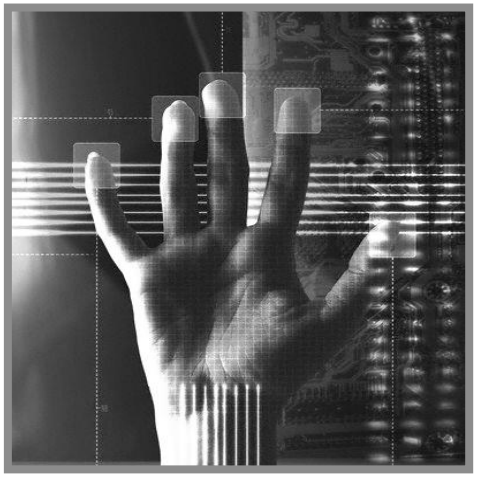
Software Engineering und Projektmanagement



Implementierung

2015W - 5. November 2015

Email: teaching@inso.tuwien.ac.at

INSO - Industrial Software
Institut für Rechnergestützte Automation | Fakultät für Informatik | Technische Universität Wien

Die Implementierung als Teil des Projekts

- **Ziele und Aufgaben der Entwicklung**
  - Umsetzung der Architektur in eine Software
  - Umsetzung der Anforderungen des Kunden
  - Zeitgerechte Lieferung der Software
  - Entwicklung einer validierbaren (abnehmbaren) Software
  - Entwicklung einer wartbaren Software
- **Definition**
  (1) „The process of translating a design into hardware components, software components, or both“
  (2) „The result of the process in (1)“ (IEEE 610.12, 1990)

Grundlagen der Implementierung

- **Zu erfüllende Bedingungen vor der Implementierung**
  - Architektur muss vorhanden sein, je ausführlicher, desto besser.
  - Ein Team muss definiert werden
  - Das Team wurde mit den Charakteristika des Entwicklungsprozesses vertraut gemacht
  - Entscheidungen bzgl. Programmiersprache und Toolunterstützung sollten bereits getroffen sein. Der Wechsel einer IDE oder anderen Tools ist meistens mit hohen Kosten und signifikanten Verzögerungen verbunden
  - Ein grober Zeitplan für die Implementierung wurde erstellt

Die Notwendigkeit einer Architektur bzw. eines Entwurfs

- Architektur und Entwurf sind der Bauplan des Produkts
- Bauplan sorgt eine saubere Umsetzung durch Aufteilung des Produkts in Komponenten mit klaren Verantwortlichkeiten
- Bauplan speziell für Projekte nötig, wenn mehrere Personen an unterschiedlichen Komponenten arbeiten
- Der Entwickler hat eine „Anleitung“ zur Orientierung
- Kreativität trotzdem nötig, WIE die einzelnen Komponenten umgesetzt werden

Architektur während der Implementierung

- **Architektur entwickelt sich während der Implementierung laufend weiter**
- **Microarchitektur wird angepasst und verfeinert**
- **Tägliche Entscheidungen der Entwickler beeinflussen die Microarchitektur**
- **Erfahrungen von Entwicklern notwendig, um die Auswirkungen von Implementierungsentscheidungen einzuschätzen**
- **Architektur wird ebenfalls von außen, durch eine Vielzahl an Stakeholdern beeinflusst (-> Change Management)**

Programmiersprachen

- **Wahl der Programmiersprachen hat wesentlichen Einfluss auf die Implementierungsphase**
- **Selten direkt für das Scheitern eines Projektes verantwortlich**
- **Beherrschung der Sprache notwendig aber nicht hinreichend für Projekterfolg**
- **Faktoren die diese Entscheidung beeinflussen:**
  - Kenntnisse und Erfahrungen des Teams
  - Eignung der Sprache für die Problem-Domäne (Enterprise Systems vs. Mobile Devices / Embedded Systems)
  - Verfügbarkeit von Entwicklungstools, Frameworks und Libraries
  - Kundenvorgabe
  - Verbreitung der Sprache
  - Verfügbarkeit von Entwickler

Frameworks als technische Basis der Implementierung

- **Definition**
  - „Ein Framework ist ein Applikationsskelett, welches vom Entwickler angepasst (spezialisiert) werden kann, um Anforderungen optimal umzusetzen.“
  - „Ein Framework ist ein wiederverwendbarer Entwurf. Dieser ist oft durch eine Menge von abstrakten Klassen und das Zusammenspiel ihrer Instanzen beschrieben.“
- **Die Verwendung von Frameworks ist eine effiziente Art der Softwarewiederverwendung (Software Reuse)**
- **Die Verwendung von Frameworks reduziert die Applikationsentwicklung auf die Spezialisierung vorgegebener Strukturen.**

Frameworks

- **Vorteile:**
  - Reduktion von Entwicklungskosten
  - kürzere Entwicklungszeiten
  - Wiederverwendung von Quellcode und bewährter architektonischer Strukturen
  - Einhaltung von Standards (z.B.: JPA)

- **Nachteile:**
  - „Vererben“ von Fehlern
  - Einarbeitungsaufwand

Abgrenzung zwischen Framework und Bibliothek

- **Definition Bibliothek (Library):**
  - „A controlled collection of software and related documentation designed to aid in software development, use, or maintenance. ...“ (IEEE-Standard 610.12, 1990)
- **Eine Bibliothek bietet wiederverwendbare Funktionalität an, während ein Framework wiederverwendbares Verhalten bereitstellt**
- **Methoden von Bibliotheken werden durch den eigenen Code aufgerufen, während bei der Verwendung von Frameworks der eigene Code durch das Framework aufgerufen wird**
- **Frameworks bestehen aus abstrakten Klassen und bieten Funktionalität in (teilweise) implementierten Methoden an. Diese Methoden können Funktionalität aus Bibliotheken beziehen.**

Vergleich Framework - Bibliothek

Bibliothek

```
public void writeDataToXls(Data data, OutputStream os) {
  HSSFWorkbook wb = new HSSFWorkbook();
  HSSFSheet sheet = wb.createSheet("data");
  HSSFCellStyle dateStyle = wb.createCellStyle();
}
```

Framework

```
@Observer(JpaIdentityStore.EVENT_USER_AUTHENTICATED)
public void loginSuccessful(User user) {
    this.log.info("Authenticated #0", user.getUsername());
    this.user = user;
    LocaleSelector ls = LocaleSelector.instance();
    ls.selectLanguage(user.getPreferredLocale());
}
```

Eigenbau oder vorhandenes Framework?

- **Auswahl des am besten geeigneten Framework ist essentiell – nachträglicher Tausch kommt nahezu einer Neuentwicklung gleich**
- **Auswahlkriterien**
  - Deckt das Framework meine Anforderungen ab?
  - Wie sind die Wartung und die Weiterentwicklung gesichert?
  - Ist das Framework kommerziell oder frei?
  - Wie ist die Verfügbarkeit von Entwicklern mit Erfahrung?
- **Eigenentwicklung nur sinnvoll, wenn**
  - Kein geeignetes Framework existiert
  - Anpassung eines vorhandenen Frameworks zumindest gleich aufwändig wäre
  - Spezielle (nichtfunktionale) Anforderungen es erfordern

Framework Typen

- **Persistence Frameworks**
  - Hibernate, JPA
- **Web Applications Frameworks**
  - Jboss SEAM
  - Struts
  - Spring Webflow
  - Ruby on Rails, Grails
- **Rich Client Frameworks**
  - Eclipse RCP
- **Grafik Frameworks**
  - Direct X
  - Core Graphics

Frameworks - Eigenschaften

- **Method Hooks**
  - Abstrakte Methoden, die vom Entwickler überschrieben werden um das gewünschte Verhalten zu erstellen
- **Hot Spots**
  - „Punkte“, an denen ein Framework spezialisiert wird
- **Whitebox Frameworks**
  - Innere Struktur muss sichtbar sein um es spezialisieren zu können
  - Spezialisierung erfolgt (meist) durch Vererbung
- **Blackbox Frameworks**
  - Enthält stabile Hot Spots
  - Spezialisierung kompositionsbasiert
    - Komponenten werden an das Framework übergeben

Die Entwicklungsumgebung (IDE)

- **Zentraler Arbeitsbereich für Entwickler**
- **Zugriff auf alle benötigten Tools**
- **Einheitliches Interface**
- **Flexibles Layouting und individuelle Gestaltung/Anpassung**
- **Beispiele:**
  - Eclipse
  - NetBeans
  - XCode
  - Kdevelop
  - Visual Studio

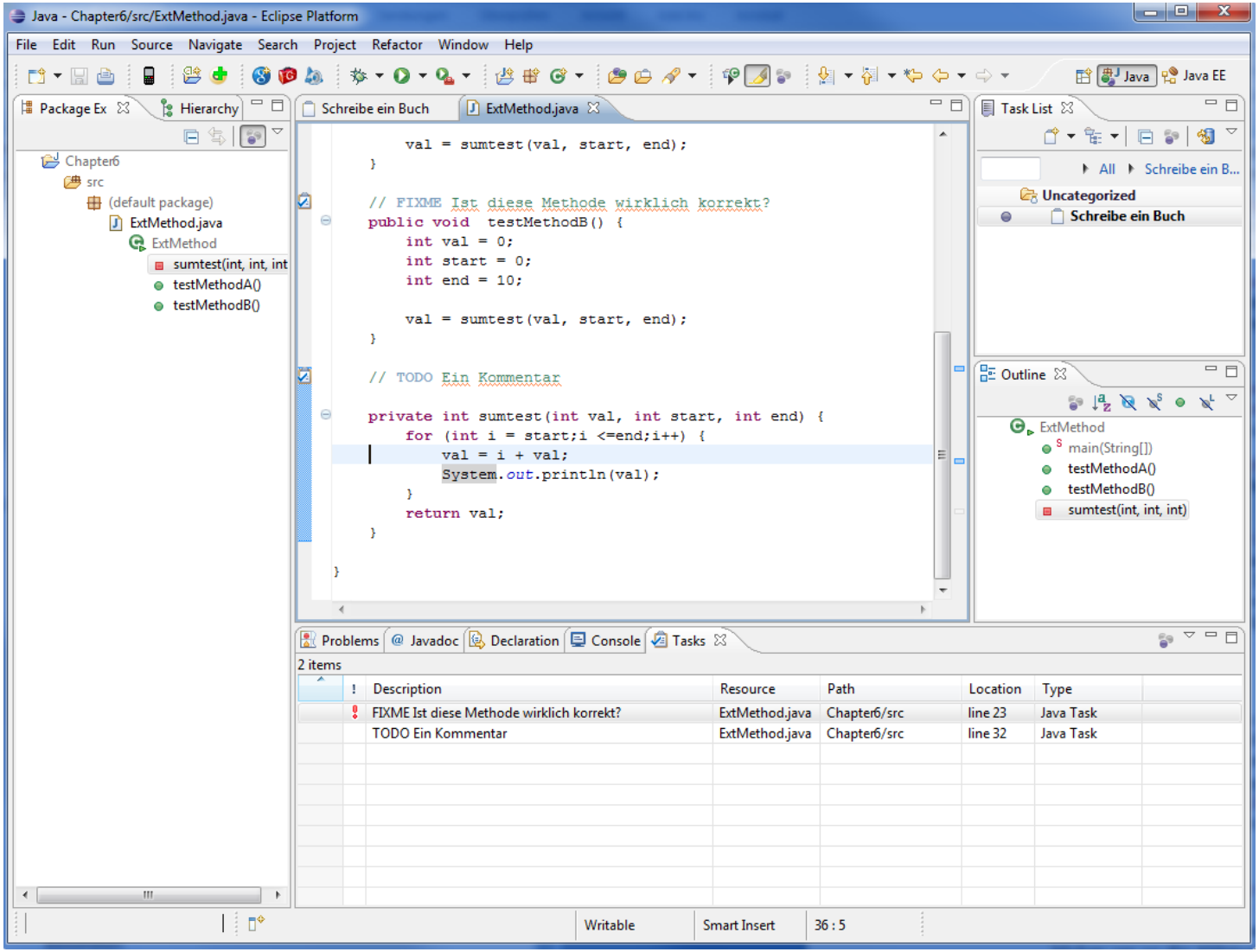
Funktionsumfang einer IDE

- **Source File Editor**
  - Syntax Highlighting
  - Code Completion
  - Snippets / Templates
  - Spezialisiert nach Dateityp (Java, C/C++, XML, …)
- **Integriert Compiler, Linker, Interpreter**
  - Inkrementelles Compilen von Code(fragmenten) im Hintergrund
- **Debugger, Profiler**
- **Unterstützung für Refaktorisierung**
- **Build Tools**
- **Unit Testing**

Funktionsumfang einer IDE (2)

- **Volltextsuche**
  - Oft auch nach Regular Expressions
  - Tagging
- **Semantische Suche**
  - Suche mit Wissen über die Struktur der Programmiersprache (AST – Abstract Syntax Tree)
- **Item Tracking**
- **Repository Synchronisierung**
- **Configuration Management**
- **Erweiterbarkeit durch Plugins**

Beispiel: Eclipse

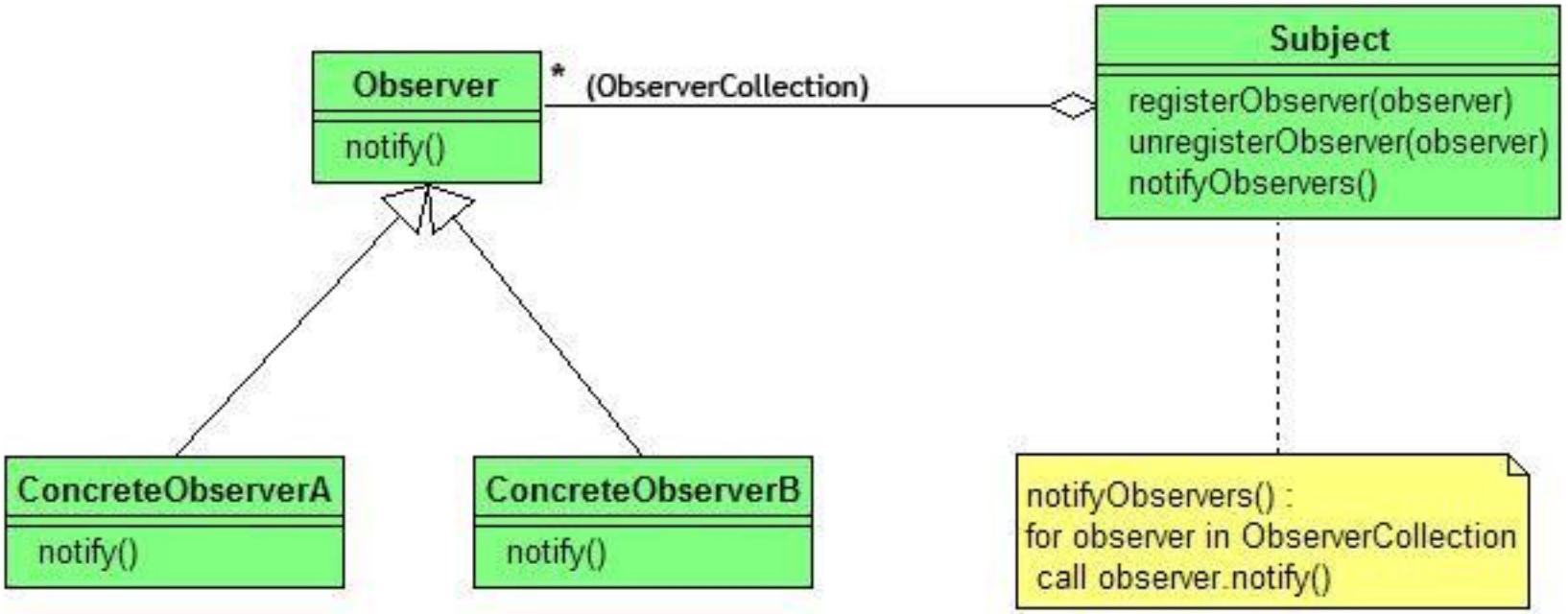


Konzepte für eine Wartungsfreundliche Implementierung

- **Entwurfsmuster (Design Patterns)**
  - Bieten für einen bestimmtes Problem eine gute und bewährte Lösung
  - Große Anzahl an verfügbaren Entwurfsmuster (Gamma et al,…)
- **Creational Patterns**
  - Abstract Factory, Builder, Singleton, Prototype, ...
- **Structural Patterns**
  - Adapter, Bridge, Decorator, Facade, Proxy, ...
- **Behavioral Patterns**
  - Command, Interpreter, Iterator, Observer, Strategy, ...

Konzepte für eine Wartungsfreundliche Implementierung 2

- **Beispiel: Observer Pattern**



Konzepte für eine Wartungsfreundliche Implementierung 3

- **Information Hiding**
  - Kommunikation zwischen Komponenten über Interfaces
- **Kopplung**
  - Maß der Abhängigkeit zwischen Klassen
  - Hohe Kopplung zeugt von einem verbesserungswürdigen Programmierstil
- **Kohäsion**
  - Maß der Abhängigkeit innerhalb von Klassen
  - Hohe Kohäsion wird meist durch Methodengruppierungen nach Zweck gewährleistet

- **Ziel: Geringe Kopplung, erreicht durch hohe Kohäsion**

Kollektiver Codebesitz

- **Jedes Teammitglied hat volle Kontrolle über den kompletten Quellcode**
- **Nicht jeder ist für den gesamten Quellcode verantwortlich**
- **Vor allem in agilen Vorgehensweisen üblich**
- **Vorteil:**
  - Jeder Entwickler kann Änderungen und Verbesserungen im gesamten Code vornehmen
  - Das Wissen über Implementierungsdetails ist besser verteilt
  - Die Auswirkungen des Ausfalls eines Teammitglieds sind geringer
- **Nachteil:**
  - Verantwortlichkeit für „schlechten“ Code ist schwerer Nachvollziehbar

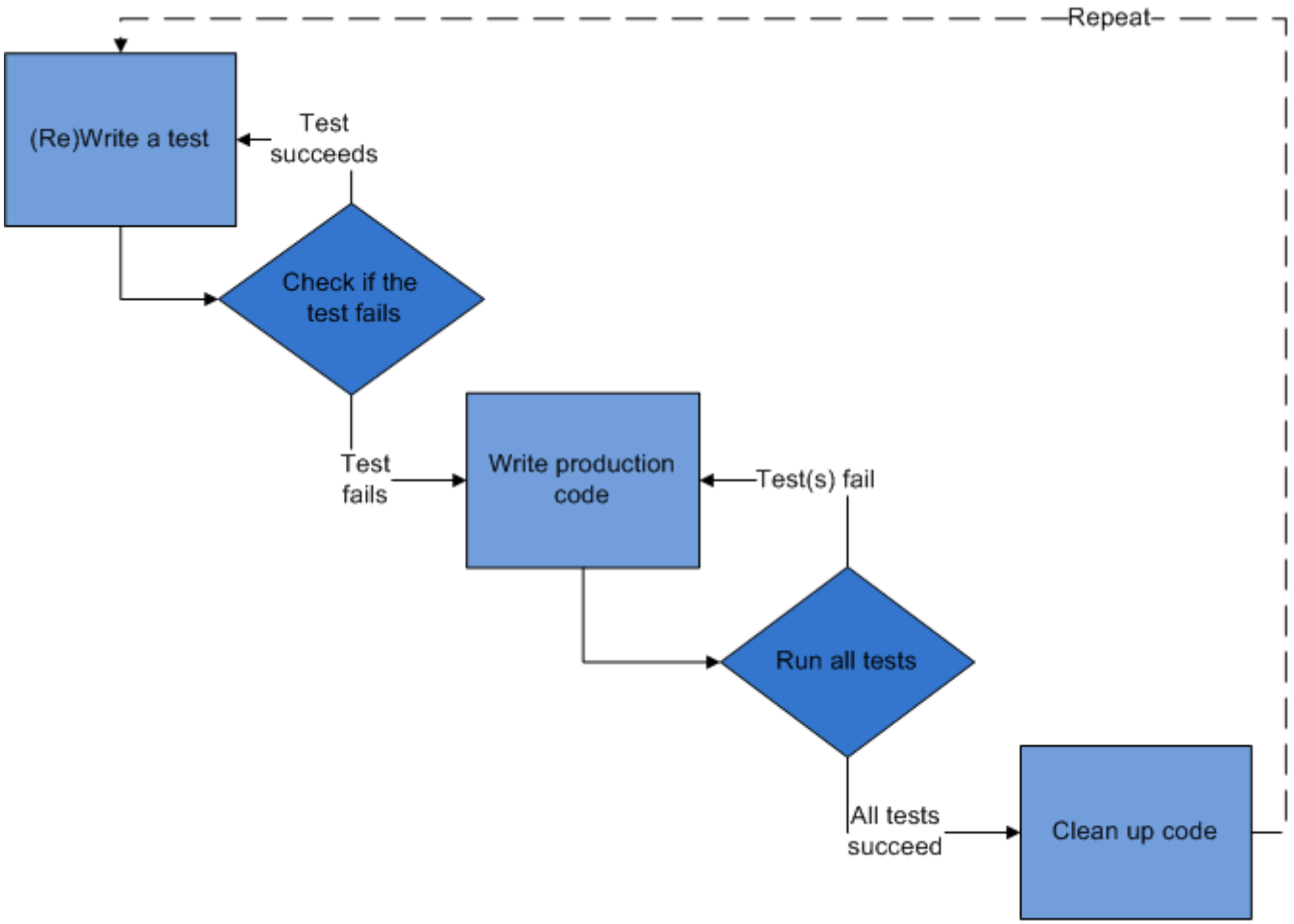
Automatisierte Tests

- **Jeder Entwickler ist auch Tester**
- **Tests oft unstrukturiert, oberflächlich und lückenhaft – da lediglich „ausprobieren“**
- **Lösung automatisierte, feingranulare Funktionstests (Unit Tests)**
- **Einfach ausführbar (Unterstützung durch IDE, Integration in Build Prozess)**
- **Sehr wichtig zur Vermeidung von ungewollten Nebeneffekten**

Test Driven Development (TDD)

- **Definition**
  „*TDD is a style of development where*
  - *You maintain an exhaustive suite of programmer tests,*
  - *No code goes into production unless it has associated tests,*
  - *You write the test first,*
  - *The tests determine what code you need to write*“. (Astels 2003)
- **Zwei goldene Regeln (Beck 2002)**
  - „*You should write new business code only when an automated test has failed*“
  - „You should eliminate any duplication that you find“

TDD - Methodik



Refaktorisierung

- **Definition:**

  „First, the purpose of refactoring is to make the software easier to understand and modify. [...]

  Only changes made to make the software easier to understand are refactorings. [...]

  Refactoring does not change the observable behavior of the software.“

  (Fowler 1999)

Refaktorisierung (2)

- **Motivation:**
  - Verbesserung des Designs von Quellcode
  - Verbesserung der Lesbarkeit von Quellcode
  - Verstehen von fremden oder altem Quellcode

- **Jede erfolgreiche Refaktorisierung ist eine Investition, die sich durch die Effizienzsteigerung bei darauffolgenden Erweiterungen amortisiert.**

Refaktorisierung (3)

- **Vorgehensweise:**
  - Schlecht strukturierten Code identifizieren
  - Ausreichende Testabdeckung sicherstellen
  - Refaktorisierung anwenden
  - Test durchführen
  - Refaktorisierung abschliessen

- **Funktionale Anpassungen am Quellcode erst nach Abschluss der Refaktorisierung**

Refaktorisierung (4)

- **Beispiele (Patterns)**

- **Ähnlich wie Design Patterns, gibt es Refaktorisierungsmuster**
  - Rename Method
  - Extract Method/Class
  - Pull Up Method
  - Split Temporary Variable

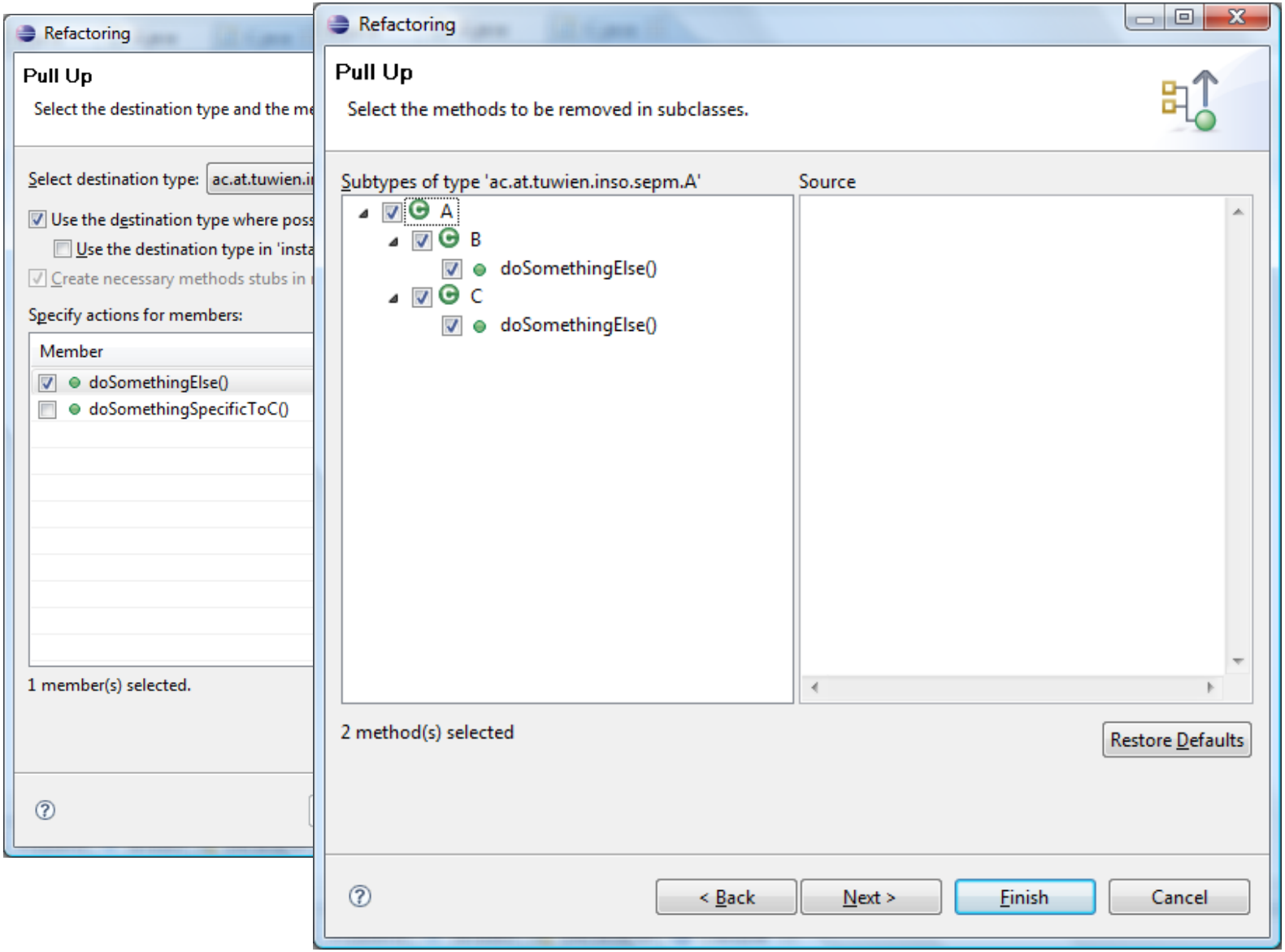
Refaktorisierung – Beispiel – Pull Up Method

```
Class A {
   doSomething();
}
Class B extends A {
   doSomethingSpecificToB();
   doSomethingElse();
}
Class C extends A {
   doSomethingSpecificToC();
   doSomethingElse();
}
```

```
Class A {
   doSomething();
   doSomethingElse();
}
Class B extends A {
   doSomethingSpecificToB();
}

Class C extends A {
   doSomethingSpecificToC();
}
```

Refaktorisierung – Beispiel – Pull Up Method in Eclipse



Dokumentation des Codes

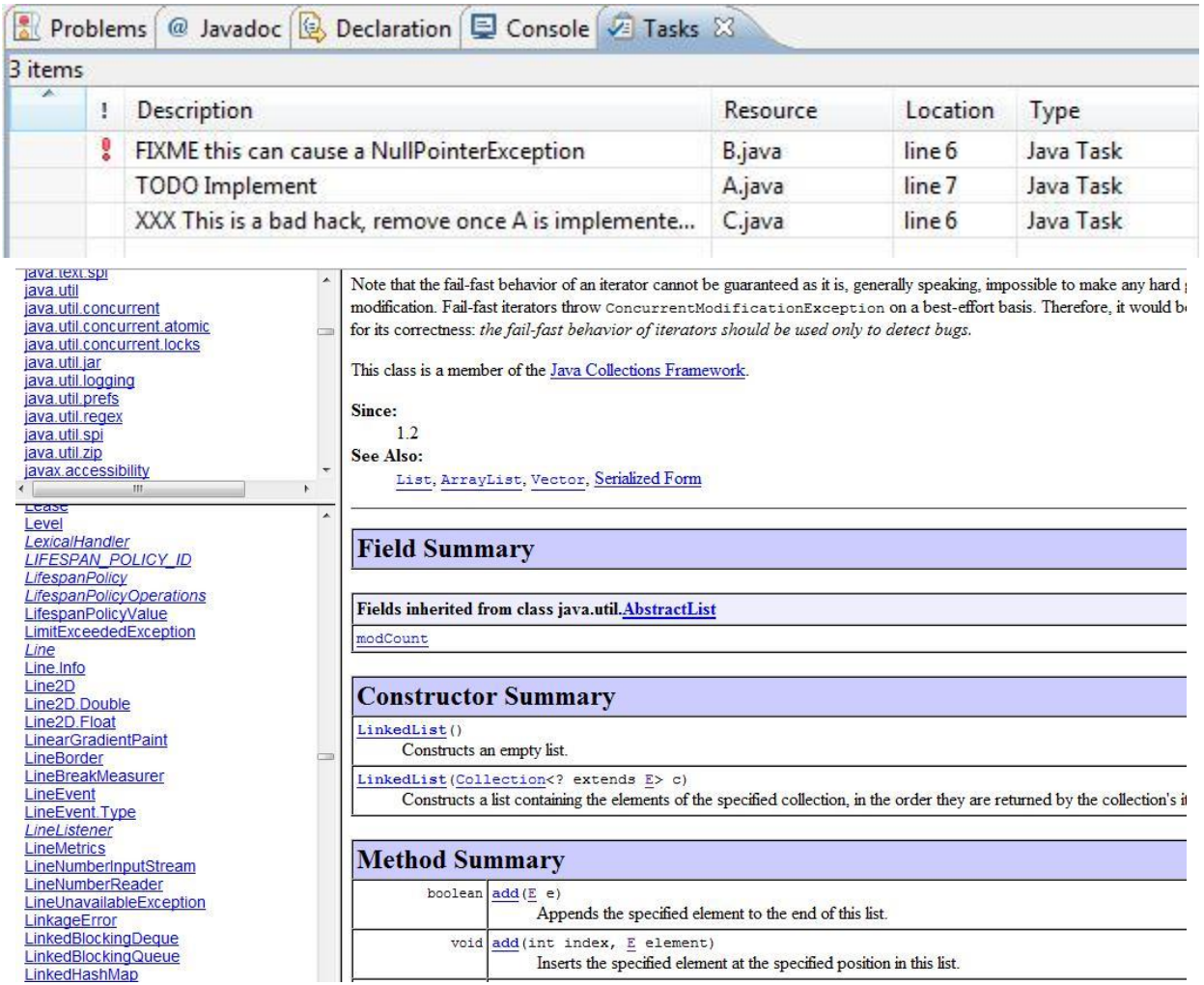
- Erhöht die Lesbarkeit und Übersicht
- Code kann auch nach Monaten wieder verstanden werden
- Code kann von anderen Entwicklern besser nachvollzogen werden
- Kommentierung
- Entwicklungsrichtlinien
- Oftmals die einzige Art der Dokumentation die zum Zeitpunkt der Ablöse des Systems noch erhalten ist

Kommentare

- **Bestandteil der Programmiersprache**
- **In Quellcode eingebettete Anmerkungen**
- **Typen**
  - Blockkommentare (z.B.: `/* […] */`)
  - Zeilenkommentare (z.B.: `// […]` oder `# […]`)
- **Nicht jede Zeile kommentieren, lediglich Intention hinter einem Codeblock**
- **Besonders wichtig: Erklärung der Umstände bei Wahl einer „unkonventionellen“ Implementierungsvariante**

Kommentare – erweiterter Nutzen

- **Auskommentieren aus Debuggründen**
- **Tagging**
  - TODO
  - NOTE
  - FIXME
  - …
- **Doku**
  - Javadoc
  - Doxygen
  - Ddoc
  - …



Entwicklungsrichtlinien

- Code muss nach längerer Zeit und von projektfremden Personen verstanden werden können
- Es werden daher Regeln hinsichtlich Programmierstil, Vorgehensweisen und Methoden festgelegt
- Stark von Programmiersprache und (oft) vom spezifischen Projekt abhängig
- Verbessern die Lesbarkeit des Quellcodes
- Sollen in jedem Projekt definiert und durchgesetzt werden
- Teilweise automatisiert überprüfbar (Checkstyle,…)

Internationalisierung (I18n)

- **„sämtliche Maßnahmen, eine Software so zu gestalten, dass diese möglichst einfach an andere Sprachen und Kulturen angepasst werden kann“**
- **Lokalisierbarkeit: Wie gut lässt sich ein Produkt übersetzen, ohne in den Source Code eingreifen zu müssen?**
- **Lokalisierung (I10n)**
  - Der „eigentliche“ Übersetzungsprozess
  - Anpassung von lokalen Datenformatierungen (Zahlen, Datum, Währung,…)
  - Hinzufügen von spezifischen Code
  - Für jede Sprache/Kultur separat notwendig
  - Vollständige Lokalisierung sehr aufwändig (Leserichtung, …)

Internationalisierung - Tipps

- **Kein „hard-coden“ von Nachrichten und Beschriftungen**
- **Extrahieren in Ressource Files**
  - Alle Teile, mit denen der Benutzer agiert
  - Kommentierung
  - Sinnvolle Gruppierung
- **Schriftgrößen nicht einschränken**
- **Nachricht nicht „stückeln“**
- **Identische Nachrichten in unterschiedlichem Kontext mehrfach speichern**
- **Gleich bei Implementierungsbeginn vorsehen – nachträgliche Umsetzung sehr teuer**

Fehlerbehandlung (Exception Management)

- **Bereits in der Entwurfsphase ist ein entsprechendes Fehlerbehandlungskonzept zu definieren**
  - Die Entscheidung wie mit Fehlern zu verfahren ist, sollte nicht in die Hand der Programmierer gelegt werden
- **Exception Handling ist in den wichtigsten Programmiersprachen vorhanden**
  - Beispiel Java: `try, catch, finally` Blöcke
  - Im try-Block befinden sich Anweisungen, die einen Fehler erzeugen können
  - Im catch-Block wird der Fehler behandelt
  - Der finally-Block ist optional. Anweisungen im finally-Block werden auf jeden Fall ausgeführt

Laufzeitprotokollierung (Logging)

- **Verwendung einer Logging-API (zB Log4J)**
  - Schlechte Praxis: mit zB system.out.println direkt auf die Konsole zu „loggen“
- **Logging APIs stellen üblicherweise folgende Log Levels zur Verfügung**
  - **DEBUG**: feingranulare Informationen zur Programmausführung
  - **INFO**: wichtige Programmereignisse, die im regulären Betrieb vorkommen
  - **WARN**: unerwartete Situationen
  - **ERROR**: ein Fehler ist aufgetreten, die entsprechende Fehlerbehandlung wurde eingeleitet
  - **FATAL**: ein nichtbehandelbarer Fehler ist aufgetreten – Programm kann nicht stabil fortgesetzt werden

Build Management

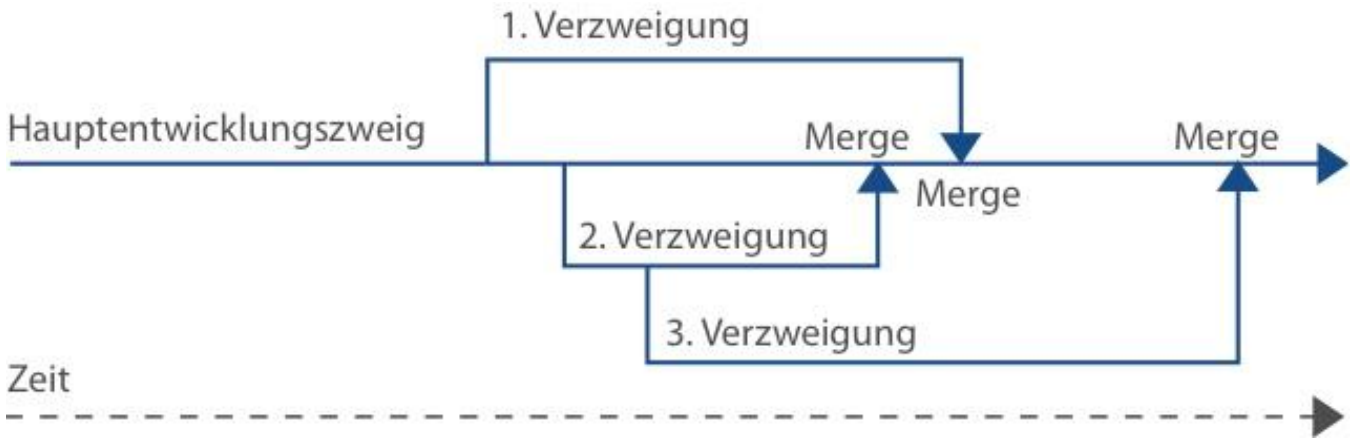
- **Ein Buildmanagementsystem ist ein Werkzeug um immer wiederkehrende Aufgaben zum Bauen einer Applikation automatisiert ablaufen zu lassen**
- **Stellt Abhängigkeiten zwischen Komponenten sicher**
- **Gängige Buildmanagementsysteme**
  - Make
  - Ant, nAnt
  - Maven

Versionsmanagement

- **Definition:**
  „Versionsmanagementsysteme sind Computersysteme, die Änderungen an Dateien und Verzeichnissen über die Zeit hinweg archivieren und in einem oder mehreren Entwicklungszweigen für die gemeinsame Bearbeitung durch mehrere Personen bereitstellen“
- **Der Datenbestand aus dem die Arbeitskopie generiert wird, wird Repository genannt**
- **Checkout**
  - Übertragung (Abholen) der Daten aus dem Repository
- **Check-in oder Commit**
  - Übertragung der Daten in das Repository

Versionsmanagement – wichtige Funktionen

- **Verzweigung / Branching**
  - Unter Branching versteht man das Anlegen einer Kopie von Objekten im Versionsmanagementsystem



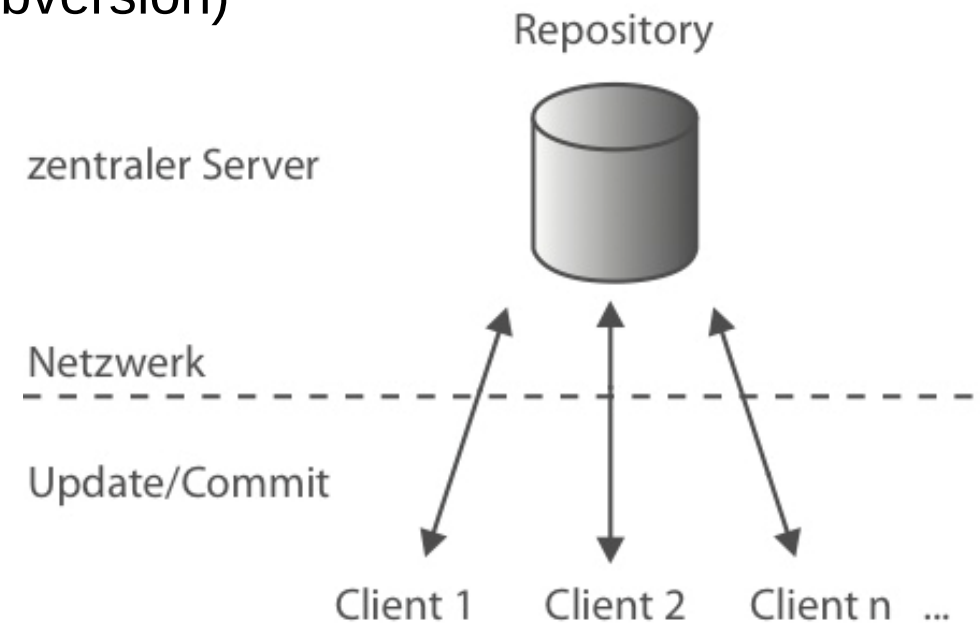
Versionsmanagement – wichtige Funktionen 2

- **Markieren / Tagging**
  - Ein Tag kennzeichnet einen konkreten Entwicklungsstand
    - Beispiel: eine zur Auslieferung bestimmte Version

- **Protokollierung / Historisierung**
  - Protokolliert welche Änderungen von welchem Benutzer zu welcher Zeit vorgenommen wurden
  - Ermöglicht es jeden beliebigen Versionsstand aus der Vergangenheit wiederherzustellen

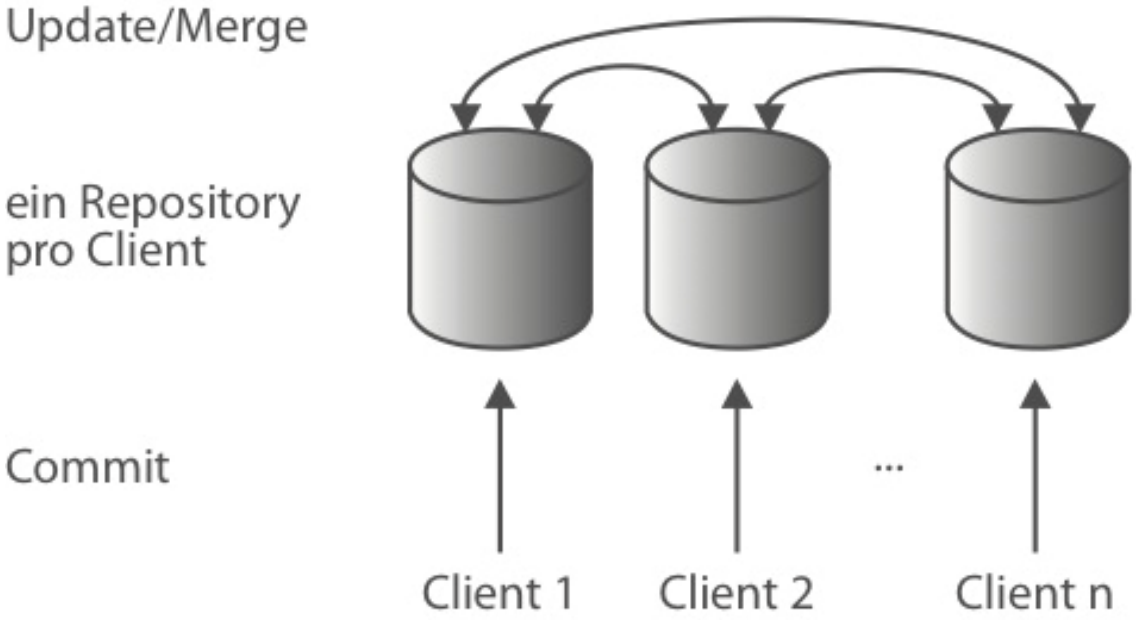
Zentrales Versionsmanagement

- **Client / Server Ansatz**
  - Zentraler Server
  - Beliebig viele Clients
- **Beispiele**
  - CVS (Concurrent Versions System
  - SVN (Subversion)
  - Perforce



Dezentrales Versionsmanagement

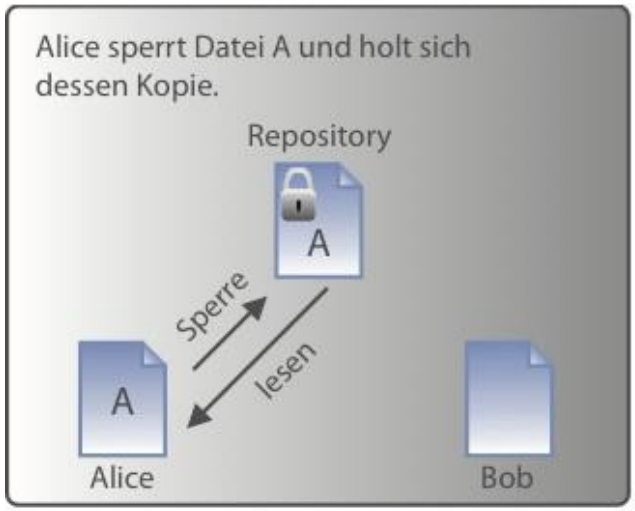
- **Kein zentrales Repository**
- **Jeder Entwickler besitzt sein eigenes Repository**
  - Änderungen werden zwischen den Repositories ausgetauscht
- **Beispiele**
  - Git
  - Bazaar

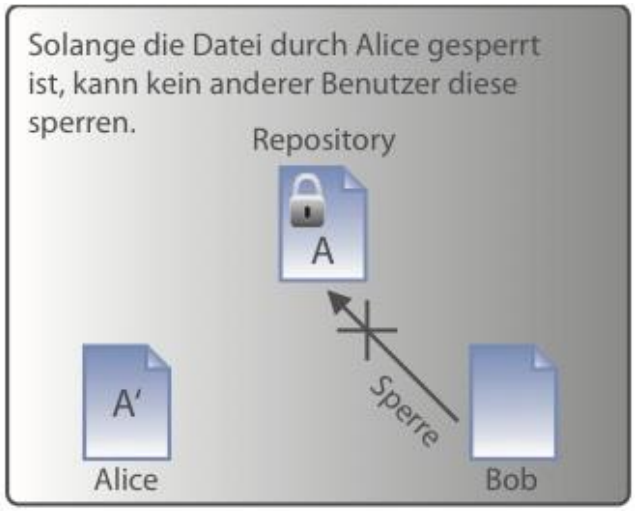


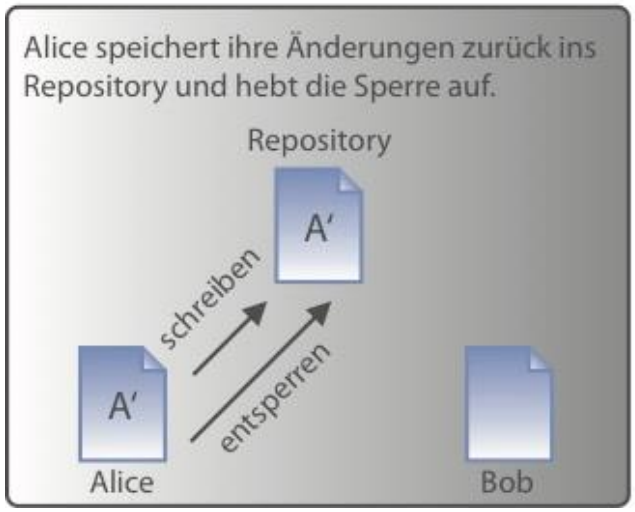
Lokales Versionsmanagement

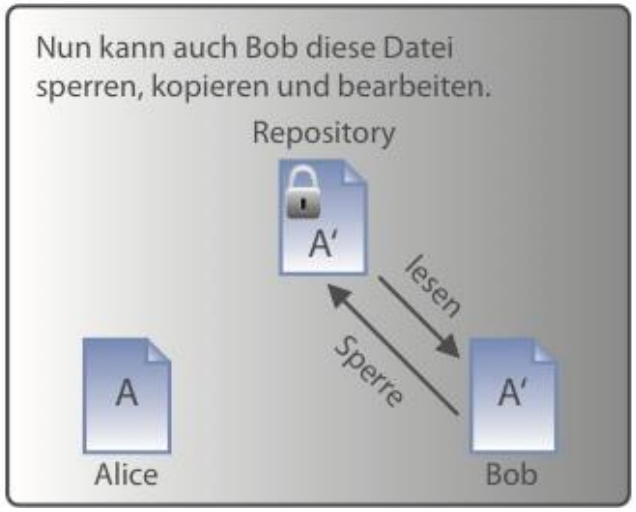
- **Älteste Form von Versionsverwaltung**
- **Beschränkt sich auf die Versionierung einzelner Dateien**
- **Nicht geeignet für Team-Projekte**
- **Beispiele**
  - RCS (Revision Control System)
  - SCCS (Source Code Control System)

Lock-Modify-Unlock Strategie (Pessimistic Revision Control)

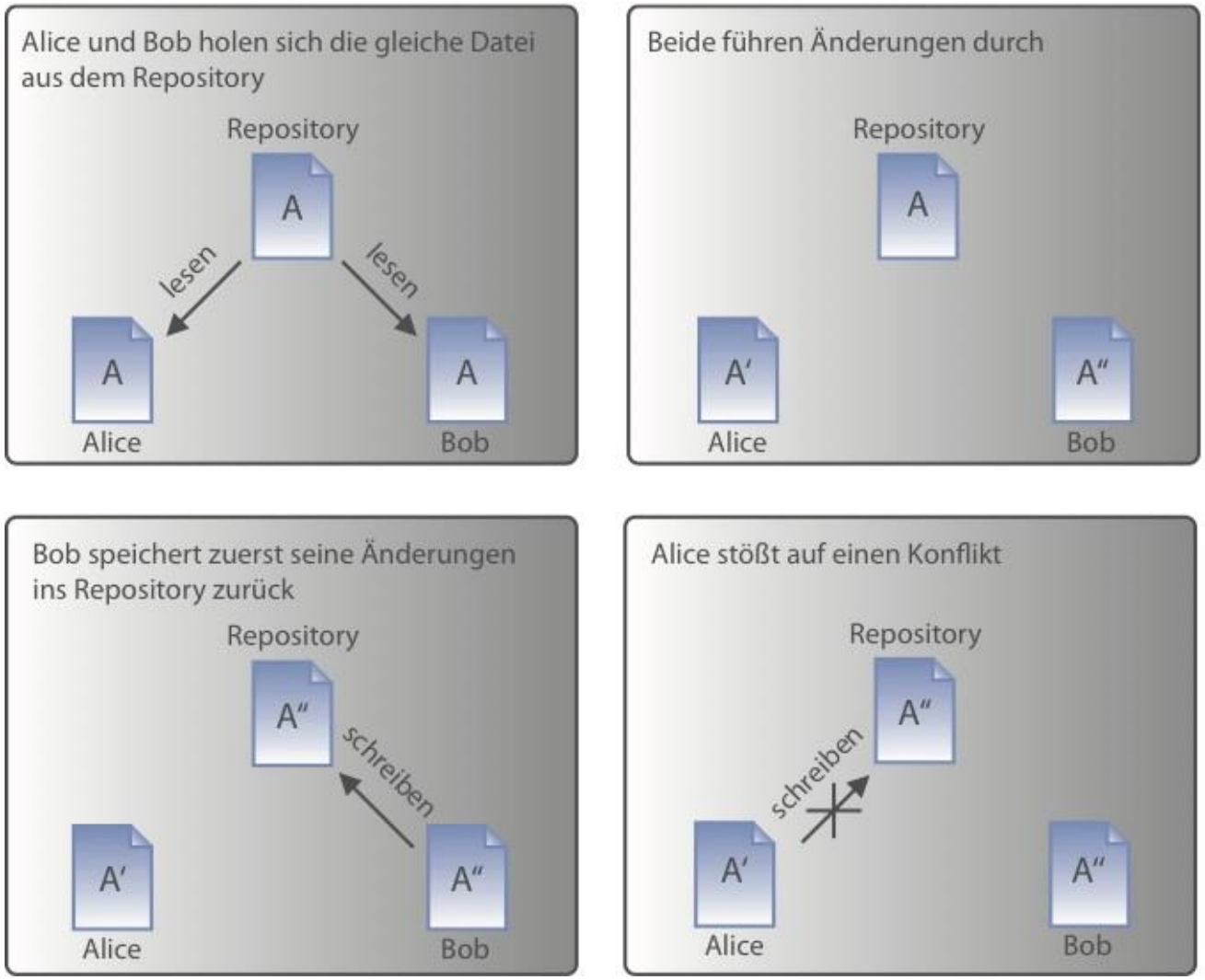




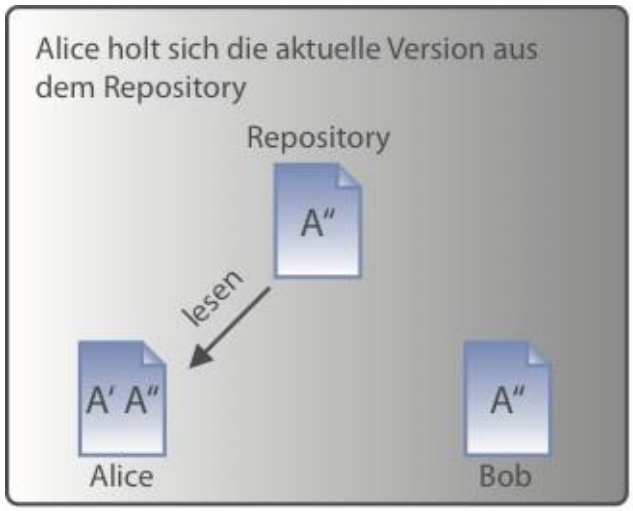


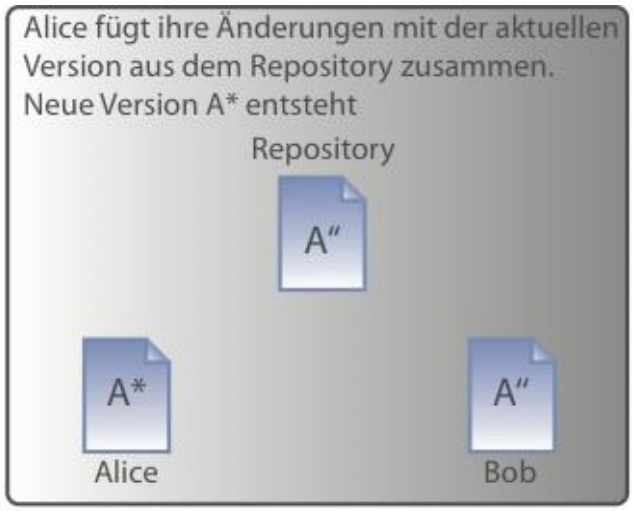


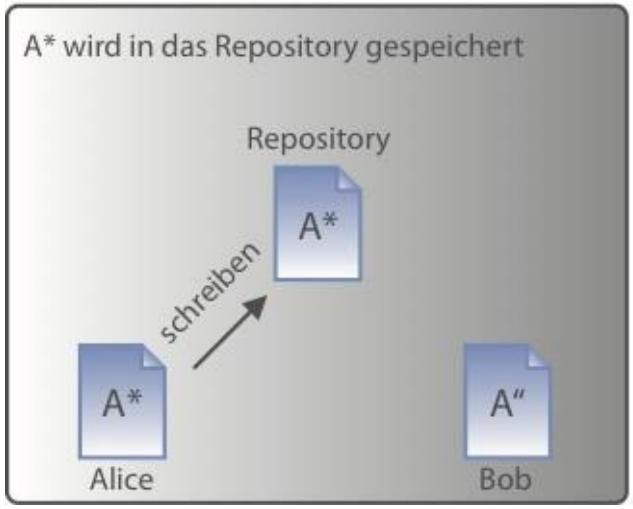
Copy-Modify-Merge Strategie (Optimistic Revision Control) 1

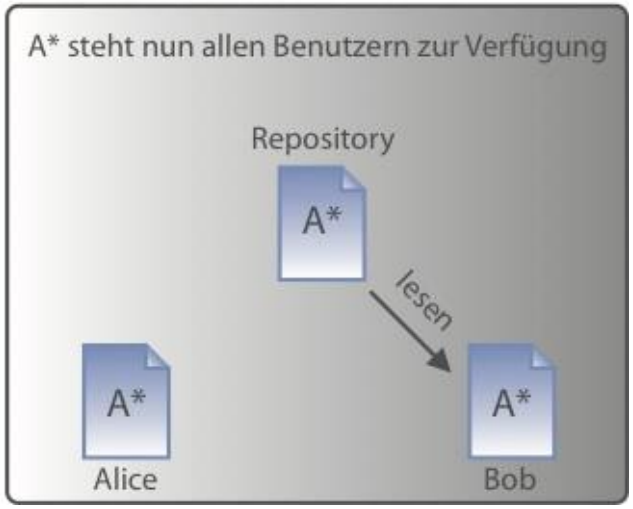


Copy-Modify-Merge Strategie (Optimistic Revision Control) 2









Zusammenfassung

- **Implementierungen im Team sind ein komplexer Prozess**
- **Die Wahl der geeigneten Frameworks können den Entwicklungsprozess positiv beeinflussen**
- **Bei der Entwicklung von (größeren) Projekten ist die Unterstützung durch Tool unerlässlich**
- **Dokumentation im Code und die Einhaltung von Entwicklungsrichtlinien erleichtern die Kommunikation und nachfolgende Wartungsphase**
- **Die Entwicklung im Team erfordert definierte Entwicklungsprozesse**