SE&PM - Zusammenfassung

Christoph Redl

Zusammenfassung und Ausformulierung der Unterlagen zu „Software Engineering und Projektmanagement“ im Sommersemester 2007 an der TU Wien (QSE).

Die praktischen Kapitel zur Modellierung sind in dieser Zusammenfassung nicht enthalten!

Inhaltsverzeichnis

1 Design Patterns

Design Patterns sind wiederkehrende Muster beim Design von Software, die zwar so abstrakt sind dass man sie nicht in Bibliotheken packen kann, aber immerhin konkret genug um sie projektunabhängig beschreiben zu können. Patterns wurden nicht einfach definiert oder erfunden, sondern haben sich im Laufe der Zeit herauskristallisiert. Es handelt sich dabei um Umsetzungsmethoden für bestimmte Problemstellungen.

Die folgenden Abschnitte beschreiben die wichtigsten Design Patterns.

1.1 Strategy Pattern

Dieses Pattern beschreibt die klassische Implementierung von Interfaces in konkreten Klassen. Das Interface beschreibt die Schnittstellen, verschiedene Realisierungsmöglichkeiten werden in konkreten Subklassen umgesetzt. Wenn der restliche Code nur auf das Interface aufbaut, so kann die verwendete Implementierung später problemlos ausgetauscht werden ohne dass der restliche Code beeinflusst wird. Das ist zum Beispiel sinnvoll wenn ein Algorithmus aus Performancegründen reimplementiert wird.

Das entsprechende UML-Diagramm wäre ein Interface das eine oder mehrere Klassen implementieren (siehe Block 1, Folie 9).

1.2 Factory Pattern

Oft ist es nicht sinnvoll eine Klasse vor ihrer Verwendung mittels new zu instanzieren. Stattdessen wird einer anderen Klasse, der sogenannten Factory-Klasse, die Aufgabe übertragen die Instanzierung vorzunehmen.

Sinnvoll wird das dann, wenn mehrere Implementierungen eines Interfaces existieren. Die Factory-Klasse kann in diesem Fall eine create-Methode anbieten, die als Rückgabetyp das Interface hat. Zur Laufzeit kann die Factory-Klasse dann intern entscheiden welche Implementierung instanziert wird, ohne dass der Aufrufer das wissen muss.

Als Erweiterung kann die Factory-Klasse selbst in einem Interface definiert sein und mehrere Implementierungen davon existieren, die z.B. unterschiedliche Instanzierungsstrategien umsetzen.

UML-Diagramm: siehe Block 1, Folie 10.

1.3 Proxy-Pattern

Dieses Pattern wird angewandt wenn eine Klasse die Aufgabe an eine andere Klasse deligiert, sich nach außen aber so präsentiert als würde sie selbst diese Funktion anbieten. Im UML-Diagramm wird die Proxy- und die „richtige“ Klasse einfach vom gleichen Interface abgeleitet (siehe Teil 1, Folie 11). Anwendbar ist das Pattern beispielsweise dann,

wenn einer Klasse scheinbar Funktionalität hinzugefügt werden soll, man die Originalklasse aber nicht verändern will oder kann. In diesem Fall schaltet man eine Proxy-Klasse davor, die die notwendige Funktionalität ergänzt (beispielsweise Security-Checks) und dann den Aufruf an die Originalklasse weiterreicht.

1.4 Object Pool

Wenn das Instanzieren einer Klasse sehr teuer ist (etwa weil große Speichermengen allokiert werden müssen, weil Files eingelesen werden, weil eine Server-Verbindung aufgebaut werden muss, etc.), dann möchte man einen Vorrat an Instanzen führen, um dann zur Laufzeit schneller welche verfügbar zu haben.

Der Object Pool wird durch eine eigene Pool-Klasse umgesetzt, die eine get-Methode anbietet. Über diese kann eine der Instanzen, die in der Pool-Klasse vorrätig gehalten wird, retourniert werden. Die Instanz kann später auch an den Pool zurückgegeben werden und steht so weiteren Clients wieder zur Verfügung (siehe Teil 1, Folie 12).

1.5 Facade-Pattern

Wenn ein Subsystem sehr viele Klassen enthält und eine bestimmte Funktionalität das „Zusammenschalten“ vieler Klassen benötigt, so ist das unter Umständen aus Sicht des Verwenders des Subsystems relativ kompliziert. Deswegen schaltet man gerne eine Facade vor, die für die wichtigsten Use Cases einfach zu verwendende Methoden zur Verfügung stellt. Intern wird dann das komplexe Subsystem verwendet.

Dadurch verliert man Flexibilität, weil die Facade nicht alle Use Cases unterstützt (sonst wird die Facade selbst zu kompliziert). Jedoch kann man die Facade auch umgehen um sich in diesen Fällen direkt an das Subsystem zu wenden. Die Facade ist also nur eine Unterstützung um in den meisten Fällen einfach mit dem System umgehen zu können. Man kann sich aber nach wie vor direkt an das Subsystem wenden um auch die von der Facade nicht unterstützten Use Cases realisieren zu können, die aber (hoffentlich) selten auftreten.

1.6 Observer-Pattern

Wenn eine Menge von Objekten über ein Ereignis benachrichtigt werden soll, das in einem Objekt einer anderen Objektmenge auftritt (etwa Eventlistener für GUI-Interaktionen), so kann dieses Pattern angewandt werden.

Observer und Observable sind beides Interfaces. Daneben gibt es noch einen Mulitcaster, an den sich ein Observable mit einem bestimmten Ereignis wenden kann, und der dann alle registrierten Observer benachrichtigt. Die Registrierung beim Multicaster geht dabei aus Sicht des Observers auch über das Observable-Objekt, das den Registrierungsaufruf dann an den Multicaster weiterleitet (siehe Teil 1, Folie 15).

1.7 Model-View-Controller-Pattern

Dieses Pattern wird vor allem bei GUI-Interaktionen eingesetzt. Die Anzeige (View), die Geschäftslogik (Controller) und die Daten (Model) werden getrennt betrachtet. Sie tauschen Ereignisse aus (Interaktionen von View zu Controller sowie Datenänderungen von Model zu View). Daneben finden auch noch „normale“ Methodenaufrufe statt (etwa von View an Model um Daten abzufragen, vom Controller zum Model um Daten zu ändern und vom Controller zur View um die Ansicht zu ändern). Es gibt dabei normalerweise für jede Funktionalität einen eigenen Controller.

1.8 Data Access Object

Beim Datenzugriff möchte man es vermeiden sich an eine bestimmte Datenquelle zu binden. Man möchte sich die Möglichkeit offen lassen, die Datenquelle möglichst dynamisch auszuwählen.

Zu diesem Zweck erzeugt man ein DAO-Interface, das dann konkrete Klassen (zum Zugriff auf bestimmte Datenquellen, z.B. SQL, XML, Textfile mit CSV, etc.) implementieren. Diese konkreten Klassen lassen sich also über eine standardisierte Schnittstelle ansprechen, greifen aber im Backend auf unterschiedliche Quellen zu.

Die Daten selbst werden vom DAO-Object als Instanzen einer Model-Klasse geliefert (siehe Teil 1, Folie 17).

2 Tools

2.1 Maven

Maven ist ein Build- und Deployment-Tool (Open Source). Es vereinfacht das Management von Java-Projekten durch die Unterstützung der Dokumentation, des Dependency-Managements und der automatisierten Build-Erzeugung (Lifecycle).

Maven kann aus gegebenen Java-Sourcen, die mit einem validen POM (Project Object Model - ein Konfigurationsfile für ein Projekt) beschrieben sind, die Projektdateien für verschiedene Entwicklungsumgebungen (z.B. Eclipse) erzeugen. Die täglichen Coding- und Debuggingaktivitäten lassen sich dann mit der gewohnten Entwicklungsumgebung durchführen. Maven führt im Anschluss aber die Aufgaben durch, die wiederkehrend und standardisiert sind, und die man daher nicht manuell durchführen möchte. Dies sind zum Beispiel die Generierung von Code aus Modellen, Compiler-Aufrufe, Durchführung von Testfällen (Unit Tests), Optimierung, Verständigung der Teammitglieder über diverse Ereignisse (z.B. gescheiterte Testfälle), Installation, Deployment.

Der Lifecycle kann dabei konfiguriert werden, wobei an verschiedenen Stellen durchzuführende Aktivitäten eingehängt werden können.

Auch das Dependency-Management wird unterstützt. Das bedeutet dass Maven automatisch feststellen kann, wenn ein Projekt von anderen Projekten (jars) abhängt, die nicht verfügbar sind. Maven kann diese dann automatisch aus dem Internet herunterladen und installieren (sofern diese auch ein valides POM-File haben). Das funktioniert zum Beispiel wenn log4j verwendet wird, dieses aber nicht lokal (im Repository) vorhanden ist.

Maven führt im Rahmen seines Lifecycles auch Unit Tests durch und gibt die Ergebnisse in verschiedenen Formaten aus (je nach Konfiguration), z.B. als Textfile, als HTML, als XML, etc..

Wichtig ist, dass Maven dazu angestoßen werden muss diese Aufgaben durchzuführen. Verknüpft man auch noch das Source Repository mit Maven, so dass beim Check-In automatisch der Build-Prozess gestartet wird (das wird zum Beispiel von Apache Continuum unterstützt), so spricht man von Continuous Integration.

2.2 log4j

log4j ist ein Logging-Framework das zu Ausgabe von Debug-, Informations- und Fehlermeldungen verwendet werden kann. Es löst das klassische System.out.println ab. Vorteilhaft ist es deshalb, weil nun mit einem Konfigurationsfile festgelegt werden kann wo geloggt wird (Textfile, Datenbank, Nachricht an das Team, etc.). Auch können Ausgaben auf verschiedenen Ebenen (Fatal, Error, Warning, Info, Debug) erfolgen, wobei hier sehr flexibel konfiguriert werden kann. Auch wird dabei der Performancenachteil von println vermieden.

2.3 jUnit

Test-Driven-Devlopment bedeutet, dass man die Testfälle vor der Implementierung (oder spätestens parallel dazu) erstellt. Wenn man dabei Rücksicht darauf nimmt, dass die Abdeckung möglichst gut ist, dann kann man die Testfälle auch als formale Spezifikation der Anforderungen sehen. Es macht dann durchaus Sinn gezielt auf die Erfüllung der Testfälle hinzuarbeiten, wenn dann nämlich nicht nur die Testfälle funktionieren werden, sondern gleichzeitig auch die Spezifikation erfüllt wird.

Unter Unit-Tests, deren Erzeugung und Ausführung von jUnit unterstützt wird, versteht man eigene Klassen die jeweils eine (oder mehrere) andere Klassen testen. Solche Testklassen bestehen aus einer Methode zur Initialisierung (Herstellen der Vorbedingungen für diesen Testfall), einer Shutdown-Methode und einer oder mehreren Testmethoden, in denen die zu testende Klasse verwendet und das Ergebnis bestimmter Operationen überprüft wird. Wichtig ist dabei, dass die Tests so entwickelt werden, dass sie keine Folgefehler verursachen. Das heißt, ein Testfall darf nicht davon ausgehen, dass ein anderer Testfall vorher erfolgreich durchgeführt wurde, da sonst die Aussagekraft eines Testfalls geschwächt wird. jUnit bietet auch Plug-Ins für Eclipse an, so dass man

die Unit-Test bequem in der IDE durchführen und sich das Ergebnis anschauen kann.

2.4 Spring

Spring ist ein Dependency-Injection-Framework.

Unter Dependency-Injection versteht man dass die innerhalb einer Klasse verwendeten Objekte von außen erzeugt und dann in die Klasse eingebracht werden. Das ist dann sinnvoll, wenn eine Klasse intern mit einem Interfacetyp arbeitet (z.B. zum Datenzugriff - siehe DAO-Pattern). Anstatt nun innerhalb dieser Klasse selbst eine Instanz einer konkreten Implementierung anzulegen und der Interface-Variablen zuzuweisen, wird diese Instanzierung von Spring vorgenommen und dann über den Konstruktur oder eine Setter-Methode an die Klasse übergeben (Constructor- vs. Setter-Injection).

Das hat den Vorteil, dass die Business-Klasse nicht wissen muss mit welcher konkreten Implementierung sie arbeitet. Das wird in einem Konfigurationsfile für Spring festgelegt und ist somit leicht änderbar. Die Umstallung von SQL auf ein Textfile mit CSV erfordert somit (entsprechende DAO-Implementierungen vorausgesetzt) nur das Editieren eines Konfigurationsfiles.

Außerdem unterstützt Spring Aspect Orientated Programming (AOP). Während klassische Features einer Software inzwischen sehr gut in einzelne Klassen aufgeteilt und somit von verschiedenen Teammitgliedern umgesetzt werden können, ist das bei Non-Functional-Features eher schwer möglich (z.B. Security). Wenn etwa vor einem Methodenaufruf die Berechtigung überprüft werden muss, so bleibt einem normalerweise keine andere Wahl als in jeder Methode eine Berechtigungsprüfung einzufügen.

Mit Spring lassen sich nun auch solche Features separat implementieren, indem diese in eigenen Klassen ausprogrammiert werden und dann bei der Übersetzung automatisch an den entsprechenden Stellen eingefügt werden.

2.5 Internationalisierung

Anstatt Strings zur Ausgabe von Benutzermeldungen (auch z.B. die Beschriftungen von Steuerelementen) statisch in das Programm zu schreiben ist es sinnvoller, diese in eigene Files auszulagern (key/value-Paare). Eigene Bibliotheken können diese Strings dann über einen Namen ansprechen und einlesen. Dadurch wird die Übersetzung von Programmen in andere Sprachen einfacher, weil der Übersetzer, der üblicherweise kein Techniker ist, nun keinen Quelltext mehr lesen (können) muss. Abgesehen davon muss der Sourcecode nicht mehr dupliziert werden wenn Versionen in mehreren Sprachen existieren.

2.6 hsqldb

HSQLDB ist eine in Java implementierte Datenbank, die sehr einfach ansprechbar und verwendbar ist. In einem einzigen jar-File befindet sich sowohl der Treiber als auch

die Datenbank selbst. Sie ist für Test- und Debugging-Zwecke geeignet, nicht für eine Produktivumgebung. Die Daten werden oft nur im Speicher gehalten, auch wenn eine Persistierung grundsätzlich möglich ist.

2.7 Swing

Swing ist ein GUI-Framework für Java das dem MVC-Pattern folgt.

3 Persistierung

Unter Persistierung versteht man das Abspeichern von Daten. Es gibt dafür sehr unterschiedliche Möglichkeiten wie z.B. eine Textdaten, ein RDBMS, eine OO-Datenbank, etc..

Die Wahl der Persistierungsmethode hängt unter anderem ab von:

- Datenstruktur: strukturiert (z.B. RDBMS), semi-strukturiert (z.B. XML) oder unstrukturiert (z.B. natürlichsprachlicher Text)
- Art: Will man nahe am Progarmmiermodell anknüpfen (z.B. objektorientiert) oder eine eigene Methode (z.B. relational) wählen? Will man die Daten in Text- oder Binärform speichern?
- Datenmenge und Zugriffshäufigkeit
- Geforderte Verfügbarkeit: Ausfallszeiten, Online- oder Offline-Backups?
- Bereits bestehende Systeme, zu denen man kommpatibel sein muss
- Notwendige Zugriffsmethoden: sequentiell, direkter Zugriff, Abfragesprachen
- Aufwand- und Nutzen-Überlegungen
- Voraussichtliche Lebensdauer der Daten

Zusammengefasst kann man sagen dass die Art der Datenspeichern sehr stark von der Anwendung abhängt. Eine klassische Business-Applikation wird beispielsweise eine andere Persistierungsmethode wählen als ein Computerspiel oder ein Bildverarbeitungsprogramm. Es gibt keine beste Methode.

3.1 Software-Architekturen

Grundsätzlich lässt sich die Architektur einer Software in Schichten zerlegen. Je nachdem wie viele Schichten man verwendet spricht man von 2-tier-, 3-tier- oder n-tier-Architkturen. Die klassische 2-tier-Architektur verfügt nur über eine Präsentations- und eine gemeinsame Business-Logic- und Data-Access-Schicht.

Moderner ist es, auch die Business-Logic vom Datenzugriff zu trennen, wobei die Verteilung dieser Schichten auf unterschiedliche Server möglich aber nicht zwingend erforderlich ist. Es gibt hier die verschiedensten Varianten.

3.2 Relationale Datenbanken

Bei der Abbildung zwischen objektorientierten und relationalen Konzepten kann (unabhängig davon ob diese manuell oder automatisiert durchgeführt wird) grundsätzlich zwischen folgenden Varianten unterschieden werden:

- Man speichert in den Objekten der Programmiersprache Collections, in denen jeweils alle Links zu anderen Objekten gespeichert sind.
- Man speichert nur die Objekte ohne Links. Erst wenn man navigieren will greift man wieder auf die relational gespeicherten Daten zu und sucht sich den entsprechenden Link.

Die erste Variante benötigt sehr viele, zum Teil auch unnötige, Collections. Die zweite hingegen erfordert viele Datenzugriffe zur Laufzeit. Deswegen wählt man oft eine Kompromisslösung, bei der nur in solchen Fällen Collections gespeichert werden, in denen die isolierten Objekte keinen Sinn ergeben (z.B. die Zeilen einer Rechnung ohne Rechnung).

Für den Datenzugriff selbst gibt es eine Data-Access-Schicht, mit der bestimmte Modelle abgefragt werden können (siehe DAO-Pattern). Diese Schicht wird oft noch um sogenannte „Service Objects“ ergänzt, die mächtiger sind und zum Teil Use Cases abbilden, die mehrere Datenzugriffe erfordern (z.B. das Eintragen einer Rechnung samt Zeilen).

Vorteil von relationalen Datenbanken im Vergleich zu objektorientierten ist, dass diese schon weitgehend standardisiert und erprobt sind. Es gibt viele Alternativen, es werden viele Plattformen unterstützt und es gibt zahlreiche Reporting-Tools und Integrationsmöglichkeiten in andere Applikationen wie etwa Office-Pakete.

Problematisch ist aber die Abbildung einer objektorientierten Programmiersprache auf eine relationale Datenbank.

3.2.1 Manuelles Mapping

Es gibt keinerlei Beschränkungen: der Designer überlegt sich, wie die Objekte in Tabellen gespeichert werden.

3.2.2 Data Mapper

Es erfolgt zwar noch kein Mapping der gesamten OO-Struktur auf Tabellen, aber es werden zumindest die Objekte der Programmiersprache so in Werte zerlegt, dass definierte SQL-Statements damit parametrisiert werden können. Eine Bibliothek die das

unterstützt ist iBatis. Dadurch können Objekte an iBatis übergeben werden, welche dann gemäß Konfiguration abgespeichert werden. Umgekehrt können auch Daten aus einer Tabelle gelesen und daraus ein Objekt rekonstruiert werden. Die Verwendung von iBatis ist weitaus einfacher als die von O/R-Mappern wie z.B. Hibernate. Alle bekannten SQL-Features können mit iBatis (das es übrigens nicht nur für Java, sondern auf für .net, Ruby, etc. gibt) genutzt werden.

3.2.3 O/R-Mapper

Hibernate ist ein Beispiel für einen O/R-Mapper. Solche Bibliotheken sollen die objektorientierten Strukturen weitgehend automatisiert auf Tabellen mappen können, ohne dass die Applikation Details darüber wissen muss. Solche Versprechungen solle man jedoch mit Vorsicht genießen, da es in der Praxis doch problematischer ist als von den Herstellern behauptet wird.

Statt der „vollen Automatisierung“ (gemäß einer Beschreibung in Form eines XML-Files) erhält man eher eine gute Trennung zwischen OO-Welt und relationaler Welt sowie eine gute Unabhängigkeit von einem konkreten DBMS. Die Rollen können aufgeteilt werden in einen OO-Entwickler, einen Persistenzexperten und einen DB-Experten.

Besonders problematisch ist das Mapping von Vererbung, da relationale Datenbanken keine Entsprechung dafür haben. Als Alternativen kann man daher entweder alle Attribute von sämtlichen Unterklassen zusammenfassen und die nicht benötigten auslassen (verursacht viele NULL-Werte) oder für jede Klasse eine eigene Tabelle erzeugen (in diesem Fall muss man für Instanzen der Subklasse aber Einträge in mehreren Tabellen vornehmen).

3.3 Objektorientierte Datenbanken

Objektorientierte Datenbanken sind nahe am Programmiermodell und benötigen daher kein Mapping zwischen der objektorientierten und der relationalen Welt. Ein Beispiel einer solchen Datenbank ist db4o.

Nachteil ist, dass es, anders als bei relationalen Datenbanken, keinerlei Standards gibt. Zwar unterscheiden sich auch RDBMS in Details, sie sind aber grundsätzlich austauschbar. Bei objektorientierten Datenbanken verwendet jeder der (wenigen) Hersteller grundsätzlich andere Schnittstellen und Abfragesprachen. Aus diesem Grund muss man bei der Verwendung der freien Lizenz von db4o übrigend auch vorsichtig sein, weil hier leicht die GPL-Klausel des „abgeleiteten Projekts“ zum Tragen kommen kann, die einen zwingt das eigene Produkt auch unter die GPL zu stellen.

Transaktionen sind übrigend auch in objektorientierten Datenbanken möglich.

4 Projektmanagement

4.1 Definition: Projekt

Ein Projekt ist ein einmaliges Vorhaben mit einem definierbaren Anfang und einem definierbaren Ende an dem mehrere Personen beteiligt sind.

Das Tagesgeschäft eines Unternehmens gehört also explizit nicht dazu. Üblicherweise haben Projekte außerdem ein beschränktes Budget zur Verfügung.

Neben dem Tagesgeschäft und Projekten gibt es noch Maßnahmen. Das sind Tätigkeiten, die zwar ein definiertes Ziel haben, die aber unter Umständen nicht einmalig sind.

4.2 Einleitung in Projektmanagement

Projektmanagement ist deswegen notwendig weil das Herstellen komplexer Software keine triviale Aufgabe ist. Ohne systematische Herangehensweise hängt er Erfolg zu stark von Glück und den Fähigkeiten der Personen ab. Durch Projektmanagement kann der Einfluss von beiden Faktoren reduziert (aber nicht eliminiert) werden.

Wie das Projektmanagement abläuft, vor allem welches Vorgehensmodell eingesetzt wird, hängt sehr stark vom Typ des Projektes ab. Es lassen sich zum Beispiel folgende Projekttypen unterscheiden:

- Kommerzielle Software
- Embedded Software
- Wissenschaftliche Software

Diese Liste ist nur beispielhaft und nicht vollständig.

Die Anforderungen an die Projekttypen sind unterschiedlich. Während bei kommerzieller Software etwa auch die Benutzerbarkeit eine Rolle spielt, ist diese bei embedded Software oft nebensächlich, da es gar keine direkte Interkation mit dem User gibt. Wissenschaftliche Software benötigt eine hohe Rechengenauigkeit, die bei kommerzieller Software möglicherweise zum Vorteil der Performance eingespart wird.

4.3 Begriffsdefinitionen

Software/Programme bezeichnet alle Instruktionen, die dem Computer vorschreiben was er tun soll. Software Engineering ist das systematische Herstellen einer Software, ähnlich wie es auch in klassischen Ingenieursdisziplinen systematische Vorgehensweisen gibt.

Analyse ist das Ermitteln der Anforderungen an eine Software, Design ist der Entwurf der internen Struktur eines Systems.

Verfikation ist die Überprüfung ob nach einer Phase der Softwareentwicklung das Ergebnis mit vorher definierten Anforderungen übereinstimmt, d.h. ob das Ergebnis einer Phase mit den Anforderungsdokumenten einer vorhergehenden Phase kompatibel ist.

Dagegen ist die Validierung die Überprüfung, ob auch die Anforderungen aus Benutzersicht umgesetzt wurden. Verifikation überprüft also ob es richtig umgesetzt wurde, Validierung überprüft ob das Richtige umgesetzt wurden.

4.4 Capability Maturity Model - Integrated (CMMI)

Dies ist ein Modell anhand dessen die Qualität einer systematischen Vorgehensweise in einem Unternehmen überprüft werden kann. Es enthält 5 aufeinander aufbauende Stufen. Während die unterste Stufe für ein ad-hoc-Vorgehen steht, stehen die höheren Stufen für immer systematischer werdendes Vorgehen. In den USA enthält man als Unternehmen nur öffentliche Aufträge wenn man sich mindestens auf Stufe 3 befindet.

4.5 People Management

Ein wesentlicher Teil des gesamten Projektmanagements in das People Management. Menschen sind nicht wie andere Ressourcen und brauchen daher auch eine spezielle Behandlung.

4.5.1 Motivation

Und Motivation versteht man alle Gründe die das Handeln der Menschen beeinflussen. Man kann unterscheiden zwischen **intrinsischer** und **extrinsischer Motivation**.

Intrinsische Motivation erwächst aus der Arbeit selbst, während extrinsische Motivation durch von außen wirkende Maßnahmen (Belohnung oder Bestrafung) erzielt werden soll (z.B. eine Stechuhr). Besonders bei Denk- und Kreativaufgaben ist instrinsische Motivation unheimlich wichtig: hier kommt man mit extrinsischen Maßnahmen nicht weit (bei Denkaufgaben lässt sich ohnehin nicht kontrollieren ob wirklich gearbeitet wird, oder ob jemand nur physisch anwesend ist).

Motivationstheorien versuchen den Prozess der Motivation und deren Auswirkungen zu beschreiben. Sie lassen sich weiter unterteilen in **Prozesstheorien** und **Inhaltstheorien**.

Prozesstheorien befassen sich mit den Gründen, wieso jemand bestimmte Handlungen setzt um bestimmte Ziele zu erreichen. Wesentliche Aspekte dabei sind das eigene

Anspruchsniveau (die „Messlatte“, die man sich selbst setzt), die Einstellung zur Organisation (jemand, der sich mit seiner Arbeit identifiziert ist besser) und die subjektive Wahrscheinlichkeit für die Erreichung von Zielen (wenn ein Ziel von Anfang an unerreichbar erscheint, dann verliert man die Antriebskraft).

Inhaltstheorien befassen sich mit den Zielen, Antrieben und Bedürfnissen der Menschen. Es lässt sich grundsätzlich unterscheiden zwischen monothematischen und polythematischen Theorien.

Monothematische Inhaltstheorien gehen davon aus, dass ein einziges Motiv ausschlaggebend für das gesamte Verhalten eines Menschen ist, während polythematische Inhaltstheorien (gelten heute als richtig) von vielen verschiedenen Motiven ausgehen.

Die Bedürfnispyramiede von Maslow

Eine von mehreren Inhaltstheorien ist die Bedürfnispyramiede von Maslow. Sie setzt sich aus folgenden Schichten zusammen:

- Selbstverwirklichung (kreativ sein, frei handeln können, Verantwortung übernehmen)
- Anerkennung (von anderen respektiert werden)
- Soziale Bedürfnisse (einer Gruppe angehören)
- Sicherheitsbedürfnisse (Langfristige Sicherheit)
- Physiologische Grundbedürfnisse (Nahrung, Wohnung, etc.)

Jede der Ebenen (von unten nach oben gelesen) gewinnt nach diesem Modell erst an Bedeutung, wenn alle darunter liegenden Schichten erfüllt sind. Beispielsweise hat jemand, dessen physiologischen Grundbedürfnisse nicht erfüllt sind, nicht das Bedürfnis nach langfristiger Sicherheit. Das ist auch der Kritikpunkt an diesem Modell: das stimmt so nicht! (nur im Ansatz)

Man muss wissen auf welcher Stufe sich jemand befindet, nur dann kann man entsprechende Motivation für ihn sicherstellen (andernfalls würde man unter Umständen auf der falschen Ebene ansetzen).

4.5.2 Produktivität steigern

Folgende Maßnahmen sind beliebt um die Produktivität in einem Unternehmen zu steigern:

- Personel zu höherer Produktivität anhalten
- Automatisierung
- Qutlitätsmaßstäbe herabsetzen

- Vorgangsweisen standardisieren

Jedoch muss man bei solchen Maßnahmen immer darauf achten dass dadurch nicht die Arbeitszufriedenheit reduziert und so die Fluktuation erhöht wird. Hohe Fluktuationsraten führen einerseits zu hohen Kosten für die Personalabteilung und andererseits wird die Arbeitsmoral herabgesetzt weil die Mitarbeiter nur noch kurzfristig gute Entscheidungen fällen („ich bin ohnehin nicht lange da“). Mitarbeiter sind also nicht problemlos austauschbar!

Auch kann man die Anzahl der Mitarbeiter in einem Projekt nicht beliebig erhöhen um die Produktivität zu steigern, da dann der Kommunikationsaufwand zu sehr erhöht wird. Vor allem sollte man Mitarbeiter nicht nachträglich zu einem Projekt hinzuziehen um eine Deadline einhalten zu können, da dadurch die Situation noch mehr verschlimmert wird („Adding manpower to a late project makes it later“ - Brook's law).

In Experimenten konnte man feststellen, dass Hauptfaktoren für hohe Produktivität in der Softwareentwicklung die Teampartner und die Arbeitsplatzgestaltung sind. Kaum Einfluss haben hingegen das Gehalt, die Programmiersprache und die Erfahrung(!).

Wichtig ist vor allem ein ruhiger Arbeitsplatz. Bis man nach einer Unterbrechung wieder konzentriert arbeitet vergehen in etwa 15 Minuten. Der U-Faktor (Umwelt-Faktor) gibt das Verhältnis dieser konzentrierten Arbeit zur gesamten Zeit an, in der der Mitarbeiter physisch anwesend war. Dieser Faktor sollte möglichst hoch sein (was durch Ruhe am Arbeitsplatz, insbesondere die Vermeidung unnötiger Störungen gesichert werden kann). Wichtig sind bei der Arbeitsplatzgestaltung vor allem genug Platz, Fenster, die Farbe der Beleuchtung, Privatsphäre, Individualität, etc..

4.5.3 Auswahl der Mitarbeiter

Die Auswahl der Mitarbeiter erfolgt üblicherweise anhand eines Schemas, das folgendem ähnelt:

1. Jobbeschreibung erstellen
2. Anforderungen an geeignete Jobinhaber formulieren
3. Ausschreiben um Bewerbungen einzuholen (geeignetes Medium auswählen!)
4. Auswerten der Bewerbungen anhand von Portfolio (Referenzen), Interviews, Tests, Anhörungen (auch die zukünftigen Kollegen hören zu).
5. Auswahl treffen

4.5.4 Schlussfolgerung

Jeder Mensch ist anders und muss als Individuum behandelt werden um langfristigen Erfolg und Motivation sicherzustellen. Andernfalls kommt es zu mangelnder Motivation, hoher Fluktuation und schließlich wirtschaftlichen Problemen für ein Unternehmen.

5 Vorgehensmodelle

5.1 Der Software-Lifecycle

Der Software-Lifecycle beschreibt die wesentlichen Phasen in der Softwareentwicklung. Er ist selbst kein Vorgehensmodell, sondern beschreibt die Phasen auf einem höheren Abstraktionslevel. Erst die unten vorgestellen Vorgehensmodelle (diese strukturieren den Entwicklungsprozess und verbessern die Kommunikation unter den Beteiligten) implementieren diese Phasen. Sie halten sich grundsätzlich alle an den Software-Lifecycle, prägen aber die einzelnen Phasen unterschiedlich stark aus und bauen an verschiedenen Stellen Iterationen ein.

Unterschiedliche Projekte erfordern unterschiedliche Vorgehensmodelle. Aus diesem Grund gibt es auch verschiedene Vorgehensmodelle die alle ihre Vor- und Nachteile haben.

Die wesentlichen Phaen des Software-Lifecycle sind:

1. Requirements: Anfoderungen feststellen
2. Spezifikation: Formale Beschreibung des Projektes
3. Planung: Betrifft die Zeit- und Kostenplanung des Projektes
4. Design: Entwurf der inneren Struktur um die nachfolgende Implementierung zu unterstützen
5. Implementierung: Die Umsetzung in Programmcode
6. Integration und Test
7. Wartung: Verbesserungen oder Korrekturen nach der initialen Auslieferung
8. Retirement: Ende der Lebenszeit einer Software (sie wird nicht mehr benötigt)

Wesentliche Einflussfaktoren bei der Wahl des Vorgehensmodells sind die Projektgröße (Anzahl der beteiligten Personen), Dauer, die Komplexität, ob mit neuartigen Technologien gearbeitet wird, das Risiko, ob die Anforderungen fix sind, etc..

Bevor die Vorgehensmodelle im Detail vorgestellt werden soll erwähnt werden, dass nicht die Notwendigkeit besteht jedes Produkt selbst zu entwickeln. Man kann durchaus

auch zukaufen, was vor allem dann sinnvoll ist, wenn ein fertiges Produkt im wesentlichen den Anforderungen entspricht und die Abweichungen leicht korrigiert werden können. Außerdem sollte der Zukauf natürlich billiger sein als die Eigenentwicklung (was normalerweise der Fall ist).

5.2 Allgemeines zu Software-Prozessen

In den folgenden Abschnitten werden die wichtigsten Vorgehensmodelle (auch: Software-Prozesse) erklärt. Ein Software-Prozess besteht üblicherweise aus Eingabe- und Ausgabeparametern, einzelnen Schritten zur Umsetzung, beteiligten Personen, Ressourcen und Variablen mit denen der Prozess parametrisiert werden kann.

Es ist aber wichtig zu verstehen, dass keines dieser Modelle unverändert in jedem Unternehmen und bei jedem Projekt eingesetzt werden kann. Eine Anpassung an betriebliche oder projektspezifische Bedingungen ist erforderlich. Diesen Vorgang nennt man Tailoring.

Oft ist es aber so, dass ein Unternehmen bestimmte Arten von Projekten immer wieder durchführt. Es ist daher zu aufwändig, für jedes Projekt das Basismodell neu zu modifizieren und anzupassen (nicht zuletzt deshalb, weil dafür sehr gute und erfahrene Leute benötigt werden). Deswegen wird in diesem Prozess der Anpassung noch der Zwischenschritt „Customization“ zwischengeschaltet. Es wird also das Basismodell an die grundsätzlichen Rahmenbedingungen angepasst (Customization) und dieses Modell dann weiter für die Verwendung in einem konkreten Projekt vorbereitet (Tailoring).

Prozessmodelle werden oft durch sogenannte „method frameworks“ unterstützt. Das sind Pläne, die für verschiedene Zwecke (z.B. das GUI, die Datenbank, etc.) vorgeben was in den einzelnen Projektphasen für diese Zwecke getan werden soll. So kann für den Zweck „Datenbank“ in der Design-Phase z.B. die Erstellung des ER-Modells und für die Implementierungsphase das Schreiben von stored procedures vorgesehen sein. Auch auf konkrete Tools kann darin bereits verwiesen werden. Dargestellt werden kann ein method framework am besten in Matrixform, wobei eine Dimension für die Phasen und die andere für die Zwecke steht.

5.3 Wasserfallmodell

Dieses Modell implementiert die Phasen des Software-Lifecycle unverändert. Die Phasen werden einfach der Reihe nach durchlaufen, wobei ein Zurückgehen zu früheren Phasen vorgesehen ist.

Vorteile

- Jede Phase muss abgeschlossen sein bevor die nächste beginnen kann, daher klare Trennung der Phasen und Risikominimierung
- Weit verbreitet

- Leicht anwendbar
- Backtracking zum vorherigen Schritt möglich

Nachteile

- Jede Phase muss abgeschlossen sein bevor die nächste beginnen kann (ist Vor- und Nachteil)
- Fixe Anforderungen notwendig
- Fehler in frühen Phasen haben fatale Auswirkungen

Einsatzgebiet

- Projekte mit fixen Anforderungen die in ähnlicher Form schon oft durchgeführt wurden

5.4 Das V-Modell

Dieses Modell besteht aus folgenden Phasen:

1. Voruntersuchung (bestehend aus Anforderungsanalyse und Systemspezifikation)
2. Analyse (genauere Analyse der benötigten Hard- und Software)
3. Design
4. Implementierung (inklusive Feinentwurf)
5. Test
6. Integration und Überführung in den Betrieb

Vorteile

- Gute Dokumentation
- Neue Mitarbeiter können sich schnell einarbeiten
- Gute Vergleichbarkeit mit anderen Projekten
- Gute Überprüfbarkeit des Fortschritts
- Gute Aufwandsabschätzung
- 2 Zweige: Spezifikation vs. umsetzen und testen
- 3 Abstraktionsebenen: user view, architecture view, implementation view

- Fehler lassen sich früh erkennen indem an bestimmten Punkten im Modell Reviews eingebaut werden

Nachteile

- Dokumentationslastig, vernachlässigt unter Umständen das eigentliche Produkt
- Anforderungen müssen fix sein: Zurückgehen in frühere Phasen ist zwar möglich aber teuer
- Fehler in frühen Phasen sind problematisch

Einsatzgebiet

- Projekte mit fixen Anfoderungen, die in ähnlicher Weise schon oft umgesetzt wurden und die hohen Qualitätsanforderungen genügen müssen

5.5 Inkrementelles Modell

In diesem Fall werden die Phasen

1. Analyse
2. Entwurf
3. Implementierung und Integration
4. Auslieferung an den Kunden

wiederholt und verzahnt ausgeführt. Das bedeutet, während beispielsweise noch die Implementierung läuft wird schon mit dem Entwurf der nächsten Version und eventuell mit der Analyse der übernächsten Version begonnen.

Das Modell zielt darauf ab, das Basisprodukt schnell an den Kunden auszuliefern und es dann weiterzuentwickln.

Vorteile

- Basisprodukt ist schnell beim Kunden (danach stufenweise Weiterentwicklung)
- Leichtere Planung (da kurzfristiger)
- Kontinuierliche Integration
- Schnelles Feedback
- Minimiert das Risiko
- Kurze Phasen

- Sich ändernde Anforderungen sind kein Problem

Nachteile

- Verschiedene Versionen passen unter Umständen nicht zusammen wenn man nicht aufpasst

Einsatzgebiet

- Lange Entwicklungszeit
- Vage Anforderungen

5.6 Iteratives Modell

Während das inkrementelle Modell das ganze Produkt in mehreren Schritten reifen lässt und die Zwischenversionen an den Kunden übergibt, konzentriert sich das iterative Modell auf einzelne Anwendungsfälle, die der Reihe nach umgesetzt werden.

Das iterative Modell (beispielsweise im RUP umgesetzt) ist also anwendungsfallgesteuert. Konkret bedeutet das, dass die einzelnen Features (z.B. der Wichtigkeit nach geordnet) der Reihe nach umgesetzt werden. Das Produkt wird also gewissermaßen vertikal geteilt und die einzelnen Teile der Reihe nach umgesetzt, während beim inkrementellen Modell eher eine horizontale Teilung vorliegt.

Jede Iteration besteht aus den Phasen:

1. Etablierung
2. Entwurf
3. Konstruktion
4. Übergang

5.7 Extreme Programming

Dies ist ein iterativer und agiler Ansatz in der Softwareentwicklung (agil bedeutet weniger strukturiert). Analyse, Design, Implementierung und Test laufen in sehr kurzen Phasen und teilweise parallel ab.

Vorteile

- Schnelle Berücksichtigung von geänderten Anforderungen
- Dokumentation nur soviel wie notwendig
- Ofte Integration (mehrmals täglich)

- Kundenorientiert
- Prototypen in kurzen Intervallen

Nachteile

- Nur bei kleinen Projekten (bis 12 Personen) einsetzbar
- Gutes Projektmanagement notwendig
- Wiederverwendbarkeit ist schwierig
- Formale Bewertung der Ergebnise ist schwierig (z.B. bei sicherheitskritischen Anwendungen)

Einsatzgebiet

- Kleine Projekte mit vagen Anforderungen
- Zeitkritische Projekte

pair programming ist eine Programmiermethode die in XP eingesetzt werden kann. Dabei werden Zweierteams gebildet, wovon jeweils einer programmiert und der andere testet bzw. zuschaut und versucht Fehler zu finden. Die Rollen können auch hin und wieder ausgetauscht werden.

5.8 SCRUM

5.8.1 Ablaufbeschreibung

SCRUM ist ebenfalls ein agiles Modell. Die Grundbausteine sind product backlog, sprint backlog und sprints.

Im Product Backlog stehen die Anforderungen an das zu entwickelnde System. Sie werden vom Product Owner definiert und die Prioritäten festgelegt.

Umgesetzt wird das Produkt von einem selbstorganisierten Team, dem so genannten SCRUM-Team. Es setzt das Produkt in kurzen Zyklen, den sogenannten sprints, um, die zwischen 2 und 4 Wochen dauern (das ist vorher festzulegen, empfohlen sind 3 Wochen). Vor jedem Sprint erfolgt ein Sprint Planning Meeting, in dem Items aus dem Product Backlog in den Sprint Backlog übertragen werden (das Team wählt dabei anhand der Prioritäten aus, berücksichtigt aber auch, was es sich in dieser Zeit zutraut). Auch ein gemeinsames Ziel für den ganzen Sprint wird festgelegt.

Während dem Sprint gibt es tägliche SCRUM-Meetings, die nicht länger als 15 Minuten dauern sollen und jeden Tag zur gleichen Zeit stattfinden. Diese werden vom SCRUM-Master (der nicht Teil des Teams ist, sondern nur dafür sorgt dass das Team in

Ruhe arbeiten kann) geleitet. Jeder erklärt kurz was er seit gestern gemacht hat, was er bis morgen vor hat und ob es irgendwelche Probleme gibt. Sollte es Probleme geben, so werden diese auf die Impediement List gesetzt. Der SCRUM-Master bemüht sich dann gemeinsam mit dem Team um eine Lösung.

Nach jedem Sprint erfolgt ein Sprint Demo, bei dem der Fortschritt präsentiert wird (auch der Product Owner ist dabei anwesend und kann die Neuerungen, wenn das technisch möglich ist, selbst testen). Die Präsentation sollte den Fokus auf „was wurde gemacht“ setzen, nicht auf „wie wurde es gemacht“ (das interessiert den Product Owner nicht, und er wird es auch nicht verstehen).

Nach jedem Sprint gibt es auch noch eine Sprint Retrospective, bei der besprochen wird was dem Team gut gefallen hat und was man im nächsten Sprint anders machen will (z.B. einen anderen Termin für das tägliche Meeting). Anschließend erfolgt der nächste Sprint (wieder mit vorherigem Sprint Planning Meeting).

Das wiederholt sich solange, bis alle wichtigen Features umgesetzt wurden und ein Release vorbereitet werden kann.

Die Phasen sind somit:

5.8.2 Phasen

1. Pregame: Ziele für die aktuelle Version festlegen
2. Sprints: Umsetzung in mehreren Sprints, inklusive Sprint Planning Meetings, Sprint-Demos, Sprint Retrospective, Sprint im engeren Sinne, SCRUM-Meetings, etc.
3. Postgame: Test und Release vorbereiten

Vorteile

- Sich ändernde Anforderungen sind kein Problem

Nachteile

- Nur bei kleinen Projekten einsetzbar

Einsatzgebiet

- Kleine Projekte mit vagen Anforderungen

5.9 Spiralmodell

Dieses Modell besteht aus mehreren Zyklen mit jeweils 4 Phasen:

1. Planung der Ziele und Alternativen dieses Zyklus
2. Risikoabschätzung für die einzelnen Alternativen und Entscheidung treffen
3. Implementierung
4. Nächste Phase planen

Vorteile

- Risikogetrieben, minimiert daher das Risiko
- Sehr flexibel
- Ziele eines Zyklus ergeben sich aus den Ergebnissen des vorhergehenden
- Testaufwand gut einschätzbar

Nachteile

- Hoher Managementaufwand weil oft Entscheidungen getroffen werden müssen
- Es werden erfahrene Projektmanager benötigt
- Aufwandsabschätzung ist wegen der Zyklen schwierig
- Nur bei großen Projekten anwendbar

Einsatzgebiet

- Große und risikobehaftete Softwareprojekte
- Inhouse-Projekte

5.10 V-Modell XT

XT steht für „extreme tailoring“ und verdeutlicht wofür das Modell steht, nämlich für sehr gute Anpassbarkeit an konkrete Betriebe oder Projekte.

Im Unterschied zum V-Modell 97 ist kein starrer Ablauf vorgegeben, sondern es kann flexibel ein Modell aus so genannten Grundbausteinen zusammengebaut werden, die je nach Projekttyp auch fehlen können (es gibt aber auch Grundbausteine, wie etwa das Qualitätsmanagement, die in allen Projekten vorgeschrieben sind).

Der Ablauf eines Projektes ergibt sich konkret, indem aus so genannten Entscheidungspunkten (das sind definierte Punkte im Projektverlauf, an denen bestimmte (Zwischen)Produkte vorliegen müssen und an denen eine Fortschrittsentscheidung getroffen wird, z.B. „Feinentwurf abgeschlossen“ - sie entsprechen in etwa den Meilensteinen) verbindet und damit eine Projektdurchführungsstrategie zusammenstellt.

Vorgehensbausteine sind mit Entscheidungspunkten geknüpft und definieren welche Produkte durch welche Aktivitäten (Tasks) von welchen Rollen erzeugt werden. Durch die Verknüpfung mit Entscheidungspunkten ergibt sich dann auch, zu welchen Zeitpunkten im Projektverlauf diese Produkte erzeugt werden.

Das Tailoring läuft konkret in folgenden Phasen ab:

1. Auswahl eines Projekttyps
2. Auswahl von optionalen Vorgehensbausteinen
3. Auswahl der Durchführungsstrategie
4. Individuelle Anpassung

Vorteile

- Gute Anpassbarkeit
- Unterstützt aktuelle Technologien
- Risikominimierung
- Verbesserte Kommunikation zwischen den Stakeholdern
- Keine Beschränkung auf eine bestimmte Anwendungsdomäne
- Unterstützung für optionale Komponenten
- Es existieren Open Source Tools zur Unterstützung des Tailoring-Prozesses

Nachteile

- Für kleine Projekte überdimensioniert

Einsatzgebiet
Das V-Modell XT kann für verschiedene Projekttypen eingesetzt werden. Derzeit unterstützt es Hard- und Softwareprojekte aus Auftragnehmer- und Auftraggebersicht. Außerdem werden Inhouse-Projekte (Auftraggeber und -nehmer ist die gleiche Firma) und die Einführung eines Prozessmodells selbst als Projekttypen unterstützt. Es eignet sich aber eher für größere Projekte und ist für kleinere - so wie das V-Modell 97 - überdimensioniert.

Bei Projekten für öffentliche Auftraggeber ist das V-Modell XT in Deutschland inzwischen verpflichtend.

5.11 The Rational Unified Process

5.11.1 Einführung

Der Rational Unified Process ist nicht ein konkretes Vorgehensmodell, sondern eine Art Framework, das das Tailoring und die Customization gut unterstützt: man wählt einfach die Teilaspekte, die man für ein konkretes Projekt oder ein konkretes Unternehmen benötigt und blendet die anderen aus. Ursprünglich wurde der RUP von Rational Software (gehört heute zu IBM) entwickelt. Auch eine Tool-Unterstützung ist vorhanden.

Der RUP entstand urspünglich aus dem Spiralmodell. Er wurde entworfen nachdem die Hauptgründe für das Scheitern von Projekten ermittelt wurden (dazu zählen z.B. schlecht definierte Anforderungen, sich ändernde Anforderungen, mangelnde Kommunikation, fehlendes Testen, etc.). Der RUP ist selbst ähnlich wie Software definiert, nämlich über ein zugrunde liegendes UML-Modell.

RUP ist ein iteratives Vorgehensmodell.

5.11.2 Key-Charakteristika

RUP verfolgt folgende Grundsätze:

- Anpassbarkeit (Tailoring, Customization)
- Balancierung der Stakeholder-Prioritäten: Nicht nur reine Software-Anforderungen, sondern die Anforderungen aller beteiligter Akteure, z.B. auch des Mangements, werden berücksichtigt
- Collaboration: Die Verbindung aller beteiligter Akteure zum Austausch von Informationen, Testberichten, Anforderungen und dergleichen wird angestrebt
- Feedback: Durch die iterative und inkrementelle Entwicklung bekommt das Entwicklungsteam Feedback von den Stakeholdern. Reagieren auf geänderte Situationen sowie Risikomanagement wird dadurch möglich.
- Modelle und Pattern: RUP versucht es zu vermeiden, dass auf Basis der Requirements direkt Code erzeugt wird. Stattdessen wird die Verwendung von Software-Design-Patterns, 4GL-Sprachen sowie UML-Modellen favorisiert.
- Qualitätssicherung ist dem RUP immanent. Etwa die tägliche Durchführung von Unit-Tests mittels Continous Integrationïst wesentlicher Bestandteil des RUP.

5.11.3 Der RUP-Lifecycle

Der Lifecycle von RUP wird durch das Spiralmodell definiert. RUP besteht aus folgenden 4 Phasen, die jeweils aus mehreren Iterationen bestehen (können):

1. **Inception phase**
   Hier werden eine Projektbeschreibung, die Erfolgsfaktoren des Projektes sowie eine Kosten- und Risikoabschätzung erstellt. Weiters wird ein grundsätzliches Use Case-Modell und ein Basis-Projektplan festgelegt. Auch Qualitätsstandards für dieses Projekt werden hier definiert.
   Grob zusammengefasst: hier geht es um die Ziele des Projektes.

2. **Elaboration phase**
   Diese Phase entspricht in etwa der klassischen Design-Phase: Hier wird die grundsätzliche Software-Architektur festgelegt und die Problem-Domain-Analyse durchgeführt (was soll konkret umgesetzt werden? was existiert bereits?).
   Grob zusammengefasst: hier geht es um die Architektur.

3. **Construction phase**
   Dies ist die eigentliche Entwicklung und entspricht der klassischen Implementierungs-Phase. Es entsteht eine erste Version der Software, die auch nach außen released werden kann.

4. **Transition phase**
   Hier findet die Einschulung der End-User, das Beta-Testing und die abschließende Qualitätskontrolle statt.

Nach jeder Phase gibt es eine Art „Checkliste“ mit Kriterien, anhand derer der Erfolg der Phase überprüft werden kann. Sollte eine Phase nicht erfolgreich verlaufen sein, so wird sie wiederholt (=Iteration). In der Construction-Phase wird außerdem auch dann eine zusätzliche Iteration durchlaufen, wenn das Projekt sehr groß ist und man daher Use Case-orientiert entwickelt (d.h. zuerst ein Use Case, dann der nächste, usw.). Es werden dann in mehreren Iterationen jeweils einige Use Cases implementiert, sodass zwischen den Iterationen herzeigbare Zwischenversionen entstehen.

Orthogonal zu diesen Phasen gibt es 9 Disziplinen, denen die einzelnen Tasks (das sind die notwendigen Arbeitsschritte, die von bestimmten Rollen durchgeführt werden um bestimmte Produkte zu erzeugen) zugeordnet werden können.

- **Business modelling discipline**
  Hier sollen die Software Engineers ein Verständnis dafür entwickeln, wie das von ihnen erstellte System in die Organisation eingefügt und dort verwendet wird.

- **Requirements discipline**
  Ermittlung der Anforderungen an das zu entwickelnde System.

- **Analysis and design discipline**
  Tasks in dieser Kategorie dienen der Beschreibung des zu entwickelnden Systems auf einer abstrakteren Ebene als Sourcecode. Die hier entwickelnden Dokumente sind die Grundlage für die Implementierung.

- **Implementation discipline**
  Die Implementierung im engeren Sinne (inklusive Unit Tests und Integration).
- **Test discipline**
  Verifizierung des Gesamtsystems (Integrationstest).
- **Deployment discipline**
  Packaging, Installation, User-Einschulung.
- **Configuration and change management discipline**
  Management von Änderungsanforderungen
- **Project management discipline**
  Projektmanagement auf 2 Phasen: für das gesamte Projekt und Iterationsweise. Das inkludiert etwa das Risikomanagement, die Projektplanung, etc.
  Nicht inkludiert werden vom RUP allerdings Aspekte wie Personalmanagement, Budgetverwaltung, etc.
- **Environment discipline**
  Es geht hier darum dem Team alles zur Verfügung zu stellen, das sie für die Erfüllung ihrer Aufgaben benötigen, z.B. Tools.

Die letzten 3 der Disziplinen werden als „Supporting Workflows“ bezeichnet, während die ersten 6 die „Engineering Workflows“ sind.

5.11.4 Vorteile

- Es gibt Tool-Unterstützung, z.B. durch Rational Rose
- Fußt auf UML
- Verifikation bei Meilensteinen möglich
- Integriertes Change-Management

5.11.5 Nachteile

- RUP ist komplizierter als viele andere Modelle
- Dokumentationslastig
- Proprietär

5.11.6 Einsatzgebiet

Wegen dem relativ komplexen Aufbau von RUP ist er eher für größere Projekte geeignet.

6 Anforderungsanalyse

Die Feststellung der Anforderungen gehört zu den schwierigsten Aufgaben im Projektmanagement. Verfehlung der Anforderungen gehört zu den Hauptgründen für gescheiterte Projekte.

Anforderungen stehen für die Forderungen des Kunden: eine Anforderung repräsentiert ein vom Kunden gewünschtes Verhalten des Systems. Diese müssen in eine systematische Form gebraucht werden.

Requirements lassen sich wie folgt kategorisieren:

- **Functional requirements**
  Beschreiben ein gewünschtet Systemverhalten das direkt die Hauptfunktionalität betrifft. Beispiele sind die unterstützten Anwendungsfälle, das korrekte Input-Ouput-Verhältnis und unter Umständen (etwa bei sicherheitskritischen Systemen) die Security (diese kann bei anderen Projekten durchaus auch ein non-functional Requirement sein).
- **Non-functional requirements**
  Das sind alle Anforderungen die nicht unmittelbar die Funktionalität betreffen. Das System kann auch ohne diese Requirements grundsätzlich genutzt werden, sie sind aber, um einen ordentlichen Betrieb zu ermöglichen, trotzdem unverzichtbar. Beispiele sind Verfügbarkeit, Usability, Erweiterbarkeit, Portierbarkeit, etc..
- **Design constraints**
  Diese betreffen nicht unmittelbar das für den Benutzer sichtbare Ergebnis sondern richten sich an die Designentscheidungen, die im Entwicklungsprozess getroffen werden. Beispiele dafür sind die verwendeten Datenformate und die fokusierte Zielgruppe. Besonders wichtig sind solche Constraints wenn das System mit bereits vorhandenen Systemen kompatibel sein muss.
- **Process constraints**
  Diese richten sich direkt an den Entwicklungsprozess. Beispiele dafür sind Anforderungen bezüglich der Ressourcen, die zur Verfügung gestellt werden oder die Art der Dokumentation, die erzeugt werden soll.

Der Prozess der Anforderungsanalyse gliedert sich in folgende Abschnitte:

1. Elicitation: Ermittlung der Anforderungen der einzelnen Stakeholder (z.B. durch Interviews, Prototyping, Meetings, Beobachten)
2. Analyse: Die ermittelten Anforderungen verstehen und Konflikte zwischen ihnen feststellen
3. Specification: Formale Niederschrift der Anforderungen

4. Validierung: Überprüfen, ob die nun fixierten Anforderungen tatsächlich den Kundenwünschen entsprechen

Ergebnis dieses Vorgangs ist eine „Software requirements specification“.

Requirements können unterteilt werden in „core requirements“ (unbedingt notwendig, sonst macht das System keinen Sinn), „disirable requirements“ (zwar nicht unbedingt notwendig, führen aber zu großem Mehrwert) und „optionale requirements“ (nice-to-have). Diese Einteinung ist orthogonal zu den oben genannten Kategorien von Requirements.

Beim Festlegen der Anforderungen sollte man 4 Richtlinien einhalten:

- value driven requirements: Die Requirements ergeben sich anhand der zu unterstützender Geschäftsfälle
- Shared-vision driven requirements: Jeder Stakeholder (Benutzer, Auftraggeber, Management, etc.) sollte seine Anforderungen an das Produkt äußern dürfen
- Change driven requirements: zukünftige Veränderungen sollten bereits bei der Erstentwicklung berücksicht werden.
- risk driven requirements: Man soll alles als Anforderungen festlegen, das potentiell wichtig ist. Man darf nicht nichts auslassen, wenn dadurch ein Risiko entsteht.

7 Design

7.1 Definition

Unter Design versteht man die Beschreibung der internen Struktur und der Schnittstellen einer Softwarekomponente sowie das Ergebnis, das bei diesem Prozess herauskommt. Design basiert auf den definierten Requirements.

Man kann unterscheiden zwischen Architektur-Design, bei dem die Komponenten und ihre Beziehungen definiert werden (z.B. die Untergliederung in Subsysteme), und dem Detail-Design, bei dem die interne Struktur einer Komponente beschrieben wird.

7.2 4+1-View-Model

Es gibt 5 Sichtweisen auf die Architektur einer Software, jeweils aus dem Blickwinkel eines anderen Stakeholders.

7.2.1 Logical View

Beschreibt die Architektur aus Sicht des Users. Hier werden funktionale Requirements beleutet, d.h. was die Software eigentlich macht.

7.2.2 Implementation View

Ist die Sicht der Programmierer und konzentriert sich auf die statische Untergliederung der Software in Fragmente (Klassen, Module, Konfigurationsfiles, Ressourcen etc.).

7.2.3 Process View

Dies ist die Sichtweise der Systemintegratoren. Während sich die Programmierer mit der Entwicklung eines Moduls beschäftigen geht es hier um die Zusammensetzung mehrere Module. Deswegen werden in dieser Sicht auch die Aspekte beleuchtet, die dabei besonders wichtig sind. Das ist vor allem die Gleichzeitigkeit, also Prozesse und Threads.

7.2.4 Deployment View

Die Sichtweise der System Engineers konzentriert sich auf die Installation von Software und beachtet daher besonders die Verbindung einer Software mit der darunter liegenden Plattform.

7.2.5 Use Case View

Diese Sichtweise ist anfangs für die Designer und später für die Tester wichtig. Sie beschreibt grundlegende Use Cases, die zum Design der Software und später zum Validieren wichtig sind.

7.3 Prinzipien des Designs

Dieser Abschnitt soll einige grundlegende Design-Prinzipien beschreiben, also Techniken, die man beim Entwurf von Software beachten sollte.

7.3.1 Coupling vs. Cohesion

Coupling meint die Zusammenschaltung mehrere Module um Funktionalität zur Verfügung zu stellen. Dagegen ist Cohesion ist Trennung von Komponenten so weit wie möglich.

Beides ist gefährlich wenn man es übertreibt: zu starkes Coupling vermindert Wiederverwendbarkeit, zu starke Cohesion führt hingegen zum Überladen einzelner Module, weil zu viel Funktionalität in ein Module gepackt wird. Es sollte daher ein Mittelweg zwischen beiden angestrebt werden.

7.3.2 Stair vs. Fork

Als „Stair“ bezeichnet man eine Interaktion zwischen Komponenten dann, wenn alle Interaktionspartner Aufgaben an andere Partner deligieren, und nicht bloß einer.

Das führt zu erhöhter Wiederverwendbarkeit von funktionalen Modulen (Code), nicht aber von Daten. Vererbung ist hier problemlos einsetzbar.

Als „Fork“ bezeichnet man eine Interaktion zwischen Komponenten dann, wenn ein Interaktionspartner die ganze Interaktion steuert. Abgesehen vom steuernden Interaktionspartner finden keine Delegationen statt.

Vorteil ist, dass Datenmodule leicht wiederverwendet werden können (nicht aber die Geschäftslogik). Die Wartung wird vereinfacht, da nur ein zentraler Interaktionspartner existiert, der das ganze Geschehen steuert.

7.3.3 Abstraktion

Dieser Begriff meint die Beschreibung auf Klassen- statt auf Objektebene.

7.3.4 Dekomposition und Modularisierung

Zusammengehörige Elemente sollten gemeinsam in ein Modul mit klaren Schnittstellen gepackt werden.

7.3.5 Encapsulation

Dies ist das typische objektorientierte Design-Paradigma, dass man Methoden und die zugehörigen Daten gemeinsam in eine Klasse packen soll.

7.3.6 Interfaces vs. Implementierung

Die Verwendung von Interfaces sollte gemäß Strategy-Design-Pattern vorangetrieben werden.

8 Implementierung

Die Implementierung folgt der Analyse und dem Design. Es geht hier um die tatsächliche Umsetzung eines Produktes in Quellcode. Auch dabei sollten einige Richtlinien eingehalten werden, die einen verständlicheren Code gewährleisten.

All die unten genannten Strategien können durch interne oder externe (z.B. von INGOs) Standards vorgegeben werden.

8.1 Objektorientierung

Objektorientierung ist gut, muss aber auch nicht immer eingesetzt werden. Es kommt hier stark auf das zu entwickelnde System an: manchmal kann durchaus prozedurale Programmierung besser geeignet sein. Objektorientierung ist übrigens ein Paradigma und nicht nur eine Eigenschaft einer Programmiersprache: man kann auch objektorientiert

programmieren, wenn das von der Sprache nicht unterstützt wird.

Objektorientierung führt zu engerer Kopplung zwischen Code und Daten als das bei prozeduraler Programmierung der Fall ist.
Ziele der Objektorientierung:

- Ein zentrales Konzept vom Design bis zur Implementierung
- Bessere Abbildung der „real world objects“ mit all ihren Eigenschaften
- Flexibilität: Gleichzeitige Entwicklung wird unterstützt („jeder arbeitet an einer anderen Klasse“)
- Wiederverwendbarkeit wird erhöht
- Durch die Encapsulation wird die Wartbarkeit erhöht

8.2 Namenskonventionen

Wenn sich alle an solche Konventionen halten (z.B. Klassen beginnen mit Großbuchstaben, Methoden mit Kleinbuchstaben), dann wird die Verständlichkeit des Codes für alle erhöht.

8.3 Formatierung des Quelltextes

Abstände zwischen Methoden, Einrückungen, ähnliche Strukturierung der Klassen erhöht die Lesbarkeit und Verständlichkeit.

Die Formatierung des Quellcodes sollte soweit wie möglich auch seine Ablaufstruktur wiedergeben. Auch sollten „Programmier-Tricks“ nicht in übertriebenem Ausmaß angewandt werden (auch nicht zum Zwecke der Performance) wenn dadurch die Lesbarkeit gefährdet wird (vieles optimiert ohnehin der Compiler).

8.4 Versionsverwaltung

Dadurch können Änderungen schnell nachvollzogen werden. Das kann auch mit Bug-Tracking-Systemen (Systeme, in denen alle Stakeholder gefundene Bugs eintragen können) kombiniert werden, sodass man nachvollziehen kann wieso eine bestimmte Änderung notwendig war.

8.5 Kommentare

Sollten aussagekräftig sein und nicht bloß das wiederholen, das ohnehin schon aus dem Quelltext ersichtlich ist.

8.6 Headerblocks

Headerblocks beschreiben Methoden samt Parameter und Return-Wert. Weitere Teile sind: der Autor, wie die Methode in das Gesamtsystem hineinspielt, wieso sie überhaupt notwendig ist (z.B. mit Verweis auf Design-Dokumente), wann sie zuletzt überarbeitet wurde, etc..

9 Integration

Integration meint das Zusammenfügen einzeln entwickelter Komponenten zu einem Teil- oder Gesamtsystem. Es gibt dafür verschiedene Strategien mit verschiedenen Vor- und Nachteilen.

9.1 Big-Bang-Integration

Damit ist die gleichzeitige Zusammenfügung aller Komponenten gemeint. Sie lässt sich nur bei sehr kleinen Projekten anwenden.

Vorteil ist, dass keine Stubs oder Driver (siehe unten) geschrieben werden müssen und dass keine Regressionstests notwendig sind.

Nachteil ist aber, dass auftretende Fehler kaum lokalisierbar sind.

9.2 Top-Down-Integration

Die Komponenten werden von höher liegenden Schichten der Software-Architektur ausgehend nach unten integriert.

Beispielsweise wird zuerst das GUI mit der Business Logic zusammengefügt, später kommt unten noch der Data-Access-Layer hinzu.

Vorteil ist, dass die höher liegenden Schichten - das sind die für den Benutzer sichtbareren - früh verfügbar sind, was sich für Demo-Zwecke gut eignet. Allerdings sind Stubs zu schreiben (das sind Softwaremodule, die die noch nicht integrierten unteren Schichten simulieren). Außerdem werden hardwarenahe Teile (das sind üblicherweise die fehleranfälligeren) erst spät integriert werden.

9.3 Bottom-Up-Integration

Hier arbeitet man sich ausgehend von den hardwarenahen Schichten nach oben vor. Vorteil ist die frühe Integration der risikobehafteteren Teile (wodurch man später auf ein stabiles Framework aufbauen kann), Nachteil ist dass man Driver (Softwaremodule, oben liegende Schichten simulieren, z.B. GUI-Interaktionen durch ein Script simulieren) schreiben muss. Der Kunde sieht lange keinen Fortschritt, weil die sichtbaren Teile erst spät integriert werden. Es müssen daher zusätzlich Prototypen erstellt werden.

9.4 Build-Integration

Darunter versteht man die gleichzeitige Integration mehrere Komponenten auf verschiedenen Ebene der Architektur, die gemeinsam einen Use Case umsetzen. Beispielsweise könnte GUI, Business Logic und Data-Access-Layer, die für den Abruf von Kontoauszügen benötigt werden, gemeinsam integriert werden. Andere Geschäftsfälle sind dann aber noch nicht abgedeckt.

Vorteil ist, dass schnell ein Use Case vollständig durchgeführt werden kann (gut für Präsentationen). Allerdings bauen oft verschiedene Use Cases auf gleichen Modulen auf, wodurch eventuell später ein Fehler in einem Modul erkannt wird, das vorher in Verbindung mit einem anderen Use Case verwendet wurde und dort funktioniert hat. Es sind in diesem Fall Änderungen und Regressionstests notwendig.

10 Test

Testen ist die Ausführung eines Programms mit der Absicht Fehler zu finden. Testen ist hingegen nicht der Versuch, nachzuweisen dass eine Software korrekt ist (das ist nämlich mit Tests nicht möglich). Ein Fehler ist dabei eine Abweichung des Programm-Verhaltens von der Spezifikation.

Ziel für einen Tester sollte es daher immer sein noch einen Fehler zu finden und nicht alle Fehler zu finden oder zu zeigen dass keine mehr enthalten sind. Fehler sollten möglichst früh gefunden werden da es teurer wird, je später man sie findet (allerdings muss man einen Kompromiss zwischen Kosten für den Test und dadurch erzieltem Nutzen finden).

Das Prinzip des Test-Driven-Devleopment wird im Abschnitt über Technik vorgestellt. Auch wenn es in klassischen Prozessmodellen oft so beschrieben wird, ist Test keine Aufgabe am Ende des Projektes, sondern sollte laufend stattfinden.

10.1 Begriffe

Es kann zwischen folgenden Begriffen unterscheiden werden die umgangssprachlich alle als „Fehler“ bezeichnet werden:

Error

Ein Fehlverhalten aus Benutzersicht, auch wenn es technisch der Spezifikation entspricht. Beispiel: Es wird das Datum im amerikanischen Format erwartet, der Benutzer gibt es im deutschen Format ein.

Fault

Ein Fehlerverhalten aus technischer Sicht, das undefiniertes Verhalten zur Folge hat.

Failure

Mehrere Faults die zu permanenten Systemschäden (z.B. zerstörter Datenbasis) führen. Es sollte auf jeden Fall vermieden werden, dass Faults zu Failures führen.

10.2 Was kann getestet werden?

Im Prinzip sollte jede Anforderung getestet werden können. Dazu müssen die Anforderungen entsprechend formuliert werden, um gegebnenfalls deren Nicht-Erfüllung nachweisen zu können. Beispiele für zu testende Aspekte: Reliability, Usability, Performance, Funktionilty, etc..

10.3 Komponenten eines Tests

Ein Test besteht aus:

- (P) Phasenorientieres Modell zur Test-Durchführung (üblicherweise: Vorbereitung, Spezifikation der Testfälle, Durchführung, Abschluss; begleitet durch Administration).
- (T) Test-Methoden (z.B. Unit-Tests)
- (R) Ressourcen (Test-Equipment)
- (O) Organisatorische Umgebung (z.B. das „4-Augen-Prinzip“)

10.4 Testlevel

Ein Test kann auf verschiedenen Ebenen erfolgen. Kein Test auf einer Ebene macht den auf einer anderen Ebene obsolet. Ein Integrationstest kann z.B. einen Unit-Test nicht ersetzen weil bei auftretenden Fehlern diese kaum lokalisierbar wären. Umgekehrt kann ein Unit-Test auch einen Integrationstest nicht ersetzen, weil man letztendlich die Funktionalität des Gesamtsystems testen will, und nicht bloß die eines Moduls.

- Unit-Test: Test eines einzelnen Moduls
- Integrationstest: Test einen Subsystems
- Systemtest: Test des Gesamtsystems
- Acceptance Test: Test, ob das System den Anforderungen des Benutzers entspricht
- Installationstest: Test, ob ein Roll-Out mit vertretbarem Aufwand möglich ist
- Regressionstest: Test der geänderten Teile nach einer Änderung

10.5 Blackbox- vs. Whitebox-Testing

Neben der Ebene eines Tests lassen sich auf Modulebene (Unit-Tests) auch noch Black- und Whitebox-Tests unterscheiden, je nachdem welches Wissen für deren Durchführung benötigt und genutzt wird.

Beim Black-Box-Test ignoriert man vorhandenes Wissen über den inneren Aufbau einer Software. Man partitioniert die Daten gemäß Spezifikation und testet die Korrektheit

der Input-Output-Relation. Gründe für Abweichungen sind natürlich nicht lokalisierbar. Überdeckung definiert sich anhand der getesteten Anwendungsfälle.

Beim White-Box-Test ist das Wissen über den inneren Aufbau der Software dagegen essentiell. Die Testfälle werden nicht nach der Spezifikation erstellt, sondern nach der implementierten Logik. Die Aufteilung in Äquivalenzklassen erfolgt daher auch nicht aus Benutzersicht, sondern anhang der implementierten Verzweigungen. Überdeckung definiert sich anhand der durchlaufenen Pfade im Programm (C0-, C1-, ..., Cn-Überdeckung).

10.6 Äquivalenzklassen und Grenzwertanalyse

Äquivalenzklassen sind Mengen von Werten für Eingabeparameter die ein gleiches oder ähnliches Verhalten zur Folge haben (sollten). Es muss daher pro Äquivalenzklasse nur ein Wert getestet werden. Besonders die Ränder der Äquivalenzklassen sind dabei interessant, da (wegen eventuell falschen relationalen Operatoren wie z.B. kleiner statt kleiner-gleich) besonders dort Fehler zu erwarten sind.

Für jede Dimension der Eingabeparameter sind gültige und ungültige Äquivalenzklassen zu erstellen.

Ergebnisse eines Tests sind Grundlage für Überarbeitungen sowie für Bestätigungen der Qualität einer Software. Zwar können Tests die Fehlerfreiheit nicht nachweisen, erfolgreiche Tests verstärken aber das Vertrauen in die Software.

11 Wartung

Unter Wartung (Maintenance) versteht man die Überarbeitung von Software nach ihrer ursprünglichen Fertigstellung. Ziel ist der Support der Software während der Betriebsphase (zwischen Akzeptanz des Users und Retirement der Software).

Man kann Maintenance aus 3 Blickwinkeln betrachten:

- Activity-View: Die Tätigkeit „Wartung“
- Process-View: Die einzelnen Schritte im Wartungsprozess
- Phase-Orientated-View: Wartung als Phase im Vorgehensmodell

Der Maintenance-Prozess ist ähnlich dem Softwareentwicklungsprozess, mit dem Unterschied, dass nach dem Delivery wieder Change-Request auftreten können und somit der ganze Prozess wieder bei der Analyse beginnt.

Der Ablauf nach dem Eintreffen eines Change-Request ist überlicherweise der folgende:

- Änderungswünsche verstehen
- Zu ändernde Stellen im Projekt finden
- Änderungen durchführen
- Regressionstests
- Änderungen in den Dokumenten nachziehen um sie konsistent zu halten
- Benutzer neu einschulen, falls das notwendig ist

11.1 Arten der Wartung

- Corrective: Bugs ausbessern (nur ca. 20% der gesamten Wartung)
- Adaptive: Anpassung an geänderte Bedingungen in der Umgebung (z.B. Portierung auf ein neues Betriebssystem)
- Perfective: Verbesserungen für den Benutzer (z.B. neue Features einbauen)
- Preventive: Zukünftige Wartungsarbeiten erleichtern (z.B. durch Verbesserung der Dokumentation)

11.2 Sonstige Aspekte der Wartung

Wartung wird vom Management oft vernachlässigt da der Nutzen nicht immer unmittelbar ersichtlich ist. Außerdem ist es schwierig Personen zu finden, die die Wartung durchführen wollen, da es oft als „second class job“ angesehen wird. Oft wird die Wartungsarbeit Outgesourct.

Nach einer Wartungsarbeit sollte man die beeinflussten Teile der Software unbedingt identifizieren um gezielte Regressionstests anwenden zu können und die ganze Software von Grund auf neu testen zu müssen.

Technisch gesehen gibt es als Wartung im weitersten Sinne das Reengineering (Neuentwicklung einer bestehenden Software unter Berücksichtigung der Änderungswünsche - sehr teuer) und Reverse Enginnering (Generierung von Beschreibungen auf abstrakterer Ebene, z.B. UML, aus bestehendem Code heraus).

12 Quellen

- Folien zur Vorlesung „Software Engineering und Projektmanagement“ im Sommersemester 2007 - QSE, TU Wien

- Deutsche und englische Wikipedia