SW-Entwicklungsmodelle

Code and Fix

Vorgehen, bei dem Codierung oder Korrektur im Wechsel mit Ad-hoc-Tests die einzigen bewussten ausgeführten Tätigkeiten der Software-Entwicklung sind.

Vorteile	Nachteile
Liefert schnell Ergebnisse	Projekt schlecht planbar
Einfache Tätigkeiten am Anfang (Code, Test, Fix)	Keine Unterstützung für Entwicklung im Team
	Aufwand für Korrekturen unangemessen hoch
	Schlecht wartbare Software

Wasserfallmodell

Die Software-Entwicklung wird als Folge von Aktivitäten/Phasen betrachtet, die durch Teilergebnisse (Dokumente) gekoppelt sind. Die Reihenfolge der Aktivitäten ist fest definiert.

Vorteile	Nachteile
Hohe Planbarkeit (Funktionalität, Zeit und Kosten)	Risiko sehr lange hoch, da Lösungskonzept nur auf dem Papier validiert
Klare Aufteilung der SWE in einzelne Phasen (Analyse, Design, Test…)	Anforderungen sind zu Beginn nie alle bekannt

Iterativ-inkrementelle Modelle

Software wird in mehreren geplanten und kontrolliert durchgeführten Iterationen schrittweise (inkrementell) entwickelt.

Vorteile	Nachteile
Flexibles Modell bei unklaren Anforderungen / Zielen	Detaillierte «upfront» Planbarkeit hat Grenzen
Gutes Risikomanagement	Braucht eine Involvierung und Steuerung durch den Kunden über die ganze Projektdauer.
Frühe Einsetzbarkeit der Software und Feedback	

Agile Softwareentwicklung

Ist eine Sammlung von Ideen (Werte, Prinzipien und Praktiken), um den iterativ-inkrementellen Softwareentwicklungsprozess flexibler und schlanker zu machen. Agile Softwareentwicklung ist kein eigenes Prozessmodell

Prozesskontrolle

Definierte Prozesskontrolle

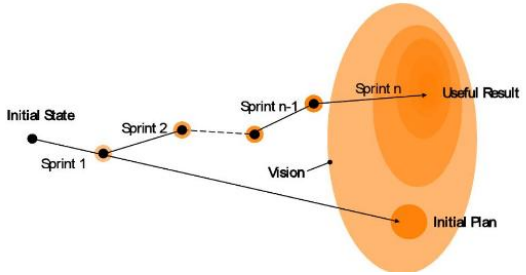
Planung wird am Anfang durchgeführt, dann Prozess gesteuert und überwacht

- Geeignet für gut planbare Problemstellungen (Anforderungen stabil und von Beginn weg bekannt)

Empirische Prozesskontrolle (Agil)

Nur Grobplanung am Anfang

- Prozess wird fortlaufend überwacht
- Rollende Planung
- Geeignet für komplexe Problemstellungen (unbekannte Anforderungen und/oder stetig ändernd)



Verbreitete Prozessmodelle

Hermes: Wasserfall/V-Modell (historisch), definiert, kaum empirisch
Scrum: iterativ-inkrementell, empirisch
Unified Process (UP): iterativ-inkrementell, definiert oder empirisch

Usability und User Experience (UX)

Usability

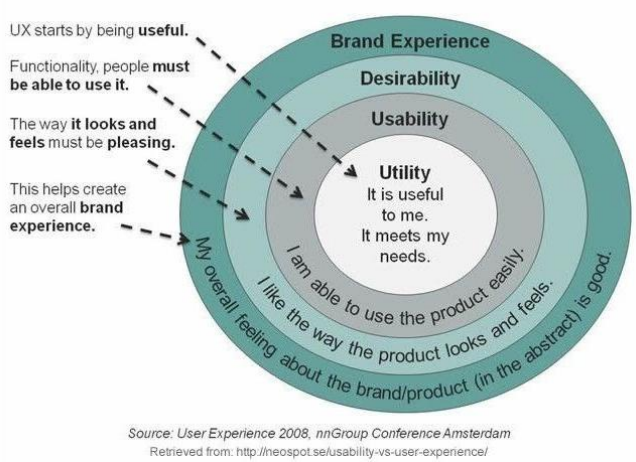
Wie einfach kann eine SW-Applikation benutzt werden

User Experience

Wie fühlt sich die App an
= Usability + Desirability

Customer Experience

Was ist der Gesamteindruck der App, der Marke, der Firma
= Usability + Desirability + Brand Experience



Source: User Experience 2008, nnGroup Conference Amsterdam
Retrieved from: http://neospot.se/usability-vs-user-experience/

Wichtigste 3 Usability-Ziele

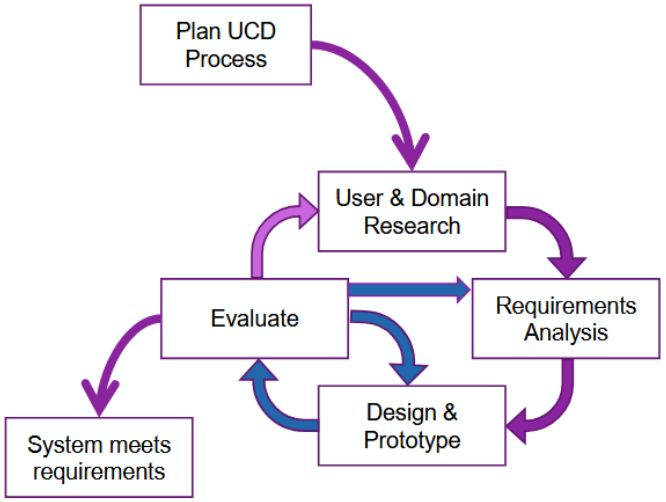
- Effektivität
- Effizienz
- Zufriedenheit

Wichtige Usability Anforderungsbereiche (DIN EN ISO 9241-110)

- Aufgabenangemessenheit
- Lernförderlichkeit
- Individualisierbarkeit
- Erwartungskonformität
- Selbstbeschreibungsfähigkeit
- Steuerbarkeit
- Fehlertoleranz

User-Centered Design (UCD)

Berücksichtigt die Bedürfnisse, Wünsche, Einschränkungen der Benutzer in jeder Phase des Design-Prozesses.



User & Domain Research

Ziele bezüglich User

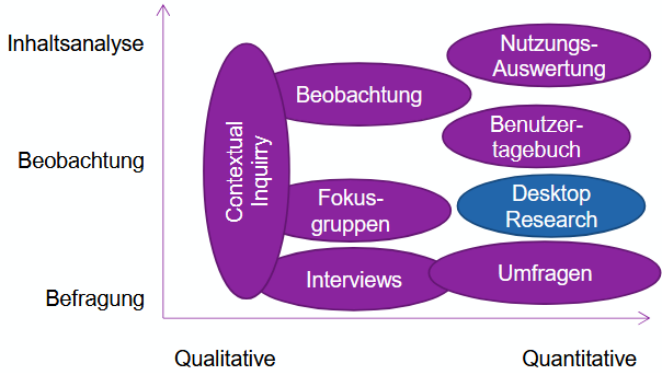
- **Wer** sind die Benutzer?
- **Was** ist ihre Arbeit, ihre Aufgaben, Ziele?
- **Wie** sieht ihre (Arbeits-)Umgebung aus?
- **Was brauchen sie**, um ihre Ziele zu erreichen?
- Welche **Sprache** sprechen sie, welche Begriffe verwenden sie?
- Welche **Normen** sind wichtig für sie (organisatorisch, kulturell, sozial)
- **Pain Points** in ihrer Arbeit (Brüche, Workarounds)

Ziele bezüglich Domäne

- Business der Firma verstehen
- Domäne verstehen
    - Sprache
    - Wichtigste Konzepte
    - Prozesse

Methoden des User & Domain Research

- Interviews
- Beobachtung
- Contextual Inquiry (= Beobachtung+Interview)
- Fokusgruppen
- Umfragen
- Nutzungsauswertung
- Desktop Research
    - Dokumentenstudium
    - Mitbewerber



Requirements Analysis

Ausgehend von den Resultaten des UCD; User-Anforderungen an das zu entwickelnde System ableiten

- Funktionale Abläufe, Interaktionen:
- Kontextszenarien, Storyboards, UI-Skizzen, Use Cases
- Konzepte, Beziehungen, Quantitäten:
- Domänenmodell
- Weitere funktionale/nicht-funktionale Anforderungen, Randbedingungen:
- FURPS-Modell (Functionality, Usability, Reliability, Performance, Supportability)

Design & Prototype

- Entwicklung des Interaktionskonzepts
- Umsetzung des Konzepts mit Interaktionsprototypen

Evaluate

Test des Interaktionskonzepts mit Benutzern und Fachexperten

Wichtige Artefakte des UCD

- *Persona*
- *Szenarien*
- *Domänenmodell*
- *Stakeholder Map*
- *Service Blueprint / Geschäftsprozessmodell*
- UI-Skizzen
- Wireframes

Persona

Eine fiktive Person, welche eine bestimmte Benutzergruppe repräsentiert.

Wichtige Info:

- Name, Alter, Geschlecht, Herkunft
- Beruf, Ausbildung, Erfahrung
- Verantwortlichkeiten, Aufgaben, Persönliche Ziele
- 1-2 Usage Szenarien
- Haltungen, Aktivitäten, Einflüsse
- Fähigkeiten, Bedürfnisse
- Umgebung
- Pain Points und Frustrationen
- Erwartungen an neue Lösung
- Foto, Kernaussage

Szenarien

Usage Szenario	Kontextszenario
Beschreibt die aktuelle Situation - Wie Benutzer seinen Job mit der heutigen Lösung erledigt - Zeigt allfällige Probleme, Workarounds (Pains) auf	Beschreibung einer Interaktion des Benutzers mit dem zukünftigen System Format gleich wie Usage-Szenarien • Beschreiben aber die Zukunft • Nur High Level, keine Lösungskonzepte

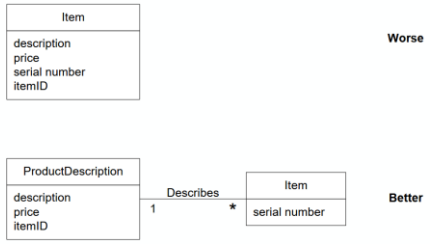
Enthalten typischerweise
- Motivation/Trigger
• Was löst Szenario aus?
- Persona und ihre Ziele
• Info, Artefakt, Emotion?
- Aktionen und Interaktionen
- Kontext
• Wo findet Szenario statt?
• Ändert der Kontext?
• Wer/was ist sonst noch involviert
- Probleme, Ablenkungen
• Welche und wie geht Persona damit um

Domänenmodell

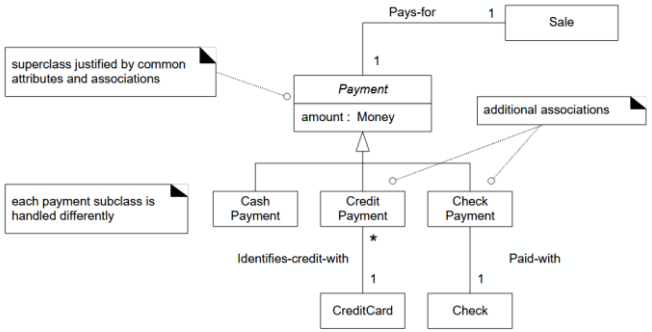
Ein Domänenmodell ist ein vereinfachtes UML-Klassendiagramm. Es zeigt fachliche Begriffe mit ihren Attributen und setzt diese Begriffe zueinander in Beziehung.

Gemeinsame Attribute extrahieren

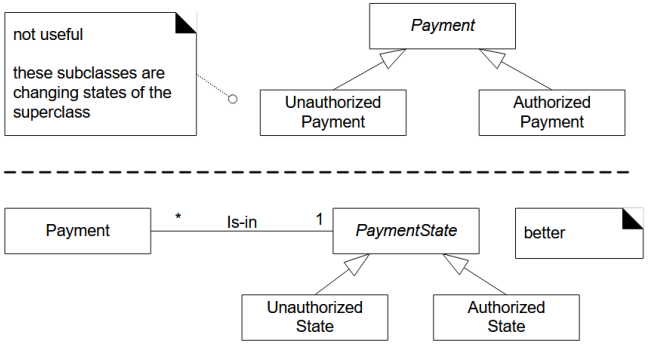
Attribute, die für alle Artikel eines Typs gleich sind, werden in eine eigene Klasse herausgezogen.



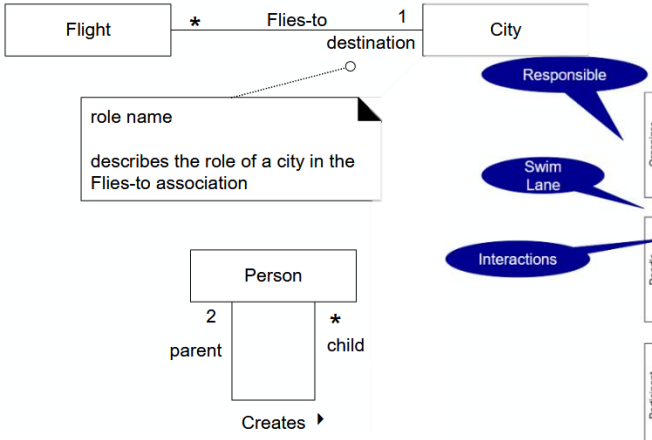
Generalisierung und Spezialisierung



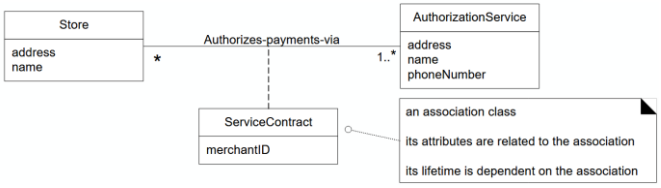
Modellierung von Zuständen



Rollen im Domänenmodell

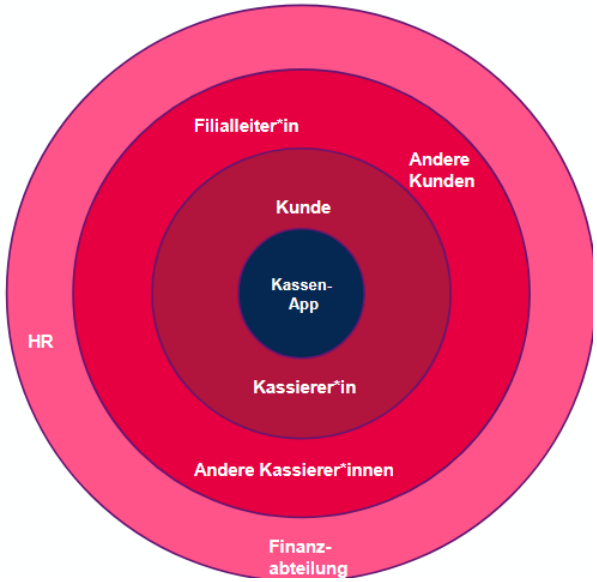


Assoziationsklassen



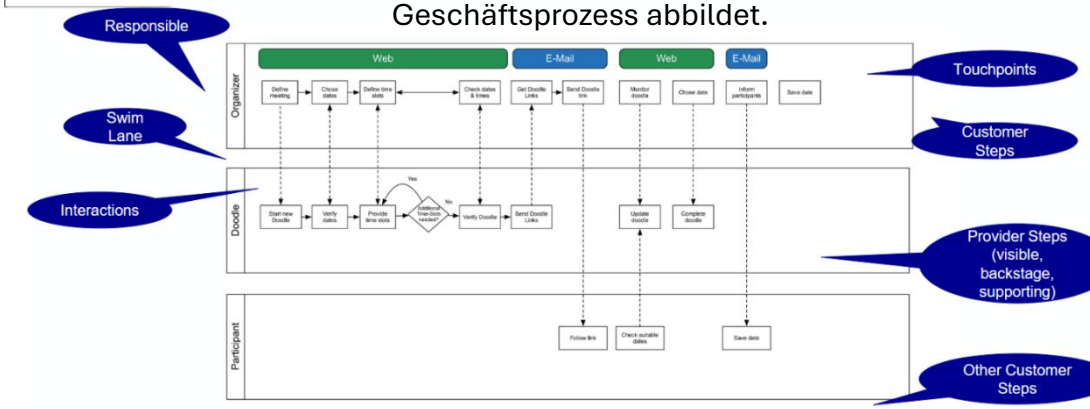
Stakeholder Map

Zeigt die wichtigsten Stakeholder (Personen/Gruppen mit Einfluss auf das Projekt) im Umfeld der Problemdomäne



Service Blueprint / Geschäftsprozessmodell

Ablaufdiagramm, welches einen Geschäftsprozess abbildet.



Use Cases

Textuelle Beschreibung einer konkreten Interaktion eines bestimmten Benutzers mit dem zukünftigen System aus Sicht des Akteurs

Akteure

- 3 Arten von Akteuren
  - Primärakteur (Primary Actor)
    - Initiiert einen Anwendungsfall, um sein (Teil-)Ziel zu erreichen
    - Erhält den Hauptnutzen des Anwendungsfalls
    - Beispiel Kasse: Kassier
  - Unterstützender Akteur (Supporting Actor)
    - Hilft dem SuD bei der Bearbeitung eines Anwendungsfalls
    - Beispiel Kasse: externer Dienstleister wie Zahlungsdienst für Kreditkarten
  - Offstage-Akteur (Offstage Actor)
    - Weitere Stakeholder, die *nicht* direkt mit dem System interagierten
    - Beispiel Kasse: Steuerbehörde

Finden von Use Cases

Schritt 1: Systemgrenzen definieren
Schritt 2: Primärakteure identifizieren
Schritt 3: Ziele/Aufgaben der Primärakteure identifizieren

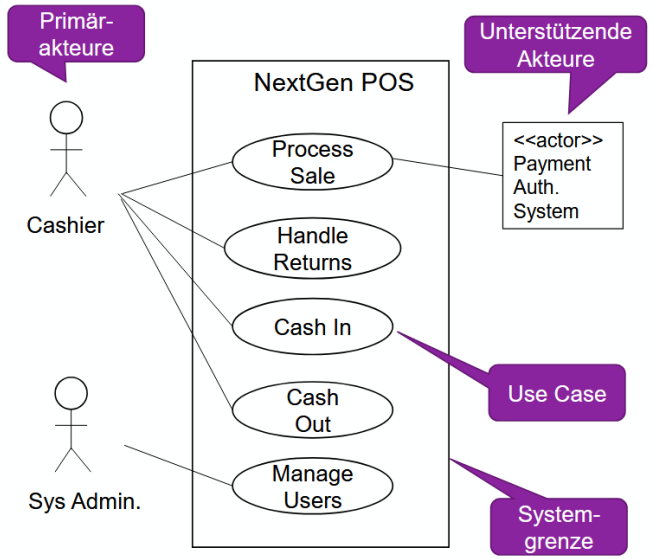
Titel eines UC

- Aktiv formulieren
  - Verb! + evtl. Objekt vorangestellt (z.B. „Kasse eröffnen“)
- Sollte Ziel des Akteurs beschreiben

Gute UC-Namen

- System initialisieren
- System aufstarten
- Artikel erfassen
- (Einen) Einkauf erfassen

Use-Case Diagramm



Brief UC

Kurze Beschreibung des Anwendungsfalls in einem Paragraphen

- Nur Erfolgsszenario
- Sollte enthalten
  - Trigger des UCs
  - Akteure
  - Summarischen Ablauf des UCs
- Wann?
  - Zu Beginn der Analyse

Casual UC

Informelle Beschreibung des Anwendungsfalls in mehreren Paragrafen

- Erfolgsszenario plus wichtigste Alternativszenarien
- Sollte enthalten
  - Trigger des UCs
  - Akteure
  - Interaktion des Akteurs mit System
- Wann?
  - Zu Beginn der Analyse

Fully-dressed UC

Detaillierte Beschreibung des Ablaufs mit allen Alternativszenarien. Die wichtigsten UCs (10%), die die Architektur bestimmen, werden im Detail ausformuliert

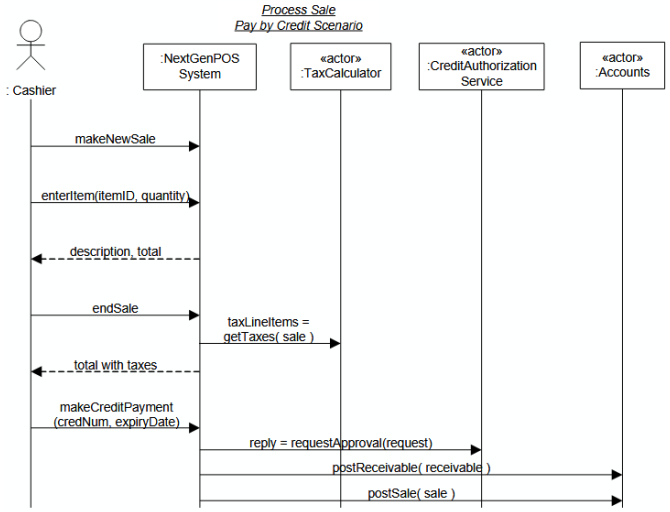
- Wann?
  - Ende der Inception- und v.a. in Elaboration-Phase

Formaler Aufbau

- UC-Name
- Umfang (Scope)
- Ebene (Level)
- Primärakteur (Primary Actor)
- Stakeholders und Interessen
- Vorbedingungen (Preconditions)
- Erfolgsgarantie/Nachbedingungen (Success Guarantee)
- Standardablauf (Main Sucess Scenario)
- Erweiterungen (Extensions)
- Spezielle Anforderungen (Special Requirements)
- Liste der Technik und Datavariationen (Technology and Data Variations)
- Häufigkeit des Auftretens (Frequency of Occurance
- Verschiedenes (Miscellaneous)

Systemsequenzdiagramm (SSD)

Zeigt Interaktionen der Akteure mit dem System. Namen der Akteure und Systeme müssen mit «:» beginnen → Instanz in UML



FURPS+

Checkliste für zusätzliche Anforderungen

- Functionality (Funktionalität)
  - Features, Fähigkeiten, Sicherheit
- Usability (Gebrauchstauglichkeit)
  - Siehe Usability-Anforderungen (LE02)
  - Accessibility (Benutzer mit spez. Bedürfnissen)
- Reliability (Zuverlässigkeit)
  - Fehlerrate, Wiederanlauffähigkeit, Vorhersagbarkeit, Datensicherung
- Performance (Performanz)
  - Reaktionszeiten, Durchsatz, Genauigkeit, Verfügbarkeit, Ressourceneinsatz
- Supportability (Unterstützbarkeit)
  - Anpassungsfähigkeit, Wartbarkeit, Internationalisierung, Konfigurierbarkeit
- +
  - Implementation
    - HW, Betriebssysteme, Sprachen, Tests, Werkzeuge,..
  - Interface
    - Schnittstellen von ext. Systemen, Protokolle
  - Operations
    - Betriebliche Aspekte
  - Packaging (Verpackung)
    - Auslieferung physisch, logisch (Container, Plugin,..)
  - Legal
    - Lizenzen, rechtl. Rahmenbedingungen

Architektur

Die Architektur definiert die tragenden Elemente der Software

Grundprinzip

Aufteilung des Gesamtsystems in möglichst unabhängige Teilsysteme. Diese können unabhängig entwickelt, weiterentwickelt, angepasst, ersetzt werden

Architektur beschreiben (N+1 View Model)

Logical View:

- Welche Funktionalität bietet das System gegen aussen an?
- Wichtige Aspekte: Schichten, Subsysteme, Pakete, Frameworks, Klassen, Interfaces

Process View:

- Welche Prozesse laufen wo und wie ab im System?
- Wichtige Aspekte: Prozesse, Threads, Wie werden Anforderungen wie Performance und Stabilität erreicht?

Development View (Implementation View):

- Wie wurde die logische Struktur (Layer, Schichten,
Komponenten) umgesetzt?
- Wichtige Aspekte: Source Code, Executables, Artefakte

Physical View (Deployment View):

- Auf welcher Infrastruktur wird ein System ausgeliefert/betrieben?
- Wichtige Aspekte: Prozessknoten, Netzwerke, Protokolle

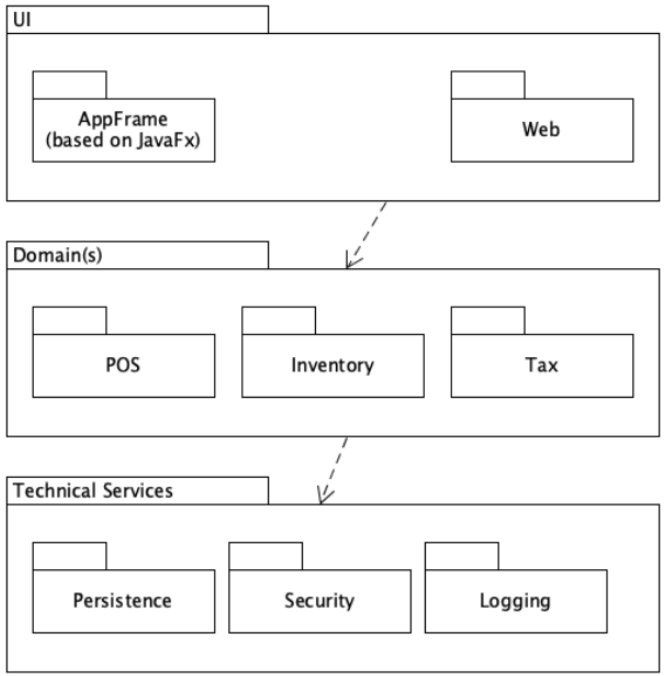
«+1» View: Scenarios (Use Cases)

- Welches sind die wichtigsten Use-Cases und ihre nichtfunktionalen Anforderungen? Wie wurden sie umgesetzt?
- Wichtige Aspekte: Architektonisch wichtige UCs, deren nichtfunktionale Anforderungen und deren Implementation

UML-Paketdiagramme

Paket enthält Klassen und andere Pakete

- Ähnlich, aber allgemeiner als Java Packages



Ausgewählte Architekturpatterns

Pattern	Beschreibung
Layered Pattern	Strukturierung eines Programms in Schichten
Client-Server Pattern	Ein Server stellt Services für mehrere Clients zur Verfügung
Master-Slave Pattern	Ein Master verteilt die Arbeit auf mehrere Slaves
Pipe-Filter Pattern	Verarbeitung eines Datenstroms (filtern, zuordnen, speichern)
Broker Pattern	Meldungsvermittler zwischen verschiedenen Endpunkten
Event-Bus Pattern	Datenquellen publizieren Meldungen an einen Kanal auf dem Event-Bus. Datensenken abonnieren einen bestimmten Kanal
MVC Pattern	Eine interaktive Anwendung wird in 3 Komponenten aufgeteilt: Model, View – Informationsanzeige, Controller – Verarbeitung der Benutzereingabe

UML Design Diagramme

Statische Modelle

Unterstützen den Entwurf von Paketen, Klassennamen, Attributen und Methodensignaturen (ohne Methodenkörper)
Beispiel: UML-Klassendiagramm

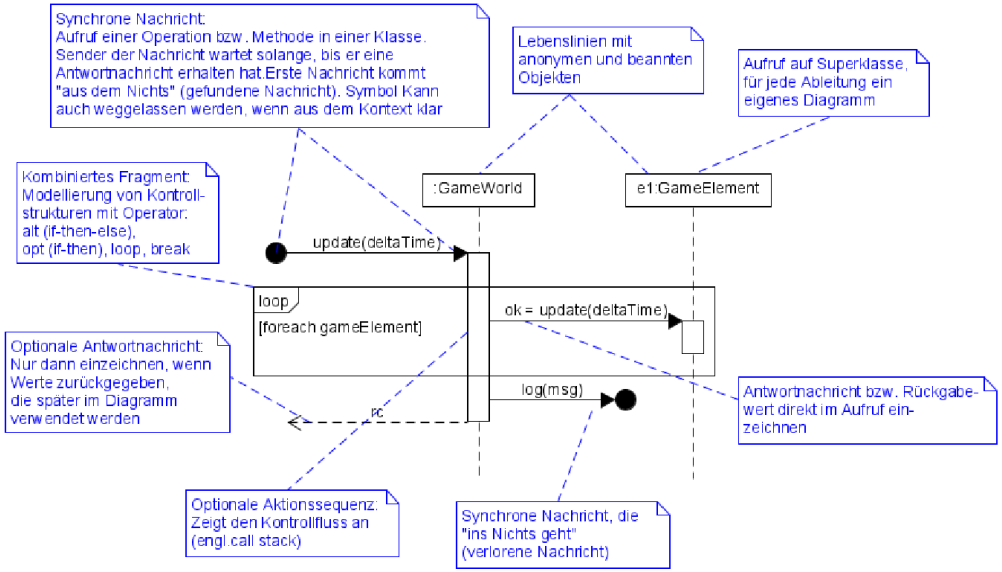
Dynamische Modelle

Unterstützten den Entwurf der Logik, des Verhaltens des Codes und der Methodenkörper
Beispiel: UML-Interaktionsdiagramm

Interaktionsdiagramme

Sequenzdiagramm

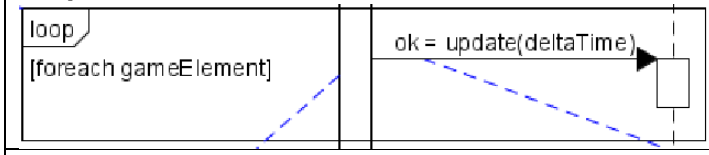
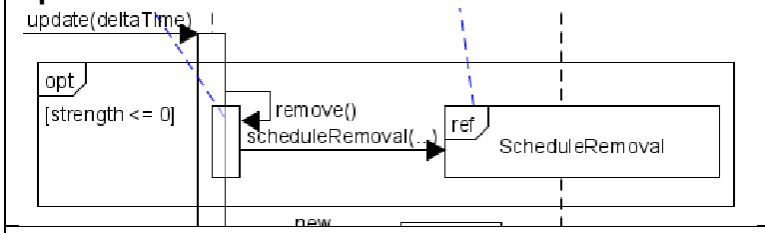
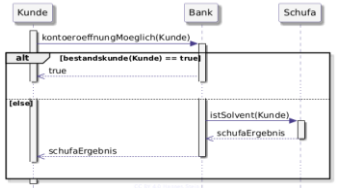
Stellt den zeitlichen Ablauf des Informations-austausches zwischen Kommunikations-partnern dar (mit Schlaufen und Bedingungen)



Wichtig:

- Die Balken über den Lebenslinien stellen aktive Zeit dar, der Balken darf also nur zwischen Aufruf und Return Pfeil sein. Bei «Return im Aufruf» darf der Balken nicht über den nächsten Aufruf hinausreichen, ausser es ist eine asynchrone Nachricht (offene Pfeilspitze).

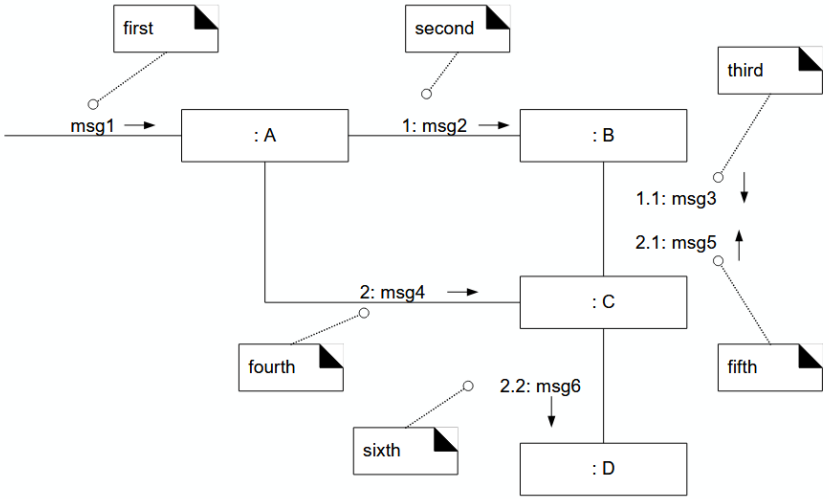
Kontrollstrukturen

loop 	for/while
opt 	if
alt 	if/else

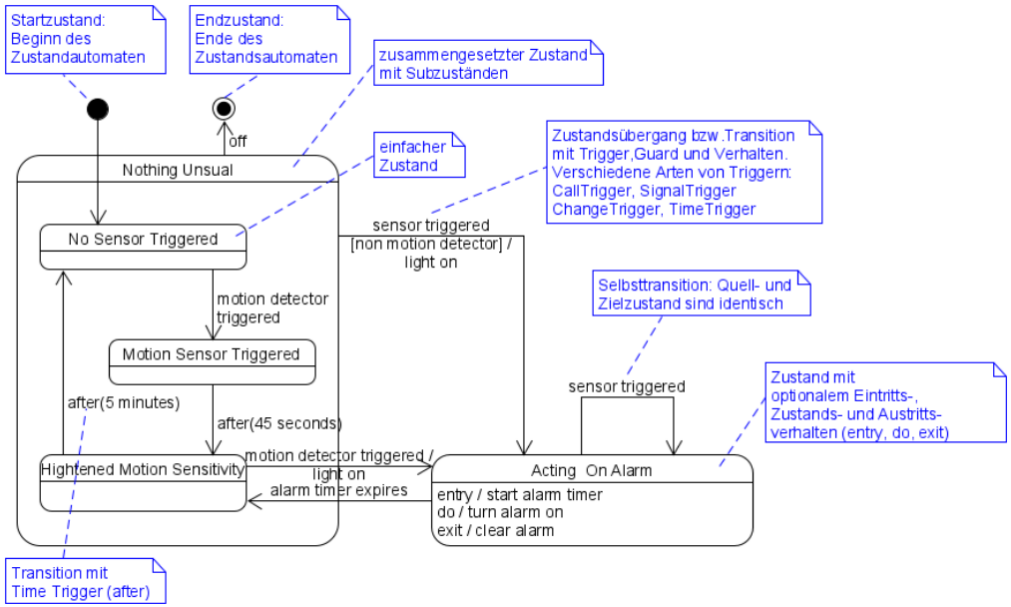
Kommunikationsdiagramm

Wichtig:

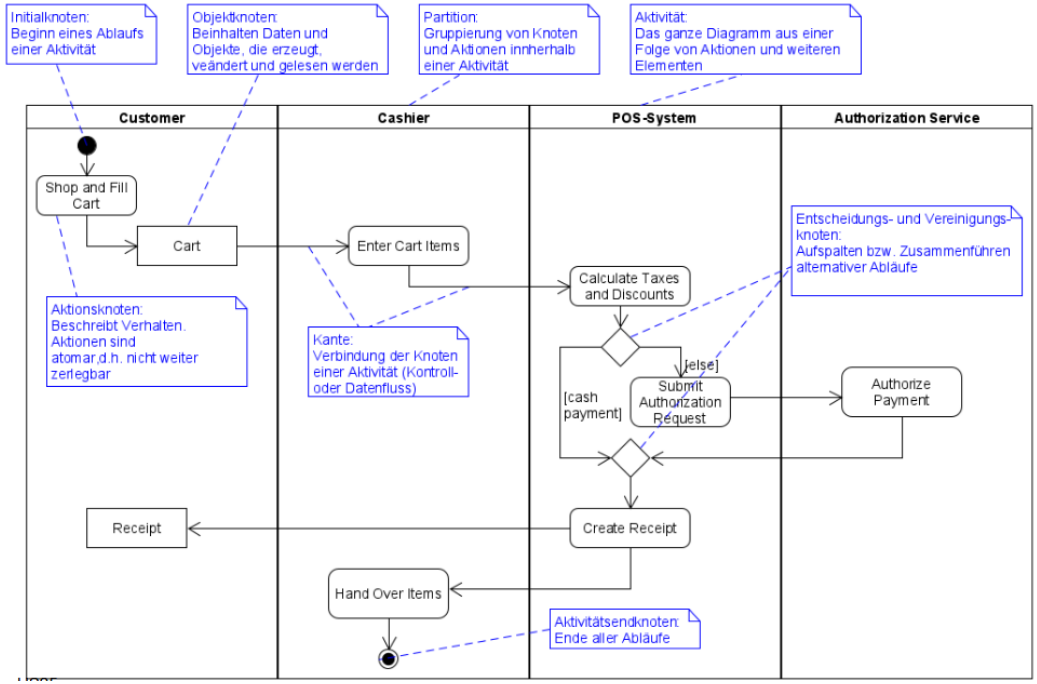
- Nachrichten müssen nummeriert sein, damit der Ablauf ersichtlich bleibt. Hierarchie: Operation *1.1:msg3* wird innerhalb von Operation *1:msg2* aufgerufen



Zustandsdiagramm



Aktivitätsdiagramm



Responsibility-Driven Design (RDD)

Denken in Verantwortlichkeiten, Rollen und Kollaborationsbeziehungen für den Entwurf von Softwareklassen.

GRASP Prinzipien

- **Information Expert:** Ein Objekt sollte die Verantwortung für eine Aufgabe übernehmen, wenn es die benötigten Informationen dazu besitzt.
- **Creator**: Ein Objekt sollte für die Erstellung von anderen Objekten verantwortlich sein, wenn eine starke Beziehung zwischen ihnen besteht.
- **Controller**: Ein Objekt sollte die zentrale Steuerungslogik in einem System repräsentieren.
- **Low Coupling**: Objekte sollten lose miteinander gekoppelt sein, um die Flexibilität und Wiederverwendbarkeit des Systems zu erhöhen.
- **High Cohesion**: Eine Klasse sollte nur zusammengehörige Funktionen und Daten enthalten, um ihre Verständlichkeit und Wartbarkeit zu verbessern.
- **Polymorphism**: Objekte sollten so entworfen werden, dass sie anhand ihrer Schnittstellen verwendet werden können, unabhängig von ihrer spezifischen Implementierung.
- **Pure Fabrication**: Künstliche Klassen sollten erstellt werden, um eine hohe Kohäsion und niedrige Kopplung zu erreichen, wenn keine natürliche Klasse die Verantwortung übernehmen kann.
- **Indirection**: Zwischen Objekten sollten indirekte Verbindungen hergestellt werden, um die Flexibilität und Wartbarkeit zu erhöhen.
- **Protected Variations**: Mechanismen sollten eingeführt werden, um Variationen in den Systemkomponenten zu schützen und unerwünschte Auswirkungen von Änderungen zu minimieren.

Implementierung, Refactoring und Testing

Implementierungsstrategien

Code-Driven Development

- Zuerst die Klasse implementieren

TDD: Test-Driven Development

- Zuerst Tests für Klassen/Komponenten schreiben, dann den Code entwickeln

BDD: Behavior-Driven Development

- Tests aus Benutzersicht beschreiben

Refactoring Patterns

Rename Method / Class / Variable

- Eine Methode/Klasse/Variable wird so umbenannt, dass sie einen aussagekräftigen Namen erhält.

Pull Up / Push Down

- Eine Methode wird in eine Superklasse / Subklasse verschoben.

Extract Interface / Superclass

- Ein Teil eines bestehenden Interfaces / Klasse wird in eine Superinterface / Superklasse extrahiert.

Extract Method

- Teil einer Methode in eine private Methode auslagern.

Extract Constant

- Symbolische Konstante verwenden.

Introduce Explaining Variable

- Grossen Ausdruck aufteilen, erklärende Zwischenvariablen einfügen.

Arten von Tests

• **Funktionaler Test (Black-Box Verfahren):** Überprüft die Funktionalität des Systems, ohne den internen Code zu kennen.
• **Nicht funktionaler Test (Lasttest etc.)**: Testet nicht-funktionale Anforderungen wie Leistung, Skalierbarkeit, usw.
• **Strukturbezogener Test (White-Box Verfahren)**: Überprüft die interne Struktur des Codes, um sicherzustellen, dass alle Pfade abgedeckt sind.
• **Änderungsbezogener Test (Regressionstest etc.)**: Überprüft, ob durch Änderungen im Code keine neuen Fehler eingeführt wurden.
• **Integrationstest:** Eine Klasse wird im Anwendungskontext eingesetzt. Es werden nun keine Mockups, sondern die richtigen referenzierten Klassen Eingesetzt. Typischerweise wird dann ein ganzes Subsystem getestet.
• **Systemtest:** Das ganze System oder die gesamte Anwendungslogik wird getestet. Typischerweise ein Black-Box-Test. Wird nicht nur während der Entwicklung, sondern auch vor einer Auslieferung an den Kunden durchgeführt
• **Abnahmetest:** Nach der Auslieferung wird die gesamte Software vom Kunden getestet. Meist ein Systemtest über das UI. Reiner Black-Box-Test. Orientiert sich an den Anforderungen des Kunden (was er für wichtig hält).

DZone Refcardz

CONTENTS INCLUDE:

Design Patterns

By Jason McDonald

ABOUT DESIGN PATTERNS

This Design Patterns refcard provides a quick reference to the original 23 Gang of Four design patterns, as listed in the book *Design Patterns: Elements of Reusable Object-Oriented Software*. Each pattern includes class diagrams, explanation, usage information, and a real world example.

Creational Patterns: Used to construct objects such that they can be decoupled from their implementing system.

Structural Patterns: Used to form large object structures between many disparate objects.

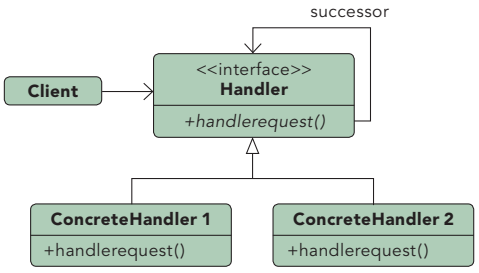
Behavioral Patterns: Used to manage algorithms, relationships, and responsibilities between objects.

Object Scope: Deals with object relationships that can be changed at runtime.

Class Scope: Deals with class relationships that can be changed at compile time.

C	Abstract Factory	S	Decorator	C	Prototype
S	Adapter	S	Facade	S	Proxy
S	Bridge	C	Factory Method	B	Observer
C	Builder	S	Flyweight	C	Singleton
B	Chain of Responsibility	B	Interpreter	B	State
B	Command	B	Iterator	B	Strategy
S	Composite	B	Mediator	B	Template Method
		B	Memento	B	Visitor

CHAIN OF RESPONSIBILITY — Object Behavioral



Purpose

Gives more than one object an opportunity to handle a request by linking receiving objects together.

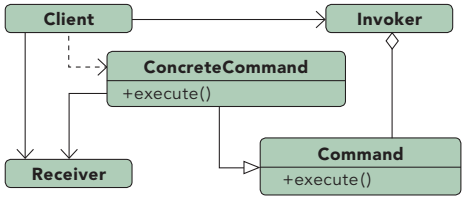
Use When

- Multiple objects may handle a request and the handler doesn't have to be a specific object.
- A set of objects should be able to handle a request with the handler determined at runtime.
- A request not being handled is an acceptable potential outcome.

Example

Exception handling in some languages implements this pattern. When an exception is thrown in a method the runtime checks to see if the method has a mechanism to handle the exception or if it should be passed up the call stack. When passed up the call stack the process repeats until code to handle the exception is encountered or until there are no more parent objects to hand the request to.

COMMAND — Object Behavioral



Purpose

Encapsulates a request allowing it to be treated as an object. This allows the request to be handled in traditionally object based relationships such as queuing and callbacks.

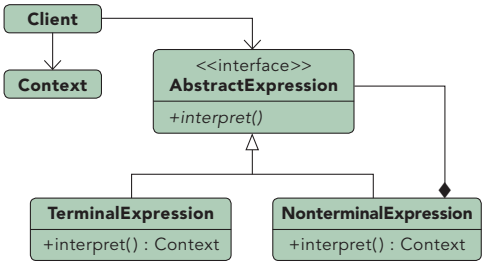
Use When

- You need callback functionality.
- Requests need to be handled at variant times or in variant orders.
- A history of requests is needed.
- The invoker should be decoupled from the object handling the invocation.

Example

Job queues are widely used to facilitate the asynchronous processing of algorithms. By utilizing the command pattern the functionality to be executed can be given to a job queue for processing without any need for the queue to have knowledge of the actual implementation it is invoking. The command object that is enqueued implements its particular algorithm within the confines of the interface the queue is expecting.

INTERPRETER — Class Behavioral



Purpose

Defines a representation for a grammar as well as a mechanism to understand and act upon the grammar.

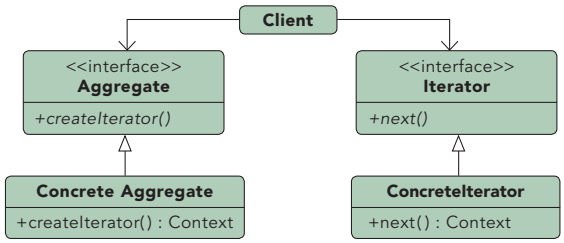
Use When

- There is grammar to interpret that can be represented as large syntax trees.
- The grammar is simple.
- Efficiency is not important.
- Decoupling grammar from underlying expressions is desired.

Example

Text based adventures, wildly popular in the 1980's, provide a good example of this. Many had simple commands, such as "step down" that allowed traversal of the game. These commands could be nested such that it altered their meaning. For example, "go in" would result in a different outcome than "go up". By creating a hierarchy of commands based upon the command and the qualifier (non-terminal and terminal expressions) the application could easily map many command variations to a relating tree of actions.

ITERATOR — Object Behavioral



Purpose

Allows for access to the elements of an aggregate object without allowing access to its underlying representation.

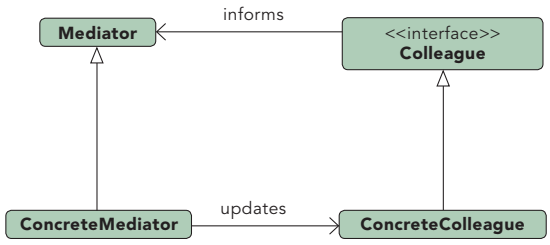
Use When

- Access to elements is needed without access to the entire representation.
- Multiple or concurrent traversals of the elements are needed.
- A uniform interface for traversal is needed.
- Subtle differences exist between the implementation details of various iterators.

Example

The Java implementation of the iterator pattern allows users to traverse various types of data sets without worrying about the underlying implementation of the collection. Since clients simply interact with the iterator interface, collections are left to define the appropriate iterator for themselves. Some will allow full access to the underlying data set while others may restrict certain functionalities, such as removing items.

MEDIATOR — Object Behavioral



Purpose

Allows loose coupling by encapsulating the way disparate sets of objects interact and communicate with each other. Allows for the actions of each object set to vary independently of one another.

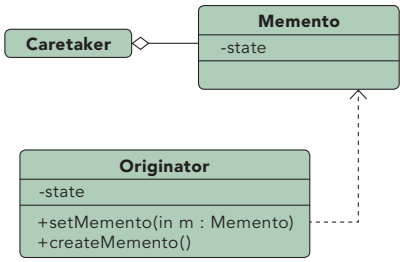
Use When

- Communication between sets of objects is well defined and complex.
- Too many relationships exist and common point of control or communication is needed.

Example

Mailing list software keeps track of who is signed up to the mailing list and provides a single point of access through which any one person can communicate with the entire list. Without a mediator implementation a person wanting to send a message to the group would have to constantly keep track of who was signed up and who was not. By implementing the mediator pattern the system is able to receive messages from any point then determine which recipients to forward the message on to, without the sender of the message having to be concerned with the actual recipient list.

MEMENTO — Object Behavioral



Purpose

Allows for capturing and externalizing an object's internal state so that it can be restored later, all without violating encapsulation.

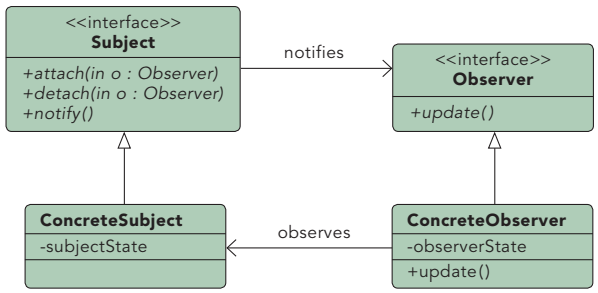
Use When

- The internal state of an object must be saved and restored at a later time.
- Internal state cannot be exposed by interfaces without exposing implementation.
- Encapsulation boundaries must be preserved.

Example

Undo functionality can nicely be implemented using the memento pattern. By serializing and deserializing the state of an object before the change occurs we can preserve a snapshot of it that can later be restored should the user choose to undo the operation.

OBSERVER — Object Behavioral



Purpose

Lets one or more objects be notified of state changes in other objects within the system.

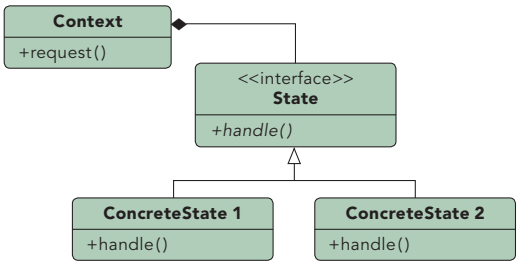
Use When

- State changes in one or more objects should trigger behavior in other objects
- Broadcasting capabilities are required.
- An understanding exists that objects will be blind to the expense of notification.

Example

This pattern can be found in almost every GUI environment. When buttons, text, and other fields are placed in applications the application typically registers as a listener for those controls. When a user triggers an event, such as clicking a button, the control iterates through its registered observers and sends a notification to each.

STATE — Object Behavioral



Purpose

Ties object circumstances to its behavior, allowing the object to behave in different ways based upon its internal state.

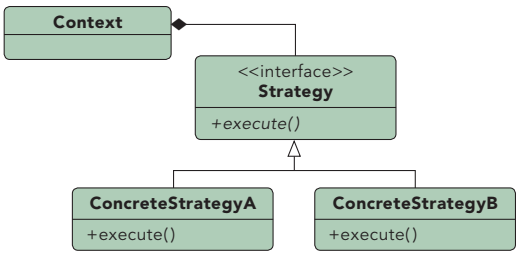
Use When

- The behavior of an object should be influenced by its state.
- Complex conditions tie object behavior to its state.
- Transitions between states need to be explicit.

Example

An email object can have various states, all of which will change how the object handles different functions. If the state is "not sent" then the call to send() is going to send the message while a call to recallMessage() will either throw an error or do nothing. However, if the state is "sent" then the call to send() would either throw an error or do nothing while the call to recallMessage() would attempt to send a recall notification to recipients. To avoid conditional statements in most or all methods there would be multiple state objects that handle the implementation with respect to their particular state. The calls within the Email object would then be delegated down to the appropriate state object for handling.

STRATEGY — Object Behavioral



Purpose

Defines a set of encapsulated algorithms that can be swapped to carry out a specific behavior.

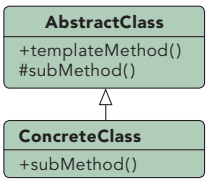
Use When

- The only difference between many related classes is their behavior.
- Multiple versions or variations of an algorithm are required.
- Algorithms access or utilize data that calling code shouldn't be exposed to.
- The behavior of a class should be defined at runtime.
- Conditional statements are complex and hard to maintain.

Example

When importing data into a new system different validation algorithms may be run based on the data set. By configuring the import to utilize strategies the conditional logic to determine what validation set to run can be removed and the import can be decoupled from the actual validation code. This will allow us to dynamically call one or more strategies during the import.

TEMPLATE METHOD — Class Behavioral



Purpose

Identifies the framework of an algorithm, allowing implementing classes to define the actual behavior.

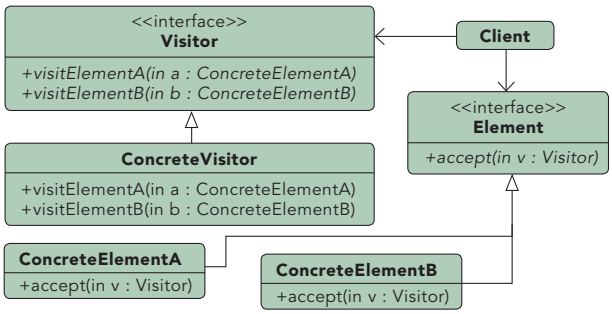
Use When

- A single abstract implementation of an algorithm is needed.
- Common behavior among subclasses should be localized to a common class.
- Parent classes should be able to uniformly invoke behavior in their subclasses.
- Most or all subclasses need to implement the behavior.

Example

A parent class, InstantMessage, will likely have all the methods required to handle sending a message. However, the actual serialization of the data to send may vary depending on the implementation. A video message and a plain text message will require different algorithms in order to serialize the data correctly. Subclasses of InstantMessage can provide their own implementation of the serialization method, allowing the parent class to work with them without understanding their implementation details.

VISITOR — Object Behavioral



Purpose

Allows for one or more operations to be applied to a set of objects at runtime, decoupling the operations from the object structure.

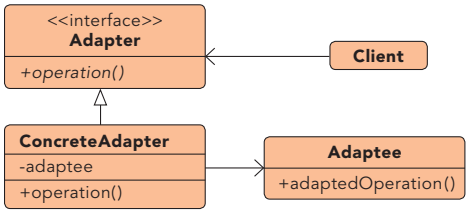
Use When

- An object structure must have many unrelated operations performed upon it.
- The object structure can't change but operations performed on it can.
- Operations must be performed on the concrete classes of an object structure.
- Exposing internal state or operations of the object structure is acceptable.
- Operations should be able to operate on multiple object structures that implement the same interface sets.

Example

Calculating taxes in different regions on sets of invoices would require many different variations of calculation logic. Implementing a visitor allows the logic to be decoupled from the invoices and line items. This allows the hierarchy of items to be visited by calculation code that can then apply the proper rates for the region. Changing regions is as simple as substituting a different visitor.

ADAPTER — Class and Object Structural



Purpose

Permits classes with disparate interfaces to work together by creating a common object by which they may communicate and interact.

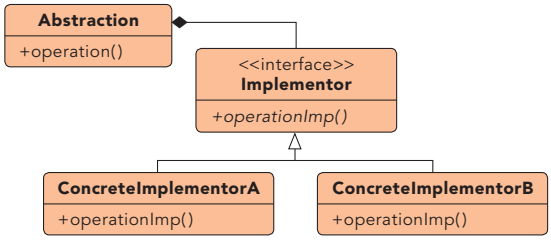
Use When

- A class to be used doesn't meet interface requirements.
- Complex conditions tie object behavior to its state.
- Transitions between states need to be explicit.

Example

A billing application needs to interface with an HR application in order to exchange employee data, however each has its own interface and implementation for the Employee object. In addition, the SSN is stored in different formats by each system. By creating an adapter we can create a common interface between the two applications that allows them to communicate using their native objects and is able to transform the SSN format in the process.

BRIDGE — Object Structural



Purpose

Defines an abstract object structure independently of the implementation object structure in order to limit coupling.

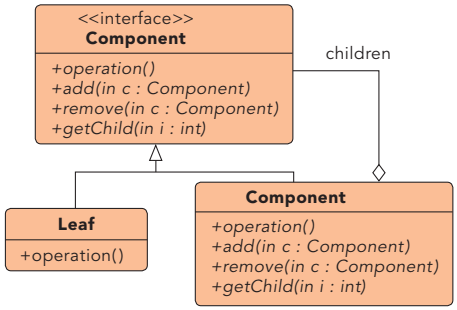
Use When

- Abstractions and implementations should not be bound at compile time.
- Abstractions and implementations should be independently extensible.
- Changes in the implementation of an abstraction should have no impact on clients.
- Implementation details should be hidden from the client.

Example

The Java Virtual Machine (JVM) has its own native set of functions that abstract the use of windowing, system logging, and byte code execution but the actual implementation of these functions is delegated to the operating system the JVM is running on. When an application instructs the JVM to render a window it delegates the rendering call to the concrete implementation of the JVM that knows how to communicate with the operating system in order to render the window.

COMPOSITE — Object Structural



Purpose

Facilitates the creation of object hierarchies where each object can be treated independently or as a set of nested objects through the same interface.

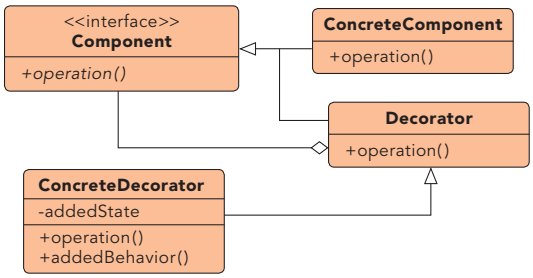
Use When

- Hierarchical representations of objects are needed..
- Objects and compositions of objects should be treated uniformly.

Example

Sometimes the information displayed in a shopping cart is the product of a single item while other times it is an aggregation of multiple items. By implementing items as composites we can treat the aggregates and the items in the same way, allowing us to simply iterate over the tree and invoke functionality on each item. By calling the getCost() method on any given node we would get the cost of that item plus the cost of all child items, allowing items to be uniformly treated whether they were single items or groups of items.

DECORATOR — Object Structural



Purpose

Allows for the dynamic wrapping of objects in order to modify their existing responsibilities and behaviors.

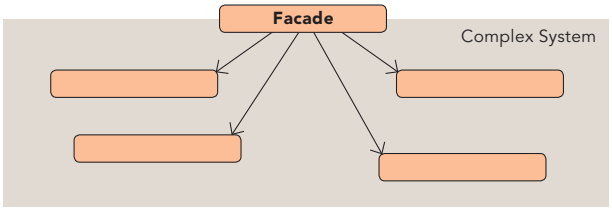
Use When

- Object responsibilities and behaviors should be dynamically modifiable.
- Concrete implementations should be decoupled from responsibilities and behaviors.
- Subclassing to achieve modification is impractical or impossible.
- Specific functionality should not reside high in the object hierarchy.
- A lot of little objects surrounding a concrete implementation is acceptable.

Example

Many businesses set up their mail systems to take advantage of decorators. When messages are sent from someone in the company to an external address the mail server decorates the original message with copyright and confidentiality information. As long as the message remains internal the information is not attached. This decoration allows the message itself to remain unchanged until a runtime decision is made to wrap the message with additional information.

FACADE — Object Structural



Purpose

Supplies a single interface to a set of interfaces within a system.

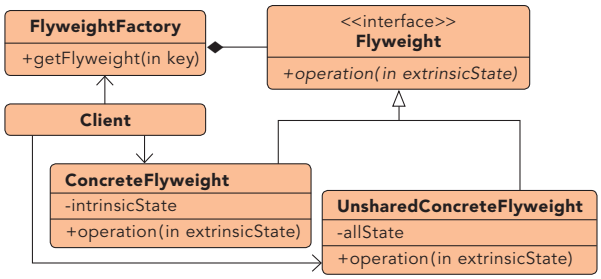
Use When

- A simple interface is needed to provide access to a complex system.
- There are many dependencies between system implementations and clients.
- Systems and subsystems should be layered.

Example

By exposing a set of functionalities through a web service the client code needs to only worry about the simple interface being exposed to them and not the complex relationships that may or may not exist behind the web service layer. A single web service call to update a system with new data may actually involve communication with a number of databases and systems, however this detail is hidden due to the implementation of the façade pattern.

FLYWEIGHT — Object Structural



Purpose

Facilitates the reuse of many fine grained objects, making the utilization of large numbers of objects more efficient.

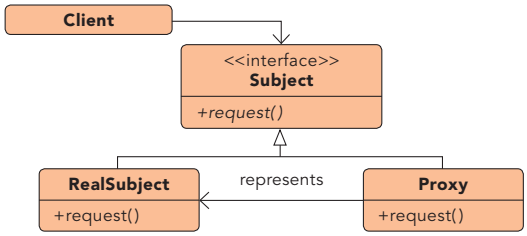
Use When

- Many like objects are used and storage cost is high.
- The majority of each object's state can be made extrinsic.
- A few shared objects can replace many unshared ones.
- The identity of each object does not matter.

Example

Systems that allow users to define their own application flows and layouts often have a need to keep track of large numbers of fields, pages, and other items that are almost identical to each other. By making these items into flyweights all instances of each object can share the intrinsic state while keeping the extrinsic state separate. The intrinsic state would store the shared properties, such as how a textbox looks, how much data it can hold, and what events it exposes. The extrinsic state would store the unshared properties, such as where the item belongs, how to react to a user click, and how to handle events.

PROXY — Object Structural



Purpose

Allows for object level access control by acting as a pass through entity or a placeholder object.

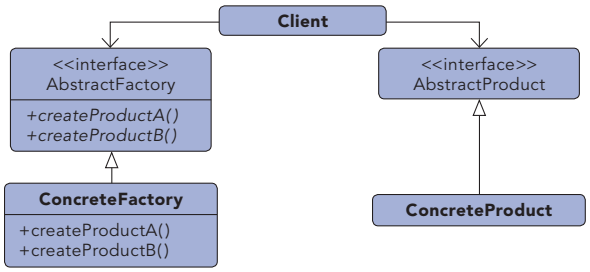
Use When

- The object being represented is external to the system.
- Objects need to be created on demand.
- Access control for the original object is required.
- Added functionality is required when an object is accessed.

Example

Ledger applications often provide a way for users to reconcile their bank statements with their ledger data on demand, automating much of the process. The actual operation of communicating with a third party is a relatively expensive operation that should be limited. By using a proxy to represent the communications object we can limit the number of times or the intervals the communication is invoked. In addition, we can wrap the complex instantiation of the communication object inside the proxy class, decoupling calling code from the implementation details.

ABSTRACT FACTORY — Object Creational



Purpose

Provide an interface that delegates creation calls to one or more concrete classes in order to deliver specific objects.

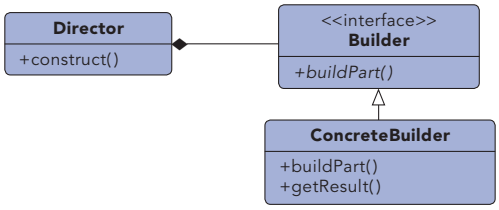
Use When

- The creation of objects should be independent of the system utilizing them.
- Systems should be capable of using multiple families of objects.
- Families of objects must be used together.
- Libraries must be published without exposing implementation details.
- Concrete classes should be decoupled from clients.

Example

Email editors will allow for editing in multiple formats including plain text, rich text, and HTML. Depending on the format being used, different objects will need to be created. If the message is plain text then there could be a body object that represented just plain text and an attachment object that simply encrypted the attachment into Base64. If the message is HTML then the body object would represent HTML encoded text and the attachment object would allow for inline representation and a standard attachment. By utilizing an abstract factory for creation we can then ensure that the appropriate object sets are created based upon the style of email that is being sent.

BUILDER — Object Creational



Purpose

Allows for the dynamic creation of objects based upon easily interchangeable algorithms.

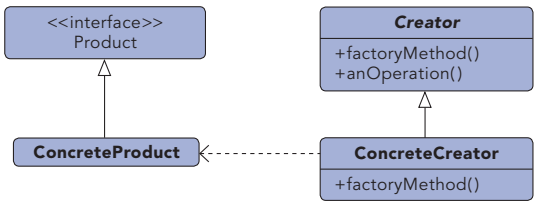
Use When

- Object creation algorithms should be decoupled from the system.
- Multiple representations of creation algorithms are required.
- The addition of new creation functionality without changing the core code is necessary.
- Runtime control over the creation process is required.

Example

A file transfer application could possibly use many different protocols to send files and the actual transfer object that will be created will be directly dependent on the chosen protocol. Using a builder we can determine the right builder to use to instantiate the right object. If the setting is FTP then the FTP builder would be used when creating the object.

FACTORY METHOD — Object Creational



Purpose

Exposes a method for creating objects, allowing subclasses to control the actual creation process.

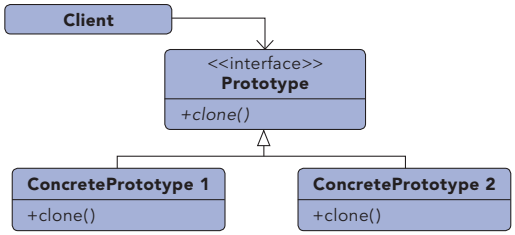
Use When

- A class will not know what classes it will be required to create.
- Subclasses may specify what objects should be created.
- Parent classes wish to defer creation to their subclasses.

Example

Many applications have some form of user and group structure for security. When the application needs to create a user it will typically delegate the creation of the user to multiple user implementations. The parent user object will handle most operations for each user but the subclasses will define the factory method that handles the distinctions in the creation of each type of user. A system may have AdminUser and StandardUser objects each of which extend the User object. The AdminUser object may perform some extra tasks to ensure access while the StandardUser may do the same to limit access.

PROTOTYPE — Object Creational



Purpose

Create objects based upon a template of an existing objects through cloning.

Use When

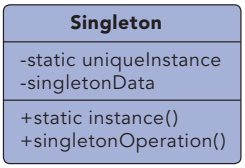
- Composition, creation, and representation of objects should be decoupled from a system.
- Classes to be created are specified at runtime.
- A limited number of state combinations exist in an object.
- Objects or object structures are required that are identical or closely resemble other existing objects or object structures.
- The initial creation of each object is an expensive operation.

Example

Rates processing engines often require the lookup of many different configuration values, making the initialization of the engine a relatively expensive process. When multiple instances of the engine is needed, say for importing data in a multi-threaded manner, the expense of initializing many engines is high. By utilizing the prototype pattern we can ensure that only a single copy of the engine has to be initialized then simply clone the engine to create a duplicate of the already initialized object. The added benefit of this is that the clones can be streamlined to only include relevant data for their situation.

SINGLETON

Object Creational



Purpose

Ensures that only one instance of a class is allowed within a system.

Use When

- Exactly one instance of a class is required.
- Controlled access to a single object is necessary.

Example

Most languages provide some sort of system or environment object that allows the language to interact with the native operating system. Since the application is physically running on only one operating system there is only ever a need for a single instance of this system object. The singleton pattern would be implemented by the language runtime to ensure that only a single copy of the system object is created and to ensure only appropriate processes are allowed access to it.

ABOUT THE AUTHOR

Jason McDonald

The product of two computer programmers, Jason McDonald wrote his first application in BASIC while still in elementary school and has been heavily involved in software ever since. He began his software career when he found himself automating large portions of one of his first jobs. Finding his true calling, he quit the position and began working as a software engineer for various small companies where he was responsible for all aspects of applications, from initial design to support. He has roughly 11 years of experience in the software industry and many additional years of personal software experience during which he has done everything from coding to architecture to leading and managing teams of engineers. Through his various positions he has been exposed to design patterns and other architectural concepts for years. Jason is the founder of the Charleston SC Java Users Group and is currently working to help found a Charleston chapter of the International Association of Software Architects.

Personal Blog: http://www.mcdonaldland.info/
Projects: Charleston SC Java Users Group

RECOMMENDED BOOK

Capturing a wealth of experience about the design of object-oriented software, four top-notch designers present a catalog of simple and succinct solutions to commonly occurring design problems. Previously undocumented, these 23 patterns allow designers to create more flexible, elegant, and ultimately reusable designs without having to rediscover the design solutions themselves.

BUY NOW
books.dzone.com/books/designpatterns

Get More FREE Refcardz. Visit refcardz.com now!

Upcoming Refcardz:

Core Seam
Core CSS: Part III
Hibernate Search
Equinox
EMF
XML
JSP Expression Language
ALM Best Practices
HTML and XHTML

Available:

Essential Ruby
Essential MySQL
JUnit and EasyMock
Getting Started with MyEclipse
Spring Annotations
Core Java
Core CSS: Part II
PHP
Getting Started with JPA
JavaServer Faces
Core CSS: Part I
Struts2
Core .NET
Very First Steps in Flex
C#
Groovy
NetBeans IDE 6.1 Java Editor
RSS and Atom
GlassFish Application Server
Silverlight 2

Visit refcardz.com for a complete listing of available Refcardz.

Design Patterns
Published June 2008

DZone

DZone communities deliver over 4 million pages each month to more than 1.7 million software developers, architects and decision makers. DZone offers something for everyone, including news, tutorials, cheatsheets, blogs, feature articles, source code and more.
"DZone is a developer's dream," says PC Magazine.

DZone, Inc.
1251 NW Maynard
Cary, NC 27513
888.678.0399
919.678.0300
Refcardz Feedback Welcome
refcardz@dzone.com
Sponsorship Opportunities
sales@dzone.com

ISBN-13: 978-1-934238-10-3
ISBN-10: 1-934238-10-4
50795
9 781934 238103
$7.95

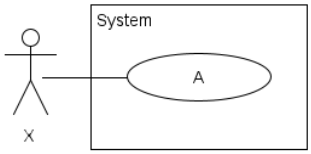
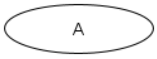
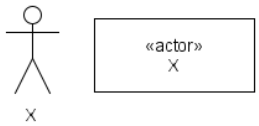
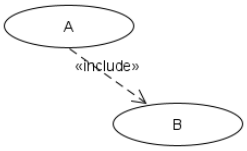
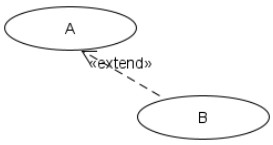
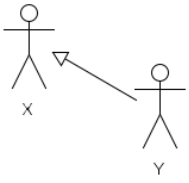
School of Engineering
InIT Institut für Informatik

Bachelor of Science (BSc) in Informatik
Modul Software-Entwicklung 1 (SWEN1)

Cheat Sheet UML

Die wichtigsten Notationselemente der UML 2

Anwendungsfalldiagramm (*Use Case Diagram*)

Name	Notation	Erklärung
System (*system*)		Grenzen zwischen dem System und den Akteuren des betrachteten Systems (SuD)
Anwendungsfall (*use case*)		Beschreibt ein Verhalten des Systems, das einem Akteur zur Verfügung gestellt wird; ein Use Case beschreibt viele verschiedene Szenarien (Standardablauf, Erweiterungen); ein Szenario ist ein bestimmter Ablauf im Use Case (Instanz)
Akteur (*actor*)		Rolle der Systembenutzer, die mit dem Use Case interagieren; für externe Systeme wird die Rechtecknotation verwendet
Assoziation (*association*)		Kommunikationsbeziehung zwischen Use Cases und Akteuren; bei mehreren Akteuren kann der initiierende Aktor mit gerichtetem Pfeil markiert werden
Include-Beziehung (*include relationship*)		A include B: notwendiges Verwenden von Use Case B durch Use Case A; in der Use-Case-Spezifikation A wird in einem Szenario der Use Case A aufgerufen (referenziert)
Extend-Beziehung (*extend relationship*)		B extend A: optionales Verwenden von Use Case B durch Use Case A; in der Use-Case-Spezifikation A wird in einem Szenario und bestimmten Bedingungen der Use Case B aufgerufen (referenziert)
Aktor Generalisierung (*actor generalization*)		Y erbt von X; Y kommuniziert mit allen Use Cases, mit denen X kommuniziert

Use-Case-Spezifikation *(Use Case Specification)*

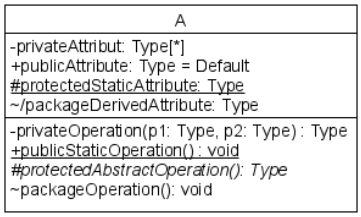
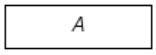
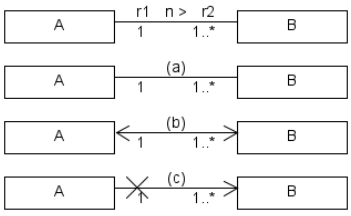
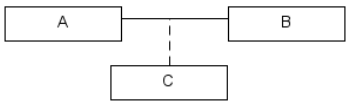
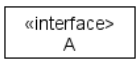
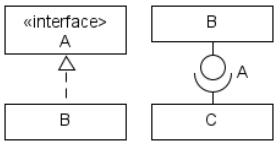
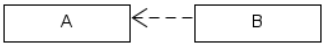
Use-Case-Name *(Use Case Name)*	Titel des Use Casest und dann das Verb: Verkauf bearbeiten (im englischen umgekehrt: Process Sale)
Umfang *(Scope)*	(fakultativer Abschnitt) Das System unter Design (SuD), das zu entwerfende System
Ebene *(Level)*	(fakultativer Abschnitt) «Anwenderziel» oder «Subfunktion»
Primärakteuer *(Primary Actor)*	Nutzt die Use Cases des Systems
Stakeholder und Interessen *(Stakeholders and Interests)*	(fakultativer Abschnitt) Für wen ist der Use Case interessant und welches Interesse hat er daran?
Vorbedingung(en) *(Preconditions)*	Was muss am Anfang gewährleistet und für den Leser mitteilenswert sein?
Erfolgsgarantie, Nachbedingungen(en) *(Success Garantee)*	Was muss nach der erfolgreichen Fertigstellung gewährleistet und für den Leser mitteilenswert sein?
Standardablauf *(Main Success Scenario)*	Das Szenario für einen typischen, unbedingten erfolgreichen Durchlauf durch den Use Case
Erweiterungen *(Extensions)*	Alternative Szenarios für Erfolg und Misserfolg
Spezielle Anforderungen *(Special Requirements)*	(fakultativer Abschnitt) Zum System gehörige, nichtfunktionale Anforderungen (Qualitätsanforderungen u.a.)
Liste der Technik- und Datenvariation *(Technology and Data Variations List)*	(fakultativer Abschnitt) Alternative I/O-Methoden und Datenformate
Häufigkeit des Auftretens *(Frequency of Occurrence)*	(fakultativer Abschnitt) Beeinflusst die Untersuchung, das Testen und die zeitliche Gestaltung der Implementierung
Verschiedenes *(Miscellaneous)*	(fakultativer Abschnitt) Beispielsweise offene Probleme

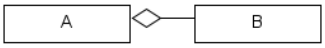
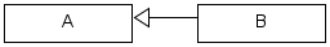
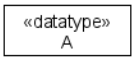
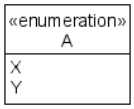
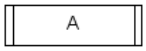
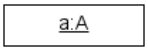
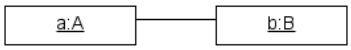
Standardablauf und Erweiterungen sind die beiden Hauptabschnitte.

Fakultative Abschnitte werden beschrieben, wenn es für das Verstehen des Use Cases nützlich und für den Leser mitteilungswert ist.

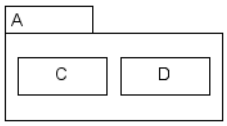
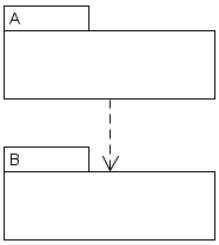
Quelle: Craig Larman, UML 2 und Patterns angewendet, mitp Professional, 2005

Klassen- und Objektdiagramm (*Class and Object Diagram*)

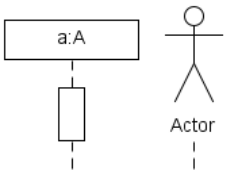
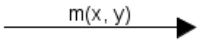
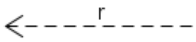
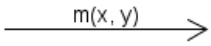
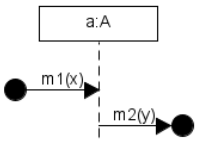
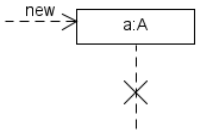
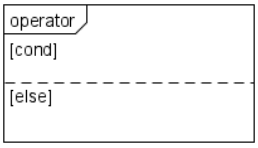
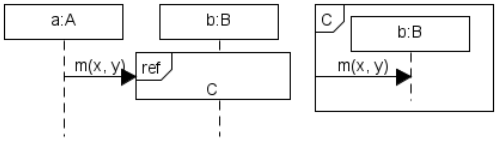
Name	Notation	Erklärung
Klasse (*class*)		Beschreibung der Struktur (Attribute) und des Verhaltens (Operationen) einer Menge von Objekten; zusätzliche Spezifikation von Visibilität, abstrakten Operationen und statischen Operationen/Attributen
Abstrakte Klasse (*abstract class*)		Klasse, die nicht instanziiert werden kann
Assoziation (*asscoiation*)		Beziehung zwischen Klassen mit optional Assoziationsname n und Leserichtung >, Rollenbezeichnungen r1, r2 und Multiplizitäten (0, 1, n, n..m, *); keine Angabe über Navigationsrichtung (a), mit Navigationsrichtung (b), in eine Richtung nicht navigierbar (c)
Assoziationsklasse (*association class*)		Nähere Beschreibung einer Assoziation; Klasse C mit Attributen und Operationen
Interface (*interface*)		Beschreibt eine Menge von öffentlichen Operationen, Merkmalen und «Verpflichtungen», die durch eine Klasse, die die Schnittstelle implementiert, zwingend bereitgestellt werden müssen
Realisierungsbeziehung (*realization relationship*)		Klasse B implementiert das Interface A; Alternative Darstellung mit Lollipop- und Socket-Notation für angebotenes und erforderliches Interface
Abhängigkeitsbeziehung (*dependency relationship*)		Drückt generell eine Abhängigkeit zwischen Modellelementen aus. B braucht A zur Erfüllung von B (z.B. als Parameter); eine spezielle Abhängigkeit ist die Verwendungsbeziehung (Stereotyp <<use>>)

Komposition (*composition*)		Existenzabhängige Teile-Ganzes-Beziehung (B ist Teil von A; wenn A gelöscht wird, werden zugehörige Instanzen von B ebenfalls gelöscht)
Aggregation (*aggregation*)		Teile-Ganzes-Beziehung (B ist Teil von A; wenn A gelöscht wird, werden zugehörige Instanzen von B nicht gelöscht)
Generalisierung / Spezialisierung (*generalization / specialization*)		Vererbungsbeziehung zwischen Klassen (B erbt von A)
Datentyp (*data type*)		Eigener Datentyp mit Attributen und Operationen
Enumeration (*enumeration*)		Eigener Aufzählungstyp und allenfalls Operationen
Aktive Klasse (*active class*)		Objekte besitzen einen eigenen Thread oder den ausführenden Prozess/Task
Objekt (*object*)		Instanz einer Klasse
Link (*link*)		Beziehung zwischen Objekten

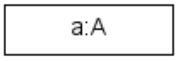
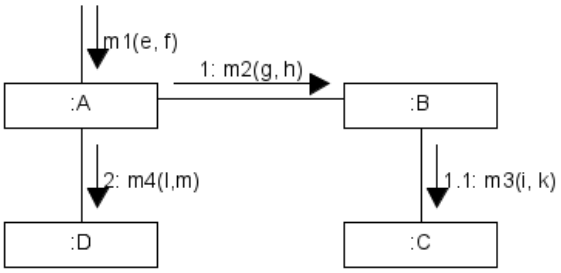
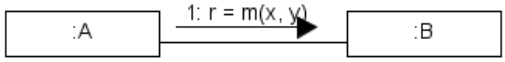
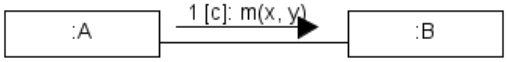
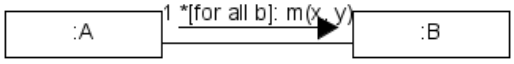
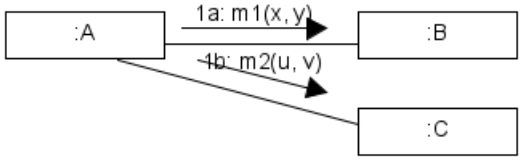
Paketdiagramm *(Package Diagram)*

Name	Notation	Erklärung
Paket (*package*)		Fasst mehrere paketierbare Elemente (Pakete, Klassen, Interfaces, Datentypen etc.) zu einer grösseren Einheit zusammen und bildet einen Namensraum
Abhängigkeitsbeziehung (*dependency relationship*)		Beschreibt eine generelle Abhängigkeit zwischen dem Paket A und B; eine Abhängigkeit kommt in der Regel dadurch zustande, dass das abhängige Paket (A) Pakete, Klassen, Interface und/oder Datentypen aus dem unabhängigen Paket (B) importiert; diese Abhängigkeit kann mit dem Stereotyp <<import>> auf dem Pfeil noch markiert werden

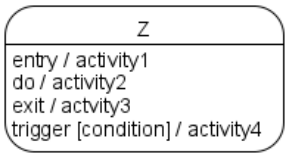
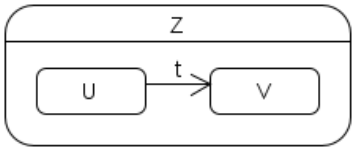
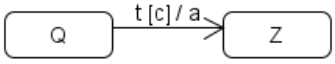
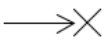
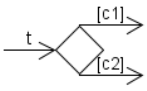
Sequenzdiagramm (*Sequence Diagram*)

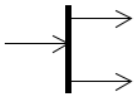
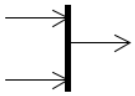
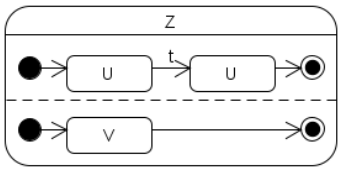
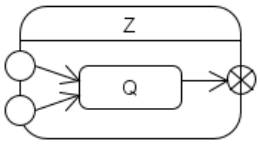
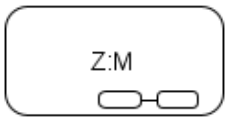
Name	Notation	Erklärung
Lebenslinie (*life line*)		An der Kommunikation beteiligte Interaktionspartner; optional mit Aktionssequenz (execution specification), die zeigt, wie lange das Objekt aktiv ist (den Kontrollfluss hat)
Synchrone Nachricht (*synchronous message*)		Verschicken der Nachricht m (Methodenaufruf) mit den Parametern x und x; Sender wartet auf eine Antwortnachricht bis er weiterfährt mit der Verarbeitung
Antwortnachricht (*return message*)		Antwort mit Rückgabewert r auf synchrone Nachricht
Asynchrone Nachricht (*asynchronous message*)		Verschicken der Nachricht m (Methodenaufruf oder Signal) mit den Parametern x und y; Sender setzt nach Absenden der asynchronen Nachricht Verarbeitung sofort fort
Gefundene, verlorene Nachricht (*found, lost message*)		Nachricht m1 von unbekanntem Sender («aus dem Nichts») bzw. Nachricht m2 an unbekannten Empfänger («ins Nichts»)
Erzeugungs-, Löschereignis (*create,destroy message*)		Zeitpunkt, zu dem ein Interaktionspartner erzeugt (new oder create) bzw. aufhört zu existieren; letzteres kann auch explizit mit einer «destroy»-Nachricht modelliert werden
Kombiniertes Fragment (*combined fragment*)		Steuerung des Kontrollflusses mit Operatoren und Bedingungen; Operatoren: loop, alt, opt, break, par
Interaktionsreferenz (*interaction reference/ use*)		Aufruf eines anderen Interaktionsdiagramms

Kommunikationsdiagramm (*Communication Diagram*)

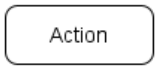
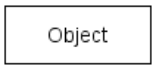
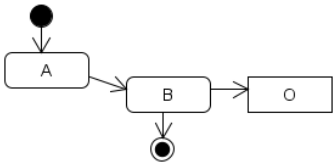
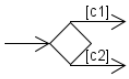
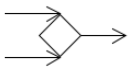
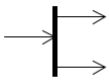
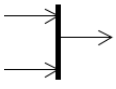
Name	Notation	Erklärung
Lebenslinie (*life line*)		An der Kommunikation beteiligte Interaktionspartner
Synchrone Nachricht (*synchronous message*)		Sender wartet auf eine Antwortnachricht; erste Nachricht m1 ohne Nummer, folgende Nachrichten (m2, m3, m4) werden hierarchisch nummeriert
Antwortnachricht (*return message*)		Antwort r auf synchrone Nachricht m
Bedingte Nachricht (*conditional message*)		Bedingte Ausführung mit Bedingung c der Nachricht m
Iteration (*iteration*)		Iteration über eine Menge von Objekten; für die Iteration über alle Collection-Elemente einer Collection ist nur die Angabe des «*» auch präzise genug
Parallele Nachricht (*parallel message*)		Nachrichten m1 und m2 werden parallel ausgeführt (1a und 1b)

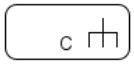
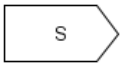
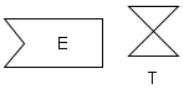
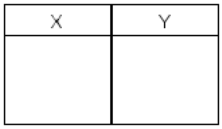
Zustandsdiagramm (*State Machine Diagram*)

Name	Notation	Erklärung
einfacher Zustand (*simple state*)		Beschreibung einer spezifischen «Zeitspanne», in der sich ein Objekt während seines «Lebenslaufs» befindet; im Zustand können Aktivitäten auf dem Objekt ausgeführt werden (entry, do, exit) und weitere «innere» Transitionen definiert werden (trigger)
Zusammengesetzter bzw. geschachtelter Zustand (*composite state*)		Zustand Z mit den Subzuständen U und V; beliebige Schachtelungstiefe möglich
Transition (*transition*)		Zustandsübergang von einem Quellzustand Q in einen Zielzustand Z mit Trigger t, Bedingung c und der Aktivität a; Trigger können sein: CallTrigger (Methodenaufruf), SignalTrigger (asynchrones Signal), ChangeTrigger (when), TimeTrigger (after)
Startzustand (*initial state*)		Beginn des Zustandsautomaten
Endzustand (*final state*)		Ende des Zustandsautomaten
Terminator (*terminate state*)		Bricht die Ausführung des Zustandsautomaten ab (Lebensdauer des Objektes ist beendet)
Flacher/Tiefer History-Zustand (*shallow/deep history*)		«Rücksprungadresse» in einen Subzustand bzw. geschachtelten Subzustand eines zusammengesetzten Zustands; flache History («H») berücksichtigt nur eine Ebene und tiefe History («H*») beliebig viele Ebenen
Entscheidungsknoten (*decision node*)		Knoten, von dem mehrere alternative Transitionen ausgehen können; an der eingehenden Kante wird der Trigger t und an den ausgehenden Kanten die Bedingungen c1, c2 spezifiziert

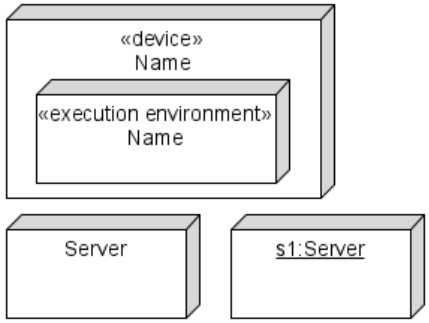
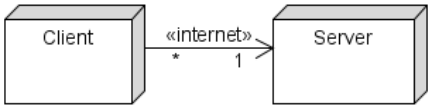
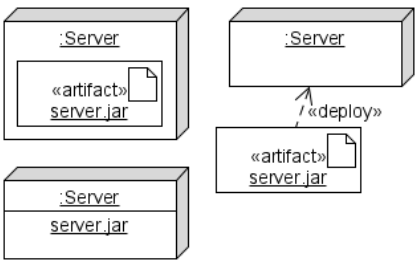
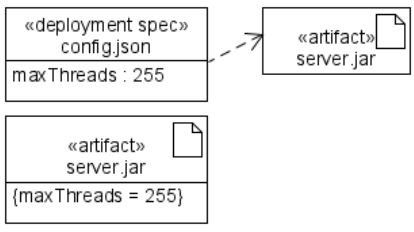
Parallelisierungsknoten (*fork node*)		Aufspaltung einer Transition in mehrere parallele Transitionen
Synchronisationsknoten (*join node*)		Zusammenführung von mehreren parallelen Transitionen in eine Transition
Orthogonaler Zustand (*orthogonal state*)		Zusammengesetzter Zustand mit Regionen, die parallel ausgeführt werden; Regionen können auch noch benannt werden
Einstiegs-/Ausstiegspunkt (*entry/exit point*)		Dienen der Übersichtlichkeit in zusammengesetzten Zuständen und Unterzustandsautomaten, um Transitionen zu ordnen und um eine Schnittstelle nach aussen zu definieren
Unterzustandsautomat (*submachine state*)		Der Zustand Z steht stellvertretend für den Zustandsautomaten M

Aktivitätsdiagramm (*Activity Diagram*)

Name	Notation	Erklärung
Aktionsknoten, Aktion (*action node, action*)		Aktionen beschreiben beliebiges Benutzer- oder Programm-Verhalten; Aktionen sind atomar, d. h. nicht weiter zerlegbar; eine Aktivität (activity) beschreibt eine Menge von Aktionen und deren Ablauf
Objektknoten, Objekt (*object node, object*)		Enthalten Daten und Objekte, die erzeugt, verändert und gelesen werden
Initialknoten (initial node)		Beginn eines Ablaufs einer Aktivität
Aktivitätsendknoten (*activity final node*)		Ende aller Abläufe einer Aktivität
Kante (*activity edge*)		Verbindung der Knoten einer Aktivität (Kontrollfluss oder Datenfluss)
Entscheidungsknoten (*decision node*)		Aufspaltung eines Ablaufs in alternative Abläufe; an den ausgehenden Kanten sind Bedingungen c1, c2 spezifiziert
Vereinigungsknoten (*merge node*)		Zusammenführung von alternativen Abläufen in einen Ablauf
Parallelisierungsknoten (*fork node*)		Aufspaltung eines Ablaufs in mehrere parallele Abläufe
Synchronisationsknoten (*join node)*		Zusammenführung von mehreren parallelen Abläufen in einen Ablauf

Ablaufendknoten (*flow final node*)		Ende von einem Ablauf einer Aktivität (bei parallelen Abläufen)
CallBehavior-Aktion (*call behavior action*)		Aktion C verweist auf eine Aktivität gleichen Namens (Unteraktivität)
SendSignal-Aktion (*send signal action*)		Asynchrone Übermittlung eines Signals S an einen Empfänger
Asynchrone Ereignis-/Zeitereignisannahme-Aktion (*accept event/time action*)		Warten auf ein Ereignis E bzw. ein Zeitereignis T
Partition (*activity partition, swimlane*)		Gruppierung von Knoten und Kanten innerhalb einer Aktivität

Verteilungsdiagramm (*Deployment Diagram*)

Name	Notation	Erklärung
Knoten (*node*)		Ein Knoten repräsentiert eine Ressource, die z.B. zur Installation, Konfiguration, Bereitstellung und Ausführung von Artefakten genutzt werden kann; Knoten können sowohl auf der Typ-Ebene als auch auf der Ausprägungs-Ebene definiert werden; zwei vordefinierte Knoten sind: Gerät (device) und Ausführungsumgebung (execution environement)
Kommunikationspfad (*communication path*)		Durch Kommunikationspfade können Knoten sowohl auf Typ- als auch auf Instanz-Ebene miteinander verbunden werden, um Nachrichten (Signale oder Operationsaufrufe) auszutauschen; optional können Kommunikationsrichtung, den Typ der Kommunikation sowie Multiplizitäten spezifiziert werden (nur auf Typ-Ebene)
Verteilungsbeziehung (*deployment*)		Die Verteilungsbeziehung (deployment) repräsentiert die Beziehung zwischen einem Artefakt und dem Knoten, auf den das Artefakt verteilt ist; es gibt drei verschiedene Darstellungsformen; vordefinierte spezielle Artefakte sind: <<library>>, <<file>>, <<document>>
Einsatzspezifikation (*deployment specification*)		Eine Einsatzspezifikation (deployment specification) ist eine spezielle Art eines Artefakts; sie beinhaltet eine Menge von Parametern, die durch Angabe von Werten die Verteilung eines Artefakts auf Knoten regelt (Konfiguration); es gibt zwei verschiedene Darstellungsarten