

Querschnittsfunktionen Architektursicht auf Programmierung, Fehlerbehandlung, Datentypen und Organisation (QM, Test, Release-Management)

Vorlesung: Software-Engineering für große, betriebliche Informationssysteme

für Universität Leipzig, Sommersemester 2004

Institut für Software- und Systementwicklung

Professur Wirtschaftsinformatik, insbes. Informationsmanagement

Hans Hartmann (Generali VIS Informatik Ges.m.b.H., Wien)

Wolfgang Keller (AMB Generali Informatik Services GmbH, Aachen)

Motto dieses Vortrages

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

Und schon gar nicht der, den Sie erfinden!

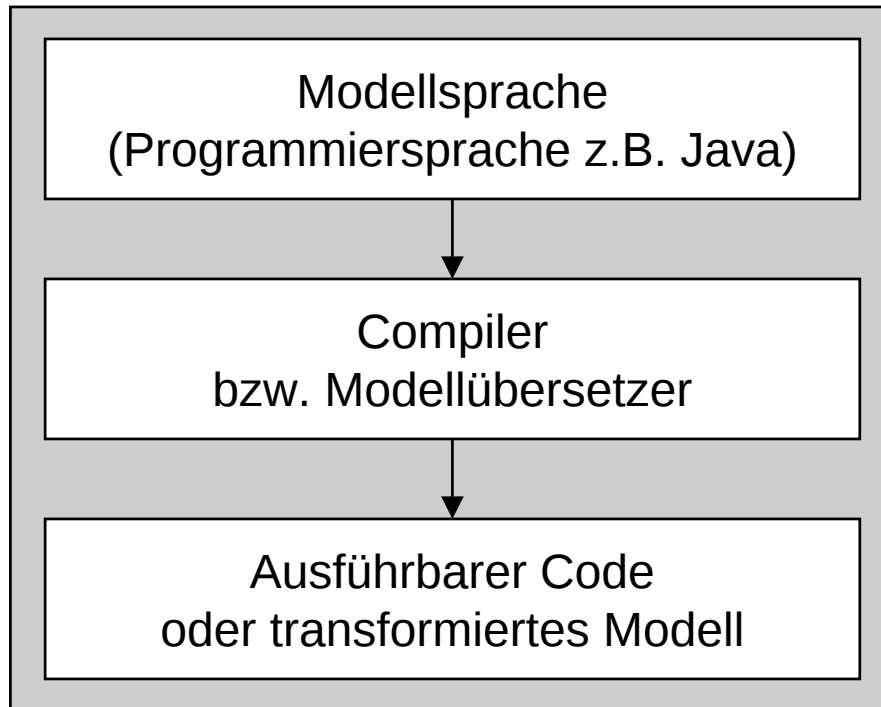
Überblick

- Idee hinter Softwareentwicklungsumgebungen
 - MDA – Model Driven Architecture
- Elemente einer SEU (Software-Entwicklungsumgebung)
- Datentypen
- Fehlerbehandlung
 - Default Error Handling
 - Error Traps
- Qualitäts-Management
- Test-Management
- Konfigurations-Management

Der Traum hinter Software-Entwicklungsumgebungen

- Abstraktionsniveau höher ziehen
 - Wie macht sich der intelligente Entwickler unentbehrlich? ☹
- Am liebsten Modellieren statt Programmieren
 - Bis einen Tag vor Auslieferung wird modelliert ☹
- Unabhängig von Plattformen arbeiten
 - Wissen wir, ob Microsoft Linux am Leben läßt ☺
- Möglichst viel Code nicht selbst schreiben
 - sondern sicher generieren lassen
 - Von ganz sicheren Generatoren
 - oder noch besser eine entsprechende Meta Maschine laufen lassen ...
 - Oder die Meta-Maschine durch eine Meta-Meta-Maschine optimieren
 - Und nur dann compilieren, wenn Interpretation zu langsam ist ...

Begriff eines Programmierstacks (Pstacks)



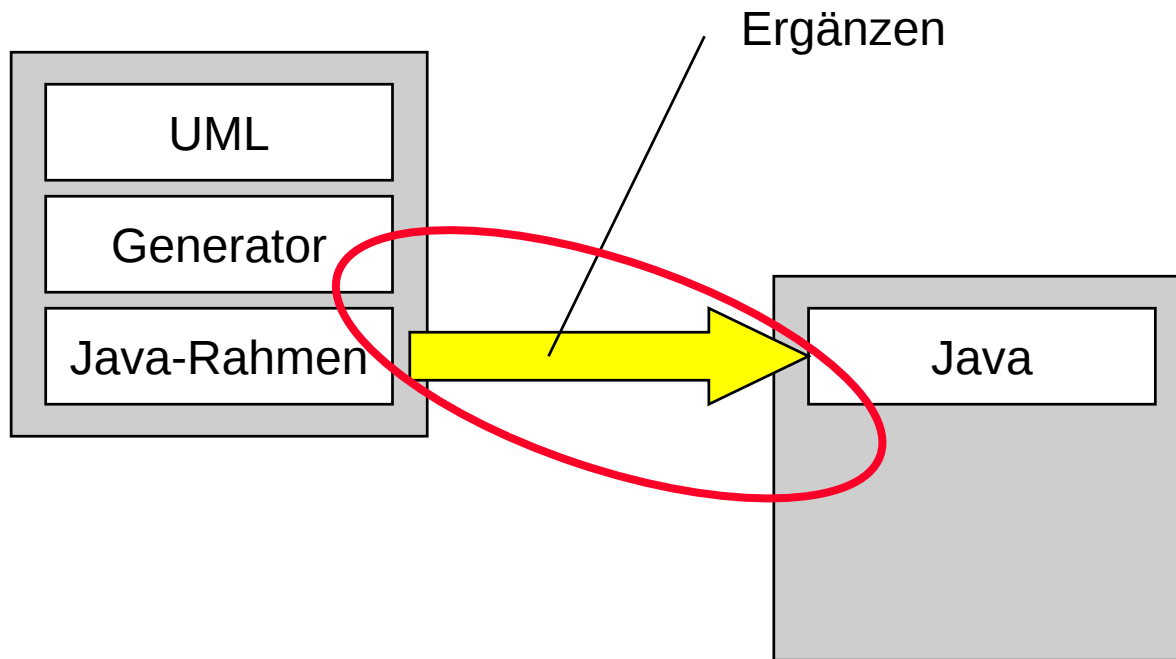
Mächtigere Ausdrucksweise
Leichter zu verstehen
Portable
Standardisiert

Maschinenabhängig
Optimiert
Konfigurierbar
Flexibel

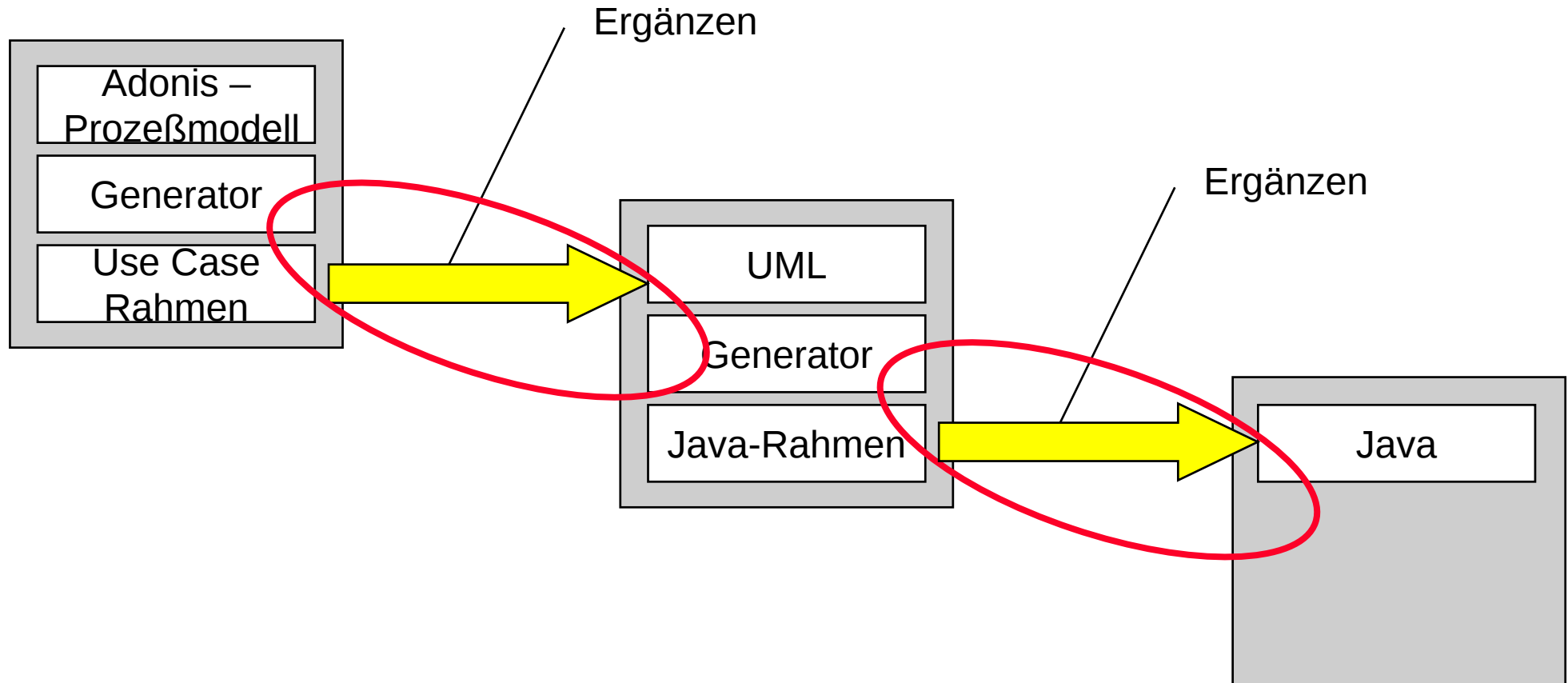
Optimiert auf eine bestimmte
Maschine

Nächstes Beispiel

2 PStacks



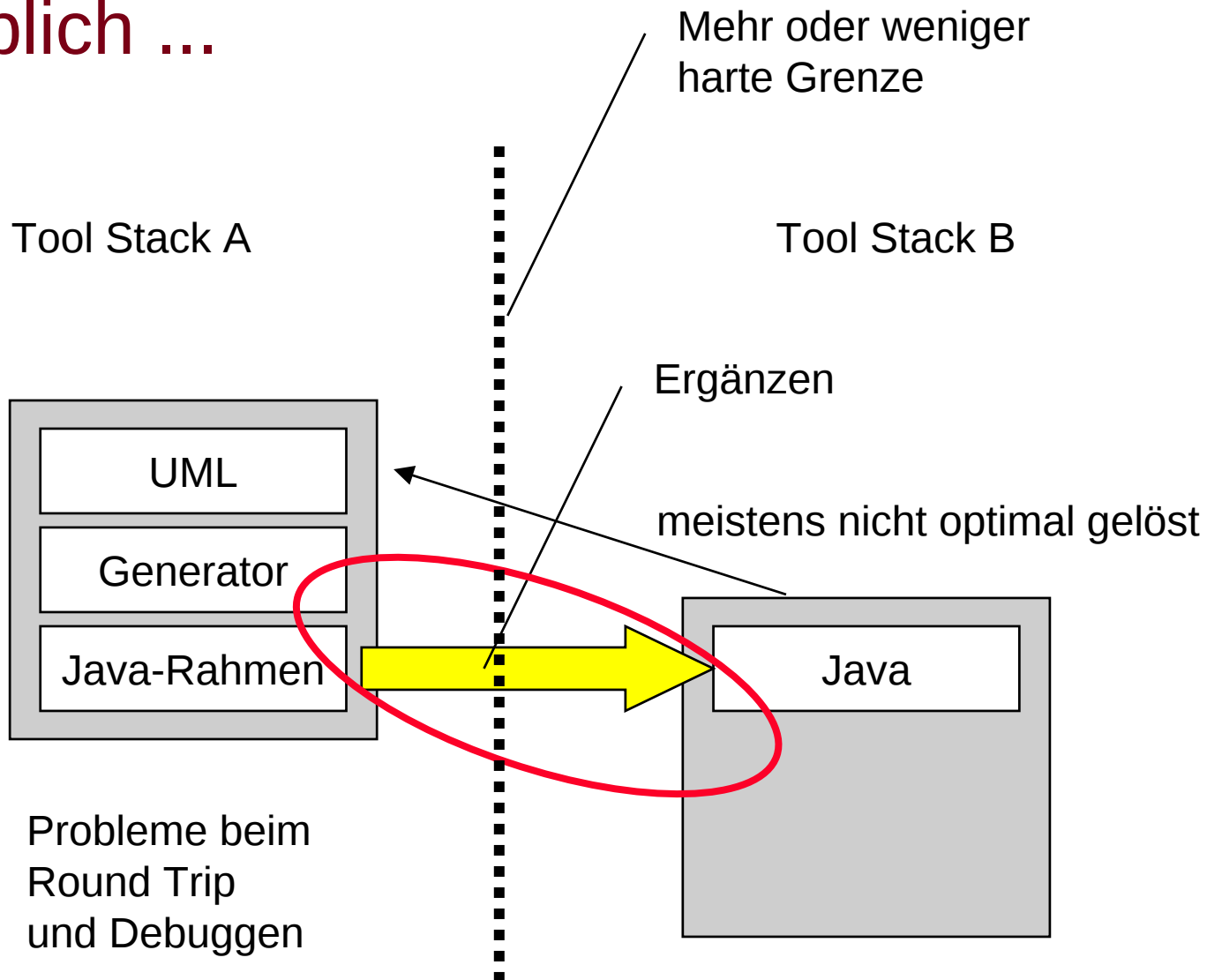
Und noch einer: der dritte PStack



Wo funktioniert so etwas gut?

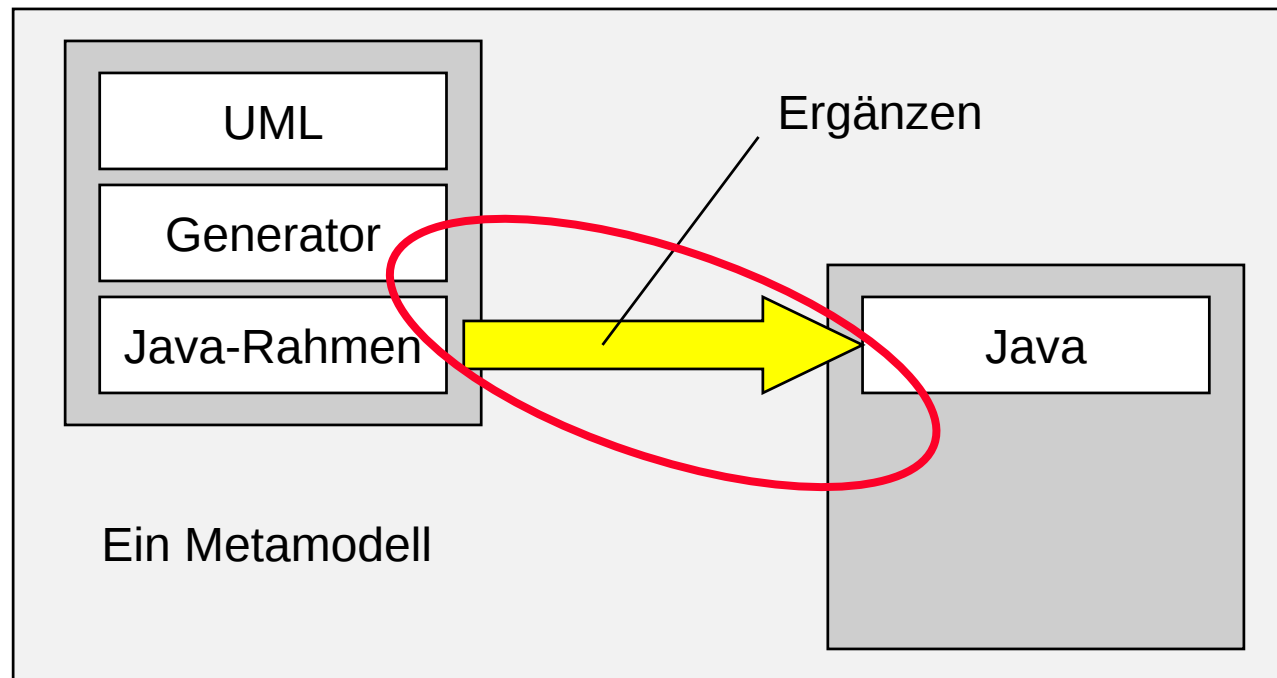
- Funktioniert gut, wenn man manuell überhaupt nichts mehr ergänzen muss
 - Compiler
 - Das war auch bei Compilern nicht immer so
 - handoptimierter Code
- Funktioniert dann nicht gut, wenn man generiert und keinen „perfekten Roundtrip“ hat,
 - also die Ergänzungen nach jeder Generierung neu machen muß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

Heute üblich ...



Besser Modellintegration

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



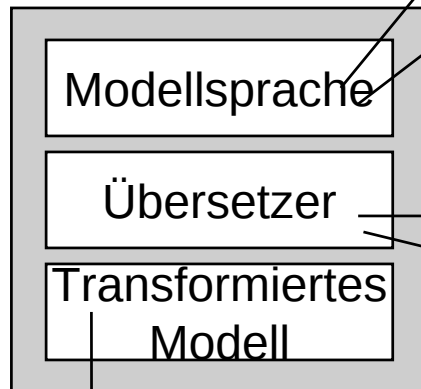
Elemente einer SEU (Software-Entwicklungsumgebung)

- Aufgabe
 - notwendige/nützliche Entwicklungswerkzeuge
 - zu einer integrierten Umgebung zusammenstellen
- Formen
 - einfach: Kommandozeile, Compiler-Programmierungsumgebung
 - komplex: Client/Server-Entwicklungsumgebung, CASE-Tool
- Entscheidungskriterien
 - Umfang und Güte interner Funktionalität
 - Offenheit für die Integration externer Werkzeuge
 - Anbindung des Konfiguration Management-Systems
 - verfügbare Anbindungen an Clients und Server

Elemente einer SEU - Werkzeuge

- Basiswerkzeuge

- Editoren
- Projektverwaltung
- Konfiguration-Management Tools
- Generatoren, GUI-Builder, Datenbank- oder EJB-Assistenten



- Komponenten und Frameworks

- Klassenbibliotheken
- Dialogelemente für Grafiken oder Daten (3D, Tabellen)
- technische/fachliche Komponenten (OR-Mapper, Finanzen)
- Compiler, Make Utilities

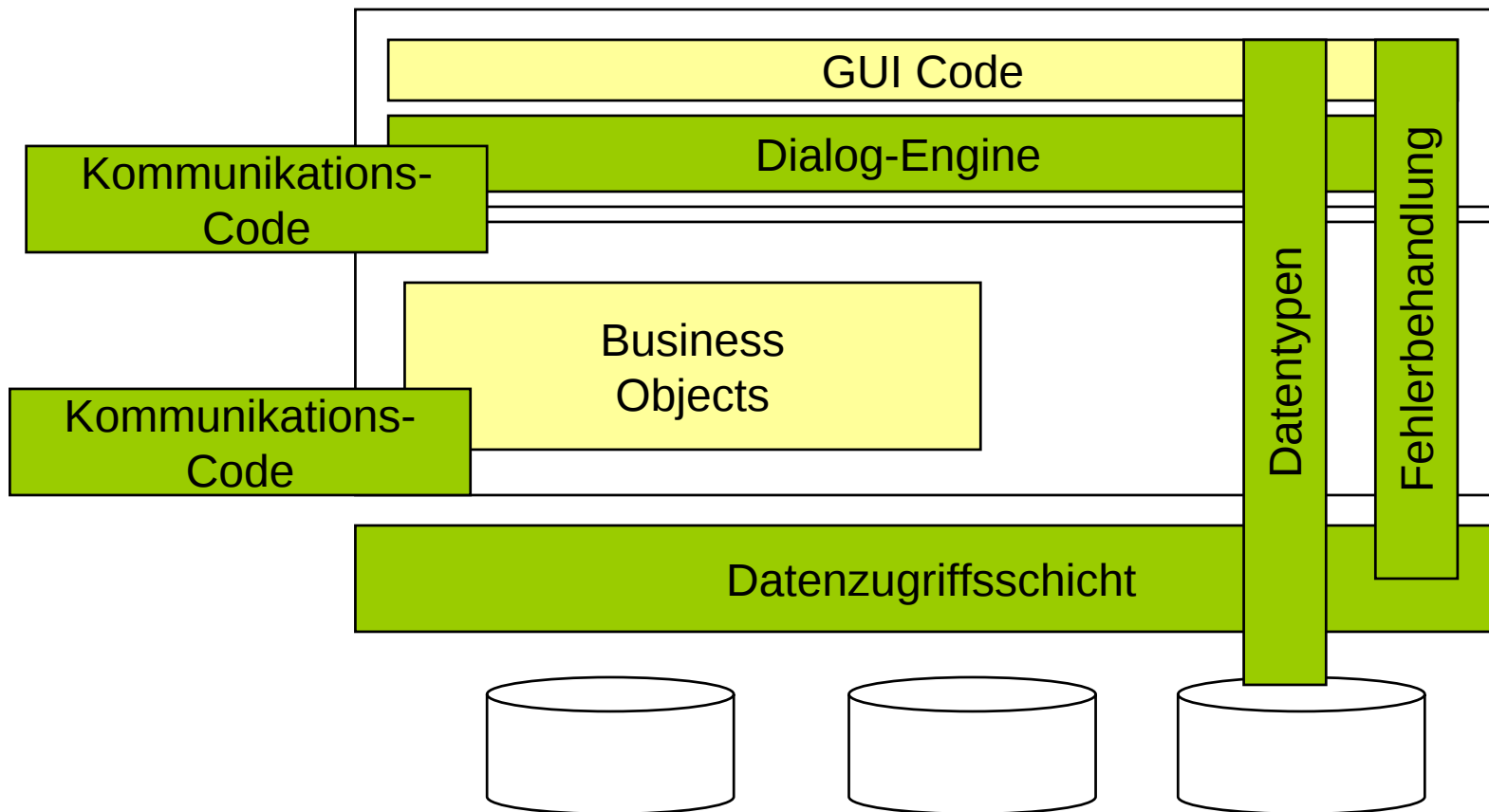
- Maschinenspezifische Tools

Linker, Debugger, Deployment-Wizards, Profiler

SEU – Erfahrungen

- unterschiedliche Entwicklungs- und Produktivplattform erfordern ein ausgefeiltes SEU- und Test-Konzept
- (Java-)IDEs gibt es wie Sand am Meer, aber für einen AppServer gibt es oft nur eine richtig passende IDE
- die Offenheit einer Umgebung kann entscheidend sein
 - externe Werkzeuge sollten möglichst gut integrierbar sein
 - umfangreiche, geschlossene Tools sind oft eher hinderlich
- IDEs führen oft zu Codegenerierung (Wizards)
 - Anpassung der Sourcen notwendig

Wo gibt es da immer noch „langweiligen fachunabhängigen“ Code



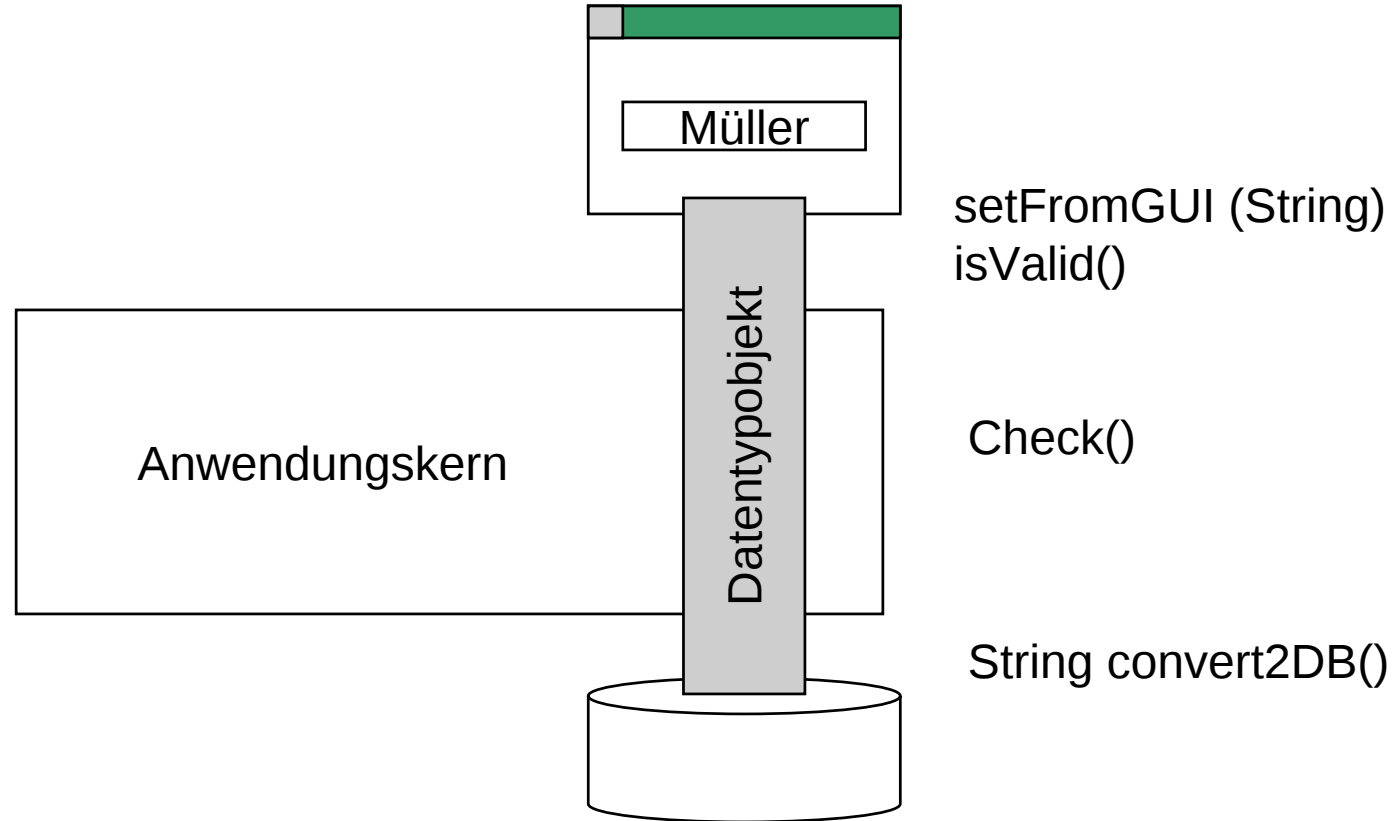
technisch
fachlich

Datentypen

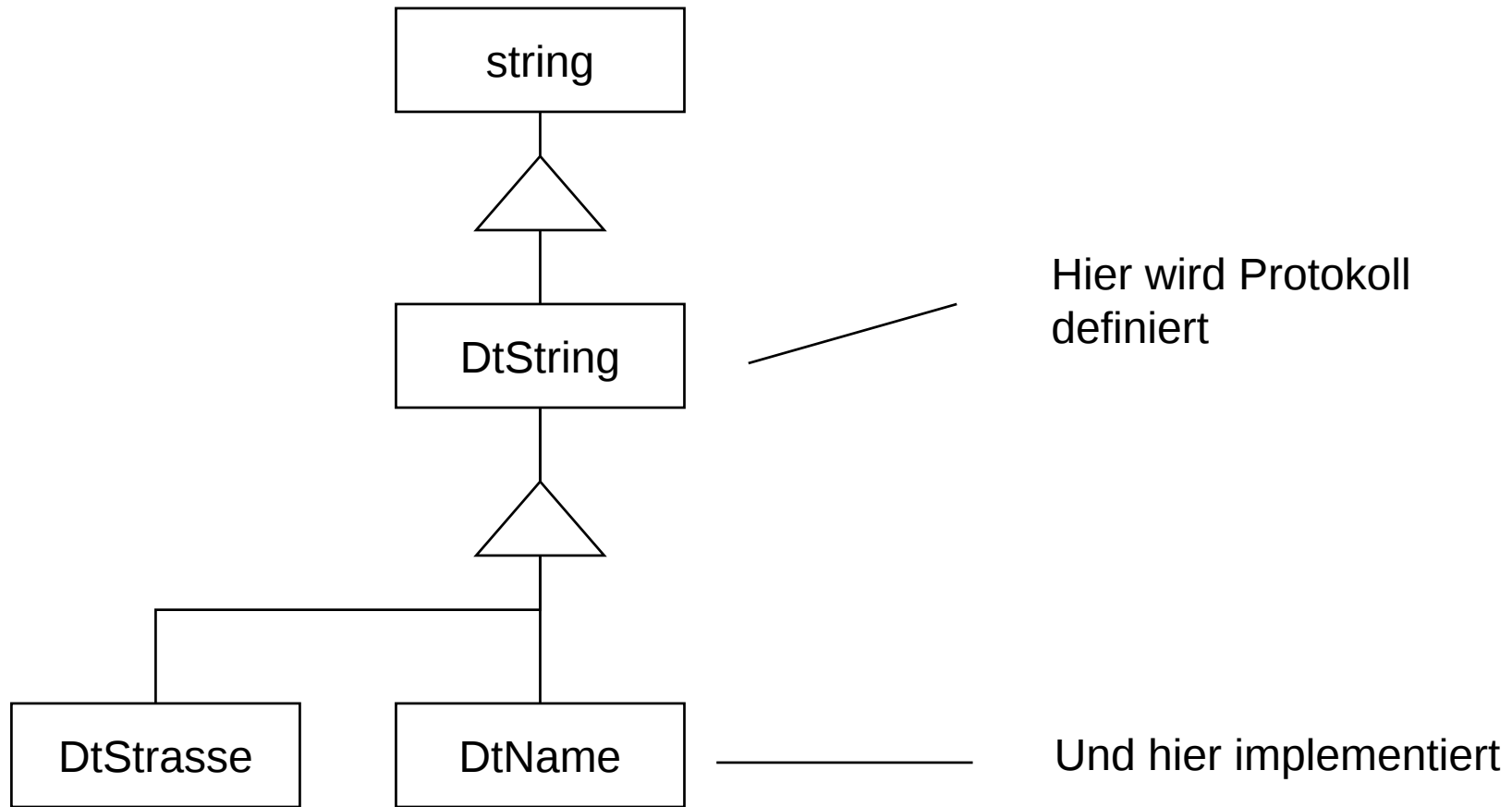
Motivation

- Basistypen der Programmiersprachen sind nicht geeignet, um fachliche Dinge auszudrücken
- int, char, string, double, decimal, ...

Datentypen – Motivation Nutzung

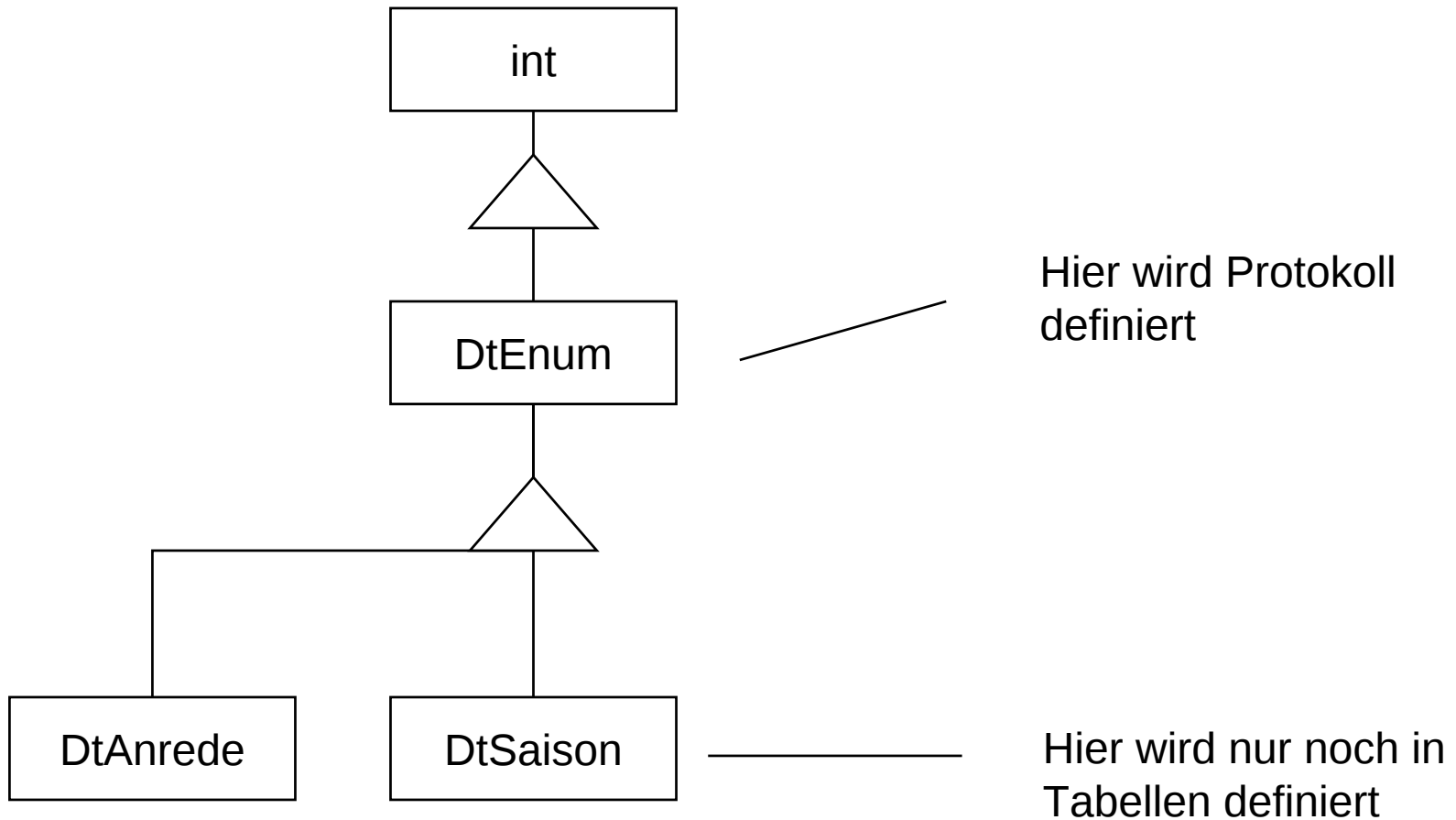


Typische Klassenhierarchien



Typische Klassenhierarchien

Weiteres Beispiel

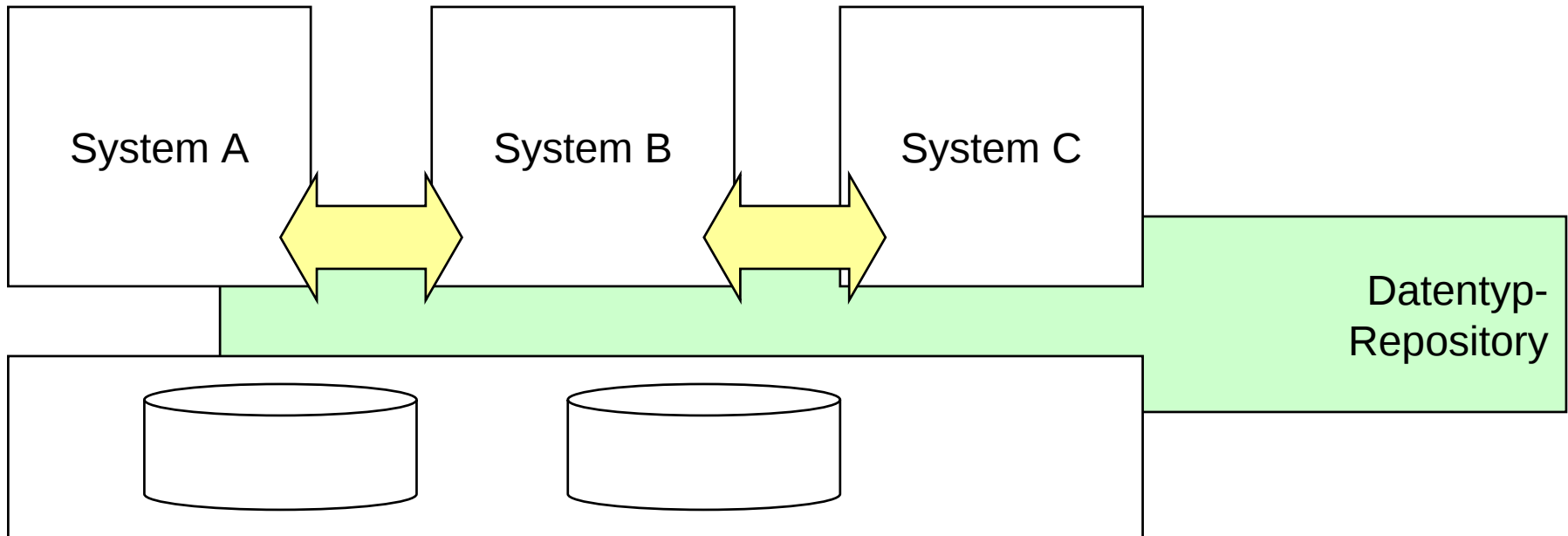


Datentypen

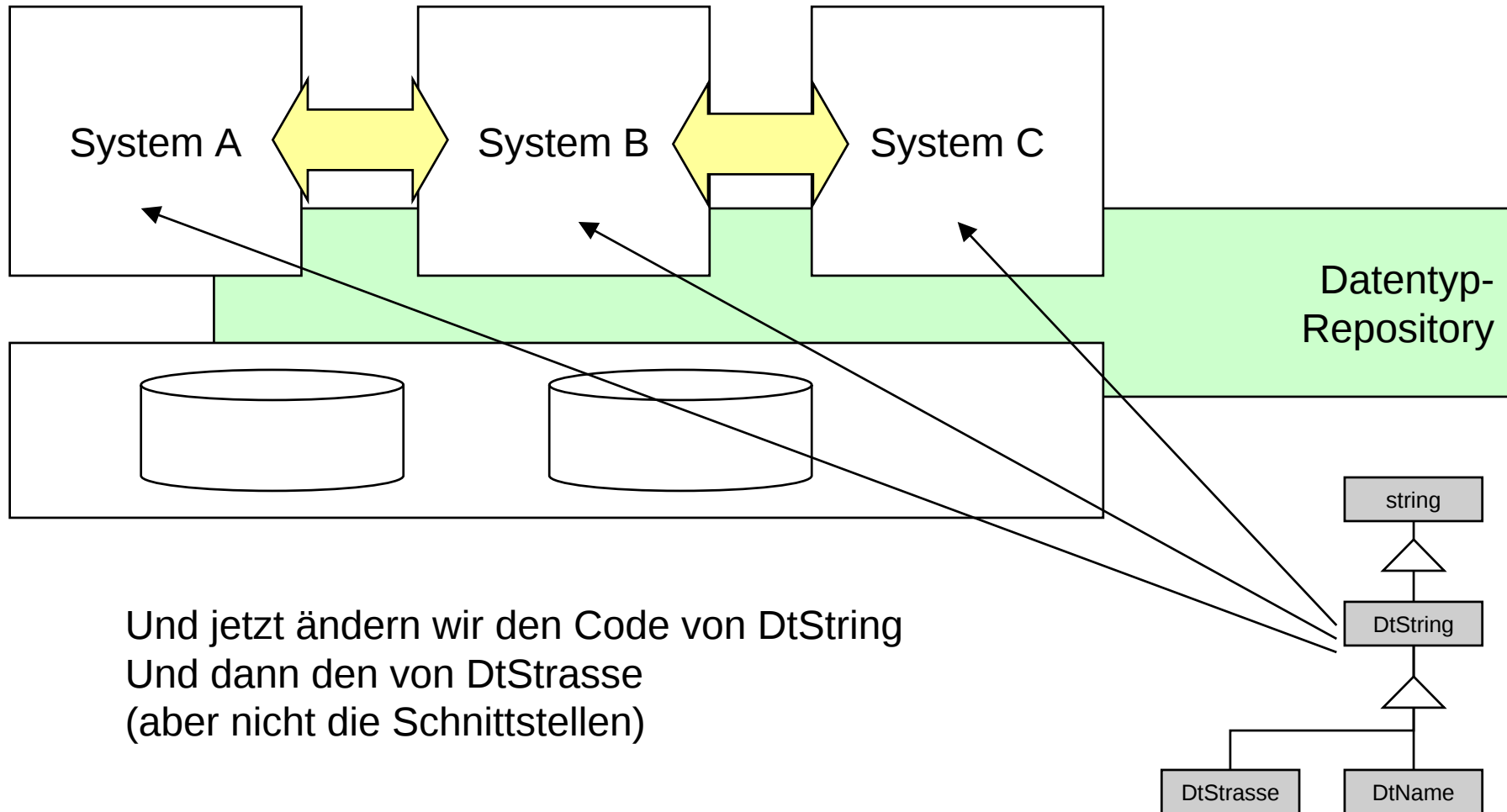
Was man meist so braucht

- Datum
- Aufzählungen, Enumerations
 - Auch dynamisch erweiterbare (z.B. Automarken, Landeskennzeichen)
- Eingeschränkte Strings
 - z.B. Vorname, Nachname
- Währungen, Geld, ...
- Numerische Typen
 - Mit speziellen Formatierungen (z.B. Polizzennummer, Schadennummern, Autokennzeichen)

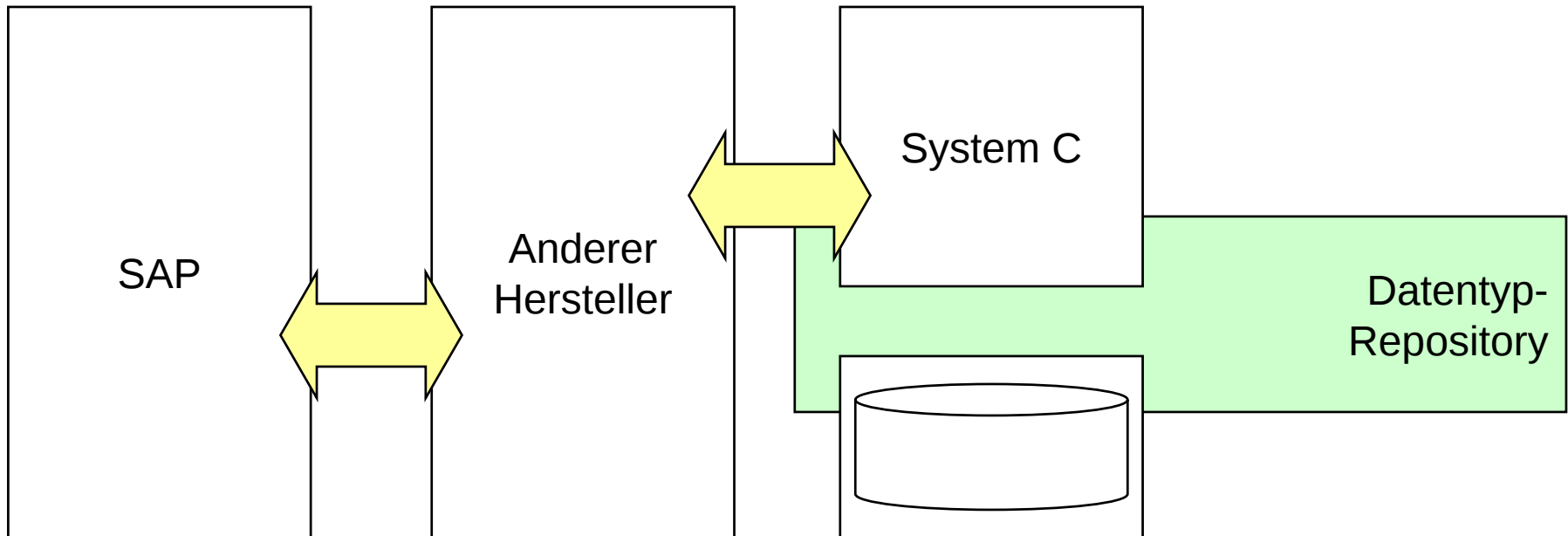
Welche Vorstellung liegt dem zugrunde?



Es ist nicht alles Gold was glänzt 😊



Die Praxis dazu sieht aber anders aus



Konvertierungen sind notwendig

Datentypen

Vor- und Nachteile

- Fachliche Typen statt technischer
- Änderungen werden überall durchgezogen
- Man kann nur schwer welche vergessen
- IDE erstellt Verwendungsnachweis
- Vereinfacht Wartung (im Normalfall über ein System schon)
- 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

Aber jetzt mal im Ernst: die Typlosigkeit von Smalltalk ist eine NIL Schwäche!

Fehlerbehandlung

Fehlerbehandlung

- Bei prozeduralen Sprachen
- Bei OO-Sprachen
 - Beispiel Java
- Beobachtung: In Programmen bestehen bis zu 80% des Codes aus Fehlerbehandlung
- Das riecht nach Automatisierung
- Früher war das nicht so. Machen wir heute mehr Fehler?

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
damit es überall gleich ist

```
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  
};
```

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

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

Solche behandelt man am
besten „weit oben“

Fehlerbehandlung mit OO-Sprachen

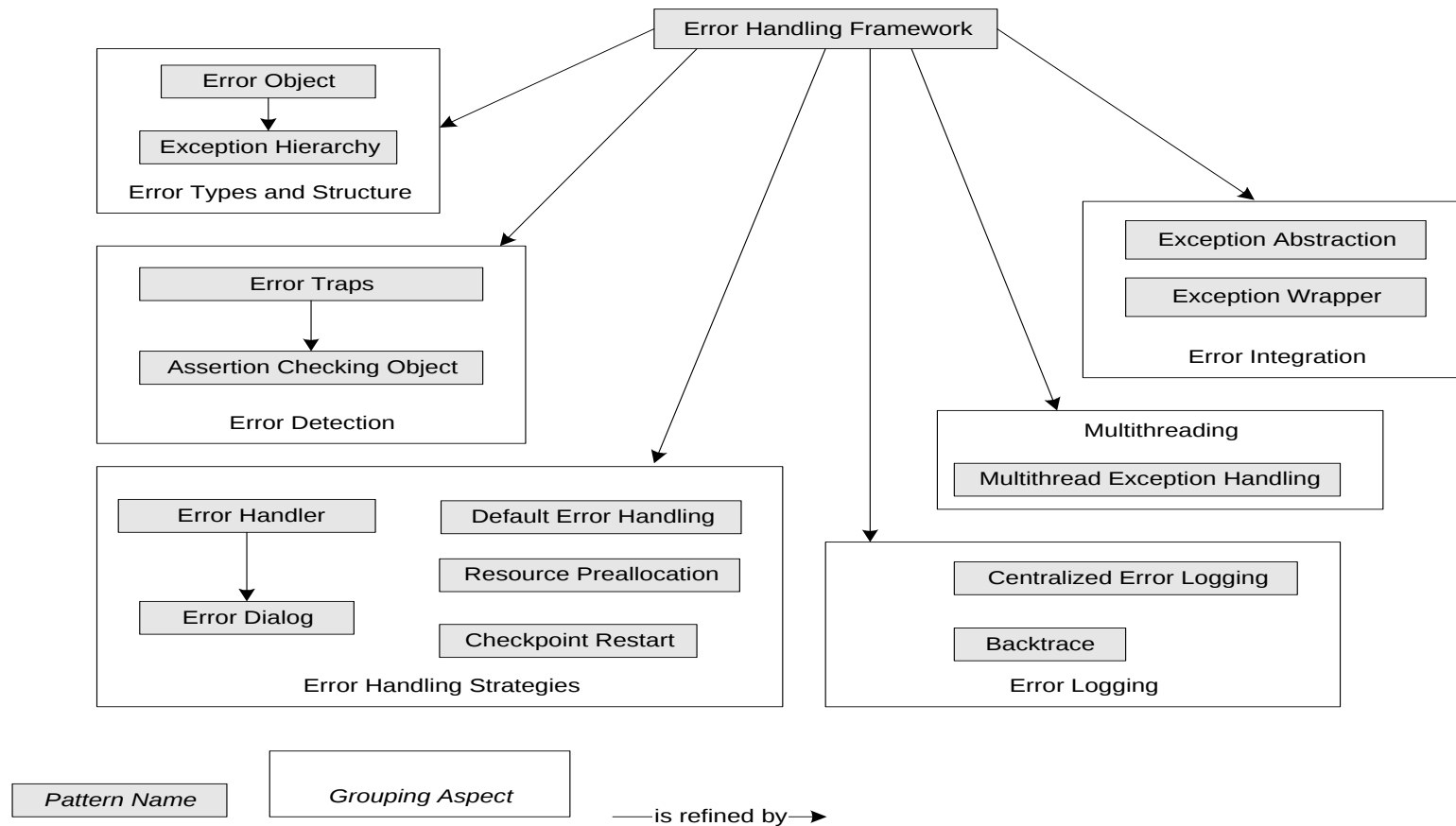
- Man hat ein Konzept in der Programmiersprache, um Fehler bequem zu behandeln
- Man kann mögliche Exceptions auch explizit im Interface definieren
- Konventionen und Disziplin sind trotzdem noch erforderlich
- Weniger „geschwätzig“ zu schreiben, als bei 3GL Sprachen
- Kritiker sagen, Exception Handling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

Fehlerbehandlung in OO Sprachen

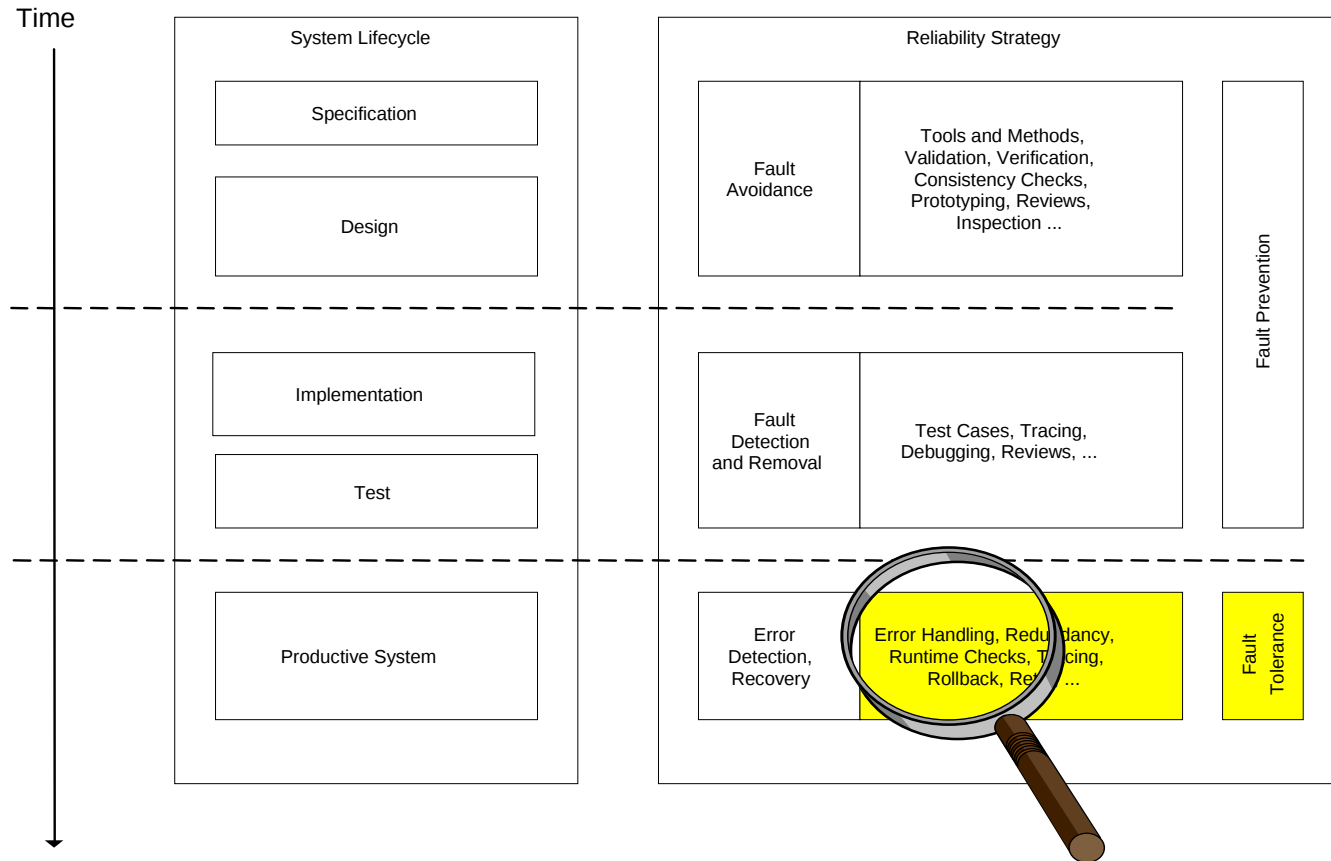
Good practices siehe ...

Klaus Renzel:
Error Handling, A Pattern Language,
<http://www.objectarchitects.de/arcus/cookbook/exhandling/index.htm>

Roadmap - Die Entwurfsmuster im Überblick



Zuverlässige Software



Babylonische Sprachverwirrung



Fehlerkategorisierung

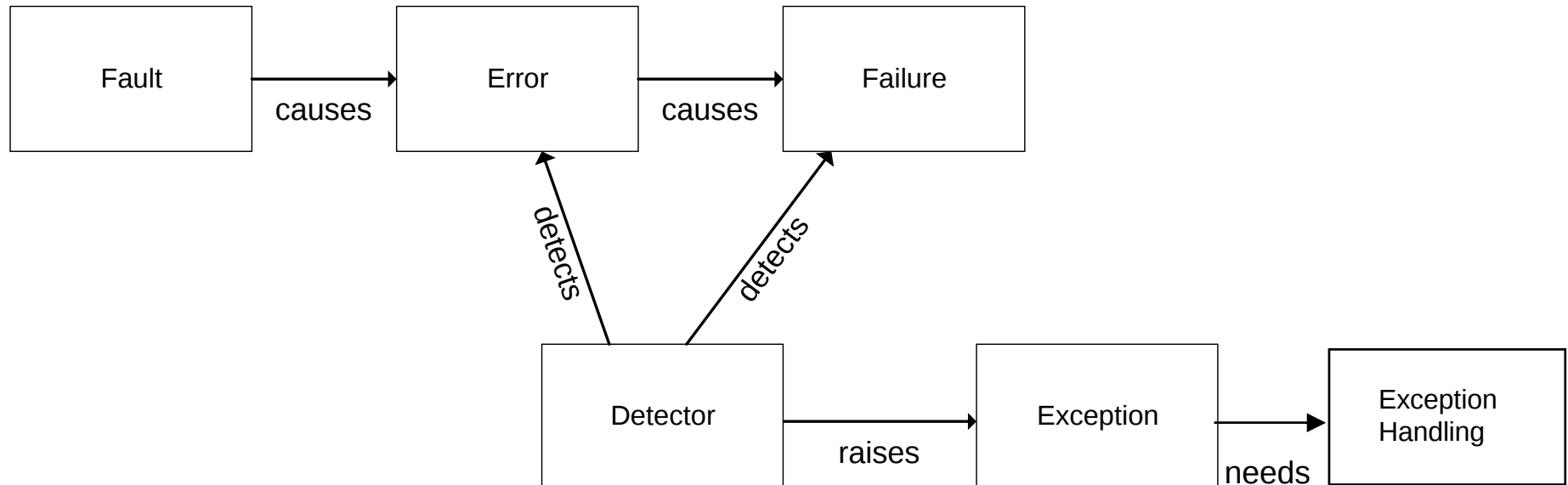
- Fehler kann intern nicht behoben werden (kontrollierter Abbruch)
=> Systemfehler, “Schwerer” Fehler
- Fehler einer Systemkomponente, der intern behandelt wird
=> Fachlicher Fehler, “Leichter” Fehler, Datenfehler
- Fehlbedienung durch Benutzer
=> Benutzerfehler

Begriffe

Software does not
match actual
requirements

e.g. out of
range error

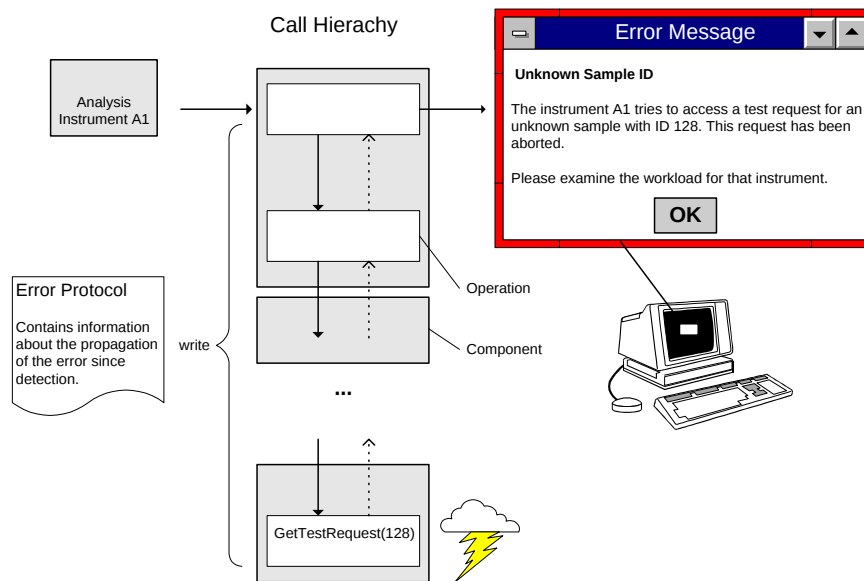
e.g. conversion
fails



Wo liegen die Hauptprobleme?

- Was zähle ich als Fehler und was nicht? (Kontextabhängigkeit)
- Wie und wann beschreibe ich das Fehlerverhalten?
- Definition und Management von Fehlertypen und Fehlermeldungen.
- Problem: Bug oder Feature
 - Wann wird ein Bug zum Change-Request?

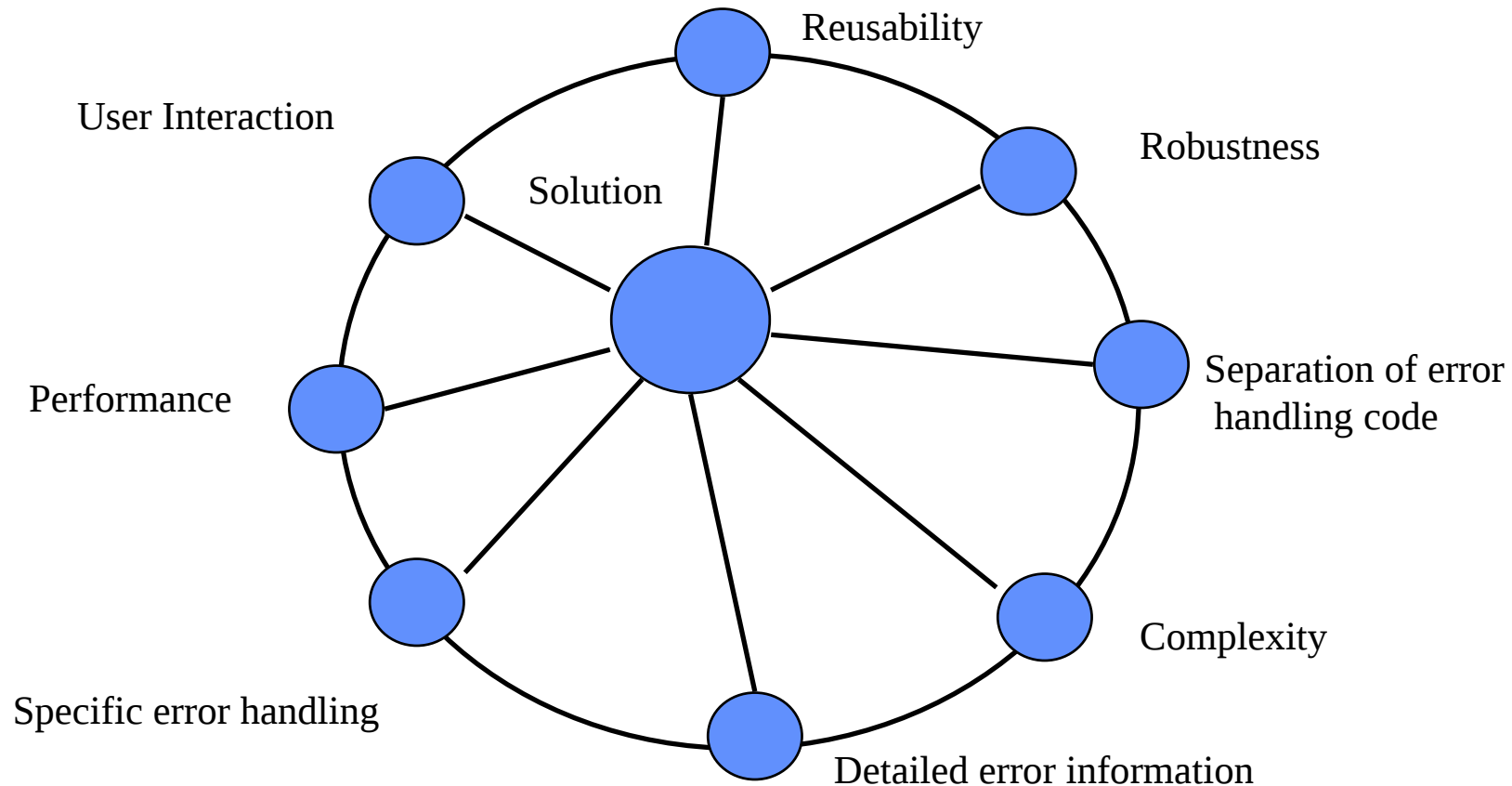
Error Handling Framework Problem



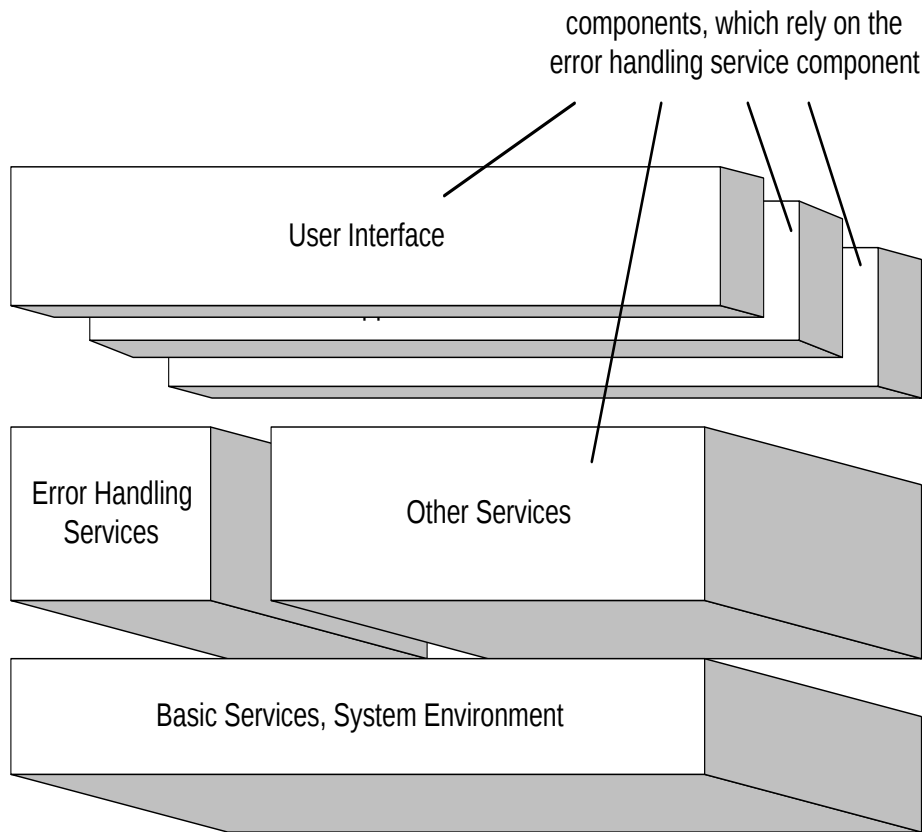
How do you design a robust and fault tolerant system?

- Keeping track of error situations.
- Informing the user about errors by suitable error messages.
- Integrating error handling facilities into the system architecture.

Error Handling Framework Forces



Error Handling Framework Solution



- Fehlerbehandlung wird eine Service-Komponente (“first class” Komponente) der Systemarchitektur
- Error-Handling Framework
- Feuermelder
- Feuerlöscher
- Notausgänge

Default Error Handling

Context and Problem

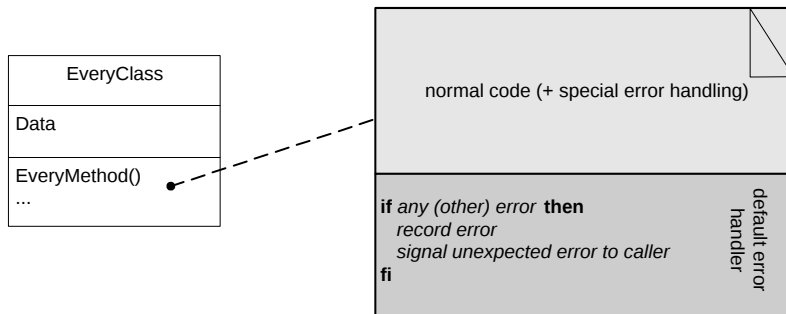
- Context
 - Within every method you have to think about possible exceptions of lower level services and how to handle them. Especially in the uppermost layers of the software it is important to handle all exceptions.
- Problem
 - How do you ensure that you handle every possible exception correctly (no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

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 fault 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

Code Example

```
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);
    [ expected 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
 - Unless you are a real good programmer...
 - Safety catches
 - Watchdogs
 - Memory Observers
 - Timeouts – coming often directly from the system software

Error Traps

Consequences

- Thesis: because the solution enriches the code of nearly every method there are strong effects on the performance of an application.
 - Speculation: NO! unless we run into problem below
 - The try/catch clause results in an “if” statement. If we deal with data accesses, we could implant 1000 try/catch clauses without slowing down the overall system
- Problem: Incorrect implementation of an error detector itself.

Quality Management

Quality Management

- So wie Eurocard: entweder Sie haben eines oder Sie brauchen eines, schlimmstenfalls macht der Entwickler sein eigenes QM
- Wirkt während der Entwicklungszeit
 - Verringert Design-Fehler (über Anforderungen-Review)
 - Verringert Entwicklerfehler durch Stilprüfung und statische Analyse
 - Setzt Normen für den Herstellungsprozess von Software (fälschlicherweise oft als Entwicklung bezeichnet)

Test Management

Test Management

- Error-Handling zur Runtime muss geschützt werden.
 - Bekannte und denkbare Fehler werden durch Code-Unit-Tests (Entwickler) und funktionale Tests (Tester) gefunden und in der Menge reduziert.
 - Durch Analyse der Error-Häufungen kann erhöhter Exception-Handlingbedarf festgestellt werden.
- Organisatorisch ist Test eine Querschnittsfunktion, die nicht nur an einem Projekt angesiedelt sein darf sondern global an allem, was in der Firma entwickelt wird.

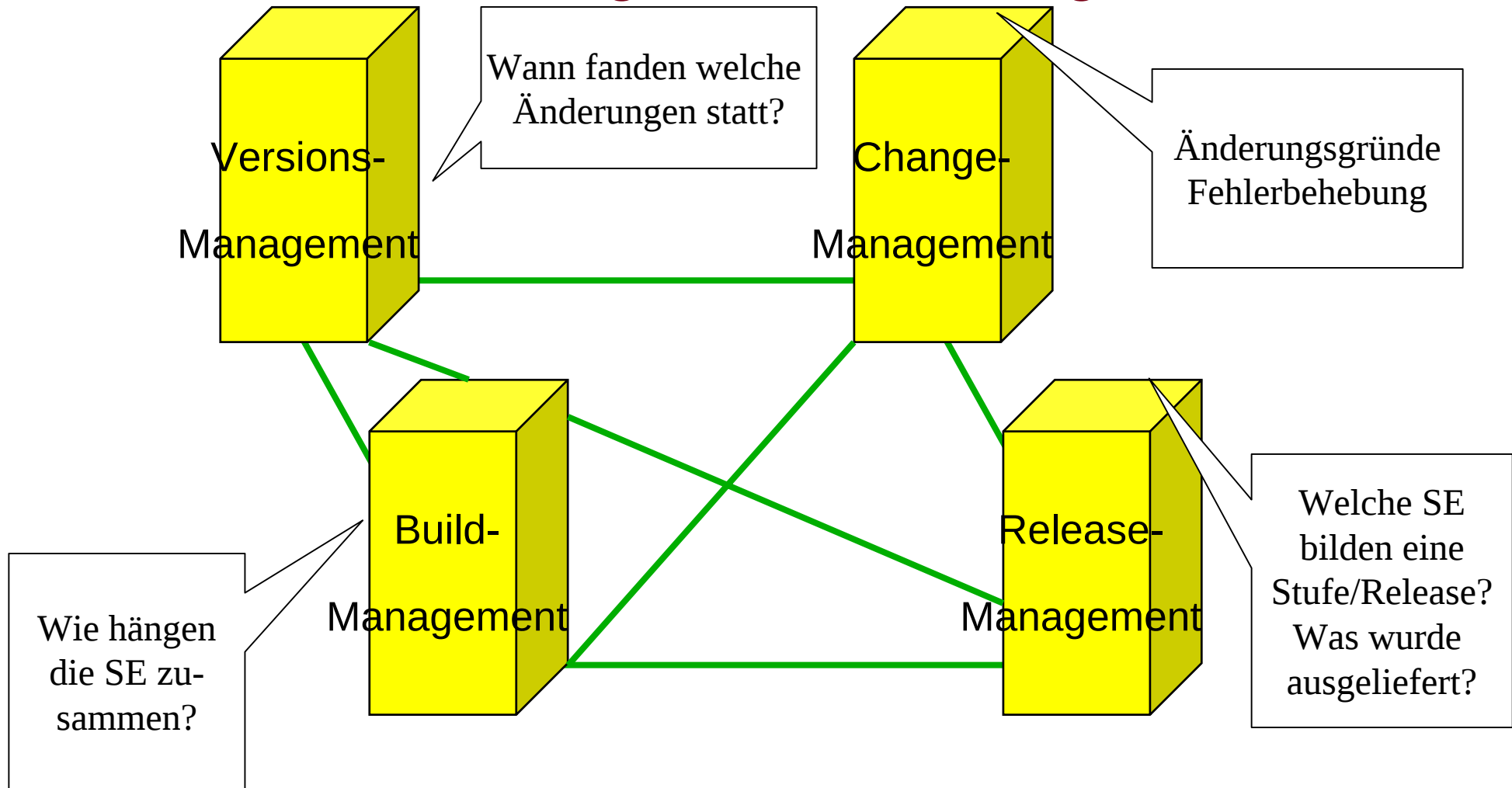
Konfiguration Management

Warum ist Konfiguration-Management eigentlich notwendig?

Software ist derzeit eine Technik mit sehr großen Notwendigkeit für Konfigurations Management

- sehr hohe Komplexität:
mehrere 1000 zusammenspielende Softwareelemente (Klassen, Module) in einem Programm; Dutzende zusammenspielende Programme (Betriebssystem, Datenbank, Compiler, Bibliotheken, ...) in einer Anwendung
- total immateriell:
flüchtige Speicherinhalte; viele Transformationen vom Quelltext zum Maschinencode

Die Säulen des Konfigurations-Management



Repositories – die zentrale Ablage

- Idee

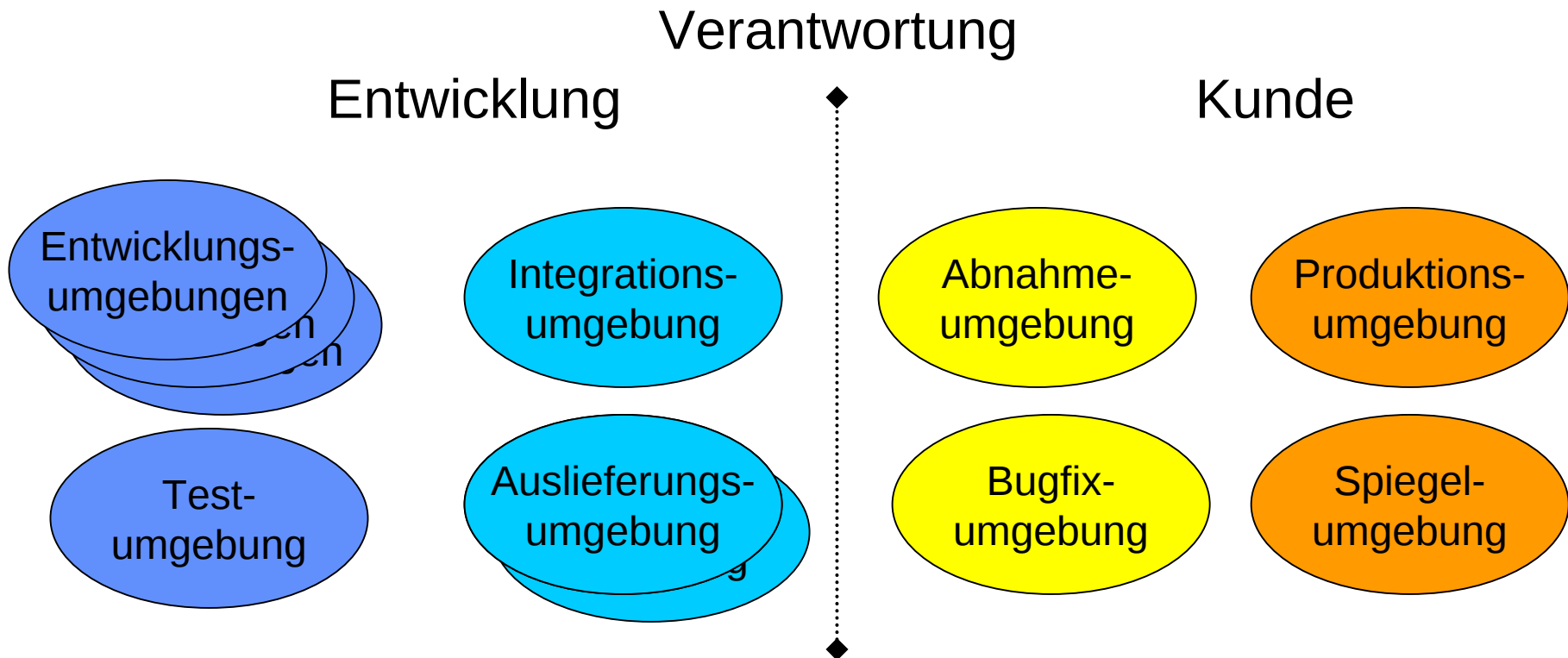
- Ordnung halten durch eine Datenbank für Projektergebnisse
- Ermöglichen einer Versionsverwaltung
- Projektunabhängigkeit durch Metamodell mit Objekttypen und Beziehungstypen zwischen Objekten

- Aufgaben

- Dokumente speichern, verwalten, vernetzen
- Vollständigkeit und Konsistenz überprüfen
- Abhängigkeiten und Betroffenheiten analysieren
- Quelldaten für Weiterverarbeitung liefern



Raumkonzept einer Anwendung



Die hellen Umgebungen sind in kleinen Projekten nicht unbedingt erforderlich, in großen Projekten jedoch schon.

Konfiguration-Management

- Entscheidungskriterien für die Toolauswahl
 - Teamarbeit (Aufgabenverteilung/Rollen, Sperren, ...)
 - Komplexität des Modells (ausreichend vs. intuitiv bedienbar)
 - integriertes Change Request Management
 - integriertes Build / Release Management
 - verfügbare Anbindungen an IDEs

Konfiguration-Management – Erfahrungen

- Alle Objekte in einem einzigen zentralen KM-Tool verwalten
- Nutzungskonzept gut auf Projekt abstimmen: realistisch bleiben, zu hohe Anforderungen (z.B. Versionierung des Betriebssystems) vermeiden
- Konfiguration Management organisatorisch einbinden
- Tools: Mit (und nicht gegen) das Tool arbeiten, Tool verstehen
- Auf gute Performance achten
- Zeitaufwand für Einführung von KM nicht unterschätzen
- Konfiguration Management lohnt sich schon bei einem Mitarbeiter (Versionierung, Baselines)