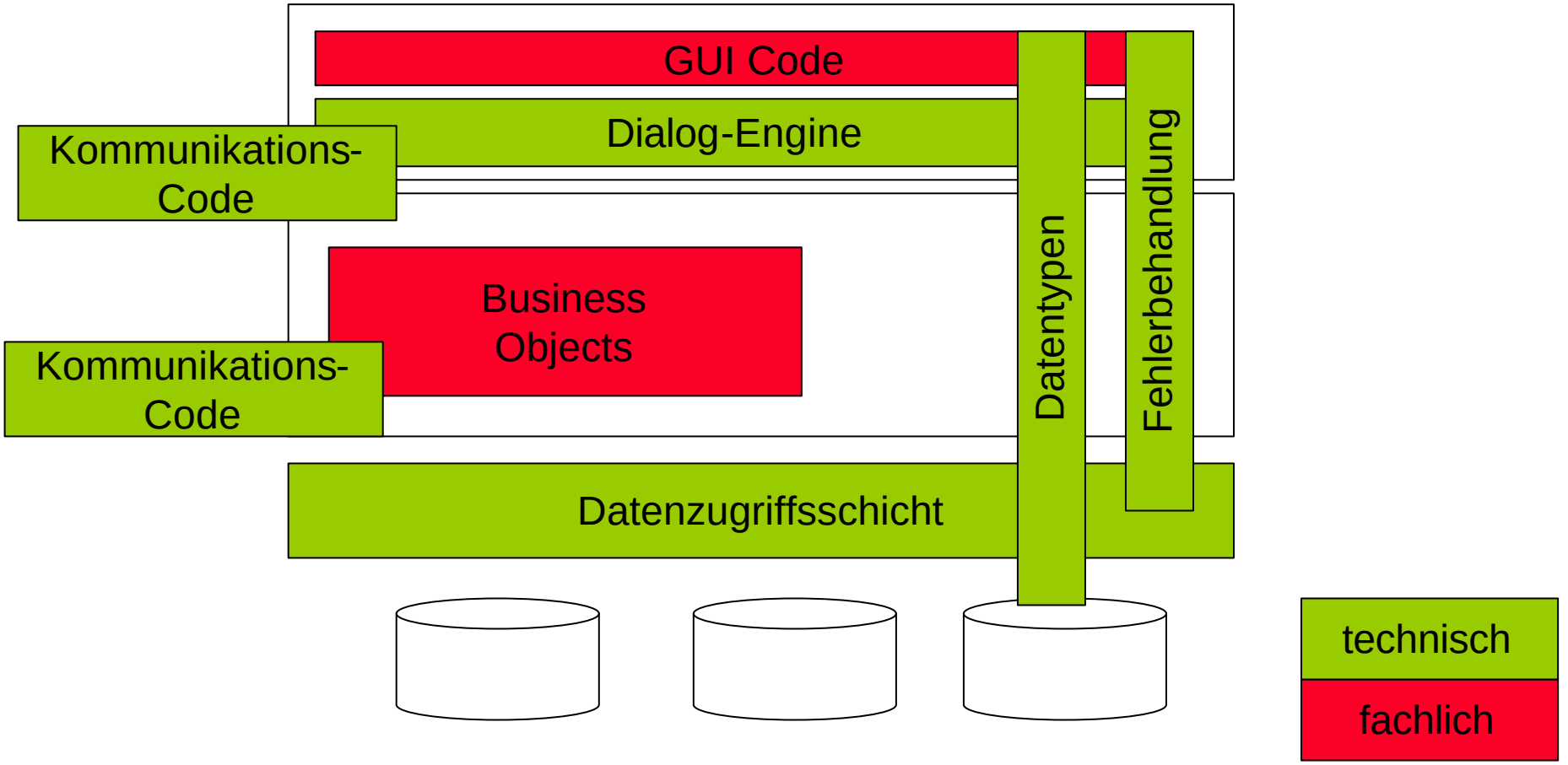
Elemente einer SEU



Elemente einer SEU – Weiteres



Wo gibt es da immer noch „langweiligen fachunabhängigen“ Code



Also ..

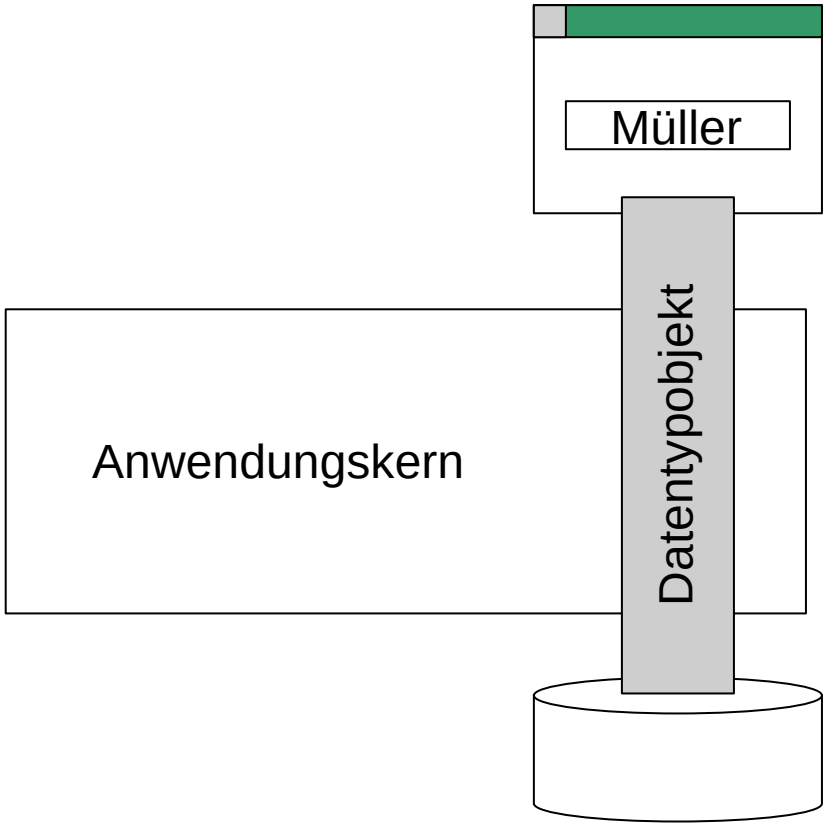
- Datentypen
- Fehlerbehandlung
- Zugriffsschichten
  - Benutzen Datentypen
  - Plus Codegenerierung basierend auf CRUD
  - Kann man nachlesen
- Dialogengines
  - Behandlung bei Spezifikation reicht aus – wie man eine State Machine schreibt kann man im Internet nachlesen

Datentypen
Motivation

- Basistypen der Programmiersprachen sind nicht geeignet, um fachliche Dinge auszudrücken

- int, char, string, double, decimal, ...

Datentypen – Motivation Nutzung

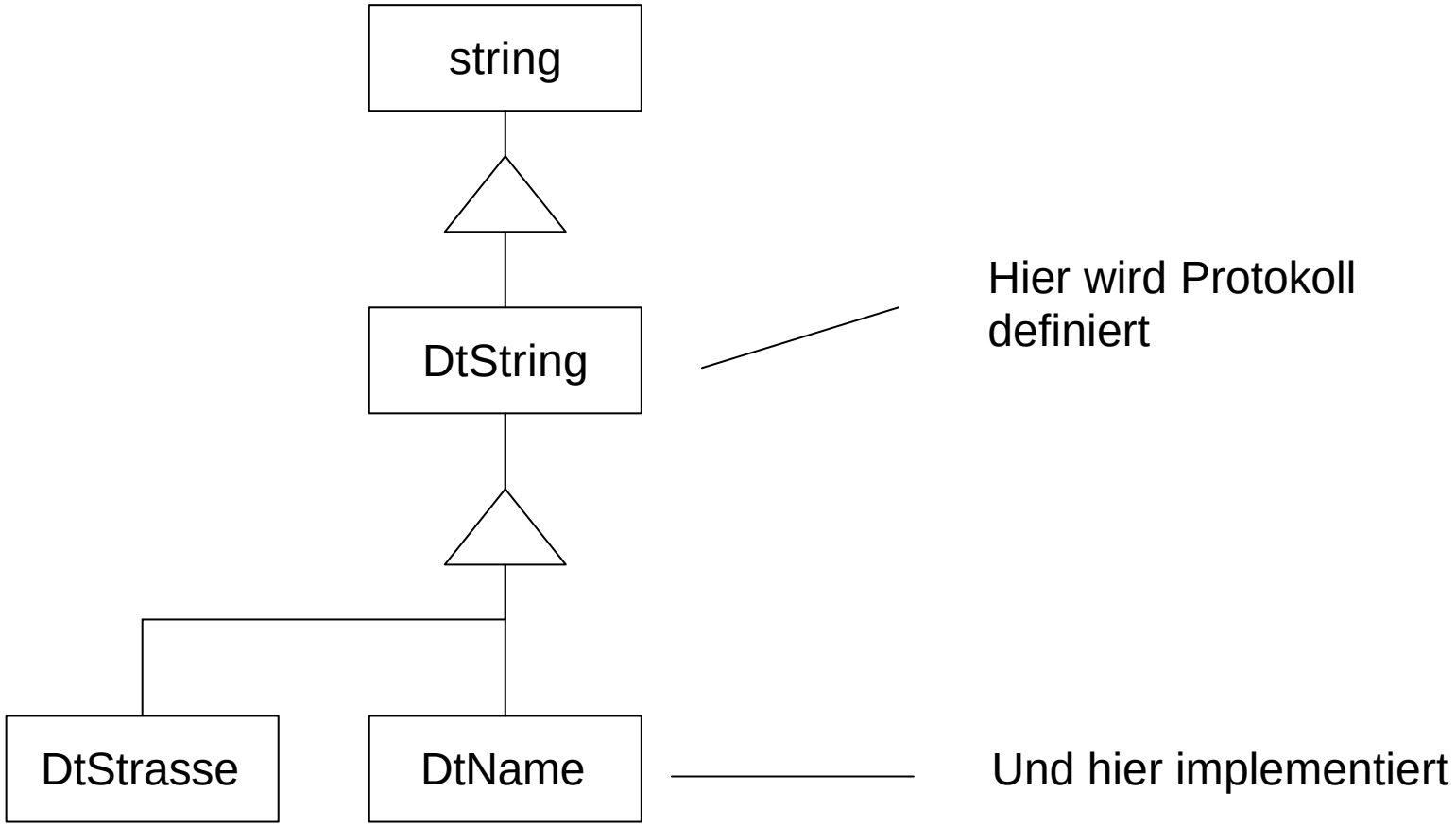


setFromGUI (String)
isValid()

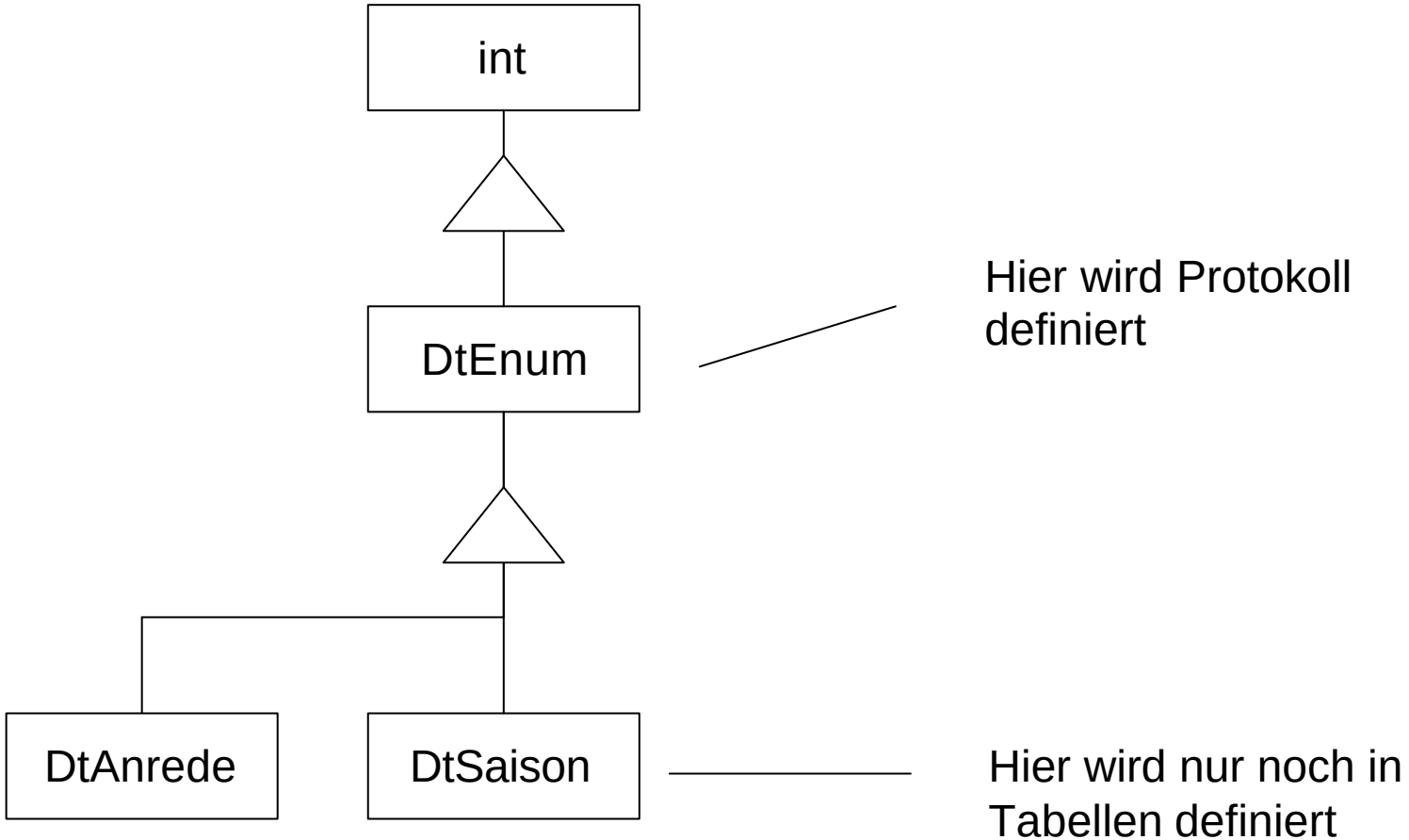
Check()

String convert2DB()

Typische Klassenhierarchien



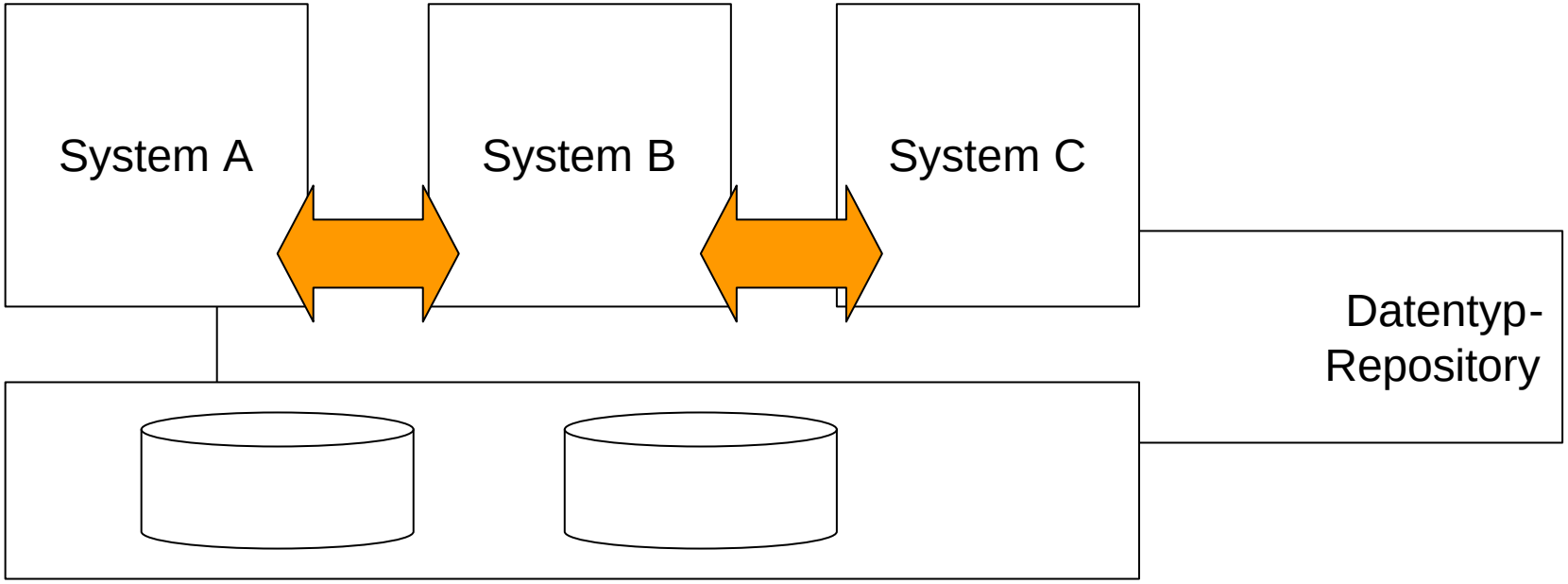
Typische Klassenhierarchien
Weiteres Beispiel



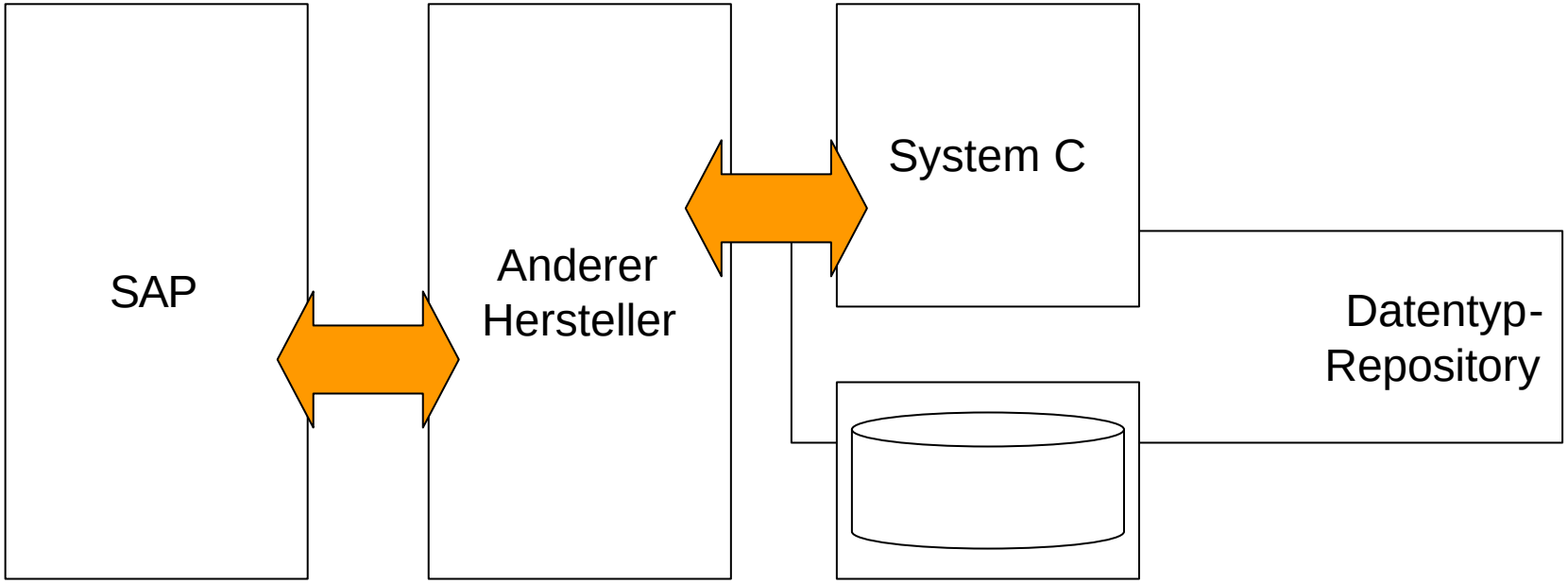
Datentypen
Was man meist so braucht

- Datum
- Aufzählungen
  - Auch dynamisch erweiterbare
- Eingeschränkte Strings
- Währungen, Geld, ..
- Numerische Typen
  - Mit speziellen Formatierungen

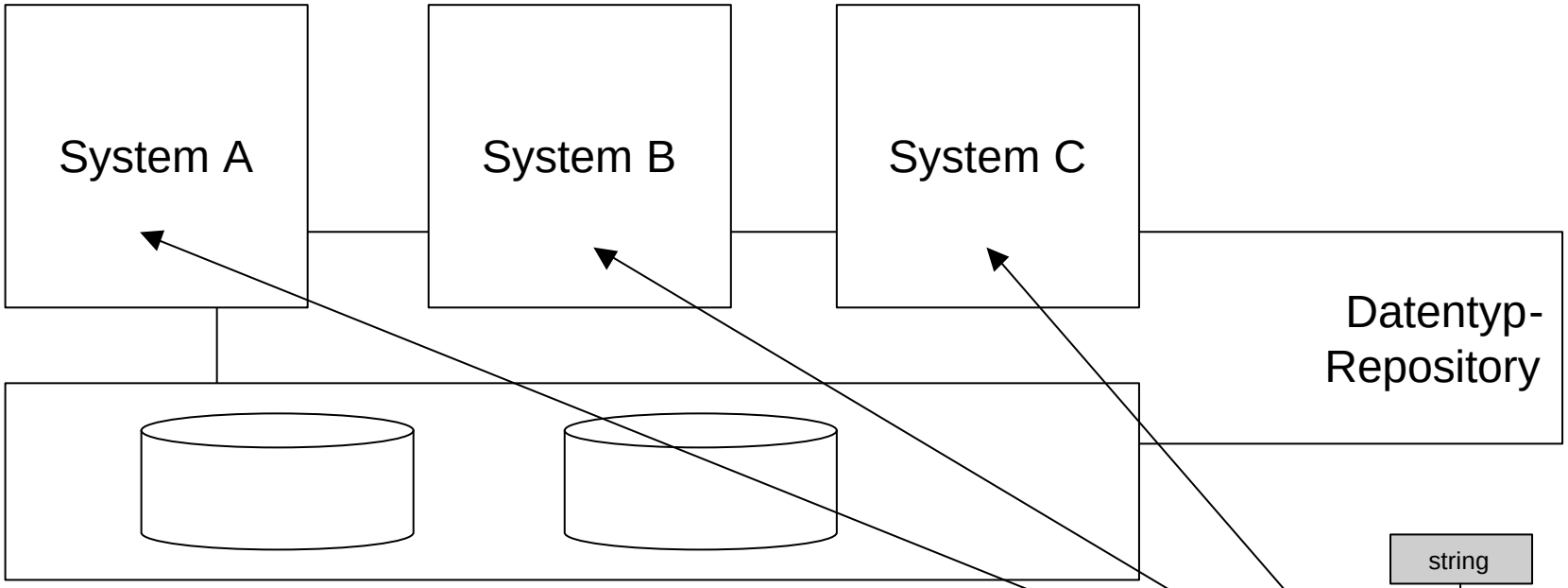
Diskussion: Welche Vorstellung liegt dem zugrunde?

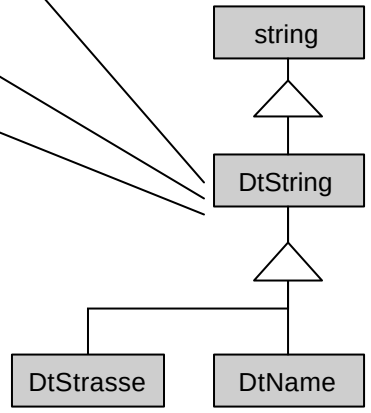


Die Praxis dazu sieht aber anders aus



Diskussion: Es ist nicht alles Gold was glänzt





Und jetzt ändern wir den Code von DtString
Und dann den von DtStrasse

Datentypen
Vor- und Nachteile

- Fachliche Typen statt technischer
- Änderungen werden überall durchgezogen
- Man kann nur schwer welche vergessen
- IDE erstellt Verwendungsnachweis
- Vereinfacht Wartung

- Änderungen in der Basisklassen verursachen Nachkompilierung überall
- Damit „Abhängigkeiten“ im Code

- Wirkung eingeschränkt, da man heute oft Systeme verschiedenster Hersteller integrieren muss
- Repositories eines Systems sind da ziemlich wirkungslos
- Repository-Konzept bei heterogenen Systemen meist nicht vorhanden

Fehlerbehandlung

Fehlerbehandlung

- Bei prozeduralen Sprachen
- Bei OO-Sprachen
  - Beispiel Java

- Beobachtung: In echten Programmen (nicht Diplomarbeiten ✍) bestehen bis zu 80% des Codes aus Fehlerbehandlung
- Das riecht nach Automatisierung

Fehlerbehandlung prozeduralen Sprachen

```
...
someDatabaseFunction(someData);
...


void someDatabaseFunction(aType someData) {
   ...
   execSQL(myCommand,..)

}
```

Nicht gut
Bei Fehlern ist man
„blind, wie ein Maulwurf“

Fehlerbehandlung prozeduralen Sprachen: Besser ...

Das ist noch ziemlich viel „Schreibarbeit“

```
...
int RC
If ((RC = someDatabaseFunction(someData)) != NoError) {
   switch RC
   1: // reagiere;
   FATAL: // brich kontrolliert ab ...
}
```

Sowas packt man besser in Makros oder Generatoren

```
void someDatabaseFunction(aType someData) {
   ...
   execSQL(myCommand, ....)
   // frage Ergebnis ab, belege RC
   // und setze ein paar Trace Infos
}
```

Man kann kontrolliert abbrechen und sich merken, wo es Probleme gab

Fehlerbehandlung mit OO-Sprachen

```
try {
  // hier könnte etwas passieren
  dangerousCommandGoesHere ...
}

Catch (aTypeOfException) {
  // do something to fight the problem
};

Catch (aFatalTypeOfException) {
  // shut down system in a controlled way
};
```

Man kann Fehler dort abfangen,
wo man etwas damit anfangen kann

Solche behandelt man am
besten „weit oben“

Fehlerbehandlung mit OO-Sprachen

- Man hat ein Konzept in der Programmiersprache, um Fehler bequem zu behandeln
- Man kann mögliche Exceptions auch explizit im Interface definieren
- Konventionen und Disziplin sind trotzdem noch erforderlich
- Weniger „geschwätzig“ zu schreiben, als bei 3GL Sprachen
- Kritiker sagen, ExH verletzt Encapsulation, Kontrollfluß und Geheminnisprinzip
- Nur wenige beherrschen es wirklich ✍

Fehlerbehandlung in OO Sprachen
Good practices siehe ...

Klaus Renzel:

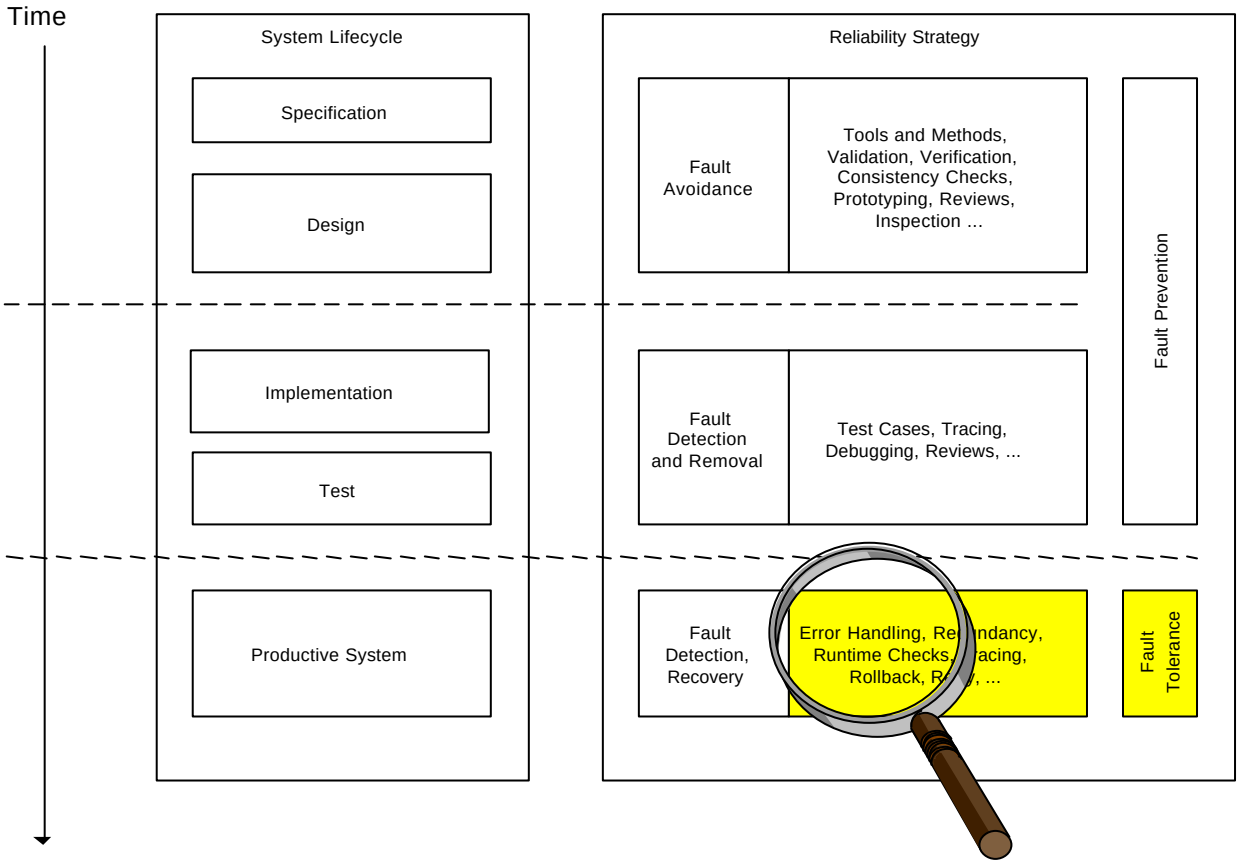
Error Handling, A Pattern Language,

http://www.objectarchitects.de/arcus/cookbook/exhandling/index.htm

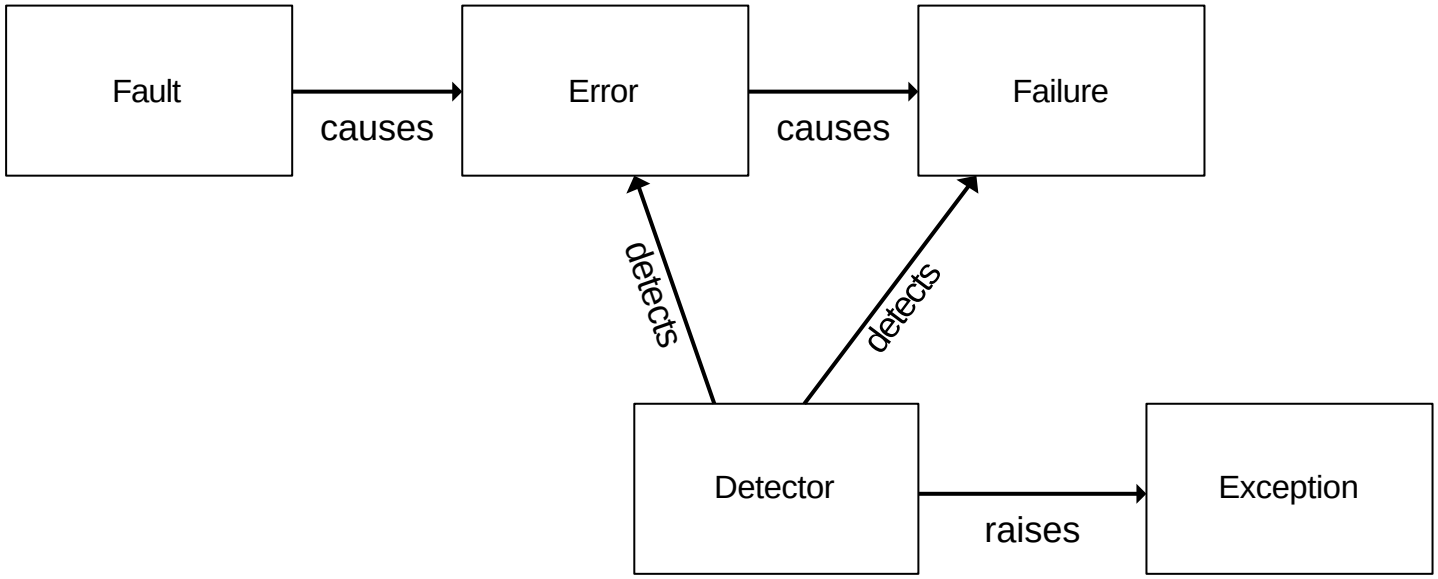
Fehlerbehandlung

- Begriffe, ...
- 10 Design Rules
- Überblick zu den ARCUS-Entwurfsmustern
  - Error Handling Framework
  - Default Error Handling
  - Error Traps

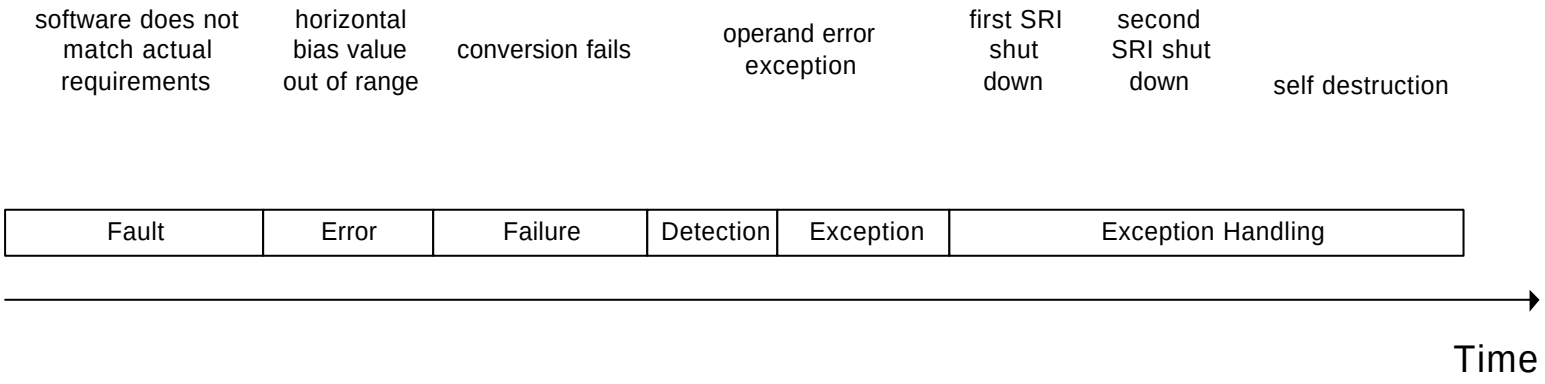
Zuverlässige Software



Begriffe


Fault
causes
Error
causes
Failure
detects
detects
Detector
raises
Exception

Beispiel: Ariane 5



Babylonische Sprachverwirrung

Programmierfehler

Schwerer Fehler

Systemfehler

Leichter Fehler

Laufzeitfehler

Anwendungsfehler

Datenfehler

Fachlicher Fehler

Benutzerfehler

Fehlerkategorisierung

- Fehler kann intern nicht behoben werden (kontrollierter Abbruch)
  => Systemfehler, “Schwerer” Fehler
- Fehler einer Systemkomponente, der intern behandelt wird
  => Fachlicher Fehler, “Leichter” Fehler, Datenfehler
- Fehlbedienung durch Benutzer
  => Benutzerfehler

Wo liegen die Hauptprobleme?

- Was zähle ich als Fehler und was nicht? (Kontextabhängigkeit)
- Wie und wann beschreibe ich das Fehlerverhalten?
- Konsistenz
- Definition und Management von Fehlertypen und -meldungen.

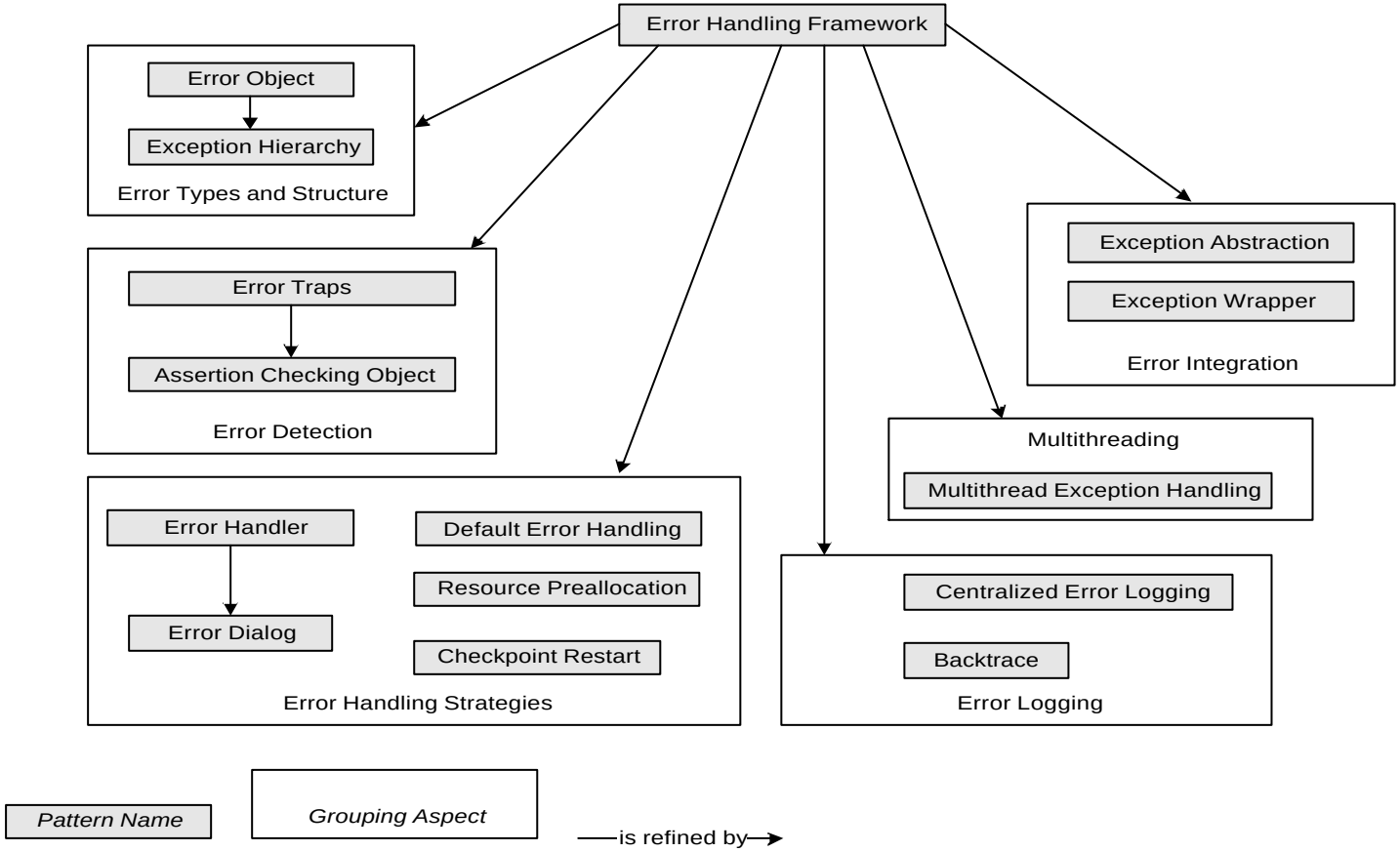
Design Rules

1. Do not throw anything around
2. Be defensive
3. Do not mix a cocktail
4. Do not break abstractions by using exceptions
5. Do not use exceptions for the normal flow of control (goto)
6. Cleanup your resources

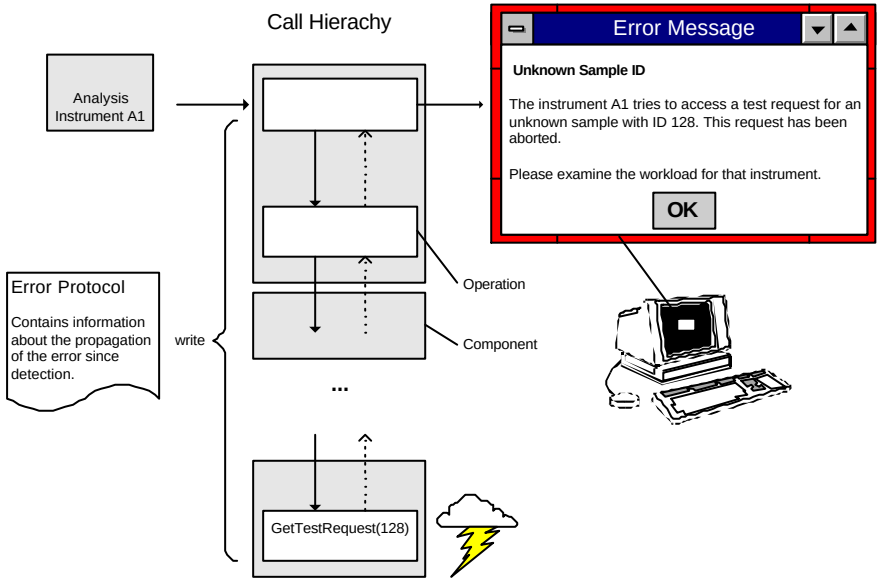
Design Rules

7. Be careful with exception that may leave
   - main functions
   - threads
   - event-handler functions in frameworks
   - destructors
8. Subtype Conformance
9. Separation of Concerns
10. Check Consistency

Die Entwurfsmuster im Überblick



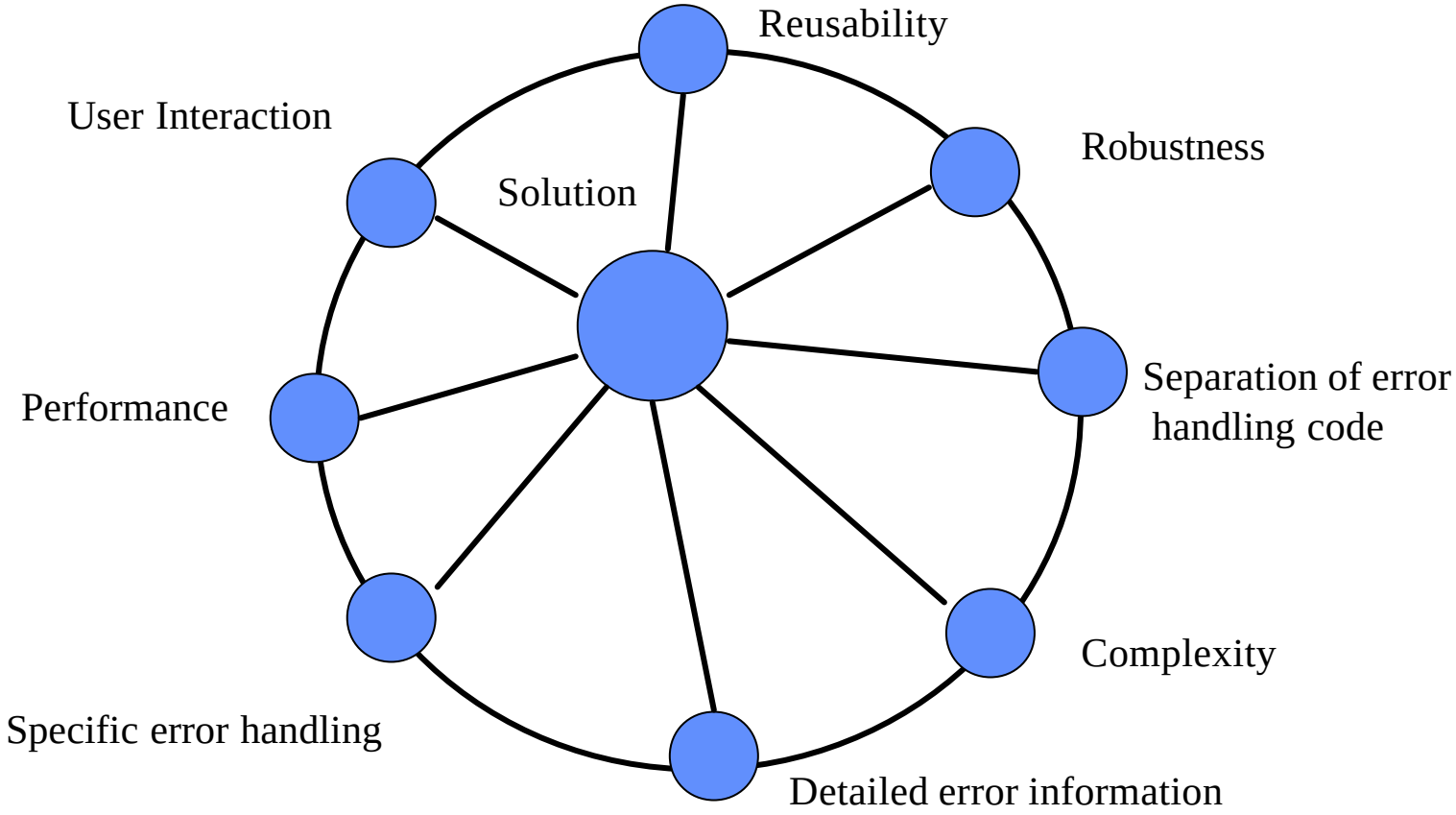
Error Handling Framework Problem



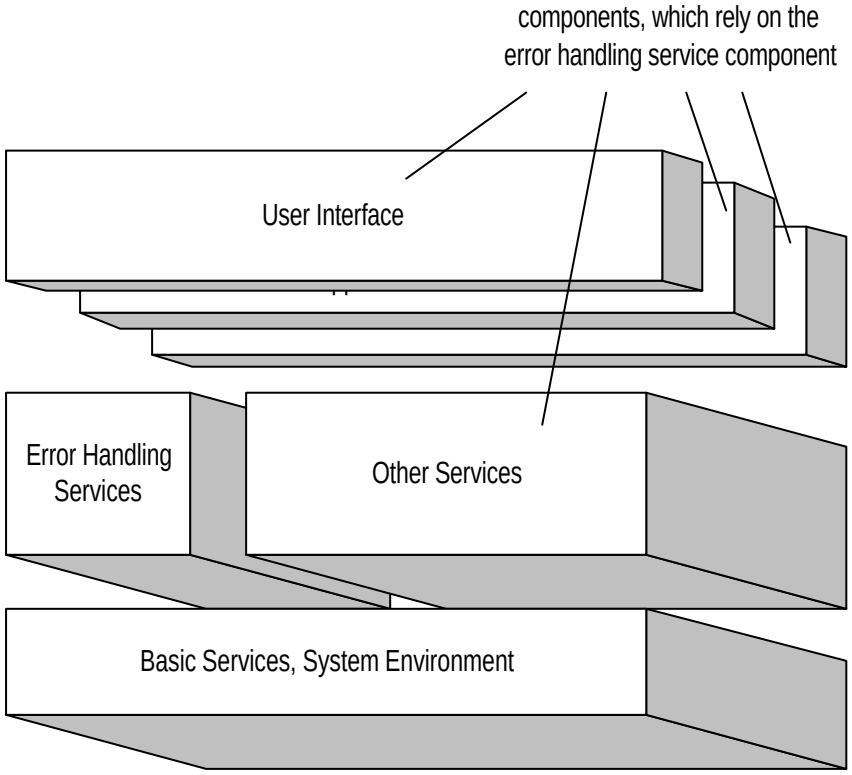
How do you design a robust and fault tolerant system?

- To keep track of error situations.
- To inform the user about errors by suitable error messages.
- To integrate error handling facilities into the system architecture.

Error Handling Framework
Forces

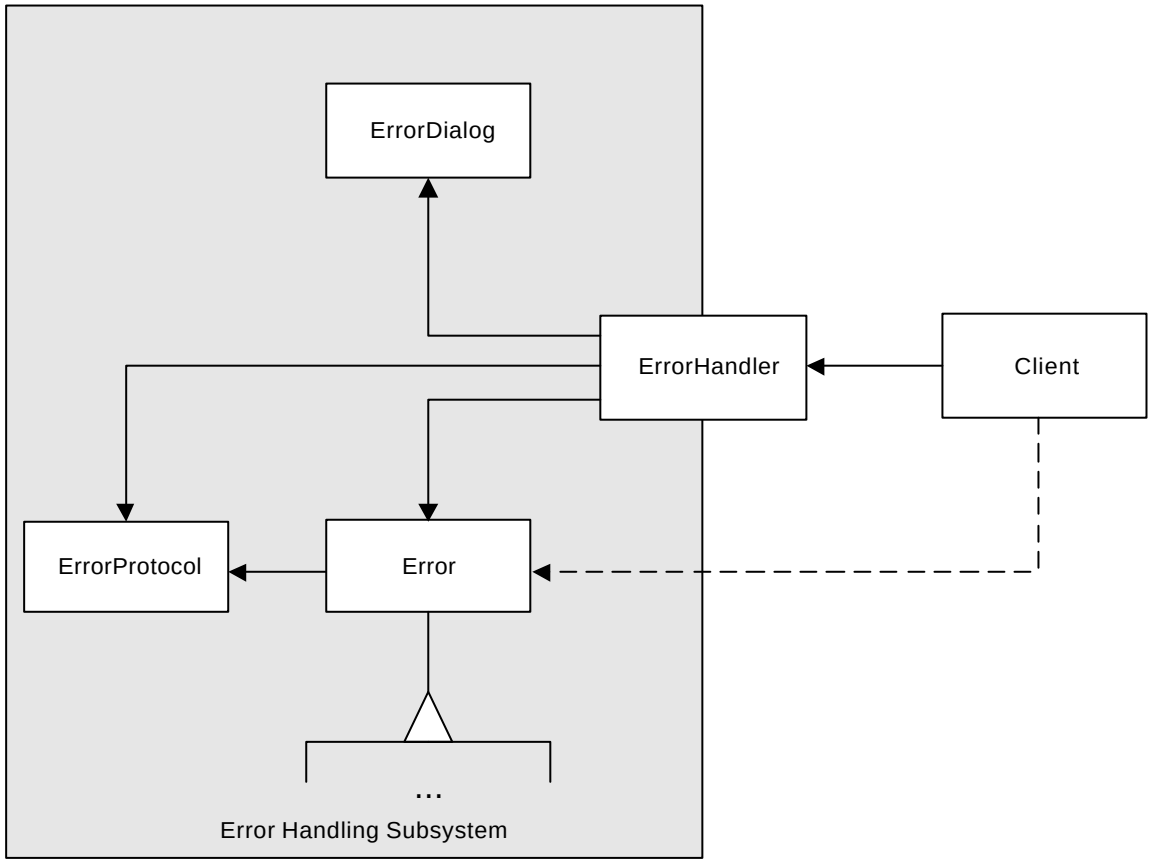


Error Handling Framework Solution



- Fehlerbehandlung wird eine “first class” Komponente der Systemarchitektur
- Feuermelder
- Feuerlöscher
- Notausgänge

Error Handling Framework Structure



Error Handling Framework Consequences

+ Controlled Panic
+ High Reliability
+ Supports encapsulation and classification of errors

– Performance penalty
– Increased Complexity (Error Handlers, Code Instrumentation)
– Integration Problems

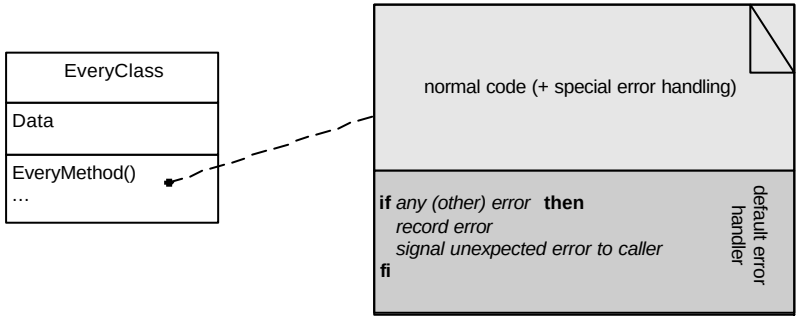
Default Error Handling
Context and Problem

- Context
  - Within every method you have to think about possible exceptions of lower level services and how to handle them. Especially in the uppermost layers of the software it is important to handle all exceptions.
- Problem
  - How do you ensure that you handle every possible exception correctly (no unhandled exception and limited damage)?

Default Error Handling Forces

- Forces
  - Relieve the developer from writing similar error handling code for every method
  - Not bother everybody with technical error handling stuff
  - Consistent error handling
  - Some error handling for unexpected exceptions is important

Default Error Handling Solution



- Provide default error hander for:
  - catching
  - logging
  - error propagation
- Unspecific
- Added to every method

Default Error Handling Implementation

- Think of the places in your code to add default error handlers.
- Consider the actions of a default error handler.
- Use macros or generators to simplify and automate default error handling.

Default Error Handling Sample Code


```
#define _EX_STD_CATCH \
        catch (ExBase & anException) { \
                ExceptionHandler::Instance()->protocolException( anException ); \
                _EX_THROW( \
                        ExBase,0, \
                        "Propagate unexpected exception derived from ExBase", \
                        anException.getExCategory() ); \
        } \
        catch (...) { \
            _EX_THROW(ExBase,0,"Unexpected Exception",ExBase::ExCat_Fatal); \
        }
```

Error Traps

- Context:
  - Before we can handle an error or failure we have to detect an error.
  - For error detection we must enrich our code with a number of run-time checks.
  - A failure can only be detected in relation to a specification of the correct behaviour.
- Problem:
  - Which indicators are useful to detect erroneous situations and where should the traps be installed in the application code?

Error Traps
Forces

- Complexity versus criticality
  - Overhead must be in relation to the severity and frequency of errors and the size of the application code.
- Performance
  - On the one hand, we want minimal performance penalties, and, on the other hand, we want to be able to detect nearly all kinds of errors as soon as possible.
- Robustness and consistency
  - It is desirable to automate error checking as much as possible because automation supports a coherent design and correct implementation.

Error Traps
Forces

- Maintainability
  - To preserve maintainability of the application code, you should avoid cluttering of the code by a mass of error detectors.
- Flexibility
  - The possibility to activate and deactivate error detectors provides more flexibility.
- Logging
  - Error detectors need access to an error log to report detection events.

Error Traps
Solution

```
METHOD AnyMethod(aType1 aParam1, aType2 aParam2, …) : aReturnType
BEGIN
      ------- error detection header ------------------------
      [ aParam1 valid ? raise exception for invalid parameter ];
      [ aParam2 valid ? raise exception for invalid parameter ];
      [ invariant holds ? raise exception for violated invariant ];
      [ precondition holds ? raise exception for violated precondition ];

      -------- normal method body ----------------------------
      …
      -- do something
      [ special test ? raise exception ];

      Result = aClass.OtherMethod(aValue);
      [ exptected Result ? raise exception ];
      …
      -------- error detection footer --------------------------
      [ invariant holds ? raise exception for violated invariant ];
      [ postcondition holds ? raise exception for violated postondition ];
      [ return value valid ? raise exception for invalid result value];
      RETURN aValue;

HANDLE
      handle exceptions raised within the block
END
```

Error Traps
Consequences

- Whether this solution detects errors as early as possible depends on the method's size.
- Not helpful to detect errors in the design specification. Specification must carefully expose the pre- and postconditions and invariants.
- Not suited to detect loops.

Error Traps
Consequences

- Because the solution enriches the code of nearly every method there are strong effects on the performance of an application.
- Problem: Incorrect implementation of an error detector itself.

Error Traps
Implementation Issues

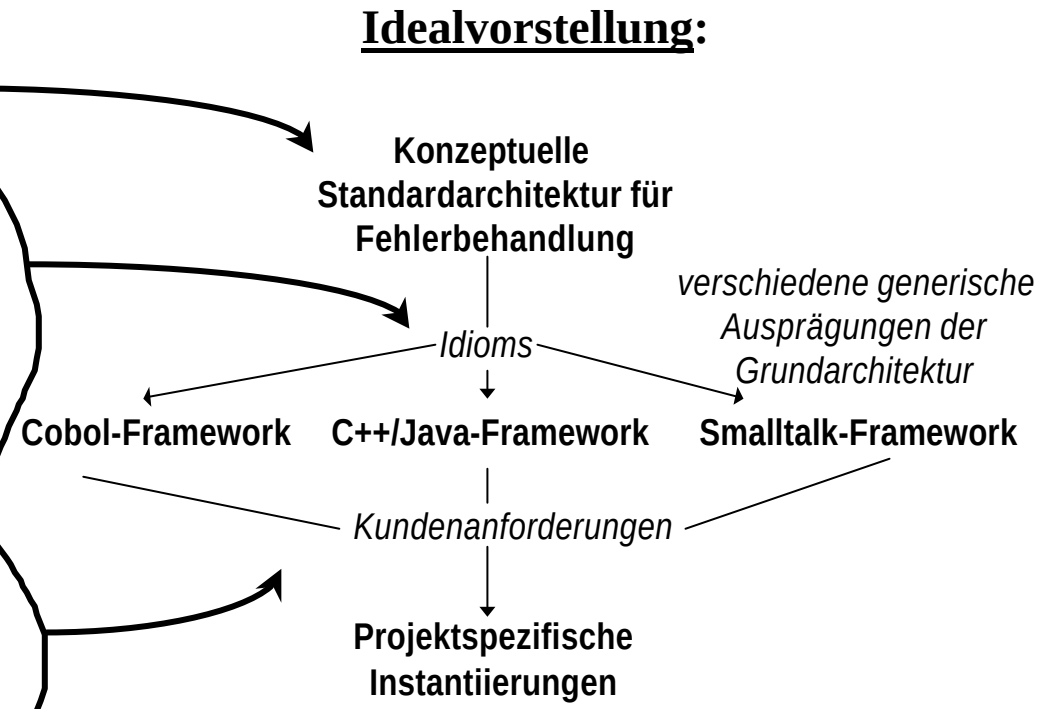
- Check methods
- Detector actions
- Activation, Deactivation
  - static vs. dynamic
- Macros
  - more readable code
  - more difficult debugging
- Generation
  - as much as possible

Error Traps
Sample Code


```
// check assertions
#define AssertTemplate(CONDITION,EXID,TEXT)                              \
 if ( !(CONDITION) )                                                        \
      throw ExAssertionFailure(#EXID,#CONDITION,TEXT,  FUNCTION  ,       \
      __LINE__,__FILE__,__DLL_NAME__,__EXE_NAME__,__EXTIME__,        \
      __USER_NAME__)

#define AssertParam(CONDITION, TEXT)                                  \
   AssertTemplate(CONDITION, EX_ILLEGAL_PARAM, TEXT)

#define AssertPrecond(CONDITION, TEXT)                                \
   AssertTemplate(CONDITION, EX VIOLATED PRECONDITION, TEXT)

#define AssertInvariant(CONDITION, TEXT)                               \
   AssertTemplate(CONDITION, EX VIOLATED INVARIANT, TEXT)

#define AssertPostCond(CONDITION, TEXT)                                \
   AssertTemplate(CONDITION, EX_VIOLATED_POSTCONDITION, TEXT)
```

Einordnung der ARCUS-Ergebnisse

- Begriffsklärung
- Sammlung von Entwurfsmustern
  - Problemdefinition
  - Lösungsskizze
  - Diskussion der Konsequenzen
  - Code-Beispiele
- Einige Design-Regeln



Fehlerbehandlung aus der Schublade?

- Nachbarsystemschnittstellen
- Richtlinien, Konzepte durch Kunden vorgegeben
- Jeder setzt eigene Standards
- Programmiersprachen ändern sich
- Verschiedene Entwicklungsumgebungen (4GL?)
- Meldungsbehandlung

Informationsquellen

- Konzepte:
  - kaum Literatur,empfehlenswert ist Bertrand Meyer
  - ARCUS, Pattern-Language
- C++: Scott Meyers, C++ Report (alte Ausgaben)
- Smalltalk: Smalltalk Report

Datenbank-Zugriffsschichten

Solchen Code will niemand schreiben ✍


```
private void insertRow (String id, String firstName, String lastName,
                        double balance) throws SQLException {

          String insertStatement =
                "insert into account values ( ? , ? , ? , ? )";
          PreparedStatement prepStmt =
                con.prepareStatement(insertStatement);
          prepStmt.setString(1, id);
          prepStmt.setString(2, firstName);
          prepStmt.setString(3, lastName);
          prepStmt.setDouble(4, balance);

          prepStmt.executeUpdate();
          prepStmt.close();
}
```

Solchen Code will niemand schreiben


```
private void deleteRow(String id) throws SQLException {
      String deleteStatement =
            "delete from account where id = ? ";
      PreparedStatement prepStmt =
            con.prepareStatement(deleteStatement);
      prepStmt.setString(1, id);
      prepStmt.executeUpdate();
      prepStmt.close();
}
```

Also läßt man ihn besser schreiben .

- CRUD
  - Wir erinnern uns .. heißt
  - Create eben gesehen
  - Read
  - Update
  - Delete eben gesehen
- Man kann solchen Code voll schematisch aus dem Objektmodell generieren

Ein paar Feinheiten sind zu beachten

- EJB‘s kennen (bis vor EJB2.0) keine Implementierungsvererbung
  - Die müsste abgebildet werden – sollte man aber eh nicht benutzen
  - Siehe diverse Patterns
- EJB‘s kennen (bis vor EJB2.0) keine Beziehungen
  - 1:1, 1:n, n:m, Aggregation
  - Die müssen von einer guten Zugriffsschicht behandelt werden
- Genauso, wie
  - Fachliche Transaktionen und Queries