# 1 UML und Agile Softwareentwicklung

Vieles spricht für ein modellbasiertes Vorgehen: Über ein Diagramm kann man besser debattieren als über tausend Zeilen Code, Lösungen werden klarer, Entwurfsfehler lassen sich früher erkennen. Programmieren geht über Studieren oder eben doch modellieren. Software Engineering liefert jedoch kein universelles Prozessmodell, das für alle Softwareprojekte gleichermassen geeignet wäre. Manche Praktiken, wie z.B. iteratives und inkrementelles Vorgehen, haben sich dennoch als nützlich erwiesen und sind in allen gängigen Vorgehensmodellen integriert. Ich werfe einen Blick dahinter, zeige die entsprechenden Tools und ihre wählbaren Prozesse.

## 1.1 Vorgehensmodelle

Die Unified Modeling Language (UML) bringt in der derzeitigen Version kein Vorgehensmodell (Prozessmodell) mit und wird auch keines mehr vorgeben wollen. Diese Einsicht kam schon im Juni 96 zu Tage, als das Gremium der Object Management Group (OMG) [1] die Unified Method in Unified Modeling umtaufte und so den Standard UML begründetete. Jeder Anwender dieses Standards ist deshalb gezwungen, seine eigene Vorgehensweise zu etablieren. Dies hat aber den Vorteil, UML in schweren (z.B. RUP) wie in leichten Prozessmodellen (z.B. XP) einsetzen zu können. Lange Zeit hat man in UML selbst einen Musterprozess vorgegeben, den „Objectory“, der sich aber nicht wirklich etablieren konnte.

Sicher kennen Sie das klassische Wasserfallmodell, das den organisatorischen Ablauf analog zu einem Wasserfall beschreibt. Die einzelnen Projektzyklen sind in einer fest vorgegebenen Reihenfolge starr zu durchlaufen. Die jeweiligen Phasen laufen sequentiell ab, eine neue Phase wird erst begonnen, wenn die vorhergehende Phase abgeschlossen ist. Diese Art von Vorgehen findet man im Softwarebau höchstens noch auf dem Papier. Auf der anderen Seite spricht man heutzutage von agilen Prozessen, welchen dann der Ruf von zu viel Freiheitsgraden anhaftet. Denn auf der einen Seite baut man bestimmte Praktiken der Softwareentwicklung weiter aus als anderswo, zum anderen lässt man einige etablierte Verfahren vollständig fallen. Diese Radikalität bringt z.B. XP häufig den Vorwurf ein, es mache die konzeptlose „Hackerei“ zum Prinzip.

Beim Versuch, ein Vorgehensmodell einzurichten, erscheint uns vor allem die Zuordnung von Aktivitäten zu Produkten (Ergebnisse) wichtig, nur so lässt sich ein konkreter Phasenplan aufstellen. Die zum Voraus erwarteten Produkte (vor allem Dokumente, Diagramme und Code) sind Grundlagen von Reviews und dienen durch ständiges Hinterfragen auch der Qualitätssicherung.

Ein Prozessmodell stellt eigentlich eine vollständige Beschreibung des Software-Entwicklungsprozesses dar. Es definiert die Aktivitäten, die bei der Entwicklung zu durchlaufen sind, und die Ergebnisse die man dabei erzeugt. Ferner gibt es über den Produktfluss Auskunft, d.h. es legt fest, welche Produkttypen der Entwickler in welchem Zustand als Eingangsinformation für seine nächste Aktivität erwartet und in welchem Zustand ein Produkttyp aus einer Aktivität heraus dem Nächsten zur Verfügung steht.

Der Einsatz eines Prozessmodells bringt gleich mehrfachen Nutzen – innerhalb eines Projekts und über Projektgrenzen hinweg. In jedem Projektmanagement wirft man sich gerne sogenannte W-Fragen an den Kopf, also im Stil: Wer, Was, Wann und Wie. Ein Prozessmodell beantwortet also ständig die Frage:„Wer macht was wann und wie?“, indem es die Rollen aller Projektbeteiligten beschreibt und sie den Aktivitäten zuordnet. Das heisst dann konkret, in der ersten Phase der Initialisierung beantwortet man sich die Frage, mit welchem Prozess und mit welchen Tools (Techniken) gelangt man vom Modell zum Code. Man merke, wer einen Prozess (zeitliches Vorgehen) und eine Technik einsetzt, der hat ein „methodisches“ Vorgehen.

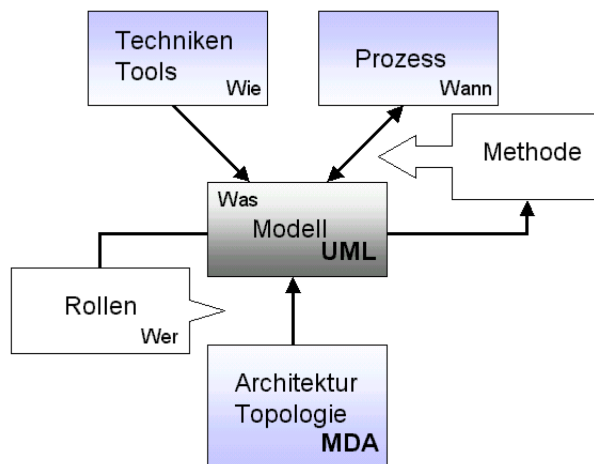


Abb. 1: Das Modellschema zum Prozess

Die UML bietet grafische Ausdrucksmittel an, um Anforderungen zu definieren sowie ein System fachlich und technisch zu entwerfen. Allerdings macht sie keine Aussage, wann, wie und von wem diese Mittel im Projektverlauf zum Zuge kommen. Während meinen Workshops taucht häufig die Frage auf, ob denn die Codegenerierung auch zu einem Prozess gehört. Nun, wie man zum Modell/Code gelangt ist eher eine Frage der Technik, z.B. mit Design Patterns, oder man generiert mit der MDA (Model Driven Architecture) schon in einer frühen Phase ganze Architekturpatterns [[2]]. Auch wenn es mittlerweile standardisierte Codesnippets wie getter und setter-Methoden, IDL-Generierung oder Listenklassen gibt, die sich mit Code-Templates auch erzeugen lassen, hört die eigentliche Generierung innerhalb der Methode einer Klasse auf. Bei der Implementierung einer Methode bzw. einer Member Funktion verwendet man genau die algorithmischen Konstrukte, welche die bewährten Struktogramme oder Flowcharts seit Jahren anbieten. Warum also nicht die klassischen Struktogramme zur Detailspezifikation von Methoden einsetzen, wenn es die Komplexität erfordert ?

Auf der anderen Seite gibt es Prozesse, die ein modellgesteuertes Vorgehen vorsehen [[3]], so dass z.B. aus dem Paketdiagramm auf Stufe Design Patterns möglich ist, Code fast konstruktionsfertig zu erzeugen. Wieweit diese Technik schon als Prozess genügt, zeige ich weiter unten innerhalb der MDA ansatzweise. Jedes Pattern beschreibt eine Musterlösung für ein Problem, das immer wieder vorkommt. Dabei bezeichnet man mit dem Ausdruck Pattern sowohl das Ergebnis, das durch die Anwendung der Regel entsteht, als auch die Regel selbst. Sowenig wie ein Synthesizer das Komponieren guter Musik garantiert, werden die neuen Architekturmethoden und Tools den Entwurf guter Bauten garantieren. Doch wie in der Musik werden sich auch die fähigsten Entwickler der neuen Instrumente bedienen.

Im Folgenden habe ich den Versuch unternommen, basierend auf dem Modellschema eine momentan vage Übersicht zu den bekannten Prozessen herzustellen. Der Begriff „schwer“ ist im Gegensatz zu „agil“ zu betrachten. Fast allen gemeinsam in der Einsatz der UML-Notation und Objektorientierung (OO), iteratives Vorgehen sowie ein vermehrtes Ausrichten auf Patterns. Auch die MDA ist eine Art prozessorientiertes Architekturmuster:

| Prozessmodell | Techniken | Toolbeispiel | Modell | Organisation |
|---|---|---|---|---|
| RUP | OO, MDA | Rose, XDE | UML | Schwer, Teams |
| V-Modell | OO, MDA | ObjectiF | UML | Mittelschwer |
| XP | OO, DUnit | ModelMaker | UML | Leicht, paarweise |
| SELECT Perspective | OO, SAD | S.Enterprise | SAD, UML | Schwer, Abteilung |

### 1.1.1 RUP

Der Rational Unified Process (RUP) ist eine Sammlung von Best Practices, die sich in vielen Projekten bei Kunden und Ämtern bewährt haben. Er kombiniert Technologie und fachspezifische Anleitungen mit vielen Vorlagen und Beispielen. Benutzer können Konfigurations- und Anpassungswerkzeuge der RUP-Plattform dazu nutzen, angepasste Projektvorgaben auf Basis ihrer spezifischen Anforderungen zu erstellen. Der RUP ist ein weit verbreiteter Standard, es zeigen sich aber deutliche Schwächen bei der Weiterentwicklung oder Änderung von Komponenten oder bestehenden Anwendungen, wenn die Organisation sie nicht von Anfang mit RUP entworfen hat.

Im Zentrum von RUP steht die Bewältigung eines komplexen arbeitsteiligen, verteilten und lang andauernden Entwicklungsprozesses durch dessen Einteilung in definierte Produktionsschritte mit definierten Ergebnissen. Dieser tayloristische Ansatz entspricht dem Bestreben des Teams nach Kontrolle und Vorhersagbarkeit.

Der RUP [4] beschreibt die internen Workflows mittels Aktivitätsdiagrammen. Es gibt sechs Kernaktivitäten, die das Vorgehen für Geschäftsmodellanalyse, Anforderungsermittlung, Analyse und Design, Implementierung, Test und Deployment festlegen (siehe Abb. 2). RUP legt bestimmte Rollen, Aktivitäten und Artefakte des Entwicklungsprozesses fest. Einer Rolle sind dann bestimmte Tätigkeiten und Artefakte (Produkte) zugewiesen. Eine Aktivität ist ähnlich dem V-Modell eine Arbeitseinheit, die sich von einem Beteiligten in einer bestimmten Rolle bearbeiten lässt. Für viele der sogenannten Artefakte liefert der RUP Vorlagen, z.B. um die Ergebnisse einer Systemabnahme festzuhalten.

Auf den Vorwurf, einen schweren Prozess zu haben, hat Rational reagiert. Rational und Object Mentor haben auf der Rational User Conference 2002 in Orlando ein Extreme Programming (XP) Plugin für den RUP vorgestellt. Die gemeinsame Entwicklung von Rational und des Vorreiters im XP-Bereich verknüpft Richtlinien und Best Practices von XP mit den Kernelementen des RUP. Damit sollten Entwickler in der Lage sein, Anwendungen schneller und flexibler zu schreiben. Rational und Object Mentor gehören zu den Gründern der Agile Alliance [5], die schnellere und unkompliziertere Software-Entwicklung fördert. Wobei die Gefahr besteht, dass die Leitplanken ausser Sicht geraten, wie auch Franz Zucol meint "Lieber einen umfangreichen, grösseren Rahmen mit der nötigen Freiheit, als die grosse Freiheit um dann doch die Rahmen bestimmen zu müssen".

Das Plugin kombiniert die Kernelemente der iterativen Entwicklung und die Nutzung komponentenbasierter Architekturen aus RUP mit der flexiblen und schnellen Methodologie von XP. Besonders kleinen und mittelgrossen Teams gibt das Plugin mit einer ready to use-Sammlung von Praktiken und Aktivitäten Starthilfe bei der Entwicklung. „Mit dem RUP-Plugin für XP können Entwickler Software einfacher und schneller erstellen, da es die grundsätzlichen Prinzipien des RUP mit den Werten und Best Practices des Extreme Programming verbindet", so Robert C. Martin, CEO, Präsident und Gründer von Object Mentor. Die aktuellen Bemühungen von Rational, den RUP XP-kompatibel zu machen und die Diskussionen im XP-Umfeld über Erweiterungen der Praktiken zeigen, dass sich die beiden Sichtweisen nicht widersprechen, sondern ergänzen können. Kombinationen und Experimente sind aus meiner Sicht nicht nur erlaubt, sondern notwendig und gefragt.

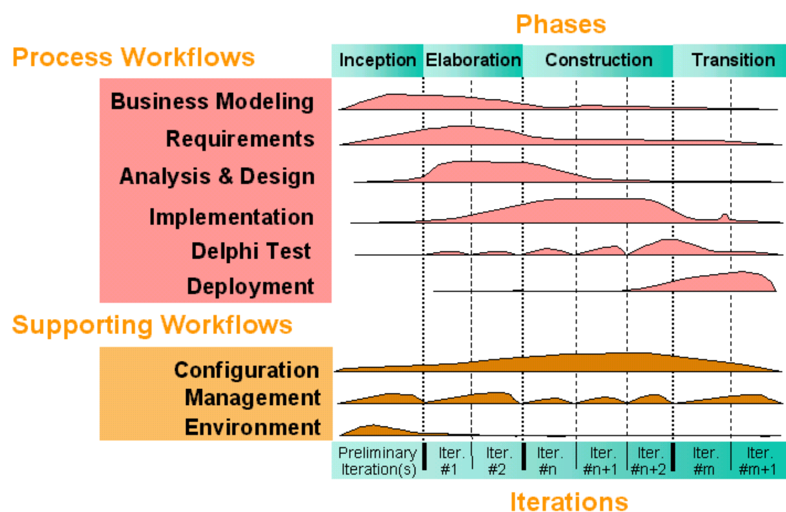


Abb. 2: Der RUP im Blick

Vorteile des RUP:

- Die entsprechenden Tools sind verfügbar
- Bewährte Sammlung an Vorlagen
- Enge Anbindung an die UML-Notation
- Quasi Standard mit grossem Kurs- und Literaturangebot
- Auch für mehrjährige Projekte geeignet

### 1.1.2 V-Modell

Das V-Modell wurde ursprünglich in Deutschland im Auftrag des Bundesministerium für Verteidigung entwickelt und ist dort seit 1992 verbindlich im Einsatz. Im Sommer 1996 wurde es, nach einer 1992 gestarteten Erprobungspha-

se, auch für den Einsatz im zivilen Verwaltungsbereich der Bundesbehörden empfohlen. Damit existiert für die Entwicklung und Wartung von IT-Systemen ein einheitlicher Standard für den gesamten öffentlichen Bereich, der sich von öffentlichen Auftraggebern und privatwirtschaftlichen Auftragnehmern gleichermassen nutzen lässt. Dieses Vorgehensmodell liefert die MID GmbH als vollständiges Referenzmodell aus und ist als Grundlage oder Instanz für ein entsprechendes Projekt anzuwenden.

Im eigentlichen V-Modell kann der Benutzer dann das Vorgehensmodell, das als Referenzmodell dient, an das konkrete Projekt anpassen. Hierzu kann wie in unserem Fall eine V-Matrix dienen. Hierbei bedient man sich vorab spezifizierter Anpassungsregeln, die zu einem konsistenten und standardisierten Projekt führen.

Nebst der eigentlichen Softwareentwicklung (SE) gibt es noch drei andere sogenannten Submodelle, die als Begleitaktivitäten zur eigentlichen Entwicklung zu betrachten sind. Es sind dies im einzelnen:

- PM für Projektmanagement
- SE für Systementwicklung
- KM für Konfigurationsmanagement
- QS für Qualitätssicherung

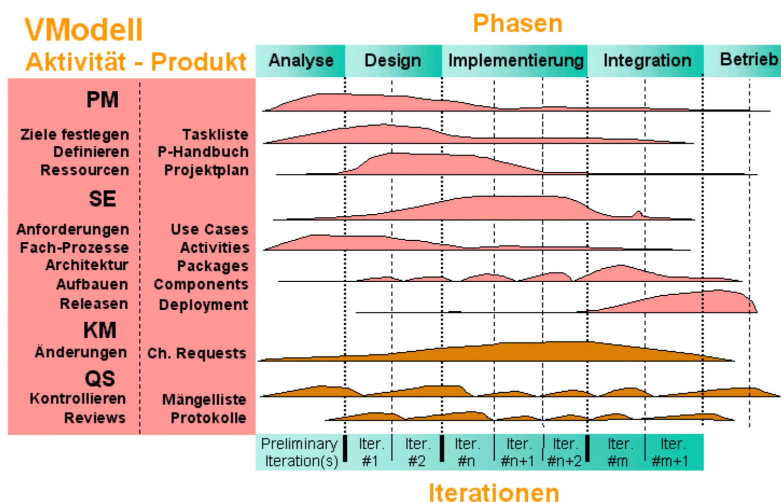


Abb. 3: Iteratives Vorgehen mit V

In den letzten Jahren hat sich in vielen Unternehmen ja die Einsicht durchgesetzt, dass man Softwarequalität verbessern muss. Wie in anderen Bereichen auch liegt der Schlüssel zum Qualitätsmanagement in der Kontrolle des Produktionsprozesses. Nur mit einem nachvollziehbaren Vorgehen ist es möglich, die Qualität der erzeugten Softwaresysteme sicherzustellen. Die 1997 freigegebene Version des V-Modells ist methodenneutral. Während die Version von 1992 nur die Softwareentwicklung nach strukturierten Methoden unterstützte, ist es nun auch möglich, moderne Paradigmas, insbesondere objektorientierte Techniken mit UML, in den Entwicklungsprozess einzubinden.

Bei einer sogenannten Elementarmethode in V steht das „wie" und „wann" im Vordergrund. Das heisst, eine Elementarmethode ergänzt die Aussage darüber, was im Rahmen einer Aktivität geschehen soll, durch die Anleitung, wie und wann es zu tun ist. Die Anwendung von Elementarmethoden ist im allgemeinen mit dem Einsatz spezieller Darstellungsmittel (meistens UML-Diagramme) verbunden, die den Produkttypen zuzuordnen sind. Beim Einsatz der UML mit dem V-Modell ist jedes Darstellungsmittel eindeutig das Ergebnis einer Elementarmethode. Es reicht aus, den Produkttypen Darstellungsmittel zuzuordnen. Die Produktflüsse des V-Modells halten dann fest, in welchen Aktivitäten das Produkt entsteht, und damit ist auch die Verwendung der entsprechenden Elementarmethoden vorbestimmt. Wünschenswert ist es, dass man nur Darstellungsmittel benutzt, die sich gegenseitig ergänzen, d.h. man bringt den UML-Diagrammen die Durchgängigkeit bei. Dadurch ergeben sich weniger Reibungsverluste beim Erstellen neuer Produkte (Ergebnisse), die auf schon vorhandenen Ergebnissen aufbauen oder sie verfeinern.

Wie funktioniert nun das V-Modell? Der Projektleiter ordnet den Projektmitarbeitern auf der Grundlage des Modells Aktivitäten in einer Art Matrix zu. Indirekt ist diese zentrale Matrix auch eine Übersicht des Lieferumfanges und gehört ins Prozesshandbuch. Auf der CD-ROM befindet sich ein Template, genannt V-Matrix, welche als Ausgangslage für jedes Projekt nach V-Modell dienen soll. Die V-Matrix ist eine Art Zuordnungstabelle, die den Einsatz von Techniken und Tools in der entsprechenden zeitlichen Phase aufzeigt und gleichzeitig das Projekt initialisiert [[6]]. Konkret lässt sich in jedem Projekt eine Kopie der Matrix aus einem Fileserver erstellen, mitsamt den Vorlagen, Checklisten und Template. In einem zweiten Schritt passt man die daraus benötigten Aktivitäten an oder streicht sie einfach. Dritter und letzter Schritt ist die Serialisierung der Aktivitäten, d.h. eine vernünftige Reihenfolge auf der

Zeitachse zu bestimmen. Diese Matrix ist dann im Prozesshandbuch für verbindlich zu erklären. Folgende Liste einer Phaseneinteilung der Diagramme in der Domäne SE sind nur exemplarisch als linear anzusehen, da in „real life" ja eine iterative Spirale vorliegt:

| Phase | Modell | Instanzen |
|---|---|---|
| Analyse | Use Case | Fach Szenario |
| Analyse | Activity | Business Units |
| Analyse/Design | Class Diagram | Objekte |
| Design | State Event | Zustandswerte |
| Implementierung | Sequence | Objektszenarios |
| Implementierung | Package | Versionen |
| Integration | Component | Binaries |
| Integration | Deployment | Geräte |

Mit in-Step als konkretem Tool (wird noch vorgestellt) ist folgendes möglich: Es macht alle Aktivitäten eines Projekts mit ihren Ein- und Ausgabeprodukten und ihrer Mitarbeiterzuordnung sichtbar. Damit bietet es dem Einzelnen eine Art To-Do-Liste und dem gesamten Projektteam Orientierung. Jeder sieht auf einen Blick, welche Aktivitäten geplant, bereit, in Bearbeitung oder durchgeführt sind. Diese Information wird über eine einprägsame Ampelsymbolik vermittelt. Gleichzeitig erkennt der Bearbeiter, welche Eingangsprodukte er für eine Aktivität benötigt und welche davon in welchem Zustand vorliegen. Ein Blick auf die Ausgangsprodukte zeigt, welche Produkte man als Ergebnis einer Aktivität erwartet und ob sie bereits vollständig und im richtigen Zustand vorhanden sind.

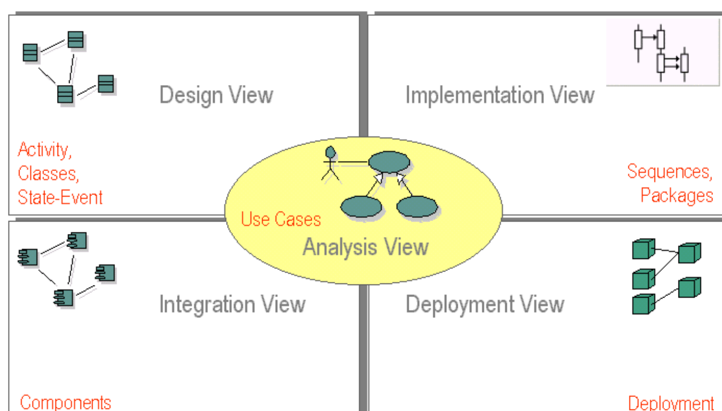


Abb. 4: Auch in V wie in RUP gibt es Phasen wie Sichten

### 1.1.3 XP

Seit etwa mehr als einem Jahr sorgt ein neues Schlagwort für Aufsehen unter den Entwicklern: Extreme Programming (XP) [7]. XP ist in mancherlei Hinsicht extrem. Demgegenüber steht, wie erwähnt, der schwergewichtige Prozess der RUP. Mit dem RUP versucht Rational auch durch Marketinganstrengungen das dazugehörige Produkt zu verkaufen. Während viele Entwickler in flachen Hierarchien mit der schlanken und entwicklungsorientierten Vorgehensweise von XP sympathisieren, muss eine schwerfällige Organisation doch auch einen schweren Prozess haben, könnte man meinen!

XP kombiniert bekannte UML-Techniken zu einem neuartigen Prozess, der auch denen dienen soll, die bis anhin keinen Prozess hatten. Die Entwicklung eines Lohnabrechnungssystems bei Daimler Chrysler diente als Pilot dazu. Spätestens seit Kent Beck's Buch und der Keynote auf der OOP'99 in München ist das Thema auch in Deutschland bekannt. Einen hohen Stellenwert bei XP hat die direkte Kommunikation zwischen den Mitgliedern, sowohl zwischen Entwicklern als auch zwischen Entwicklungsteam, Kunden und Anwendern. Dies wird durch kleine Teams, vor allem paarweises Programmieren und der direkten Mitwirkung des späteren Benutzers gefördert. Eingespielte Kommunikation soll bei XP die meisten Dokumente überflüssig machen, so dass XP Wert auf ein schlankes System legt.

Nach etlichen Recherchen habe ich in einem State-Event (Abb. 6) die wichtigsten Schritte von XP modelliert. Die Applikation ist stufenweise in kurzen Iterationen erweiterbar. Vor jedem Zyklus stehen die Anforderungen des Kun-

den, von denen die Entwickler den Aufwand schätzen. Der Kunde wählt dann die tatsächlich zu realisierenden Aufgaben aus. Das Ergebnis jeder Iteration ist eine lauffähige Software. Wichtig ist nun folgendes: Das Entwicklerpaar schreibt vor dem Bau einer neuen Funktionalität einen Test, welcher das gewünschte Verhalten festlegt. Mit Hilfe von Techniken wie DUnit [[8]] kann man auf Knopfdruck jederzeit kontrollieren, ob der Test zum Erfolg führt. Zudem ist das Refactoring, d.h. die Umstrukturierung der Unit ohne ihre Funktionalität zu verändern, ein wesentlicher Bestandteil von XP. Durch meist kleine Schritte, wie Felder kapseln, Parameter extrahieren oder Vererbung durch Delegation ersetzen, wird der Code kontinuierlich klarer und verständlicher, d.h. der Code dokumentiert sich fast selbst. Hier wissen wir alle, dass nur Superprofis einen Blick für solch selbstlesende Code-Muster haben. Ist eine Klasse mit eigenen Zustandsvariablen bestückt oder will man die möglichen Ereignisse mit ihren Übergängen genauer festhalten, dann sind zusätzliche State-Events oder Sequenzdiagramme immer erforderlich.

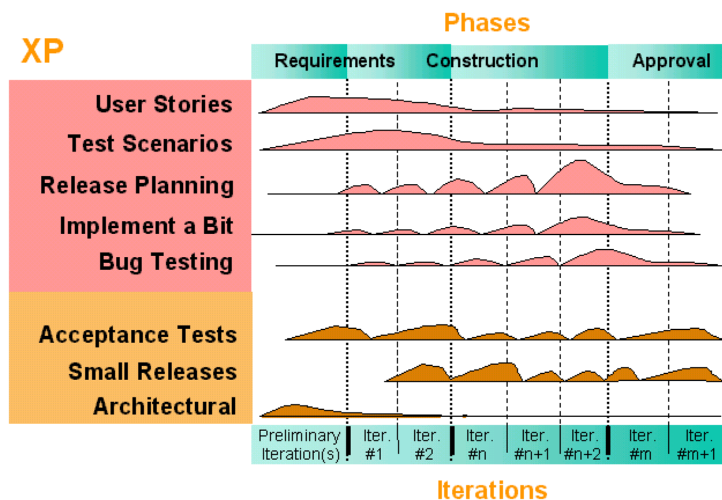


Abb. 5: XP mit schlanken Iterationen zum schnellen Testen

Unter bestimmten Bedingungen scheint XP das ideale Vorgehensmodell zu sein, weil es sowohl Entwicklern als auch Kunden und Benutzern entgegen kommt. Diese Bedingungen sind:

- Kunde verfügbar und vor Ort
- Mittlere Projektgrösse (2 - 8 Entwickler)
- Kommunikative Mentalität

Erste positive Erfahrungen haben wir in einem Fall mit folgendem Vorgehen gemacht: Die XP-Release-Planung kann man durch Use Case ersetzen, der Kunde priorisiert die Use Case, und die Iterationsplanung findet dann anhand der detaillierten Use Case mit Aktivitätsdiagrammen statt. Natürlich nimmt die Gefahr zu, dass man spekulativ entwikkelt, weil der Kunde nicht immer für offene Fragen zur Verfügung steht und das rudimentäre Design keine Antwort liefert. „Trotzdem bleiben die Türme von Hanoi stehen“, meint Franz Zucol. Erst die Erfahrung der kommenden Jahre wird zeigen, wie weit XP sich bestätigt, ohne seinen eigentlichen Vorteil, leichtgewichtig zu sein, wieder einzubüssen, da ja gleichzeitig qualitativ hochwertige und kundenorientierte Software entstehen soll, die meistens sorgfältiges Design erfordert.

### 1.1.4 XP Techniken

Die folgenden Techniken lassen sich auch in anderen Prozessen gewinnbringend einsetzen, demzufolge z.B. das Refactoring [[9]] eine Renaissance erlebt. Im Refactoring lässt sich das Design fortlaufend in kleinen, funktionserhaltenden Schritten verbesseren. Finden zwei Entwickler Codeteile, die schwer verständlich sind oder unnötig kompliziert erscheinen, verbessern und vereinfachen sie den Code. Sie tun dies in disziplinierter Weise und führen nach jedem Schritt die Unit Tests [[10]] aus, um keine bestehende Funktion zu zerstören. Gewöhnlich wird jeder Funktionsblock durch einen Testfall untermauert. Die Unit Tests werden gesammelt, gepflegt und nach jedem Kompilieren ausgeführt. Jeder Testfall wird auf die einfachste denkbare Weise erfüllt. Es wird keine unnötig komplexe Funktionalität programmiert, die momentan nicht gefordert ist

Die Programmierer arbeiten stets zu zweit am Code [[11]] und diskutieren während der Entwicklung intensiv über Entwurfsalternativen. Sie rotieren in der Regel stündlich ihre Arbeit. Das Ergebnis ist eine höhere Codequalität, verbesserte Produktivität und erhöhte Wissensverbreitung. Der gesamte Code gehört dem Team. Jedes Paar soll jede Möglichkeit zur Codeverbesserung jederzeit wahrnehmen und einbringen. Eine fortlaufende Integration nach mehrmals täglichem Build-Prozess stellt durch eine Versionsverwaltung sicher, dass man die Termine einhält. Damit erreicht

man, die als Timeboxing bekannte Unveränderbarkeit des Termins zu umgehen, da am Ende einer Iteration bereits die Detailplanung der nächsten Iteration steht.

### 1.1.5 Vergleich XP und RUP

Ein Vergleich birgt immer Zündstoff, zumal hinter RUP eine mächtige Industrie steht, die dem kostenlosen Kontrahenten XP Paroli bieten wollen. Im Gegensatz zu RUP wird bei XP eine vertikale Gruppeneinteilung vorgeschlagen, da in XP eine Spezialisierung unerwünscht ist, jedes Teammitglieder sollte alle Teile des Systems kennen. Wechselnde Programmiererpaare betonen diesen Aspekt. Eine Rollenverteilung in RUP bevorzugt Spezialisierung (Analytiker, Designer, Codierer, Tester, etc.). Frappant ist der Unterschied in der Designstrategie festzustellen. XP verwirklicht iteratives Design, das einfach beginnt und auf der Technik des Refactoring basiert. RUP startet mit einem möglichst vollständigen Design. Modell und Implementierung sind dann laufend synchronisiert (Roundtrip). XP stellt Kommunikation über Dokumentation. Reviews dienen dazu, fehlende Dokumentation durch Audits zu kompensieren. Das Management beschränkt sich in einem XP-Projekt auf wenige Aktivitäten. Diese umfassen hauptsächlich Kontrollen der Zeitschätzungen der Teammitglieder und das Anfertigen von Statistiken als Spiegel des Projektfortschritts. Im RUP kommt dagegen Detailplanung und Plankontrolle zum Einsatz.

In all diesen Eigenschaften zeigen sich die unterschiedlichen Schwerpunkte von RUP und XP: Im Zentrum von XP steht das Bestreben, die zentrale Tätigkeit des Softwarebaus im Team aufzuwerten, und die Entwickler mit Tools und Techniken auszustatten, die diese gekonnt einsetzen können.

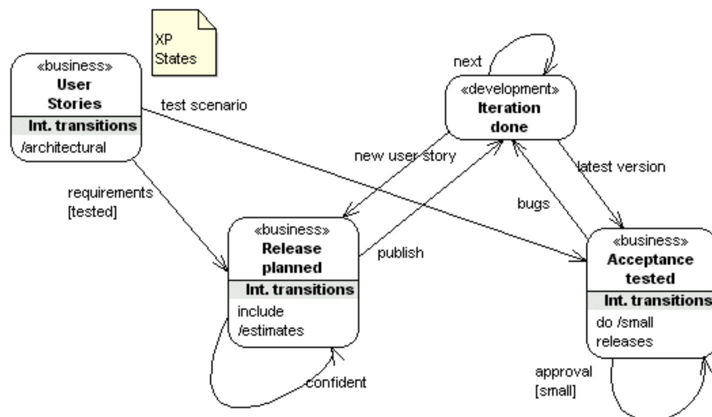


Abb. 6: Dynamischer Ablauf des XP Prozessmodells

## 1.2 MDA und UML 2.0

Dass ein Architekturstil nicht nur etwas mit dem Bau von Komponenten zu tun hat, sondern auch mit den Personen und Rollen in der Organisation, die solche Komponenten generieren, macht MDA als Technik wie auch als Prozess interessant. Die bereits erwähnten Prozessmodelle sind vor allem Use Case getrieben, auch XP setzt mit den User Stories die Anforderungen an den Anfang. Wenn aber die Anforderungen schon im Modellansatz bestehen, spricht wenig dagegen, diese modellgetriebene Architektur bei ähnlichen Anforderungen immer wieder einzusetzen. MDA gibt es bereits für existierende Software [[12]], die sich mittels Reverse Engineering umwandeln lässt. Zumal MDA auf der höheren Abstraktion ja sprachunabhängig ist. Nun, was meint Model Driven Architecture (MDA) im Alltag?

Jede Sprache mit der zugehörigen Entwicklungsumgebung ist von einer inneren Architektur abhängig, z.B. dem Framework der GUI, der Struktur zwischen Interface und Implementierung oder den typischen Datenbankzugriff-Komponenten mit ihren Connection-Strings. Ein UML-Generator sollte diese Architektur kennen, ansonsten man die technischen Details in den Modellen einbauen muss. Mein Paketdiagramm sollte also bereits wissen, ob ich z.B. dbExpress statt ADO einsetzen werde, wenn ich das Modell zum Generieren benutze.

Um diese „Probleme“ bei der Generierung zu lösen, schlägt die OMG vor, ein „Platform Independent Model“ (PIM) durch einen MDA-Generator zu erzeugen. Ein PIM ist ein UML-Modell, in dem die technischen Details nicht ersichtlich sind. Somit befindet sich das Modell auf einer höheren Abstraktionsebene und ist unabhängig von technischen Fakten wie der bekannten Programmiersprache oder Datenbanktechnik. Das PIM muss aber genug Details besitzen, um das Modell durch definierte Abbildungsregeln und Architekturmuster auf die Zielsprache abzubilden, d.h. zu generieren. Wer sich jetzt fragt, wie die Abgrenzung zu Design Patterns zu setzen ist, der weiss aus eigener Erfahrung,

dass ein Tool aus einem Pattern direkt sprachspezifischen Code produziert. So etwas wie ein Design Pattern Generator für Geschäftsprozesse gibt es nicht. Also positioniert sich die MDA einen Schritt vor das Design und lässt in der Analyse bereits sprachunabhängige Klassendiagramme generieren.

Kernidee der MDA ist also die schrittweise Verfeinerung des Design ausgehend von einer Modellierung der Applikations-/Geschäftslogik, die völlig unabhängig von der technischen Implementierung und der umgebenden IT-Infrastruktur ist. Dieses PIM-Modell könnte grob so aussehen:

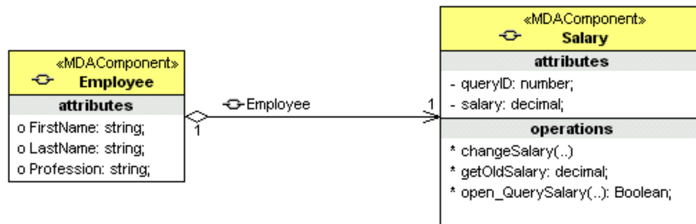


Abb. 7: Ein kleines MDA

Das kleine Modell soll das abstrakte Typensystem verdeutlichen, ähnlich einer IDL, d.h. jeder mögliche Wert der `queryID` kann als Typ eine Zahl sein, z.B. integer, longint etc. Der neuralgische Punkt des MDA ist die exakte Definition einer Plattform. Wenn nun das Tool aus dem PIM ein plattformspezifisches PSM generiert, sollte Einigkeit über die zu implementierende Plattform herrschen. Ist nun eine Plattform CLX, .NET oder J2EE, was ist wenn die Plattform eine Sprache mit Objektmodell oder sogar eine Spezifikation wie CORBA hat. Im Hinblick auf die Ausarbeitung solcher Details steht die MDA-Bewegung noch ziemlich am Anfang. Bleiben wir optimistisch.

### 1.2.1 Vom PIM zum PSM

Mit der Unterscheidung zwischen PIM und PSM definiert die MDA einen Prozess für die Entwicklung von Systemen in der Analyse-Phase. Durch Transformation lässt sich aus einem PIM ein PSM erzeugen. Im PSM sind Informationen über die eingesetzte Softwareinfrastruktur enthalten, wie z.B. ein J2EE-Applikationsserver, ein dbExpress Provider, oder ein .NET-Framework.

Die Transformation kann ausgehend vom PIM über mehrere unterschiedlich abstrakte PSM hin bis zum Code erfolgen. In der MDA sind die notwendigen Schritte bei dieser Transformation definiert und durch den Einsatz von Tools halb oder vollständig automatisierbar. Der Vorteil: Beim Austausch einer Middleware muss der Entwickler also nur die geeignete Transformation verwenden, um aus dem unveränderten PIM ein neues, passendes PSM zu generieren. Zentral ist hier das Modellieren der Geschäftslogik und -anwendung ohne Details zu ihrer technischen Umsetzung. Darauf aufbauend werden technologiespezifische Modelle erstellt, die eine konkrete Umsetzung in einer gewählten Technologie, wie z.B. J2EE, CLX, oder .NET beschreiben.

Die Aufteilung in plattformunabhängige und plattformspezifische Modelle ermöglicht eine langfristige Anpassbarkeit an zukünftige Technologien. Auch das Wissen über die zugehörige Fachlogik soll einfliessen. Für Domänen wie z.B. Finanz- oder Versicherungswesen, Telekommunikation, Medizinaltechnik sollen die typischen abstrakten Prozesse schon vordefiniert sein und für die Erstellung der PIM als sogenannte „domain-specific core models“ bereitgestellt werden. Für den Austausch der Modellinformationen über Toolgrenzen hinweg wird XML Metadata Interchange (XMI) eingesetzt. UML und XMI sind ebenfalls Standards der OMG.

### 1.2.2 Der Turmbau zu UML

Der MDA Ansatz birgt zusätzliches Potential für die Wiederverwendbarkeit von Modellen, da diese ja plattformunabhängig sind. Durch den Einsatz von standardisierter Transformation ist auch die Qualitätssicherung erhöht. Nach OMG sind dies die Hauptvorteile von MDA:

- Reduzierte Kosten bei der Entwicklung
- Erhöhte Qualitätssicherung
- Schnellere Integration neuer Technologien

Auch zum Modellaustausch hat die OMG das entsprechende Konzept, XMI, schon als Standard verabschiedet. Der Name zeigt es bereits deutlich: XMI ist ein Mitglied der XML-Sprachfamilie. Die XMI-Norm definiert für das UML-Metamodell eine XML Document Type Definition (DTD). Sie stellt die Grammatik der Sprache zur Verfügung.

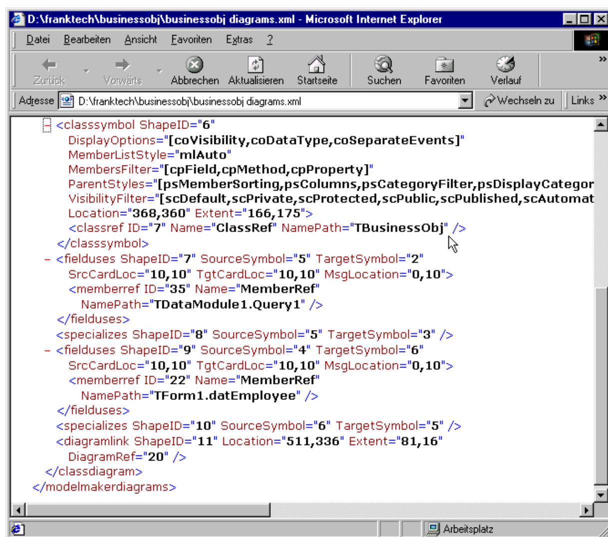


Abb. 8: Export der MDA-Modelldaten als XMI

Es ist festzustellen, dass die meisten UML-Tools sich betreffend der verfügbaren Diagramme und Modellinformationen einander nähern. Auch wenn z.B. mit Stereotypen die Notation erweiterbar ist, die Elemente normieren sich. Diese Tatsache führte zum Bedürfnis, diese Modelldaten auch unter den CASE-Tools austauschen zu können. Einfach gesagt, besteht eine XMI-Datei aus einem Header- und einem Content-Bereich. Der Header bestimmt das Metamodell und der Content demzufolge das Modell mit den zugehörigen Elementen. Das Metamodell hat als oberste Klasse das Element. Diese Elemente sind dann pro Diagramm verfügbar.

In einer heterogenen Entwicklungslandschaft, auch im Hinblick auf UML 2.0, lassen sich künftig Ergebnisse zwischen verschiedensten Tools austauschen. Manchmal steht dahinter eine weitreichendere, strategische Entscheidung, nämlich sich als Unternehmen nicht von den Herstellern der eingesetzten Tools abhängig zu machen und aus einem beliebigen Tool „Architektur“ zu erzeugen.

Auch in der modernen Architektur ist der Detaillierungsgrad anscheinend verlorengegangen: Die auf dem Brett entstandene Zeichnung und das architektonische Resultat haben sich für unnötig erachtete Ornamente und Schnörkel erwiesen. Doch die innere Komplexität der Gebäude ist gewachsen. Aussenwände sogenannter intelligenter Gebäude (Intelligent Buildings) verbergen hinter der glatten Fassade ein äusserst kompliziertes Innenleben. Damit wird klar, dass nebst der Architektur auch die Funktionalität einen hohen Detaillierungsgrad aufweisen kann. Der Trend wird sein, die Architektur zu vereinfachen (J2EE, .NET) aber die Funktionalität zu erhöhen. Denn in den seltensten Fällen wird künftig jeder Teil eines Modells ein anderer Typ sein, das entspräche der Arbeit mit Einzelelementen. Ziel ist die mehrmalige Wiederverwendung von Variationen eines Standardtyps, um so ein generelles Ordnungsprinzip zu erreichen. Techniken wie ein Common Type System, zielen jetzt schon darauf ab.

## 1.2.3 Generative Programmierung

Weitere Techniken sind im Gespräch, die alle eine gewisse Gemeinsamkeit mit MDA aufweisen, die sich „generative Programmierung mit adaptiven Mustern“ nennt. Diese Technik lohnt sich vor allem dort, wo man lange in der gleichen Fachrichtung entwickelt, also gute Kenntnisse der Business Domain hat. Die Umgebung von Fachanwendungen impliziert meistens eine ähnliche Struktur, so dass weitere Anwendungen wie eine Instanz aus einem Metamodell entstehen könnten. Bei der Entwicklung eines CAD-Tools oder einer Software aus der Regelungstechnik sehe ich kaum eine Chance, von einem besseren Reuse als bisher zu profitieren. Auch der Begriff „Intentional Programming“ gehört zur Thematik, welcher voderhand ein interessanter Microsoft-Traum ist, aber durchaus Potential im Zusammenhang mit dem Applikations-Framework .NET besitzt.

Muster im Grossen wie im Kleinen, von Architekturmustern über Entwurfsmuster bis hin zu Codemustern, spielen bei der objektorientierten Entwicklung eine Rolle, wenn es darum geht, Effizienz und Qualität gleichermassen entscheidend zu steigern. Ohne ein Tool allerdings, das die Verwendung solcher Muster aktiv unterstützt, ist nicht viel gewonnen. Zu vieles beschränkt sich dann auf das fehleranfällige, manuelle Abschreiben und Kopieren von Vorlagen.

Die Musterbasierte Anwendungsentwicklung soll diese Schwierigkeit umschiffen. Mit PBE (Pattern by Example) ist eine auf Beispielen basierende Entwicklungstechnik realisierbar. Bewährte Lösungen lassen sich schrittweise in automatisch wiederverwendbare Muster überführen. In PBE kann man Musterdefinitionen erstellen, durch einfaches Markieren parametrisieren und auf Knopfdruck für die gewünschte Zielsprache expandieren. Bei einer konventionellen Vorgehensweise sind Design Patterns, die als theoretische Konzepte existieren, immer wieder neu im jeweiligen

Kontext zu implementieren; mit PBE werden sie jeweils nur einmal als Musterdefinition implementiert und sind dann für verschiedene Sprachen und Kontext erweiterbar.

Kommen wir zur Template Metaprogrammierung, im folgenden auch „statisch" genannt, da der Compiler sie im Gegensatz zu dynamischen Programmen nicht zur Laufzeit, sondern während der Übersetzung eines Programms ausführt. Während „normale" Programme Daten verarbeiten, verarbeiten Metaprogramme andere Programme. Das erste Metaprogramm geht wohl auf Erwin Unruh zurück, der einen C++-Compiler dazu überredete, Primzahlen als Warnungen auszugeben. Ein einfaches Template zum Schluss zeigt einen ersten Ansatz, das wohl jedes Tool schon besitzt:

```
unit ArrayProp List;
//DEFINEMACRO:Items=name of array property
//DEFINEMACRO:TObject=type of array property
//DEFINEMACRO:ItemCount=Method returning # items
//DEFINEMACRO:FItems=TList Field storing items

  TCodeTemplate = class (TObject)
  private
    <!FItems!>: TList;
  protected
    function Get<!ItemCount!>: Integer;
    function Get<!Items!>(Index: Integer): <!TObject!>;
  public
    property <!ItemCount!>: Integer read Get<!ItemCount!>;
    property <!Items!>[Index: Integer]: <!TObject!> read Get<!Items!>;
  end;
```

Nach dieser hinreichenden Übersicht erfahren Sie Aktuelles von folgenden Tools:

- ModelMaker (Eigener Artikel)
- MetaBase von gs-soft
- ObjectiF von MicroTool
- Sparx Enterprise von Sparxsystems
- XDE von Rational
- StP von Aonix

Quellen und Links: siehe Fussnoten

Max Kleiner, November 2002, max@kleiner.com

---

[1] www.omg.org

[2] Kleiner: Designermodell in: Der Entwickler 1.2001

[3] Kleiner, Angerer: Delphi Design Patterns, S&S Verlag, 2003

[4] Kruchten: The Rational Unified Process, Addison Wesley, 2000

[5] www.agilealliance.org

[6] Kleiner: UML mit Delphi, S&S Verlag, 2000

[7] Kent Beck: Extreme Programming Explained, Addison Wesley, 1999

[8] DUnit: http://sourceforge.net/projects/dunit/

[9] Martin Fowler: Refactoring - Improving the Design of Existing Code. Addison Wesley, 1999.

[10] Immo Wache: DUnit Rahmenwerk in: Der Entwickler 5.2001

[11] www.extremeprogramming.org/rules/pair.html

[12] www.liantis.com