Software Engineering I

Kontrollfragen zum Buch v1.0

Kälin Thomas, Abteilung I
WS 06/07

VORWORT

Bei diesem Dokument handelt es sich grob gesagt um eine Zusammenfassung des Buches „UML 2 und Patterns angewendet" von Craig Larman. Es enthält zu allen wichtigen Themen Kontrollfragen für die Repetition. Als Grundlage habe ich die deutsche Version der 1.Auflage (ISBN 3-8266-1453-4) für mein Studium verwendet. Die Seitenzahlen in diesem Dokument beziehen sich also auf dieses Buch und können bei neueren Auflagen variieren.

Bis zu Kapitel 17 wurden die Fragen vom Modul-Assistenten geschrieben. Da diesem dann anscheinend die Lust vergangen ist, diesen Part zu übernehmen, habe ich mich dazu durchgerungen, für die übrigen Kapitel selber Repetitionsfragen zu schreiben.

Die Reihenfolge in diesem Dokument richtet sich nach der Reihenfolge, in welcher wir das Buch laut Musterplan durcharbeiten sollten. Das bedeutet also, dass einige Kapitel früher, andere später behandelt wurden.

Für Fehlermeldungen, Anregungen oder Lob könnt ihr mich natürlich jederzeit unter meiner HSR-Adresse (tkaelin@hsr.ch) erreichen.

Viel Erfolg bei der SE-Prüfung!
Thomas Kälin

Kapitel 1: Objektorientierte Analyse und Design 4
Kapitel 2: Iterativ, evolutionär und agil 5
Kapitel 4: Inception ist nicht die Anforderungsphase 7
Kapitel 5: Evolutionäre Anforderungen 8
Kapitel 6: Use Cases 8
Kapitel 30: Beziehungen zwischen Use Cases 10
Kapitel 7: Andere Anforderungen 10
Kapitel 8: Iteration 1 - Grundlagen 11
Kapitel 9: Domain Models 11
Kapitel 31: Verfeinerung des Domänenmodells 13
Kapitel 10: System-Sequenz Diagramme 14
Kapitel 11: Operation Contracts 14
Kapitel 24: Schnelle Aktualisierung der Analyse 15
Kapitel 29: UML-Zustandsdiagramme und -modellierung 15
Kapitel 28: UML-Aktivitätsdiagramme und –modellierung 16
Kapitel 12: Von den Anforderungen zum Entwurf – Iterativ 16
Kapitel 13: Logische Architektur und UML-Pakete 17
Kapitel 15: UML-Interaktionsdiagramme 18
Kapitel 16: UML-Klassendiagramme 18
Kapitel 17: GRASP: Objekte mit Verantwortlichkeiten entwerfen 19
Kapitel 18: Beispiele für den Objektentwurf mit GRASP 20
Kapitel 19: In Entwürfen auf Sichtbarkeit achten 21
Kapitel 20: Entwürfe in Code umsetzen 21
Kapitel 22: UML Tools und UML als Blaupause 22
Kapitel 23: Iteration 2 – Weitere Patterns 22
Kapitel 25: GRASP: Mehr Objekte mit Verantwortlichkeiten 22
Kapitel 26: GoF Design Patterns anwenden 23
Kapitel 34: Verfeinerung der logischen Architektur 24
Kapitel 35: Paketentwurf 25
Kapitel 36: Weitere Objektentwürfe mit GoF Patterns 25
Kapitel 38: UML-Verteilungs- und Komponentendiagramme 26
Kapitel 39: Architektur dokumentieren: UML & das N+1 View Model 26
Kapitel 21: Test-Driven Development und Refactoring 27

KAPITEL 1: OBJEKTORIENTIERTE ANALYSE UND DESIGN

1. Frage

Die UML ist…
*** eine OOA Methode**
*** eine Diagramm Notation**
*** ein Designwerkzeug**
*** eine Programmiersprache**
*** ein Modellierungswerkzeug**
*** ein Dokumentationswerkzeug**

„Die UML ist eine Standardnotation für die Diagrammerstellung." Somit ist die Antwort 2 (eine Diagramm Notation) korrekt. (Buch S. 40)

2. Frage

Use Cases dienen…
*** zum Design eines objektorientierten Systems**
*** dem Erfassen von Anforderungen eines Systems**
*** der Analyse eines strukturierten Systems**
*** der Beschreibung von Benutzerinteraktionen**

OOD ist eng mit einer Aktivität verbunden, die als Vorbedingung des Entwurfs aufgefasst werden kann: der Anforderungsanalyse, die häufig schriftliche Use Cases (Anwendungsfälle) umfasst.

Es ist also die Antwort 2 korrekt. (Buch S. 40 / 44)

3. Frage

Beschreiben Sie kurz in eigenen Worten, auf welche drei Arten die UML laut Larman eingesetzt werden kann.

- UML als Skizze: Informelle und unvollständige Diagramme (häufig Handskizzen auf Whiteboards). Zur Analysierung von schwierigen Teilen eines Problems.
- UML als Blaupause: detaillierte Entwurfsdiagramme für Reverse Engineering (vorhandenen Code mit UML-Diagrammen visualisieren) oder zur Codegenerierung (Basiscode wird direkt aus UML-Diagrammen generiert und vom Programmierer vervollständigt).
- UML als Programmiersprache: Ausführbarer Code wird automatisch generiert ohne vom Entwickler eingesehen zu werden. Diese Arbeiten nur in einer UML-„Programmiersprache". Diese Methode ist noch nicht ausgereift.

Mehr Informationen auf Seite 47.

4. Frage

Was beschreibt Larman in Kapitel 1 als besonders wichtige Fähigkeit eines Entwicklers objektorientierter Software?

Eine kritische Fähigkeit in der OO-Entwicklung ist die gekonnte Zuweisung von Verantwortlichkeiten an Softwareobjekte. (Buch S. 42)

5. Frage

Welche Sichtweisen auf das Konzept der Klasse in UML stellt Larman vor? Wo würden Sie diese unterschiedlichen Sichtweisen jeweils einsetzen bei Analyse, Design und Programmierung?

- Konzeptuelle (begriffliche) Perspektive: Die Diagramme werden so interpretiert, dass sie Dinge in einer Situation des interessierenden Gegenstandsbereiches beschreiben. -> Konzeptuelle Klasse

- Spezifikations-Perspektive: Die Diagramme beschreiben Softwareabstraktionen oder Komponenten mit Spezifikationen und Interfaces, aber ohne Bindung an eine bestimmte Implementierung (C++, Java, …) -> Softwareklasse
- Implementierungs-Perspektive: Die Diagramme beschreiben Softwareimplementierungen in einem bestimmten Verfahren (wie bsp. Java) -> Implementierungsklasse

Mehr Informationen auf Seiten 48 & 49.

6. Frage

Was ist der wesentliche Unterschied zwischen objektorientierter Analyse (OOA) und objektorientiertem Design (OOD)?

Analyse ist eine Untersuchung des Problems und der Anforderungen, nicht der Lösung. Wie soll es eingesetzt werden? Was sind seine Funktionen?

Bei der objektorientierten Analyse liegt die Betonung darauf, die Objekte in dem Problembereich zu finden und zu beschreiben.

Entwurf oder Design betont die konzeptuelle Lösung, die die Anforderungen erfüllt, im Gegensatz zur Implementierung der Lösung. Letztlich können Entwürfe implementiert werden und diese Implementierung drückt den tatsächlichen und vollständig realisierten Entwurf aus.

Beim objektorientierten Entwurf (Objektentwurf, Design) liegt die Betonung darauf, geeignete Softwareobjekte und ihr Zusammenwirken (collaboration) zu definieren, um die Anforderungen zu erfüllen.

Mehr Informationen auf Seite 42 & 43.

KAPITEL 2: ITERATIV, EVOLUTIONÄR UND AGIL

1. Frage

Was ist der Unterschied zwischen dem Unified Process (UP) und dem Rational Unified Process (RUP)?

Der Unified Process hat sich zu einem populären iterativen Software-Entwicklungsprozess für die Erstellung objektorientierter Systeme entwickelt.

Der RUP ist eine detaillierte Verfeinerung des UP, welcher eine grosse Akzeptanz gefunden hat. (S. 54)

2. Frage

Was heisst iterative und inkrementelle Entwicklung?

Ein Schlüsselverfahren im UP ist die iterative Entwicklung. Es handelt sich um einen Lebenszyklusansatz, bei dem die Entwicklung in eine Folge von kurzen, zeitlich begrenzten Miniprojekten zerlegt wird, die als Iterationen bezeichnet werden. Das Ergebnis jedes Miniprojekts ist ein getestetes, integriertes und ausführbares partielles System.

Da das System im Laufe der Zeit wächst, bezeichnet man diesen Ansatz auch als iterative und inkrementelle Entwicklung. (S. 55)

3. Frage

Was charakterisiert das Wasserfallmodell, auf jeden Fall so, wie es normalerweise fälschlich verstanden wird?

Bei einem Wasserfall- (oder sequenziellen) Lebenszyklus wird versucht, vor dem Programmieren alle oder die meisten Anforderungen bis ins Detail zu definieren und häufig auch einen gründlichen Entwurf zu erstellen. Ausserdem wird versucht, direkt zu Beginn „zuverlässige" Ablauf- und Zeitpläne aufzustellen. (S. 59)

4. Frage

Warum funktioniert dieses so verstandene Wasserfallmodell schlecht? Nennen Sie drei Gründe.

- Falsche Grundannahme: Spezifikationen sollen vorhersagbar und stabil sein und sollen am Beginn eines Projektes korrekt definiert werden können, sodass die spätere Änderungsrate gering ist.
- Das Feedback nach den einzelnen Zyklen (UP) fehlt beim Wasserfallmodell.
- Zeitpläne können selten eingehalten werden

Seite 59 / 60

5. Frage

Was versteht man unter Timeboxing?

Eine zeitliche Beschränkung einer Iteration. Terminüberschreitungen sind unzulässig. Wird festgestellt, dass der Endtermin nicht eingehalten werden kann, wird empfohlen den Umfang der Aufgaben zu verringern. (S. 58 / 59)

6. Frage

Wieviele Prozent der Use Cases (Anwendungsszenarien) werden typisch vor der ersten Iteration detailliert analysiert?

Beim Anforderungsworkshop vor der 1. Iteration werden nur gerade 10% der Use Cases detailliert analysiert. 90% befinden sich immer noch auf einer abstrakten Ebene. (S. 61 / 1.)

7. Frage

Nach welchen Kriterien werden diese Use Cases ausgewählt?

1) architektonisch wichtig (Kernarchitektur)
2) hohen Geschäftswert aufweisen (für das Geschäft wichtige Funktionen)
3) hohes Risiko beinhalten (500 Transaktionen parallel, hohe Last)
(S. 61 / 1.)

8. Frage

Was ist eine Phase in einem Prozess?

Eine Phase ist eine Haupteinteilung, welche aus einer oder mehreren Iterationen bestehen kann. (S. 69)

9. Frage

Welches sind die Phasen des UP und was sind ihre Ziele und Resultate?

1. Inception – ungefähre Vision, Business Case, Umfang, vage Schätzung (Machbarkeitsphase).
2. Elaboration – verfeinert Vision, iterative Implementierung der Kernarchitektur, Bearbeitung hoher Risiken, Identifikation der meisten Anforderungen und des Umfangs, realistischere Schätzungen.
3. Construction – iterative Implementierung der restlichen risikoärmeren und leichteren Elemente und Vorbereitung für das Deployment. (grösster Teil des Entwicklungszyklus!)

4. Transition – Betatests, Deployment. Am Ende der Transition Phase wird das Produkt zum produktiven Einsatz freigegeben.

(S. 69)

10. Frage

Definieren Sie den Begriff „discipline" im Kontext des UP und geben Sie einige Beispiele.

Der UP beschreibt Arbeitsaktivitäten innerhalb so genannter Diziplinen. Eine Disziplin ist ein Satz von Aktivitäten (und zugehörigen Artefakten) in einem Themenbereich.

Beispiele: Business Modeling, Anforderungen, Design, Implementierung, Test, Deployment (S. 70)

11. Frage

Definieren Sie den Begriff „artifact" im Kontext des UP und geben Sie einige Beispiele.

Im UP ist Artefakt der allgemeine Terminus für jedes Arbeitsprodukt.

Beispiele: Code, Webgrafiken, Datenbankschemas, Textdokumente, Diagramme, Modelle. (S. 70)

KAPITEL 4: INCEPTION IST NICHT DIE ANFORDERUNGSPHASE

1. Frage

Erklären Sie in eigenen Worten was während der Inception passiert?

Die Inception ist eine erste kurze Phase, um eine gemeinsame Vision für das Projekt zu formulieren und den Umfang festzulegen. Sie umfasst die Analyse von 10% der Use Cases, eine Analyse der wesentlichen nichtfunktionalen Anforderungen, die Erstellung eines Business Case und die Vorbereitung der Entwicklungsumgebung, sodass die Programmierung in der folgenden Elaboration-Phase beginnen kann.

Diese Phase sollte nur eine bis einige wenige Wochen umfassen. Es ist vergleichbar mit einer Machbarkeits-Analyse, in welcher ungenaue Schätzungen oder Pläne entstehen. (Seite 84-87)

2. Frage

Welche Dokumente müssen währen der Inception erstellt werden?

Einen Business-Case und ein Vision-Artefakt. Ausserdem sollten 10% der Use Cases ausführlich und schriftlich ausgearbeitet werden. Alle anderen Artefakte sollten als optional angesehen werden. (Seite 87, 4.4)

3. Frage

Welche Entscheidung wird während dieser Phase gefällt?

Es wird entschieden, ob das Projekt wirklich durchgeführt wird, bzw. ob es überhaupt machbar ist. (Seite 84, 4.1)

KAPITEL 5: EVOLUTIONÄRE ANFORDERUNGEN

1. Frage

Was bedeutet FURPS+?

FURPS+ ist ein Merksatz für die Qualitätsmerkmale, bzw. Qualitäts-Anforderungen von Software.

F	unctionality:	Features, Fähigkeiten, Sicherheit
U	sability:	Humanfaktoren, Hilfe, Dokumentation
R	eliability:	Misserfolgsquote, Wiederanlauffähigkeit
P	erformance:	Reaktionszeiten, Durchsatz, Genauigkeit, Verfügb.
S	upportability:	Anpassungsfähigkeit, Wartbarkeit, Konfigurierbark.

Das + in FURPS+ verweist auf zusätzliche Unterfaktoren, Beispiele:

Implementation	Ressourcenbegrenzung, Sprachen, Werkzeuge
Interface	Bedingungen durch Schnittstellen
Operations	Systemmanagement auf betrieblicher Ebene
Packaging	Verpackung, bsp: ein physischer Kasten
Legal	Lizenzierung, usw.

- Seite 93, 5.4
- Übung 03

2. Frage

Was ist der Unterschied zwischen Funktionalen und Nicht-Funktionalen Anforderungen?

Funktionale Anforderungen:
- Was muss das System können?

Nichtfunktionale Anforderungen:
- Qualitätsmerkmale
- Benutzbarkeit
- Zuverlässigkeit
- Wartbarkeit
- Leistungsanforderungen (Performance)

Anmerkung: Auch hier könnte noch Functionality (Funktionalität) aus dem FURPS+-Schema genannt werden. Einige Beispiele: Angemessenheit, Genauigkeit, Ordnungsmässigkeit, Sicherheit. Siehe dazu auch Übung 3!

3. Frage

Nennen sie zu 2 funktionale Anforderungen!

- Erfassen von Zahlungen
- Drucken von Berichten

KAPITEL 6: USE CASES

1. Frage

Kurz & Prägnant: Der Sinn von Use Cases?

Use Cases sind schriftlich fixierte Geschichten. Use Cases zu schreiben ist eine sehr verbreitete Methode, um Anforderungen zu entdecken und festzuhalten. Im Wesentlichen geht es darum, funktionale Anforderungen zu entdecken und aufzuzeichnen.

2. Frage

Welches sind die drei Use Case Formate?

- **Kurz (brief):** Knappe Zusammenfassung in einem Absatz. Wird normalerweise für den Standardablauf verwendet. Wird während der Anforderungsanalyse verwendet. Dauert nur wenige Minuten.
- **Informell (casual):** Ein informelles Absatzformat. Mehrere Absätze, die verschiedene Szenarios beschreiben. Wird auch während der Anforderungsanalyse eingesetzt.
- **Voll ausgearbeitet (fully dressed):** Alle Schritte und Varianten werden ausführlich beschrieben. Es gibt unterstützende Abschnitte.

Buch Seite 104

3. Frage

Welche Abschnitte kann ein Use Case beinhalten?

Ein voll ausgearbeiteter Use Case kann folgende Abschnitte enthalten:

- Use-Case-Name	Use Case Name
- Umfang	Scope
- Ebene	Level
- Primärakteur	Primary Actor
- Stakeholder und Interessen	Stakeholders and Interests
- Vorbedingungen	Preconditions
- Erfolgsgarantie, Nachbedingungen	Success Guarantee
- Standardablauf	Main Success Scenario
- Erweiterungen	Extensions
- Spezielle Anforderungen	Special Requirements
- Liste der Technik- / Datenvariationen	Technology and Data Variations
- Häufigkeit des Auftretens	Frequency of Occurence
- Verschiedenes	Miscellaneous

Buch Seite 115

4. Frage

Nennen Sie die zwei verschiedene Styles um einen Use Case zu schreiben?

- **Essentieller Stil:** Lassen Sie die Benutzerschnittstelle aussen vor und konzentrieren Sie sich auf die Absichten der Akteure
- **Konkreter Stil:** Bei diesem Stil sind bereits Entscheidungen über die Benutzerschnittstelle in den Use-Case-Text eingebettet. Der Text zeigt möglicherweise Screenshots des Fensters, beschreibt die Navigation, die GUI, …

Buch Seite 118 / 119.

5. Frage

Welche Eigenschaften hat der Name eines Use Cases?

Benennen Sie den Use Case ähnlich wie das Anwenderziel. Im Englischen sollte zuerst ein Verb gefolgt von einem Objekt, im Deutschen ein Objekt gefolgt von einem Verb verwendet werden. (Buch Seite 123)

6. Frage

Wer hilft alles mit Use Cases zu schreiben?

Kunde, Systemanalytiker, Endanwender, Entwickler, Softwarearchitekt Der Systemanalytiker leitet den Anforderungsworkshop. (Buch Seite 134)

KAPITEL 30: BEZIEHUNGEN ZWISCHEN USE CASES

1. Frage

Welche Möglichkeiten haben Sie zur Strukurierung von Use Cases? Geben Sie eine kurze Definition.

- **Include:** Dies ist die häufigste und wichtigste Beziehung. Häufig kommt ein partielles Verhalten in mehreren Use Cases vor. Statt diesen Text jedes Mal zu duplizieren, sollte er in einen separaten Subfunktion Use Case ausgelagert werden und an passender Stelle per Verweis eingebunden werden. Dies wird im UC normalerweise so dargestellt: Include UC Blabla. Mehr Informationen auf Seite 506-509
- **Extend:** Die Idee besteht darin, einen Erweiterungs- oder Addition Use Case zu erstellen und darin zu beschreiben, wo und unter welchen Bedingungen er das Verhalten eines Basis Use Case erweitert. In der Praxis wird die Extend-Methode hauptsächlich dann verwendet, wenn es aus irgendeinem Grund unerwünscht ist, den Base Use Case zu ändern. Mehr Informationen auf Seite 510/511

2. Frage

Welches UML-Notationselement wird für die Darstellung von «include» und «extend» verwendet?

Es wird ein gestrichelter Pfeil verwendet. (Seite 512)

3. Frage

In welche Richtung weist der Pfeil in diesem Notationselement bei «include» und «extend»?

- Include: Der Basis-Use-Case zeigt auf den eingebundenen Use Case
- Extend: Der Erweiterungs-Use-Case zeigt auf den Basis-Use-Case. Die Bedingung und der Erweiterungspunkt können bei der Linie angegebenwerden.

(Mehr Informationen auf Seite 512)

KAPITEL 7: ANDERE ANFORDERUNGEN

1. Frage

Nennen sie 4 weitere Requirements Artifakte?

- Supplementary Specification
- Glossary
- Vision
- Business Rules (Domain Rules)

2. Frage

Erklären sie den Zeck bei den oben genannten Artifakte in 1-2 Sätzen.

- **Supplementary Specification:** Erfasst andere Anforderungen, Informationen und Bedingungen, die nicht zwanglos in den UC oder dem Glossary untergebracht werde können, unter anderem auch die systemweiten URPS+-Qualitätsattribute oder Anforderungen. (Buch Seite 143).
- **Glossary:** In seiner einfachsten Form besteht das Glossary (Glossar) aus einer Liste relevanter Termini und ihren Definitionen. Man sollte so früh wie möglich mit dem Glossary beginnen, da es sich schnell zu einem nützlichen Repositorium entwickelt. (Buch Seite 151)

- **Vision**: Die Vision sollte nicht lang sein und nicht versuchen, Anforderungen im Detail festzuschreiben. Und sie sollte einige Informationen aus dem Use Case Model und der Supplementary Specification zusammenfassen. (Buch Seite 148)
- **Business Rules (Domain Rules)**: schreiben vor, wie eine Domäne oder ein Geschäft arbeiten darf. Sie sind keine Anforderungen an eine Anwendung, obwohl die Anforderungen einer Anwendung häufig durch BR beeinflusst werden. Die Geschäftspolitik, physikalische Gesetze und Regierungsgesetzt sind häufig BR. (Buch Seite 153)

KAPITEL 8: ITERATION 1 - GRUNDLAGEN

3. Frage

Beschreiben Sie die erste Iteration in eigenen Worten (2-3 Sätze)?

Es werden schwierige, risikoreiche Probleme zuerst in Angriff genommen. Es werden nur Untermengen der vollständigen Anforderungen bearbeitet. Es wird lauffähiger Code generiert, kein Wegwerf-Code!

4. Frage

Nach welchen Kriterien wird die nächste Iteration geplant?

- **Risiko**: Umfasst sowohl die technische Komplexität als auch andere Faktoren wie beispielsweise die Ungewissheit über Aufwand oder Nutzbarkeit.
- **Abdeckung**: Bedeutet, dass alle wesentliche Teile des Systems in frühen Iterationen wenigstens angefasst werden, vielleicht eine „breite und flache", Komponenten-übergreifende Implementierung.
- **Kritikalität**: Bezieht sich auf Funktionen, die für den Kunden einen hohen geschäftlichen Wert haben.

Weitere Informationen: Buch Seite 165

KAPITEL 9: DOMAIN MODELS

1. Frage

Für was ist ein DomainModel nützlich?

Ein Domänenmodell ist das wichtigste - und klassische - Modell der OO-Analyse. Wir haben ein Modell, das uns den Problembereich aus einer konzeptuellen Perspektive zeigt. Ein Domänenmodell ist eine visuelle Repräsentation der problemrelevanten Konzepte einer Domäne. (Seite 168-170)

2. Frage

Was ist der "Representational Gap"?

Dies ist eine zentrale Idee bei der OO: Verwenden Sie Namen aus dem Domänenmodell als Vorlage für die Namen von Softwareklassen in der Domänenschicht, deren Objekte die entsprechende fachbereichsspezifischen Informationen enthalten. Diese Idee trägt den Namen: „Low Representational Gap" (LRG). (Seite 174)

3. Frage

Sollte dieser "Gap" (Frage 2) klein oder gross sein? Und Wieso?

Er sollte so klein wie möglich sein, um einen geringen Unterschied zwischen unseren mentalen Modellen und den Softwaremodellen zu haben. Dabei geht es nicht nur um eine philosophische Feinheit, sondern um Zeit und Geld. Die

Repräsentation erleichtert bei Anpassungen das Verständnis des bestehenden Codes. Ausserdem wird durch die Objekttechnik ein leichter erweiterbares System erzeugt. (Seite 174)

4. Frage

Attribut vs. Konzeptionelle Klasse. Wie erkenne ich den Unterschied (Faustregel)?

Wenn wir eine konzeptuelle Klasse X im Fachbereich nicht als Zahl oder Text auffassen, so ist X wahrscheinlich eine konzeptuelle Klasse und kein Attribut. (Seite 181)

5. Frage

Wann sind Assoziationen sinnvoll?

Es ist nützlich, diejenigen Assoziationen zu ermitteln und zu zeigen, die benötigt werden, um den Informationsbedarf der aktuellen, zu entwickelnden Szenarios zu befriedigen, und die das Verständnis der Domäne fördern können. Eine Assoziation ist eine problemrelevante Beziehung zwischen Klassen. (Seite 184)

Sie sollten erwägen, die folgenden Assoziationen in einem Domänenmodell aufzunehmen (Seite 185):
- Assoziationen, für die das Wissen über die Beziehung für eine gewisse Zeitdauer festgehalten werden müssen („Need-to-Remember")
- Assoziationen, die aus der Commen Association List abgeleitet sind (Seite 189 ff)

6. Frage

Schreiben sie die Notation eines Attributes auf?

[Sichtbarkeit][Name] : [Datentyp][Multiplizität][= Standardwert][Property]

Beispiele:

- date : Date	Privates Feld „date", Typ Date
+ pi : Real = 3.14 {readOnly}	Public Feld „pi", Typ Real, Def. 3.14
middleName : [0..1]	Privates Feld „middleName", 0..1 Werte

Anmerkung:
- Ohne das Sichtbarkeitszeichen wird meist von einem privaten Feld ausgegangen.
- Attribute, die aus anderen Informationen berechnet oder abgeleitet werden können, werden mit dem Symbol „/" vor dem Attributnamen gekennzeichnet.

Weitere Informationen / Beispiele: Seite 192

7. Frage

Ist das Domain Model korrekt? Ja/Nein mit Begründung (max. 2 Sätze)

Nein, da Modelle nur Annäherungen an den Fachbereich sind, den wir verstehen wollen. Das Domänenmodell ist hauptsächlich ein Werkzeug des Verstehens und der Kommunikation innerhalb einer bestimmten Gruppe, welches inkrementell weiterentwickelt wird. (Seite 200)

KAPITEL 31: VERFEINERUNG DES DOMÄNENMODELLS

1. Frage

Was bringt das Benützen von „Superclass" und „Subclass" für Vorteile?

Eine Oberklasse und Unterklassen in einem Domänenmodell zu identifizieren, hilft uns, die Konzepte allgemeiner, verfeinert und abstrakter zu verstehen. Wir können uns damit ökonomischer ausdrücken, das Problem besser verstehen und verringern die Redundanz unserer Informationen. Dies führt später (Entwurf, Implementierung) zu besserer Software. (Seite 516)

2. Frage

Was ist mit der „100% Regel" gemeint?

100% der Definitionen der Oberklassen sollten auf die Unterklasse anwendbar sein. Die Unterklasse muss zu 100% mit folgenden Aspekten der Oberklasse übereinstimmen: den Attributen und Assoziationen. (Seite 518)

3. Frage

Für was ist die „is a" Regel?

Alle Elemente einer Unterklasse müssen Elemente der Menge ihrer Oberklasse sein. In der natürlichen Sprache kann diese Beziehung normalerweise informell getestet werden, indem man die Aussage „Unterklasse ist eine Oberklasse" bildet. (Beispiel: Hund ist ein Tier). (Seite 519)

4. Frage

Geben Sie zwei eigene Beispiele für „Superclass" und „Subclass"?

Oberklasse:	Bär	Maus
Unterklasse:	Eisbär	Optische Maus

5. Frage

Wie ist die UML Notation einer Abstrakten Klasse?

Der Klassenname wird kursiv dargestellt oder aber mit dem Schlüsselwort {abstract} markiert. (Seite 525)

6. Frage

Was ist der Unterschied zwischen einer Aggregation und einer Komposition? (Notation & Technisch)

Notation:
- Aggregation: Wird mit einer weissen Raute („leer") angezeigt
- Komposition: Wird mit einer schwarzen Raute („voll") angezeigt

Technisch:
- Aggregation: Ein Objekt referenziert intern auf andere Objekte. Diese referenzierten Objekte sind jedoch nicht an die Lebzeit des „Hauptobjektes" gebunden und können auch alleine existieren. Beispiel: Auto – Räder.
- Komposition: Ein Objekt kann nur zu einer Kompositums-Instanz gehören. Das Objekt muss ausserdem immer zu einem Kompositum gehören (keine freien Objekte). Das Kompositum ist für die Erstellung und Löschung des Objektes verantwortlich. Beispiel: Mensch – Gehirn. (Anmerkung: Das Kompositum ist die Klasse, welche intern ein Objekt der anderen Klasse besitzt.)

7. Frage

Was nützen Pakete in einem Domain Model?

Ein Domänenmodell kann leicht so gross werden, dass es wünschenswert wird, es in Pakete mit eng verwandten Konzepten zu zerlegen, um die Verständlichkeit zu verbessern und eine parallele Analysearbeit zu ermöglichen.

Ein UML-Paket wird mit einem bereiterten Ordner dargestellt. Untergeordnete Pakete können in einem Paket dargestellt werden. Der Paketname steht auf dem Reiter, wenn das Paket Elemente enthält; andernfalls steht dieser in der Mitte des Orders selbst. (Seite 535)

KAPITEL 10: SYSTEM-SEQUENZ DIAGRAMME

1. Frage

Für was ist ein SSD hilfreich?

Ein System-Sequenzdiagramm ist ein leicht und schnell zu erstellendes Artefakt, das Input- und Output-Ereignisse des zu entwerfenden Systems darstellt. Der Use-Case-Text und die darin beschriebenen Systemereignisse bilden den Input für die SSD-Erstellung. SSD bilden den Input für die Operation Contracts und – was besonders wichtig ist – für den Objektentwurf. (Seite 204)

2. Frage

Für welche Fälle sollte ein SSD gezeichnet werden?

Richtlinie: Zeichnen Sie je ein SSD für den Standardablauf eines Use Case sowie für häufig vorkommende oder komplexe alternative Szenarios. (Seite 206)

3. Frage

Auf was ist bei der Namenswahl der Events zu achten?

Systemereignisse sollten nach der abstrakten Absicht, und nicht nach dem konkreten, physischen Eingabegerät benannt werden. (Seite 208)

KAPITEL 11: OPERATION CONTRACTS

1. Frage

Welches ist der wichtigste Teil eines Kontrakts?

Die Nachbedingungen. Diese beschreiben Änderungen des Zustands von Objekten in dem Domain Model. Zu diesen Zustandsänderungen zählen: Instanzen erstellen, Assoziationen herstellen oder aufheben und Attribute ändern. (Seite 215)

2. Frage

In welcher Zeitform werden die 'Postconditions' formuliert?

Drücken Sie Nachbedingungen in der Vergangenheitsform aus. (Seite 216)

3. Frage

Wann sind Kontrakts sinnvoll und wann nicht?

Erstellen Sie einen Contract für Systemoperationen, die komplex sind und möglicherweise unterschiedliche, aber nur geringfügig voneinander abweichende Ergebnisse erzeugen oder deren Use Cases nicht klar sind. (Seite 218)

KAPITEL 24: SCHNELLE AKTUALISIERUNG DER ANALYSE

1. Frage

Wann sollte ein Update der Anforderungen gemacht werden?

Gegen Ende der Iteration in der Elaboration-Phase sollte ein Anforderungsworkshop durchgeführt werden, bei dem mehrere Anforderungen untersucht und detailliert ausgearbeitet werden. Die zuvor vollkommen analysierten Use Cases werden erneut betrachtet und wahrscheinlich anhand der Einsichten verfeinert werden.

Das Domänenmodell kann je nach Bedarf auch um die neuen Erkenntnisse erweitert werden, muss aber nicht. (Seite 424)

2. Frage

Welche Artefakte sollen angepasst werden?

Use Cases (detaillierte Ausarbeitung), System-Sequenzdiagramme (Kollaborationsbeziehungen mit anderen Systemen), Domänenmodell (Je nach Nutzen, Bedarf). Operation Contracts werden üblicherweise keine neuen erstellt. (Seite 424)

KAPITEL 29: UML-ZUSTANDSDIAGRAMME UND -MODELLIERUNG

1. Frage

Für was ist ein Zustandsdiagramm nützlich?

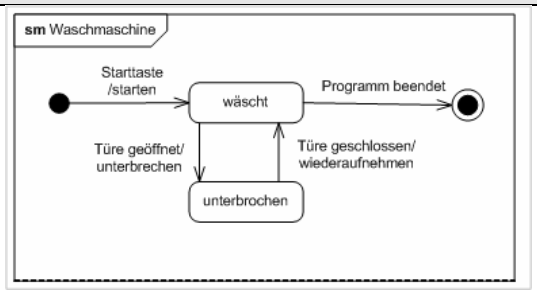
Zustandsdiagramme zeigen eine dynamische Sicht. Es zeigt die relevanten Ereignisse und Zustände eines Objektes sowie das Verhalten eines Objektes in Reaktion auf ein Ereignis. Man sollte nur für zustandsabhängige Objekte mit komplexem Verhalten ein Diagramm erstellen, nicht für zustandsunabhängige Objekte. (Seite 498 / 499)

2. Frage

Aus welchen 3 Hauptelementen besteht ein Zustandsdiagramm?

- **Ereignis:** ein wichtiges oder relevantes Vorkommnis.
- **Zustand:** Gesamtverfassung eines Objektes während einer gewissen Zeitspanne (der Zeit zwischen zwei Ereignissen).
- **Übergang:** Eine Beziehung zwischen zwei Zuständen, die anzeigt, dass das Objekt beim Eintreten eines Ereignisses von seinem jetzigen Zustand in den folgenden Zustand übergeht.

Erweiterte Informationen auf Seite 498/499.



3. Frage

Welche Information kann man in einem Zustandsdiagramm erkennen?

Wir haben eine dynamische Sicht. Wir können Veränderungen des Zustandes eines Objektes auf eintretende Ereignisse beobachten. Man könnte auch sagen, dass wir den „Lebenszyklus" eines Objektes sehen. (Seite 498)

KAPITEL 28: UML-AKTIVITÄTSDIAGRAMME UND –MODELLIERUNG

1. Frage

Welche Aspekte / Eigenschaften / Verhalten können in einem Aktivitätsdiagramm gut dargestellt werden?

Aktivitätsdiagramme zeigen sequenzielle und parallele Aktivitäten in einem Prozess. Das Diagramm kann sowohl einen Kontrollfluss als auch einen Datenfluss zeigen.

Es existieren Elemente für Aktionen, Partitionen, Splittingknoten (Fork), Synchronisationsknoten (Join) und Objektknoten.

2. Frage

Was wird häufig mit Hilfe eines Aktivitätsdiagram dargestellt?

Es wird gerne zur Visualisierung von Geschäftsabläufen, Prozessen und Use Cases verwendet. Das Verfahren ist besonders wertvoll, wenn sehr komplexe Prozesse modelliert werden sollen.

3. Frage

Erklären Sie den Unterschied zu einem Zustandsdiagram?

Bei einem ZD werden die Zustände eines Objektes grafisch dargestellt. Bei den AD hingegen wird die Zusammenarbeit zwischen verschiedenen Aktoren in einem Prozess gezeigt.

KAPITEL 12: VON DEN ANFORDERUNGEN ZUM ENTWURF – ITERATIV

1. Frage

Nennen Sie die 3 Kernaussagen dieses Kapitel. (Eigene Worte, kurz, max. 1–2 Sätze)

- **Anforderungsanalyse:** Der Fokus ist darauf gerichtet, das Richtige zu tun, also die wichtigsten Ziele und Bedingungen zu verstehen.
- **Entwurfsarbeit:** Der Schwerpunkt liegt darauf, die Dinge richtig zu tun, also eine gekonnte Lösung zu entwerfen, um die Anforderungen der Iteration zu erfüllen.
- **Frühe Änderungen provozieren:** Es wird versucht, Änderungen so früh wie möglich zu provozieren, damit in späteren Iterationen mit stabileren Anforderungen und Programmen gearbeitet werden kann.

Weitere Informationen: Seiten 226 / 227

KAPITEL 13: LOGISCHE ARCHITEKTUR UND UML-PAKETE

1. Frage

Was ist die Logische Architektur?

Die logische Architektur ist die globale Organisation der Softwareklassen in Pakete (oder Namensräume), Subsysteme und Schichten. Sie wird als logische Architektur bezeichnet, weil sie keine Entscheidungen darüber trifft, wie diese Komponenten auf verschiedene Betriebssystemprozesse oder physische Computer verteilt werden. (Seite 231)

2. Frage

Nennen Sie einige typische Schichten eines Software Systems? (Auswendig)

- User Interface
- Anwendungslogik und Domänenobjekte (Problem Domain)
- Technische Dienste (Datenhaltung)

(Andere Typische Schichten auf Seite 235)

3. Frage

Was ist der Vorteil der Schichten Architektur?

Programmierte Schichten / Klassen können wieder verwendet werden. Da die höheren Schichten nur auf tiefere Schichten, aber nicht umgekehrt, zugreifen, können diese tieferen Schichten als Kernelemente in anderen Projekten wieder verwendet werden. Andersherum können die höheren Schichten bei Anforderungswechsel schneller getauscht werden.

4. Frage

Was ist der Unterschied zwischen „Domain Model" und „Domain Layer"?

Das Domänenmodell ist eine Visualisierung der relevanten Geschäftsfälle. Es ist ein Konzept, Teil der konzeptuellen Analyse, ein Modell der realen Gegebenheiten.

Die Domänenschicht ist ein Teil der konkreten Software, also des Entwurfes. (Seite 237 / 238)

5. Frage

Was sollte im Bezug auf das Model-View Prinzip bei UI und Domain beachtet werden? (Stichwortartig)

Aus UInt1 3-13: Model-View ist die bekannteste Architektur für die Trennung von UI und Problem Domain. Sie trennt drei Bereiche:

- Bereich der PD, auch Model genannt
- Bereich des UI, unterteilt in Darstellung (View) und Behandlung (Controller)

Wir haben also zwei Schichten, wovon der UI-Bereich Aktionen entgegen nimmt. Diese Nachrichten des UI an die Domänenschicht sind die Nachrichten, die in den System-Sequenzdiagrammen dargestellt werden. (Seite 240)

KAPITEL 15: UML-INTERAKTIONSDIAGRAMME

1. Frage

Was ist der Unterschied zwischen einem Sequenz- und einem Kommunikations-Diagramm?

Sequenzdiagramme veranschaulichen Interaktionen in einer Art „Zaunformat", bei dem jedes neue Objekt jeweils auf der rechten Seite hinzugefügt werden.

Stärken: Zeigt klar die (zeitliche) Abfolge von Nachrichten
Schwächen: Zwang, Diagramme nach rechts zu erweitern

Kommunikationsdiagramme veranschaulichen Objektinteraktionen in einem Graph- oder Netzwerkformat, bei dem Objekte an einer beliebigen Stelle des Diagramms stehen können.

Stärken: Grosse Raumökonomie, Flexible Positionierung neuer Objekte
Schwächen: Abfolge von Nachrichten ist schwerer nachverfolgbar

Informationen aus Buch Seite 250-252.

2. Frage

Welcher Fehler, im Bezug auf statische und dynamische Diagramme, wird vielfach von Anfängern begangen?

Die meisten UML-Anfänger kennen Klassendiagramme und glauben normalerweise, dass dieser Diagrammtyp der einzig wichtig Typ beim OO-Entwurf ist. Falsch! Obwohl die Klassendiagramme, die eine statische Sicht zeigen, tatsächlich nützlich sind, sind die dynamischen Interaktionsdiagramme – oder genauer: die Vorgänge der Modellierung dynamischer Interaktionen – unglaublich wertvoll.

Richtlinie: Verwenden Sie Ihre Zeit auch auf die dynamische Objektmodellierung mit Interaktionsdiagrammen und nicht nur auf die statische Objektmodellierung mit Klassendiagrammen.

Informationen aus Buch Seite 253.

KAPITEL 16: UML-KLASSENDIAGRAMME

1. Frage

Welche Elemente kommen im Klassendiagramm dazu?

Im Vergleich zum konzeptionellen Modell (Domain Model) kommen folgende Elemente hinzu:
- Attribute: Sichtbarkeiten, Typen, Standardwerte, Eigenschaften
- Operationen: Beim konzeptionellen Modell werden Operationen komplett weggelassen. Sichtbarkeiten, Parameter, Rückgabewerte, Eigenschaften.

2. Frage

Was ist mit den Getters und Setters im Klass Diagram speziell?

Zugriffsoperationen (get…(), set…()) werden in Klassendiagrammen häufig weggelassen oder herausgefiltert, weil sie das Diagramm unübersichtlich machen und einen geringen Zusatznutzen bieten. (Seite 284)

3. Frage

Was hat die Aggregation für eine Funktion?

Aggregation ist in der UML eine vage Art der Assoziation, die locker (wie viele gewöhnliche Assoziationen auch) Teil-Ganzes-Beziehungen nahe legt. Rat des UML-Schöpfers: kümmern Sie sich nicht weiter um die Anwendung der Aggegation in der UML; Wenden sie stattdessen an geeigneter Stelle die Komposition an. (Seite 290)

4. Frage

Welche drei Bedingungen erfüllt eine Komposition?

Die Komposition ist eine strikte Art einer Teil-Ganzes-Aggregation. Eine Komposition impliziert folgende 3 Punkte:

- Eine Instanz des Teils gehört zu einem gegebenen Zeitpunkt genau zu einer Kompositumsinstanz.
- Das Teil gehört immer zu einem Kompositum
- Das Kompositum ist für die Erstellung und Löschung seiner Teile verantwortlich.

Gutes Beispiel hierfür: Kompositum ist die Hand, die Kompositionen sind die einzelnen Finger. Diese gehören immer genau zu einer Hand, können nicht alleine existieren und sterben, sobald die Hand stirbt.

Anmerkung: Weitere Erklärungen zur Komposition finden sich in den Fragen zum Kapitel 31.

Buch: Seite 290

KAPITEL 17: GRASP: OBJEKTE MIT VERANTWORTLICHKEITEN ENTWERFEN

1. Frage

Für was steht GRASP?

GRASP ist die Abkürzung für General Responsibility Assignment Software Patterns. (Allgemeine Softwarepatterns für die Zuweisung von Verantwortlichkeiten). (Seite 315)

2. Frage

Was ist am Namen GRASP irreführend und wieso?

Genau genommen müsste man von GRAS Patterns anstatt von GRASP Pattern sprechen. Die zweite Version klingt jedoch (laut Autor) besser. Der Name ist ein Spiel mit dem englischen Wort grasp (begreifen), im deutschen funktioniert dies leider nicht. (Seite 315)

3. Frage

Nennen Sie die 9 GRASP Patterns.

- Creator
- Information Expert
- Low Coupling
- Controller
- Hight Cohesion
- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

4. Frage

Erklären Sie folgende GRASP Patterns in eigenen Worten (1-2 Sätze):
- Creator
- Information Expert
- Low Coupling
- Controller
- High Cohesion

- **Creator:** Wer ist verantwortlich für die Erstellung eines Objektes?
- **Information Expert:** Ein allgemeines Prinzip für des OO und der Zuweisung von Verantwortlichkeiten. Weisen Sie die Verantwortlichkeiten der Klasse zu, die über die zu deren Erfüllung erforderlichen Informationen verfügt.
- **Low Coupling:** Wie kann der Einfluss von Anpassungen an Klassen minimiert werden? Weisen Sie Verantwortlichkeiten so zu, dass die Kopplung (Abhängigkeit) gering bleibt.
- **Controller:** Welches erste Objekt jenseits der UI-Schicht empfängt und koordiniert eine Systemoperation?
- **High Cohesion:** Wie kann man Objekte fokussiert, verstehbar und handhabbar halten und neben bei Low Coupling unterstützen? - Weisen Sie Verantwortlichkeiten so zu, dass der Zusammenhang (Kohäsion) hoch bleibt.

KAPITEL 18: BEISPIELE FÜR DEN OBJEKTENTWURF MIT GRASP

1. Frage

Was wird als Schlüsselpunkt des OO-Entwurf und der Codierung bezeichnet?

Die Zuweisung von Verantwortlichkeiten und der Entwurf von Kollaborationsbeziehungen sind sehr wichtige und kreative Schritte des Entwurfs, und zwar sowohl bei der Diagrammerstellung als auch bei der Codierung (Seite 342)

2. Frage

Wozu werden die Systemoperationen in den SSD verwendet?

Die Systemoperationen in den SSDs werden als Startnachrichten verwendet, die in die Controller-Objekte der Domänenschicht hineingehen. (Seite 344)

3. Frage

Wie wird Sichtbarkeit definiert?

Sichtbarkeit (engl: Visibility) ist die Fähigkeit eines Objektes, ein anderes Objekt sehen oder referenzieren zu können. (Seite 351)

4. Frage

Was sollten Sie tun, wenn es mehrere Entwurfsoptionen gibt?

Wenn es alternative Entwurfsoptionen gibt, betrachten Sie die Implikation aller Alternativ im Hinblick auf die Kohäsion und Kopplung etwas näher und beachten Sie einen möglichen zukünftigen Entwicklungsdruck auf die Alternativen. Wählen Sie eine Alternative mit einer guten Köhäsion, Kopplung und Stabilität im Hinblick auf wahrscheinliche Änderungen in der Zukunft (Seite 357)

5. Frage

Wann soll die Initialisierung (Start Up-Use Case) entworfen werden?

Ein Start Up-Use Case enthält einige initialisierende Systemoperationen, die beim Starten der Applikation ausgeführt werden. Entwerfen Sie die exakte Initialisierung zuletzt. Sie können jedoch während der Entwicklung bereits Teile davon implementieren. (Seite 363)

6. Frage

Was sagt das Command-Query Separation Principle aus?

Es handelt sich hierbei um einen Klassiker der OO-Entwurfsprinzipien für Methoden. Dieses Prinzip fordert, dass eine Methode eine und nur eine der beiden folgenden Funktionen erfüllen sollte:
- Sie sollte eine Aktion ausführen (aktualisieren, koordinieren), die häufig Nebenwirkungen wie die Änderung des Zustands von Objekten haben kann.
- Sie sollte eine Abfrage ausführen, die Daten an den Aufrufer zurückgibt und keine Nebenwirkungen hat.

(Seite 372)

KAPITEL 19: IN ENTWÜRFEN AUF SICHTBARKEIT ACHTEN

1. Frage

Welche Sichtbarkeiten existieren?

- **Attributsichtbarkeit**: Diese existiert, wenn B ein Attribut von A ist. Diese Form der Sichtbarkeit ist verhältnismässig dauerhaft, weil sie so lange besteht, wie A und B existieren. (Beispiel: Instanzvariable)
- **Parametersichtbarkeit**: Diese existiert von A nach B, wenn B als Parameter an eine Methode von A übergeben wird. Diese Sichtbarkeit ist relativ temporär, weil sie nur innerhalb des Gültigkeitsbereiches der Methode Bestand hat.
- **Lokale Sichtbarkeit**: Existiert von A nach B, wenn B als lokales Objekt innerhalb einer Methode von A deklariert wird. Diese Sichtbarkeit ist relativ temporär, weil sie nur innerhalb des Gültigkeitsbereiches der Methode Bestand hat.
- **Globale Sichtbarkeit**: Existiert von A nach B, wenn B für A global ist. Diese Sichtbarkeit ist relativ dauerhaft, weil sei bestand hat, solange A und B existieren.

KAPITEL 20: ENTWÜRFE IN CODE UMSETZEN

1. Frage

Was dient als Input für die Codeerstellung? Wie heisst das UP-Artefakt der Codeerstellung?

Die UML-Artefakte, die bei der Entwurfsarbeit erstellt wurden – die Interaktionsdiagramme und DCDs – dienen als Input für die Codeerstellung.

In der Terminologie des UP gibt ein Implementation Model. Es enthält alle Implementierungsartefakte (Quellcode, Datenbankdefinitionen). (Seite 384)

2. Frage

Welche Richtlinie betreffend Verwendung von Interfaces empfiehlt Larman?

Wenn ein Objekt ein Interface implementiert, deklarieren Sie die Variable mit Hilfe des Interfaces und nicht der konkreten Klasse. (Seite 388)

KAPITEL 22: UML TOOLS UND UML ALS BLAUPAUSE

1. Frage

Welche 3 Arten des Engineering werden unterschieden?

- Forward Engineering (Erzeugung von Code aus Diagrammen)
- Reverse Engineering (Erzeugung von Diagrammen aus Code)
- Round-Trip Engineering (Es werden beide Richtungen unterstützt)

KAPITEL 23: ITERATION 2 – WEITERE PATTERNS

1. Frage

Was sollte am Ende der Iteration 1 (Elaboration) erreicht worden sein?

- Die gesamte Software ist gründlich getestet worden. Die Idee des UP besteht darin, Qualität und Korrektheit früh, realistisch und fortlaufend zu verifizieren.
- Kunden haben sich regelmässig an der Begutachtung des partiellen Systems beteiligt, um Feedback für die Anpassung und Klärung von Anforderung zu liefern.
- Das System ist über alle Subsystem hinweg vollkommen integriert. Es hat sich in Form eines internen Baseline-Release stabilisiert.

(Seite 418)

2. Frage

Was sollte zu Beginn der Iteration 2 (Elaboration) in Angriff genommen werden?

- Ein Iterationsplanungsmeeting, um zu entscheiden, welche Arbeit in der nächsten Iteration zu erledigen ist.
- Ein weiterer zweitägiger Anforderungsworkshop findet statt, in dem mehrerer Use Cases in komplett ausgearbeiteter Form geschrieben werden. In einem gekonnt durchgeführten UP-Projekt sind die Anforderungen, die für die frühen Iterationen ausgewählt werden, nach ihrem Risiko und der Höhe des geschäftlichen Nutzens geordnet.

(Seite 418)

KAPITEL 25: GRASP: MEHR OBJEKTE MIT VERANTWORTLICHKEITEN

1. Frage

Bis jetzt haben wir fünf GRASP-Patterns angewendet: Information Expert, Creator, High Cohesion, Low Coupling und Controll. Nennen Sie die 4 noch nicht behandelten GRASP Patterns.

- Polymorphismus (Polymorphism)
- Indirektion (Indirectionm = Umlenkung)
- Pure Fabrication
- Protected Variations

(Seite 430)

2. Frage

Erklären Sie die übrigen 4 GRASP-Patterns in jeweils 2-3 Sätzen.

- **Polymorphism:** Wer ist verantwortlich, wenn das Verhalten je nach Typ variiert? Wenn verwandte Alternativen oder Verhaltensweisen je nach Klasse variieren, weisen Sie Verantwortlichkeiten für das Verhalten mit polymorphen Operationen den Typen zu, bei denen das Verhalten variiert.

Testen Sie nicht den Typ eines Objekts und verwenden Sie keine bedingte Logik, um typabhängige Alternativen auszuführen. Verwandte Patterns: Adapter, Composite, Strategy, Null-Objekt (Seite 430-436)
- **Pure Fabrication:** Wer ist verantwortlich, wenn Sie verzweifelt sind und die Prinzipien der High Cohesion und des Low Coupling nicht verletzen wollen? Weisen Sie einen hochkohäsiven Satz von Verantwortlichkeiten einer künstlichen oder geeigneten Klasse zu, die kein Konzept der PD repräsentiert, sondern willkürlich geschaffen wird, um eine hohe Köhäsion, eine Geringe Kopplung und die Wiederverwendbarkeit zu unterstützen. Verwandte Patterns: Praktisch alle Patterns sind Pure Fabrications! (Seite 436 – 440)
- **Indirection:** Wie kann man Verantwortlichkeiten zuweisen, um eine direkte Kopplung zu vermeiden? Weisen Sie die Verantwortlichkeit einem zwischengeschalteten Objekt zu, das zwischen anderen Komponenten oder Diensten vermittelt, sodass diese nicht gekoppelt sind. Verwandte Patterns: Adapter, Facade, Observer (Seite 440-441)
- **Protected Variations:** Wie weist man Verantwortlichkeiten Objekten, Subsystemen und System so zu, dass Variationen oder Instabilitäten in diesen Elementen keinen unerwünschten Einfluss auf andere Elemente haben? Identifizieren Sie die Stellen, an denen Variationen oder Instabilitäten zu erwarten sind. Weisen Sie Verantwortlichkeiten zu, um diese Stellen durch eine stabile Schnittstelle einzuschliessen. Beispiele: Datenkapslung, Interfaces, Polymorphismus, Umlenkung.(Seite 441-447)

3. Frage

Was sagt das Liskov Substituion Principle (LSP) aus?

Es ist ein Prinzip der „Protected Variations". Es sagt grob gesagt das aus, dass eine Instanz einer Unterklasse immer an Stelle seiner Oberklasse sollte verwendet werden können. (Seite 443)

4. Frage

Was bedeutet „Don't talk to Strangers"?

Dies ist ein Merksatz, mit wem Sie aus einer Methode heraus kommunizieren dürfen. Das sind das this-Objekt, ein Paramter der Methode, ein Attribut von this, ein Element einer Collection und ein lokal erstelltes Objekt. (Seite 444)

5. Frage

Erläutern Sie das Open-Closed-Prinzip.

Module sollten sowohl offen (für Erweiterungen, anpassbar) als auch geschlossen sein(das Modul ist im Hinblick auf Methodenänderungen geschlossen, die Auswirkungen auf Clients haben). (Seite 447)

KAPITEL 26: GOF DESIGN PATTERNS ANWENDEN

1. Frage

Erklären Sie in eigenen Worten das Adapter Pattern (GoF).

Adapter verwenden Interfaces und Polymorphismus um zu veränderlichen APISs in andere Komponenten eine Umlenkungsstufe hinzuzufügen. Ein Beispiel: Es existiert ein Interface mit einer Methode doSomething(). Die konkreten Implementierungen des Interfaces implementieren diese Methode. (Seite 450)

2. Frage

Erklären Sie mit eigenen Worten das Factory Pattern.

Eine Factory ist ein Pure-Fabrication-Objekt. Die Aufgabe eines Factory-Objektes ist es, andere Objekte zu erstellen. Die Klasse verfügt über Methoden, die Instanzen einer Klasse zurückgeben (Seite 454)

3. Frage

Erklären Sie das Singleton Pattern (GoF).

Manchmal ist es wünschenswert, dass von einer Klasse nur eine einzelne Instanz erstellt werden kann. Dazu wird der Konstruktor private deklariert und über eine öffentliche Methode die Instanz des Klasse abgerufen (Seite 456).

4. Frage

Erklären Sie das Strategy Pattern (GoF).

Von einem Algorithmus sollen verschiedene Varianten implementiert werden. Dazu wird ein stabiles Interface mit der variablen Methode erstellet. Die verschiedenen Varianten der Algorithmen werden in den Implementierungen des Interfaces gemacht. (Seite 460)

5. Frage

Erklären Sie das Composite Pattern (GoF).

Es soll eine Gruppe von Objekten polymorphisch auf dieselbe Weise behandelt werden, wie ein nicht zusammengesetztes Objekt. Dazu wird ein Interface definiert. Die Implementierung des Interfaces enthält intern wieder eine Referenz auf ein Objekt des Interfacestypen. (Seite 464)

6. Frage

Erklären Sie das Facade Pattern (GoF).

Es werden verschiedene Subsysteme hinter einer Klasse versteckt. Diese Klasse funktioniert als Facade und definiert einzelne Kontaktpunkte zu den Subsystemen. Eine Fassade leistet normalerweise keine Arbeit sondern konsolidiert oder vermittelt zwischen zugrunde liegenden Subsystemobjekten. (Seite 473)

KAPITEL 34: VERFEINERUNG DER LOGISCHEN ARCHITEKTUR

1. Frage

Was ist das Ziel der Elaboration-Phase?

Ein Ziel dieser Phase besteht darin, am Ende der Iterationen über einen Entwurf und eine Implementierung der Kernarchitektur zu verfügen. (Seite 569)

2. Frage

Wie wird in UML Kopplung zwischen Paketen dargestellt?

Mittels Abhängigkeitslinien. Dies sind gebogene, undurchgezogene Linien mit offener Spitze (Seite 570).

3. Frage

Wie wird die Dynamik in der logischen Architektur dargestellt?

Diese kann mit einem Interaktionsdiagramm verständlicher gemacht werden. Dabei werden bestimmte Beziehungen ignoriert, um architektonisch wichtige Interaktionen hervorzuheben. (Seite 571 – 572)

KAPITEL 35: PAKETENTWURF

1. Frage

Worin besteht das grundlegende Ziel des Paketentwurfs.

Das Ziel besteht darin, einen robusten physischen Paketentwurf zu erstellen, bei dem Pakete mit hoher Kopplung stabilisiert sind. Als Grundsatz gilt: Je weiter unten ein Paket im UML-Diagramm ist, desto stabiler sollte es sein. (Seite 588 – 589)

2. Frage

Was wird über zyklische Abhängigkeiten ausgesagt?

Diese sind unerwünscht, da durch zyklische Abhängigkeiten eine höhere Kopplung vorhanden ist. Lösen Sie die Typen, die an dem Zyklus beteilt sind, heraus und fügen Sie sie in ein neues, kleineres Paket ein. (Seite 591)

KAPITEL 36: WEITERE OBJEKTENTWÜRFE MIT GOF PATTERNS

1. Frage

Beschreiben Sie die Begriffe Lazy Initialization und Eager Initialization.

Es handelt sich hierbei um Begriffe der Objekterstellung. Die erste Variante („faul") erzeugt Objekte erst dann, wenn Sie wirklich benötigt werden. Die zweite Variante („gierig") erstellt die Objekte während des StartUp-Use Cases. (Seite 598)

2. Frage

Was sagt das „Convert Exceptions"-Pattern?

Innerhalb eines Subsystems sollten Sie es vermeiden, Ausnahmen niedrigrer Ebenen auszulösen, die aus niedrigeren Subsystemen oder Diensten stammen. Konvertieren Sie stattdessen die Ausnahme der niedrigeren Ebene in eine Ausnahme, die auf der Ebene des Subsystems bedeutsam ist (Seite 601).

3. Frage

Was ist ein Proxy (GoF)?

Ein direkter Zugriff auf ein bestimmtes Objekt ist unerwünscht oder unmöglich. Fügen Sie eine Umlenkungsebene mit einem Proxy-Objekt hinzu, das das eigentliche Objekte vertritt, dasselbe Interface implementiert und den Zugriff darauf kontrolliert oder erweitert. (Der Proxy implementiert das Interface und hat intern eine Referenz auf ein weiteres Objekt mit demselben Interface). (Seite 606)

4. Frage

Erläutern Sie das Pattern „Abstract Factory" (GoF).

Es löst die Frage, wie Klassen erstellt werden können, die ein gemeinsames Interface implementieren. Es wird ein Grund-Interface erstellt. Die anderen Factory implementieren dieses Interface. Für jede Klasse wird eine eigene

Factory von der „Abstract Factory" implementiert. Meist wird mittels des Singleton-Patterns darauf zugegriffen (Seite 610-611).

KAPITEL 38: UML-VERTEILUNGS- UND KOMPONENTENDIAGRAMME

1. Frage

Was sind Verteilungsdiagramme?

Ein Verteilungsdiagramm zeigt die Zuweisung von konkreten Softwareartefakten zu Rechnerknoten. Es zeigt die Verteilung von Softwareelementen auf die physische Architektur. (Seite 650)

2. Frage

Welche Typen von Knoten existieren?

- **Device Node:** Eine physische Einheit, die über Rechenleistung und Speicher verfügt.
- **Exceution Environment Node:** Eine Software-Ressource, die auf einem äusseren Konten läuft und selben einen Dienst zur Verfügung stellt.

3. Frage

Was ist eine Komponente?

Eine Komponente repräsentiert einen modularen Teil eines Systems, der seinen Inhalt einkapselt und dessen Manifestation innerhalb seiner Umgebung austauschbar ist. Ein Heimunterhaltungssystem ist eine gute Analogie für Komponenten: modular, selbstständig, austauschbar und arbeitet über Standardschnittstellen. (Seite 651 – 652)

KAPITEL 39: ARCHITEKTUR DOKUMENTIEREN: UML & DAS N+1 VIEW MODEL

1. Frage

Erklären Sie, was ein SAD ist.

Ein SAD (Software Architecture Document) ist ein UP Artefakt, welches die Architektur dokumentiert. Es dient dazu, in einem Team eine gemeinsame Basis für die weitere Arbeit zu schaffen oder neue Entwickler mit der Struktur des Systems vertraut zu machen (Seite 654).

2. Frage

Welches ist der Grundgedanke bei der Erstellung eines SAD?

Grundgedanke: Was muss ich sagen, bzw. als UML-Diagramm darstellen, um neuen Entwicklern zu helfen, die Grundstruktur des Systems schnell zu verstehen? (Seite 654)

3. Frage

Was sind die vier Sichten des 4+1 View Model?

Logik, Prozess, Verteilung und Daten. Das „+1" ist eine Zusammenfassung der wichtigsten Use-Cases (Seite 655).

4. Frage

Welche Sichten existieren allgemein?

- **Logik:** Struktur der Software (Schichten, Pakete, Klassen, Interfaces). Sicht auf das UP Design Model.
- **Prozess:** Prozesse und Threads.
- **Deployment:** Physische Verteilung von Prozessen und Komponenten.
- **Daten:** Überblick über die Datenflüsse, Datenhaltung.
- **Sicherheit:** Überblick über die Sicherheitsschemas.
- **Implementierung:** Quellcode, ausführbare Programme, ….
- **Entwicklung:** Daten über die Einrichtung der Entwicklungsumgebung.
- **Use Case:** Zusammenfassung der architektonisch wichtigen Use Cases und ihrer nicht funktionalen Anforderungen.

Seite 655-656

5. Frage

In welcher Phase des UP sollte das SAD erstellt werden?

Normalerweise wird erwartet, dass der grösste Teil des Inhalts des SAD am Ende der Elaboration-Phase vorliegt. Am Ende der Transition Phase wird dieser möglicherweise überprüft und überarbeitet. (Seite 668)

KAPITEL 21: TEST-DRIVEN DEVELOPMENT UND REFACTORING

1. Frage

Was wird unter „Test-Driven Development" verstanden?

Beim Unit Testing im TDD-Stil in der OO wird der Testcode vor der Klasse geschrieben, die getestet werden soll. Dies hat verschiedene Vorteile:
- Die Unit Tests werden tatsächlich geschrieben
- Die Zufriedenheit des Programmierers führt dazu, konsistentere Tests zu schreiben.

(Seite 402)

2. Frage

Was versteht man unter Refactoring?

Refactoring ist eine strukturierte, disziplinierte Methode um vorhandenen Code umzuschreiben oder neu zu strukturieren, ohne sein externes Verhalten zu ändern. Code, der gut refaktorisiert worden ist, ist kurz, knapp und enthält keine duplizierten Anweisungen.