Q & A Moderne Programmiertechniken und Methoden

KE 1

- Können Interfaces eine Implementierung vorgeben? (eigentlich nutzt man dann doch eher Abstrakte Klassen, oder nicht?)
- Nominale vs. strukturelle Typenkonformität -> semantische Übereinstimmung -> Was ist „Dynamische Konformität“, welche von Android angeboten wird?
- S. 37 -> Welches Wort fehlt?

Ertu:

- - PDF Seite 20: Dynamische Konformität?
- - Ist es überhaupt wichtig?
- - PDF Seite 21: Was heißt "angeboten" und "benötigt", habt ihr ein Beispiel für die Visualisierung?
- - PDF Seite 21: "allerdings ist das benötigte Interface einer Komponente in der Regel das angebotene einer anderen (vgl. Abbildung 1.2)."
- - PDF Seite 23, Abschnitt 1
- - PDF Seite 33, "
- Tatsächlich werden in
- C# und JAVA, denen beiden die Mehrfachvererbung fehlt, Familieninterfaces anstelle von
- abstrakten Klassen eingesetzt, wenn die sie implementierenden Klassen von anderen, nicht
- die Familie repräsentierenden Klassen erben sollen.
- - PDF Seite 37: Disskusion des Codes wäre gut
- - PDF Seite 39: Können wir nochmal über Server/Item Interfaces sprechen bitte?
- - PDF Seite 42: Ich verstehe die Interface Injection nicht
- - PDF Seite 44: Wann ist die Dependency Injection nicht einsetzbar
- - PDF Seite 48/9: Letzter Absatz bzw. erster Absatz auf Seite 49. Ich behaupte, wer sowas schreibt, will überhaupt nicht das es verstanden wird. Es ist ein Flex.
- Klausurfragen aus den Protokollen:
- · Was ist ein Interface?
  - - Ertu: Der IEEE definiert ein Interface als "eine gemeinsame Grenze, über die hinweg Information gereicht wird."
  - Im Software Engineering ist mit Ganze praktisch immer Modulgrenze gement. Ein Interface definiert lediglich die Methodensignaturen einer Klasse, implementiert sie aber nicht.
  - Klasseninterfaces: Schnittstellenspezifikationen, die durch öffentliche Access modifier (in JAVA public) an Ort und Stelle der Implementierung deklariert werden
- · Kennzeichen interfacebasierter Programmierung
  - - Ertu:
    - Interfaces als Typen (in Java und C#) anstelle von Klassen bei der Typisierung von

Variablen (Variablendeklarationen)

- Interfaces ermöglichen Polymorphismus und dynamisches Binden ohne Vererbung
- Interfaces als Schnittstelle: Nach IEEE: gemeinsame Grenze, über die hinweg Information gereicht wird, hinter der die Klasse ihre Entwurfsentscheidungen verbergen kann (Geheimnisprinzip)
- „Interface-Vererbung“: Interfaces sind in der Regel kein Ersatz für Mehrfachvererbung (Man vererbt am ehesten die Schnittstellen, aber keine Implementierung)
- Entkopplung durch Verwendung von Interfaces
- Interfaces als Rollen
- Aufgerufene Klassen implementieren Interface welche Aufrufende Klassen für Variablen verwenden
- In JAVA und C# definieren Interfaces genau wie Klassen Typen (vgl. Kurs 01814), geben aber — anders als Klassen — keine Implementierungen vor.

- · Interfaces als Rollen.
  - - Ertu:
    - Verwendung von Interfaces als Typen von Variablen
      - Variable drückt häufig eine Beziehung eines Objektes zu einem anderen aus. Jede Beziehung definiert Rollen, die die Funktion der involvierten Objekte festschreiben

- · Unterschied zwischen einem Black Box- und einem White Box-Framework
- Bei Blackbox-Frameworks erfolgt das Aufrufen über *Komposition* und *Delegation* bzw. Forwarding, bei Whitebox-Frameworks über Vererbung und offene Rekursion.
- · Prinzipien der interfacebasierten Programmierung
- Prinzip wiederverwendbaren objektorientierten Designs: „Program to an interface, not to an implementation“
- Interface Segregation Principle ISP (im Zusammenhang mit partiellen Interfaces), entspricht entkopplung von Implementierung.
- Dependency Inversion Principle (Umkehrung von Abhängigkeiten mit Interfaces)
- Geheimnisprinzip
- · Nominale vs. Strukturelle Typkonformität
- S.20 https://vu.fernuni-hagen.de/lvuweb/lvu/file/FeU/Informatik/2014WS/01814/oeffentlich/1814-Schnuppermaterial.pdf
- Zur strukturellen Typkonformität reicht es aus, wenn der konforme Typ zu dem er konform sein soll enthält.
- Für die nominale Konformität muß zusätzlich und explizit die Erweiterung eines (oder Ableitung von einem) anderen Typ angegeben werden.
- · Öffentliche vs. Veröffentliche Interfaces
- Die öffentliche Schnittstelle im Wesentlichen dem entspricht, was man in JAVA and ähnlichen Sprachen mit Schlüsselwort public herstelt.
- Die veröffentliche Schnittstelle deklariett eine unveränderliche ???(Webservice)
- · Klassifizierung von Interfaces

- anbieten vs ermöglichend
- und allgemein vs kontextspezifisch

- · Dependency Injection (warum, wie eingesetzt und Varianten davon, wie ausgeführt)
- Warum: Abhängigkeit von referenzierten Klassen verringern.
- Wie: Die benötigte Instanz nicht von dem abhängigen Objekt selbst, per Konstruktioraufruf, erzeugen zu lassen, sondern sie von außen in dieses hineinzubringen, eben zu injizieren. (Setter-Injection, Constructor-Injection, Interface-Injection)
- · Arten von Interfaces
  - Siehe Tabelle PDF Seite 30
- · Server/Client und Server/Item eingesetzt bei Black Box- oder White Box-Frameworks
  - Client/Server bei White-Box-Frameworks
  - Server/Client bei Black-Box-Frameworks
- · Wann kann DI nicht eingesetzt werden
  - 1) Wenn die Erstellung der Dependency-Instanz von Parametern der zu erstellenden Dependency abhängt. (Falls z.B: Parameter "uni" auf "fernuni-hagen" steht soll nicht "Universität" Objekt erstellt werden sondern "Fernuniversität". Ohne das Objekt zu erstellen, hat man aber keinen Zugriff auf diesen Parameter, der ja erst bei Erstellung angegeben wird.)
  - Oder aus dem Script: "Die Verwendung der Dependency injection kommt immer dann nicht infrage, wenn die Erzeugung der Abhängigkeit (also die Zuweisung des Objekts, zu dem die Abhängigkeit besteht, an eine Variable) von Bedingungen abhängig ist, deren Erfüllung nur durch Code in der abhängigen Klasse selbst erkannt werden kann."
  - 2) Außerdem wird es schwierig, wenn die Abhängigkeit nicht zu einem genau definierten (und von außen feststelllbaren Zeitpunkt) eingerichtet wird. Wie z.B. wenn das Client-Objekt eingerichtet wird. Der Assembler kann den Zeitpunkt nicht abpassen, um diese Abhängigkeit herzustellen.
  - 3) Auch unmöglich ist sie für temporäre Variablen, da ihre Belegung flüchtig ist. Der Assembler wüsste überhaupt nicht, wann die temporäre Abhängigkeit gebraucht wird. Das selbe gillt auch für formale Parameter (Parameter die man einer Methode gibt)
- · Alternativen zu DI
- Die Verwendung von Factories und sog. Service locators sind gebräuchliche Alternativen.

- · Arten von ermöglichenden interfaces (mit je einem Beispiel [aus Java])
- - Ertu:
  - - Server/Client
    - 2 interface IA { void run(int i); }
    - 18 // classes
    - 19 class A implements IA {
    - 20 public void run(int i) {
    - 21 // do fancy stuff with the provided i
    - 22 }
    - 23 }
    - 24
    - 25 class Stuff {
    - 26 public void f(IA a) {
    - 27 // do some stuff to provide 'i'

- 28 int i = 1;
- 29 a.run(i);
- 30 }
- 31 }
- 32
- Runnable, EventListener, Observable

- - Server/Item
  - 3 interface IB { int m(); }
  - 41 class D {
  - 42 IB h;
  - 43 C k;
  - 44
  - 45 public void f() {
  - 46 int i = k.n(h);
  - 47 // do something fancy with 'i'
  - 48 }
  - 49 }
  - Comparable, Printable

- · Umkehrung der Ausführungskontrolle und Black Box-Frameworks erklären
  - - Ertu: Frameworks, bei der die Implementierung vom Entwickler versteckt wird. Man klinkt seinen Code anhand von sogenannten 'Plug points' in das Framework ein, sie werden durch das Framework aufgerufen. (Hollywood-Prinzip: 'Don't call us, we call you') & Inversion of Control, der Aufruf findet über Komoposition und Delegation bzw. Forwarding statt.
- · Motivation für den Einsatz von Interfaces
- Trennung von Anwendungslogik zu Schnittstellen macht es möglich, die interne Logik zu ändern ohne das die tausenden Klassen (die potenziell von einer Klasse und ihrem Interface) abhängen, ihre Aufrufe bzw. den Aufruf der Methoden anpassen müssen. (Geheimnisprinzip)
- Es macht es auch möglich, dass eine Methode überhaupt nicht weiß, welche Implementierung welcher Klasse aufgerufen wird und dies in Runtime ausgetauscht werden kann. Es kann also eine Methode 'füttern' geben die ein Interface des Typs 'Tier' akzeptiert und die Objekte der Tiere z.B: Hund, Katze, Maus müssen nur das Interface Tier und ihre eigene Implementierung der Methode 'füttern' implementieren.
- Umsetzung von Polymorphismus ohne Vererbung und dynamisches Binden
- · Dependency inversion principle und Typen (Möglichkeiten, wo das Objekt herkommt)
- Module höherer Ebenen sollten nicht von Modulen niedrigerer Ebenen abhängen.
- Beiden sollten von Abstraktionen abhängen.
- Abstraktionen sollten nicht von Details abhängen.
- Details sollten von Abstraktionen abhängen.

- · Dependency injection (Arten und Grenzen davon)
- Constructor injection, Setter injection und Interface injection.
- · Andere Arten, die Dependency von Klassen zu vermeiden
  - Factory Pattern (Nutzen einer Factory-Methode), Service/Resource-Locators
- · Fragile-base-class problem
  - - Ertu: Ist in diesem Kurs nicht behandelt, würde mich wundern wenn es in einer Klausur

vorkommt. Ist aus dem Kurs 01814

- · Ist JUnit als Black Box- oder White Box-Framework einzuordnen
  - - Ertu: Whitebox-Framework

KE 2

- Abbildung -> Wie sind Vor- + Nachbedingung + Tautologien herzuleiten?
- S. 55 -> "Die für das mit der Vererbung einhergehende Subtyping notwendige Aufweichung bzw. Verschärfung ergibt sich damit in EIFFEL automatisch; im schlimmsten Fall wird die Vorbedingung logisch äquivalent zu true und die Nachbedingung äquivalent zu false." -> Wie ist das gemeint?

Ertu:
- PDF Seite 67: Verstehe die Falle nicht ganz, zweiter Absatz

- Klausurfragen aus den Protokollen:

· Wo gibt es Verified by Contract

- Verified Design by Contract (Beispiel: Javas Typsystem) ist bereits prüfbar, bevor man konkreten Code ausführt z.B: in der Kompilierphase.
- Microsoft's Initiative (Code Contracts aus .NET)

· Welche DbC Schlüsselwörter in Eiffel

- Nicht klausurrelevant
- - old
- - is
- - require
- - not
- - ensure
- - end

· was unterscheidet Eiffel von JML und was ist an JML besonders

- Eiffel + JML:
  - Schlüsselwörter zur Kennzeichnung von Vor- & Nachbedingungen
  - Möglichkeit auf den „alten“ Wert einer Variablen zurückzugreifen und auf die Ergebniswerte von Methoden
  - Die korrekte Vererbung von Vor-, Nachbedingungen und Invarianten zur Erhaltung der Substituierbarkeit sicherstellen
- JML darüber hinaus:
  - Keine Nebenwirkungen in Zusicherungen bei Funktionsaufrufen (Methodenaufrufen)
  - Spezifikation und Programme sowie möglichst auch Spezifikationssprache & Programmsprache sollten klar getrennt seint

· was sind Nachteile von DbC

- Fehlende Redundanz
- Vorbedingung versagt ihren Dienst, wenn die Methoden, auf die sie zurückgreift, fehlerhaft sind
- Zirkuläre Abhängigkeiten
- Methoden, die zur Überprüfung einer Zusicherung ausgeführt werden, können (Neben)Effekt haben

· Design by Contract-Sprachen: was müssen Spezifikationssprachen können

- Spezifikationssprache muss mindestens auf den „alten“ Wert einer Variable zurückgreifen zu können
- Zusicherungen -> deskriptiv
- Anweisungen -> Präskriptiv
- Syntax der Programmiersprache

· genauere Erläuterungen zum Ausschluss von nebeneffektvollen Methoden bei EIFFEL und JML

- ist in beiden Fällen nicht auszuschließen und nicht erlaubt
- in JML gibt es das Pure-Tag um nebeneffektferei Methoden manuell zu kennzeichnen

· Was sind DbC-Sprachen und was müssen diese mitbringen? Wie unterscheiden sich Eiffel, Java und JML darin?

- siehe oben
- Java asserts erfüllen keine der Eigenschaften

· Wofür steht JML?

- Java Model Language

· Wie kann man das Verhalten von Interfaces explizit machen?

- DbC, gerade durch JML, und Unit-Tests

· Was ist jetzt schon einsetzbar für die Prüfung von Verified Design by Contract?

- Javas Typsystem als Beispiel vDBC

· Problem bei Invarianten?

- ?

· Wer prüft das Pure Tag?

- keiner bzw. nicht der Compiler, es wird manuell gesetzt

KE 3

- Formeln, S. 97 (109)

Ertu:
- PDF Seite 108 - 111: Formeln für Fehlerlokatoren nochmals durchsprechen
    - zweite Formel: linker Teil multipliziert mit L1
    - m = Anzahl gescheiter Testfälle; a = Anzahl aller Testfälle
- PDF Seite 112: Tabelle mit falsch positiven disskutieren
    - positiv: es wurde ein Fehler nachgewiesen

Klausurfragen aus den Protokollen:

- - was sind Unit-Tests
    - Komponententests
    - Unter einem Testfall versteht man die Spezifikation einer bestimmten Eingabe und der dazu von einem Programmteil oder ganzen Programm erwarteten Ausgabe. Testfälle prüfen die Korrektheit des Programms oder eines Teils davon, indem es mit den spezifizierten Eingaben aufgerufen und sein Ergebnis mit dem erwarteten verglichen wird.

· Was tun, wenn eine Datenbank nicht verfügbar ist und diese für einen Test von Bedeutung ist

- Die Datenbank wird gemockt, das heißt für den Test durch ein Mock-Objekt ersetzt

· Was ist ein Mock Objekt und wie kann man es einer Klasse zur Verfügung stellen/Wie können Mock-Objekte erzeugt werden?

- Mock-Objekte müssen nicht die gesamte Funktionalität der Objekte der Originalklasse anbieten, sondern nur die, die für das Testen gebraucht wird. Die Funktionalität kann dabei simuliert werden.
- Mock-Objekte könenn entweder klassich über den Konstruktor bzw. via Dependency Injection (in Sprachen mit starker Typprüfung muss das Mock-Objekt zuweisungskompatibel, d. h., entweder vom gleichen oder von einem Subtyp sein) oder via Mock-Frameworks mithilfe der Möglichkeit der Metaprogrammierung realisiert werden.

KE 4

- Beispiel S. 110f. (122f.)
- Komposition, S. 112 (124) -> Implementierung zweier Interfaces (erbende Objekt + beerbtes Objekt?)
- Forwarding vs. Delegation, S. 112f. (124f.)
- Funktionsweise Observer Pattern
- Funktionsweise Visitor Pattern

Klausurfragen aus den Protokollen:

· Forwarding und Delegation

- Der Unterschied liegt in der Bindung von this: Beim Forwarding bezeichnet this das Objekt, an das weitergeleitet wurde, bei der Delegation das, das delegiert hat. Die Unterscheidung ist sprachspezifisch — die meisten objektorientierten Programmiersprachen bieten nur Forwarding.

· was sind ggf. Probleme bzgl. der Vererbung, warum würde man das mit Delegation ersetzen wollen?/ Allgemeine Nachteile von Vererbung?

- Nachteile von Vererbung:
  - - Verteilung der Deklaration auf verschiedene Klassen (unübersichtlich)
  - - Vererbung von nicht benötigten Attributen und Methoden
  - - Superklasse lässt sich nicht zur Laufzeit tauschen
  - - Austauschbarkeit der Subklassen nicht immer gegeben
  - - Keine Kontrolle über die Superklasse im schlimmsten Fall (Frameworks bzw. Libraries)
  - - Fragile base class problem (Bei Änderungen der Superklasse schießen sich alle erbenden Klassen womöglich ins Knie)
    - Wikipedia: "Die Entwickler einer „zerbrechlichen“ Basisklasse, die keine genaue und vollständige Kenntnis über die Nutzung ihrer Implementierungen haben können, sind bei einer Änderung nicht in der Lage, die negativen Konsequenzen vorauszuahnen, die sich für spezialisierende Klassen hieraus ergeben."

· Replace Inheritance with Delegation erklären (inkl. Vor und Nachteile)

- - Create a field in the subclass that refers to an instance of the superclass. Initialize this field to this.
- - Change each method defined in the subclass to use the delegate field. Compile and test after changing each method.
- - Remove the subclass declaration and replace the delegate assignment with an assignment to a new object.
- - For each superclass method used by a client, add a simple delegating method.
- - Compile and test.

· Unterschied zwischen Forwarding und Delegation erklären

- Forwarding: Eine einfache Weiterleitung. Wenn ich über eine Instanzvariable eine andere Klasse aufrufe und diese wiederrum implizit über 'this' andere aufruft, die dann in der neuen Klasse gesucht werden.
- Im Speziellen wird Forwarding verwendet, wenn bei einer Komposition Rekursion auf die ehemalige Superklasse umgesetzt werden soll
  - class Super:
    - public method1() {
      - method2(); # method1 searches for method2 in Super class, it's forwarding the call by using this implicitly: this.method2()
    - }
    - public method2() {}

- class Sub:
    - private Super _super;
    - public caller() {
        - _super.method1();
    - }

- Delegation: Wenn auf Methoden der Klasse zurückgegriffen wird, von der weitergeleitet wurde. Also im Beispiel oben:
- class Super:
    - private Sub sub_clazz;
    - Super(sub_clazz) {
        - this.sub_clazz = sub_clazz;
    - }
    - public method1() {
        - sub_clazz.subclazz_method(); # The call is delegated back to the delegator, so the delegatee (The super class) is calling methods from the delegator (the sub class)
    - }
- class Sub:
    - private Super _super;
    - public caller() {
        - _super.method1();
    - }
    - public subclazz_method() {}

· Role Object Pattern beschreiben

- Entwurfsmuster für die Realisierung von Rollen
- Es gibt dabei Subjekte (Objekte in der Rolle des Subjektes) und Rollen (Objekte in der Rolle der Rolle)
- Die Rollen bieten verschiedene Sichten auf das Subjekt und geben Anfragen an das Subjekt via Forwarding oder Delegation weiter
- Rollen können zeitweise zugewisen werden.
- Rollen sind ergänzend, aber nicht alleinig subjektdefinierend.

· was ist offene Rekursion?

- * Wenn aus einer Superklasse heraus eine Instanz von sich selbst adressiert wird (Rekursion), ist an der Stelle noch nicht klar, ob es sich um eine Instanz einer der Unterklassen handelt (dementsprechend ist die Instanz hier offen)
    - - https://stackoverflow.com/questions/6089086/what-is-open-recursion

· Composite Pattern erklären

- Instanzen einer Klasse können zwei Rollen spielem: die eines Ganzen oder eines Teils
- Eine Komponente besteht dabei aus atomaren Objekten (Rolle: Teil) und Komposita (Rollen: Teil und Ganzes)
- Eine Komponente kann auch ohne Kompositum bestehen, aber ein Kompositum nur mit einer Komponente
- Wenn man explizit die Rolle des Ganzen und des Teiles modellieren will, dann kann man für beides

verschiedene Interfacesignaturen verwenden
- Ein Beispiel für die Verwendung des Composite Patterns ist die Zusammensetzung von Testsuiten aus TestCases (Test) in älteren Versionen von JUnit

· welche Designpatterns bei jUnit mit runBare() Methode

- Template Method Pattern (siehe JUnit3 Hook-Methoden Beispiel wie 'setUp')

· Welche Entwurfsmuster sind bei JUnit in Verwendung und inwiefern sind diese dort umgesetzt?

- TestCases/TestSuites: Composite Pattern

Visitor Pattern erklären

- Es gibt Situationen, in denen in einem Teilbaum der Klassenhierarchie alle Klassen die gleichen Methoden implementieren, die Implementierungen sich aber derart voneinander unterscheiden, dass die Vererbung innerhalb der Klassenhierarchie nicht zu Zwecken der Wiederverwendung genutzt werden kann.

KE 5

Klausurfragen aus den Protokollen:

· Was sind Refactorings?

- Refactorings sind Änderungen der internen Softwarestruktur mit dem Ziel die Struktur verständlicher und einfacher änderbar zu machen.
- Damit man Refactorings anwenden kann gibt es die Bedingung, das sich beobachtbares Verhalten durch diese nicht ändern darf. Zudem muss das Programm nach dem Refactoring compilierbar sein.

· Bedingungen für die Verwendbarkeit von Refactoring Tools?

- sie benötigen in der Regel den abstrakten Syntaxbaum (engl. abstract syntax tree, AST) des Programms

· Warum sind Refactorings wichtig?

- Refactorings sind wichtig, um die Struktur, Wartbarkeit und Lesbarkeit des Programmcodes zu verbessern.

· Acht Vorbedingungen von Delegation statt Vererbung?/ Vorbedingungen zu Replace Inheritance with Delegation

- Die Superklasse darf nicht abstrakt sein.
- Alle Konstruktoren der Superklasse, die benötigt werden, um eine brauchbare Instanz zu erhalten, an die delegiert werden kann, müssen zugreifbar sein. Sie dürfen also in JAVA insbesondere nicht private oder protected deklariert sein.
- Das Programm darf keine Zuweisungen von Instanzen der zu refaktorisierenden Klasse an Variablen vom Typ der Superklasse oder dessen Supertypen enthalten, denn mit der Elimination der Vererbungsbeziehung geht auch die Aufgabe der Subtypenbeziehung einher.
- Da es sich bei dem Refctoring um Forwarding handelt, dürfen Instanzen der zu refaktorisierenden

Klasse nicht von ihrer Superklasse aus offen rekursiv, also über this, aufgerufen werden.

- In JAVA können nur Methodenaufrufe weitergeleitet werden: Greifen Klientinnen der Klasse auf von der Superklasse geerbte Felder zu, so würden diese Zugriffe nach dem Refactoring ins Leere gehen.
- Sichtbarkeitsregeln der jeweils verwendeten Programmiersprache:
- Wenn beispielsweise in JAVA eine Methode mit dem Access modifier protected geerbt wird und vererbende und erbende Klasse nicht im selben Paket sind, dann ist diese Methode nach dem Refactoring nicht mehr zugreifbar und kann auch nicht per Forwarding aufgerufen werden.
- Bestimmte Subtypbeziehungen müssen aufrechterhalten werden, selbst wenn keine sie verlangenden Zuweisungen in einem Programm vorhanden sind:
- So dürfen beispielsweise keine entsprechenden expliziten Typtests (in JAVA per instanceof oder getClass()) vorkommen und die Ableitung von sog. Unchecked exceptions wie Error und RuntimeException darf nicht durch eine von Throw-able ersetzt werden, damit der Compiler keine fehlenden Exception handler bzw. Throws-Klauseln moniert
- Wenn auf den Instanzen einer Subklasse synchronisierte Methoden aufgerufen werden, die teilweise von einer Superklasse geerbt werden, und sich diese Aufrufe nach dem Refactoring auf zwei verschiedene Instanzen verteilen, klappt die Synchronisation u. U. nicht mehr (da jetzt zwei anstelle eines Monitors herangezogen werden).

· Objekt-Schizophrenie

- Eine Objektschizophrenie liegt vor, wenn eine Instanz mehrere Identitäten hat (z.B. Kunde und Lieferant).

· Zwei Beispiele für Refactoring nennen?

- Vererbung durch Delegation ersetzen (Replace Inheritance with Delegation)
- Bedingungen durch Polymorphismus ersetzen (Replace Conditional with Polymorphism)

· Replace Conditions with Polymorphism erklären

- Pholemorphie nutzen
- Vorgehen:

1. Eine abstrakte Oberklasse und entsprechende ableitende Klassen werden erstellt.
2. Die Methoden jeweils so angepasst, dass Typunterscheidungen z.B. via Switch Case nun in die Methode der passende ableitende Klasse ausgelagert wird.
3. Je nach Typ wird die passende Methode in der passenden ableitenden Klasse via Dynamisches Binden aufgerufen.

· Zu "Bedingungen durch Polymorphie ersetzen": Warum macht dieses Refactoring Sinn?

- die Wartbarkeit und Übersichtlichkeit wird erhöht
- es ist einfacher und sicherer neue Typen zu ergänzen; der Compiler zwingt einen dazu, die abstrakte Methode der Superklasse zu impementieren

KE 6

Unterschied Reflexion und Introspection

- Relexion: Oberbegriff für Introspektion, Interzeption/Interzession und Modifikation
- Introspektion: seinen Aufbau und seine eigene Ausführung beobachten
- java.lang.reflect falsch benannt

Was ist der Unterschied zwischen einem Webplan, dem aspektorientierten Programm und zusammengesetzem Programm

- Aspektorientierte Programm: Das Programm mit Aspekten (Vor dem zusammenfügen anhand des Webplans)
- Webplan: Gibt vor wie aus Aspekten und Basisprogramm (oder Aspekten alleine) das ursprüngliche Programm zusammengesetzt werden kann.
- Zusammengesetztes Programm: Das ursprüngliche Programm

Was sind crosscutting concerns?

- - Das Gegenteil dazu sind core concerns. Also Funktionalitäten des Hauptprogramms. Erweitert werden diese mit Funktionalitäten die nicht Teile sondern im Prinzip das gesamte betreffen. Diese nennt man crosscutting concerns, z.B: wäre das Logging oder die Security solch Eines.

(Folie 229/240)
Klausurfragen aus den Protokollen:

· Was ist Metaprogrammierung

  - Unter Metaprogrammierung versteht man, wenn ein Programm (genauer: ein Metaprogramm) ein anderes Programm oder sich selbst bzw. die jeweilige Ausführung beobachtet oder gar manipuliert.
  - Metaprogrammierung ist rekursiv anwendbar.

· Was ist ein Metaprogramm

  - siehe oben

· Wofür braucht man Metaprogrammierung

  - Standardwerkzeuge (bspw. in JUnit verwendet)
  - Spracherweiterungen, wenn eine Programmiersprache eine entsprechende Abstraktion nicht hat
  - dynamisch konfigurierende Systeme (passen bspw. Schnittstellen an)
  - künstliche Intelligenz (bspw. genetische Programmierung)

· Drei Arten von Reflexion

  - Interspektion: Aufbau und Ausführung beobachten
  - Interzession: Ablauf beeinflussen
  - Modifikation: Bestandteile ändern/austauschen und ergänzen

· Wofür verwendet man aspektorientierte Programmierung

  - vorwiegend für die Sicherstellung der nichtfunktionalen Eigenschaften eines Programms
  - weitere konkrete Beispiele: Logging, Debugging, Transaktionsverwaltung, Tracing, Authentisierung, Exception Handling, Persitenz

· Was ist AOP Intercession

- AOP = Aspektorientierte Programmierung
- Interzession = Programm kann eigenen Ablauf beeinflussen
- In der Praxis: Abfangen von Methodenaufrufen und Variablenzuweisungen und Prüfung auf Zulässigkeit

· Ziele der aspektorientieren Programmierung

- Ein Programm nach mehreren Gesichtspunkten organisieren
- Dadurch sollen Programme besser strukturiert werden
- Programmteile, die nicht räumlich aggregiert sind, zusammen verwenden
- Häufig nicht funktionale Aspekte: bspw. Logging von Methodenaufrufen, -parametern und -ausgaben
- Scattering (verteilte Programmteile) und Tangling (Spaghetti-Code) vermeiden

· Aspektj (gerade wie das aufgebaut ist; zudem welche Probleme kann es geben)

- siehe einzelne Fragen unten

· Was ist AOP?

- Metaprogrammierung hat Vorteile, aber kann auch viel Schaden anrichten
- die Aspektorientierte Programmierung verwendet unkritische Teile davon
- Anforderungen an ein Programm, die viele Stellen betreffen, aber nicht räumlich aggregiert werden können, lokalisieren
- Beispiel: Logging von Methodenaufrufen, -parametern und -ausgaben

· Welche Programmierkonstrukte gibt es bei AspectJ (Aspect, Advice, Pointcut, Joinpoint)?

- Joinpoints - Punkte, an denen ein aspektorientierte Programme ansetzen könnten
- Pointcuts - eine eingeschränkte Menge an Joinpoints, wo das AOP tatsächlich ansetzt

· Drei wichtigsten Bestandteile von Aspekten

- Advice (was ist zu tun? Logik)
- Jointpoint (alle Punkte an dem potentiell ein Aspekt integriert werden kann (methodenaufrufe (vor/nachher) Variablenzugiffe (vorher/nachher)))
- Pointcut (Menge an Jointpoints, an denen der Advice aufgerufen wird)

· Weaver und Weaving Plan

- Weben/Weaving = Prozess, um Zusammenhänge der aspektorientierten Programmierung wiederherzustellen
- Weber/Weaver webt nach Webplan/weaving plan

· Sprachen für die KI-Metaprogrammierung

- Lisp und Prolog

· Wann ist ein Aspekt fertig kompiliert?

- Es kommt darauf an, da das Weaving sowohl zur Übersetzungs- als auch zur Laufzeit erfolgen kann (?).

· Nachteile der aspektorientierten Programmierung (/ was gibt es für Probleme bei aspektorientierter Programmierung)

- schränkt Lesbarkeit von Programmen für Mensch und Compiler ein
- unklarer Kontrollfluss
- Webeplan verletzt Geheimnisprinzip und muss so bei Änderungen berücksichtigt werden
- ausgelagerte Aspekte brauchen ggf. Zugriff auf den Kontext
- ggf. kann eine neue Methode „zufällig“ und unbeabsichtigt in die intensionale Spezifikation fallen
- Programmteile können sich nicht gegen einweben von Aspekten wehren
- Enge Kopplung zwischen dem Webplan und dem aspektorientierten Programmteilen

· Ruft sich ein Compiler selber auf?

- Nein, er muss aufgerufen werden
- "Ein Programm kann als Text eingelesen und in der Folge analysiert bzw. manipuliert werden. Voraussetzung ist dann allerdings, dass man aus dem Programm heraus Zugriff auf einen Compiler bzw. Interpreter hat, was nicht immer der Fall ist (es sei denn, beim Metaprogramm handelt es sich selbst um einen Compiler bzw. Interpreter). Wenn man diese erst nachbilden muss, wird die Metaprogrammierung zu einer recht beschwerlichen Angelegenheit."

KE 7

Klausurfragen aus den Protokollen:
