Software-Qualität

Best Practices für Portal-Projekte mit .NET

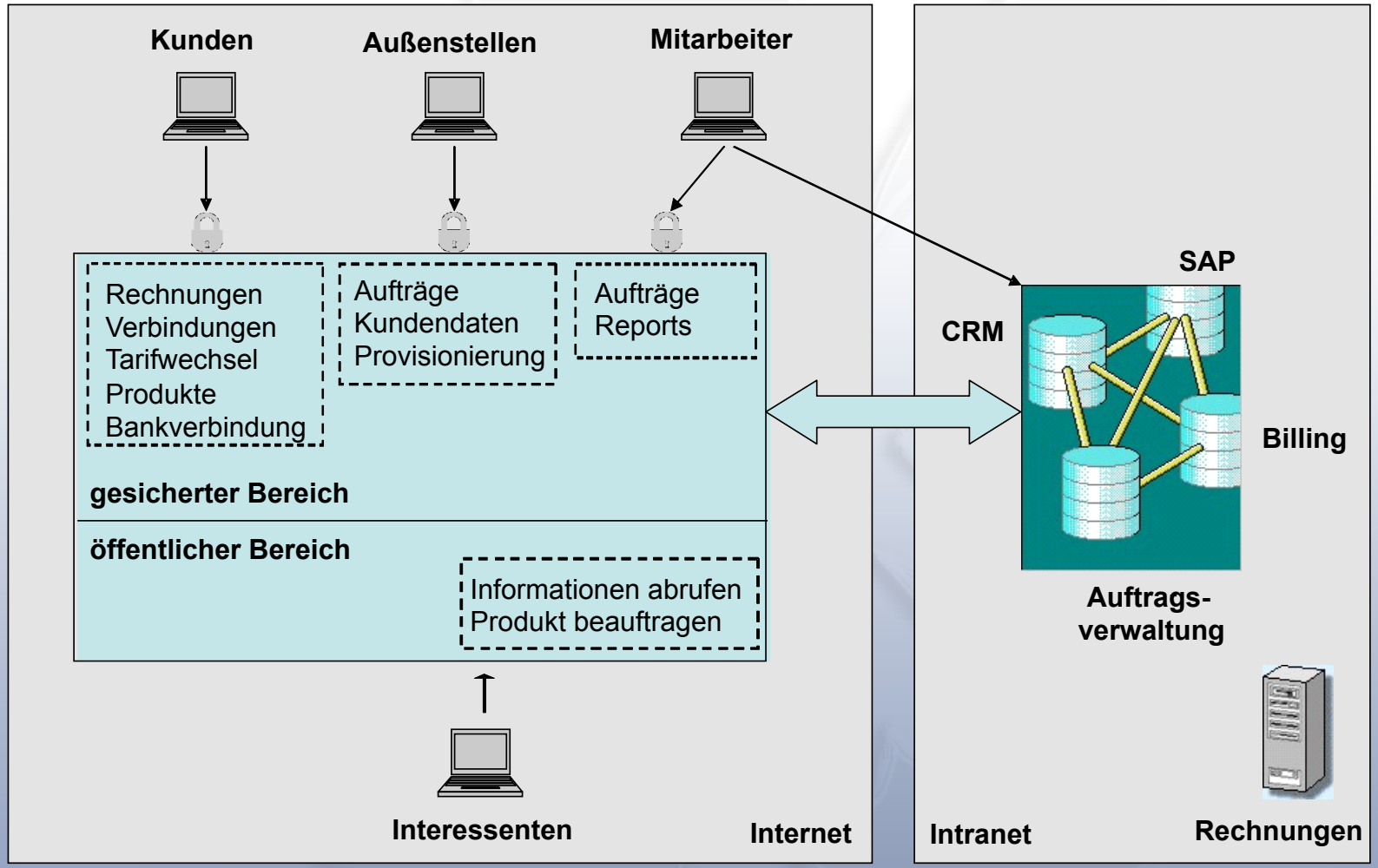
Manuel Lengert
mlengert@icommit.de
iCommit Integrationslösungen GmbH

BASTA! VISUAL STUDIO .NET

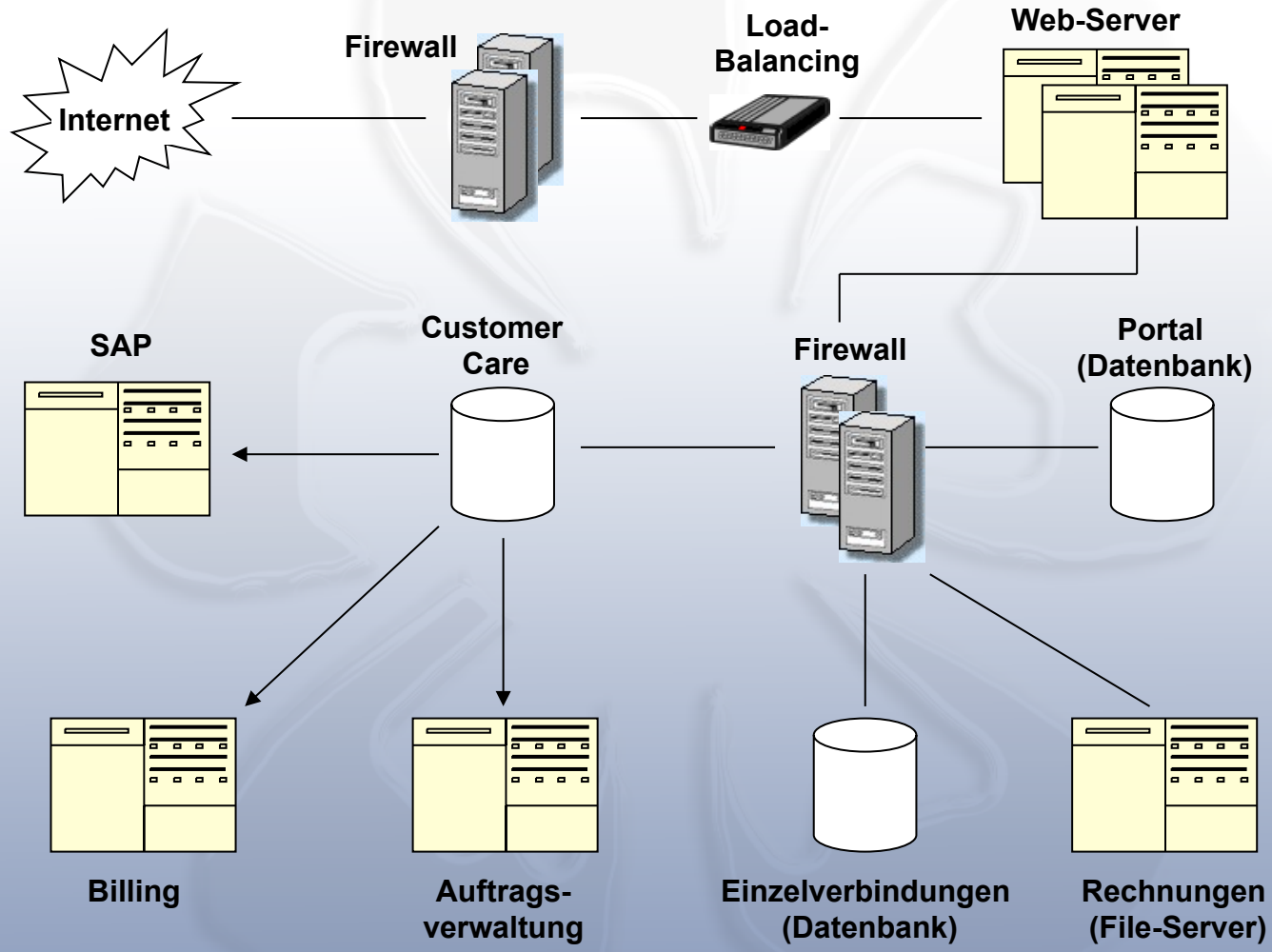
Inhalt

1. Portal
2. Probleme in der Praxis
3. Lösungsansätze
   a. Rekonstruktionsfähigkeit
   b. Defensive Programmierung
   c. Automatisierte Code-Reviews

Portal (1)


Kunden
Außenstellen
Mitarbeiter
Rechnungen
Verbindungen
Tarifwechsel
Produkte
Bankverbindung
Aufträge
Kundendaten
Provisionierung
Aufträge
Reports
gesicherter Bereich
öffentlicher Bereich
Informationen abrufen
Produkt beauftragen
Interessenten
Internet
SAP
CRM
Billing
Auftrags-
verwaltung
Intranet
Rechnungen

Portal (2)


Internet
Firewall
Load-
Balancing
Web-Server
SAP
Customer
Care
Firewall
Portal
(Datenbank)
Billing
Auftrags-
verwaltung
Einzelverbindungen
(Datenbank)
Rechnungen
(File-Server)

Portal - Kennzahlen

- 6 Entwickler, 2 Tester, 2 Administratoren, 1 Projektleiter
- Integrationsprojekt insgesamt: 40 Mitarbeiter

- 140.000 Rechnungen
- 260.000 Benutzer
- 10.000 Erst-Logins / Monat
- 300.000 Sessions / Monat

- Technische Plattform
  - Microsoft.NET ⇨ OOP, umfangreiche Klassenbibliothek
  - VB.NET ⇨ VB6-Know-how vorhanden
  - Oracle ⇨ strategische Unternehmensentscheidung

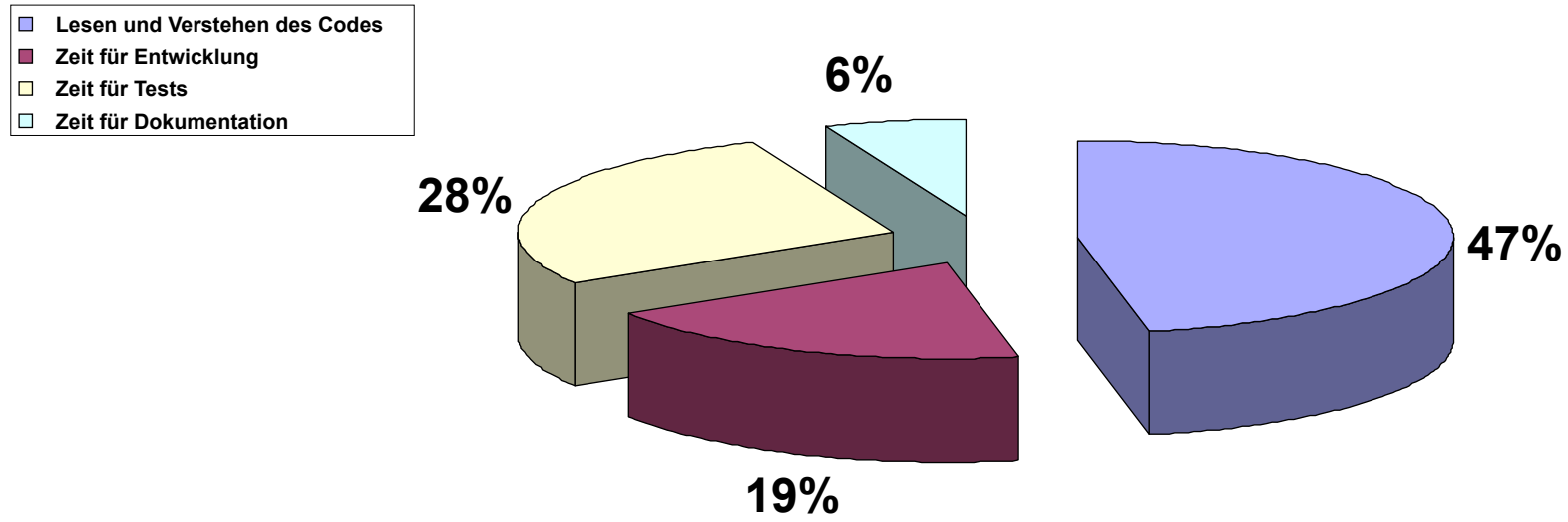
Probleme in der Praxis

- aufwändige Fehleranalyse
- schwer zu reproduzierende Fehler(situationen)
- langwierige Einarbeitung in bestehenden Code

- Warum sind die Wartungskosten so hoch?
- Vertrauen in die IT?

Zu hohe Wartungskosten...

„Software Quality, producing practical and consistent software“
by Ben-Manachem



“47% of a developer ‘s time
is spent analyzing code manually.“

Konsequenzen

Gartner

„At least 80 percent of applications put into production environment will fail due to poor quality issues through 2007.“

Entscheidungsprozesse

- Outsourcing von Projekten
- Offshoring von Entwicklungsabteilungen
- Kauf von Standard-Software

„Kernfrage“

Ist es möglich, qualitativ hochwertige und gleichzeitig kosteneffiziente Portale zu entwickeln?

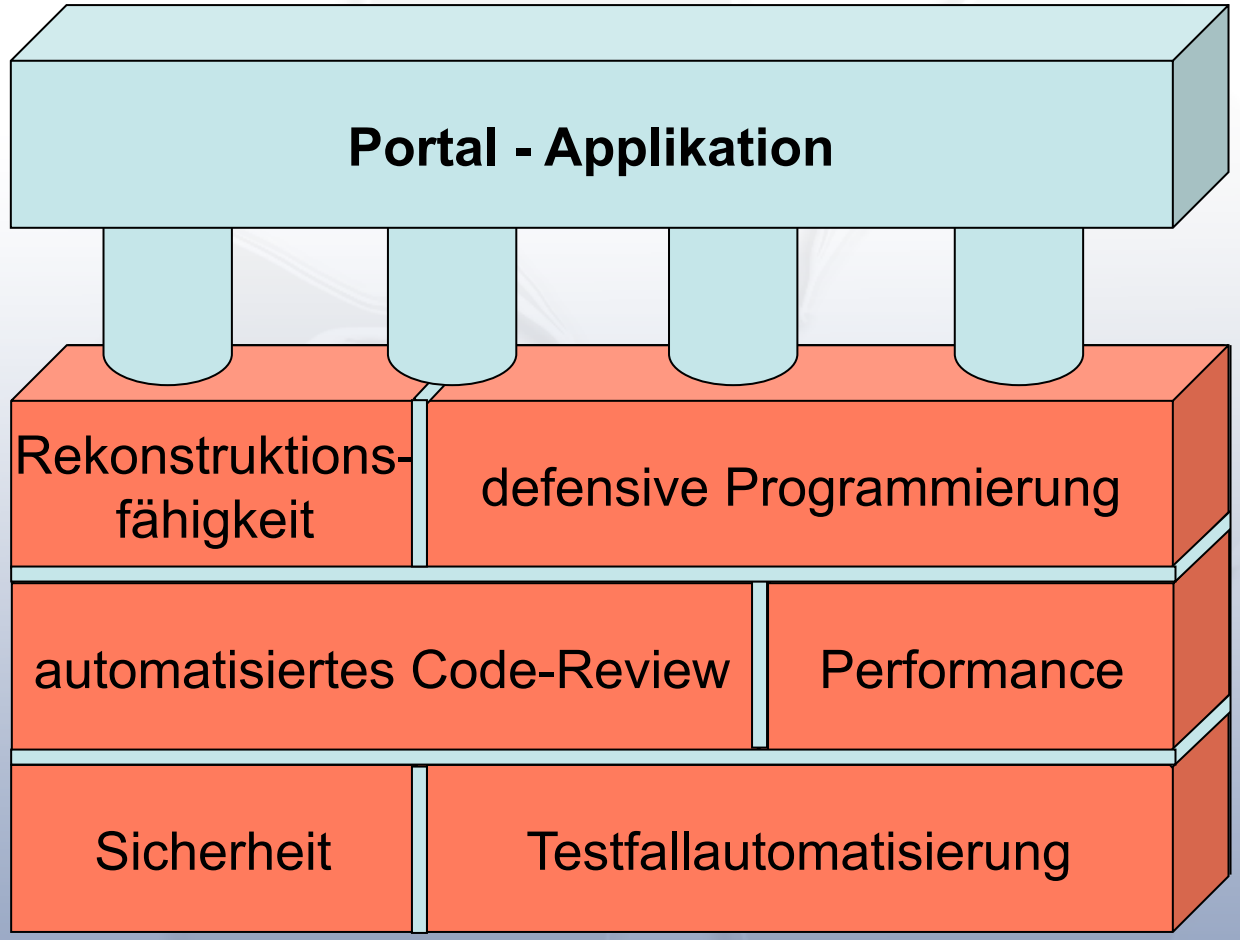
Ziel dieses Vortrags

Vorstellung praktischer Maßnahmen

- Eindämmung der Wartungskosten
- Verbesserung der Softwarequalität

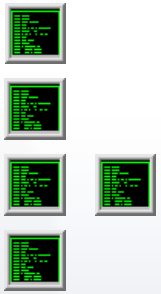
Gordischer Knoten?

Das „Fundament“



Fehlerbehandlung

- ASP: weiße Seite?
- ASP.NET: gelbe Seite?
- Datenbank:
- Fehlerbehandlung:



⇨ Die bereitgestellten Fehlerinformationen sind nicht aussagekräftig!

Fehleranalyse

- keine konkreten Variablen- und Parameterwerte
- Verlauf der bisherigen Benutzer-Interaktionen ist nicht nachvollziehbar
- spezielle Datenkonstellation ist nicht bekannt
- Anwender hat unvollständige Informationen angegeben

- Einarbeitung in den Source
- Testen von mehreren, unterschiedlichen Use-Cases
- produktionsähnliche Daten in der Testumgebung herstellen

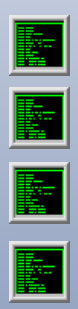
Rekonstruktionsfähigkeit

Wieso Rekonstruktionsfähigkeit?

Weil viele Informationen gesammelt werden müssen, ...

... um zum Zeitpunkt des Fehlers den aktuellen Zustand der Klasse, der lokalen Methode und der Session rekonstruieren zu können.

- Call Stack
- Benutzer-Interaktionen
- Session-Variablen
- Session-Informationen



In fünf Schritten zum Ziel...

... eine rekonstruktionsfähige Portal-Applikation

1. Verwaltung der Web-Server-Sessions in der DB
2. Verwaltung der Session-Variablen in der DB
3. Authentifizierung der Portal-Anwender zum frühstmöglichen Zeitpunkt
4. Protokollierung der Benutzer-Interaktionen in der DB
5. Vermeidung von Datenlöschungen und Protokollierung von Datenänderungen

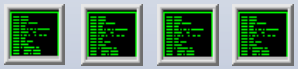
Rekonstruktionsfähigkeit: Wie?

Data Tier

- Datenlöschungen verhindern
- Datenänderungen einschränken
- Datenänderungen protokollieren

Business Tier

- Exception-Klasse
- jede Methode: try/catch
- catch: Zustand protokollieren
  - ⇨ ToString()
  - ⇨ Klasse, Methode, ...
  - ⇨ Verkettete Liste



- Stack Trace:
  - ⇨ Methode – Exception 1:1

Fazit

- Reduktion der Einarbeitungszeit
- Reduktion der Wartungskosten

⇨ Nichts geht ohne Session!!!

- ASP.NET-Applikation
- SOAP-Web-Service
- Grunddatenänderung
- Migration
- Desktop-Applikation

Rekonstruktionsfähigkeit, aber...

„Aufwand“

- kalkulierbarer Aufwand
- einmaliger Entwicklungsaufwand ⇔ vielfacher Wartungsaufwand

„Performance“

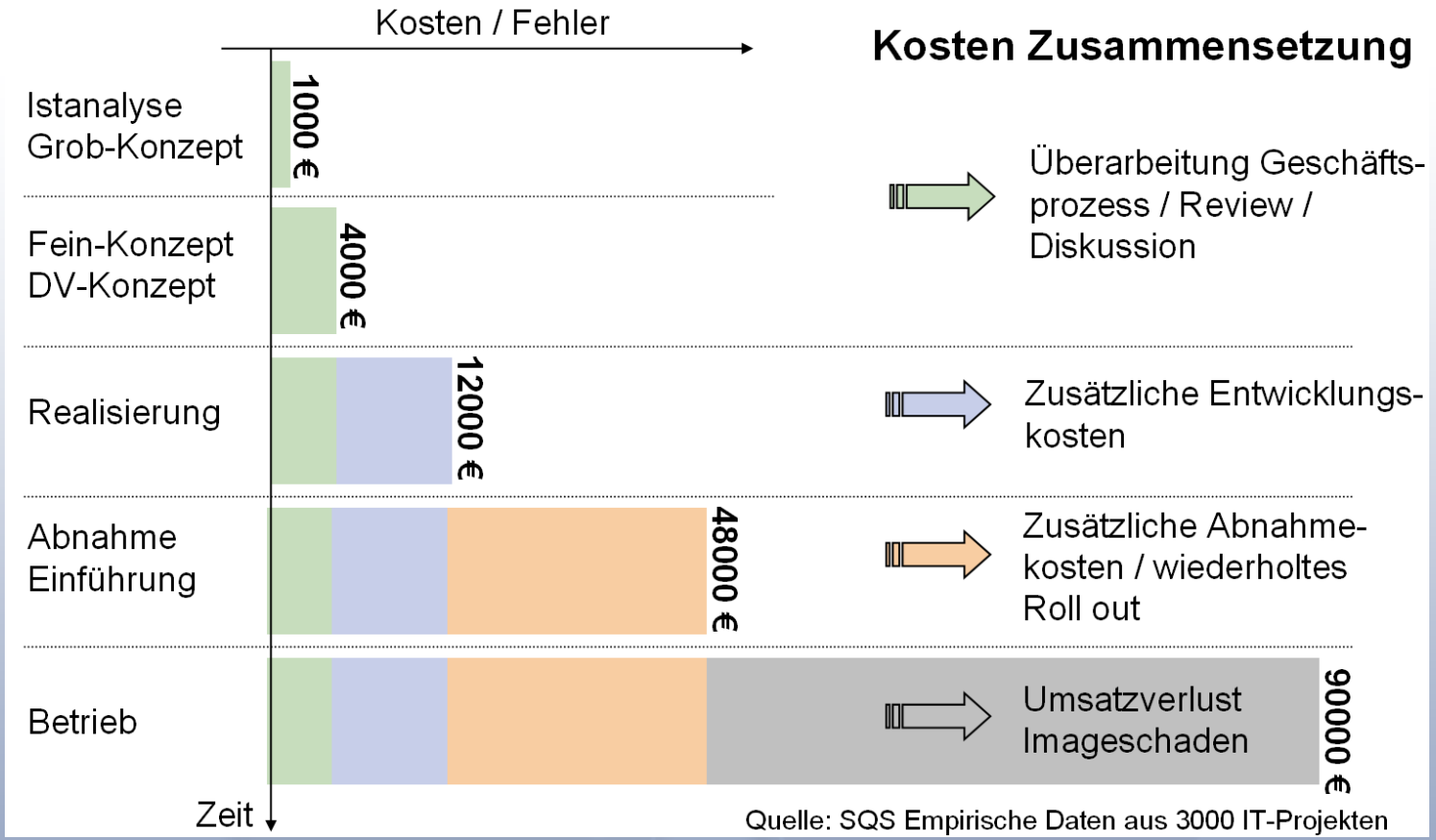
- Im Fehlerfall spielt Performance eine untergeordnete Rolle
- Dauer der Zusammenstellung von Fehlerinformationen ⇔ Dauer der Fehlersuche

„Speicherplatz“

- Protokolle erfordern nicht unerheblichen, zusätzlichen Speicherplatz
- Backup-/Löschjobs, z.B. rollierend für Daten, die älter als 3 Monate sind

Defensive Programmierung

Wie teuer ist ein Fehler?



Wieso defensive Programmierung?

Weil Fehler so früh wie möglich erkannt werden können, ...

...indem so viele Fehlersituationen wie möglich berücksichtigt werden!

- Je mehr Fehlersituationen abgefragt werden, desto stabiler die Software.
- Jede Fehlersituation muss bereits im Entwicklungsprozess hinderlich sein!
- Fehlerfreie Software gibt es nicht.
- Aber man kann zuverlässige Software mit einer sehr geringen Rest-Fehlerrate erstellen.

Ein Beispiel

E-Mail-Adresse eines Users ändern:

Data Tier:

- SetUserEmail()

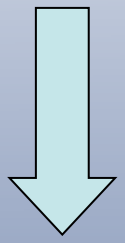
Defensive Programmierung: Wieso?

- Es gibt nur genau einen Weg, der zum Ziel führt.

  ⇨ Alle Schranken müssen erfolgreich passiert werden!

- Auf allen Ebenen wird so defensiv wie möglich programmiert.

- Je „tiefer“ die Software-Schicht, desto mehr Aufwände sind in die defensive Programmierung einzubringen:



1. Presentation Tier
2. Business Tier
3. Data Tier

Defensive Programmierung: Wie?

Beispiel: E-Mail-Adresse eines Users ändern

- Existiert der angegebene User?
- Ist der User inaktiv oder gesperrt?
- Wurde die eMail-Adresse tatsächlich geändert?
- Wer führt die Änderung durch?
- Ist derjenige berechtigt, die Änderung durchzuführen?
- Ist sichergestellt, dass die Änderung nachvollziehbar ist?

SetUserEmail()

- mit vollständiger Fehlerprüfung

Wie viel Fehlerbehandlung muss eine Source enthalten?

- Jede Zeile Source-Code kann potentiell mind. einen Fehler verursachen.
- Mind. 50% des Source-Codes besteht aus Fehlerbehandlung.
- Erfahrungswert: ca. 70% Error-Handling

5 Argumente, ...

... warum der Session-Key ein Eingabe-parameter sein sollte:

1) Revision
2) Protokollierung
3) Vermeidung von Fehlern
4) Vermeidung falscher Anwendungen
5) Sicherheit

Fazit

- Verlassen Sie sich nicht auf die Standard-Mechanismen der jeweiligen Software-Schicht! Die gesetzten Annahmen ändern sich und werden schnell außer Kraft gesetzt...
- Überprüfen Sie alle Voraussetzungen, die für den aktuellen Verarbeitungschritt erfüllt sein müssen.
- Stecken Sie den Rahmen für die Verarbeitung möglichst eng!
- Misstrauen Sie grundsätzlich allen Eingaben. Überprüfen Sie Eingabeparameter auf jeder Ebene der Software-Schicht. Im Zweifelsfall: 3x.
- Binden Sie jegliche Funktionalität an die Session des Users. Nochmals: Nichts geht ohne Session!

Defensive Programmierung

Vergleich: Datenbankprozedur „SetUserEmail()“

- „Anzahl Exceptions“:
  - normale Implementierung: 1 Exception 
  - defensive Implementierung : 11 Exceptions 
- „Anzahl Zeilen Source-Code“:
  - normale Implementierung: 22 Zeilen 
  - defensive Implementierung: 125 Zeilen 

⇨ 5 fach längerer Code !

Automatisierte Code-Reviews

Automatisierte Code-Reviews

Fakten

- Wartung verursacht > 70% der Kosten von Software (TCO)
- Sourcen werden meist nicht vom Autor gewartet
- Ein Entwickler verbringt ca. 47% seiner Zeit mit Einarbeitung in Code

Anforderungen an den Source-Code

- Jeder Entwickler muß Sourcen innerhalb mit einem angemessenen Aufwand verstehen können
- Homogenität

Vorgehensweise

- Codier-Richtlinien
- Code-Reviews

Automatisierte Code-Reviews

Probleme

- Wer überprüft die Codier-Richtlinien?
- Wer führt Code-Reviews durch?
- hoher Anspruch an die Disziplin der Entwickler
- Lösung schwierigerer technischer Probleme ⇔ langweilige Routineaufgaben

Lösungsansatz

- Automatisierung mit Hilfe von Tools
- Integration in die Entwicklungsumgebung (plug-in)
- Integration in den Build-Prozess (make)

Codier-Richtlinien

Namenskonventionen

- systematische Benennung von Entitäten
- Dateien, Assemblies, Namespaces, Klassen, Konstanten, Variablen, Parameter, Präfixe, Methoden, UI-Controls
- Am Namen soll sich die Verwendung ablesen lassen

Programmiermuster

- Zugriff auf Member nur über Properties, selbst innerhalb der Vererbungshierarchie (private, protected, public ⇔ get / set, read only)
- keine „Optional“ Parameter ⇨ Methoden überladen
- Kardinalitäten, z.B. < 7 (# Parameter, # Methoden, Vererbungstiefe)
- Exception-Handling, Logging
- sicherer Code (Reduzierung der Fehlerrate)

Formatierungsregeln

- Zeilenlänge, Dateilänge, Methodenlänge, Kommentierung
- bessere Lesbarkeit

Was wird erkannt?

Toter Code

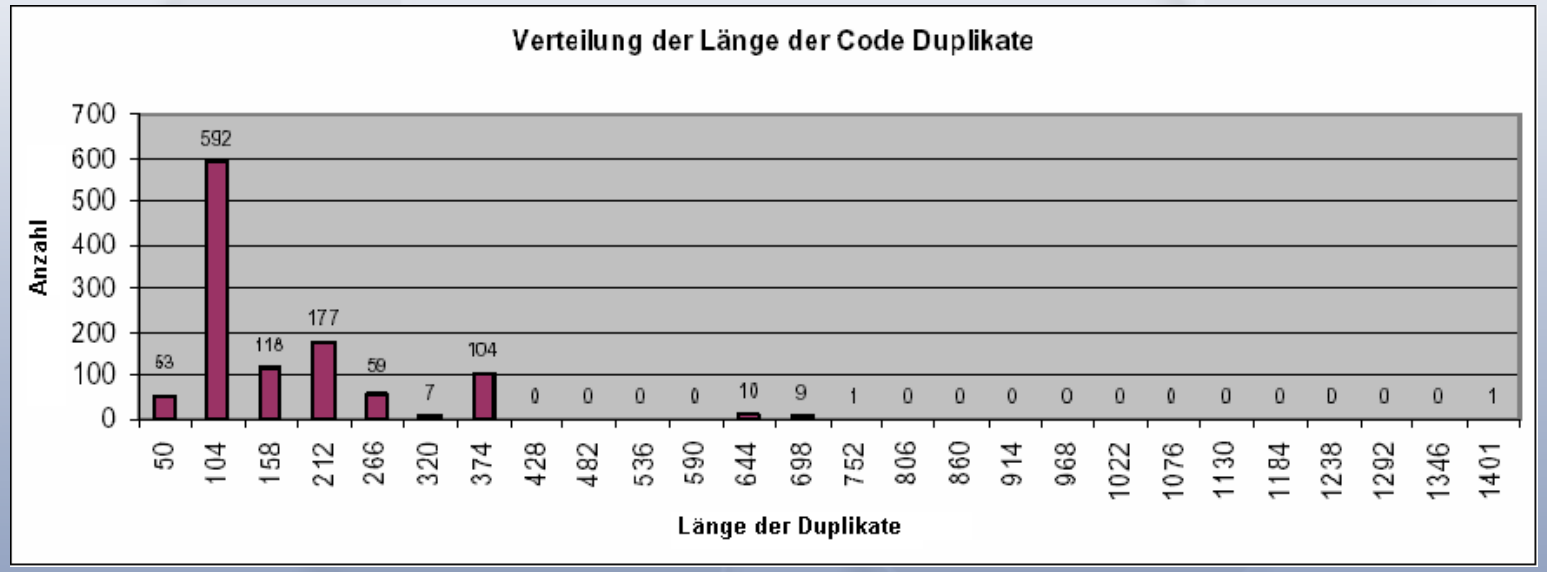
- Variablen, Methoden, Klassen, Typen, Assemblies
  - ⇨ unnötig hoher Speicherbedarf
  - ⇨ Verschlechterung der Performance
  - ⇨ unnötige Wartungsaufwände

Code-Duplikate

- Copy & Paste
  - ⇨ hoher Testaufwand
  - ⇨ hoher Wartungsaufwand
  - ⇨ hoher Dokumentationsaufwand
  - ⇨ unnötig hoher Speicherbedarf
  - ⇨ unnötig große Assemblies

Code-Duplikate

- Beispiel: Rechnungsportal ⇨ VB 6 - Komponenten (Business Tier)
- 15 Projekte (.dll)
- 148 Klassen (.cls)
- 137.844 Lines of Code (LOC)



Leistungsfähigkeit von Tools

- Überprüfung von Namenskonventionen, Codierrichtlinien, Design-Patterns
- Erweiterung der Regelsets durch Regeleditoren
- Code-Browser, Graphen, Baumansichten
- Code-Duplikate
- toter Code
- Auto-Fix

⇨ nachgelagerte Verbesserung des Codes

⇨ hoher Lerneffekt

Tools (1)

FxCop http://www.gotdotnet.com/

- Design Guidelines for Class Library Developers
- Hersteller: Microsoft
- Analyse der Metadaten einer Assembly (MSIL)
- “Constructors should not call virtual methods defined by the class”
- Implementation eigener Regeln



Tools (2)

Project Analyzer

- Hersteller: “Aivosto Oy” http://www.aivosto.com/
- statische Analyse:
  Projektdateien, Klassendateien (Source Code)

CAST Application Mining Suite

- Hersteller: CAST Software http://www.castsoftware.com/
- Multilinguales Repository,
  mehrere Parser, Refactoring
- Abhängigkeitsanalyse zwischen einzelnen Softwareschichten

Zusammenfassung

- ⇨ Fehlerfreie Software gibt es nicht
- ⇨ Wartungskosten können kalkulierbarer gemacht werden
- ⇨ Argumentieren Sie für mehr Zeit und Budget in der Entwicklungsphase
- ⇨ Protokollieren Sie so viele Informationen wie möglich
- ⇨ Erleichtern Sie sich Ihren Alltag durch Automatisierung „unbeliebter“, aber notwendiger Tätigkeiten

Herzlichen Dank für Ihre Aufmerksamkeit

Fragen ???

Manuel Lengert

mlengert@icommit.de

iCommit Integrationslösungen GmbH