Masterarbeit

Automatische grafische Dokumentation von Statemaschinen

Ausgeführt am Institut für Software Technologie(IST)

Unter der Anleitung von:
Univ.-Prof. Dipl.-Ing. Dr.techn.
Wolfgang Slany

Durchgeführt von:
Bakk.techn. Abdalla Hassanin

Graz, 08.02.2010

Inhaltsverzeichnis

1 Einführung

1.1 Problemstellung

In Softwareprojekten ist eine ausführliche und genaue Dokumentation der entwickelten Software von wesentlicher Bedeutung. Jede Besprechung im Projekt zwischen dem Kunden und den Entwicklern oder zwischen den Entwicklern selbst soll dokumentiert und der Inhalt und die getroffenen Entscheidungen der Besprechung festgehalten werden. Besprechungen innerhalb des Entwicklerteams sollen beschreiben, welche Themen bzw. Probleme diskutiert und was entschieden bzw. definiert wurde.
Solche Dokumentationen dienen als Referenz sowohl für den Entwickler als auch für den Kunden, um die Korrektheit der Implementierung nachzuweisen oder um aufzuzeigen, welche Änderungen während des Implementierungsprozesses definiert wurden. Außerdem helfen sie beim Testen der Funktionalität der einzelnen Module (*Funktionalitätstest*) und dienen als Referenz, wenn der Kunde seine Software in Zukunft weiterentwickeln möchte.

In der Softwarewelt gibt es zwei Hauptarten von Dokumentationen:

- **Dokumentation vom Kunden:** Diese Dokumentation beschreibt die Aufgabe der zu implementierenden Software und wird vom Kunden geschrieben. Konkret beschreibt sie die Funktionalität der Software. Diese Dokumentation wird erstellt, bevor mit der Implementierung begonnen wird und dient dem Entwickler als Referenz. Weiters hilft sie bei der Erstellung des Prototyps, beim Planen und bei der zeitlichen Abschätzung der Implementierungsphase.
- **Dokumentation vom Entwickler:**. Sie beschreibt, was implementiert wurde bzw. was die Software wirklich macht. Der Entwickler kann sie während oder auch nach der Implementierung erstellen. Allerdings bedeutet das Schreiben dieser Dokumentation viel Aufwand. Der Entwickler muss außerdem ein Testpflichtenheft verfassen, welches beschreibt, wie die Funktionalität der Software getestet werden kann. (Der *Funktionalitätstest* wird im Kapitel „XP Implementieren“ näher erläutert.)

Der Entwickler schreibt seine Dokumentation nicht für den Kunden, sondern für andere Entwickler, die mit ihm in einem Team zusammenarbeiten. Sie enthält die technische Beschreibung für die Funktionsweise der Komponente und dem damit verbundenen Algorithmus. Soll die Software von einem anderem Entwickler weiterentwickelt werden, stellt dieses Dokument eine gute Referenz dar.

Meine Arbeit beschäftigt sich mit der automatisierten Dokumentation von Software.

1.2 Zielsetzung

Ausgehend von einer existierenden Statemaschinenlogik, die in C# implementiert ist, soll eine Software „Statemaschinenvisualizer“ (in Kurzform SMV) entwickelt werden, die beliebige damit erstellte Statemaschinen in grafischer Form übersichtlich zur Anzeige bringen kann. Die Informationen, die zur Erstellung der Grafiken benötigt werden, sollen direkt aus den Typinformationen des Sourcecode ermittelt werden. Dazu wird der in C# vorhandene „Reflektions“-Mechanismus verwendet, der es zur Laufzeit erlaubt, Informationen über die implementierten Objekte auszulesen.

Dabei ist zu untersuchen, ob und in welcher Form die Statemaschinen mit Zusatzinformationen in Form von Attributen versehen werden müssen, um eine möglichst anschauliche Darstellung zu erhalten. Das Ziel ist die Erreichung einer guten „Default“-Ansicht auch ohne zusätzliche Attribute.

Im ersten Schritt soll die gewünschte Darstellung der Statemaschinen erarbeitet werden. Dabei ist zu berücksichtigen, dass die Statemaschinen ineinander verschachtelt sein können und es einfach möglich sein soll, durch diese Ebenen zu „wandern“.

In weiterer Folge werden Detailinformationen festgelegt, die zur Darstellung kommen sollen. Nach der Festlegung der gewünschten Darstellung ist ein Algorithmus zu entwerfen, der ausgehend von den über den „Reflektions“-Mechanismus zur Verfügung gestellten Informationen automatisch ein übersichtliches grafisches Layout einer Statemaschine erstellen kann.

Dieser Algorithmus ist dann in Form von C# Sourcecode zu implementieren und soll die resultierende Grafik in einem scrollbaren Fenster ausgeben können. Der Inhalt des Fensters soll ausgedruckt werden können.

Der SMV soll den Datentyp „Statemaschine“ als Eingabe übernehmen und erstellt anhand des Codes der Statemaschine eine grafische Dokumentation, die die implementierte Statemaschine beschreibt.

Der Entwickler schreibt einen neuen Code bzw. ändert den existierenden Code, der die Statemaschine beschreibt. Er muss sich nicht um die Erstellung oder Aktualisierung der Dokumentation kümmern, die den Ablauf der Statemaschine beschreibt. Es gilt lediglich, den Code von der Statemaschine (Datentyp von der Statemaschine) an den SVM zu übergeben, um anhand des Codes ein grafisches Diagramm zu erhalten, welches den Ablauf der Statemaschine zeigt. Die Dokumentationserzeugung direkt vom Code hat viele Vorteile:

- Der Aufwand für die Erstellung der Dokumentation wird erspart.
- Die Dokumentation ist mit dem Code verbunden. Wird der Code geändert, erscheinen die Änderungen beim nächsten Ausführen der Anwendung in der Dokumentation.
- Aufgrund des vorgegeben Codes kann die Statemaschine von anderen Entwicklern auf einfache Weise weiterentwickelt werden.
- Die grafische Dokumentation zeigt den Ablauf der Statemaschine besser als die klassische Textdokumentation.

2 Extreme Programming XP

In diesem Kapitel wird die Softwareentwicklungsmethode beschrieben, die ich für dieses Projekt verwendet habe. Diese Methode nennt sich Extreme Programming (kurz XP).

2.1 Entstehung

XP wurde von *Kent Beck, Ward Cunningham* und *Ron Jeffries* im Jahr 1990 entwickelt.

Der Anfang war das *Daimler Chrysler*-Projekt „Chrysler Consolidated Compensation“, bei dem ein Abrechnungssystem entwickelt werden sollte, in dessen Rahmen man allerdings mit einigen Problemen zu kämpfen hatte. Die Aufgabe von *Kent Beck, Ron Jeffries* und *Martin Fowler* war es, diese Entwicklungsprobleme zu beseitigen, um das Projekt wieder in die richtigen Bahnen zu lenken. Nach Abschluss der ihrer Aufgabe haben sie ihre gesammelte Erfahrung in Büchern bzw. im Internet publiziert.

Das erste Projekt, in dem XP-Regeln verwendet wurden, war im Jahre 1991, das *WyCash*-Projekt, an dem der Softwareentwickler *Ward Cunningham* gearbeitet hat. Den Namen „Extreme Programming“ hat es damals noch nicht gegeben. Beck hat die von ihm, *Ward Cunningham* und *Ron Jeffries* entwickelte Softwaremethode erst im Jahr 1995 so genannt.

Ein Beispiel für bekannte Projekte, bei denen XP-Regeln umgesetzt wurden, ist „Chrysler Consolidated Compensation“ bei *Daimler Chrysler*. *Kent Beck* war der Projektleiter, der den Auftrag mit zehn Entwicklern realisiert hat.

2.2 Was ist Extreme Programming (XP)?

Extreme Programming (XP) ist eine leichte, effiziente, risikoarme, flexible, kalkulierbare, exakte und vergnügliche Art und Weise der Softwareentwicklung. XP ist für Projekte konzipiert, die sich mit Teams von zwei bis zehn Programmierern umsetzen lassen, die durch die vorhandene EDV-Umgebung in keinem besonders hohen Maße eingeschränkt

sind und bei denen vernünftige Tests innerhalb weniger Stunden ausgeführt werden können ([Beck, Kent 2000]). Es ermöglicht, langlebige Software zu erstellen und auf sich rasch ändernde Anforderungen zu reagieren.

XP setzt nicht auf neue Konzepte für Softwareentwicklung, sondern integriert bestehende Techniken zu einem Vorgehensmodell, bei dem es keinen organisatorischen Ballast gibt. Es fokussiert sich nur auf das Wesentliche. Der Code ist immer der Kern der Softwareentwicklung, deswegen fokussieren die XP-Regeln auf Code z.B. Unit-Test, Refactoring, Programmieren in Paaren, fortlaufende Integration und ähnliche. Für die Entwickler bedeutet die Erstellung von klassischer (= nicht agiler) Dokumentation einen großen Aufwand und ist fehleranfällig, da es keine automatisierte Überprüfung der Übereinstimmung mit dem Code für sie gibt. Außerdem verhindert die klassische Dokumentation, dass die Entwickler auf neue Anforderungen schnell reagieren können, da zuerst die Dokumentation aktualisiert werden muss und dann erst implementiert werden kann. Deswegen erfordert XP vom Entwickler nicht viel klassische Dokumentation, sondern ein ständiges Refactoring und Unit-Tests, da der Unit-Test auch die Dokumentation ist, die vom Entwickler geschrieben wird und den Code testet.

XP ist darauf ausgelegt, Software auf Kundenwunsch zu realisieren und das genau dann, wenn der Kunde dies braucht.

XP bejaht die Ungewissheit, mit der die Softwareentwicklung verbunden ist, stellt aber keinen Freibrief zum Chaos aus. Es folgt vielmehr einem klaren, strukturierten Vorgehen und stellt die Teamarbeit, Offenheit und stetige Kommunikation zwischen allen Beteiligten (Manager oder Leiter des Teams, Entwickler und Kunden) in den Vordergrund. Kommunikation ist eine Grundsäule dieses Vorgehensmodells. Sollte die Kommunikation eines Teams gestört sein oder vom Vorgesetzten behindert werden, kann XP nicht funktionieren ([Multi]).

XP-Teams sollen nicht unter Druck gebracht werden, entsprechend soll es keine Überstunden geben und die Arbeit sollte gleichmäßig auf das Gesamtteam aufgeteilt werden. Daraus resultiert Spaß an der Arbeit und eine hohe Motivation im Team, was wiederum zu langfristiger Arbeitsfähigkeit und geringer Fluktuation führt.

Der Kunde muss nicht bereits am Anfang des Projekts die genaue Anforderung an die Software kennen, er gibt lediglich eine ungefähre Beschreibung für seine Anforderung an. Der Kunde muss seine Anforderungen priorisieren, natürlich kann er diese Priorität jederzeit ändern. Auch die Anforderungen können jederzeit geändert werden.
Damit XP besser als die anderen Softwareentwicklungsmethoden sein kann, muss es schneller beim Erstellen der Software sein sowie höhere Softwarequalität und Zufriedenheit des Kunden erreichen. Der Kunde bekommt laut XP-Methodik ein einsatzbereites Produkt, an dessen Herstellung er aktiv teilgenommen hat.
Angefangen mit einer ersten kleinen Version der Software vergrößert sich der Entwicklungsrahmen ständig. Neue Funktionalität wird permanent entwickelt, integriert und getestet. Um zu der zu entwickelnden Funktionalität zu gelangen, werden gewöhnlich jeweils die Schritte Risikoanalyse, Nutzenanalyse, die Bereitstellung einer ersten ausführbaren Version (Prototyping) und ein Akzeptanztest durchgeführt. Das Modell lässt sich als *agil, iterativ* und *inkrementell* charakterisieren.

XP unterscheidet sich von anderen traditionellen Softwareentwicklungsmethoden hauptsächlich in folgenden Punkten:

- Es gibt immer zur richtigen Zeit Feedback vom Kunden, Tester oder vom Benutzer – dies wird durch kurze Zyklen erreicht.
- Das XP-Projekt ist immer unter Kontrolle durch Tests, die von den Programmierern (Unit Test) und Kunden (funktionale Tests) geschrieben werden, um Fehler frühzeitig zu erkennen und das System weiterzuentwickeln.
- Die Abhängigkeit von einem evolutionären Designprozess, der so lange andauert, wie das System besteht.
- Teamgeist bei der Zusammenarbeit von Programmierern mit ganz gewöhnlichen Fähigkeiten.
- Die Abhängigkeit von den Verfahren, die sowohl den kurzfristigen Instinkten der Programmierer als auch den langfristigen Interessen des Projekts entgegenkommen.

- Das Vertrauen darauf, dass mündliche Kommunikation, Tests und Quellcode die Struktur und den Zweck des Systems zum Ausdruck bringen.
- Einen inkrementellen Planungsansatz, bei dem schnell ein allgemeiner Plan entwickelt wird, der über die Lebensdauer des Projekts hinweg weiterentwickelt wird.
- Die Fähigkeit, die Implementierung von Leistungsmerkmalen flexibel zu planen und dabei die sich ändernden geschäftlichen Anforderungen zu berücksichtigen.

XP verzichtet auf eine Reihe von Elementen der klassischen (traditionellen) Softwareentwicklung, um so eine schnellere und effizientere Kodierung zu erlauben. Die dabei entstehenden Defizite versucht sie durch stärkere Gewichtung anderer Konzepte (insbesondere der Testverfahren) zu kompensieren.

2.3 Methoden der Softwareentwicklung

Um XP besser verstehen zu können, betrachten wir zunächst die Klassifikation der Softwareentwicklungsmethoden. Diese unterteilen sich in solche mit und ohne Methode. Die Entwicklungsmethoden mit Methode werden wieder unterteilt in „klassische“ und „agile“ Methoden. Die folgende Abbildung veranschaulicht die Einteilung.

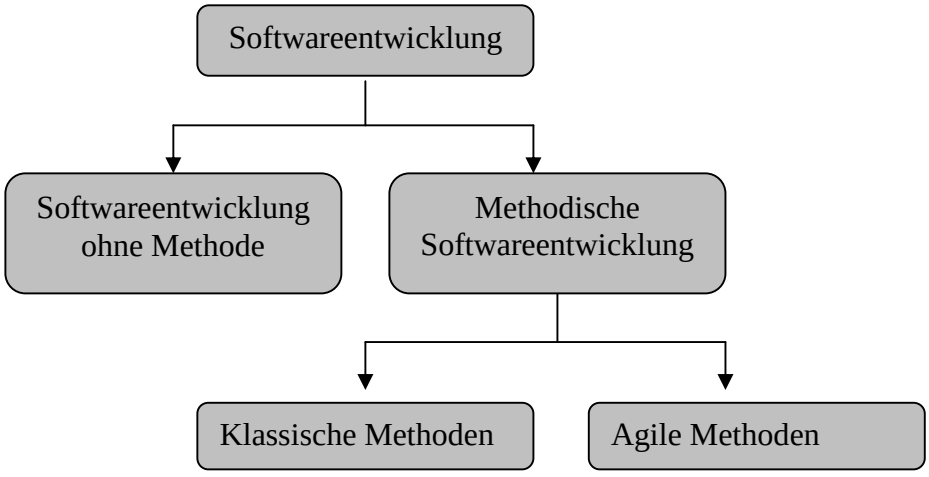


Abbildung 2.3.1: Klassifikation der Softwareentwicklungsmethoden

Die Anforderungen an ein System machen das zentrale Element jeder Softwareentwicklung aus. In ihnen spiegelt sich die Erwartung der Kunden. Der Erfolg

des Projekts lässt sich am besten am Erfüllungsgrad der Anforderungen messen. Wegen der zentralen Bedeutung der Anforderungen hat jedes Vorgehen zur Softwareentwicklung, ob methodisch oder nicht, eine eigene Art, mit Anforderungen umzugehen ([XP Übersicht und Bewertung], [Bögli 2003]).

2.3.1 **Softwareentwicklung ohne Methode**

Softwareentwicklung ohne Methode wird oft auch als reines „Hacking “ oder „Code & Fix“ bezeichnet und steht und fällt mit der Erfahrung der Entwickler. Dieses Vorgehen wird meist von unerfahrenen Softwareentwicklern eingesetzt. Sie haben das Gefühl, jede neue Benutzeranforderung bzw. jede Teilanforderung erst einmal in Programmcodes umsetzen zu müssen, um schon eine gewisse Substanz an Programmcodes zu haben. Der Programmcode wird, um Zeit zu sparen, allerdings nur ganz beschränkt getestet. Ebenso wird auf das Kommentieren weitgehend verzichtet. Die Überlegungen zur programmtechnischen Umsetzung finden nur im Kopf eines einzelnen Programmierers statt. Die Folge ist, dass das Programm für andere, und meist auch nach kurzer Zeit für den Programmierer selbst, unverständlich und nicht erweiterbar wird. Im Endeffekt ist das Programm nur für einen einzigen, sich nicht ändernden Spezialfall einsetzbar, wenn überhaupt ([XP Übersicht und Bewertung]).

2.3.2 **Methodische Softwareentwicklung**

Die methodische Softwareentwicklung wird heute in folgende zwei Kategorien aufgeteilt: Auf der einen Seite stehen die klassischen und auf der anderen die agilen Methoden. Der Hauptunterschied besteht in der Planbarkeit der Entwicklung. Im Zentrum der klassischen Methoden liegt das Ziel, den Ablauf der Entwicklung so gut wie möglich im Vorhinein planbar zu machen, damit man sich dann möglichst an den initialen Plan halten kann. Die agilen Methoden ermöglichen es andererseits, auf Unvorhergesehenes flexibel zu reagieren, indem iterativ immer nur mit einem sehr kurzen Zeithorizont genau geplant wird. Die Planbarkeit und die Stabilität der Anforderungen beeinflussen deshalb maßgeblich die Wahl der Entwicklungsmethode ([XP Übersicht und Bewertung], [Bögli 2003]).

2.3.2.1 Eigenschaften der Klassischen Methoden

Beispiele für klassische Methoden sind Wasserfallmodell, V-Modell, ergebnisorientiertes Phasenmodell, Wachstumsmodell, Spiralmodell und objektorientiertes Vorgehensmodell. Allen klassischen Methoden gemein ist, dass die Softwareentwicklung in eine Folge von Tätigkeiten aufgeteilt wird, während gewisse Ergebnisse erarbeitet werden. Es kann sein, dass die Methoden die Tätigkeiten nur einmal vorsehen oder dass diese iterativ wiederholt werden. Jedenfalls gilt für alle klassischen Methoden, dass sie im Gesamtablauf eine Abfolge von wenigen Phasen beschreiben. Beim Wasserfallmodell sind dies beispielsweise Systemdefinition, Anforderungen, Konzipierung und Realisierung ([Kauflin 2003]).

Der zeitlich große Abstand zwischen der Anforderungsanalyse sowie der Fertigstellung und Einführung der Software stellt dabei das Hauptproblem dar. Zudem lassen die klassischen Methoden nur noch in beschränktem Maße ein unmittelbares Feedback der Kunden während der Softwareentwicklung zu. Im Allgemeinen werden die Anforderungen detailliert analysiert und präzise dokumentiert. Rechtlich gesehen möchten sich beide Parteien, Auftraggeber und Auftragnehmer, also Kunden und Softwareentwickler, mit dieser möglichst genauen Beschreibung auch vertraglich absichern. Dennoch besteht das allgemein bekannte Problem bei der Softwareentwicklung, dass schließlich zur Unzufriedenheit aller ein System abgeliefert wird, das nur den Anforderungen der Vergangenheit genügt und die aktuellen Bedürfnisse nicht berücksichtigt ([XP Übersicht und Bewertung], [Bögli 2003]).

2.3.2.2 Eigenschaften der agilen Methoden

Die Softwareunternehmen wollen immer besser und fehlerfreier sein. Um in unserer schnelllebigen Zeit überlebensfähig zu bleiben, werden sich nur diejenigen Softwareentwicklungsansätze behaupten, die die variablen Kosten, Zeit, Umfang und Qualität am schnellsten und am längsten der Umwelt anzupassen vermögen. XP hat diesbezüglich sehr gute Chancen. Softwareentwicklungsansätze, die sich diese Aufgabe zum Leitsatz vorgenommen haben, nennt man agile Verfahren ([XP Übersicht und Bewertung]).

Beispiele für agile Methoden sind Extreme Programming (XP), Scrum, Crystal, Feature Driven Development (FDD), Rational Unified Process (RUP), Dynamic Systems Development Method (DSDM), Adaptive Software Development (ASD), Breed, Agile Modelling, Pragmatic Programming, Lean Programming, Test Driven Development und Open Source Software Development ([Kauflin 2003] and [Fowler 2003]). Bei agilen Methoden wird die Zusammenarbeit mit Kunden höher gewichtet als das Aushandeln eines Vertrages. Zugunsten einer höheren Flexibilität wird auf eine bis ins Detail ausgefeilte Anforderungsanalyse verzichtet. XP löst dieses Problem beispielweise mittels einfacher *User-Stories* ([Bögli 2003]).

Manche agile Methoden wie z.B. Rational Unified Process (RUP) beschreiben den Gesamtablauf der Softwareentwicklung ähnlich wie klassische Methoden durch eine Abfolge von wenigen Phasen. Andere agile Methoden wie z.B. Extreme Programming arbeiten eher mit Werten, Variablen und Prinzipien. Wichtigstes Ziel agiler Entwicklungsmethoden ist es aber, dem Kunden möglichst schnell Nutzen zu erbringen, wobei primär nur die wichtigsten Funktionalitäten erstellt werden. In den weiteren Iterationszyklen kommen dann die zusätzlichen Funktionen ergänzend hinzu ([XP Übersicht und Bewertung]).

Zur Illustration der Philosophie agiler Methoden sei hier das „Manifest für agile Softwareentwicklung“ angeführt, welches 2001 von der Agile Software Development Alliance entwickelt worden ist ([Westphal 2001]).

Wir zeigen bessere Wege zur Softwareentwicklung auf, indem wir Software entwickeln und anderen dabei helfen. Wir bevorzugen

- *Menschen und Zusammenarbeit vor Prozessen und Werkzeugen*
- *funktionierende Software vor umfassender Dokumentation*
- *Zusammenarbeit mit dem Kunden vor vertraglicher Verhandlung*
- *Reaktion auf Veränderung vor Einhaltung eines Plans*

Das heißt, während die Punkte rechts wertvoll sind, wertschätzen wir die Punkte links mehr ([XP Übersicht und Bewertung]).

In den neunziger Jahren haben sich zwei Gruppen von Agilen gebildet:
Bei der ersten Gruppe spricht man von „*Meta-Prozessen*“. Sie geben nur Rahmenbedingungen an und überlassen die Festlegung der konkreten Prozesse der Erfahrung und Professionalität der Teams. In diese Kategorie fallen das „Adaptive Systems Development“ (ASD) von *Jim Highsmith*, die „Crystal“-Methodenfamilie von *Alistair Cockburn* und „Scrum“ von *Ken Schwaber*, um nur einige zu nennen.
Allgemein üblich ist ein ähnliches Vorgehensgrundmuster: Ein Start-Workshop, in dem kurz geplant wird, eine Arbeitsphase, die durch viel Teamarbeit gekennzeichnet ist und ein Feedback-Workshop zur kritischen Begutachtung des Erreichten. Diese Arbeitsphasen werden permanent wiederholt, wobei die Phasendauer sich den aktuellen Gegebenheiten anpasst. Scrum beispielsweise führt alle 30 Tage ein Feedbackmeeting durch. Konnte eine Aufgabe innerhalb der 30-tägigen Entwicklungsphase nicht realisiert werden, wird sie für den nachfolgenden Iterationszyklus erneut auf die Liste (Product, Backlog) mit den wichtigsten zu implementierenden Aufgaben gestellt. Bei dringenden Prioritätsänderungen unterbricht man die laufende Entwicklungsphase an Ort und Stelle und ruft unverzüglich eine Sitzung (Startworkshop) ein, um alle Aufgaben von neuem zu priorisieren. Die Meta-Prozesse verzichten auf große Dokumentationen und vertrauen auf die Künste der Teammitglieder. Sie müssen ein gesundes Maß an Selbstorganisationstalent mitbringen.
Die zweite Gruppe der agilen Verfahren bilden die „*konkreten Verfahren*“. Sie schlagen dem Entwickler detailliertes Prozessverfahren und Arbeitsprinzipien vor, wie er am besten anpassungsfähige (agile) Software entwickelt. Zu diesen Vertretern gehören beispielsweise das „Feature Driven Development“ (FDD) von *Jeff DeLuca*, das Dynamic System Development Method (DSDM) eines internationalen Gremiums, aber auch Extreme Programming (XP) von *Kent Beck, Ron Jeffries* und Konsorten. Die Ansätze der agilen Entwicklung sind noch lange nicht abgeschlossen. Sie werden ständig von anderen

Personen weiterentwickelt, wie es auch bei der Open Source Bewegung der Fall ist. Jeder richtet sich seine Produkte und Verfahren nach eigenem Geschmack ein.
Alle agilen Verfahren stammen aus Beobachtungen der Menschen bei ihrer Arbeit oder allgemeingültigen Anpassungstheorien der Biologie (ASD), Soziologie (Crystal) oder Verfahrenstechnik (SCRUM, Scrum von Ken Schwaber initiierter Agiler Softwareentwicklungsprozess.) ([Sini 2003],[Cold 2002]).

2.3.3 **Einordnung von Extreme Programming**

Bei der Klassifizierung der Vorgehensmodelle begegnet man nebst den Begriffen „klassische“ und „agile Methoden“ auch den Begriffen „leicht-“ und „schwergewichtige Prozesse“. Bei den leichtgewichtigen Prozessen wird alles weggeworfen, was seinen Nutzen für das Projekt verloren hat, bei den schwergewichtigen wird alles nach Plan durchgeführt. XP ist den agilen Methoden und den leichtgewichtigen Vorgehensmodellen zuzurechnen ([Dicke 2002]).

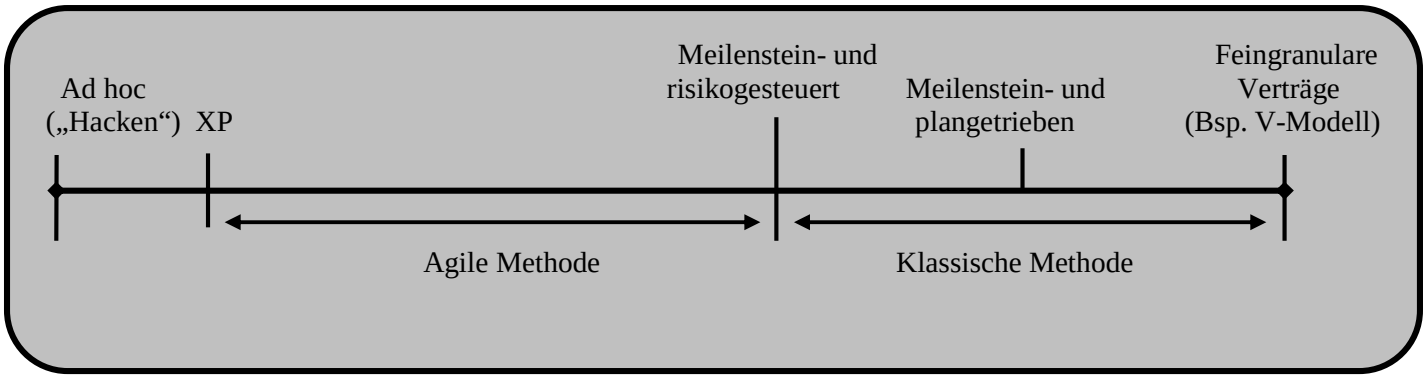


Abbildung 2.3.3.1: Planungsspektrum mit ansteigender initialer Planungsorientierung ([XP Übersicht und Bewertung]).

Abbildung 2.3.3.1 zeigt, wie leicht- und schwergewichtige Entwicklungsmethoden auf einem Spektrum ansteigender initialer Planungsorientierung angesiedelt werden können. Das linke Ende besteht aus dem Extremen undisziplinierten „Hackens“, bei dem überhaupt kein initialer plan existiert. Danach kommen verschiedenste agile Methoden, dann klassische Methoden, die sowohl Meilenstein - und risikogesteuert als auch Meilenstein- und plangetrieben sein können. Das rechte Ende repräsentiert das streng prozessorientierte Vorgehen über feingranulare Verträge. Der Übergangsbereich

zwischen agilen und klassischen Methoden bleibt aber flexibel. In der Abbildung wird klar erkennbar, dass XP nicht mit dem ad-hoc-Vorgehen des völlig planlosen, undisziplinierten „Hackens“ gleichzusetzen ist, auch wenn es am unteren Ende der agilen Methoden steht ([Wasserberg 2003]). Um die Flexibilität von Extreme Programming gegenüber sich ändernden Anforderungen zu zeigen, wird oft die Metapher vom Autofahren verwendet. Beim Autofahren ist es notwendig, ständig die Aufmerksamkeit auf Straße und Umwelt zu richten, um auf unvorhergesehene Ereignisse möglichst adäquat und schnell reagieren zu können. Genauso sollte der Programmierer das Projekt nicht aus den Augen lassen, um möglichst geringe Reaktionszeiten zu erzielen. Kein Autofahrer würde behaupten, eine Fahrt von A nach B komplett im Voraus planen zu können. Auch wenn das Ziel bekannt ist, können unmöglich alle Ereignisse vorausgesehen werden. Die Grundfunktionen beim Autofahren wie Beschleunigen, Bremsen oder Spurwechseln sind vom Fahrziel unabhängige Faktoren. Ziel ist es, einen Prozess zu finden, der die Anwendung der einzelnen Funktionen im richtigen Moment ermöglicht ([Katterl 2004]).

Der Unterschied von Extreme Programming zu herkömmlichen agilen Methoden liegt im Begriff „Extrem“. Aber welche Praktiken sind beim Extreme Programming denn eigentlich extrem? Das Attribut extrem rührt daher, dass bewährte Praktiken der Softwareentwicklung ins Extreme geführt werden. Die Ideen von XP sind eigentlich nicht neu, höchstens die konsequente Anwendung und Kombination sind verstärkte Praktiken von bekannten Ansätzen, wie die folgende Tabelle zeigt ([XP Übersicht und Bewertung], [Beck, Kent 2003]).

Bewährte Praktiken	**…. ins Extreme geführt**
„Code Reviews“ sind gut	Code wird andauernd begutachtet, Pair - Programming
Testen ist gut	Ständiges Testen
Design ist wichtig	Ständiges Redesign und Refactoring
Einfachheit ist gut	Alles so einfach wie möglich
Architektur ist wichtig	Ständige Architekturweiterentwicklung
Kurze Iterationszyklen sind gut	Extrem kurz: Minuten bis Stunden

2.4 Warum wird XP als Methode zur Entwicklung von Software verwendet?

Bei der traditionellen Softwareentwicklung galt die allgemeine Annahme, dass die Kunden einmal und eindeutig erklären, welche Wünsche sie haben und was sie brauchen. Die Software wurde dann mit sämtlichen Funktionen entwickelt, und dies zur vollkommen Zufriedenheit von Kunde und Entwickler. Bei der traditionellen Softwareentwicklung ging man von einem vorhersagbaren Prozess mit festem, von Anfang bis Ende festgelegtem Ablauf des Projekts aus, wobei alle Phasen detailliert protokolliert wurden.

Folgende Probleme traten beim traditionellen Ansatz auf: Da die Kunden oftmals ihre Meinung bezüglich Anforderungen änderten und die Entwickler sich in gewissen Belangen irrten, kam einerseits die Software meist teurer als geplant, und die Kunden erhielten nicht das, was sie sich erhofft hatten ([Jansen, Schmitt 2003]).

XP zielt genau darauf ab, dieses Problem zu lösen. Bei XP kann der Kunde jederzeit die Anforderungen ändern, welche sofort von den Entwicklern übernommen werden, wodurch kein Einfluss auf Planung und Kosten entsteht.

Und das ist der wichtigste Vorteil von *Extreme Programming*, da der Kunde die tatsächlichen Anforderungen zu Beginn eines Softwareprojektes häufig selbst nicht genau kennt oder im Sinne der Entwickler nicht konkret genug benennen kann. Durch diese unklare Definition scheitern viele Projekte bereits in der Planungsphase. Dieses Scheitern zeigt sich jedoch erst nach Fertigstellung. Als Ergebnis erhält der Kunde eine Software, mit der er nicht zufrieden ist.

Softwareentwicklungsprojekte wurden immer komplexer, und die Gefahr des Scheiterns nahm zu. Dieses Phänomen wurde als „Softwarekrise“ bezeichnet. Das Grundproblem der Softwareentwicklung besteht im Risiko, wofür hier einige Beispiele folgen:

- *Terminverzögerungen:* Projekte können nicht termingerecht abgeschlossen werden. Das klassische Projektmanagement ist nicht in der Lage, mit den in der Softwareentwicklung auftretenden technischen, sozialen und ressourcenbedingten Aspekten ein fristgerechtes Fertigstellen der Software zu garantieren.

- *Projektabbruch:* Nach mehreren Terminverzögerungen und damit verbundenen Budgetüberschreitungen wird das Projekt abgebrochen. Es zeichnet sich ab, dass die entstehende Software kein akzeptables Kosten-Nutzen-Verhältnis bieten wird.
- *Fehlerrate:* Die neue Software soll in Betrieb genommen werden, weist aber eine so hohe Fehlerrate auf, dass sie nicht verwendet werden kann. Die Software wird folglich doch nicht eingesetzt und das alte System muss, soweit möglich, noch den Betrieb aufrechterhalten.
- *System wird unrentabel:* Fehlerrate oder Änderungskosten der in Betrieb genommenen Software steigen so hoch an, dass das System schneller ausgetauscht werden muss als ursprünglich angenommen.
- *Geschäftsprozess falsch verstanden:* Bei der Analyse der Benutzeranforderungen und Spezifikation der zu entwickelnden Software wurde nicht ausreichend in Betracht gezogen, dass sich die zu unterstützenden Geschäftsprozesse ändern. Die Software löst nach einer gewissen Zeit das ursprüngliche Problem nicht mehr. Der Kunde wird unzufrieden. Das Problem ist auch bekannt als „moving target“. Je nach Dauer der Softwareentwicklung und Änderungsgeschwindigkeit der Geschäftsprozesse kann dies sogar noch vor Fertigstellung der Software auftreten.
- *Falsche Funktionsfülle:* Es werden viele Funktionen eingebaut, die dem Kunden keinen echten Nutzen bringen. Dies verlängert die Entwicklungszeit, verteuert damit die Softwareentwicklung, macht die Software wartungsunfreundlicher und wird letztendlich vom Kunden meist nicht gewürdigt.
- *Personalwechsel:* Aufgrund eines schlechten Projektmanagements und damit einhergehender Missstimmung im Team werden im Allgemeinen alle guten Programmierer des Projekts nach spätestens zwei Jahren überdrüssig und kündigen.

Nachdem wir die meisten Risiken, die während der Softwareentwicklung auftauchen können, aufgelistet haben, stellt sich nun die Frage: *Wie geht XP mit den oben angeführten Risiken um?*

XP kann mit diesen Risiken auf allen Ebenen des Entwicklungsprozesses umgehen, ist zudem sehr produktiv, ermöglicht qualitativ hochwertige Software und macht in der Praxis großen Spaß.

Zu allen diesen Risiken versucht XP Gegenmaßnahmen zu bieten, die wie folgt beschrieben werden:

- *Terminverzögerung:* XP fordert kurze Releasezyklen mit einer maximalen Dauer von einigen Monaten, sodass sich Terminverzögerungen immer im Rahmen halten. In XP werden innerhalb eines Release in ein- bis vierwöchigen Iterationen vom Kunden geforderte Funktionen implementiert, damit dieser ein detailliertes Feedback zum Arbeitsfortschritt erhält. Innerhalb einer Iteration wird XP in Schritten von ein bis drei Tagen geplant, damit das Team bereits während einer Iteration Probleme lösen kann. Schließlich fordert XP, dass die Funktionen mit der höchsten Priorität zuerst implementiert werden, sodass diejenigen Funktionen, die sich nicht zum geforderten Liefertermin realisieren lassen, von geringer Bedeutung sind.
- *Projektabbruch:* XP fordert den Kunden auf, die kleinste geschäftlich sinnvolle Version zu wählen, sodass die Fehleranfälligkeit minimiert wird, bevor die Software die Produktionsreife erreicht und somit größeren Wert erhält.
- *Das System wird unrentabel:* XP erstellt und behält eine umfassende Menge von Tests, die nach jeder Änderung (mehrmals täglich) ausgeführt werden, um sicherzustellen, dass die Qualitätsanforderungen erfüllt werden. XP hält das System stets in erstklassigem Zustand. Man lässt nicht zu, dass sich unnötige Prozesse und Komponenten ansammeln.
- *Fehlerrate:* XP testet sowohl aus der Perspektive der Programmierer, indem Tests zur Überprüfung einzelner Funktionen geschrieben werden, als auch aus der Perspektive des Kunden, indem Tests zur Überprüfung der einzelnen Leistungsmerkmale des Programms entwickelt werden.
- *Das Geschäftsziel wurde falsch verstanden:* XP macht den Kunden zum Mitglied des Teams. Die Projektspezifikation wird während der Entwicklung fortwährend

weiter ausgearbeitet, sodass sich Lernerfahrungen des Teams und des Kunden in der Software widerspiegeln können.

- *Das Geschäftsziel ändert sich:* XP verkürzt die Releasezyklen, weshalb innerhalb des Entwicklungszeitraums einer Version weniger Änderungen auftreten. Während der Entwicklung einer Version kann der Kunde neue Funktionen durch solche ersetzen, die bislang noch nicht implementiert sind. Das Team bemerkt nicht einmal, ob es an neu hinzugekommenen Funktionen arbeitet oder an Leistungsmerkmalen, die vor Jahren definiert wurden.
- *Falsche Funktionsfülle:* XP besteht darauf, dass nur die Aufgaben mit der höchsten Priorität angegangen werden.
- *Personalwechsel:* XP fordert von den Programmierern, dass sie für die Aufwandsschätzung und die Fertigstellung der eigenen Aufgaben selbst die Verantwortung übernehmen, gibt ihnen Feedback über die tatsächliche Fertigstellungsdauer, damit sie ihre Einschätzung realistischer treffen können und respektiert diese Schätzung. Es gibt klare Regeln dafür, wer solche Einschätzungen vornehmen und ändern kann. Daher ist es unwahrscheinlicher, dass Programmierer frustriert werden, weil jemand etwas offensichtlich Unmögliches von ihnen verlangt. XP fördert auch die Kommunikation im Team und dämpft damit das Gefühl der Vereinsamung, das oft Grund für die Unzufriedenheit mit einer Position ist. Schließlich beinhaltet XP ein Modell für Personenwechsel. Neue Teammitglieder werden dazu ermutigt, nach und nach immer mehr Verantwortung zu übernehmen und erhalten während der Arbeit untereinander und von erfahreneren Programmierern Unterstützung.

XP ist eine erfolgreiche Art von Softwareentwicklung, weil die Kundenzufriedenheit dabei immer die höchste Priorität hat bzw. diese immer zu erreichen versucht ([XP Übersicht und Bewertung]).

2.5 Bestandteile von XP

XP basiert auf bestimmten Werten, Prinzipien, Variablen und Praktiken. Es braucht dieses Viergespann und diesen Hintergrund, um XP komplett zu verstehen sowie sinnvoll

und produktiv einzusetzen, da es nicht einfach mit Hilfe seiner technischen Mittel betrieben werden kann.
Während die Werte das vermitteln, was XP fördern will, dabei aber sehr abstrakt bleiben, liefern die Prinzipien Gründe für diese abstrakten Werte und bilden damit eine Brücke hin zu den Praktiken, die mit XP verbunden sind und seinen eigentlichen Arbeitsprozess bilden. Allein gestellt ergeben die Praktiken keinen Sinn. Daher ist es wichtig, die dahinter liegenden Gründe und Prinzipien zu betrachten.

2.5.1 **Die Werte**

2.5.1.1 Kommunikation

Die meisten Probleme im Projektteam rühren von mangelhafter Kommunikation zwischen den Teammitgliedern, in der Regel die Hauptursache für den Abbruch eines Projekts. Beispiele für Kommunikationsprobleme sind:

- Jemand bespricht etwas Wichtiges nicht mit anderen Teammitgliedern.
- Programmierer anderen nichts von einer wichtigen Designänderung mit.
- Programmierer den Kunden nicht die richtigen Fragen, sodass eine wichtige Entscheidung falsch ausfällt.
- Manager stellen den Programmierern nicht die richtigen Fragen und erhalten nicht die richtigen Informationen über den Projektstatus.

Kommunikationsprobleme können viele Ursachen haben. Beispielsweise überbringt ein Programmierer seinem Manager eine schlechte Nachricht und wird dafür bestraft. Ein Kunde teilt dem Programmierer etwas Wichtiges mit, dieser scheint diese Information jedoch zu ignorieren ([Beck, Kent 2000]).
Um die Kommunikationsprobleme zu beseitigen, soll der Coach diese Probleme so früh wie möglich entdecken und identifizieren.
Die Kommunikation hilft z.B. beim Auftreten überraschender Fehler. In den meisten Fällen existiert im Team bereits das Wissen, um einen solchen Fehler zu bereinigen und zu vermeiden. Doch auch wenn keine direkte Lösung gefunden werden kann, bietet die Kommunikation zumindest die Chance, diesen Fehler anzusprechen und damit das

erneute Auftreten in anderen Situationen zu vermeiden. Kommunikation bedeutet hierbei jedoch nicht, alibihalber irgendwelche Gespräche zu führen, denn private Dinge helfen dem Team nicht dabei, bessere Software zu schreiben.
Kommunikation im Sinne des XP-Konzepts soll durch den ständigen fachlichen Austausch das Team als Ganzes stärken, indem Wissen verteilt wird und so dem Projekt zu Gute kommt.
Bei XP werden viele Techniken angewendet, die Kommunikation erfordern. Dabei handelt es sich beispielsweise um Komponententests, Programmieren in Paaren und Aufwandsabschätzung von Aufgaben. Diese haben den Effekt, dass Programmierer, Kunden und Manager miteinander kommunizieren müssen ([Beck, Kent 2003]).

2.5.1.2 Einfachheit

Für die Lösung eines Problems wird immer die einfachste Methode ausgewählt. Die einfachsten Lösungen sind leichter nachzuvollziehen und bedürfen weniger Erklärungen. Für XP gilt hierfür die Leitfrage „What is the simplest thing that could possibly work?“ ([Beck, Kent 2000]). Es geht also darum, überflüssige Komplexität zu vermeiden. Die Lösung soll sich einfach auf das zu behandelnde Problem konzentrieren. Die Behandlung von Aspekten die nicht direkt das Problem betreffen, sollen vermieden werden, auch auf die Gefahr hin, dass schon morgen die Anforderungen vielleicht wieder ganz anders sind. Dabei darf man jedoch nicht zu vereinfacht arbeiten. Unterliegt das Problem z.B. sicherheitskritischen Einschränkungen, muss auch bei der einfachsten Lösung der Aspekt der Sicherheit in Betracht gezogen werden. Die einfachste Lösung spart Zeit bei der Erweiterung der Software. XP -Entwickler sind überzeugt, dass es besser ist, heute etwas Einfaches zu tun und morgen im Falle einer Änderung etwas mehr zu zahlen, anstatt heute etwas Kompliziertes zu entwickeln, das vielleicht niemals verwendet werden kann.
Einfachheit und Kommunikation gehören zusammen, denn je einfacher etwas gebaut wird, desto geringer ist der Kommunikationsaufwand. Und je mehr kommuniziert wird, desto eher kristallisiert sich heraus, welche Funktionalitäten benötigt werden, um das System möglichst einfach zu halten ([Beck, Kent 2003]).

2.5.1.3 Feedback

Um Qualität zu sichern, ist Feedback nötig. Der Entwickler erhält dieses durch den Partner beim Pair-Programming, die automatisierten Modultests und am Ende eines Arbeitstages durch die Integrationstests. Der Kunde erhält durch die häufigen Releases ebenso Feedback. Zudem wird er laufend über den Projektstand und den Projektfortschritt informiert ([Beck, Kent 2003]).

Feedbacks werden zu verschiedenen Zeiten eingesetzt, z.B. erhalten die Programmierer durch Komponententests von Minute zu Minute konkrete Feedbacks über den Zustand des Systems. Durch die Zeitabschätzung erhalten Manager oder Kunden Feedback über den Zeitplan.

2.5.1.4 Mut

In XP-Projekten muss der Entwickler Mut haben, mit den anderen Teammitgliedern zu sprechen (Manager, Kunden und Kollegen) und erkannte Probleme anzugehen. Die Werte Kommunikation, Feedback und Einfachheit verlangen Mut. Bei der Kommunikation ist es verständlich, dass es Menschen schwer fallen kann, das Gespräch zu suchen und unangenehme Dinge wie Fehler oder schlechtes Design anzusprechen. Schlechtes Feedback zu bekommen, dieses anzunehmen und damit umzugehen erfordert ebenfalls Mut. Dasselbe gilt für das Prinzip der Einfachheit, ganz bewusst Aspekte weg- bzw. unbeachtet zu lassen. Eine Portion Courage braucht es ebenfalls, unnötig komplizierte Codes zu eliminieren und wieder komplett neu mit der Problemlösung zu beginnen ([Beck, Kent 2003]).

Schließlich verlangt XP, dass der Entwickler nur jene Probleme lösen soll, die er momentan wirklich hat und nicht die, die später vielleicht auftauchen könnten. Mit Mut löst er die heutigen Probleme heute und die morgigen Probleme morgen.

2.5.1.5 Weitere Werte

Zum Teil werden auch sieben Werte für XP genannt, wobei dann Lernen, Respekt und Qualität im Sinn von Qualitätsbewusstsein ergänzend hinzukommen ([Westphal 2004]).

Respekt gegenüber den anderen Projektmitgliedern und Interesse am Lernen sind unabdingbare Voraussetzungen für alle Beteiligten an XP-Projekten.
Der Wert Respekt wurde von *Kent* in seiner zweiten komplett überarbeiteten Auflage von „Extreme Programming explained“ (2004) eingeführt und wird auch zum evolutionär gewachsenen XP2 gezählt. Genauso gewachsen ist der Wert des Respekts, der sich aus allen vier vorherigen Werten ergibt. Wenn Teammitglieder sich und die Arbeit, die sie für ein Projekt leisten, nicht respektieren, kann auch XP das Projekt nicht retten. Respekt sichert dem Projekt und dem Team die Loyalität der Mitglieder und liefert auch Motivation und Antrieb, um die Arbeit erfolgreich zu bestreiten.
Qualität ist bereits eine der vier Variablen (siehe Variable), und ein angemessenes Qualitätsbewusstsein ist sehr wichtig für alle Beteiligten. Hier wird nun deutlich, dass XP nur mit ganz bestimmten Mitarbeitenden möglich ist ([XP Übersicht und Bewertung]).

2.5.2 **Die Prinzipien**

Die vier Werte geben nur den Stil von XP vor, sind aber zu vage, als dass sie sich in dieser Form direkt in die Praxis umsetzen ließen. Deshalb wurden aus ihnen zunächst einige konkrete Prinzipien abgeleitet. Diese sollen helfen, die vier Werte in die Praxis umzusetzen und zu konkretisieren. Denn was z.B. für die eine Person einfach ist, kann für eine andere kompliziert sein. Die abgeleiteten Prinzipien können in primäre und sekundäre Prinzipien unterteilt werden ([Beck, Kent 2003], [XP Übersicht und Bewertung]).

2.5.2.1 **Primäre Prinzipien**

- *Unmittelbares Feedback:* Die Zeit zwischen einer Aktion und deren Feedback soll kurz sein.
- *Einfachheit anstreben:* Im Gegensatz zu klassischen Entwicklungsmethoden wird bei der Softwareentwicklung nach XP nicht im Hinblick auf mögliche Erweiterungen oder auf Wiederverwendbarkeit programmiert. Oberste Priorität hat die Einfachheit. XP fordert also, die heute anliegende Aufgabe gut zu erledigen (Tests, Refactoring, Kommunikation) und der Fähigkeit zu vertrauen, Komplexität

bei Bedarf zu einem späteren Zeitpunkt hinzufügen zu können ([XP Übersicht und Bewertung]).

- *Inkrementelle Veränderung:* Änderungen im XP-Projekt sollen in kleinen Schritten erfolgen, was die Fehlerwahrscheinlichkeit verringert.
- *Änderungen willkommen heißen:* Die Grundeinstellung gegenüber Änderungen ist a priori positiv. Es wird nicht aus Unmut oder Angst an schlechten Lösungen festgehalten, sondern das bisher Erstellte aufgegeben und ein neuer Weg eingeschlagen ([XP Übersicht und Bewertung]).
- *Qualitätsarbeit leisten:* Qualität ist zentraler Baustein der Motivation und wird von allen Mitarbeitenden angestrebt. Wird schlechter Qualität nicht konsequent entgegengewirkt, entsteht ein System, das kaum mehr weiter zu entwickeln ist. ([XP Übersicht und Bewertung])

2.5.2.2 Sekundäre Prinzipien

Zu den weniger zentralen Grundsätzen, den sekundären Prinzipien, gehören ([Beck, Kent 2003]).

- *Lernen lehren:* Von doktrinären Aussagen und Vorgaben wird generell abgesehen. Lernen und der ständige Wille, sich zu verbessern, stehen im Zentrum von XP. Aus dem unmittelbaren Feedback soll gelernt werden. ([XP Übersicht und Bewertung]).
- *Kleine Anfangsinvestitionen:* Große Budgets verleiten zu Auswüchsen ([XP Übersicht und Bewertung]).
- *Spiele, um zu gewinnen:* Die Einstellung und der Mut sind entscheidend für den Projekterfolg. Dienst nach Vorschrift oder das Bestreben, sich möglichst nichts zuschulden kommen zu lassen, verhindern den Erfolg. Die Softwareentwicklungsprojekte sollen spielen, um zu gewinnen und nicht spielen, um nicht zu verlieren. Softwareentwicklung, die auf Sieg spielt, tut alles, was dem Team diesen Sieg ermöglicht, und nichts, was nicht zum Sieg beiträgt ([Beck, Kent 2000]).
- *Gezielte Experimente:* Mutige Entscheidungen sind durch Experimente abzustützen, um das Risiko in vernünftigem Rahmen zu halten. Das Ergebnis einer

Designsitzung sollte daher eine Reihe von Experimenten sein, mit denen sich Fragen, die während der Sitzung gestellt wurden, überprüfen lassen und kein fertiges Design. Das Ergebnis einer Diskussion der Anforderungen sollte ebenso eine Reihe von Experimenten sein. Jede abstrakte Entscheidung sollte getestet werden ([Beck, Kent 2000]).

- *Offene, ehrliche Kommunikation:* Dem Management müssen auch unangenehme Nachrichten mitgeteilt werden können, und zwar frühzeitig und ohne dafür bestraft zu werden. Auf der anderen Seite müssen die Programmierer in der Lage sein, die Konsequenzen der Entscheidungen von anderen zu erklären. Sie müssen einander sagen können, wo Probleme im Code vorhanden sind und dürfen keine Angst haben, ihre Befürchtungen zu äußern und dem Management schlechte Nachrichten frühzeitig zu überbringen ([Beck, Kent 2000]). Unterschiedliche Meinungen müssen ausdiskutiert und Verstöße gegen Standards angesprochen werden können. Wenn man bemerkt, wie sich jemand umsieht und nach möglichen Zuhörenden Ausschau hält, bevor er eine Frage beantwortet, dann ist dies bereits ein ernstes Anzeichen dafür, dass das Projekt in größeren Schwierigkeiten steckt ([XP Übersicht und Bewertung]).
- *Verantwortung übernehmen:* Wenn jemandem vorgeschrieben wird, was er zu tun hat, wird er im Laufe der Zeit tausend Möglichkeiten finden, seine Frustration zum Ausdruck zu bringen, was sich meist zum Nachteil des Teams und häufig zum Nachteil des Betroffenen selbst entwickelt. Die Alternative besteht darin, dass Verantwortung übernommen statt zugewiesen wird. Das heißt aber nicht, dass man immer genau das tun darf, wozu man gerade Lust hat. Daher gilt, mit dem Instinkt der Mitarbeitenden arbeiten, nicht dagegen. Denn Mitarbeitende können nicht über längere Zeit zu etwas gezwungen werden, das ihrem inneren Gefühl widerspricht z.B. übermäßiges Dokumentieren ([XP Übersicht und Bewertung]).
- *An örtliche Gegebenheit anpassen:* XP muss immer wieder den konkreten Gegebenheit im Hier und Jetzt angepasst werden und kann nicht als feste Vorgabe einfach 1:1 übernommen werden.

- *Reisen mit leichtem Gepäck:* Jeglicher überflüssige Ballast im Projektvorgehen soll vermieden und weggeworfen werden ([XP Übersicht und Bewertung]).
- *Ehrliches Messen:* Mut zu ehrlicher Schätzung und Messung des Fortschritts und der Leistung sind Voraussetzung für XP ([XP Übersicht und Bewertung]).

2.5.3 **Die Variablen**

In jedem Projekt werden folgende vier Variablen kontrolliert:

- Kosten
- Qualität
- Zeit
- Umfang

Der Zusammenhang zwischen diesen Variablen ist so, dass bei drei festgelegten Faktoren die vierte Variable ebenfalls gegeben ist. Die Idee dahinter ist, dass der Kunde höchstens drei Variablen nach seinem Wunsch bestimmen kann. Es ist die Aufgabe der Entwickler, dem Kunden den Einfluss auf den vierten Faktor zu erläutern ([Armin 2004]). Dieser Zusammenhang wird auch „3+1-Regel“ genannt. Dies gilt prinzipiell für alle Softwareprojekte, egal ob XP eingesetzt wird oder nicht.
In XP können die Programmierer den Wert der vierten Variablen bestimmen. Manchmal glauben Manager und Kunden, dass sie die Werte aller vier Variablen festlegen können. Wenn das der Fall ist, leidet stets die Qualität darunter, da niemand bei sehr hoher Belastung gute Arbeit leisten kann. Es ist zudem wahrscheinlich, dass die Einhaltung des Zeitrahmen nicht möglich ist. Die Software wird zu spät fertig und taugt womöglich nicht einmal etwas.
Die Lösung besteht darin, diese Variablen sichtbar zu machen. Wenn alle Beteiligten – Programmierer, Kunden und Manager – alle vier Variablen vor Augen haben, können sie sich bewusst entscheiden, welche davon sie festlegen wollen. Ist die sich daraus ergebende vierte Variable nicht akzeptabel, können sie die Ausgangswerte ändern oder drei andere Variable auswählen ([Beck, Kent 2000]).

Abbildung 2.5.3.1 zeigt die vier Variablen im Projekt (Umfang, Kosten, Zeit, Qualität), gegeben durch das Quadrat in der Mitte (blaues Quadrat). Ändern sich ein bis drei Variablen im Laufe des Projekts, so bewegen sich die entsprechenden Eckpunkte des Quadrates für diese Variablen an eine neue Position. Die restlichen Variablen passen sich dann an, indem sie quasi an eine neue Position gezogen bzw. geschoben werden. Soll beispielsweise in einem Projekt die Qualität der Ergebnisse verbessert, die Zeit im Projekt allerdings verkürzt und sollen die Kosten eingehalten werden, so geht dies nur mit einem verminderten Umfang im Projekt, wie das schraffierte verzerrte Viereck andeutet ([XP Übersicht und Bewertung]).

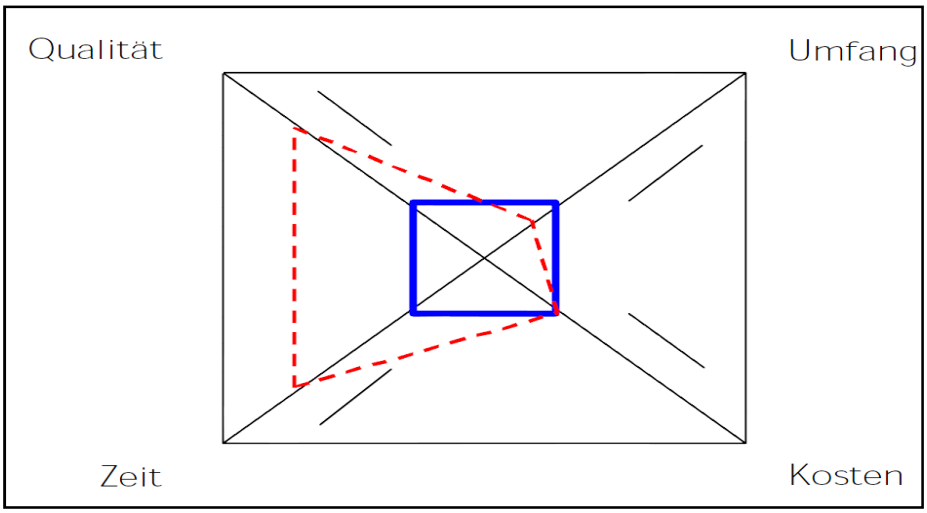


Abbildung 2.5.3.1: XP Variablen ([XP Übersicht und Bewertung])

- *Kosten:* Den stärksten Einfluss auf die Kosten hat das Personal. Je mehr Leute im Projekt beschäftigt werden, desto höher steigen die Kosten, und zwar nicht linear, wobei auch noch die Wirkung verzögert eintritt. Die Verzögerung entsteht durch die Einarbeitungszeit von neuen Mitarbeitenden, die nicht lineare Kostensteigerung hängt mit dem Anwachsen des Koordinations- und Kommunikationsaufwandes bei zunehmender Zahl an Teammitgliedern zusammen. Falls dann bei einem in Verzug geratenen Projekt noch weitere Personen hinzugezogen werden müssen, kann die Verzögerung auf den Zeitplan noch größer ausfallen. In diesem Fall empfiehlt es sich, Personen schrittweise einzustellen ([XP Übersicht und Bewertung]). Kontraproduktiv ist auch das in gewissen Firmen vorhandene Prestigedenken, ein

„150-Personen-Projekt“ leiten zu müssen. Solche Projekte sind besonders in Softwarefragen äußerst schwierig zu leiten ([Beck, Kent 2003]). Kosten können auch durch Hilfsmittel wie größere Bildschirme oder schnellere Rechner, bessere Ausbildung oder arbeitsfreundlichere Umgebung entstehen. Investitionen in die Motivation der Arbeitnehmenden werden beim XP groß geschrieben, von Überstunden wird dagegen dringend abgeraten ([Beck, Kent 2001]).

In der Abbildung 2.5.3.2 wird der unterschiedliche Kostenverlauf bei Änderung je nach Entwicklungsmethode sichtbar. Im Gegensatz zur Verwendung von XP steigen bei der Anwendung von klassischen Methoden die Kosten mit fortschreitendem Projektstatus exponentiell ([Beck, Kent 2003]).

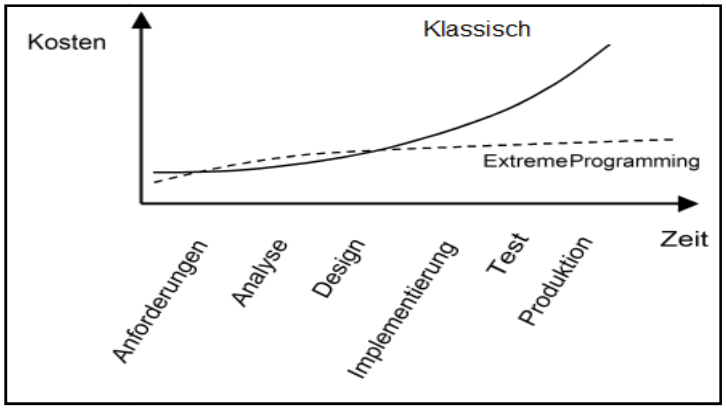


Abbildung 2.5.3.2: Änderungskosten ([Wikipedia])

- *Qualität:* Bei der Qualität unterscheidet *Kent Beck* zwischen innerer und äußerer Qualität. Die äußere Qualität ist jene, welche der Kunde in Form von Fehlern, Aussehen von Benutzeroberfläche und Performance des Systems wahrnimmt ([Beck, Kent 2001]). Die innere Qualität bezeichnet die Güte des Designs und der Tests. Mit geringer innerer Qualität kann zwar kurzfristig Zeit gespart werden, längerfristig baut man aber ein System, dessen Wartung unmöglich wird ([XP Übersicht und Bewertung]).

  Qualität hat einen nicht zu vernachlässigenden Einfluss auf die Motivation der Mitarbeitenden. Die Entwickler können sich besser mit Ihrem Arbeitsresultat identifizieren und sind mit Ihrer Aufgabe zufriedener, je höher die Qualität ihres

Produkts ist. Produkte von minderwertiger Qualität herzustellen, hat eine demoralisierende Wirkung ([Hofer 2002]).

Durch Sparmaßnahmen bei der Qualität lassen sich meist nur kurzfristige Gewinne erzielen, was sich mit einem „Leben auf Pump“ vergleichen lässt. Kurzfristig ist man vielleicht schneller, die Mängel fordern später aber ihren Tribut zurück ([Beck, Kent 2003]).

- *Zeit:* Die Hebel Zeit und Umfang sind besser geeignet, verändert werden zu können. Problematisch ist nur, dass man dazu neigt, sich bezüglich des Arbeitspensums zu überschätzen, ohne die zeitlichen Konsequenzen genügend zu berücksichtigen ([Beck, Kent 2001]).

  Zusätzlich kann Zeit die Qualität verbessern oder die Möglichkeit schaffen, mehr Funktionalität zu implementieren. Falls zu viel Zeit zur Verfügung steht, kann der negative Effekt auftreten, dass wertvolle Nutzerfeedbacks unnötig lange hinausgezögert werden ([Meyer 2004]). Zu knappe Zeitberechnung hingegen wirkt sich auf die Qualität meist negativ aus ([Beck, Kent 2003]).

  Oft sind die Zwänge, denen die Variable Zeit unterliegt, von außen gegeben. So ließ sich z.B. die Jahr-2000-Umstellung ebenso wenig verschieben wie jährlich wiederkehrende Anpassungen einer Software (z.B. für die Erstellung der Steuererklärung nach neuen Richtlinien) oder ein Projekt, das für eine Messe realisiert werden soll. Die Variable Zeit liegt daher oft nicht im Entscheidungsbereich der Projektmanager, sondern in dem des Kunden ([Beck, Kent 2003]).

- *Umfang:* Die Kunden haben oft keine genaue Vorstellung bezüglich des Umfangs. Nicht selten wird ihnen erst nach dem ersten Release klar, was sie eigentlich wollten. Die Variable Umfang lässt sich meist am leichtesten verändern ([Beck, Kent 2003]). Eine wichtige Schlussfolgerung ist, allen Beteiligten die Notwendigkeit des Zusammenhangs dieser vier Variablen vor Augen zu führen. Wenn Programmierer, Kunde und Manager transparent über Veränderungen und deren Auswirkungen kommunizieren, können sie bewusst entscheiden, welche Variablen sie festlegen wollen. Falls die sich daraus ergebende vierte Variable

nicht akzeptabel ist, besteht immer die Möglichkeit, die Ausgangswerte der drei übrigen Variablen zu ändern ([XP Übersicht und Bewertung]). *Kent Beck* empfiehlt, dass der Projektmanager den Umfang verwaltet. Denn wenn er den Umfang gezielt kontrolliert, kann den Kunden und übergeordneten Managern die Kontrolle über Kosten, Qualität und Zeit aufgetragen werden ([Beck, Kent 2003]).

Wie reagiert XP auf mögliche Veränderungen bezüglich des Umfangs? Einerseits erreichen Projekte unter XP durch die kurzen Iterations- und Release-Zyklen häufiger Feedback, andererseits ermöglicht XP genauere Aufwandschätzungen durch die Unterteilung in kleine User-Stories ([XP Übersicht und Bewertung]). Bessere Aufwandschätzungen verringern das Risiko, Funktionen weglassen zu müssen. Wichtig ist, dass der Kunde bei jeder Planung den Geschäftsnutzen jeder User-Story bewertet, so dass die wichtigsten Anforderungen zuerst implementiert werden. Deshalb sind bei einer Umfangreduzierung nur gerade die weniger wichtigen Funktionen betroffen ([Beck, Kent 2003]).

2.5.4 **Die 12 Praktiken**

Die 12 Praktiken werden im Kapitel 4 „XP Implementieren“ besprochen.

2.6 Wann sollte XP nicht verwendet werden?

Die Softwareentwicklungsmethode Extreme Programming ist nicht für jedes Projekt geeignet. Damit die XP-Regeln in einem Projekt umgesetzt werden können, muss das Projekt folgende Bedingungen erfüllen:

- Unternehmenskultur: Manche Unternehmen haben kein Interesse, neue Technologien zu verwenden. Jene, die XP einsetzen wollen, sollen ihren Mitarbeitern die Freiheit geben, die Entwicklungsumgebung für XP umzustrukturieren z.B. Unit-Test implementieren, Integrationsrechner etc.
- Flache Kostenänderungskurve: XP ist nicht für Projekte geeignet, bei denen man im Voraus alle Gefahren auszuschließen versucht. Das Vorgehen, zuerst nur einige Funktionen zu erstellen, stößt hier aus Sicherheitsgründen oft an Grenzen. Aus

diesen Gründen ist die Annahme einer flachen Kostenänderungskurve nicht anwendbar.

- Kleine Teams: Das optimale XP-Team umfasst bis ca. 12 Mitglieder, ansonsten hat das Team Kommunikationsprobleme. Damit das XP-Team gut kommunizieren kann, sollen sich die Entwickler bzw. XP-Team-Mitglieder in örtlicher Nähe zueinander befinden. Ohne Kommunikation kann XP nicht eingesetzt werden.
- Gewaltentrennung: XP teilt das Projektteam in zwei Gruppen, nämlich Geschäftsseitenteam und Entwicklungsseitenteam. Jedes Team trifft nur die Entscheidungen in seinem Bereich. Beispielsweise schätzen die Entwickler die Zeit ab, die Geschäftsseite priorisiert die Funktionen. Jedes Team muss die Entscheidungen des anderen Teams respektieren.
- Persönlichkeitsprofil der Entwickler: An die Entwickler werden äußerst hohe Anforderungen bezüglich ihrer sozialen Kompetenz gestellt. Der gemeinsame Erfolg muss über dem eigenen Erfolg stehen. Diese Anforderung ist außerordentlich anspruchsvoll und für heutige Arbeitnehmende schwierig, da unser gesamtes Bildungssystem auf individueller Leistungsbewertung basiert. Die Entwickler müssen sich und ihre Handlungen selbstdiszipliniert und kritisch hinterfragen. Wichtig sind ebenfalls Prozessreflexion und die Fähigkeit, den Prozess den aktuellen Gegebenheiten anzupassen ([XP Übersicht und Bewertung]).
- Erfahrenes Team: XP empfiehlt, dass die Entwickler Erfahrung haben. Sie müssen keine Profis sein, sollten aber auch nicht als Anfänger agieren.
- Disziplin: Um den gesamten XP-Entwicklungsprozess nicht zu gefährden, ist das Arbeiten unter Einhaltung einer strikten Disziplin von Seiten der Beteiligten Voraussetzung. Außer dem Coach verfügt XP über kein Instrumentarium, um einen Mangel an Disziplin auszugleichen ([XP Übersicht und Bewertung]).
- Automatisierbare Tests: Bei XP muss die Entwicklungsumgebung die Möglichkeit anbieten, Tests mit geringem Aufwand und auch in kurzer Zeit durchzuführen (Automatisierungstest z.B. NUnit Tool für Visual Studio, JUnit für Java).
- EDV-Umgebung: Tägliche Integration und Plattformwechsel (von Development auf Test und Produktion) sollten nicht durch aufwändige Prozesse erschwert sein.

Zudem muss häufiges Kompilieren möglich sein, da sonst das Feedback ausfällt ([XP Übersicht und Bewertung]).

- Das Arbeitszimmer muss nach den Vorschlägen des XP eingerichtet sein, beispielsweise benötigt ein Programmiererpaar einen großen Tisch. Mit rollenden Stühlen bewegen sich die Entwickler schnell von einem PC zum anderen.
- Konfigurationsmanagement-Tools für die Verwaltung der vielen Releases bzw. das inkrementelle Vorgehen.
- Verwenden der objektorientierten Programmiersprachen macht das Refaktorisieren durch Vererbungsmechanismus leicht.
- Ein XP-Experte soll Mitglied im Projektteam sein.
- Jedes Mitglied soll lernbereit und teamfähig sein.

2.7 Rollen in einem XP-Projekt

Um ein XP-Projekt erfolgreich durchzuführen, sollten die folgenden Rollen abgedeckt sein: Innerhalb des Projekts werden die Entscheidungsbefugnisse klar definiert. Geschäftsleute sind verantwortlich für die geschäftlichen Entscheidungen, die Entwickler für die technischen Fragen. So trifft jede Seite nur die Entscheidungen in jenen Bereichen, in denen sie kompetent ist ([XP Übersicht und Bewertung]).

2.7.1 **Kunde**

Vom Kunden wird beim XP sehr viel erwartet. Einerseits muss er vor Ort sein, damit er den Entwicklern kompetent zur Seite stehen und die Entwicklungsziele vorgeben kann, andererseits muss er seine Anforderungen als einfache User-Stories ohne große Komplexität auf Kärtchen, den so genannten Story-Cards, formulieren und den Entwicklern erklären können. Diese User-Stories sind nachher vom Kunden zu priorisieren, wobei er die Reihenfolge der Implementierung festzulegen hat. Zudem muss der Kunde Akzeptanztests formulieren und die Resultate der Programmierer beurteilen können. Nebst seiner zeitlich hohen Beanspruchung muss der Kunde auch im Sinne einer offenen, ehrlichen Kommunikation kritikfähig sein. Schwierig für den Kunden ist die Tatsache, das Projekt beeinflussen, aber nicht kontrollieren zu können. Wenn aber der

Kunde die Erfahrung macht, dass die Entwickler nicht das fertigstellen, was sie ihm versprechen, oder dass das, was sie fertigstellen, zu fehlerhaft ist, fehlen die Voraussetzungen für die Durchführung von XP ([Beck, Kent 2003], [XP Übersicht und Bewertung]).

2.7.2 **Entwickler bzw. Programmierer**

Die Entwickler müssen in einem hohen Maß teamfähig sein, da gerade das Entwickeln in Paaren eine hohe Sozialkompetenz voraussetzt. Dies ist insbesondere wichtig, da in XP der Programmierer neben der eigentlichen Programmierung auch für die Aufwandschätzung und für das Erstellen der Tests verantwortlich ist, also Aktivitäten, die immer auch mit den anderen Personen koordiniert werden müssen. Einzelgänger, die nicht über ihre Arbeit diskutieren können oder nicht kritikfähig sind, eignen sich nicht für ein XP-Team. Der Teamerfolg muss in jedem Moment über dem eigenen Erfolg stehen. Ängste, sich zu blamieren, veraltete Ansichten zu vertreten, nutzlos zu werden oder den hohen Anforderungen nicht zu genügen, müssen überwunden werden, um erfolgreich in einem XP-Team mitarbeiten zu können. Dies ist gerade für die Programmierer, die nicht immer besonders kommunikativ sind, eine besondere Herausforderung ([Beck, Kent 2003], [XP Übersicht und Bewertung]).

2.7.3 **Coach**

Der Coach ist für den Prozess im Ganzen und die Disziplin im Projekt zuständig. Er erkennt, wenn einzelne Beteiligte eine falsche Richtung einschlagen und macht das Team darauf aufmerksam. Ohne zu bevormunden, muss er das Team führen, was viel Erfahrung und Feingefühl voraussetzt. Die Nichteinhaltung der XP-Prinzipien würde den gesamten XP-Prozess gefährden, weshalb die Rolle des Coach eminent wichtig ist. Er muss zudem entscheiden, wann und wie oft er intervenieren soll, da zu starke Führung das Team wiederum unselbstständig werden lässt ([Beck, Kent 2003], [XP Übersicht und Bewertung]).

2.7.4 **Terminmanager**

Der Tracker verfolgt ständig den Projektstatus, sammelt Informationen über den geleisteten Aufwand der einzelnen Personen und vergleicht ihn mit den geplanten Werten. Er fragt die Entwickler und Entwicklerinnen immer wieder, wie viel Zeit sie schon für diese Aufgabe aufgewendet haben und wie lange sie noch zur Fertigstellung der Aufgabe brauchen. Die effektiv benötigte Zeit wird dann mit der geschätzten Dauer verglichen und Abweichungen werden begründet. Falls unerwartete Probleme auftreten, ist es Aufgabe des Trackers, zu reagieren und Gegenmaßnahmen einzuleiten. Da er selber dafür verantwortlich ist, welche Kennzahlen er wann auswertet und da er selber zu entscheiden hat, wann eine Kennzahl einen kritischen Wert erreicht hat, ist sein Gespür und sein Verantwortungsbewusstsein maßgeblich für den Projekterfolg verantwortlich. Die Schwierigkeit der Tracker-Rolle besteht darin, die Informationen bei den Entwicklern zu sammeln, ohne den Arbeitsablauf unnötig zu stören ([Beck, Kent 2003], [XP Übersicht und Bewertung]).

2.7.5 **Projektmanager**

Da die wichtigen Entscheidungen vom Kunden und vom Team gleichermaßen getroffen werden, ist die Rolle des Projektmanagers eher im Hintergrund angesiedelt. Seine Aufgabe ist es, die Projektergebnisse zu präsentieren und sie allenfalls gegenüber Vorgesetzten zu verteidigen. Es kann vorkommen, dass er für Fehler kritisiert wird, für die er nicht verantwortlich ist, da das Team relativ große Freiheit hat. Der Projektmanager ist auch für das zwischenmenschliche Klima im Projekt zuständig. Dieses sollte sich durch Angstfreiheit auszeichnen, Fehler sollten als Chancen für den Lernprozess begriffen werden ([Beck, Kent 2003], [XP Übersicht und Bewertung]).

2.7.6 **Tester**

Die Rolle des Testers ist eine weniger zentrale, da der größte Testaufwand, die Unit-Tests zu definieren und zu implementieren, von den Entwicklern durchgeführt wird. Aufgabe des Testers ist es, den Kunden zu unterstützen, indem er mit diesem zusammen die

Akzeptanztests formuliert und durchführt ([Beck, Kent 2003], [XP Übersicht und Bewertung]).

2.7.7 **Berater**

Der Berater spielt eher eine Nebenrolle im XP und wird nur sporadisch benötigt. XP-Projekte benötigen eher viele Generalisten und wenige Spezialisten. Durch das Pair-Programming mit wechselnden Partnern erfolgt bereits ein breiter Wissensaustausch. Aufgrund der Tatsache, dass quasi jeder alles macht, damit alle den Überblick bewahren, spezialisieren sich die Programmierer nicht auf ein besonderes Themengebiet. Deshalb kann es sein, dass zur Lösung sehr spezieller Probleme ein zusätzlicher Berater hinzugezogen werden muss. Der Berater erarbeitet zusammen mit den Entwicklern Lösungen und ist Ansprechpartner für technische Problemstellungen, die das Fachwissen der einzelnen Entwickler übersteigen ([Beck, Kent 2003], [XP Übersicht und Bewertung]).

3 XP implementieren

In diesem Kapitel beschreibe ich die 12 Praktiken von XP. Für jede Praktik erkläre ich, wie sie in meinem Projekt umgesetzt wurde, so dies möglich war.

Die zwei Hauptschritte für jedes XP-Projekt sind

- Planung
- Entwicklung (Design, Programmieren und Testen)

Die 12 Praktiken von XP können nach diesen beiden Schritten klassifiziert werden.

3.1 Planung

Bei XP wird geplant, selbst wenn die Anforderungen am Anfang des Projektes nicht in einem Dokument festgehalten werden und die Flexibilität im Umgang mit Kundenanforderungen oberste Priorität hat. Damit dies gelingt, wird nicht nur einmal umfangreich am Anfang geplant, wie in vielen anderen Prozessmodellen, sondern kontinuierlich von Iteration zu Iteration (siehe Abbildung 3.1.1). Wenn es sein muss, tageweise oder sogar mehrfach am Tag ([XP Übersicht und Bewertung]).

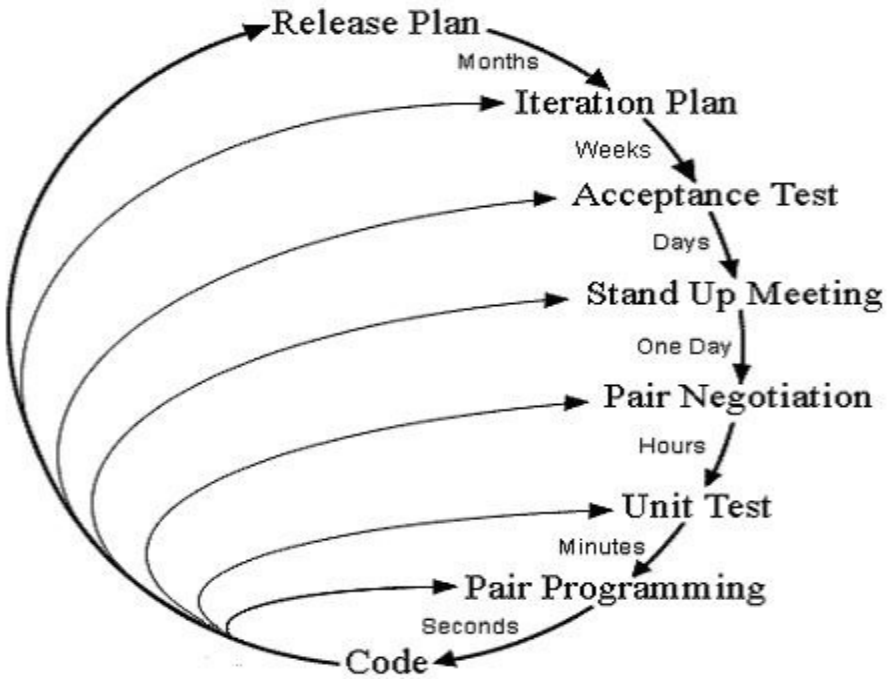


Abbildung 3.1.1: Planning /Feedback Loops ([Wikipedia])

Ein XP-Projekt wird in eine Serie von „Releases“ (Versionen) aufgeteilt. Unter einem Release versteht man eine innerhalb einer festgesetzten Zeit implementierte Anzahl von Software-Leistungsmerkmalen, die ein geschäftliches Erfordernis (Geschäftswerte) erfüllen. Die Entwicklungsdauer beträgt gewöhnlich zwischen einem bis drei Monaten. Danach wird dieses Stück Software dem Kunden übergeben ([NeMa 2001]).

Die Planung im XP passiert auf vier Ebenen.

1. Projektplanung
2. Iterationsplanung
3. Entwicklungsplanung
4. Codemanagement

3.1.1 **Projektplanung**

Die Projektplanung findet immer am Anfang jedes Release-Zyklus statt (Abbildung 3.1.1). Der Kunde schreibt seine Wünsche an die Software kurz auf Kärtchen (Story-Cards, siehe Abbildung 3.1.1.1), welche die User-Stories darstellen. Auf jedes Kärtchen kommt ein Leistungsmerkmal. Folglich schätzen die Programmierer für jedes dieser "Geschichtskärtchen" deren Schwierigkeitsgrad in idealen Entwicklungstagen ab, sodass sich der Kunde ein Bild davon machen kann, wie lange die Implementierungen der einzelnen Leistungsmerkmale dauern könnten. Die ideale Entwicklungszeit ist nicht die Zeit, die ein einzelner Programmierer aufzuwenden bereit ist (Entwicklergeschwindigkeit), sondern nur die unter idealen Bedingungen stattfindende Zeit. Unterbrechungen durch Meetings, Urlaub oder andere Faktoren sind hier nicht berücksichtigt. Da die Leistungsfähigkeit der Programmierer begrenzt ist, kann es vorkommen, dass nicht alle Wünsche implementierbar bzw. zu umfangreich für ein vorgegebenes Datum sind. In diesem Fall muss sich der Kunde für eine Anzahl wichtigster „Kärtchen“ entscheiden. Eine Alternative wäre, größere Geschichten in kleinere Geschichtchen aufzuteilen. Anzufügen ist, dass es sich hier nur um eine Vorauswahl durch den Kunden handelt. Die definitive Geschichtszuteilung an die Programmierer erfolgt erst in der nächsten „***Verpflichtungsphase***“. Diesen bis hierher

besprochenen Handlungsprozess des Schreibens, Schwierigkeitsschätzens und Aufteilens von Geschichten nennt man ***„Erforschungsphase“*** ([Sini 2003]).

Abbildung 3.1.1.1: Story -Card([Sini 2003])

In der ***Erforschungsphase*** sehen wir bereits, wie zwei wichtige Werte von XP zum Zug kommen: Einerseits die Kommunikation und andererseits die ehrliche Aufwandsabschätzung.

Verpflichtungsphase*:* Nach der *Erforschungsphase* weiß der Kunde, wie lange der Zyklus (die Version) dauert. Anhand dieser Information kann er nun seine User-Stories priorisieren und entscheiden, welche er in diesem Release implementiert haben will.

Auch in dieser Phase legt die Geschäftsseite das Datum der nächsten Version fest und die Entwicklung verpflichtet sich dazu, dieses einzuhalten. Die Verpflichtungsphase umfasst vier Spielzüge ([Beck, Kent 2000]):

1. *Nach Wert sortieren*: Die Geschäftsseite teilt die Story-Cards auf drei Stapel auf:
    - Die Leistungsmerkmale, ohne die das System nicht funktionieren kann (Basis -Komponenten) ([Beck, Kent 2000]).
    - Die Leistungsmerkmale, die weniger wichtig sind, aber entscheidenden Geschäftsgewinn liefern ([Beck, Kent 2000]).
    - Die Leistungsmerkmale, die wünschenswert, aber nicht unbedingt erforderlich sind ([Beck, Kent 2000]).
2. *Nach Risiko sortieren:* Die Entwicklung teilt die Story-Cards auf drei Stapel auf:

- Die Leistungsmerkmale, die sie genau einschätzen kann ([Beck, Kent 2000]).
- Die Leistungsmerkmale, die sie halbwegs einschätzen kann ([Beck, Kent 2000]).
- Die Leistungsmerkmale, die sie überhaupt nicht einschätzen kann (in diesem Fall muss der Kunde die Story-Card in zwei oder mehr Story-Cards aufteilen.) ([Beck, Kent 2000]).

3. *Geschwindigkeit festlegen:* Die Entwicklung teilt der Geschäftsseite mit, wie schnell das Team in idealer Entwicklungszeit pro Kalendermonat programmieren kann ([Beck, Kent 2000]).
4. *Umfang festlegen*: Die Geschäftsseite wählt die Story-Cards für eine Version aus, indem sie entweder ein Datum festgelegt, an dem die Entwicklung abgeschlossen sein muss und Karten anhand der Aufwandsschätzung und der Projektgeschwindigkeit auswählt oder indem sie Story-Cards auswählt und das Datum berechnet ([Beck, Kent 2000]).

Die letzte Phase des Planungsspiels heißt ***Steuerungsphase***. In ihr wird der Versionsumfang in weitere kleinere, ein bis drei Wochenportionen unterteilt, den sogenannten Iterationen. Die darin enthaltenen wichtigsten Geschichten werden vom Kunden vor jeder Iteration neu festgelegt und von den Programmierern adaptiert.
Während der Implementierungsphase kann es hierbei wegen überschätztem Tempo oder Einführung neuer Leistungsmerkmale zu Plankorrekturen kommen. Eine beidseitige Neueinschätzung der Geschichten wie in der Erforschungsphase ist damit unumgänglich. Die drei besprochenen Phasen (Erforschungsphase, Verpflichtungsphase und Steuerungsphase) werden dabei immer zyklisch durchlaufen ([Sini 2003]).
In meinem Projekt wurde die Projektplanung wie folgt umgesetzt: Ich habe mich mit meinem Kunden zum ersten Mal für ca. zwei Stunden zusammengesetzt. Er hat mir mündlich erklärt, was die Software machen soll und ich habe das Punkt für Punkt notiert. Näheres dazu im Punkt „Zielsetzung“ im Einführungskapitel. Eine Woche später haben

wir uns noch einmal zusammengesetzt und eine Liste von Story-Cards bzw. Geschichten (ohne Sortieren nach Priorität) zusammengeschrieben.

- Die Statemaschine und ihre verschachtelten Statemaschinen anzeigen, alle States in der Statemaschine und ihre verschachtelte Statemaschinen mit allen Ausgängen anzeigen.
- Es ermöglicht dem Benutzer, die anzeigenden Ausgänge umzustellen und festzulegen, welche Ausgänge von den States in der Darstellung angezeigt werden sollen.
- Die Statemaschine und ihre verschachtelten Statemaschinen anzeigen und dabei nur die States anzeigen, die Default (Next, Prev, Esc und Error) Ausgänge haben.
- Einfache Statemaschine anzeigen: Alle States in der Statemaschine mit allen Ausgängen anzeigen und die verschachtelten Statemaschinen ignorieren.
- Nur States in der Statemaschine anzeigen, die Default Ausgänge haben.
- Ausdrucken der angezeigten graphischen Darstellung.
- Jeder State in der Statemaschine soll mit einem Quadrat in der Grafik repräsentiert werden, im Quadrat muss der Name des State eingetragen werden.
- Die Verbindung zwischen den States soll mit Linien repräsentiert werden.
- Linien zwischen den States, die nicht unmittelbar nacheinander folgen sollen, in Tooltip-Fenster anzeigen. Dieses Fenster soll Informationen enthalten, von welchem State man zu welchem State gelangen kann.
- Wenn der State in einer Statemaschine (SM1) eine andere Statemaschine (SM2) verschachtelt, muss der Name von der verschachtelten Statemaschine(SM2) ins Quadrat als Link eingetragen werden. Wenn man auf den Link klickt, bekommt man die Darstellung von den verschachtelten Statemaschinen (SM2). Man kann zwischen beiden Statemaschinen SM1 und SM2 mittels zweier Knöpfe - einer nach rechts und einer nach links - navigieren.
- Es soll eine Checkbox „Einfache Darstellung“ geben. Wenn diese aktiv ist, werden die Linien auf der rechten und linken Seite des States nicht angezeigt. Es werden ausschließlich die Linien zwischen den States die sich unmittelbar untereinander befinden aufgezeichnet.

Story Card

Titel: Alles Aufzeichen NO: 1

Release:

Beschreibung:

Die Statemachine und ihre verschachtelten Statemachinen anzeigen, alle States in der Statemachine und ihre verschachtelte Statemachinen mit allen Ausgängen anzeigen.

Datum: 01.03.2006 Abschätzung:

Schwierigkeit: Hoch

Story Card

Titel: Konfiguration NO: 2

Release:

Beschreibung:

Es ermöglicht dem Benutzer, die anzeigenden Ausgänge umzustellen und festzulegen, welche Ausgänge von den States in der Darstellung angezeigt werden sollen.

Datum: 01.03.2006 Abschätzung:

Schwierigkeit: Tief

Story Card

Titel: Einfache SM, Default Ausgänge NO: 5

Release:

Beschreibung:

Nur States in der Statemachine anzeigen, die Default Ausgänge haben.

Datum: 01.03.2006 Abschätzung:

Schwierigkeit: Tief

Story Card

Titel: SM,Verschachtelte SM, Default Ausgänge NO: 3

Release:

Beschreibung:

Die Statemachine und ihre verschachtelte Statemachinen anzeigen, nur die States die Default (Next, Prev, Esc und Error) Ausgänge haben anzeigen.

Datum: 01.03.2006 Abschätzung:

Schwierigkeit: Hoch

Story Card

Titel: State Aufzeichnen NO: 7

Release:

Beschreibung:

Jeder State in der Statemachine soll mit einem Quadrat in der Graphik repräsentiert werden, im Quadrat muss der Name von dem State eingetragen werden.

Datum: 01.03.2006 Abschätzung:

Schwierigkeit: Tief

Story Card

Titel: Einfache SM, Alle Ausgänge NO: 4

Release:

Beschreibung:

Einfache Statemachine anzeigen, alle States in der Statemachine mit allen Ausgängen anzeigen und die verschachtelte Statemachine ignorieren.

Datum: 01.03.2006 Abschätzung:

Schwierigkeit: Mittel

Story Card

Titel: Ausdrucken NO: 6

Release:

Beschreibung:

Ausdrucken der angezeigten graphischen Darstellung.

Datum: 01.03.2006 Abschätzung:

Schwierigkeit: Hoch

Story Card

Titel: Verbindung der States NO:8

Release:

Beschreibung:

Die Verbindung zwischen den States und ihre Ausgänge der States soll mit Linien repräsentiert werden.

Datum: 01.03.2006 Abschätzung:

Schwierigkeit: Tief

Story Card

Titel: ToolTip-Fenster anzeigen NO:9

Release:

Beschreibung:

An Linien zwischen den States die nicht unmittelbar nacheinander folgen soll es eine Stelle geben, wo man mit der Maus darüber fahren kann und dann ein kleines ToolTip-Fenster erscheint. Dieses Fenster soll Informationen enthalten, von welchem State man zu welchen State gelangen kann.

Datum: 01.03.2006 Abschätzung:

Schwierigkeit: Mittel

Story Card

Titel: verschachtelte SM verlinken NO:10

Release:

Beschreibung:

Wenn der State in einer Statemachine (SM1) eine andere Statemachine (SM2) verschachtelt, muss den Name von der verschachtelten Statemachine (SM2) ins Quadrat als Link eingetragen werden, wenn man auf den Link geklickt hat, bekommt die Darstellung von den verschachtelten Statemachine (SM2). Man kann zwischen beide Statemachinen SM1 und SM2 mittels zweier Knöpfe - einer nach Rechts und einer nach Links - navigieren.

Datum: 01.03.2006 Abschätzung:

Schwierigkeit: Hoch

Story Card

Titel: Einfache Darstellung-Checkbox NO:11

Release:

Beschreibung:

Es soll eine Checkbox „Einfache Darstellung" geben, wenn diese aktiv ist werden die Linien auf der rechten und linken Seite des States nicht angezeigt. Es werden ausschließlich die Linien zwischen den States die sich unmittelbar untereinander befinden aufgezeichnet.

Datum: 01.03.2006 Abschätzung:

Schwierigkeit: Mittel

Abbildung 3.1.1.2: Erster Durchgang der Geschichtsschreibung

Danach habe ich in jede Karte meine Abschätzung eingetragen und dem Kunden die Karten gegeben, damit er für jede Karte das zugehörige Release festlegen kann. Nach dieser Ergänzung sehen die Karten wie folgt aus:

Story Card

Titel: Alles Aufzeichen NO: 1

Release:

Beschreibung:

Die Statemachine und ihre verschachtelten Statemachinen anzeigen, alle States in der Statemachine und ihre verschachtelte Statemachinen mit allen Ausgängen anzeigen.

Datum: 01.03.2006 Abschätzung: 1 woche

Schwierigkeit: Hoch

Story Card

Titel: Konfiguration NO: 2

Release:

Beschreibung:

Es ermöglicht dem Benutzer, die anzeigenden Ausgänge umzustellen und festzulegen, welche Ausgänge von den States in der Darstellung angezeigt werden sollen.

Datum: 01.03.2006 Abschätzung: 1 woche

Schwierigkeit: Tief

Story Card

Titel: SM,Verschachtelte SM, Default Ausgänge NO: 3

Release:

Beschreibung:

Die Statemachine und ihre verschachtelte Statemachinen anzeigen, nur die States die Default (Next, Prev, Esc und Error) Ausgänge haben anzeigen.

Datum: 01.03.2006 Abschätzung: 2 woche

Schwierigkeit: Hoch

Story Card

Titel: Einfache SM, Alle Ausgänge NO: 4

Release:

Beschreibung:

Einfache Statemachine anzeigen, alle States in der Statemachine mit allen Ausgängen anzeigen und die verschachtelte Statemachine ignorieren.

Datum: 01.03.2006 Abschätzung: 1 woche

Schwierigkeit: Mittel

Story Card

Titel: Einfache SM, Default Ausgänge NO: 5

Release:

Beschreibung:

Nur States in der Statemachine anzeigen, die Default Ausgänge haben.

Datum: 01.03.2006 Abschätzung: 2 Wochen

Schwierigkeit: Tief

Story Card

Titel: Ausdrucken NO: 6

Release:

Beschreibung:

Ausdrucken der angezeigten graphischen Darstellung.

Datum: 01.03.2006 Abschätzung: nicht implementierbar

Schwierigkeit: Hoch

Story Card

Titel: State Aufzeichnen NO:7

Release:

Beschreibung:

Jeder State in der Statemachine soll mit einem Quadrat in der Graphik repräsentiert werden, im Quadrat muss der Name von dem State eingetragen werden.

Datum: 01.03.2006 Abschätzung: 1 Woche

Schwierigkeit: Tief

Story Card

Titel: Verbindung der States NO:8

Release:

Beschreibung:

Die Verbindung zwischen den States und ihre Ausgänge der States soll mit Linien repräsentiert werden.

Datum: 01.03.2006 Abschätzung: 1 woche

Schwierigkeit: Tief

Story Card

Titel: ToolTip-Fenster anzeigen NO:9

Release:

Beschreibung:

An Linien zwischen den States die nicht unmittelbar nacheinander folgen soll es eine Stelle geben, wo man mit der Maus darüber fahren kann und dann ein kleines ToolTip-Fenster erscheint. Dieses Fenster soll Informationen enthalten, von welchem State man zu welchen State gelangen kann.

Datum: 01.03.2006 Abschätzung: 1 Woche

Schwierigkeit: Mittel

Story Card

Titel: verschachtelte SM verlinken NO:10

Release:

Beschreibung:

Wenn der State in einer Statemachine (SM1) eine andere Statemachine (SM2) verschachtelt, muss den Name von der verschachtelten Statemachine (SM2) ins Quadrat als Link eingetragen werden, wenn man auf den Link geklickt hat, bekommt die Darstellung von den verschachtelten Statemachine (SM2). Man kann zwischen beide Statemachinen SM1 und SM2 mittels zweier Knöpfe - einer nach Rechts und einer nach Links - navigieren.

Datum: 01.03.2006 Abschätzung: 3 Wochen

Schwierigkeit: Hoch

Story Card

Titel: Einfache Darstellung-Checkbox NO:11

Release:

Beschreibung:

Es soll eine Checkbox „Einfache Darstellung" geben, wenn diese aktiv ist werden die Linien auf der rechten und linken Seite des States nicht angezeigt. Es werden ausschließlich die Linien zwischen den States die sich unmittelbar untereinander befinden aufgezeichnet.

Datum: 01.03.2006 Abschätzung: 1 Woche

Schwierigkeit: Mittel

Abbildung 3.1.1.3: Zweiter Durchgang der Geschichtsschreibung

In der Abbildung 3.1.1.3 im oberen Bereich sieht man, dass das Feature auf Karte Nr. 6 (Ausdrucken) nicht implementierbar ist. Im Folgenden ist die Zusammenfassung für die Abbildung 3.1.1.3 beschrieben:

Das Aufteilen der Karten in Releases erfolgte wie unten beschrieben:

- *Story Nr. 5*: Nur States in der Statemaschine anzeigen, die Defaultausgänge (Next, Prev, Esc und Error Ausgang) haben.
- *Story Nr.7*: Jeder State in der Statemaschine soll mit einem Quadrat in der Grafik repräsentiert werden, in das Quadrat ist der Name des State einzutragen.
- *Story Nr.8*: Verbindungen zwischen den States sollen mit Linien repräsentiert werden.

Release 1

- *Story Nr.4*: Einfache Statemaschine anzeigen: Alle States in der Statemaschine mit allen Ausgängen anzeigen und die verschachtelten Statemaschinen ignorieren.
- *Story Nr.3*: Die Statemaschine und ihre verschachtelten Statemaschinen anzeigen, wobei nur die States die Defaultausgänge (Next, Prev, Esc und Error) haben, angezeigt werden sollen.
- *Story Nr.10*: Wenn der State in einer Statemaschine (SM1) in eine andere Statemaschine (SM2) verschachtelt wird, muss der Name von den verschachtelten Statemaschinen (SM2) ins Quadrat als Link eingetragen werden. Wenn man auf den Link geklickt hat, bekommt man die Darstellung von den verschachtelten Statemaschinen (SM2). Man kann zwischen den beiden Statemaschinen SM1 und SM2 mittels zweier Knöpfe - mit einem nach rechts und mit einem nach links - navigieren.

Release 2

- *Story Nr.1*: Die Statemaschine und ihre verschachtelte Statemaschinen anzeigen, alle States in der Statemaschine und ihre verschachtelten Statemaschinen mit allen Ausgängen anzeigen.
- *Story Nr.2*: Es ermöglicht dem Benutzer, die anzeigenden Ausgänge umzustellen und festzulegen, welche Ausgänge von den States in der Darstellung angezeigt werden sollen.
- *Story Nr.9*: Linien zwischen den States, die nicht unmittelbar nacheinander folgen, sollen Tooltip-Fenster anzeigen. Dieses Fenster soll Informationen enthalten, von welchem State man zu welchem State gelangen kann.
- *Story Nr.11*: Checkbox „Einfache Darstellung“.
- *Story Nr.6*: Ausdrucken der angezeigten grafischen Darstellung.

Release 3

Meine Zeitabschätzung war wie folgt:
Story Nr.5 dauert 2 Wochen; Story Nr.7 dauert 1 Woche; Story Nr.8 dauert 1 Woche
à Zeitaufwand für Release ist ein Monat.

Story Nr.4 dauert 1 Woche; Story Nr.3 dauert 2 Wochen; Story Nr.10 dauert 3 Wochen
à Zeitaufwand für Release 2 ist sechs Wochen.

Story Nr.1 dauert 1 Woche; Story Nr.2 dauert 1 Woche; Story Nr.9 dauert 1 Woche; Story Nr.11 dauert 1 Woche
à Zeitaufwand für Release 3 ist vier Wochen.

Wie oben angeführt haben wir das ganze Projekt (alle User-Stories) in drei Releases aufgeteilt. Beim ersten Release werden Story 5, 7 und 8 implementiert, beim zweiten Release werden Story Nr. 4, 2 und 10 und beim dritten Release Story Nr. 1, 2, 9 und 11 implementiert.
Bei der Abschätzung habe ich die logische Sortierung der Aufgaben betrachtet. So kommt beispielsweise die Karte Nr.1 sicher nach der Implementierung der Karten Nr.4 und 5.

Deswegen benötigt die Abschätzung von Karte Nr. 7 nur eine Woche, obwohl die Aufgabe schwierig ist.

Als nächstes habe ich einen Prototyp händisch aufgezeichnet. Ein Beispiel für die Aufzeichnung einer Statemaschine sehen Sie in Abbildung 3.1.1.4. In der Folge wurde dem Kunden ein Prototyp angezeigt und mit ihm durchgesprochen. Die Änderungen bzw. die Kundenwünsche waren folgende: Alle States, die den „Next“ Ausgang haben, müssen als erstes nacheinander aufgezeichnet werden. Unmittelbar nach jedem State mit „Next“ Ausgang muss sein „Next“ Ausgang aufgezeichnet werden. Das heißt, die Änderung der Abbildung 3.1.1.4 kann wie folgt beschrieben werden: Der State S5 kommt unmittelbar nach dem State S3, der „Next“ Ausgang von S3 geht nach S5. Nach dem State S5 kommt S6, da der Ausgang von State S5 State S6 ist. Aus demselben Grund kommt S7 nach S6. State S4 und S8 kommen ganz am Ende des Diagramms, da sie keinen „Next“ Ausgang haben (Abbildung 3.1.1.5).

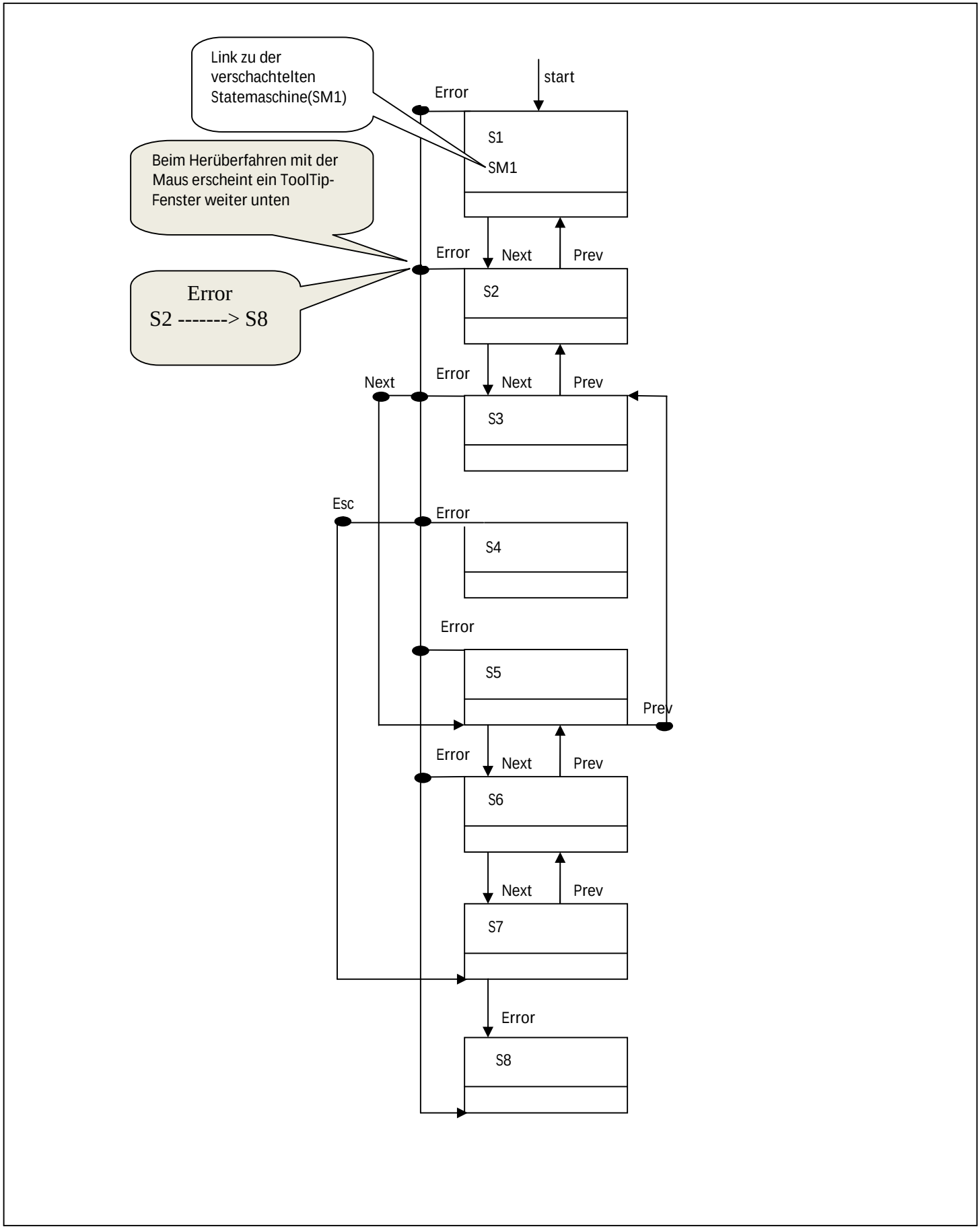


Abbildung 3.1.1.4: Der erste Prototyp für die Darstellung der Statemaschine

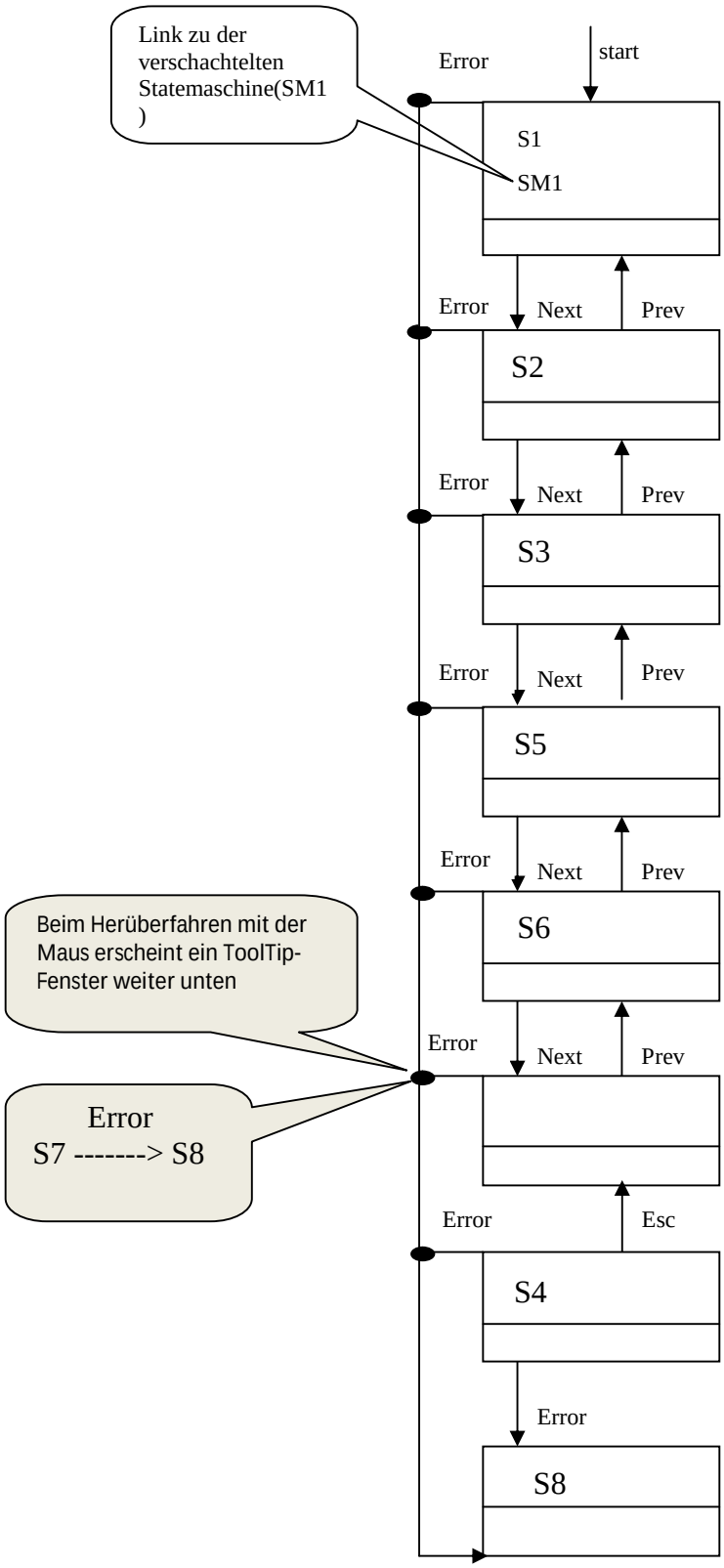


Abbildung 3.1.1.5: Der zweite Prototyp für Darstellung der Statemaschine

3.1.2 **Iterationsplanung**

Um gegenüber Änderungen flexibel zu bleiben, werden die Releases in mehrere kürzere Iterationen mit einer Dauer von ein bis drei Wochen unterteilt. Auch bei den Iterationen wird eine Iterationsplanung aufgestellt ([Jeffries 2001]).

Die Regeln in einer Iteration sind ähnlich wie beim Planungsspiel, und sie treffen nur die Entwickler. Bei Iterationen kommen die gleichen drei Phasen (Erforschung, Verpflichtung und Steuerung) vor.

Erforschungsphase: In dieser Phase schreiben die Programmierer (mit eventueller Hilfe vom Kunden) keine Geschichten mehr, sondern „Aufgaben“ (engl.: *tasks*). Eine Geschichte wird in mehrere kleinere Aufgaben unterteilt, welche ihrerseits zu Iterationen gruppiert werden. Eine Geschichte kann in einer oder mehreren Iterationen gelöst werden, das hängt von der Schwierigkeit der Aufgabe ab (eine schwierige Aufgabe bedeutet mehrere Iterationen). In den seltensten Fällen wird eine ganze Geschichte aus nur einer Iteration bestehen. Ein ganzes Leistungsmerkmal ist meistens nicht in wenigen Tagen oder Wochen implementierbar. Gelegentlich umfasst eine Aufgabe mehrere Leistungsmerkmale. Manchmal bezieht sie sich auch auf gar keines, sondern dient nur als Hilfsmittel zu einer Funktion. Im Folgenden gehen die Programmierer genau umgekehrt vor wie in der Erforschungsphase des Planungsspiels: Zuerst kommt die Auswahl der wichtigsten Aufgaben und erst anschließend deren Schwierigkeitsabschätzung. Große Aufgaben teilt man dabei genauso in kleinere Aufgaben, wie es der Kunde im Planungsspiel mit den story-cards macht ([Sini 2003]).

Verpflichtungsphase: In dieser Phase werden die Aufgaben zugeteilt. Um die Wahrscheinlichkeit zu erhöhen, dass eine Geschichte beendet werden kann, sollte ein Programmierer nur Aufgaben innerhalb einer Geschichte und nicht aus verschiedenen Geschichten auswählen. Noch besser ist es, wenn sich ein Programmierer oder ein Programmiererpaar für alle Aufgaben einer einzelnen Geschichte entscheiden.

Neu beim Iterationsspiel – verglichen mit Planungsspiel – ist der Begriff des *Belastungsfaktors*. Er gibt das Verhältnis zwischen der Implementierungsgeschwindigkeit eines Programmierers und der Aufgabenschwierigkeit einer Iteration wieder. Besteht eine zweiwöchige Iteration aus Aufgaben im Umfeld von z.B. 10 idealen Entwicklungstagen,

und schafft ein Programmierer aus Erfahrung nur 6, ergibt das den Belastungsfaktor von 6/10=0.6. Der mögliche Aufgabenumfang der nachfolgenden Iteration sollte folglich mit 0.6 multipliziert werden, falls sich dieser Wert im Verlaufe des Projekts nicht drastisch ändert ([Sini 2003]).

Steuerungsphase: In dieser erscheinen die vier Arbeitsschritte „Programmieren“, „Testen“, „Zuhören“ und „Design entwerfen“. Der Programmierer übernimmt eine Aufgabe, sucht sich einen Partner, mit dem er zuerst gemeinsam Tests für die Aufgabe (Unit-Tests) schreibt. Folglich implementieren sie die eigentliche Aufgabe, um die Tests zu erfüllen und geben am Ende den Code eventuell an Kollegen zur erneuten Inspektion weiter. Die Freigabe einer Aufgabe oder Geschichte erfolgt erst nach Bestehen aller Unit- bzw. Akzeptanztests (Funktionstest) auf einem Integrationsrechner ([Sini 2003]).

In der Abbildung 3.1.2.1 wird der Unterschied zwischen traditionellen Softwareentwicklungsmethoden und XP bezüglich Iterationen dargestellt ([Schwartzhaupt 2003]).

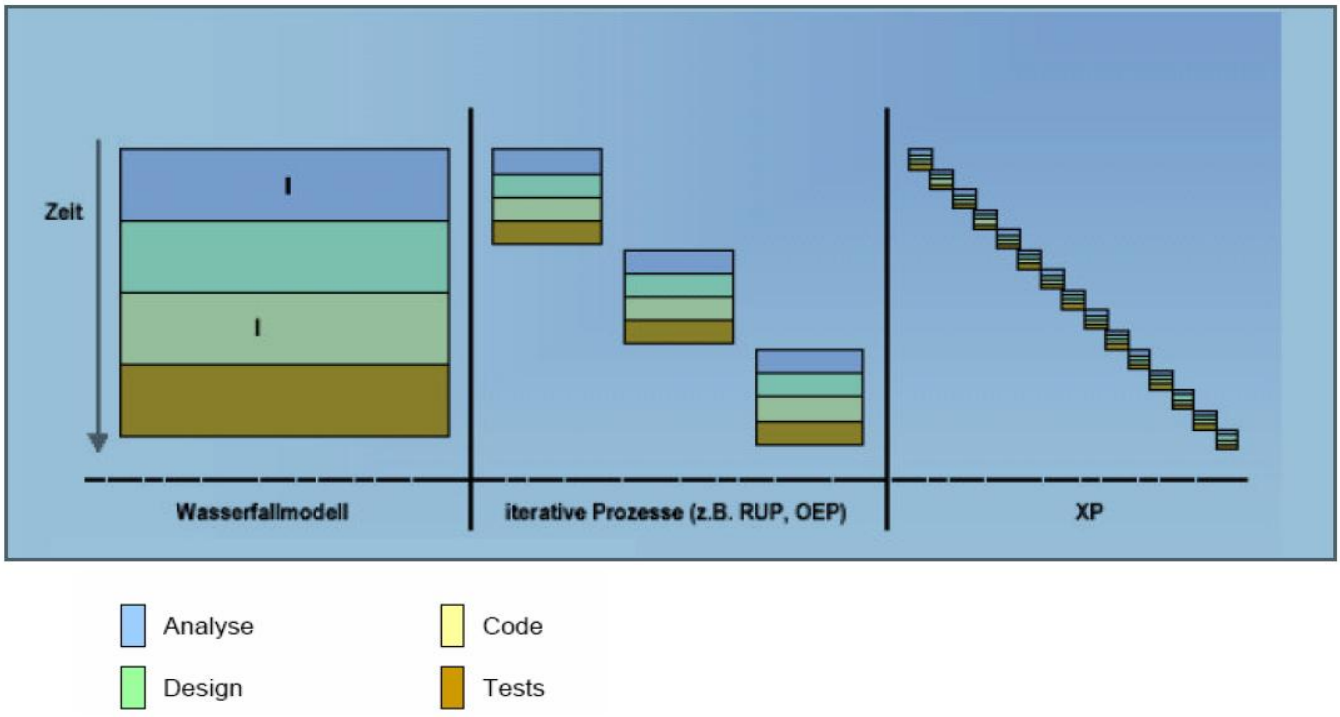


Abbildung 3.1.2.1: Iterationszyklen ([XP Übersicht und Bewertung])

Klassische Verfahren wie z.B. das Wasserfallmodell nehmen erst eine zeitlich und mengenmäßig umfangreiche Analyse vor. Danach folgen ebenso umfangreich Design,

Implementieren und Testen des Programms, bis überhaupt das Erstrelease entsteht. Iterative Prozessmodelle, wie z.B. der Rational Unifed Process (RUP), kennen kleinere Schritte und mehrere Iterationen mit lauffähigen Zwischenreleases, zu denen der Kunde sein Feedback geben kann, um das endgültige Release gemäß seinen Wünschen zu gestalten. XP kennt sehr viele Iterationen mit lauffähigen Zwischenreleases. Das jeweilige Feedback des Kunden wird sofort umgesetzt und er sieht die Software seinen Wünschen entsprechend „reifen“ ([XP Übersicht und Bewertung]).

Ich habe die *Iterationsplanung* wie folgt verwendet:
Wir (der Kunde und ich) haben vereinbart, dass wir uns einmal pro Woche zusammensetzen, damit der Kunde einen Einblick bekommt, was schon implementiert ist und was nicht, welche Probleme auftreten, usw. Es soll auch festgelegt werden, was in der nächsten Woche gemacht werden soll. Entsprechend ist für die Iteration ein Zeitraum von einer Woche anberaumt. In jedem Release werden die Aufgaben – sofern diese eine bestimmte Größe überschreiten – in mehrere kleinere Aufgaben aufgeteilt.
Während der Arbeit an Release 1 habe ich die Geschichte 5 (siehe Abbildung 3.1.1.3) beispielsweise in drei Aufgaben untergeteilt:

- 5a “Statemaschine-Info“: Implementieren einer Klasse: Die Klasse bekommt die Statemaschine als Datentyp „Object“, mittels des „Reflektions“-Mechanismus wird ein Objekt davon instanziiert. Anhand vom instanziierten Objekt werden alle benötigten Daten für die Darstellung extrahiert, z.B. Next States, Esc States, die Ausgänge von den einzelnen States, …
- 5b „Ausgänge Sortieren“: Alle States in der Statemaschine und ihre Ausgänge sortieren, sodass alle States, die einen „Next“ Ausgang haben, zuerst aufgezeichnet werden. Die States werden sortiert, sodass der „Next“ Ausgangsstate unmittelbar unter dem Vorgängerstate aufgezeichnet wird.
- 5c „Quadratmessung Festlegen“: Die Abmessungen vom Quadrat, welches die States repräsentiert, festlegen.

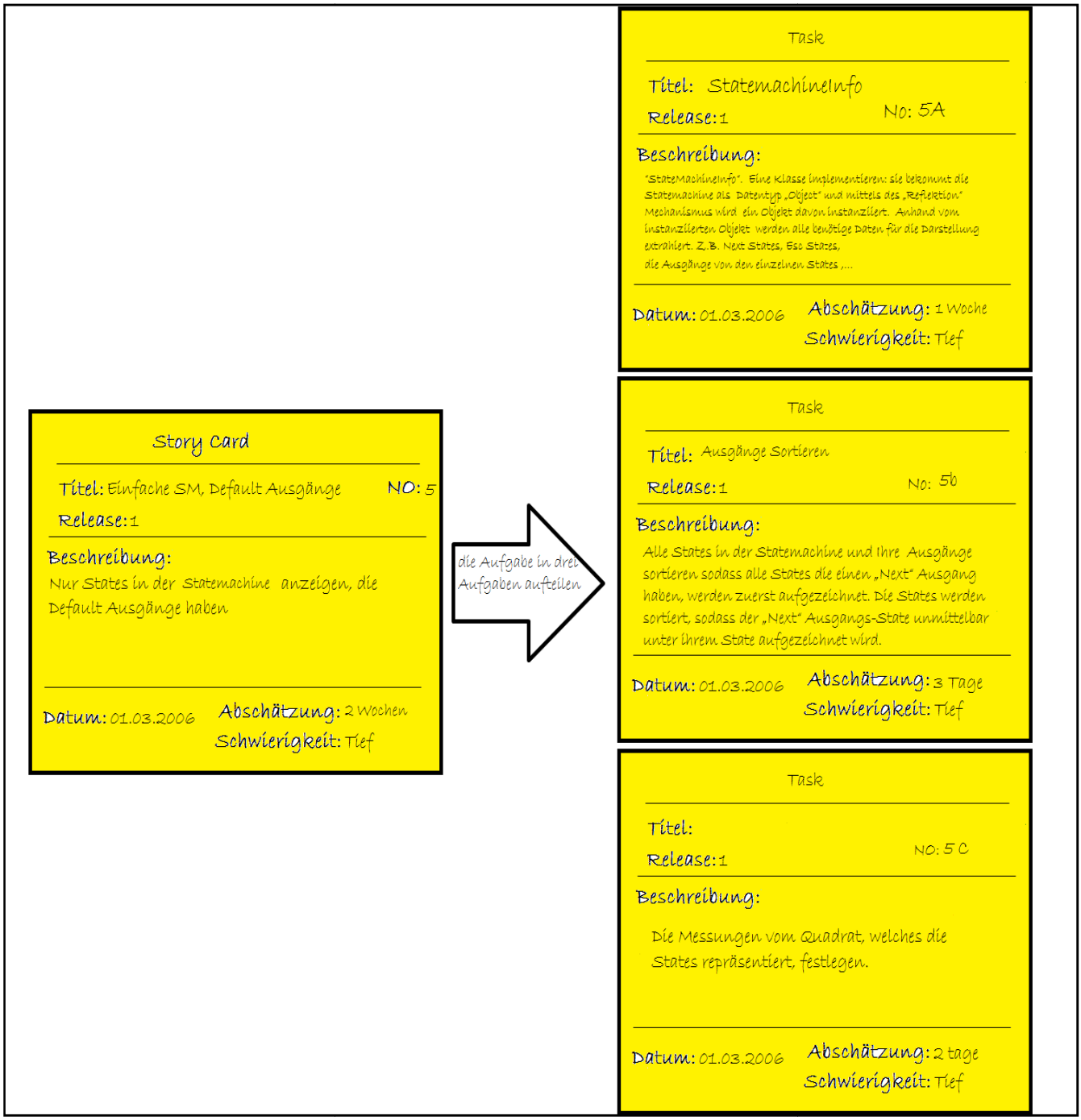


Abbildung 3.1.2.2: Aufteilung der Aufgabe 5 in drei kleine Aufgaben

3.1.3 **Kurze Releasezyklen**

Jede Version sollte möglichst klein gehalten werden und gleichzeitig die wertvollsten Geschäftsanforderungen erfüllen. Die Version muss als Ganzes sinnvoll sein, d.h. man kann ein Leistungsmerkmal nicht halb implementieren und schon ausliefern, um den Entwicklungszeitraum zu verkürzen.

Viel besser ist es, jeweils nur einen oder zwei Monate im Voraus zu planen als sechs Monate oder ein Jahr. Eine Firma, die ihren Kunden eine umfangreiche Software liefert, kann diese Software möglicherweise nicht häufig durch neue Versionen aktualisieren. Trotzdem sollte auch in diesem Fall versucht werden, den Entwicklungszyklus zu verkürzen [Beck, Kent 2000].

3.1.4 **Metaphern im Planen**

Schon bei der Planung ist eine offene und ehrliche Kommunikation von zentraler Bedeutung. Ebenfalls wichtig ist ein gemeinsames Vokabular, wobei bei XP besonders Metaphern zum Zuge kommen. Mittels eines für alle verständlichen Bildes werden komplizierte Sachverhalte vereinfacht beschrieben und damit eine gemeinsame Sprache während des gesamten Projektverlaufs geschaffen. Gute Metaphern zu finden ist nicht einfach, aber sind sie einmal vorhanden, erleichtern sie die Kommunikation erheblich, was wiederum die beiden XP-Werte Kommunikation und Einfachheit betrifft. Sinnvolle und leicht zu handhabende Metaphern werden sehr oft mittels Brainstorming oder anderer Kreativitätstechniken gefunden ([Beck, Kent 2003], [XP Übersicht und Bewertung]).

In meinem Projekt haben wir die Wörter „Statemaschine“ (Abkürzung SM) und „State“ (Abkürzung S) verwendet.

3.2 Entwicklung

An dieser Stelle wird erläutert, wie XP den Entwicklungsprozess wie z.B. Entwicklungsdesign, Programmieren und Testen (Unit-Test, User-Test) behandelt und welche Aktivitäten bei Design, Programmieren und Testen ausgeführt werden.

3.2.1 **Design**

Das Design in XP geht von einem einfachen, mit Metaphern unterstützten, Design aus, um die sich ändernden Kundenwünsche effektiv und bildhaft realisieren zu können. Damit dies auch effizient geschehen kann, wird konsequent ein Refactoring eingesetzt.

3.2.1.1 Einfaches Design

Bei XP geht man davon aus, dass sich die Kundenanforderungen häufig ändern. Deshalb wird nicht versucht, das ideale Design schlechthin zu entwerfen, das alle künftigen Ausbaumöglichkeiten berücksichtigt. Das einfachste Design ist bei XP durch folgende vier Regeln definiert ([XP Übersicht und Bewertung]):

1. Einfaches Design muss alle Tests bestehen, die die Umsetzung der Kundenanforderungen validieren.
2. Das Design muss jede Idee ausdrücken, die der Entwickler zum Ausdruck bringen muss.
3. Kein Programmcode darf doppelt vorhanden sein.
4. Das einfachste Design besitzt am wenigsten Klassen und Methoden ([XP Übersicht und Bewertung]).

Je einfacher das Design ausfällt, desto einfacher werden das Testen, die Wartung und das Refactoring ([Beck, Kent 2003]).

3.2.1.2 Refaktorisieren

Der Wunsch nach Techniken zur inkrementellen Verbesserung und Modifikation von Programmcodes ist fast so alt wie die Programmierung selbst. Ziel einer transformationellen Softwareentwicklung ist die Zerlegung des Softwareentwicklungsprozesses in kleine, systematisch durchführbare Schritte, die aufgrund lokaler Anwendung überschaubar werden (siehe Abbildung 3.2.1.2.2).

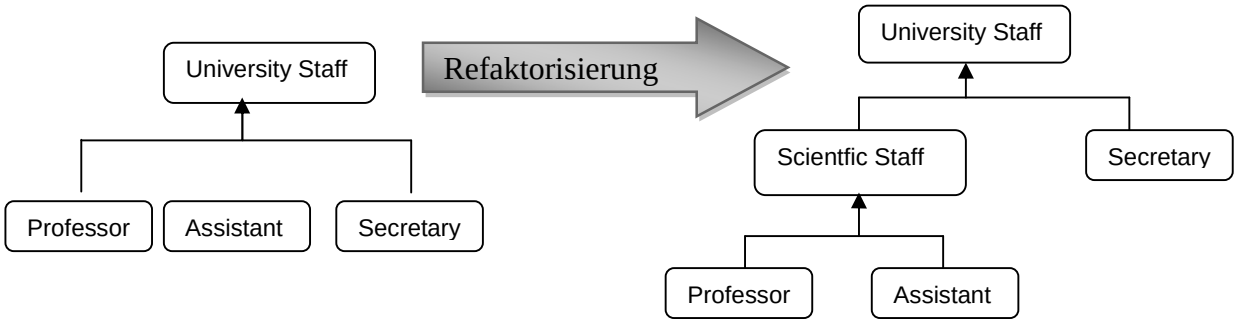


Abbildung 3.2.1.2.1: Restrukturierung einer Klassenhierarchie als korrektheitserhaltendes Refactoring

Wenn die Entwickler merken, dass die Regeln des einfachen Designs verletzt worden sind, wenden sie ein Refactoring an. Ziel das Refactoring ist die kontinuierliche Verbesserung des Codes bzw. des Designs der Software. Falls Programmteile kompliziert oder schwer verständlich codiert wurden, verbessert das Programmiererpaar diesen Teil. Beim Refactoring werden keine neuen Funktionalitäten programmiert, sondern die Struktur des Codes wird verbessert, wie Abbildung 3.2.1.2.2 zeigt. Dieser Vorgang muss diszipliniert erfolgen und alle Unit-Tests müssen erneut durchgeführt werden, damit dabei nicht aus Versehen bestehende Funktionalität überschrieben wurde. Da die Tests aber automatisiert sind, bedeutet die wiederholte Durchführung von Tests keinen großen zusätzlichen Aufwand ([Beck, Kent 2003], [XP Übersicht und Bewertung]).

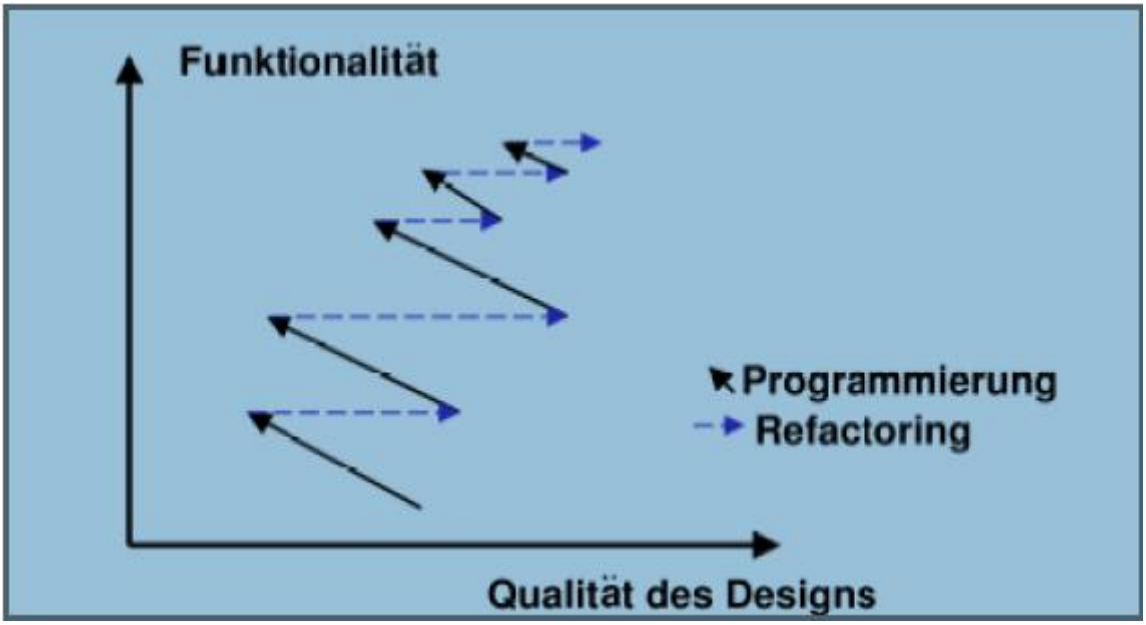


Abbildung 3.2.1.2.2: Refactoring

Abbildung 3.2.1.2.2 zeigt, wie abwechslungsweise Funktionalitäten implementiert werden und dann wieder ein Refactoring durchgeführt wird. Dies setzt die Akzeptanz des Kunden für den Prozess voraus, da bei klassischen Vorgehensmodellen bei Zeitmangel normalerweise Tests und Refactoring als erstes eingeschränkt oder sogar unterlassen werden ([XP Übersicht und Bewertung]).

Ich habe im Code viele Stellen optimiert.

Zum Beispiel :

```
protected void DrawArrowFromStateToState1( Rectangle source, Rectangle target, string label, string
egress,string targetName,string currentStateName)
{
        if (source.Y >= target.Y) //To Up
        {
                // Here a part of code, which does a specific task befor setting the
                // string s1 if  source.Y>=target.Y
                string s1 = "\n";
                for (int i = 0; i <= currentStateName.Length + 5; i++) s1 += " ";
                s1 += label + "\n";
                s1 += currentStateName +" ";
                for (int i = 0; i <= label.Length; i++) s1 += "-";
                s1 += "-> " + targetName + "\n";
                // Here a part of code, which do a specific task after set the string s1 when
                //source.Y>=target.Y
                 return;
        }
        Else
        {
                // Here a part of code, which does a specific task befor setting
                // the string s1 when source.Y<target.Y
                string s1 = "\n";
            for (int i = 0; i <= currentStateName.Length + 5; i++) s1 += " ";
            s1 += label + "\n";
            s1 += currentStateName +" ";
            for (int i = 0; i <= label.Length; i++) s1 += "-";
            s1 += "-> " + targetName + "\n";
            // Here a part of code, which do a specific task after set the string s1 when
            //source.Y<target.Y
               return;
        }
}
```

Refaktorisierung

```
protected void DrawArrowFromStateToState1( Rectangle source, Rectangle target, string label, string
egress,string targetName,string currentStateName)
{
        if (source.Y >= target.Y) //To Up
        {
                // Here a part of code, which does a specific task befor setting the
               // string s1 if  source.Y>=target.Y
                string s1 = this.CreateToolTipString(currentStateName, label, targetName);
              //Here a part of code, which do a specific task after set the string s1 when
              //source.Y>=target.Y
                return;
        }
        Else
        {
                // Here a part of code, which does a specific task befor setting
                // the string s1 when source.Y<target.Y
                 string s1 = this.CreateToolTipString(currentStateName, label, targetName);
               //Here a part of code, which do a specific task after set the string s1 when
               //source.Y<target.Y
                return;
        }
}
string CreateToolTipString(string from, string label, string to)
{
            string returnstring = "\n";
            for (int i = 0; i <= from.Length + 5; i++) returnstring += " ";
            returnstring += label + "\n";
            returnstring += from +" ";
            for (int i = 0; i <= label.Length; i++) returnstring += "-";
            returnstring += "-> " + to + "\n";
            return returnstring;
}
```

3.2.2 **Programmieren**

3.2.2.1 Programmierstandards

Das Entwicklungsteam definiert Richtlinien, wie der Code geschrieben werden soll, beispielsweise wie die Variablen, Klassen und Methoden benannt werden müssen, sodass die Namen beim Lesen besagen, was die Variablen und Klassen machen. Natürlich sollten vorhandene Standards (z.B. bei Java oder Microsoft) verwendet werden, oder man kann Templates für häufig geschriebene Klassen die selber definieren (z.B. Templates bei Visual Studio). Bei Einhaltung der Programmierstandards verstehen die Entwickler auf Standards beruhende Codes sehr viel schneller und besser.

3.2.2.2 Fortlaufende Integration

Die kontinuierliche Integration ist der Herzschlag eines Projekts. Die XP-Entwickler sollen ihren Code mehrmals täglich freigeben. Das verringert die Wahrscheinlichkeit der Kollision zweier XP-Paare. Alle Teammitglieder wissen immer, was freigegeben wurde und was nicht, sodass keine zwei Paare am selben Problem arbeiten. Es dürfen nicht zwei Programmiererpaare gleichzeitig integrieren weshalb für die Integration ein separater Rechner verwendet werden soll. So wissen die Entwickler sofort, ob sie ihre Änderungen integrieren dürfen oder ob der Integrationsrechner von einem anderen Programmiererpaar besetzt ist.

Um den Integrationsprozess richtig durchzuführen, wird zuerst der gesamte bisher freigegebene Code auf den Integrationsrechner geladen. Anschließend folgen alle neuen Freigabekandidaten, wofür man am besten ein Integrationsmanagement-Tool benützt (z.B. SVN oder SourceSafe, Team System von Microsoft). Es protokolliert sämtliche Änderungen und macht sie bei Fehlern wieder rückgängig. Ohne diese Tools verliert man leicht die Übersicht über die vielen Versionen. Nach der Integration lässt man alle Unit-Tests noch einmal auf der Integrationsmaschine laufen (natürlich müssen die Entwickler auf ihrem Rechner alle Unit-Tests laufen lassen, bevor sie auf dem Integrationsrechner arbeiten). Dieser Testvorgang nimmt zusätzlich etwa zehn Minuten Zeit in Anspruch. Funktioniert alles einwandfrei, wird der Code zur offiziellen Versionen freigegeben. Bei Problemen erfolgt eine sofortige Fehlersuche und Fehlerbehebung. Dies dürfte in der

Regel nicht allzu schwer fallen, da der Fehler bei einem der neuen Freigabekandidaten liegt. Je kürzer die Wartezeit zwischen einem getestetem Code und seiner Integration, desto einfacher sind die Fehler lokalisierbar. Schließlich erinnert man sich besser, woran man heute geschrieben hat als letzte Woche. Häufiges Integrieren erlaubt auch das schnellere gemeinsame Testen mit Endkunden, was der Forderung des XP-Verfahrens nach kurzen Releasezyklen entspricht ([Sini 2003]).

Bei der Integration habe ich den gesamten Code auf meinem Entwicklungsrechner mittels SourceSafe von Microsoft Tool geladen (ich hatte keinen extra Integrationsrechner). Dann wurde der neue Code integriert, die Unit-Tests durchgeführt und, wenn alles in Ordnung war, der Code eingecheckt. Da ich alleine arbeitete, waren in den meisten Fällen keine Fehler vorhanden,.

In der Praxis gibt es den Integrationsrechner sehr selten, da es, falls ein XP-Team aus 12 Entwicklern besteht, d.h. aus sechs Paaren, und jedes Paar öfters am Tag seine Änderung einbinden wollte, immer Warteschlangen beim Integrationsrechner gäbe.

Jedes Paar lädt den gesamten Code auf seinen Rechner. Mittels Integrationsmanagement-Tool kommt automatisch der neue Code ins System – aber nur lokal auf dem Entwicklungsrechner – dann werden alle Unit-Tests auf dem lokalen Rechner durchgeführt. Sind alle Tests bestanden, werden die Änderungen am Server, an dem der gesamte Code vorhanden ist, eingecheckt.

In meinem Projekt habe ich als Versionsverwaltung das Tool „Source Safe“ von Microsoft verwendet.

3.2.2.3 Gemeinsame Verantwortung

Der Code gehört allen Programmierern. Jedes Paar hat die Pflicht, Codeverbesserungen jederzeit wahrzunehmen. Es gibt keinen Code-Teil, für den nur eine einzelne Person verantwortlich ist. Dies ist nur möglich, da dank der automatisierten Tests nach einem Refactoring auf einfache Weise nachgewiesen werden kann, dass sich keine neuen Fehler eingeschlichen haben. Wäre dies nicht der Fall, würde der Entwickler, der den Code ursprünglich programmiert hat, die Verantwortung für den Code nicht mehr übernehmen, nachdem ein anderer Entwickler in seinem Codeteil programmiert hat. Die Frage der

Verantwortung wäre ungelöst. So liegt die Verantwortung bei Codefehlern hier immer beim ganzen Team ([Beck, Kent 2003], [XP Übersicht und Bewertung]).
Weil ich alleine programmiert habe, habe ich immer alleine die Verantwortung für den ganzen Code übernommen.

3.2.2.4 Programmieren in Paaren

Der ganze Code wird von jeweils zwei Personen geschrieben, die mit einer Tastatur und einer Maus an einem Rechner arbeiten. Die beiden nehmen zwei unterschiedliche Rollen ein. Die eine Person ist der Fahrer, die andere der Beifahrer. Der Fahrer programmiert den Code, während der Beifahrer überlegt, ob es nicht eine einfachere Lösung gibt, Einwände vorbringt, nachfragt und kontrolliert. Die beiden Programmierer wechseln ihre Rollen alle paar Minuten und die Partner ca. jeden halben Tag. Dies erzielt den positiven Effekt, dass der Wissensaustausch unter ihnen aufs Äußerste forciert wird und dass durch Abgänge einzelner Programmierer auch nicht das ganze erarbeitete Wissen verloren geht. Zudem können Anfänger viel von den erfahrenen Kollegen lernen. Dass zwei Programmierer an ein- und demselben Rechner arbeiten, erscheint auf den ersten Blick sehr ineffizient, hat aber nebst dem Wissensaustausch auch den Vorteil, dass die Software viel weniger Fehler aufweist und ein sauberes Design implementiert wird ([Beck, Kent 2003], [XP Übersicht und Bewertung])).
Da ich alleine arbeitete, konnte ich diese Praktik nicht umsetzen.

3.2.2.5 40-Stundenwoche

Der Plan in XP-Verfahren und die 12 Praktiken organisieren die Arbeit im XP-Projekt insofern positiv, dass die Entwickler viele Aufgaben in 40 Stunden erledigen können, ohne Überstunden machen zu müssen, wie sie oft zur Einhaltung von ihren Abschätzungen bzw. von Lieferterminen notwendig sind. Die Schätzungen erlauben es den Programmierern, kritisch mit sich selbst umzugehen. Niemand soll mehr als die notwendigen 40 Stunden pro Woche arbeiten. Überstunden wirken sich meist negativ auf Qualität und Stimmung aus. Der Mensch ist keine Maschine und braucht Erholungszeit. Mehr als ein bis zwei Nächte an Überstundenarbeit sind nicht verkraftbar. Möchte jemand

zusätzliche Pausen einlegen oder eine mehrtägigeAuszeit nehmen, wird das von den Teammitgliedern respektiert.

Die Praxis zeigt hier manchmal ein anderes Bild. Programmierer sind es gewohnt, bis zur Erschöpfung vor dem Bildschirm zu sitzen. Nicht nur die Uhr bestimmt den Zeitplan, sondern vor allem die innere Befindlichkeit. Es wird oft solange gearbeitet, bis man einschläft.

Ich konnte diese Praktik nicht umsetzen, da ich immer neben diesem Projekt tätig war und daher immer mehr als 40 Stunden in der Woche gearbeitet habe.

3.2.2.6 Kunde vor Ort

Die Integration der Geschäftsseite (des Kunden) im XP-Team ist besonders wichtig. Da XP ohne ständige Kommunikation nicht funktioniert, muss der Kunde vor Ort sein. Seine rasche Verfügbarkeit ist notwendig, um Fragen zu beantworten und Prioritäten zu setzen. Idealerweise soll der Kunde im selben Raum arbeiten wie die Entwickler. Abbildung 3.2.2.6.1 zeigt die Kommunikationsarten und ihren Nutzen, welche schneller, welche langsamer und welche überhaupt funktionieren. Wichtig ist, dass der Kunde über ein relevantes Fachwissen verfügt und entscheidungsbevollmächtigt ist. Von Vorteil ist auch, wenn es sich immer um die gleiche Ansprechperson handelt, um Kontinuität zu gewährleisten.

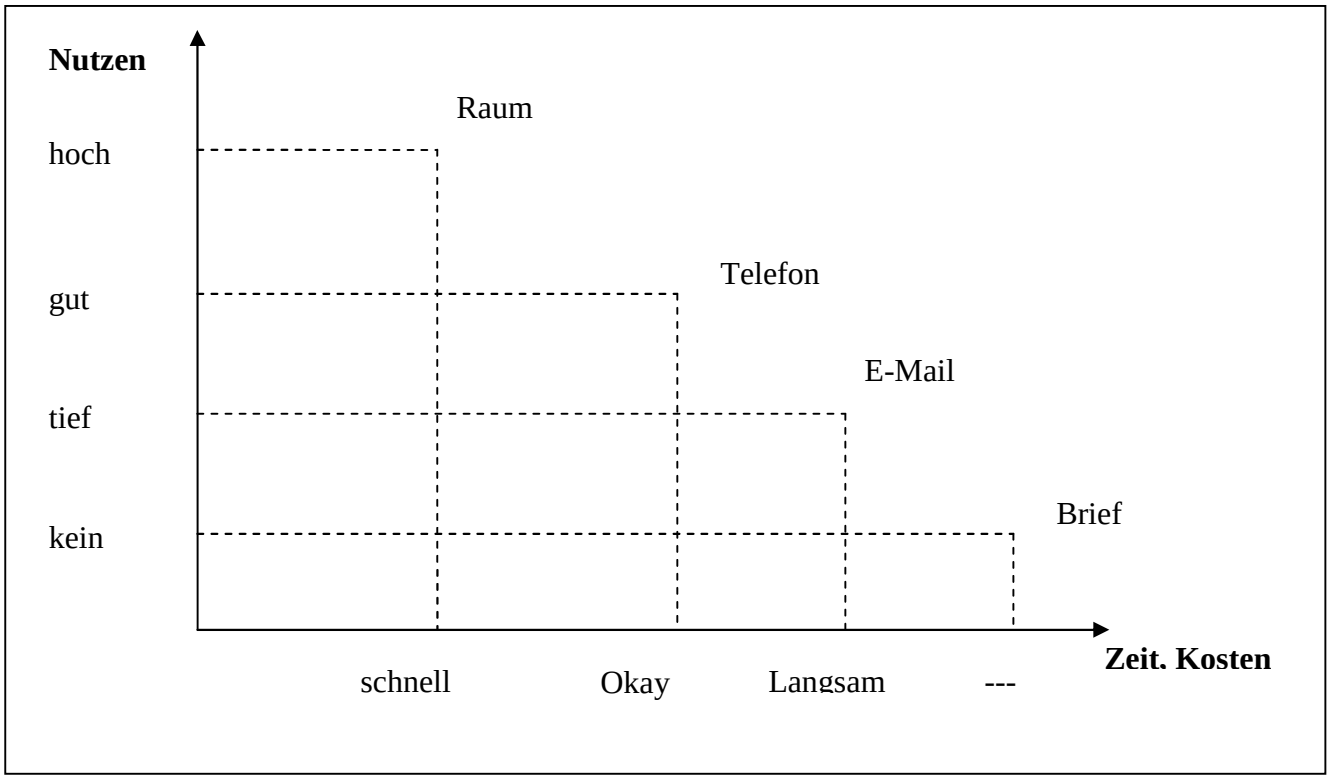


Abbildung 3.2.2.6.1: Kommunikationsarten

Mein Kunde und ich haben immer im selben Raum gearbeitet. Er arbeitet auch als Entwickler und war daher ständig darüber informiert, welcher Teil schon fertig war, welcher sich gerade in Arbeit befand und was in die nächste Version implementiert wurde.

3.2.3 **Testen**

Das XP-Projekt wird ständig größer und komplexer. Qualitätseinbußen sind die logische Folge. Deswegen ist die Durchführung von Tests (Unit Tests und Funktionalität Tests) notwendig. Dabei ist es gleichgültig ob XP oder ein anderer Softwareentwicklungs-prozess verwendet wird. Eine Aufgabe wird nicht weitergegeben, bevor alle dazugehörigen Tests zu 100 % erfüllt sind. Deswegen ist aus meiner Sicht das Testen die Wichtigste der zwölf XP-Regeln. Tests sind auf mehreren Ebenen zu entwickeln. Beginnend mit einfachen Methoden-Tests über Unit-Tests bis hin für die Überprüfung der Korrektheit von Geschäftsabläufen, welche durch die User-Stories repräsentiert werden, müssen Tests vorhanden sein.

3.2.3.1 Unit-Tests

Alle Entwickler verfassen, bevor sie zu programmieren beginnen, ihre Unit-Tests. Vor jedem Code-Baustein für die eigentliche Software wird erst ein Test definiert und implementiert. Schlägt dieser fehl, wird der Code so lange umgeschrieben, bis der Test positiv verläuft. Abwechselnd werden nun ein neuer Test und ein neuer Code geschrieben, bis alle geforderten Eigenschaften im Test geprüft worden sind. Zudem erhalten die Entwickler innerhalb weniger Minuten Feedback, so dass sie in kürzester Zeitspanne einen kleinen Erfolg erleben und sich die Zeit des mühsamen Fehlersuchens und Fehlereingrenzens ersparen ([XP Übersicht und Bewertung]).

XP entstammt dem objektorientierten Umfeld, weshalb die Tests als Klassen entwickelt werden. Ein Unit-Test vergleicht im Normalfall die Resultate mit den erwarteten Werten (Soll-/Ist-Vergleich, expected result). Wichtig ist, dass alle Tests gesammelt und mittels Softwareunterstützung automatisiert werden können. Dazu werden Tools und Testframeworks eingesetzt (z.B. JUnit für Java, NUnit für alle .Net Programmiersprachen C#, C++, …). So kann ohne Aufwand die ganze Testbatterie nach einer Code-Änderung wieder gestartet werden ([Beck, Kent 2003]).

Mit Hilfe des NUint Tools habe ich einige Unit-Tests geschrieben. Optimal wäre das das Schreiben von Unit-Tests für jede Funktion,, aufgrund des Zeitaufwands war für mich die Umsetzung dieses Ziels nicht möglich. Hier sind Beispiele für von mir geschriebene Unit-Tests:

```
/// <summary>
      /// Summary description for StatemaschineInfoTest.
      /// </summary>
      [TestFixture]
      public class StatemaschineInfoTest
      {
            protected StatemaschineInfo smInfo = new
            StatemaschineInfo(typeof(StatemaschineG1));
            protected StatemaschineG1 mysm = new StatemaschineG1();

            [Test]
            public string  CurrentStartStateTest()
            {
            try
```

```
    {
        Assert.AreEqual(this.mysm.StartState.ToString(),
       smInfo.CurrentStartState.ToString());
    }
    catch (Exception ex)
    {
        Trace.WriteLine(ex.Message);
        return ex.Message;
    }
    return "OK";

}
    [Test]
    public string GetAllStatesTest()
    {
    try
    {
        Assert.AreEqual(this.mysm.States.Count, smInfo.AllStatesName.Count);
        foreach (State st in this.mysm.States)
        {

          Assert.AreEqual(true,smInfo.AllStatesName.ContainsValue(Functions.
          Token(st.ToString())));
        }
    }
    catch (Exception ex)
    {
        Trace.WriteLine(ex.Message);
        return ex.Message;
    }
    return "OK";

    }
    [Test]
    public string  NameOfStatemaschineTest()
    {
    try
    {
        Assert.AreEqual("StatemaschineG1", smInfo.NameOfStatemaschine);
    }
    catch (Exception ex)
    {
        Trace.WriteLine(ex.Message);
        return ex.Message;
    }
    return "OK";

    }
[Test]
public string LongestNameOfStatesTest()
{
    try
    {

          Assert.AreEqual("State_____________________________________________
                _51", smInfo.LongestNameOfStates);
```

```
    }
    catch (Exception ex)
    {
        Trace.WriteLine(ex.Message);
        return ex.Message;
    }
    return "OK";
}
[Test]
public string LongestNameOfEmbeddedStatemaschineTest()
{
    try
    {
        Assert.AreEqual("SMTest2",
          smInfo.LongestNameOfEmbeddedStatemaschine);
    }
    catch (Exception ex)
    {
        Trace.WriteLine(ex.Message);
        return ex.Message;
    }
    return "OK";
}
[Test]
public string GetEmbeddedStatemaschineTest()
{
    try
    {
        State st1 =(State) smInfo.CurrentStatemaschine.States[0];
        State st2 = (State)smInfo.CurrentStatemaschine.States[1];
        State st3 = (State)smInfo.CurrentStatemaschine.States[2];

        Assert.AreEqual("SM", smInfo.GetEmbeddedStatemaschine(st1));
        Assert.AreEqual("SMTest2", smInfo.GetEmbeddedStatemaschine(st2));
        Assert.AreEqual("SMTest2_1", smInfo.GetEmbeddedStatemaschine(st3));
    }
    catch (Exception ex)
    {
        Trace.WriteLine(ex.Message);
        return ex.Message;
    }
    return "OK";
}
[Test]
public string GetAllEgressesTest()
{
    try
    {
        State st1 = (State)smInfo.CurrentStatemaschine.States[0];
        Hashtable Isegresses = smInfo.GetAllEgresses(st1);
        State st2 = (State)smInfo.CurrentStatemaschine.States[1];
        State st30 = (State)smInfo.CurrentStatemaschine.States[29];

        Hashtable egresses = new Hashtable();
        egresses.Add(st2, Egresses.Next + "," + Egresses.ESC);
        egresses.Add(st30, Egresses.Error);
```

```
            bool combareResult = this.IsEqual(Isegresses, egresses);
            Assert.AreEqual(true, combareResult);
        }
        catch (Exception ex)
        {
            Trace.WriteLine(ex.Message);
            return ex.Message;
        }
        return "OK";
    }
    private bool IsEqual(Hashtable ht1, Hashtable ht2)
    {
        if (ht1.Count != ht2.Count) return false;
        foreach (object key in ht1.Keys)
        {
            if (!ht2.ContainsKey(key)) return false;
            object valu1 = ht1[key];
            object value2 = ht2[key];
            if (!valu1.Equals(value2)) return false;
        }
        return true;
    }
}
```

In diesem Beispiel sind manche Tests Black-Box-Tests, d.h. sie sind vor der Methodenimplementierung geschrieben worden. Manche Tests sind White-Box-Tests, d.h. sie wurdenparallel oder nach der Methodenimplementierung geschrieben.
Hier ist auch ein Beispiel für die Ergebnisse des Tests:

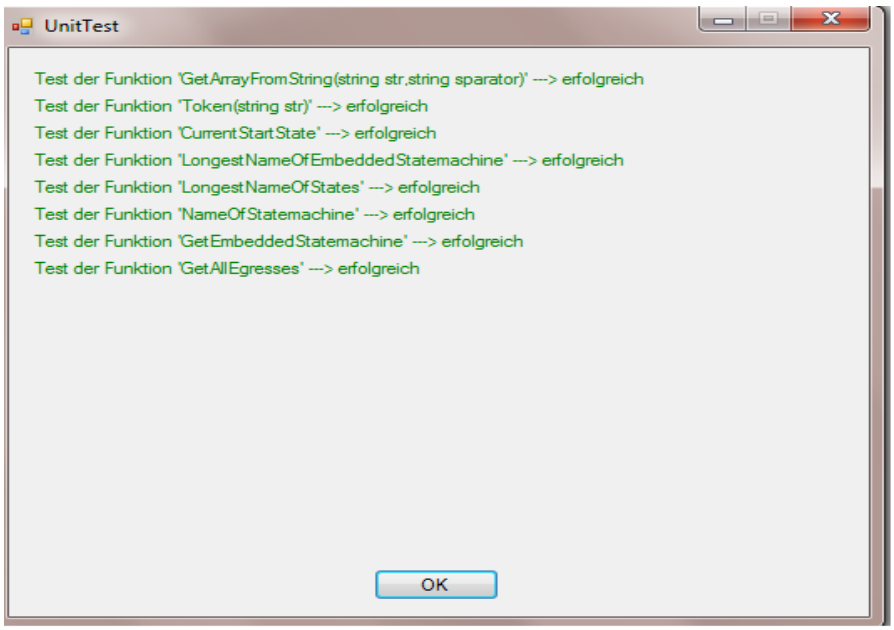


Abbildung 3.2.3.1.1: Unit-Test Ergebnis

3.2.3.2 Akzeptanztest

Dieser Test wird vom Kunden geschrieben. Das Ziel ist die Sicherstellung, dass die vom Kunden gewünschten Funktionen implementiert wurden. Eine Geschichte beschreibt ein Leistungsmerkmal, ein Akzeptanztest testet dieses Merkmal an einem konkreten Beispiel. Mindestens ein Akzeptanztest pro Geschichtskärtchen sollte, am besten für jede Aufgabe, geschrieben werden . Der Test besteht theoretisch aus drei Teilen: Einer Vorbedingung (Setup), dem Test selbst und einer Nachbedingung (Kontrolle) [Astels, David et al 2002], [Sini 2003]).

Der Kunde hat für mein Programm den Akzeptanztest wie folgt durchgeführt:

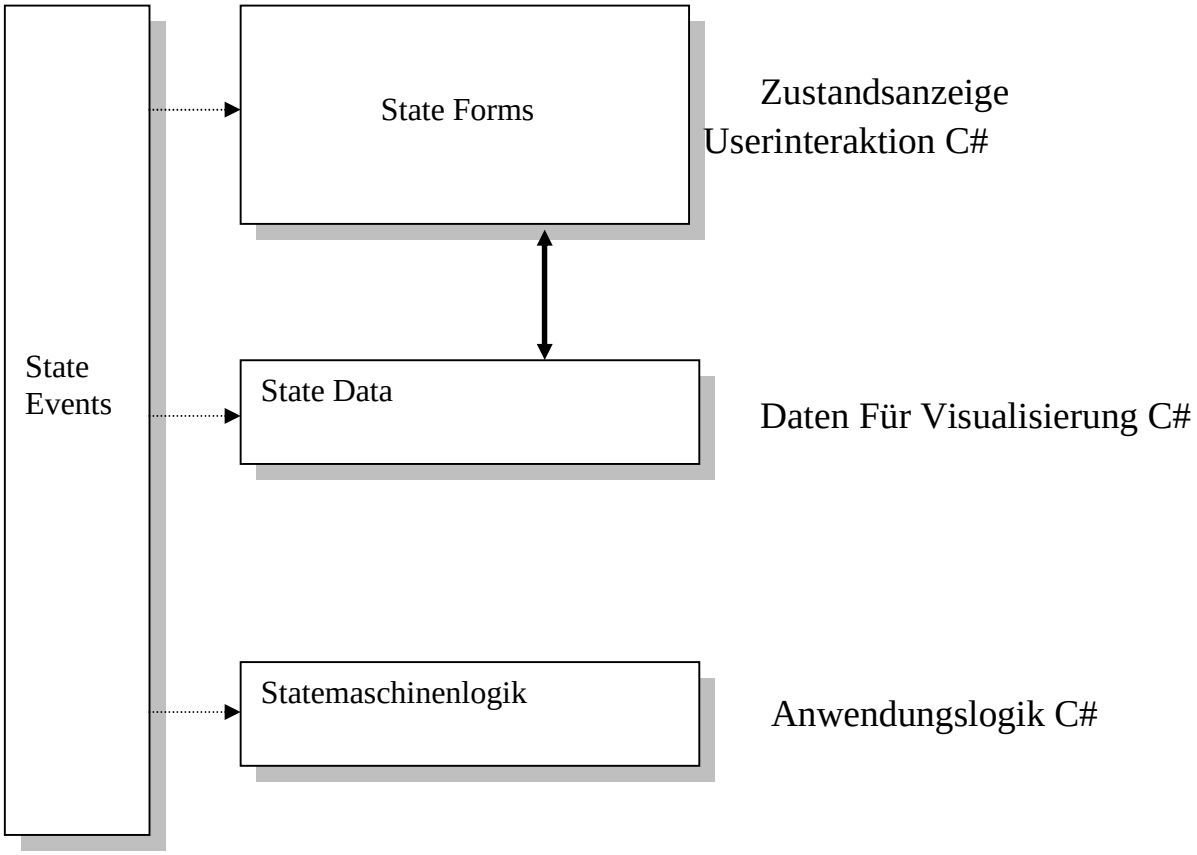
1. Er zeichnet die Statemaschine in einem Diagramm auf.
2. Er schreibt (implementiert) die entsprechende Klasse, die die Statemaschine beschreibt.
3. Er ließ mein Programm auf der Statemaschine laufen.
4. Er kontrolliert, ob das Diagramm in Schritt 1 mit dem Ergebnis in Schritt 3 übereinstimmt.

4 Die Statemaschine

Im Folgenden soll die Funktionsweise der Statemaschinenlogik näher beschrieben werden.

Die Statemaschinenlogik kann für Funktionen verwendet werden, die eine Statelogik benötigen z.B. Daten aus der Datenbank laden, Daten aus Geräten lesen und am Bildschirm anzeigen, ….

4.1 Grundstruktur



Statemaschinenlogik: Die Statemaschinenlogik beinhaltet im Wesentlichen zwei Objekte: Statemaschine und State. Jede Statemaschine besteht aus beliebig vielen States. In jedem State kann wiederum eine eigene Statemaschine ablaufen (verschachtelte Statemaschinen). Details dazu siehe unten.

State Data: Der Datenaustausch zwischen der Statemaschinenlogik und den State Forms erfolgt über einen Datenpool (State Data), der verschiedenste Datenobjekte beinhalten kann, die zyklisch oder eventgesteuert aufgefrischt werden.

State Forms: GUI für die Darstellung der Daten und des Ablaufs der Statemaschine. Dabei soll der Grundsatz verfolgt werden, möglichst wenig Anwendungslogik in den Forms zu realisieren. Dies verfolgt den Zweck, die Oberfläche leichter änderbar zu machen. Zudem kann man auch eine Form für verschiedene States in verschiedenen Stellen verwenden (trennen die Darstellung vom Login, Model-View-Controller Pattern MVC).

State Events: Über eine zentrale Eventverwaltung wird eine synchrone Abarbeitung der Abläufe erzwungen. Dies erleichtert die Durchschaubarkeit der zeitlichen Interaktionen und erlaubt eine einfache, da zentrale, Synchronisierung. Die Statemaschine arbeitet grundsätzlich zyklisch, d.h. sie wird periodisch über die State Events aufgerufen. Jeder State hat eine Methode „*UpdateState(UpdateStateArgs e)*“, die zyklisch mit einem bestimmten Takt (normalerweise 50 ms) aufgerufen wird. Ein Statewechsel ist nur synchron zu diesem Takt möglich.

4.2 Statemaschinenlogik

Die Statemaschinenlogik soll ein Grundgerüst zur Verfügung, stellen um mit geringem Aufwand Statemaschinen implementieren und debuggen zu können.

Dabei wurden folgende Anforderungen berücksichtigt:

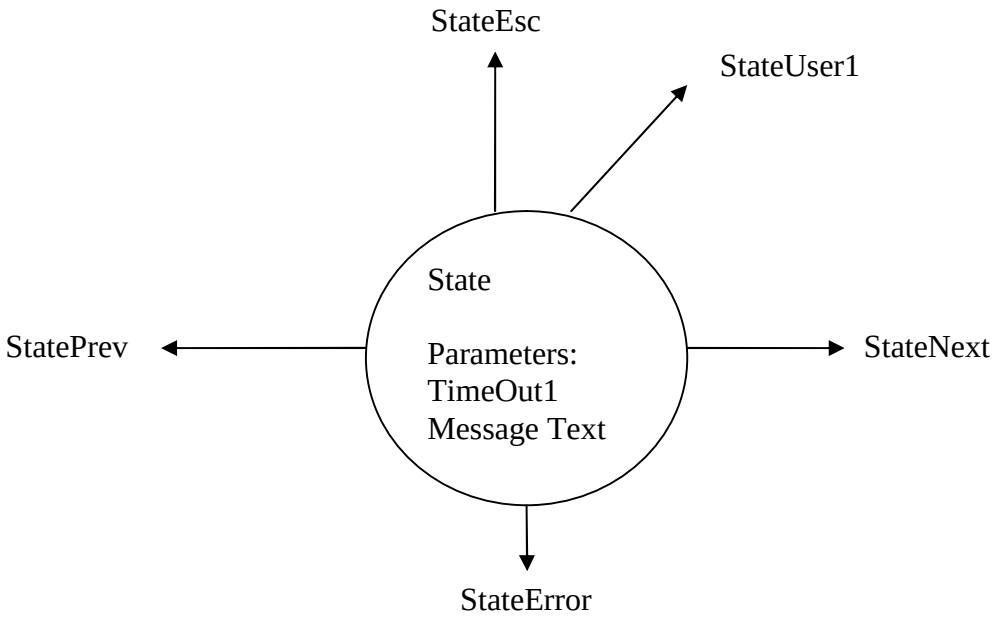
Anforderung	**Auswirkung**
Einzelne States sollen in mehreren Statemaschinen eingesetzt werden können	Funktionalität, die in einem State bereits fertig implementiert wurde, kann in anderen Statemaschinen eingesetzt werden
Jeder State ist über Parameter parametrisierbar	Ähnliche Funktionalität in einem State braucht nur einmal implementiert werden; über Parameter ist das Verhalten parametrisierbar.

Die Abfolge der einzelnen States wird nicht im State selbst, sondern durch die Statemaschine festgelegt.	Dies ist eine Grundvoraussetzung für die Wiederverwendbarkeit von States in anderen Statemaschinen.
Die Implementierung soll so erfolgen, dass die Logik der States und der Statemaschine selbsterklärend sind	An einer zentralen Stelle in der Statemaschine ist die Verschachtelung und damit die Abfolge der States implementiert. Die Logik der States ist einfach nachvollziehbar.
Die Abfolge der einzelnen States und die dabei notwendigen Basisabläufe werden von der Basis Statemaschinenlogik implementiert.	Die Anwendungs-Statemaschine braucht sich um keine Details zu kümmern.
Die Statemaschinenlogik soll mit „Standardtools“ debugfähig sein.	Die Statelogik wird in C# mit entsprechenden Klassen realisiert. Damit können alle Debugtools von Visual Studio.NET verwendet werden.
Der Ablauf der Statemaschine muss über Log Funktionen verfolgt werden können.	Durch die Realisierung mit .NET Objekten stehen alle in der Anwendung vorhandenen Logfunktionen zur Verfügung.
Die syntaktische Korrektheit eines States soll automatisch überprüft werden	Durch die Realisierung mit .NET Objekten wird dies vom Compiler automatisch gewährleistet.

4.2.1 **States**

Das Grundelement der Statemaschinenlogik ist ein einzelner State. Jeder State wird als eigenes Objekt realisiert:

- Über globale Properties bzw. Objektvariablen kann der State parametrisiert werden (von der Statemaschine).
- Jeder State hat exakt definierte Übergänge zu Folgestates.
- Welcher Folgestate bei einem solchen Übergang eingenommen wird, wird aber nicht vom State selbst, sondern durch entsprechende „Verdrahtung“ durch die Statemaschine bestimmt.
- Der Stateübergang wird aber vom State ausgelöst.



- Jeder State hat folgende Methoden:

  public override State UpdateState(UpdateStateArgs e); Diese Methode wird zyklisch aufgerufen. Hier wird die Statelogik implementiert. In der Logik wird entschieden, welcher Folgezustand eingenommen wird. Dies wird durch den Returnparameter festgelegt, z.B If (Timeout) return StateNext;

public override State EnterState(); Diese Methode wird unmittelbar aufgerufen, bevor der State eingenommen wird. Kann optional implementiert werden.

public override void LeaveState(); Diese Methode wird unmittelbar aufgerufen, nachdem der State verlassen wurde. Kann optional implementiert werden.

- Wird die Methode EnterState bzw. LeaveState nicht benötigt, muss sie auch nicht implementiert werden.
- Zusätzlich zu den drei Basismethoden können beliebig viele weitere Methoden implementiert werden, die z.B. die Parametrisierung eines States vereinfachen oder lokale Hilfsfunktionen darstellen.
- In jedem State können die Standard Logfunktionen verwendet werden. Damit kann der Ablauf der Statemaschine mitprotokolliert werden.
- Das Logging wird für den prinzipiellen Ablauf von den Basisobjekten durchgeführt (=Stateabfolge). Für spezielle Zwecke kann jedoch jeder State beliebige Daten selbst loggen.

Beispiel für eine mögliche Implementierung eines States:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using DiTest.State;
using DiTest.Controls;

namespace GraphicStatemaschine.Statemaschinen
{
    public class StateX:State
    {
        //in Basic class  State StateNext;
        //in Basic class  State StatePrev;
        //in Basic class  State StateEsc;
        public State StateUser1;

        //Parameters
        public long Timeoute1;

        // local variables
        long Counter;
        StateTimer timer2;
```

```
        // Methods
        public StateX()
        {
            timer2 = new StateTimer(this);
        }
        public override State Enter()
        {
            this.Counter = 0;
            return this;
        }
        public override State UpdateState(UpdateStateArgs args)
        {

            base.UpdateState(args);
            if (this.StateTimer.SecTick)
            {
                // do something every second
            }
            if (this.StateTimer.MinTick)
            {
                // do something ervery minute
            }
            if (this.StateTimer.Value > 2300) return this.StateNext; // return
StateNext after 2300 ms
            if (Counter > Timeoute1) return this.StateNext; // if the time is up the
next will be started (without User aktion)
            if (this.Command.Value == "Next") return this.StateNext; // if the user
pressed the button Next, the next will be started
            if (this.Command.Value == "Prev") return this.StatePrev; // if the user
pressed the button Prev, the previous  state will be started
            if (this.Command.Value == "Esc") return this.StateEsc; // if the user
pressed the button Esc, the Esc state will be stated
            return this; // else the current state stay aktiv
        }
       // Wenn keine spezielle LeaveState Behandlung notwendig ist, braucht sie auch
       // nicht implementiert zu werden.
    }
}
```

4.2.1.1 Zeitsteuerung, State Timer

Unter StateTimer ist eine Implementierung von Timern zu verstehen, die in den einzelnen States verwendet werden kann. Diese Timer werden automatisch mit dem Basistakt der States (50ms) betrieben und können für Zeitsteuerungen von jedem State verwendet werden. Jeder State hat einen StateTimer der in State (Basisklasse) automatisch angelegt wird. Dieser Timer beginnt mit EnterState zu laufen.

Benötigt ein State zusätzliche Timer, so sind diese im Konstruktor des State anzulegen:

```
        public StateX()
        {
            timer2 = new StateTimer(this);
        }
```

Beispiel für die Anwendung eines Timers:

```
if (this.StateTimer.SecTick)
{
   // do something every second
}
if (this.StateTimer.MinTick)
{
   // do something ervery minute
}
if (this.StateTimer.Value > 2300) return this.StateNext; // return StateNext after
2300 ms
```

Für Zeitsteuerungen wird empfohlen, diese Timer zu verwenden, da diese den Basistakt der Statemaschine automatisch berücksichtigen (egal ob der Zyklus 10ms, 50ms oder 100ms beträgt).

Weitere Methoden:

Start(); // Startet einen Timer

Stop(); // Stoppt einen Timer

4.2.1.2 Zähler

Die Klasse StatCounter implementiert einen Zähler, der automatisch z.B. pro Sekunde um eins hinauf oder hinunter zählt.

Properties:

Direction: CountDirection.Down, CountDirection.Up.

Preset: Mit jedem EnterState wird der StateCounter auf diesen Wert gesetzt.

MsTick: Nach dieser Anzahl an ms wird der Zähler um eins erhöht.

AutoReload: Gibt an, ob der Zähler nach Erreichen eines Maximal (Minimal) Wertes automatisch neu geladen werden soll.

ReloadAfter: Gibt den Maximal/Minimal Wert an, nach dem der Zähler wieder mit dem *Preset* Wert geladen wird; nur wenn AutoReload=true.

4.2.1.3 Standard Variablen eines States

In der Basisimplementierung von State sind einige Variablen fix vordefiniert, die in allen abgeleiteten Klassen verwendet werden können und sollen.

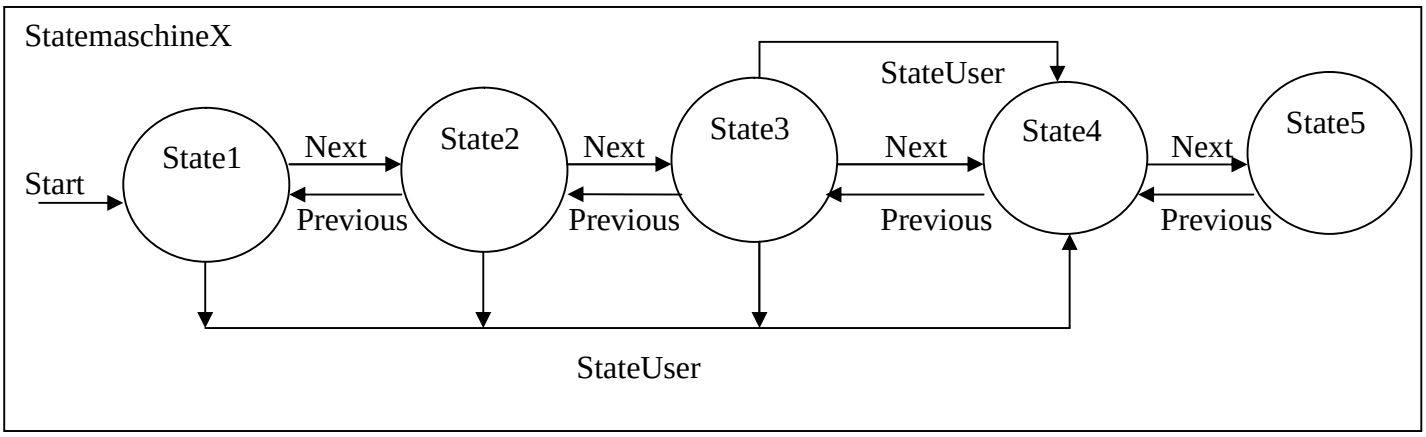
```
//Folgestaes
State StateNext; // Folgezustand Next
State StatePrev; // Folgezustand Previous
State StateEsc; //Folgezustand Esc
// Timer
StateTimer StateTimer; // beginnt mit EnterState zu laufen
// Counter
StateCounter StateCounter; // Zähler
```

4.3 Statemaschine

Jede Statemaschine ist ein eigenes Objekt, das einen Ablauf beschreibt, der aus einzelnen States zusammengesetzt ist:

- Alle benötigten State-Objekte werden instanziiert.
- Über die „Verdrahtung“ der einzelnen States wird der Ablauf der Statemaschine festgelegt.
- Durch diese „Verdrahtung“ ist die Stateabfolge an einer zentralen Stelle beschrieben (= implementiert und grafisch dokumentiert).
- Bei der „Verdrahtung“ können die einzelnen States wenn nötig parametrisiert werden.
- Jede Statemaschine hat folgende Methoden:
  public override void Enter (); Diese Methode wird unmittelbar vor dem Start der Statemaschine aufgerufen.
  public override void Leave (); Diese Methode wird unmittelbar nach dem Start der Statemaschine aufgerufen.
- Im Konstruktor der Statemaschine müssen alle Initialisierungen durchgeführt werden.

Beispiel für eine Statemaschine:



Code

```
using System.Collections.Generic;
using System.Linq;
using System.Text;
using DiTest.State;
using DiTest.Test.DisplayStatemaschine.Statemaschinen;

namespace GraphicStatemaschine.StateMaschinen
{
    public class StatemaschineX : Statemaschine
    {
        public StatemaschineX()
        {
            State1 st1 = new State1();
            State2 st2 = new State2();
            State3 st3 = new State3();
            State4 st4 = new State4();
            State5 st5 = new State5();
            this.States.Add(st1);
            this.States.Add(st2); // this means st1.StateNext=st2  and
st2.StatePrev=st1
            this.States.Add(st3);
            this.States.Add(st4);
            this.States.Add(st5);
            this.StartState = st1;
           //special states follow wiring
            st1.StateUser = st2.StateUser = st3.StateUser = st4;
        }
    }
}
```

4.3.1 **Verschachtelte Statemaschinen**

Das Statemaschinenkonzept sieht vor, dass einzelne Statemaschinen beliebig ineinander verschachtelt sein können.

Dabei wird in einem State (S) der übergeordneten Statemaschine (SM1) eine neue Statemaschine (SM2) angelegt und abgearbeitet.

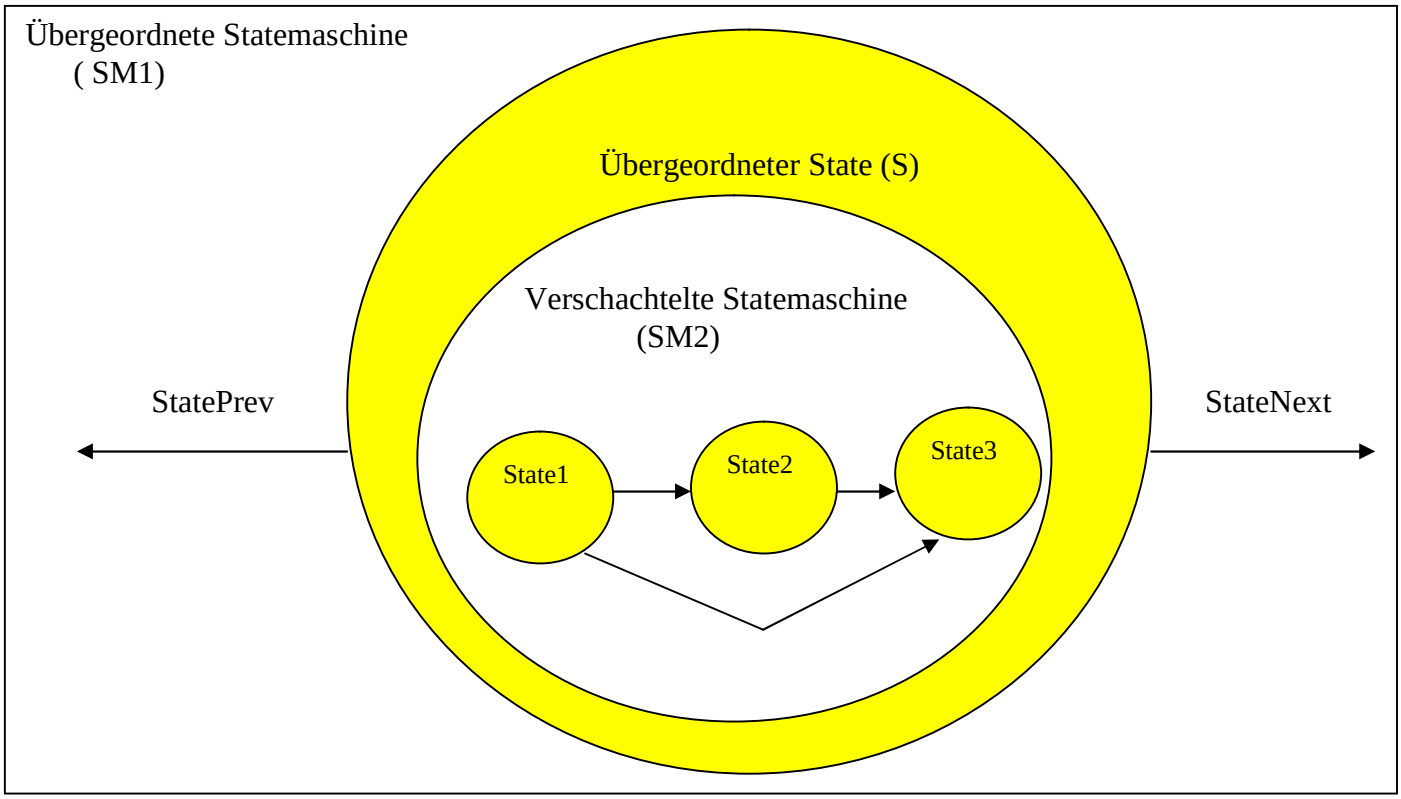


Abbildung 4.3.1.1: Verschachtelte Statemaschine

Damit dies einfach verwaltbar wird, gibt es im Basis-State eine Funktion „SetStatemaschine“, mit der innerhalb des State eine eigene Statemaschine betrieben werden kann.

Bei jedem UpdateState() Ereignis wird diese Statemaschine dann automatisch angetrieben (=UpdateState Aufrufe werden an die Statemaschine weitergeleitet).

Die untergeordnete Statemaschine wird mit jedem „EnterState“ neu gestartet.

Beispiel:

```
public class State1 : State
    {
        public State StateUser;
        public State1()
        {
            SetEmbeddedStatemaschine(new SM2());
        }
        public override State UpdateState(UpdateStateArgs args)
        {
            base.UpdateState(args);
            if (this.EmbeddedStatemaschine.Finished()) return this.StateNext;

            return this;
        }
    }
```

Die Zuordnung einer Statemaschine kann auch dynamisch zur Laufzeit erfolgen.

Z.B: stateX.SetStateMaschine(new SMxy());

Setzt eine neue Statemaschine am stateX; Beim nächsten stateX.Enter Ereignis wird die Statemaschine gestartet.

4.4 State Forms

State Forms sind GUI Formulare, die den aktuellen Zustand einer Statemaschine sowie beliebige andere Daten anzeigen und Benutzereingaben weiterleiten können. Die Kommunikation mit der Statemaschine wird über Datenobjekte abgewickelt. Die Realisierung dieser Formulare kann über WinForms bzw. WinForm Usercontrols erfolgen.

Eine mögliche Realisierung:

- Jedes State Form wird als UserControl realisiert
- Ein Basis UserControl wird erstellt, um die Abfolge von einzelnen UserControls zu implementieren.
- Das Basis UserControl kann dann in ein WinForm oder in eine Web Page eingebettet werden.

Ein State Form trifft selbst keine Entscheidung darüber wann die nächste Form angezeigt wird, bzw. welches die nächste Form ist. Dies wird über die Statemaschinenlogik (States) gesteuert. Die Statemaschinenlogik bietet selbst keine Implementierungen für ein GUI bzw. GUI Basisklassen. Entsprechende Formulare müssen außerhalb der Statelogik implementiert werden und über das Interface IStateFormControl eingebunden werden.

4.4.1 **State Form Manager**

Zur Unterstützung der Statemaschinenlogik wird ein eigener State Form Manager entwickelt. Dieser ist für das Anzeigen und Weiterschalten der Formulare zuständig.

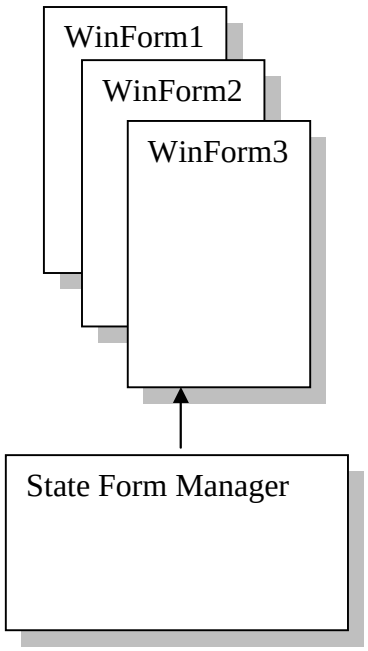


Abbildung 4.4.1.1: Forms in der Statemaschine

Weiters löst der State Form Manager das Ereignis zum Update der Controls auf den State Forms aus.

4.4.2 **Update der Controls**

Das Update der Controls auf den Forms wird über den Form Manager gesteuert. Jede State Form hat eine virtuelle Funktion *void UpdateControls(UpdateControlArgs e);*

Diese wird vom FormManager zyklisch aufgerufen. In dieser Funktion muss jede Form seine Controls updaten.

Beispiel:

public override void UpdateControls(UpdateControlArgs e) {

This.TextBox1.Text =(String)Forms.Data;}

In diesem Beispiel wird an das GUI Control „TextBox1“ eine Referenz auf Forms.Data übergeben. Dabei wird ein Update des Controls ausgelöst. Das Control entscheidet selbst, ob es neu gezeichnet werden muss oder nicht.

4.5 Statedata

Über den Datenobjekt Pool Statedata wird der Datenaustausch zwischen den Statemaschinen und den Stateforms abgewickelt. Dazu werden Datenobjekte definiert, die in diesem Pool abgelegt werden können.

Beispiel:

```
public class DataTransfer
    {
        public DataTransfer(string Message, string text1, string text2)
        {
            this.Message = Message;
            this.Text1 = text1;
            this.Text2 = text2;
        }
        public string Message { get; set; }
        public string Text1 { get; set; }
        public string Text2 { get; set; }
    }
```

In dem State

```
public class State1 : State
    {
        public State StateUser;
        DataTransfer datatransfer;
        public State1()
        {
           SetEmbeddedStatemaschine(new SM2());
           this.datatransfer = new DataTransfer("Message1", "Text1", "Text2");
        }
        public override State UpdateState(UpdateStateArgs args)
        {

            base.UpdateState(args);
            if (this.EmbeddedStatemaschine.Finished()) return this.StateNext;
            Forms.Data = this.datatransfer;
            return this;
```

```
        }
    }
```

In Form

```
public override void UpdateControls(UpdateControlsArgs args)
 {
    base.UpdateControls(args);

   this.TextBox1.Text=((DataTransfer)Forms.Data).Message1;
 }
```

4.5.1 **Update der Daten**

Das Update der Daten erfolgt über den Event *UpdateData(UpdateDataArgs args):*, der zyklisch im Statemaschinentakt aufgerufen wird. Dabei werden alle zu dem jeweiligen Zeitpunkt benötigten Daten von der Dau[1] gelesen und in Statedata eingetragen. Weiters werden bestimmte Werte direkt vom Ablauf im aktuellen State bzw. der Statemaschine gesetzt.

4.6 Zusammenspiel State, Stateform, Statedata

Im Konstruktor des State kann die Form, die angezeigt werden soll und die Datenaustauschklasse instanziiert werden.

```
public StateLicenseUpdate()
{
     ActiveForm = typeof(FormLicenseUpdate);
     myData = new DataLicenseUpdate();
}
```

Mit Hilfe des Objekts myData kann nun der State der Stateform notwendige Daten übergeben bzw. Daten von der Form ablesen, sofern die Form diese Daten dort bereitstellt. Damit die Form Zugriff auf das Datenobjekt hat, muss sie sich beim Laden (zuvor steht das Objekt nicht zur Verfügung) das Objekt holen:

```
private void FormLicenseUpdate_Load(object sender, EventArgs e)
{
     // Set here the link to the data since it's not available
     // at the constructor.
     myData = (DataLicenseUpdate)Forms.Data;
}
```

[1] Dau (Data Acquisition Unit) ist ein Programm, das parallel zur Statemaschinenanwendung läuft bzw. laufen kann, um die Daten vom Auto zum Beispiel zu simulieren

5 Die Implementierung der Arbeit

In diesem Kapitel gebe ich dem Leser einen Überblick über das Programm bzw. die Implementierung der Arbeit.

5.1 Die Entwicklungsumgebung (.Net Framework)

Ich habe die Entwicklungsumgebung „Microsoft Visual Studio 2008 .Net Framwork 3.5“ verwendet. Die Programmiersprache ist C#, da die Statemaschinen, die der SMV aufzeichnet, auch mit C# geschrieben wurden.

5.2 Der Algorithmus

Die Idee dahinter ist die Realisierung bzw. Codierung eines Algorithmus, , damit das resultierende Programm (Statemaschinenvisualizer SMV) den Datentyp „Statemaschine“ als Eingabe übernimmt und diese „Statemaschine“ in grafischer Form als Ausgabe anzeigt.

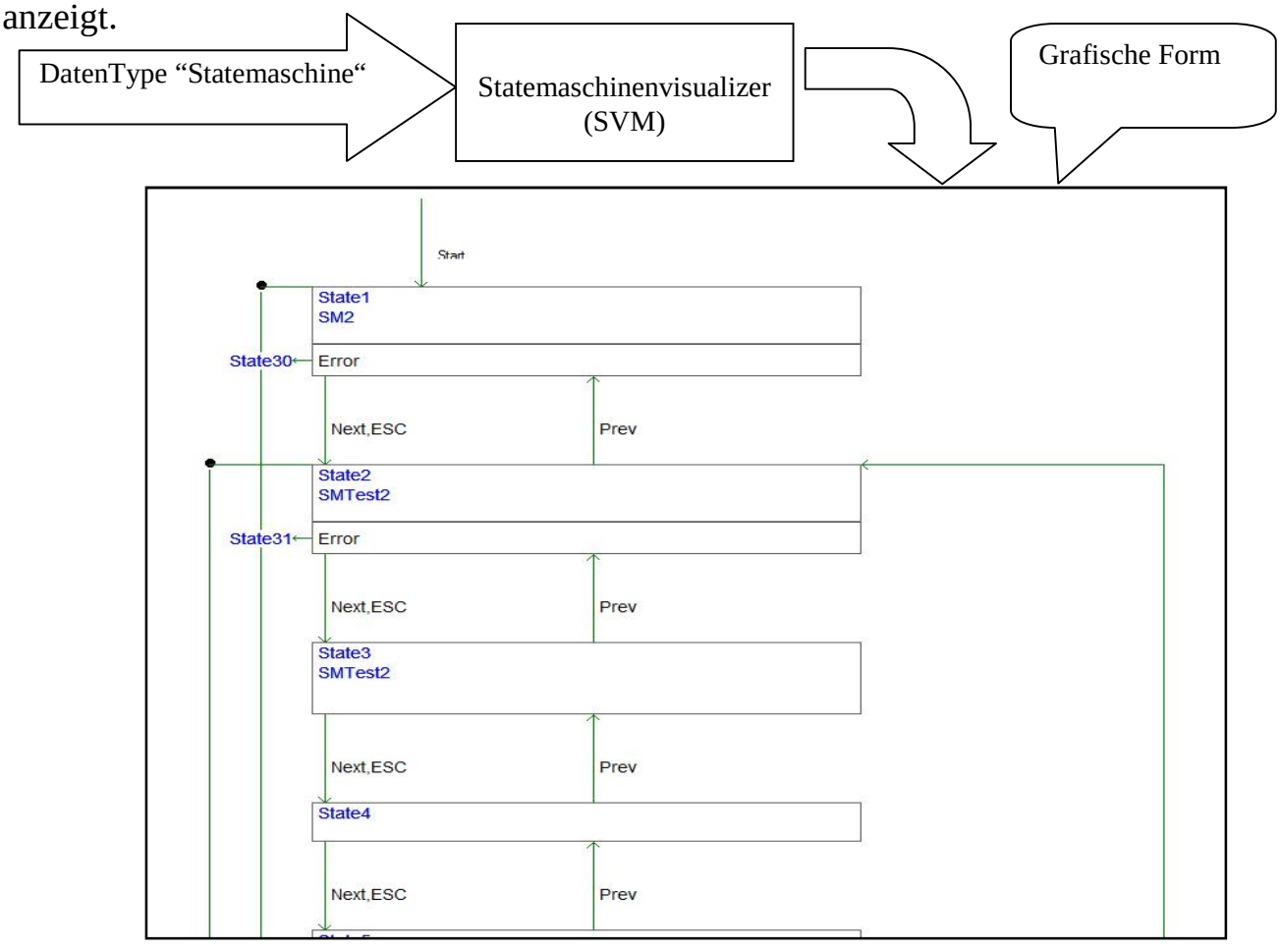


Abbildung 5.2.1: Eingabe und Ausgabe des Programms

Es muss eine Klasse “StatemaschineInfo“ implementiert werden, die aus dem Datentyp „Statemaschine“ alle benötige Informationen für die Aufzeichnung extrahiert, z.B. eine Liste von allen States in der Statemaschine, welcher State von diesen States ist der Startstate, die Ausgänge von jedem State, verschachtelte Statemaschinen,…

```
public class StatemaschineInfo
      {
            private  Statemaschine  CurrentStatemaschine =null;
            public StatemaschineInfo( object  smtyp)
            {
                  Type tp = (Type)smtyp;
                  if (!tp.IsSubclassOf(typeof(Statemaschine)))
                  {
                        MessageBox.Show("Die eingegebene Statemaschine ist
                         ungültig","Fehler");
                        return;
                    }
```

Für das Extrahieren der Information von der Statemaschine wird der in C# vorhandene „Reflektions“-Mechanismus verwendet. Das heißt, es wird eine Instanz von den eingegebenen Daten-Typen instanziiert.

```
                 ConstructorInfo[] ci = tp.GetConstructors();
                  ParameterInfo[] pi   = ci[0].GetParameters();
                  if (pi.GetLength(0) == 0)
                  {

                   CurrentStatemaschine=(Statemaschine)Activator.CreateInstance(tp);
                  }
                  else
                  {
                        object[] args = new Object[pi.GetLength(0)];

            CurrentStatemaschine=(Statemaschine)Activator.CreateInstance(tp,args);
                  }
                  if (CurrentStatemaschine == null)
                  {
                        MessageBox.Show("Die eingegebene Statemaschine kann nicht
                         instansiert werden","Fehler");
                        return;
                  }
```

Anhand vom instanziierten Objekt kann man alle benötigten Informationen, z.B. für den Start state, ablesen:

```
public  State  CurrentStartState
   {
   get
   {
       if (CurrentStatemaschine != null)
            return CurrentStatemaschine.StartState;
       else return null;
   }
   }
```

Die Form „MainForm“ verwendet diese Informationen, um die Statemaschine am Bildschirm anzuzeigen.
Als wichtige Information für das Aufzeichnen ist die Liste von den States in der Statemaschine :

```
protected void FillAllStates()
            {
                  foreach(State st in CurrentStatemaschine.States)
                  {
                        if (!allStatesName.ContainsKey(st))
                        allStatesName.Add(st,Functions.Token(st.ToString()));
                  }
            }
public  Hashtable AllStatesName
        {
            get { return allStatesName; }
        }
```

Weitere Informationen von der Statemaschine sind:

- Name der Statemaschine
- Der längste Statename in der Statemaschine
- Die verschachtelten Statemaschinen
- …

Jeder State wird als Quadrat dargestellt, die Ausgänge des States zu anderen States werden mit Pfeilen repräsentiert. In diesem Quadrat steht der Name des State, der Link zu der Statemaschine, die von diesem State verschachtelt ist, falls vorhanden. In dem unteren

Teil des Quadrats stehen die Namen der Ausgänge. Von jedem Ausgang wird ein Pfeil gezeichnet, der auf den Namen des Ausgangsstate zeigt. Wenn der Ausgangsstate unmittelbar über oder unter dem State in der Grafik liegt, zeigt der Pfeil direkt auf den State selbst (nicht den Namen von dem State), und der Name vom Ausgang wird neben den Pfeil geschrieben.

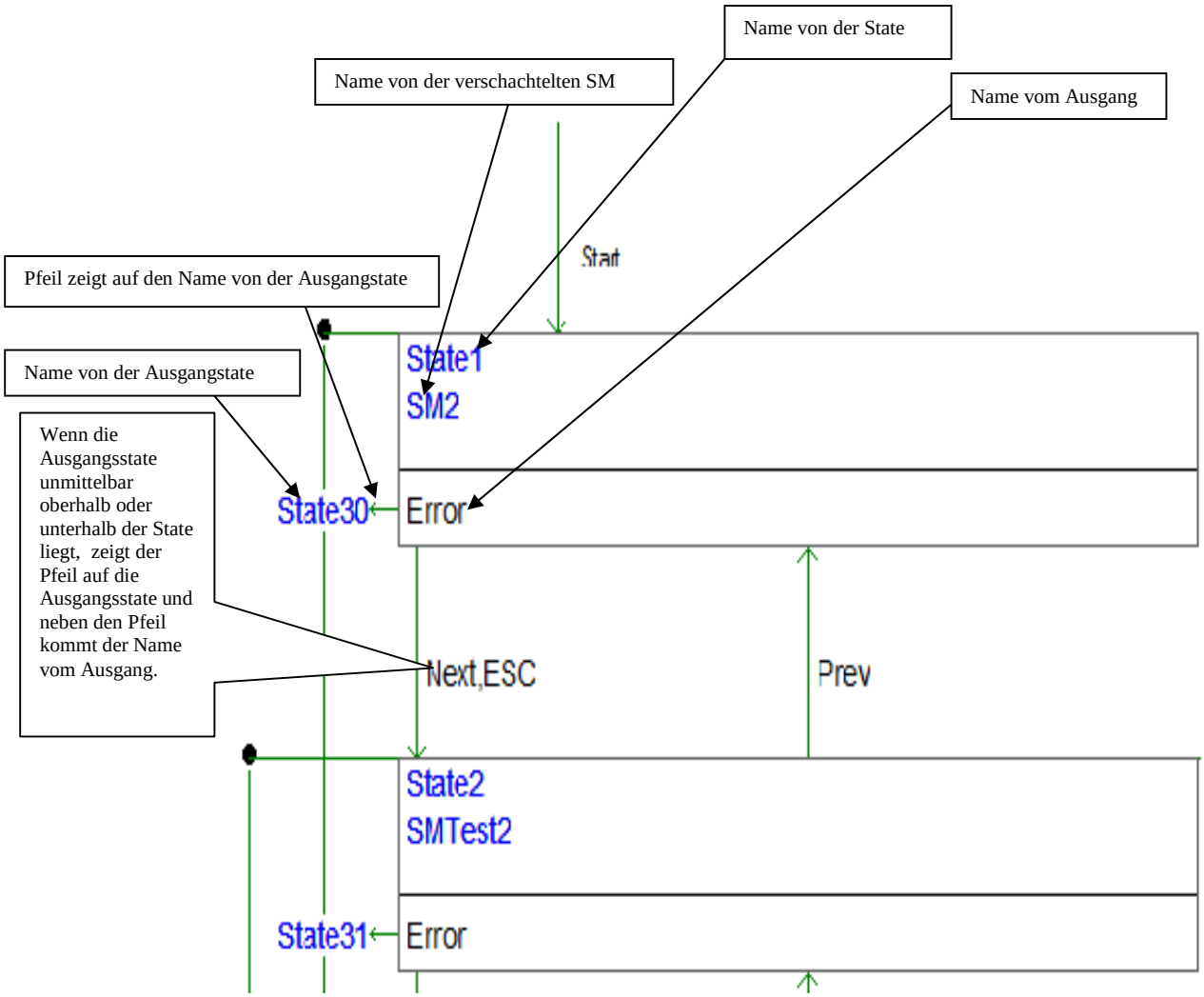


Abbildung 5.2.2: SM in grafischer Form

Zuerst werden die States, die einen „NextAusgang“ haben, gezeichnet. Es wird immer der Startstate der Statemaschine als erster State gezeichnet, danach werden alle anderen States in der Statemaschine dargestellt. Zum Schluss werden die Pfeile zwischen den States gezeichnet und mit den Ausgangsnamen beschriftet.

5.3 Überblick über den Code

Ich habe versucht, das MVC-Konzept (MVC Model View Controller, Modell /Präsentation/ Steuerung) so gut als möglich zu realisieren. Das heißt, die Daten, die Logik und die Benutzeroberfläche sind voneinander getrennt.

Der Code besteht aus mehreren Klassen, die beiden wichtigsten sind:

- StatemaschineInfo: die Klasse repräsentiert die Daten und Logik
- MainForm: die Klasse dient zum Anzeigen der Grafik

Weiters gibt es die Unit TestKlasse „StatemaschineInfoTest“. Sie ist für den Unit Test der StatemaschineInfo Klasse-Funktionalität zuständig.

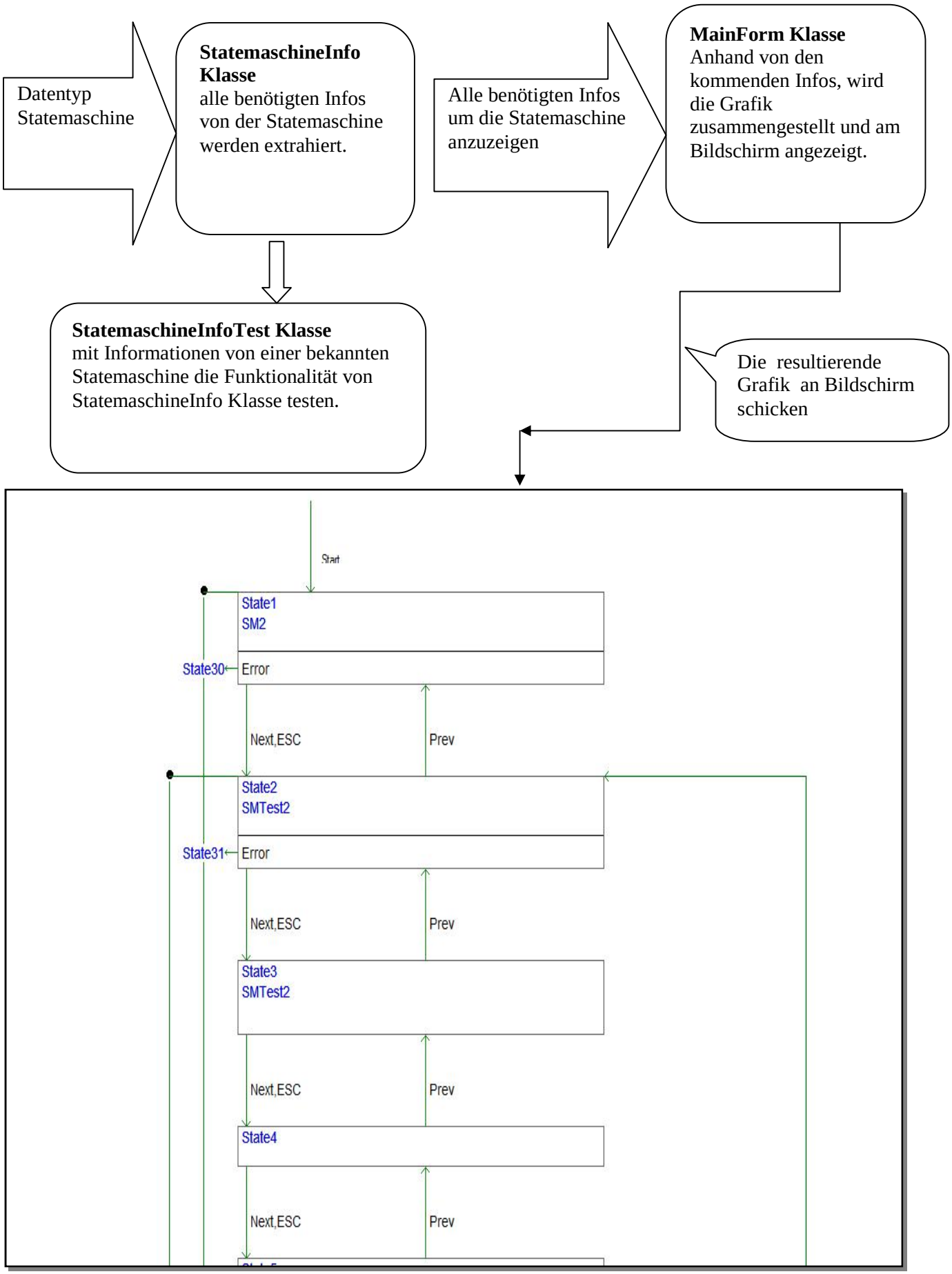


Abbildung 5.3.1: Zusammenhang zwischen die Haupt Klassen

Die wichtigsten Funktionen in der Klasse StatemaschineInfo:

- *GetEmbeddedStatemaschineOfEachState()*: Sie erstellt eine Liste mit allen States, die die Statemaschine einbindet sowie den Namen von ihren verschachtelten Statemaschinen
- *GetAllEgressesForEachState()*: Sie erstellt eine Liste mit allen States in der Statemaschine und ihre Ausgangstates

Die wichtigsten Funktionen in der Klasse MainForm:

- *DrawNextStates(string egress)*: Zeichnet alle States in der Statemaschine, die „NextAusgänge“ haben und auch ihre Ausgänge.
- *DrawState(State state, string egr, State nextstat)*: Diese Funktion zeichnet das Quadrat, welches den angegebenen State „state“ repräsentiert.
- *DrawArrowFromStateToState(Rectangle source, Rectangle target, string label, string egress,string targetName,string currentStateName)*: Zeichnet die Pfeile von State “Source” zu State “target”.
- *DrawEgresses(State state, string ergress, Hashtable ht)*: Zeichnet den angegebene State „state“ und ihre Ausgangstates

Als Beispiel für die grafische Ausgabe einer Statemaschine betrachten Sie die folgende Statemaschine „StatemaschineX“:

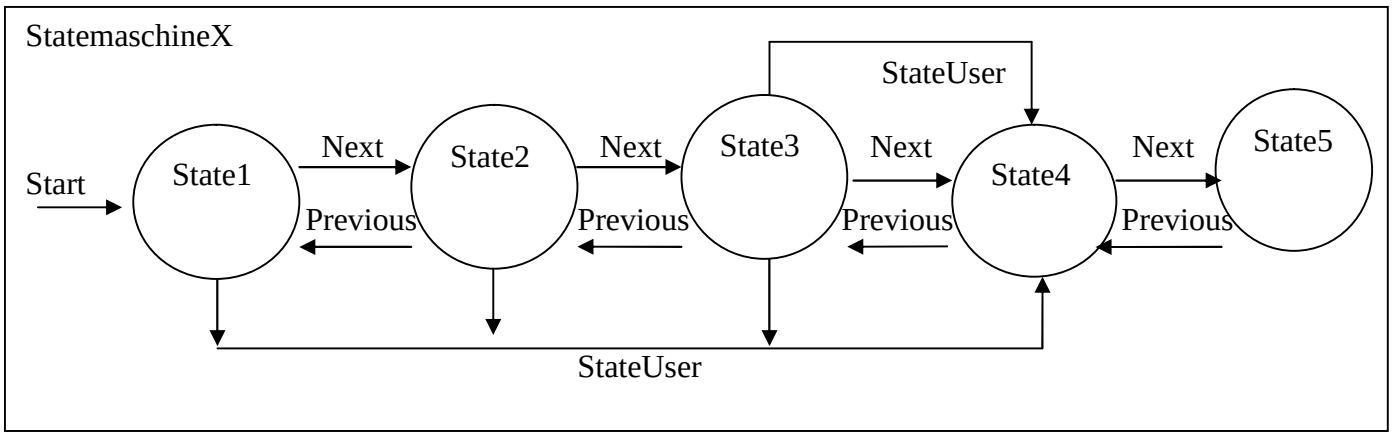


Abbildung 5.3.2 : StatemaschineX

Wie der folgende Code zeigt, besteht sie aus fünf States State1, State2, State3, State4 und State5,

```
State1 st1 = new State1();
State2 st2 = new State2();
State3 st3 = new State3();
State4 st4 = new State4();
State5 st5 = new State5();
```

Beim Hinzufügen zweier States st1 und st2 in die Statemaschine mittels add-Funktion (`this.States.Add(st1);this.States.Add(st2);`) wird automatisch als nächster State nach st1 der State st2 und der vorherige State von st2 ist st1 (`st1.StateNext = st2;st2.StatePrev = st1;`). Natürlich kann man diesen Default ändern.

Die State1, State2 und State3 haben einen Ausgang namens „StateUser“. Dieser Ausgang zeigt auf State4 für alle diese drei States (`st1.StateUser = st2.StateUser = st3.StateUser = st4;`)

Diese State1 verschachtelt eine komplette Statemaschine „SM2“

```
 public class State1 : State
{
    public State StateUser;
    DataTransfer datatransfer;
    public State1()
    {
       SetEmbeddedStatemaschine(new SM2());
       this.datatransfer = new DataTransfer("Message1", "Text1", "Text2");
    }
    public override State UpdateState(UpdateStateArgs args)
    {

        base.UpdateState(args);
        if (this.EmbeddedStatemaschine.Finished()) return this.StateNext;
        Forms.Data = this.datatransfer;
        return this;
    }
}
```

Und hier ist der komplette Code von der Statemaschine „StatemaschineX“

```
public class StatemaschineX : Statemaschine
    {
        public StatemaschineX()
        {
            State1 st1 = new State1();
            State2 st2 = new State2();
            State3 st3 = new State3();
            State4 st4 = new State4();
            State5 st5 = new State5();
            this.States.Add(st1);
            this.States.Add(st2); // this means st1.StateNext=st2  and
st2.StatePrev=st1
            this.States.Add(st3);
            this.States.Add(st4);
            this.States.Add(st5);
            this.StartState = st1;
            // Spezielle Folgestate Verdrahtung
            st1.StateUser = st2.StateUser = st3.StateUser = st4;


        }
    }
```

Wenn wir diese Statemaschine „StatemaschineX“ an das Programm als Eingabe schicken, so sieht die grafische Darstellung von der Statemaschine wie folgt aus:

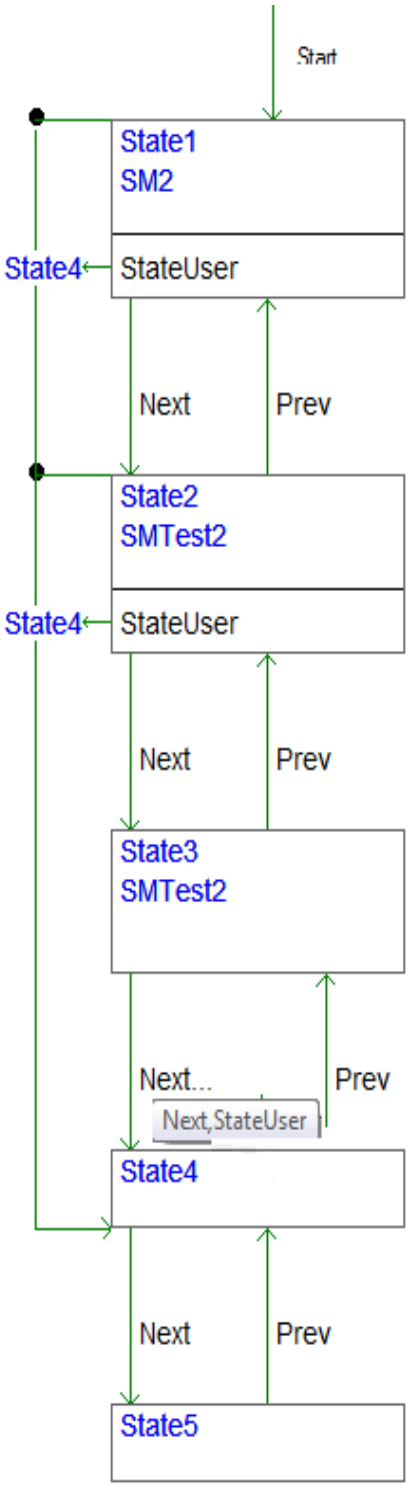


Abbildung 5.3.3: Die grafische Form von StatemaschineX

6 Anhang

6.1 Literatur- und Quellenverzeichnis

6.1.1 **Bücher**

[Astels, David et al 2002]: *A Practical Guide to Extreme Programming*, Upper Saddle River, Prentice Hall Verlag

[Beck, Kent 2000]: Extreme Programming: Die revolutionäre Methode für Softwareentwicklung in kleinen Teams, München, Addison Wesley, SBN 3-8273-1709-6.

[Beck, Kent 2001]: Extreme Programming planen. München: Addison-Wesley Verlag

[Beck, Kent 2003]: *Extreme Programming*. Das Manifest. München: Addison-Wesley Verlag

[Jeffries 2001]: Jeffries, Ron / Anderson, Ann / Hendrickson, Chet (2001) *Extreme Programming installed.* München: Addison-Wesley.

[NeMa 2001]: *Extreme Programming in Practice* . USA Addison Wesley Longman Verlag, ISBN 0-201-70937-6

6.1.2 **Diplomarbeiten**

[Sini 2003]: Sini Zivkovic, Extreme Programming - Theorie und praktische Erfahrung : Vom Apfelkuchen bis zur Datenbank, Oktober 2003. http://diuf.unifr.ch/people/fuhrer/studproj/zivkovic/xp_da_public.pdf

[XP Übersicht und Bewertung]: Rolf Dornberger / Thomas Habegger Extreme Programming, Eine Übersicht und Bewertung ; ISBN 3-03724-067-9, Dezember 2004 http://web.fhnw.ch/personenseiten/rolf.dornberger/Documents/Lectures/swe/Stuff/67_DiscussionPaper_ExtremeProgramming_1_pdf.pdf

6.1.3 **Internetseiten**

[Agila]: https://www.inf.fu-berlin.de/wiki/bin/viewfile/SE/SeminarAgileProzesse2006?rev=2;filename=02_agile_xp_pager.pdf

[Armin 2004]: http://www.iai.uni-bonn.de/III//lehre/praktika/xp/xp2004a/ext/data/XP%20Einf%c3%bchrung%2013.01.04.pdf

[Bögli 2003]: http://www.ifi.uzh.ch/groups/req/courses/seminar_ws03/12_Boegli_Anforderungen_Ausarbeitung.pdf

[Cold 2002]: http://www.sigs.de/publications/os/2002/01/coldewey_OS_01_02.pdf

[Dicke 2002]: http://danae.uni-muenster.de/~lux/seminar/ss02/Dicke.pdf

[Fowler 2003]: http://www.martinfowler.com/articles/newMethodology.html#N4001F8

[Frank Westphal]: http://www.frankwestphal.de/ExtremeProgramming.html

[Heyer 2002]: http://www.informatik.uni-leipzig.de/ifi/lehre/Heyer9900/kap25/sld007.htm

[Hofer 2002]: http://www.sea.unilinz.ac.at/teaching/ss2002/kvxp/download/T02%20%5B4%5D%20%20XP%20Einf%FChrung.pdf

[Jansen, Schmitt 2003]: http://www.gm.fh-koeln.de/~winter/pm-ai6/ss2003/V01XP.pdf

[Katterl 2004]: http://www.swe.uni-linz.ac.at/teaching/lva/ws03-04/se_uebung/05_gruppen/g1_ploesch/XP_03.pdf

[Kauflin 2003]: http://www.ifi.uzh.ch/groups/req/courses/seminar_ws03/01_Kauflin_AgileMethoden_Folien.pdf

[Meyer 2004]: http://www.ifi.uzh.ch/groups/req/courses/seminar_ws03/02_Meyer_XP_Ausarbeitung.pdf

[Multi]: http://www.multilateral.ch/index.cfm?nav=03_Development&nav2=01_Methodologie

[Schwartzhaupt 2003]: http://home.arcor.de/susirichter/SWP/extreme.pdf

[Wasserberg 2003]: http://content.tmyaw.com/downloads/xp.2003-06-02.pdf

[Westphal 2001]: http://www.frankwestphal.de/AgileSoftwareentwicklung.html

[Westphal 2004]: http://www.frankwestphal.de/ExtremeProgramming.html

[Wikipedia]: http://de.wikipedia.org/wiki/Extreme_Programming

[XPExchange]: http://www.xpexchange.net/german/intro/aboutXP.html

Deutsche Fassung:
Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008
Genehmigung des Senates am 1.12.2008

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am 08.02.2010 Hassanin Abdalla
(Unterschrift)

Englische Fassung:

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

08.02.2010 Hassanin Abdalla
date (signature)