Zusammenfassung Stoff des Moduls SE3

Version: 0.2
Autor: Michael Klenk mklenk@hsr.ch
Grundlage: Die Illustrationen und der Inhalt dieser Zusammenfassung basieren auf den Unterrichtsunterlagen zum Modul SE3 der HSR von Thomas Letsch, Lothar Müller, Grässle und Schacher

Inhaltsverzeichnis:

Generics ... 2
Wildcards ... 2
Raw-Type ... 2
Soft-, Weak, und Phantom Referenzen ... 3
ReferenceQueue ... 3
Aspekt-Orientierte Programmierung AOP ... 4
AspectJ ... 4
Agile Methoden ... 5
7 Prinzipien (nach Cockburn) ... 5
6 Konsequenzen (nach Cockburn) ... 5
Agiles Manifest ... 5
Methodenvergleich ... 6
Scrum ... 7
Feature Driven Development (FDD) ... 8
XP Extreme Programming ... 9
Crystal ... 11
Requirements Engineering ... 13
Anforderungen ... 13
Beziehung zwischen Anforderungen ... 13
Formulierung von Anforderungen ... 13
Advanced UML ... 14
M:M Beziehung ... 14
Dreieck-Struktur ... 14
Doppel V Struktur ... 15
Verhaltensmodellierung ... 15
Parallelzustände ... 16
Object Constraint Language OCL ... 16
Expressions in OCL ... 17
xUML ... 20
Black-Box Modeling ... 20
xUML Model ... 20
Actions in xUML ... 21
Control Structures in xUML ... 22
Aspect Oriented Analysis ... 23
Problem ... 23
What are Aspects? ... 23
Implementing Aspects ... 23
Summary ... 24
Metamodeling ... 24
UML Profiles ... 25
UML Stereotype ... 25
Domain Specific Languages ... 25
Model Driven Architecture ... 26
Why UML is not enough ... 26
Basics of MDA ... 26
MDA Roles ... 27
Transformations ... 27
Code generation ... 28
Geschäftsprozessmodellierung ... 28
Begriffe ... 28

Detaillierte Prozessmodelle 29
Leichte Prozessmodelle 29
Wie Geschäftsprozesse modellieren? 30
Business Rules Ansatz 31
Begriffe 31
Geschäftswissen 31
Darstellung von Geschäftsregeln 31
Implementierung von Geschäftsregen 32
Daten-Schnittstelle 34

Generics

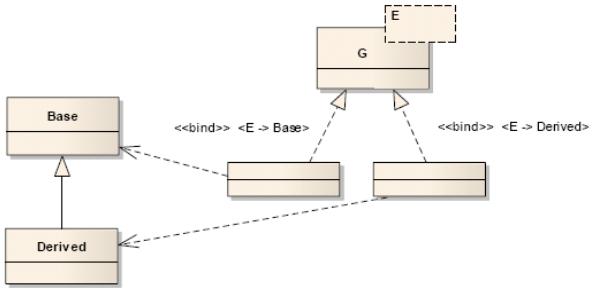
Ziel ist es generische Klassen und Container zu schreiben, die es trotzdem erlauben zur Kompillierzeit Typenfehler zu entdecken.

Definition:

```
class Classname<E> {}   Classname<Type> = new Classname<Type>();
```

<E>: Formaler Typ-Parameter (formal type parameter) Oft auch Typ-Variable (type variable) genannt.

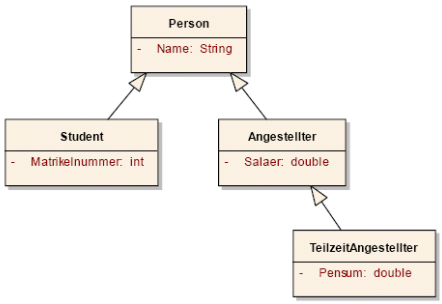
Achtung!
Bei Zuweisungen muss man aufpassen.
G<Derived> kann nicht G<Base> zugewiesen werden! G<Derived> ist **kein** abgeleiteter Typ von G<Base>!



Wildcards

Bei Methoden welchen Listen von unbekanntem Typ übergeben werden, können Wildcards verwenden um die Typsicherheit zu garantieren.

```
void printCollection(Collection<?> c) {
      for (Object e : c) {
            System.out.println(e);
      }
}
```



Upper Bound Wildcards

Problem: Der Parameter muss eine Liste von Personen oder davon abgeleitete Klasse sein!

```
List<String> erzeugeNamensListe(List<? extends Person> pPersonenListe)
```

Wichtig! Schreibender Zugriff auf diese Liste ist **nicht möglich**, da nicht sichergestellt werden kann dass die eingefügte neue Person kompatibel zu der übergebenen Liste ist.

Lower Bound Wildcard

Problem: Der Parameter muss eine Liste von Angestellter oder einer höheren Klasse sein.

```
void alleAngestellte(List<? super Angestellter> pAngestellte)
```

Wichtig! Lesender Zugriff auf diese Liste ist **nicht möglich**, da nicht sichergestellt werden kann dass das zurückgegebene Objekt zu Angestellter kompatibel ist.

Raw-Type

Es gibt von generischen Klassen jeweils nur eine Definition. Bei der Verwendung einer generischen Klasse ohne Typ spricht man von Raw-Type. Dieser Typ wird verwendet um kompatibel zu alten Java Versionen zu sein.

Erasure

Erasure löscht alle generische Typ-Informationen und ersetzt diese durch den Upper-Bound der Typ-Variablen (normalerweise Object), generische Typen durch deren Raw-Type und fügt wenn nötig Casts hinzu.
Bei der Verwendung eines Raw-type kann es vorkommen dass beim Erasure Prozess zusätzliche Bridge Methoden erzeugt werden müssen, welche Methoden für verschieden Typen anbieten und intern einen Cast vollziehen.
Erasure löscht alle generische Typ-Informationen und ersetzt diese durch den Upper-Bound der Typ-Variablen (normalerweise Object), generische Typen durch deren Raw-Type und fügt wenn nötig Casts hinzu.

Soft-, Weak, und Phantom Referenzen

Wird ein Objekt nicht mehr über ein anderes Object referenziert kann es über den Garbage Collector gelöscht werden. Beim löschen ist zu berücksichtigen dass Objekte in der finalize Methode wiederbelebt werden können (Resurrection, Achtung finalize wird nur 1-mal aufgerufen!). Es ist nun der Bedarf da dass Objekte angelegt werden können welche trotz Referenzen bei Bedarf von Garbage Collector gelöscht werden können. Lösung: Schwache Referenzen. In Java gibt es 3 Varianten dieser Referenzen:

- Soft-Referenzen
  "Behalte Objekt so lange es geht. Lösche erst, wenn nicht mehr genügend Memory vorhanden ist."
- Weak-Referenzen
  "Lösche Objekt sobald es von der Applikation nicht mehr referenziert wird.
- Phantom-Referenzen
  "Benachrichtige mich bevor das Objekt gelöscht wird."

ReferenceQueue

Eine Referenz-Queue erlaubt es dem Programm herauszufinden, wann ein Objekt Soft-, Weak- resp. Phantom-Reachable wird. Wenn Objekt in seinen schwachen Zustand kommt (soft, weak, phantom) wird die entsprechende schwache Referenz vom Garbage-Collector in die Queue geschrieben.
Zugriffe auf diese Liste sind möglich über die Methoden:
`public Reference<? extends T> poll()`
Löschen und Rückgabe der nächste Referenz oder null wenn leer.
`public Reference<? extends T> remove()`
Löschen und Rückgabe der nächste Referenz. Blockierendes Warten.
`public Reference<? extends T> remove(long timeout)`
Löschen und Rückgabe der nächste Referenz oder null nach timeout.

Aspekt-Orientierte Programmierung AOP

Die Idee von Aspektorientierter Programmierung ist es, Programmlogik, Security, Logging, Persistenz und andere völlig unabhängige Anforderungen an ein Programm sauber trennen zu können.
Die Idee, Code-Fragmente z.B. für Security können nachträglich an den gewünschten Stellen in die Applikationslogik eingewoben werden. Dazu braucht es zur Laufzeit einen sogenannten Weaver.

AspectJ

Pointcut	Eine Stelle an der Code eingefügt werden kann. Entspricht einem Filter auf alle möglichen Join Points
Advice	Ein Code Stück das eingewoben werden soll
Aspect	Pointcut + Advice
Join Points	Mögliche Stellen an welchen Code eingefügt werden kann. Methoden oder Konstruktoren aufrufe, Zugriffe auf Attribute, Exceptions

Pointcut

Name	Funktion
call()	Aufruf einer Methode. Der Kontext ist derjenige der aufrufenden Methode.
execution()	Ausführung einer Methode. Der Kontext ist derjenige der aufgerufenen Methode.
set()	Zuweisung eines Attributes
get()	Referenzieren (lesen) eines Attributes
handler()	Ausführen eines Catch Blockes
within()	Ausführung eines Join Points innerhalb des Source-Codes eines bestimmten Types. Bestimmung zum Kompilationszeitpunkt.
this()	Ausführung eines Join Points des aktuellen Objektes, wenn dessen Laufzeit-Typ eines bestimmten Types entspricht. Bestimmung zur Laufzeit.
target()	Ausführung eines Join Points des Ziel-Objektes, wenn dessen Laufzeit-Typ eines bestimmten Types entspricht. Bestimmung zur Laufzeit.
args()	Ausführung eines Join Points wenn dessen Argumente den definierten Typen entsprechen.

Pointcuts können kombiniert werden. Negation **!**, UND **&&,** OR **||**
".." in der Parameterliste bedeutet beliebige Parameter, ein "*" steht für einen beliebigen Typ.
Ein Name Type Patter gefolgt von einem "+" trifft auf diesen Typ und alle Subtypen zu.

Advice

Name	Funktion
before()	Wird vor dem Joint Point ausgeführt.
after()	Wird nach dem Joint Point ausgeführt.
after() returning	Wird nach dem Joint Point ausgeführt und erlaubt Zugriff auf return-Wert.
after() throwing	Wird nach dem Joint Point ausgeführt wenn eine Exception geworfen wurde und erlaubt Zugriff auf diese Exception.
around() und proceed()	Anstelle des Joint Point wird around ausgeführt. Innerhalb von around kann der Join Point mit proceed zur Ausführung gebracht werden.

Agile Methoden

7 Prinzipien (nach Cockburn)

1. Interactive face-to-face communication is the cheapest and fastest channel for exchanging information.
2. Exess methodology weight is costly.
3. Larger teams need heavier methodologies.
4. Greater ceremony is appropriate for projects with greater criticality.
5. Increasing feedback and communication reduces the need for intermediate deliverables.
    a. Deliver a working peace of the system quickly enough that the sponsor can tell whether the team understood the requirements properly.
    b. Reduce the team size, putting everyone enough together that they can simply tell each other what they are doing instead of writing internal documents to each other.
6. Discipline, skills, and understanding counter process, formality, and documentation.
7. Efficiency is expendable in non bottleneck activities.
    a. Do whatever you can to speed up the work at the bottleneck activity.
    b. People at the non bottleneck activities can work inefficiently without affecting the overall speed of the project!
    c. Applying the principle of expendable efficiency yields different methodologies in different situations, even keeping the other principles in place.

6 Konsequenzen (nach Cockburn)

1. Adding people to a project is costly.
2. Team size increases in large jumps.
3. Teams should be improved, not enlarged.
    a. Send them to courses to improve their skills.
    b. Seat them closer together to reduce communication cost.
    c. Improve their amicability and teamwork.
    d. Replace some of the people on the team with more talented (and more highly paid) people.
4. Different methodologies are needed for different projects.
5. Lighter methodologies are better, until they run out of steam.
6. Methodologies should be stretched to fit.
    a. choose larger-category methodology and remove excess weight
    b. choose smaller-category methodology and adopt it up

Agiles Manifest

Alle Agilen Software Entwicklungsmethoden basieren auf dem Agilen Manifest und den dazugehörigen agilen Prinzipien. Das Manifest enthält keine genauen Richtlinien wie genau die Software Entwicklung zu geschehen hat.

"We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more.“

12 Principles

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
3. Working software.
4. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
5. Business people and developers must work together daily throughout the project.
6. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. The best architectures, requirements, and designs emerge from self-organizing teams.
8. Continuous attention to technical excellence and good design enhances agility.
9. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
10. Simplicity - the art of maximizing the amount of work not done - is essential.
11. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.

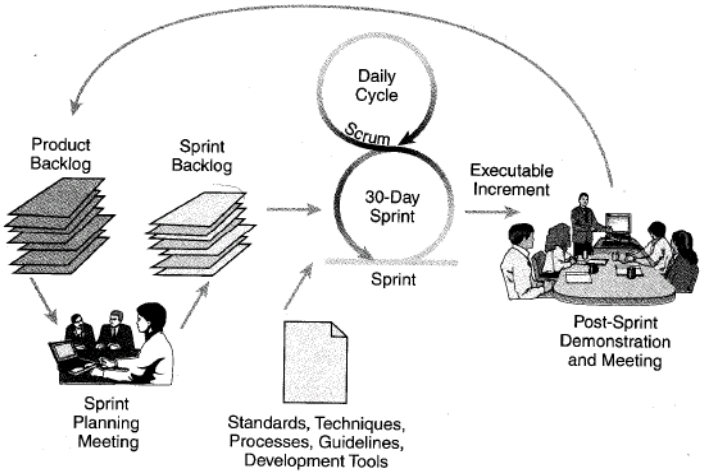
Methodenvergleich

	Crystal Clear	**XP**	**Scrum**	**FDD**
iteration cycle delivery cycle	< 3m < 3m	1d – 1m ?	Daily Cycle in a 30day Sprint	Not fix
programming	Pair code reviews	pair programming	Not mentioned	chief programmer pair programming code inspections
automatic unit testing	Recommended	Required	Not mentioned	Not mentioned
customer	Easy access, > 1h /w	On site, fulltime	Role of product owner	features scheduled early
coding conventions	Not mentioned	Required	Free team organisation	Not mentioned
Simple design, refactoring	recommended	Required		overall model first, get it right the avoid unnecessary refactoring
documentation	Required	Not required	Not mentioned	Not mentioned

Scrum

Srcum ist eine sehr einfache und leicht strukturierte Vorgehensweise. Sie besteht aus folgenden Komponenten:

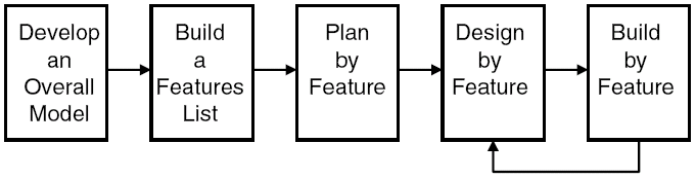
4 Tätigkeiten	RUP 150
3 Rollen	RUP 40
4 Artefakte	RUP 80



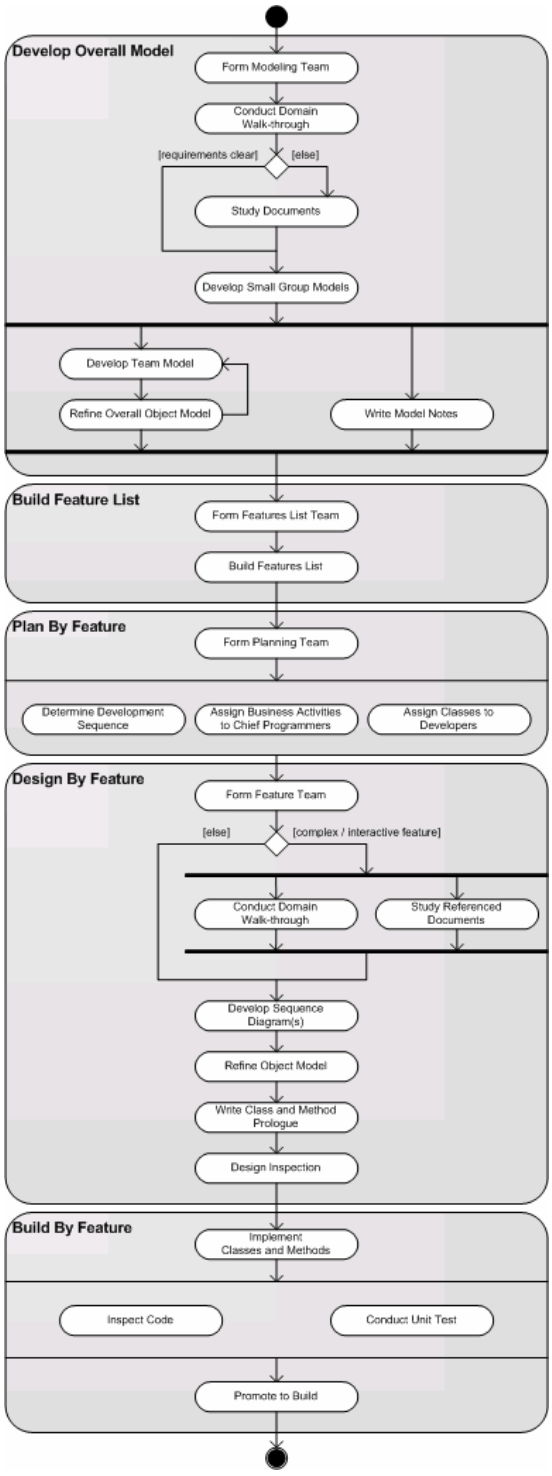
Product Backlog	product features business & technology provided by all stakeholders prioritized by Product Owner
Release Backlog	feature list for next release subset of Product Backlog selected by Product Owner
Product Owner	Person oder Gruppe? priorisiert Product Backlog selektiert Release Backlog Kommunikationskanal für Team nach aussen exzellente Fachkenntnis Entscheidungsbefugnis
Sprint Planning Meeting	Product Owner, Team, ScrumMaster select features for next Sprint
Team	Entwickler managen sich selbst teilt Rollen auf wählt Methoden & Werkzeuge
ScrumMaster	Projektleiter Einhaltung der Scrum-Regeln Hindernisse beseitigen Sprint-Fortschritt kontrollieren
Sprint Backlog	feature list for next sprint
Sprint	Team sprintet zum Ziel: Sprint Backlog Backlog eingefroren: ScrumMaster schützt Team Team organisiert sich selbst
Scrum	daily 15 - max. 30 min. facilitated by ScrumMaster all team members issues & obstacles, but no solutions each participant adresses ... What did you do since the last Scrum? What will you do before the next Scrum? What impediments got in the way of your work?

Sprint Backlog Graph	to monitor progress developer: - time invested in task - percent completion graph: work remaining vs. days of sprint
Executable Increment	Increment of Potentially Shippable Product Functionality
Post-Sprint Meeting	review progress demonstrate features to customers review project

Feature Driven Development (FDD)



FDD wurde als schlanke Methode von Jeff De Luca im Jahre 1997 definiert, um ein großes zeitkritisches Projekt (15 Monate, 50 Entwickler) durchzuführen. Seitdem wurde FDD kontinuierlich weiterentwickelt. FDD stellt den Feature-Begriff in den Mittelpunkt der Entwicklung. Jedes Feature stellt einen Mehrwert für den Kunden dar. Die Entwicklung wird anhand eines Feature-Plans organisiert. Eine wichtige Rolle spielt der Chefarchitekt (engl. Chief Architect), der ständig den Überblick über die Gesamtarchitektur und die fachlichen Kernmodelle behält. Bei größeren Teams werden einzelne Entwicklerteams von Chefprogrammierern (engl. Chief Programmer) geführt.
FDD definiert ein Prozess- und ein Rollenmodell, die gut mit existierenden klassischen Projektstrukturen harmonieren. Daher fällt es vielen Unternehmen leichter, FDD einzuführen als XP oder Scrum. Außerdem ist FDD ganz im Sinne der agilen Methoden sehr kompakt.



XP Extreme Programming

Die Vorgehensweise XP existiert mittlerweile in 2 Versionen. V1. 2000 und V2 2004.

XP Version 1	**XP Version 2**
4 Werte • Communication • Simplicity • Feedback • Courage	5 Werte • Communication • Simplicity • Feedback • Courage • Respect
12 Praktiken	13 Primäre Praktiken
	11 Folge Praktiken (Corollary Practices)

XP besteht aus folgenden Teilbereichen. (Neue Teile der Version 2 sind *kursiv*)

4 / 5 Werte

- Communication
  - Person-to-person mutual understanding of the problem environment.
  - Through minimal formal documentation
  - Maximum face-to-face interaction.
- Simplicity
  - The simplest thing that could possibly work.
  - Make it simple today.
  - Create an environment in which the cost of change tomorrow is low.
- Feedback
  - Optimism is an occupational hazard to programming.
  - Feedback is the treatment.
  - Hourly builds, frequent functionality testing with customers.
  - Embrace change by constant feedback.
- Courage
  - Discipline requires courage.
  - Doing what's right, even when pressured.
  - Alignment of team's values creates conviction.
- *Respect*
  - *XP teams care about each other and the project.*
  - *I am important and so are you.*

12 Praktiken (XP v1)

Planning game • customer stories = required feature • 3 x 5 card • developers estimate • customer prioritize • release plan = stories per release • plan = speculation • plan updated often	Small releases • as small as possible • most valuable business requirements → sense of accomplishment • releasable = not necessarily released
Metaphor	Simple design

• broad view of the project's goal • overall coherent theme • broad sweep of the project ≈ "architecture", but • understandable by customers	• needed functionality • not potential functionality • simplest design needed for functionality = best design → emergent, growing design = no big design upfront
Refactoring • simplest design first then refactor • using design patterns • before adding new functionality refactor existing code	Testing • test-driven development • write test first • unit tests, using e.g. JUnit • re-test continuously → listen (requirements) → test (write first) → code (simplest) → design (refactor)
Pair programming • 2 persons, 1 workstation • continuous code inspection • pairs frequently change (e.g. daily)	Collective ownership • anybody can change any code
Continuous integration • minimum daily builds → test case → code → integrate immediately	40-hour week • Sustainable development • avoid overtime • extended overtime is a productivity-reducing technique
On-site customer • real customer in team • available for questions • sets priorities	Coding standards • strict coding standards needed for o clear, readable code o collective ownership o refactoring

Zusammenfassung v1:

- emergent design, no big up-front design
- collective ownership
- continuous refactoring
- all 12 practices tightly coupled → all 12 practices strictly required
- for small, collocated teams

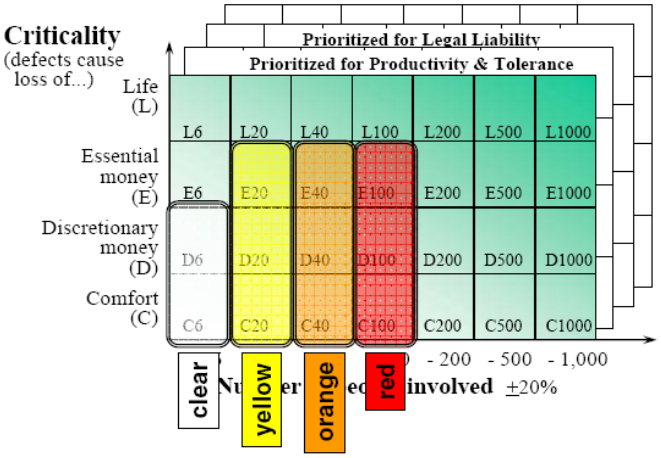
13 Primäre Praktiken 11 sekundäre Praktiken (XP v2)

Primary Practices	**Corollary Practices**
• Sit Together • Whole Team • Informative Workspace • Energized Work • Pair Programming • Stories • Weekly Cycle • Quarterly Cycle • Slack • Ten-Minute Build	• Real Customer Involvement • Incremental Deployment • Team Continuity • Shrinking Teams • Root-Cause Analysis • Shared Code • Code and Tests • Single Code Base • Daily Deployment • Negotiated Scope Contract

• Continuous Intergration • Test-First Programming • Incremental Design	• Pay-Per-Use

Crystal

Crystal ist eine Sammlung von Agilen Methoden die nach Grösse des Projektes gegliedert sind. Sie folgen den 7 Prinzipien von Cockburn. Das erste Ziel ist es funktionierende Software zu erstellen, als zweites Ziel sollten aber Folgearbeiten und nächste Projekte beachtet werden. Das Entwickeln von Software wird fast mit einem Spiel verglichen dass es zu gewinnen gibt.
Als Beispiel werden hier die Varianten Clear und Orange gegenübergestellt:



Crystal clear	Crystal orange
3-6 Leute im selben Raum	Bis zu 40 Personen im selben Gebäude
Roles & Teams	
Roles: sponsor, senior designer, designer/programmer, user (part-time) **Teams:** single team of designers / programmers **Seating:** single big room, or adjacent offices	**Roles:** Sponsor, Business expert, Usage expert, Technical facilitator, Business analyst/designer, Project Manager, Architect, Lead designer / programmer, Designer / programmer, Design Mentor, Reuse Point, Writer, Tester, UI designer. **Teams:** System planning, Project monitoring, Architecture, Technology, Functions, Infrastructure, External test.
Product and milestones	
Products: • Release sequence, schedule of user viewings, deliveries. • Actors-goals list & annotated use cases. • Design sketches & notes as needed, screen drafts. • Common object model • Running code, migration code, test cases • User manual **Publish:** each **Review:** each **Declare:** • Requirements stable enough to design to • UI stable enough to document to • Application correct enough to deliver	**Products:** • Requirements doc • Release sequence, schedule, status report • UI design doc • Common object mode • Inter-team specs • Usage manual • Code • Test cases **Workshop:** • Mid- & post-increment methodology review **Publish:** • Each work product, • Iteration & increment deliveries **Review:** • Each work product, Iteration deliveries, Test cases,

	• Final delivery **Declare:** • Each work product stable enough to review, • Application correct enough to deliver.
Standards	
Policy: • Delivery increments every 2 + 1 months • Tracking by milestones, not by work products • Requirements in annotated usage scenarios (use cases) • Mandatory regression testing of application function • Peer code reviews • Direct user involvement • Ownership model for work products **Local standards** (up to the team): • Coding style • UI style • Regression test framework	**Policy:** • Delivery increments every 3 + 1 months • Tracking by milestones, not by work products • Mandatory regression testing of application function • Direct user involvement • Ownership model for work products • 2 user viewings per release • Use cases completed down to failure cases • Single, common object models (not analysis & design models) • Downstream activities start as soon as upstream is "stable enough to review" **Local standards** (set and maintained by team): • Work product templates, Coding style, UI style • Regression test framework
Tolerance	
Policy standards mandatory, but equivalent substitution permitted (e.g. "Scrum") Full tolerance on techniques: any technique allowed Quite wide tolerance on work products • Many alternatives to intermediate products accepted • e.g., paper, whiteboard, online notes • Low precision in early stages • High precision only required for production work products Assess quality of communications, not quality of work products	**Policy standards** mandatory, but equivalent substitution permitted (e.g. "Scrum") Full tolerance on techniques: any technique allowed • Recommended techniques • Semantic modeling, RDD, facilitated sessions Tolerance on work products • Minor deviation from templates permitted • Few alternatives to intermediate products accepted

Requirements Engineering

Anforderungen

Eine Anforderung beschreibt eine oder mehrere **gewünschte Eigenschaften** oder **gewünschte Verhaltensweisen** eines **Systems**. Wichtig ist das das System als solches klar definiert ist. Anforderungen und Ziele werden oft verwechselt. Folgende Unterscheidung kann helfen:
Anforderung

- beschreibt eine Eigenschaft oder Verhalt eines Systems
- ist auf den Betrachtungsgegenstand gerichtet
- Blickrichtung nach "innen"

Ziel

- beschreibt eine Wirkung oder Folge, die aus der Verwendung des Systems entsteht
- ist auf das Umfeld des Betrachtungsgegenstands gerichtet
- Blickrichtung nach "aussen"

Qualitätsmerkmale einer guten Anforderung sind:

- ist klar
- ist präzis
- ist eindeutig
- ist komplett
- ist geschäftsbezogen
- ist nicht zu technisch
- ist breit abgestützt
- beschreibt was man will, nicht wie es gelöst werden soll!

Zusätzlich zu der eigentlichen Anforderung hat es sich bewährt folgende Informationen auch in einer Anforderung zu übernehmen: **ID** (eindeutige Kennung), **Typ** (funktional, was tut das System/ nicht funktional, wie gut wird die Anforderung erbracht), **Priorität, Verantwortlichkeit, Status, Grund, Thema, Verweise**.

Beziehung zwischen Anforderungen

In sysML sind folgende Beziehungen definiert:

contained	Anforderung "enthält" Anforderung Erlaubt den Aufbau von Anforderungshierarchien. Anforderungen können geordnet und für verschiedene Bereiche der jeweils angemessene Detaillierungsgrad gewählt werden.
satisfy	Modellelement "erfüllt" Anforderung Gibt an, dass ein Modellelement eine Anforderung erfüllt.
trace	Anforderung "kann zurückverfolgt werden auf" ein Modellelement Diese Beziehung gibt die Herkunft einer Anforderung an. Beispielsweise kann eine Anforderung an ein IT-System zurückgehen auf ein Element des Modells des Geschäftssystems.
verify	Testfall "verifiziert" Anforderung Gibt an, dass ein Testfall eine Anforderung verifiziert.

Formulierung von Anforderungen

Bei der Auswahl einer Anforderungssprache helfen die folgenden Punkte:

- Die Sprache muss dem Zielpublikum angepasst sein. (wer soll die Sprache lesen, wer soll in der Sprache schreiben)

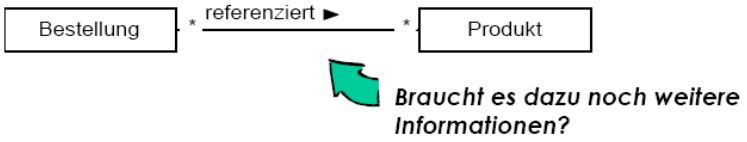
- Die Sprache muss von allen Beteiligten verstanden werden, insbesondere von der Fachseite.
- Es kann auch ein Mix von verschiedenen Sprachen verwendet werden.
- Je mehr Formalität desto besser, solange die Verständlichkeit nicht leidet.
- Grafische Sprachen sind oft übersichtlicher.
- Die Sprache muss der benötigten Präzision Rechnung tragen:

Beispiele für Anforderungssprachen

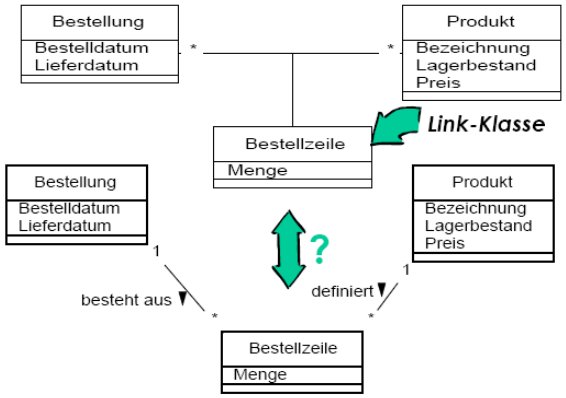
- freier, natürlichsprachiger Text
- natürlichsprachiger Text mit Glossarunterstützung
- formalisierter natürlichsprachiger Text mit Glossarunterstützung
- Unified Modeling Language (UML)
  - Anwendungsfallmodelle
  - Geschäftsobjektmodelle
  - Verhaltensmodelle
- Systems Modeling Language (SysML)
  - Anforderungsdiagramm

Advanced UML

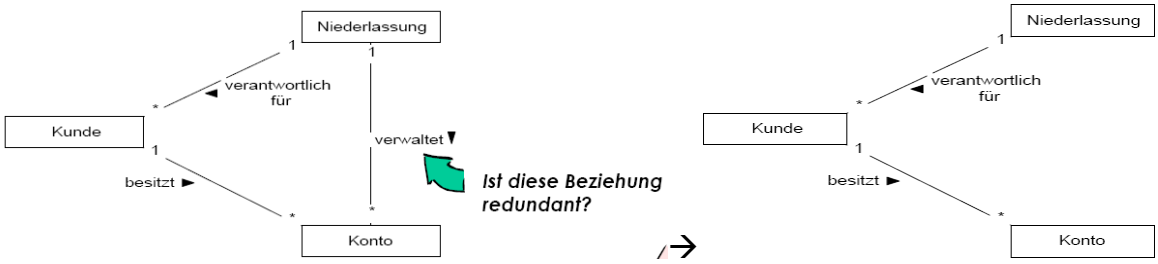
M:M Beziehung



Mögliche Lösung wenn zusätzliche Information in der Referenz abgelegt werden muss.

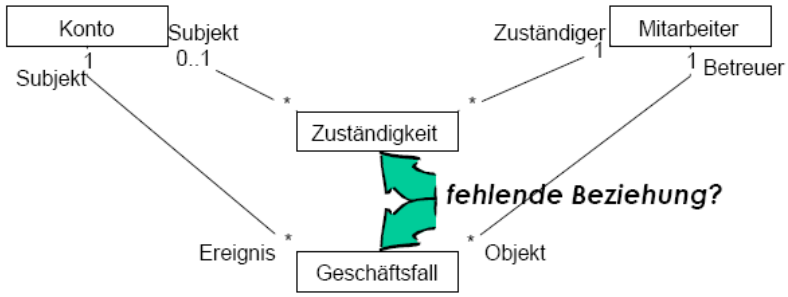


Dreieck-Struktur

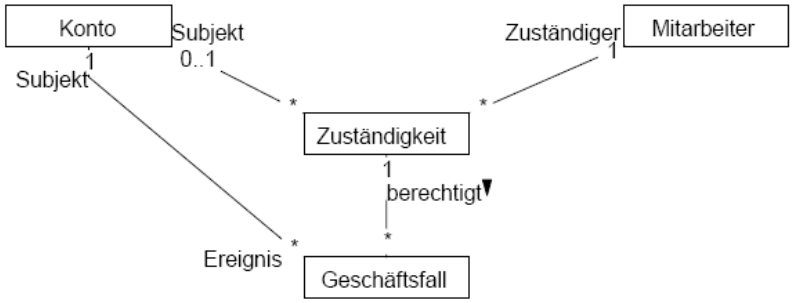


Auflösung redundanter Beziehungen.

Doppel V Struktur

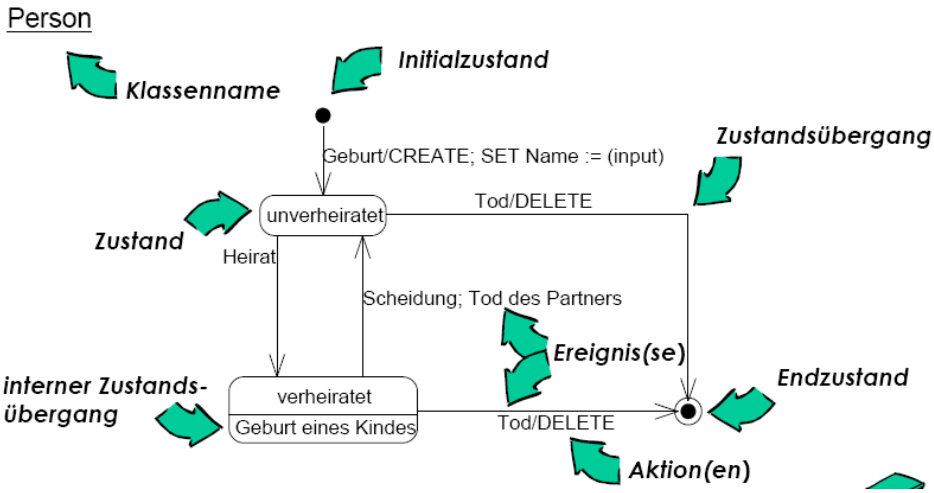


Durch einfügen einer solchen zusätzlichen Beziehung, muss geprüft werden on die dadurch entstehenden Dreiecksbeziehungen nötig sind oder auch aufgelöst werden können.



Verhaltensmodellierung

In der Verhaltensmodellierung werden die für die Klassen relevanten Ereignisse identifiziert. Das Verhalten der einzelnen Objekte, in Reaktion auf diese Ereignisse, wird mit Zustandsdiagrammen spezifiziert:

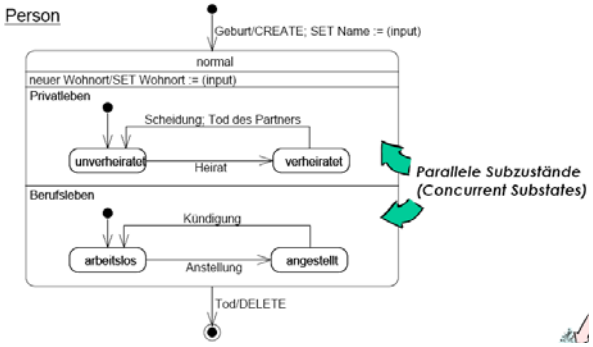


Superzustand

Tritt ein bestimmtes Verhalten in mehreren Zuständen auf so kann es sich lohnen, dieses gemeinsame Verhalten in einem Superzustand (Super State) zusammenzufassen. Das Zustandsdiagramm wird dadurch übersichtlicher.
Innerhalb eines Superzustands wird der Zustand des Objekts immer durch genau einen seiner Subzustände definiert (Ausnahme: parallele Zustände), d.h. ein Superzustand ist ein Pseudozustand der lediglich gemeinsames Verhalten seiner Subzustände spezifiziert.
Jeder Zustandsübergang, der für den Superzustand gilt, gilt automatisch auch für dessen Subzustände.

Parallelzustände

Durch die Modellierung paralleler Subzustände im Verhalten einer Klasse lassen sich überlappende, d.h. voneinander unabhängige Verhaltensmuster separieren.



Synchronisationspunkte

Mit Hilfe von Synchronisationspunkten kann das Leben eines Objekts temporär auf zwei (oder mehr) parallele Threads aufgeteilt und dann wieder zusammengeführt werden.

Übergangsbedingung

Eine Übergangsbedingung (Guard) ist ein boolscher Ausdruck und wird im Ereignisausdruck in eckigen Klammern, nach dem Ereignisnamen angegeben.
Übergangsbedingungen sind explizit formulierte Bedingungen für die Zulässigkeit eines Zustandsübergangs. Jeder Zustandsübergang kann mit einer Bedingung verknüpft werden, d.h. sie wird nur dann ausgeführt, wenn der entsprechende Bedingungsausdruck (guard condition) wahr ist.
So kann dasselbe Ereignis bei gleichem Objekt und identischem Ausgangszustand je nach Bedingung verschiedene Folgen haben.

Finden von Ereignissen

Durch Standardfragen lassen sich Ereignisse finden, die im Leben eines Geschäftsobjektes eine Rolle spielen:

- Welche Ereignisse machen das Geschäftsobjekt für das System relevant bzw. irrelevant?
- Welche Ereignisse definieren oder ändern Attributwerte?
- Welche Ereignisse stellen Beziehungen zu anderen Geschäftsobjekten her oder lösen sie?
- Welche Ereignisse bewirken eine Zustandsänderung des Geschäftsobjektes?

Object Constraint Language OCL

Graphical UML provides no (formal) features to model

- data constraints
- complex constraints
- operation behavior
- derivations
- (complex) computations

OCL fills this gap by providing a side-effect free textual expression language for model elements.

Accessing Attributes

attributes of UML classes:

- `t.attribute`

roles of UML classes:

- `t.role`
- `t.role1.role2.…`

operations of types and UML classes:

- `t.operation(…)`
- `t::operation(…) (class operations)`

operations of collections:

- `coll->operation(…)`

operation parameters:

- `parameter`

Context of an OCL Expression

OCL expressions are always expressed for a specific context:

- (an instance) of a particular class
  e.g. class `customer`
- a particular attribute of (an instance) of a particular class
  e.g. attribute `birth date` of `class customer`
- a particular operation of (an instance) of a particular class
  e.g. operation `setCreditLimit` of class `customer`
- a particular parameter of an operation of (an instance) of a particular class
  e.g. parameter `creditLimit` of operation `setCreditLimit` of class `customer`

The context of an OCL expression is implicitly given if it is attached to a UML model element.

Stereotype of OCL Expressions

OCL expressions are always of a particular stereotype:

- invariant, denoted as **inv**, e.g.
  context customer inv:…
  context customer inv myName:…
- initial value, denoted as **init**, e.g.
  context customer.creditLimit init:…
  context customer.creditLimit init myName:…
- derivation rule, denoted as **derive**, e.g.
  context customer.creditLimit derive:…
  context customer.creditLimit derive myName:…
- query as a body of an operation, denoted **body**, e.g.
  context customer.openAmount(): real body:…
  context customer.openAmount(): real body myName:…
- precondition of an operation, denoted as **pre**, e.g.
  context customer.setCreditLimit(A: real) pre:…
  context customer.setCreditLimit(A: real) pre myName:…
- postcondition of an operation, denoted as post, e.g.
  context customer.setCreditLimit(A: real) **post**:…
  context customer.setCreditLimit(A: real) post myName:…
- combinations are allowed as well, e.g.
  context customer.setCreditLimit(A: real)
  pre:…
  post:…

Expressions in OCL

The following Expressions are defined in OCL:

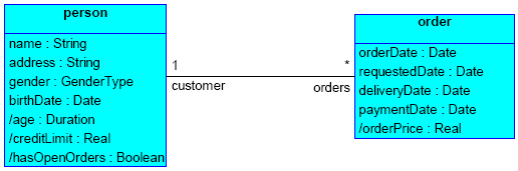
- Boolean Expression evaluates to true or false
- Numeric Expression evaluates to a Boolean, integer or real

- Time Expression evaluates to a new time ore Boolean
- String Expression evaluates to Boolean, integer or string

Examples:

```
context person
inv: 0 <= creditLimit and creditLimit <= 2000
derive: age = Date::now.minus(birthDate)

context order
inv: requestedDate.isAfter(orderDate) and
deliveryDate.isAfter(orderDate)
inv: not customer.address.oclIsUndefined()
```



In class "person":

```
'credit limit' in 0 ... 2000 (constraint)
'/age' := now – 'birth date' (derivation)
```

In class "order":

```
'requested date' > 'order date' and
'delivery date' > 'order date' (constraint)
customer.address is_defined (constraint)
```

Conditional Expressions

OCL supports constructs where one expression decides on the application of other expressions (not to be confused with procedural if/then/else constructs!):

```
if boolExp then exp1 else exp2 endif
```

Collection Expressions

OCL supports a number of operators to form expressions that operate on collections and evaluate to a **boolean:**

- `coll = coll`
- `coll <> coll`
- `coll->isEmpty()`
- `coll->notEmpty()`
- `coll->includes(obj)`
- `coll->includesAll(coll)`
- `coll->excludes(obj)`
- `coll->excludesAll(coll)`

OCL supports a number of operators to form expressions that operate on collections and evaluate to an **integer** or **real**:

- `coll->size()`
- `coll->count(obj)`
- `coll->sum()`

OCL supports a number of operators to form expressions that operate on collections and evaluate to a **collection**:

sub-collections:

- `coll->select(boolExp)`
- `coll->select(v|boolExp)`
- `coll->select(v:T|boolExp)`
- `coll->reject(boolExp)`
- `coll->reject(v|boolExp)`
- `coll->reject(v:T|boolExp)`

new collections:

- `coll->collect(Exp)`

- `coll->collect(v|Exp)`
- `coll->collect(v:T|Exp)`

OCL supports type conversions between the different types of collections as well as composition and decomposition of collections:
Conversion:

- `coll->asSet()`
- `coll->asOrderedSet()`
- `coll->asBag()`
- `coll->asSequence()`

(De)composition:

- `coll->append(coll)`
- `coll->first()`
- `coll->last()`
- `coll->at(n)`

OCL supports a number of operators to form expressions that provide typical set operations and evaluate to a **collection**:

- `coll->union(coll)`
- `coll->intersection(coll)`
- `coll->minus(coll)`
- `coll->`
- `symmetricDifference(coll)`

Example:

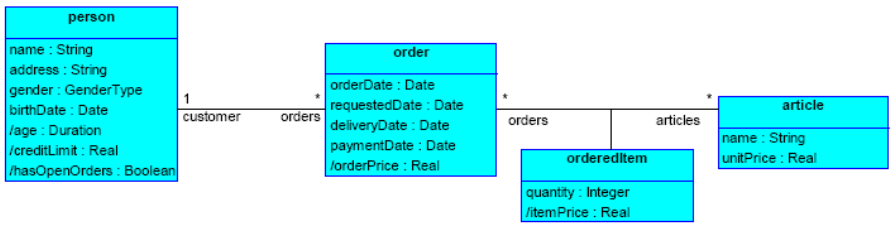
```
context person
inv: orders-> select(oclIsUndefined(paymentDate))-> size() <= 1
inv: creditLimit >= orders->
select(oclIsUndefined(paymentDate)).orderPrice-> sum()
```

Quantifier Expressions

OCL supports universal and existential quantification known from predicate logic:

- `coll->forall(boolExp)`
- `coll->forall(v|boolExp)`
- `coll->forall(v:T|boolExp)`
- `coll->exists(boolExp)`
- `coll->exists(v|boolExp)`
- `coll->exists(v:T|boolExp)`

Example:



context person

```
derive: hasOpenOrders = orders->exists(oclIsUndefined(paymentDate))
```

context order

```
inv: articles->forall(a1,a2|A1 <> A2 implies A1.unitPrice <>
A2.unitPrice)
```

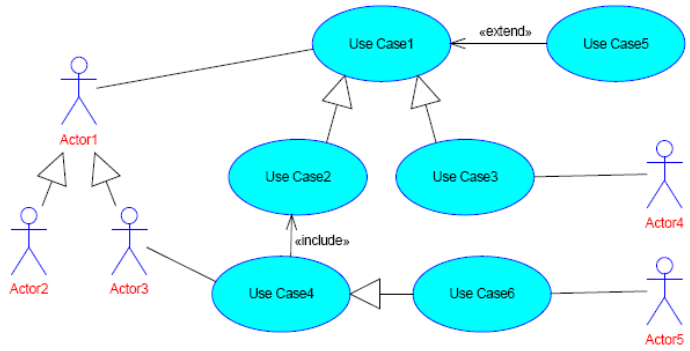
xUML

Black-Box Modeling

xUML black-box models are represented by

- actors representing user roles
- use cases representing accessible, goal-oriented functionalities
- actor/use case and use case/use case associations
- black-box sequence diagrams specifying interactions within a use case
- requests and responses with parameters as atomic interactions between actor and system

Actor Rules



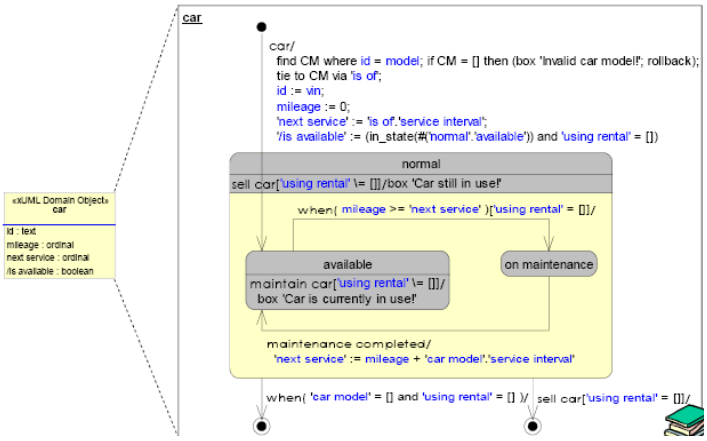
- any use case directly connected to an actor is available to that actor
- any use case that is included by another use case that is available to an actor is also available to that actor
- any use case that is included by another use case that is a generalization of a use case that is available to an actor is also available to that actor
- any use case that is an extension of another use case that is available to an actor is also available to that actor
- any use case that is an extension of another use case that is a generalization of a use case that is available to an actor is also available to that actor
- any use case that is a specialization of another use case that is available to an actor is also available to that actor
- any use case that is available to a more general actor is also available to the more specific actor

xUML Model

Glass-Box Modeling

xUML glass-box models are represented by

- classes that
  - are instantiable and destroyable
  - have attributes representing value-based properties
  - may have derived attributes that specify (complex) derivations
  - may be specializations of other classes
  - react on requests



- associations that
    - represent relation-based properties of classes
    - are instantiable and destroyable
    - are constrained by multiplicities
    - may have association classes

xUML glass-box models are furthermore represented by

- state diagrams that
- constrain valid sequences of requests for a class
- proactively react on conditions and time events
- specify the reactions of a class on those requests, conditions and time events
- may specify complex integrity constraints

Actions in xUML

Object Actions

Object actions are used to create or destroy instances of classes:

- `create ObjectVariable from ClassName`
- `create ObjectVariable from ClassName by Event`
- `delete ObjectsExpression`

Instances are initialized by their birth events. It is highly recommended to name birth events after the class names (see following slide). "ObjectVariable" is a temporary variable that receives the instance created. Instances may also be created implicitly via association class mechanism and deleted implicitly via death events.

Assignment Actions

Assignment actions are used to assign values to attributes and temporary variables:

- `Variable := Expression`
- `Attribute := Expression`
- `ObjectsExpression . Attribute := Expression`
- `/Attribute := Expression`
- `return Expression`

Using "ObjectsExpression", multiple assignments to the same attribute of a set of instances may be done in one go.
Assignments to a derived attribute will not evaluate the expression before the assignment.
Return assigns the Expression to the value of a call expression.

Association Actions

Association actions establish or destroy links (i.e. instances of associations) between instances of classes:

- **gain** and **lose** actions establish or destroy links from the master's perspective to its details, e.g.
    - `gain ObjectsExpression via Role`
    - `lose ObjectsExpression via Role`
- **tie**, **cut**, and **swap** actions establish, destroy or replace links from the detail's perspective to its master, e.g.
    - `tie to ObjectExpression via Role`
    - `cut from ObjectExpression via Role`
    - `swap to ObjectExpression via Role`
- **link** and **unlink** actions establish or destroy links between instances from the perspective of a third instance, e.g.
    - `link ObjectExpression via Role with ObjectExpression via Role`
    - `unlink ObjectExpression via Role from ObjectExpression via Role`

Transaction Actions

Transaction actions allow grouping multiple actions in multiple instances into a single atomic unit of work:

- `begin, begin(TXName)`
- `commit, commit(TXName)`
- `abort, abort(TXName)`
- `rollback, rollback(TXName)`
- `reject`

Transactions may be started and terminated automatically or explicitly. If explicitly controlled, they even may be nested.
abort and rollback may be automatically issued on unexpected or unknown events in state diagrams.

Control Structures in xUML

Sequences

Sequences of actions are formed by the action separator ";". The action separator may also be used as an action terminator. Actions may be grouped by parentheses so that they may be considered as a single action.

Options

Single actions may be performed optionally, e.g.

- `if BooleanExpression then Action`
- `if BooleanExpression then Action else Action`

Note that the actions in an if-then-else structure must be simple actions and not control structures.

Selections

Multiple actions may be forked by means of the switch action, e.g.

```
switch Expression
if Expression then Action
if Expression then Action
…
else Action
```

Iterations

And finally, actions may be repeated by means of the foreach action,
e.g.

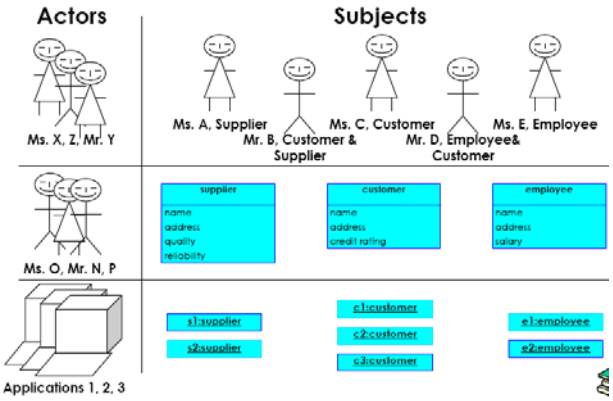
- `foreach Variable in ListSetExpression do Action`
- `foreach Variable in ListSetExpression where BooleanExpression do Action`
- `foreach ObjectsExpression do Action`
- `foreach ObjectsExpression where BooleanExpression do Action`

Note that the last two forms involve a context movement to each individual instance processed by the iteration.

Aspect Oriented Analysis

Problem

Real world concepts (Person) are modeled from different perspectives (suppliers, customers, employees) in order to support different objectives (purchasing, sales, and HR applications):

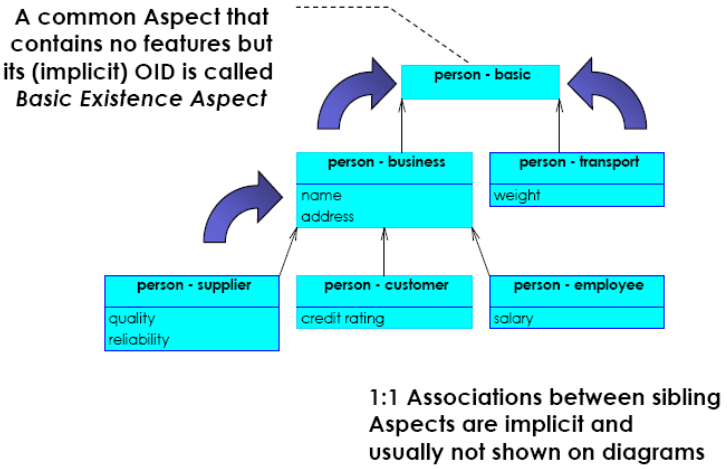


- There are more instances in the applications (7) than in the real world (5)
- Individual concepts (i.e. their features) and instances (i.e. their values) overlap
- Redundant implementation of conceptual features in individual applications
- Redundant instance information is maintained in individual applications
- Individual instances may easily become out of sync
  (i.e. what happens, if person B moves to a new address or if person D dies?)

What are Aspects?

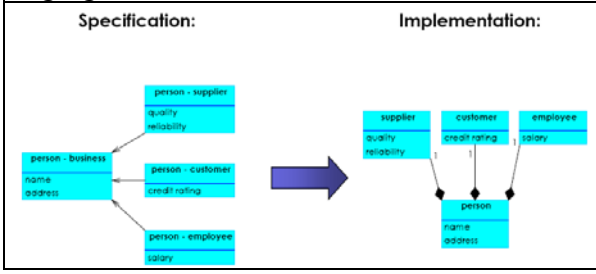
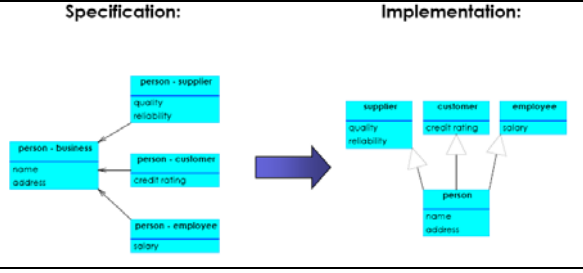
An Aspect is an abstraction of a real world object based on a viewpoint-specific view and includes:

- the state vector (attributes) relevant to that viewpoint
- the associations relevant to that viewpoint
- the type interface (operation signature) relevant to that viewpoint
- the behavior (operation semantics) relevant to that viewpoint



Implementing Aspects

There are different ways to implement an aspect.

Agregation	Multiple Inheritance
	

Summary

What is an Object?

- An Object is a thing or concept in the real world that is/will be relevant for a IT system.
- An Aspect is a viewpoint-specific model of a real world Object including features such as state, associations, signature and behavior.

Or in other words: An Aspect is the abstraction of an Object.

- A Class is a technical (programming language) construct to implement one or many Aspects.
- An Instance is a physical model (transient or persistent) that represents a (fragment of a) real world Object inside an IT system.

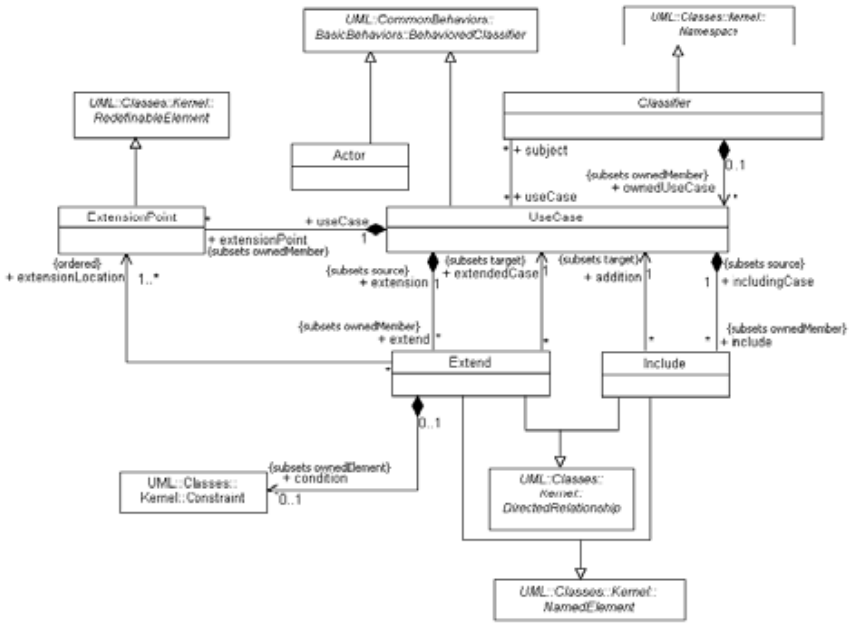
What are Aspects used for?

Aspects are used to ...

- separate different views about a real world object (separation of concerns)
- model multiple concurrent lives of a real world object
- partition a business area into distinct applications that support individual viewpoints
- partition a large project into smaller projects based on individual viewpoints
- identify commonalties and differences between new and existing models in applications that must be interfaced (reused) by a new application
- extend an existing implementation by adding a new Aspect to it
- properly implement advanced features such as aggregation, multiple, parallel inheritance, concurrent behavior and distribution.

Metamodeling

- a metamodel is a model that describes a model
- the UML metamodel describes the Unified Modeling language, i.e. the UML language specification uses an UML class model (the UML metamodel) to describe the UML language
- an UML case tool implements the UML metamodel (or parts of it) in its data base
- UML users can safely ignore the UML metamodel (In fact don't mention it at all, it will only confuse them!)
- A lot of constraints on the metamodel are defined in OCL.



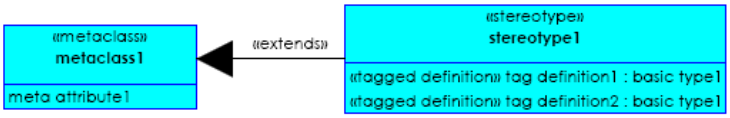
Example of the Metamodel of UML

UML Profiles

- Starting with version 1.1 UML included stereotypes and tagged values to make simple extensions.
- In later revisions the notion of a Profile was defined in order to provide more structure and precision to the definition of stereotypes and tagged values.
- A profile defines limited extensions to a reference metamodel with the purpose of adapting the metamodel to a specific platform or domain.
- A Profile is a kind of Package that extends a reference metamodel. The primary extension construct is the Stereotype.

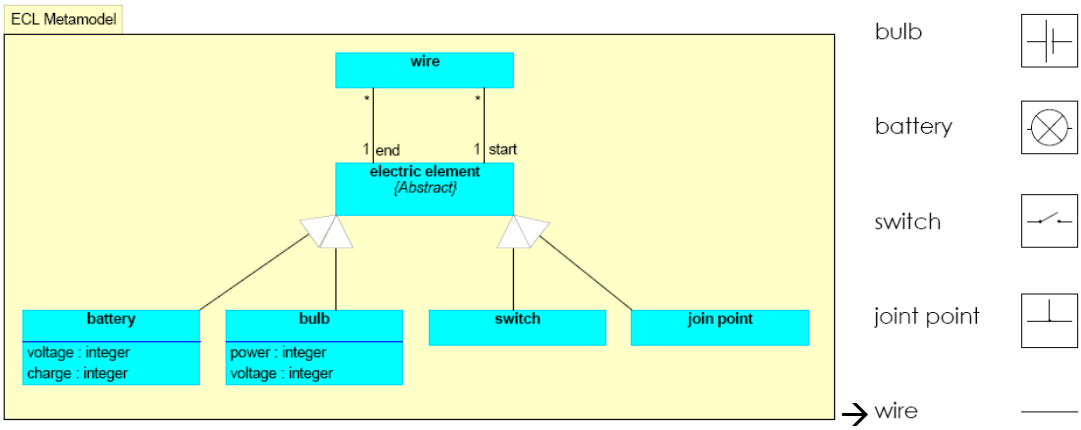
UML Stereotype

- A stereotype defines how an existing metaclass may be extended and enables the use of platform or domain specific terminology or notation.
- Stereotype is a kind of Class that extends Classes through Extensions.
- Just like a class, a stereotype may have properties, which may be referred to as tag definitions.
- When a stereotype is applied to a model element, the values of the properties may be referred to as tagged values.



Domain Specific Languages

Often it makes sense to define a new language from scratch. The syntax of such a new language can be defined in a new metamodel.
Sample, a Language for electric circuits:



Model Driven Architecture

Why UML is not enough

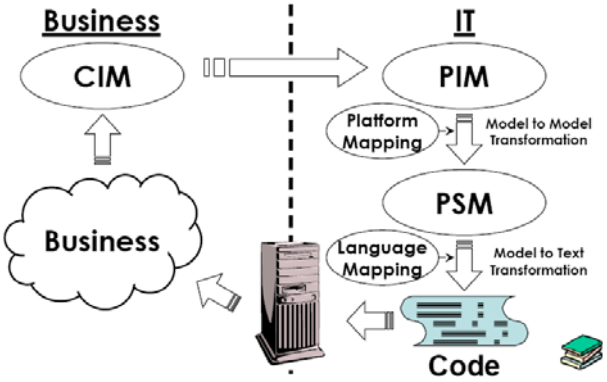
UML has its roots in software development and is designed to describe software artifacts. However,

- the UML models are not precise enough to transform them into code
- not all software artifacts may be represented by UML (e.g. GUIs/style, computations, decisions, databases, etc.)
- the UML implies the object oriented paradigm
- the world is not just software…
- there are also technical systems…
- and business systems

Basics of MDA

Definition of MDA:
„The MDA defines an approach to IT system specification that separates the specification of system functionality from the specification of the implementation of that functionality on a specific technology platform. To this end, the MDA defines an architecture for models that provides a set of guidelines for structuring specifications expressed as models."

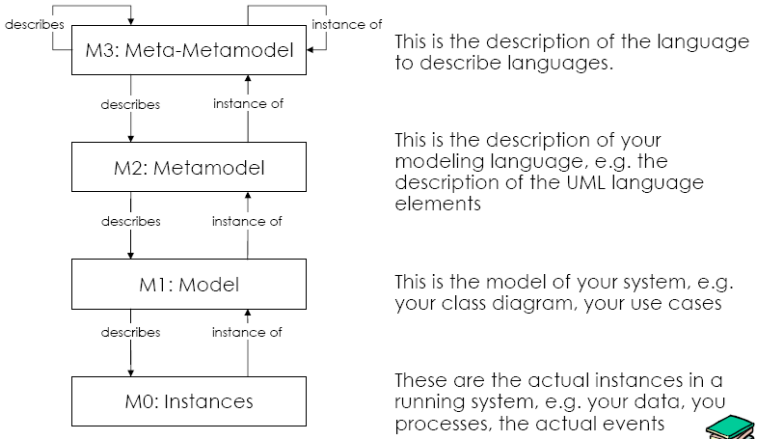
Different Variations of MDA:



- **CIM**: Computation Independent Model
  - represents a model that purely describes the business
  - serves as the basis to decide about IT support of the business
- **PIM**: Platform Independent Model
  - represents a model that defines an IT system's functionality in implementation-independent form
  - expressed in UML
- **PM**: Platform Model
  - represents a metamodel of a particular platform (J2EE, .NET, RDB, etc.)
  - typically expressed as a UML profile
- **PSM**: Platform Specific Model

    - represents a model that specifies the mapping of a PIM onto a particular platform (J2EE, .NET, RDB, etc.)
    - expressed in UML and applies a UML platform profile
- **PSI**: Platform Specific Implementation (Code)
    - represents the final implementation artifacts (source code, configuration files, etc.) of an implemented IT system
    - expressed in a textual (programming) language

MDA Roles

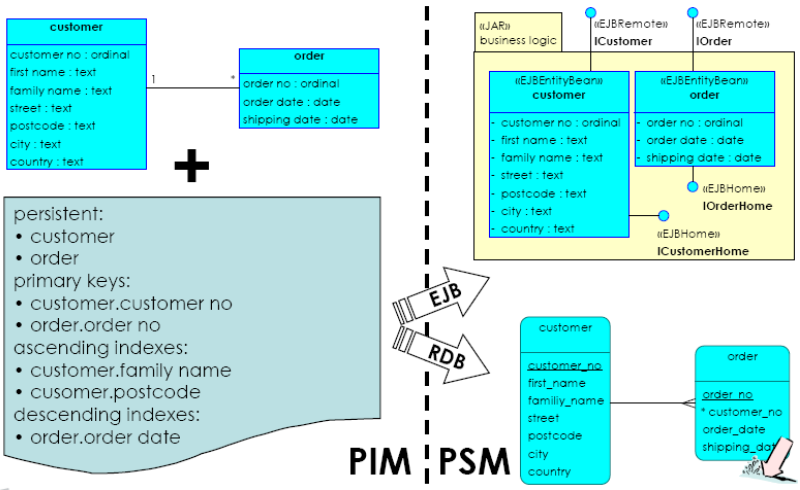


- Business/IT Analyst
    - Builds the CIM
    - Builds the PIM
- Architect
    - Defines the architecture
    - Defines coding style
- Designer
    - Understands the PIM and the architecture
    - Tags the PIM
- Transformation Builder
    - Implements M2M transformations
    - Implements M2T transformations

Transformations

MDA transformations convert artifacts into other artifacts:

- **Model to Text** (M2T) Transformations
    - create texts from models expressed in MOF-based languages
    - updates texts from models expressed in MOF-based languages
    - used in forward engineering
- **Text to Model** (T2M) Transformations
    - create models expressed in MOF-based languages from texts
    - updates models expressed in MOF-based languages from texts
    - used in reverse engineering
- **Model to Model** (M2M) Transformations
    - create models expressed in MOF-based languages from other models expressed in MOF-based languages
    - updates models expressed in MOF-based languages from other models expressed in MOF-based languages
    - used in forward and reverse engineering



Example of a Transformation:

Code generation

- Name mangling
  Model names must be transformed into names that conform with the target language syntax
- Type mapping
  Generic UML types must be mapped to types of target language
- X% of 100% versus 100% of X%
  Generating everything of a little bit is better than generating a little bit of everything
- Protected regions
  Incompletely generated components require manual completion, which must be protected at regeneration
- Reverse/round trip engineering
  Incompletely generated components require manual completion, which may compromise compatibility with model and thus difficult resynchronization.

Geschäftsprozessmodellierung

Ein Geschäftsprozess ist eine Folge von Schritten um ein Geschäftsresultat zu erzielen.
Ein Geschäftsprozess kann Teil eines anderen Geschäftsprozesses sein oder andere Geschäftsprozesse enthalten bzw. diese anstossen.
Geschäftsprozesse gehen oft über Abteilungen und Betriebsgrenzen hinweg und gehören zur Ablauforganisation eines Betriebs.

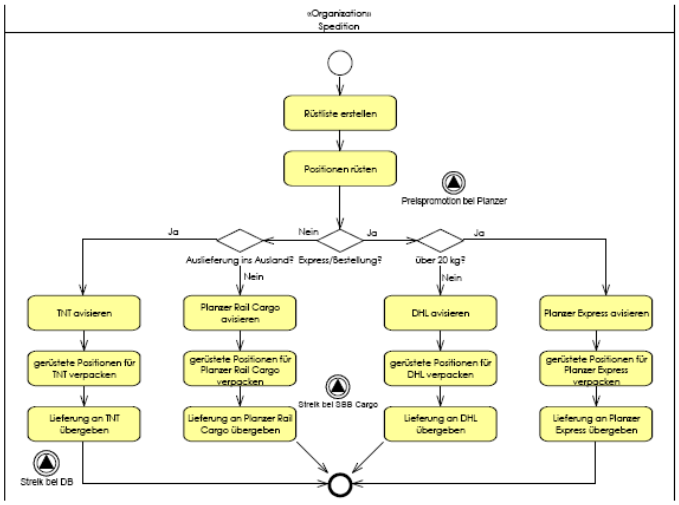
Begriffe

- **Prozessor**
  Ein Prozess wird durch einen Prozessor ausgeführt. Der Prozessor kann ein Mensch oder eine Maschine sein.
- **Geschäftsaktivität**
  Ein einzelner Schritt eines Geschäftsprozesses, die typischerweise durch einen Prozessor in entsprechender Rolle ausgeführt wird.
- **Workflow**
  Die vollständige oder teilweise Automatisierung eines Geschäftsprozesses, bei der Dokumente, Informationen oder Aufgaben von einem Teilnehmer gemäss einer Menge von Regeln zu einem anderen zur Weiterbearbeitung weitergeleitet werden.

- **Modell**
  Eine auf bestimmte Zwecke ausgerichtete vereinfachende Beschreibung der Wirklichkeit
    - Deskriptive Modelle
      werden zur Analyse eines existierenden Systems verwendet
    - Präskriptive Modelle
      werden zur Gestaltung eines neuen Systems verwendet
- **Geschäftsprozessmodell**
  Eine zweckorientierte, vereinfachte Abbildung eines oder mehrerer Geschäftsprozesse.

Detaillierte Prozessmodelle

Klassische detaillierte Prozessmodelle beschreiben die Ausführung von Prozessen:



- Sie optimieren Effizienz und Produktivität.
- Sie enthalten viele Regeln und stellen ihre Einhaltung sicher.
- Sie sind ein Instrument des Qualitätsmanagements.
- Sie eignen sich für **Routinearbeiten**, die häufig wiederholt werden.
- Sie können mit Hilfe von Workflow Management Systemen automatisiert werden.
- Stossen schnell auf Grenzen bei wissenintensiven Prozessen oder solchen die sich schnell verändern, unübersichtlich bei vielen Varianten.

Leichte Prozessmodelle

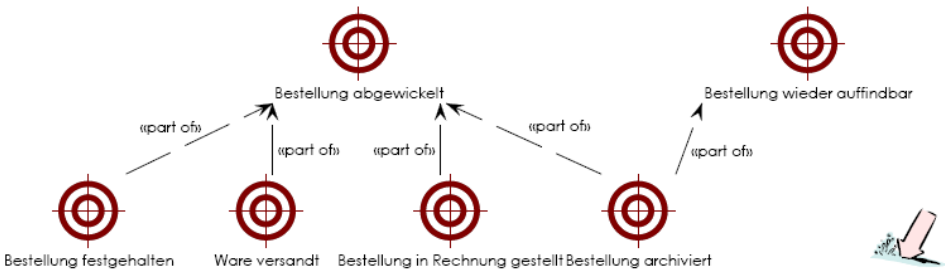
Um **wissensintensive** Geschäftsprozesse zu modellieren, bevorzugen wir **leichte Prozessmodelle**.
leicht bedeutet:

- nur so wenige Details wie nötig (aber nicht weniger)
- nur die wichtigsten Kontroll- und Datenflüsse
- möglichst wenig Regeln in Prozessmodellen zu "betonieren"

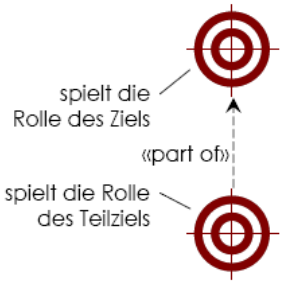
Dadurch können die Modelle auch in der Praxis **gepflegt** werden.

Leichte Prozessmodelle lassen sich zielorientiert aufbauen.

- Eine Geschäftsaktivität
    - darf nicht zum Selbstzweck ausgeführt werden
    - muss ein Geschäftsziel verfolgen
- Die Geschäftsziele können ebenfalls modelliert werden. Hier ein Beispiel zur Bestellungsabwicklung:



Geschäftsziel



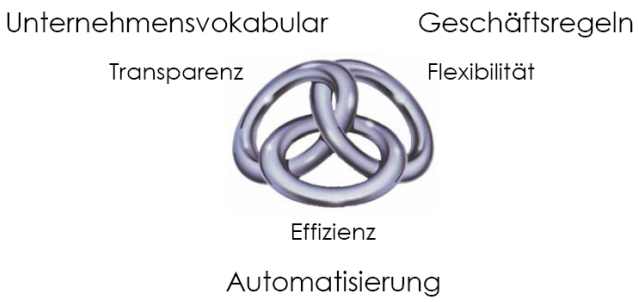
- Geschäftsziel / Ziel
  Ein gewünschter Zustand, der durch eine Geschäftsaktivität herbeigeführt werden kann.
- Das Ziel ist die Postcondition der Geschäftsaktivität.
- Ein Ziel kann in weitere Zeile ("Teilziele") zerlegt werden.
- Sind alle Teilzeile eines Ziels erreicht ist damit auch das Ziel selbst erreicht. (Die Teilziele sind notwendig und hinreichend für das Ziel)
- Die Beziehung zwischen einem Teilziel und seinem Ziel heisst «part of».
- An der Erreichung von Teilzielen kann parallel gearbeitet werden.
- Ein Ziel, dass nicht weiter in Teilziele zerlegt ist wird auch "Blattziel" genannt

Wie Geschäftsprozesse modellieren?

- Je nach Zweck der Geschäftsprozessmodellierung:
- bestehende Prozesse dokumentieren:
  nicht zu detaillierte Prozessmodelle, aber ggf. mit physischen Elementen (z.B. Ablagen)
- bestehende Prozesse überwachen:
  nur messbare Aktivitäten identifizieren und instrumentieren
- Verantwortlichkeiten für Prozesse und Aktivitäten festlegen:
  Prozess-Skizzen und Job-Beschreibungen
- bestehende Prozesse verbessern/vereinheitlichen:
  Varianten von groben Prozessmodellen ggf. mit "Tiefenbohrungen"; ev. Gegenüberstellung mit neu gestalteten Prozessen
- bestehende Prozesse zur Verbesserung simulieren:
  detaillierte Prozessmodelle mit Erfahrungs-Kennzahlen
- grundsätzlich neue Prozesse identifizieren und gestalten:
  innovative Ansätze wie SSM oder BMM
- das Zusammenspiel verschiedener Prozesse gestalten:
  Prozesslandkarte und Orchestrierung
- bestehende Prozesse als Grundlage für die Gestaltung der ITUnterstützung verwenden:
  Identifikation zu unterstützender Aktivitäten und der verantwortlichen Ausführenden
- bestehende Prozesse so weit wie möglich automatisieren:
  detaillierte Prozessmodelle zur Orchestrierung von IT-Services

Business Rules Ansatz

- Das Business, nicht Technologie muss die treibende Kraft für die IT-Entwicklung sein.
- Das Geschäftswissen ist eine extrem wertvolle Ressource eines Unternehmens.
- Das Geschäftswissen sollte innerhalb eines Unternehmens (weitgehend) konsistent angewandt werden.



- Geschäftsaktivitäten sollten so weit wie möglich automatisiert sein, d.h. von IT-Systemen ausgeführt werden.

Begriffe

- Geschäftsregel / Business Rule:
  Geschäftsregeln sind fachliche Aussagen, die immer wahr sind oder immer wahr sein sollten.
  - Aus Geschäftssicht: Eine Direktive, die das Geschäftsverhalten beeinflussen oder leiten soll, um damit eine Geschäftspolitik zu unterstützen, die als Reaktion auf eine Chance, Bedrohung, Stärke oder Schwäche formuliert wurde.
  - Aus IT Sicht: Eine Aussage, die Aspekte des Geschäfts definiert oder einschränkt. Damit wird beabsichtigt, die Struktur des Geschäftes festzulegen oder das Verhalten des Geschäfts zu kontrollieren oder zu beeinflussen.
- Business Rule Ansatz
  Eine Menge von Techniken und Technologien, die sowohl Business Engineering als auch IT Systementwicklung abdecken, mit dem Ziel, agile Geschäfts- und IT-Systeme zu betreiben, welche direkt durch Geschäftsleute kontrolliert werden.
- Business Rule Engine
  Eine technische Software-Komponente, die für die effiziente Ausführung und Überwachung von Geschäftsregeln verantwortlich ist.

Geschäftswissen

Im Business Rules Ansatz wird Geschäftswissen aus folgenden Bestandteilen formuliert:

- **Konzepte**
  wichtige fachliche Elemente (z.B. "Lieferant", "Bestellung", "Produkt", etc.) sowie ihre (präferierte) Bezeichnung und Definition
- **Fakttypen**
  Mögliche Aussagen über Konzepte, z.B. "Lieferant liefert Produkt", "Bestellung hat Rabatt Prozent), etc.
- **Regeln**
  Verknüpfung von Fakttypen zu Richtlinien und Vorschriften, z.B. "Bestellung hat Rabatt 10%, falls …"

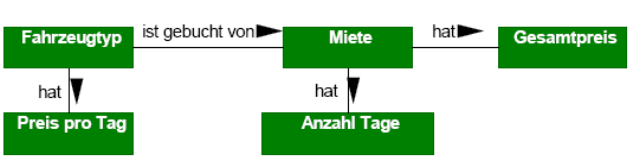
Business Rules Mantra:
Regeln bauen auf **Fakttypen** auf, **Fakttypen** bauen auf **Konzepten** auf.

Darstellung von Geschäftsregeln

Geschäftsregeln sollten idealerweise so formuliert sein, dass sie

- für den (Business-)Menschen verständlich sind
- in möglichst natürlicher Sprache
- in Deutsch, Französisch, Italienisch, Englisch, …
- deklarativ und nicht prozedural sind
- das Ziel, nicht den Weg formulieren ("what not how")
- keinen Anwendungszeitpunkt und keine Reihenfolge spezifizieren

Formale Sprache

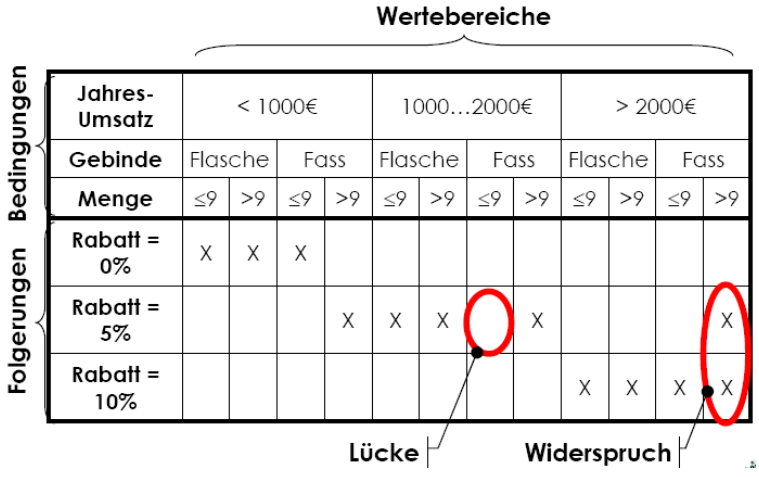


Es ist notwendig, dass der Gesamtpreis von jeder Miete ist die Anzahl Tage * Preis pro Tag von dem Fahrzeugtyp der *ist gebucht von* dieser Miete.

Die Formulierung muss sich an der Zielgruppe, den Lesern und Autoren, orientieren
Mit folgenden Mitteln kann eine Prosa-Regel formalisiert werden:

- Verwendung der Substantivkonzepte aus den Vokabular im Regeltext
- Verwendung der Fakttypen aus dem Vokabular im Regeltext
- Verwendung von Gliederungshilfen (Umbrüche, Listen, usw.) zur Darstellung des Regeltextes
- Verwendung einer definierten Regelsprache im Regeltext
- Verwendung von Textschablonen zur Darstellung des Regeltextes

Entscheidungstabelle



Entscheidungstabellen werden verwendet, um eine Menge zusammenhängender Geschäftsregeln darzustellen
Entscheidungstabellen bieten folgende Vorteile:

- Sie sind sehr kompakt
- Lücken, Widersprüche und Redundanzen werden offensichtlich
- Sie können für Ableitungs- und Prozessregeln verwendet werden

Implementierung von Geschäftsregen

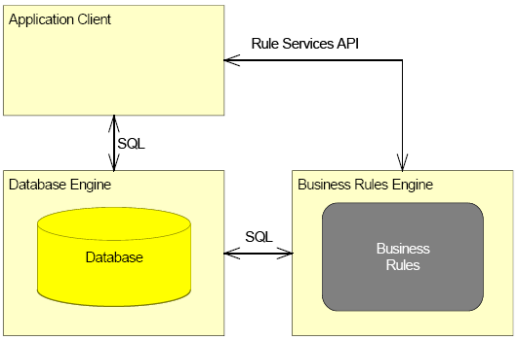
Will man Geschäftsregeln automatisieren kann dies mit unterschiedlichen Technologien erfolgen, beispielsweise

- durch manuell programmierten Code
- durch die Funktionalität von Datenbanken (Views, Triggers, …)
- durch eine Rule Engine

Die Service-Architektur

+ Einfach in eine existierende Umgebung integrierbar
+ Gute Unterstützung für (komplexe) Ableitungsregeln
+ Gute Transparenz; oft sogar Erklärungskomponente verfügbar
+ Viele kommerzielle Produkte verfügbar

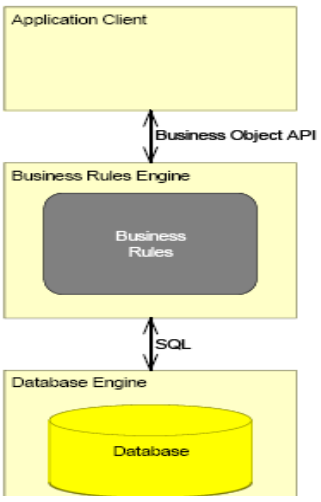
− Weniger gut für Einschränkungen geeignet
− Geschäftsregeln können umgangen werden, d.h. BRE muss nicht aufgerufen werden
− Performance kann problematisch sein



Layer Architektur

+ Gute Unterstützung für Einschränkungen und Ableitungsregeln
+ Kann nur schwer umgangen werden, d.h. Daten sind durch Einschränkungen geschützt
+ Ableitungsregeln können transparent integriert werden (keine Unterscheidung zwischen deklarierten und abgeleiteten Fakten)

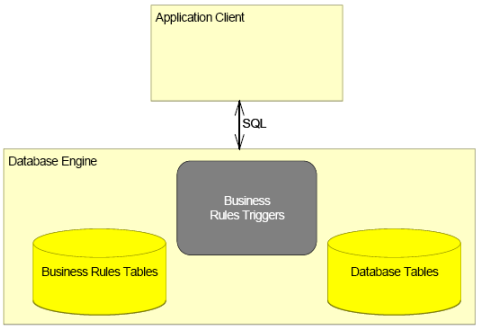
− Prozessregeln schwierig abbildbar
− Typischerweise proprietärer Zugriffs-Mechanismus auf Daten
− Performance kann problematisch sein
− Nur wenige kommerzielle Produkte verfügbar



Datenbank Architektur

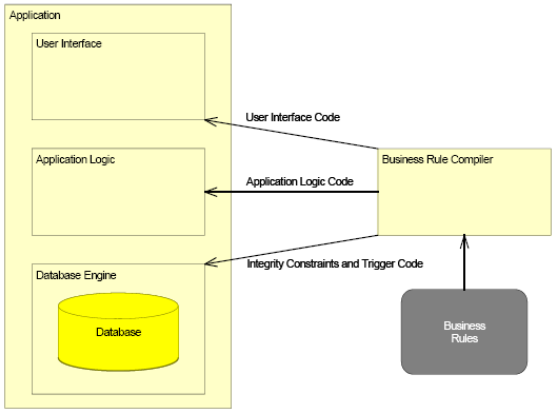
+ Geschäftsregeln können nicht umgangen werden
+ Gute Unterstützung von Einschränkungen
+ Gute Performance, da Regeln nahe an Daten sind

− Beschränkt geeignet für Ableitungsregeln
− Prozessregeln schwierig abbildbar
− Sehr technische Regelsprache
− Aktive Regeln nicht transparent
− Nur für relationale Datenbanken



Kompilations-Architektur

+ Sehr gute Performance
+ Redundante Anwendung von Regeln (z.B. Server und Client)
+ Unterstützt alle Arten von Regeln
− Geschäftsregeln können umgangen werden
− Aktive Regeln kaum mehr sichtbar (Transparenz)



− Unter Umständen schwer in eine bestehende Umgebung integrierbar
− Nur wenige kommerzielle Produkte verfügbar

Daten-Schnittstelle

Im Rahmen einer Service-Architektur können deklarierte Fakttypen auf zwei Arten einer Rule Engine zur Verfügung gestellt werden:

- Via Push-Schnittstelle
  Bei einem Service-Aufruf werden der Rule Engine sämtliche für die Entscheidung relevanten Informationen mitgegeben
- Via Pull-Schnittstelle
  Bei einem Service-Aufruf holt sich die Rule Engine sämtliche für die Entscheidung relevanten Informationen selber

Push-Schnittstelle

+ keine allgemeine Schnittstelle der Rule Engine auf die Geschäftsdaten notwendig
+ oft kein zusätzlicher DB-Zugriff notwendig, da die notwendigen Daten im Aufrufer bereits vorhanden sind

− nicht geeignet für grosse Datenmengen
− bei Regel-Änderungen muss ggf. die Aufruf-Schnittstelle angepasst werden

Pull-Schnittstelle

+ die Aufruf-Schnittstelle bleibt auch bei Regel-Änderungen stabil
+ auch für grosse oder unbekannte Datenmengen geeignet
− generischer Zugriff der Rule Engine auf Geschäftsdaten erforderlich
− ev. resultieren redundante DB-Zugriffe von Aufrufer und Rule Engine