Info 3 Intro

WS 2025/2026

Alexander Kramer

Process - How to build a Web App?

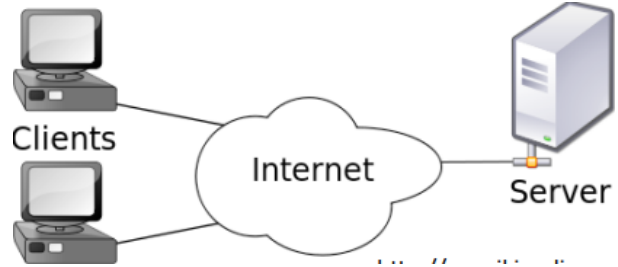
First rough Idea - Activities, not Phases

- Requirements Engineering/Definition | Product Concept
  - Lastenheft / Pflichtenheft
- System and Software Design
- Implementation & Integration
- Test and Deployment / “Maintenance”

Architecture - How to build a Web App?

Client-Server-Modell

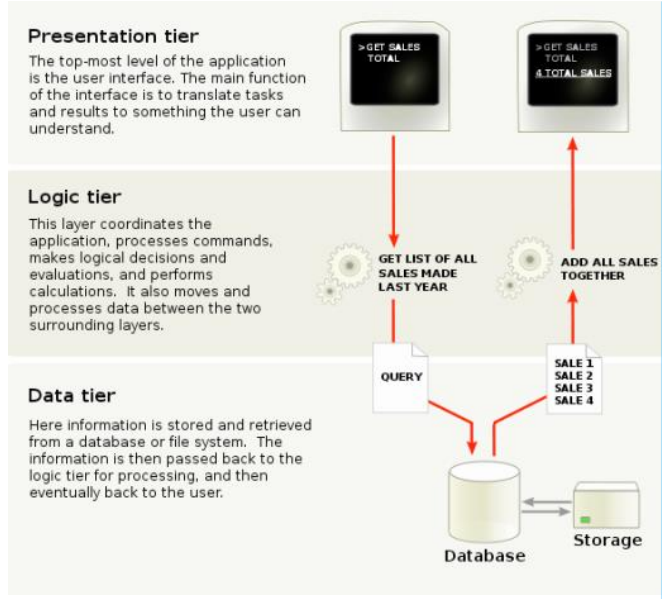
- partitions application between providers (server) and consumers (clients) of a service
- multiple clients
- clients initiates communication
- usually connected via network



https://en.wikipedia.org/wiki/File:Client-server-model.svg

3 Tier Architecture

- *Tier* often implies physical separation
- The domain and data source should never be dependent on the presentation.



http://en.wikipedia.org/wiki/File:Overview_of_a_three-tier_application_vectorVersion.svg

CRUD

- Most client-server-communication can be reduced on simple commands (+ payload)
- Frameworks like *Rails* or *DJango* make it extremely easy to build (even generate) a WebApp with
  - **Create / Read / Update / Delete** Operations on Resources

- E.g. Shop: **Create, Read, Update, (Delete)** of Resource “**Order**” implements Ordering Items

- Often used for web-based client-server applications

http://martinfowler.com/bliki/LayeringPrinciples.html

Active Record

- Architectual pattern
- Object as wrapper between business (program logic and data tier)
- An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data.
- f.e. concept in ruby on rails

http://martinfowler.com/eaaCatalog/activeRecord.html
https://en.wikipedia.org/wiki/Active_record_pattern

Info3

Introduction to GIT & Software Configuration Management

Alexander Kramer

Source Code Management

- Software Revision Control:
  - Means managing versions of source code

Software Configuration Management

- “SCM practices taken as a whole define how an organisation builds and releases products and identifies and tracks changes.”

  Stephen p. Berczuk, SCM Patterns, P. xxvii

https://de.wikipedia.org/wiki/Software-Configuration-Management

SCM - Main Features

- Document Changes & Authors
- Easily integrate source code from different Authors
- Undoing/Reverting Changes, short and long term
- Parallel Futures: Branches, Merging
- Sandboxing
- Version and Release Management

SCM – Basic Actions 1/2 (still local)

- **Add**: Put a file into the repo for the first time, i.e. begin tracking it with Version Control.
- **Revision**: What version a file is on (v1, v2, v3, etc.). - e.g. Subversion: The whole Repo is at a certain revision.
- **Head**: The latest revision in the repo.
- **Check out:** Download a file from the repo.
- **Check in / commit:** Upload a file to the repository (if it has changed). The file gets a new revision number, and people can “check out” the latest one.

SCM – Basic Actions 2/2 (still local)

- **Checkin Message / commit message:** A short message describing what was changed.
- **Changelog/History:** A list of changes made to a file since it was created.
- **Update/Sync:** Synchronize your files with the latest from the repository. This lets you grab the latest revisions of all files.
- **Revert:** Throw away your local changes and reload the latest version from the repository.

DVCS - New Terminology

- **push**: send a change to another repository
  (may require permission)
- **pull/fetch**: grab a change from a repository
  - Fetch is getting the remote structure without changing something local
  - Pull is getting the remote content, which can lead to merge(conflict)
- **clone**: clone a complete repository (init and pull in one sweep)

Sidestep: .gitignore

Exclude various files from consideration

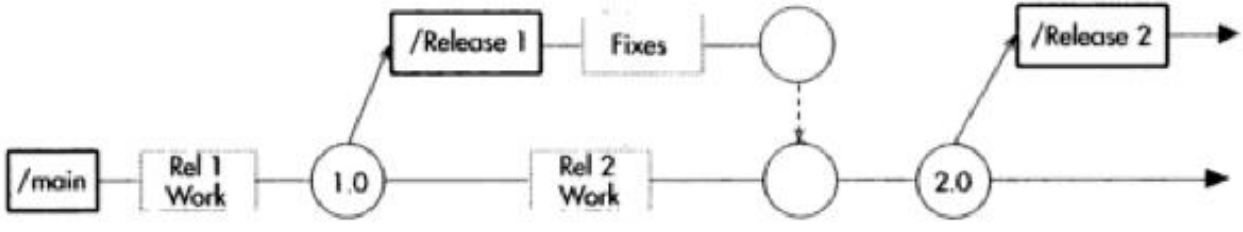
Edit the .gitignore file (ex.: Rails app)

```
.bundle
db/*.sqlite3*  //database files
 log/*.log     // log-files
 *.log
 /tmp/         // temporary files
 doc/          // documentation
*.swp          // swap-files
*~ .DS_Store   // OS-files (in this case MacOS)
```

https://github.com/github/gitignore

Release Prep Codeline

Release Line

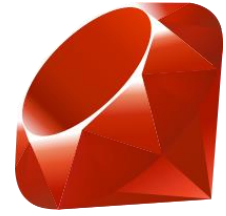


- Stabilize a codeline for an upcoming release while also enabling new work to continue
- Keep Release Line to be able to release fixes to that release

Introduction to Python

Info 3

Alexander Kramer



From Wikipedia:

https://en.wikipedia.org/wiki/Python_(programming_language)

> **Python** is a high-level, general-purpose programming language. Its design philosophy emphasizes code readability with the use of significant indentation via the off-side rule.[33]
>
> Python is dynamically typed and garbage-collected. It supports multiple programming paradigms, including structured (particularly procedural), object-oriented and functional programming. It is often described as a "batteries included" language due to its comprehensive standard library.[34][35]
>
> Guido van Rossum began working on Python in the late 1980s as a successor to the ABC programming language and first released it in 1991 as Python 0.9.0.[36] Python 2.0 was released in 2000. Python 3.0, released in 2008, was a major revision not completely backward-compatible with earlier versions. Python 2.7.18, released in 2020, was the last release of Python 2.[37]
>
> Python consistently ranks as one of the most popular programming languages.[38][39][40][41]

What is the Programming Model?

- Python: Object-Oriented Language - everything is an Object!
  - But: no Encapsulation (no private or protected)

- Objects are defined in Classes.
  - JavaScript: Prototype based Language.
  - Java: Strange Mixture - primitive & Object Types

- Supports Structured & Functional Programming as well.

What is the typing model?

Refresher: Static vs. Dynamic typing

- **Static** Typing
  - at Compile Time: Type is defined in the variable declaration (java)
- **Dynamic** Typing
  - at Run Time: the actual Type a variable refers to

What is the typing model?

Bruce Tate: Strong vs. Weak Typing

- **Strong** Typing
  - check the type of a variable; e.g. Java
- **Weak** Typing
  - everything is just a number, e.g. C - the type just tells the compiler how much memory to allocate

Python: Dynamic & Strong Typing

```
def add_something(sth):
  return 4 + sth


add_something("four")



// Throws an Error, because 4 is type of integer
// print(type(4)) -> <class ‘int’>
```

What are the core features that make the language unique?

- Indentation matters.
- Big amount of open source libraries available. PyPI · Der Python Package Index
- Good Documentation and Books available, widely used as first programming language.

Web Applications

Info 3

Alexander Kramer

HTTP – HyperText Transfer Protocol

- **1. Request-Response Model**: HTTP operates on a client-server model, where the client sends a request to the server, and the server responds with the requested resource or an error message. For instance, when you enter a URL in your browser, your browser makes an HTTP request to the server hosting the website, which then responds with the HTML page.

- **2. Stateless Protocol**: HTTP is stateless, meaning each request from a client to a server is treated as an independent transaction and does not retain any memory of previous requests. This characteristic makes the protocol simple and fast but requires additional mechanisms (like cookies or sessions) to maintain stateful information across multiple requests.

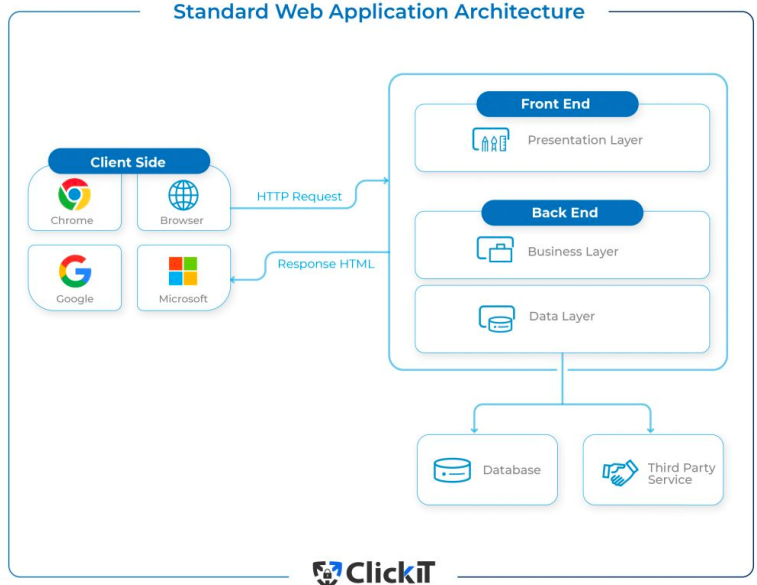
- **3. Methods**: HTTP defines several request methods that indicate the desired action to be performed on the resource. Some of the most common methods include: - `GET`: Requests a representation of a specified resource, often just retrieving data. - `POST`: Sends data to the server, commonly used for submitting form data or uploading files. - `PUT`: Replaces all current representations of the target resource with the uploaded content. - `DELETE`: Removes the specified resource. - `HEAD`: Similar to `GET`, but it only requests the headers and not the body.

- 4. **Status Codes:** HTTP responses include status codes that indicate the result of the request. Common status codes include: - `200 OK`: The request was successful. - `404 Not Found`: The requested resource could not be found. - `500 Internal Server Error`: The server encountered an unexpected condition.

- **5. Headers**: Both HTTP requests and responses can include headers, which are key-value pairs that provide additional information about the request or response. For example, headers can indicate the content type, content length, encoding, or cookies.

https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview

Web Application



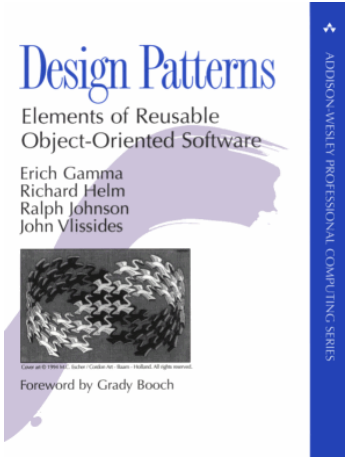
https://www.clickittech.com/devops/web-application-architecture/

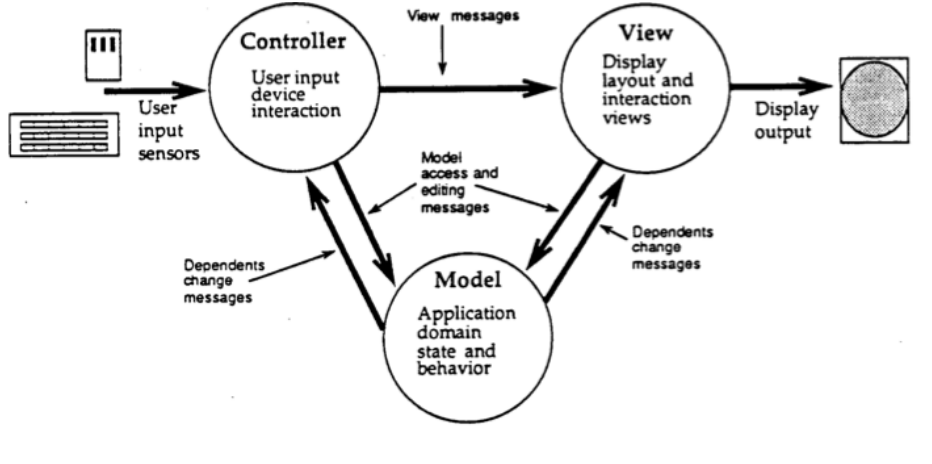
MVC

- **Model:** Data & Business Rules
- **View:** Generates User Interface
- **Controller:** Orchestrate the Application

Reminder: GoF: Patterns in MVC

- Observer, Composite, Strategy
  (GoF did not describe MVC as a pattern)





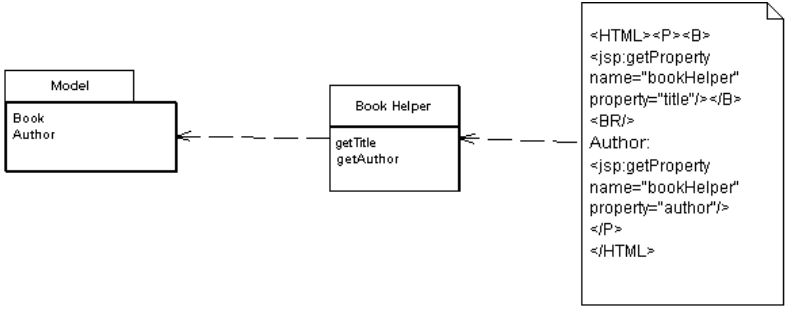
RESTful routes provided by the Users Resource

HTTP request	URL	Action	Purpose
GET	/users	index	page to list all users
GET	/users/1	show	page to show user with id 1
GET	/users/new	new	page to make a new user
POST	/users	create	create a new user
GET	/users/1/edit	edit	pate to edit user with id 1
PUT	/users/1	update	update user with id 1
DELETE	/users/1	destroy	delete user with id 1

Was ist eine mögliche Abfrage die man aus der Kombination von Request und URL erreichen will.

Template View

Renders information into HTML by embedding markers in an HTML page.




```
<div class="col-md-4">
  <h5>About the project:</h5>
  <p>{{ project.description }}</p>
  <br>
  <h5>Technology used:</h5>
  <p>{{ project.technology }}</p>

</div>
```

http://martinfowler.com/eaaCatalog/templateView.html

Django ORM

Django uses a different approach than the *Active Record* pattern, primarily utilizing an **Object-Relational Mapping (ORM)** system that emphasizes explicit field definitions for model relationships. While Active Record ties data access closely to the object, Django's ORM separates concerns, allowing for more flexibility and control over database interactions.

Understanding Active Record in Django

What is Active Record?

Active Record is a design pattern used in software engineering for managing data in relational databases. It allows an object to represent a row in a database table, with methods for inserting, updating, and deleting records. This pattern is commonly associated with frameworks like Ruby on Rails.

Django's Approach to Active Record

Django does not implement the Active Record pattern directly but follows a similar concept through its Object-Relational Mapping (ORM). In Django, models are defined as Python classes, and each model corresponds to a database table. Here's how Django's ORM relates to Active Record:

Model Definition: Each model class defines fields that correspond to database columns.

CRUD Operations: Django provides built-in methods for creating, reading, updating, and deleting records, similar to Active Record.

Explicit Relationships: Django requires explicit definitions for relationships between models, such as ForeignKey and ManyToManyField.

https://john-santosuosso.com/posts/django-vs-activerecord/

ORM

- Database definition in Django
- Access on the data through object methods

```
from django.db import models

class Project(models.Model):
    title = models.CharField(max_length=100)
    description = models.TextField()
    technology = models.CharField(max_length=20)
```

definition

```
def project_detail(request, pk):
    project = Project.objects.get(pk=pk)
    context = {
        "project": project
    }
    return render(request, "projects/project_detail.html", context)
```

retrieval

```
<h1>{{ project.title }}</h1>
```

usage

Intro to Software Engineering

Info 3

Alexander Kramer

What is Software Engineering?

Software engineering is concerned with *theories*, *methods* and *tools* for professional software development.

Software engineering is concerned with *cost-effective software development*.

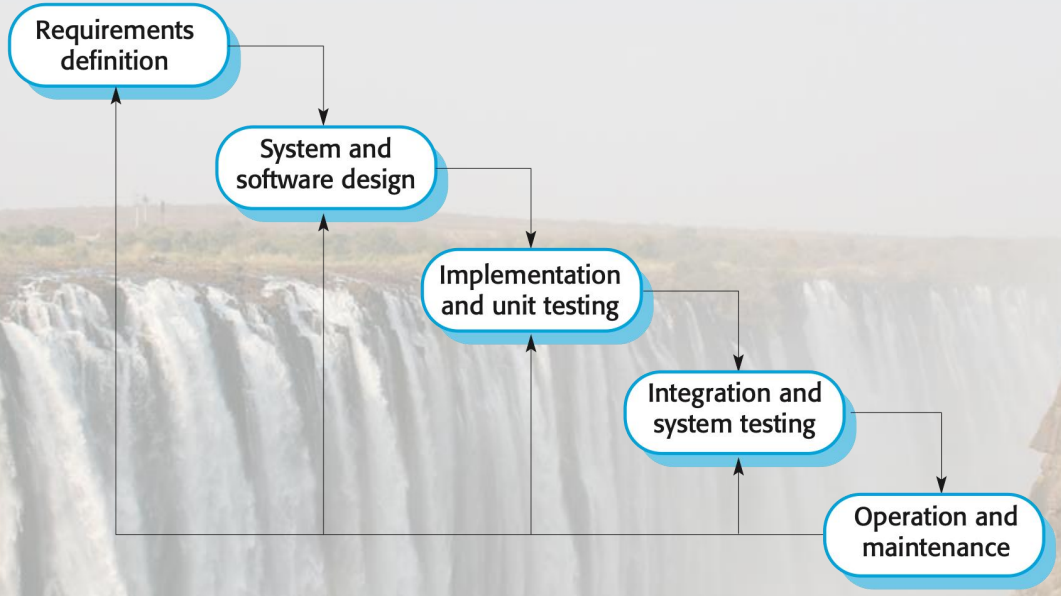
SE: Definition SWEBOK

The IEEE Computer Society defines software engineering as:

- (1) The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software

- (2) The study of approaches as in (1).

Waterfall Model of SE



http://www.flickr.com/photos/pjah73/8039270488/

Software process activities

- **Software specification**, where customers and engineers define the software that is to be produced and the constraints on its operation.
- **Software development**, where the software is designed and programmed.
- **Software validation**, where the software is checked to ensure that it is what the customer requires.
- **Software evolution**, where the software is modified to reflect changing customer and market requirements.

Application types

- **Stand-alone applications**
  - These are application systems that run on a local computer, such as a PC. They include all necessary functionality and do not need to be connected to a network.
- **Interactive transaction-based applications**
  - Applications that execute on a remote computer and are accessed by users from their own PCs or terminals. These include web applications such as e-commerce applications.
- **Embedded control systems**
  - These are software control systems that control and manage hardware devices. Numerically, there are probably more embedded systems than any other type of system.

Beispiele für diese Art von Softwares?

Application types

- **Batch processing systems**
  - These are business systems that are designed to process data in large batches. They process large numbers of individual inputs to create corresponding outputs.
- **Entertainment systems**
  - These are systems that are primarily for personal use and which are intended to entertain the user.
- **Systems for modeling and simulation**
  - These are systems that are developed by scientists and engineers to model physical processes or situations, which include many, separate, interacting objects.

Beispiele für diese Art von Softwares?

Application types

- **Data collection systems**
  - These are systems that collect data from their environment using a set of sensors and send that data to other systems for processing.
- **Systems of systems**
  - These are systems that are composed of a number of other software systems.

Beispiele für diese Art von Softwares?

Product Specification: Generic Products vs. Customized Products

- **Generic products**
  - The specification of what the software should do is owned by the software developer and decisions on software change are made by the developer. (or the organisation owning the software.)
- **Customized products**
  - The specification of what the software should do is owned by the customer for the software and they make decisions on software changes that are required.
- => Requirements Engineering will be completely different!

Essential attributes of good software

Product characteristic	Description
Maintainability	Software should be written in such a way so that it can evolve to meet the changing needs of customers. This is a critical attribute because software change is an inevitable requirement of a changing business environment.
Dependability and security	Software dependability includes a range of characteristics including reliability, security and safety. Dependable software should not cause physical or economic damage in the event of system failure. Malicious users should not be able to access or damage the system.
Efficiency	Software should not make wasteful use of system resources such as memory and processor cycles. Efficiency therefore includes responsiveness, processing time, memory utilization, etc.
Acceptability	Software must be acceptable to the type of users for which it is designed. This means that it must be understandable, usable and compatible with other systems that they use.

Frequently asked questions about software engineering 1/2

Question	Answer
What is software?	Computer programs and associated documentation.
What are the attributes of good software?	Good software should deliver the required functionality and performance to the user and should be maintainable, dependable and usable.
What is software engineering?	Software engineering is an **engineering discipline** that is concerned with all aspects of software production.
What are the fundamental software engineering activities?	Software **specification**, software **development**, software **validation** and software **evolution**.
What is the difference between software engineering and computer science?	Computer science focuses on theory and fundamentals; software engineering is concerned with the **practicalities of developing and delivering useful software.**
What is the difference between software engineering and system engineering?	**System engineering** is concerned with all aspects of computer-based systems development **including hardware,** software and process engineering. Software engineering is part of this more general process.

Frequently asked questions about software engineering 2/2

Question	Answer
What are the key challenges facing software engineering?	Coping with increasing diversity, demands for reduced delivery times and developing trustworthy software.
What are the costs of software engineering?	Roughly 60% of software costs are development costs, 40% are testing costs. For custom software, evolution costs often exceed development costs.
What are the best software engineering techniques and methods?	Depends on the Type of Software: e.g. Games should be created using a series of prototypes, safety critical control systems require a complete and analyzable specification to be developed.
What differences has the web made to software engineering?	The web has led to the availability of software services and the possibility of developing highly distributed service-based systems.

Software Processes

Info 3

Alexander Kramer

Topics covered

- Software process models
- Process activities
- Coping with change
- The Rational Unified Process
  - An example of a modern software process.

Capter 1 – Software Process Models

The software process

- A structured set of ***activities*** required to develop a software system.

- Many different software processes but all involve:
  - **Specification** – defining what the system should do;
  - **Design and implementation** – defining the organization of the system and implementing the system;
  - **Validation** – checking that it does what the customer wants;
  - **Evolution** – changing the system in response to changing customer needs.

- A *software process model* is an abstract representation of a process. It presents a description of a process from some particular perspective.

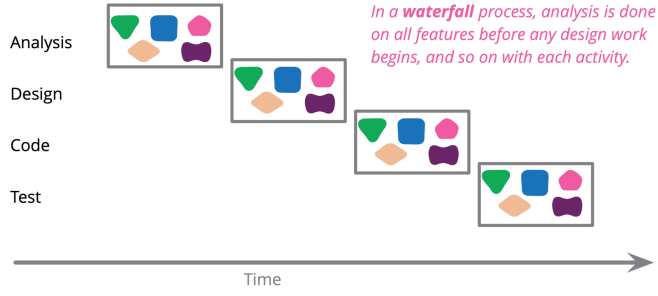
Software process models

- **The Waterfall model**
  - Break down by Activities.
  - Plan-driven model. Separate and distinct phases of specification and development.

- **Incremental development**
  - Break down by Functionality.
  - Specification, development and validation are interleaved. May be plan-driven or agile.
  - Agile: respond to changes, planning is also incremental

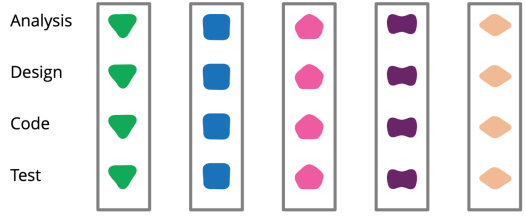
However: The *Activities* are also Part of Agile Approaches.

all features at once



*In a **waterfall** process, analysis is done on all features before any design work begins, and so on with each activity.*

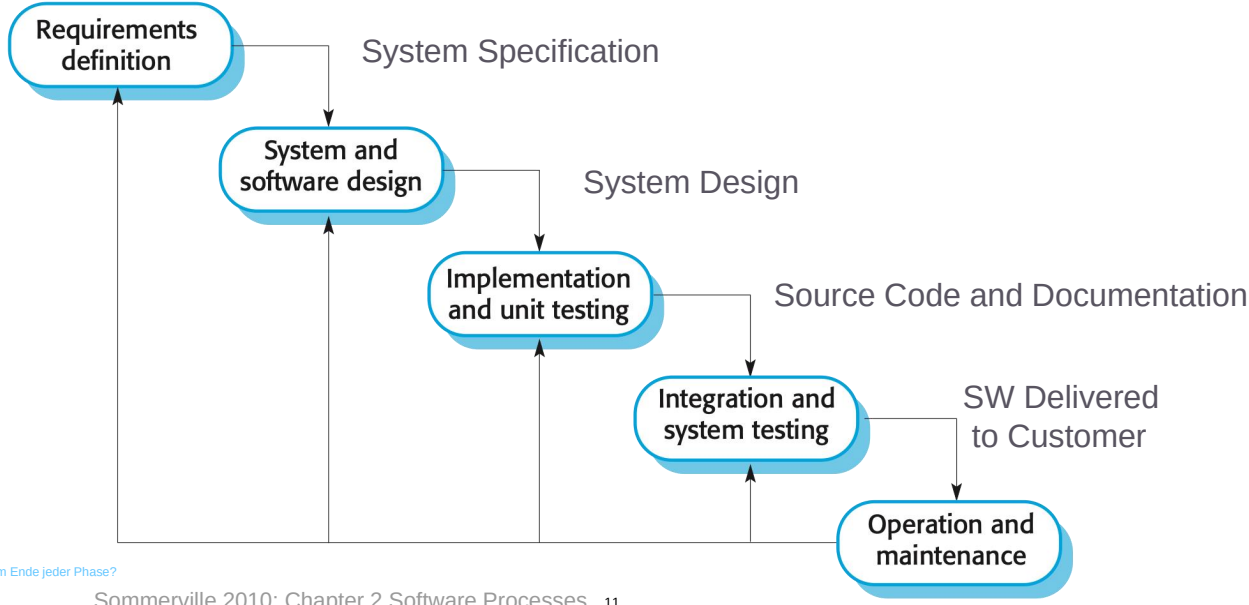
one feature at a time



*In an **iterative** process, all activities are done for one feature before work begins on another feature.*

Waterfall Development

The waterfall model



Was steht am Ende jeder Phase?

Waterfall model problems

- Inflexible partitioning of the project into distinct stages makes it difficult to respond to changing customer requirements.
  - Therefore, this model is only appropriate when the requirements are well-understood and changes will be fairly limited during the design process.
  - Few business systems have stable requirements.

- The waterfall model is mostly used for large systems engineering projects where a system is developed at several sites.
  - In those circumstances, the plan-driven nature of the waterfall model helps coordinate the work.

- The waterfall model works best if requirements are well understood.

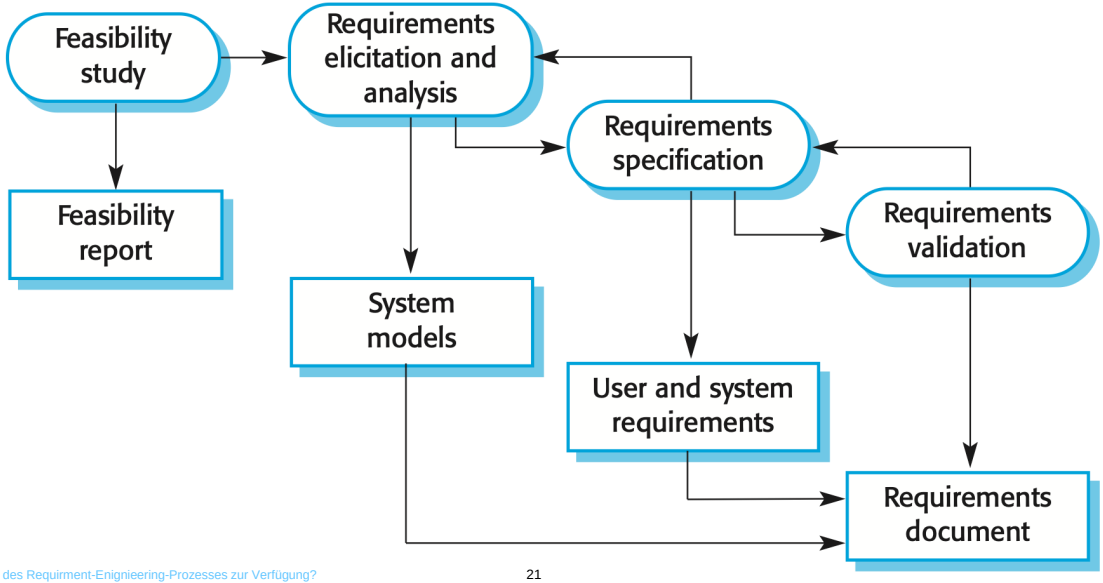
Incremental Development

Incremental development problems

- The process is not visible.
  - Managers need regular deliverables to measure progress. If systems are developed quickly, it is not cost-effective to produce documents that reflect every version of the system.
- System structure tends to degrade as new increments are added.
  - Unless time and money is spent on refactoring to improve the software, regular change tends to corrupt its structure. Incorporating further software changes becomes increasingly difficult and costly.
- Increments cannot always be delivered + deployed to a production environment.
  - can disrupt business processes

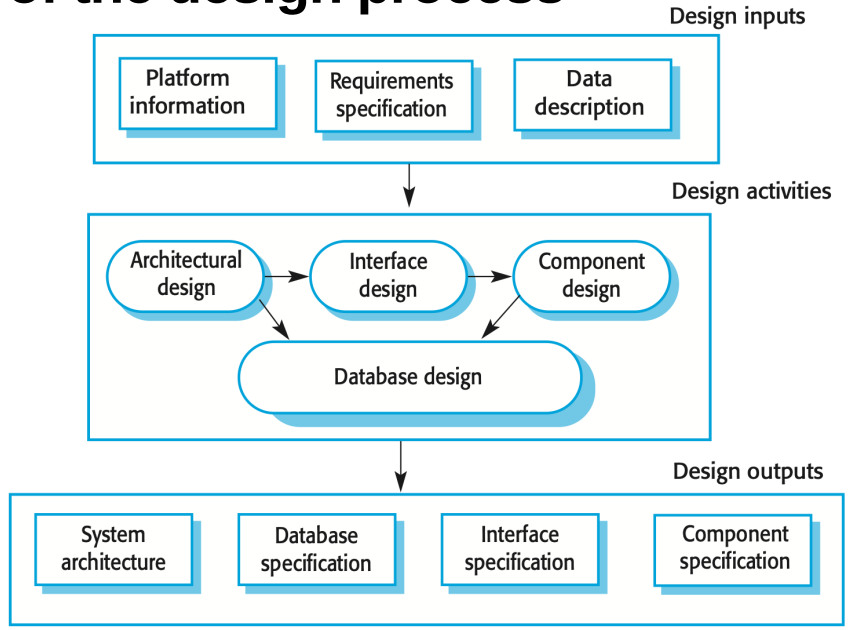
Software Specification

Software Specification: The requirements engineering process



Design and Implementation

Design and Implementation: A general model of the design process



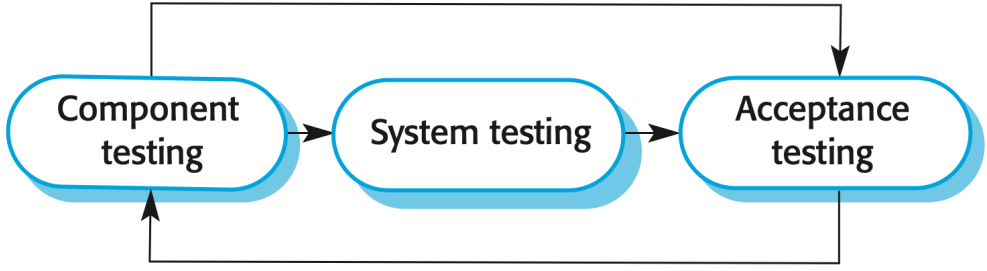
Was steht am Ende eines Design Prozesses zur Verfügung?

Software validation

Software validation

- **Verification**: does system conform to specification?
- **Validation**: does it meet the requirements?

- Stages of Testing:



Was wird in welcher Stage getestet? Beispiel?

Key points

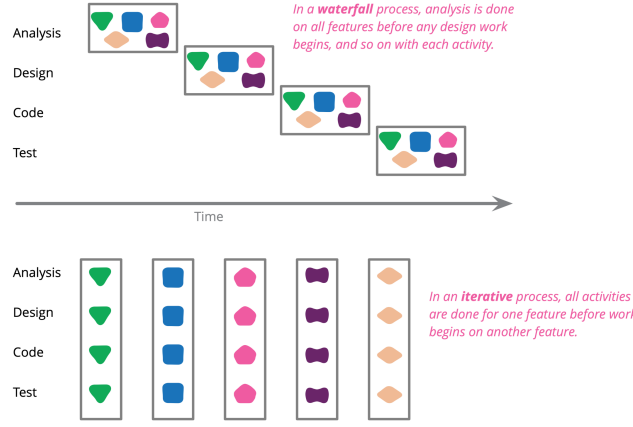
- **Requirements engineering** is the process of developing a software specification.
- **Design and implementation processes** are concerned with transforming a requirements specification into an executable software system.
- **Software validation** is the process of checking that the system conforms to its specification and that it meets the real needs of the users of the system.
- **Software evolution** takes place when you change existing software systems to meet new requirements. The software must evolve to remain useful.

Chapter 2 – Software Processes

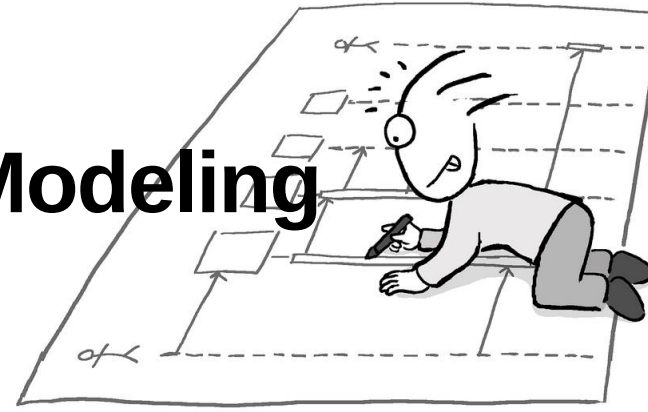
Part 2

Conclusion

- There are many different Process Models for Software Development
- In Practice, you'll find them in even more Variations
- The Mayor Activities - *Analysis*, *Design*, *Code*, *Test* - are needed in all of them - including Agile Approaches.



Info3: System Modeling



'진정한 개발자'는 코드를 작성한다

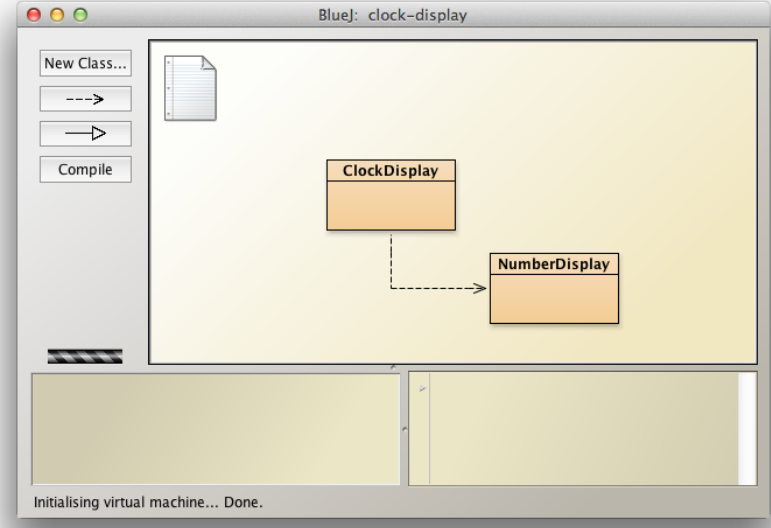
Alexander Kramer

http://www.flickr.com/photos/joone/3050331298

Topics Covered

- What is System Modelling?
- The UML
- Different Views of the System

System Modeling: Abstraction



System Modeling is a abstract representation of different aspects and views of a system, existing or planned.

Number display is the more abstract representation, while ClockDisplay is the more detailed, refined representation.

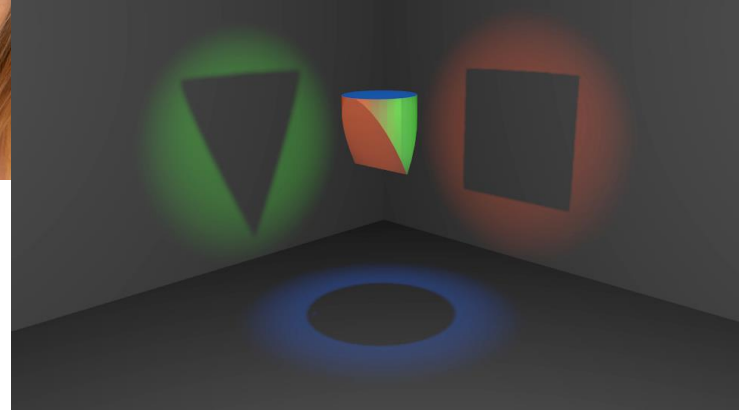
Unified Modeling Language

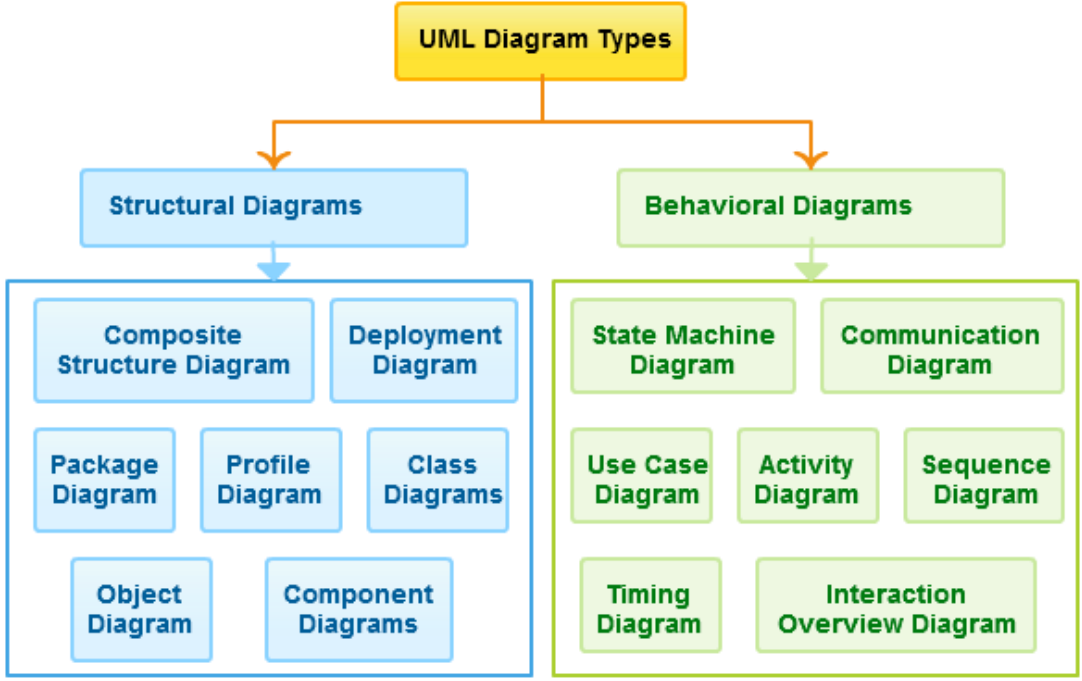
- “The *Unified Modeling Language* is a visual language for specifying, constructing, and documenting the artifacts of systems.
- It is a general-purpose modeling language that can be used with all major object and component methods, and that can be applied to all application domains”

https://de.wikipedia.org/wiki/Unified_Modeling_Language

What is shown in the UML?

Different Perspectives





http://creately.com/blog/diagrams/uml-diagram-types-examples/

How is the UML used?

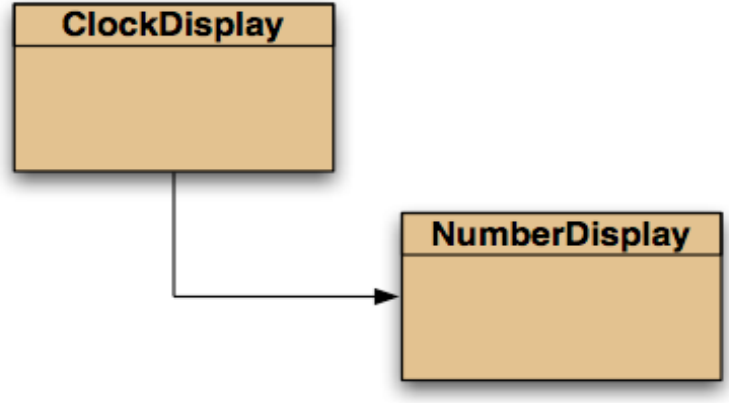
Sommerville: Use of graphical models

- As a means of facilitating discussion about an existing or proposed system
  - Incomplete and incorrect models are OK as their role is to support discussion.
- As a way of documenting an existing system
  - Models should be an accurate representation of the system but need not be complete.
- Helps to understand legacy code!
- As a detailed system description that can be used to generate a system implementation
  - Models have to be both correct and complete.

UML Diagrams

Structural Diagrams

Class diagram

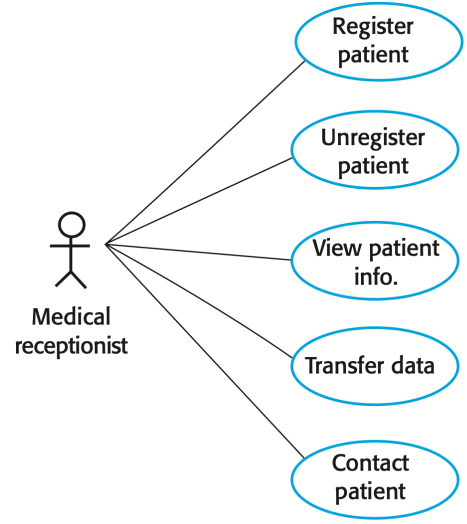


- **Classes**: Declared in Source Code.
  Static - independent of program execution

Behavioral Models / Diagrams

Use Case Diagrams

Use Cases – Example in the MHC-PMS involving the role ‘Medical Receptionist’



What persona should be able to do / has to do what in terms of a *system?*

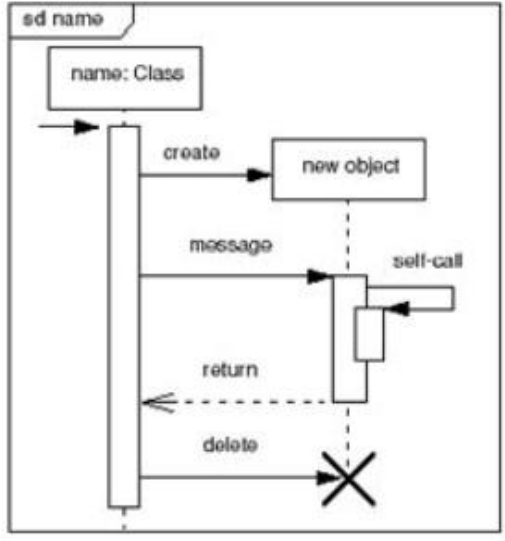
Sequence Diagram

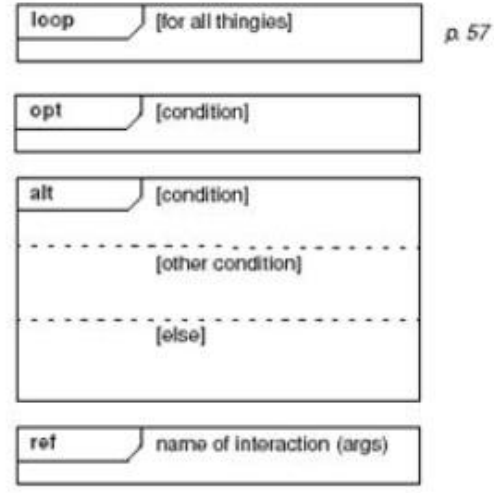
Sequence diagrams (Sommerville)

- *Sequence diagrams* are part of the UML and are used to model the interactions between the actors and the objects within a system.
- A *sequence diagram* shows the sequence of interactions that take place during a particular use case or use case instance.
- The objects and actors involved are listed along the top of the diagram, with a dotted line drawn vertically from these.
- Interactions between objects are indicated by annotated arrows.

Sequence Diagram - Variants

Sequence Diagram *p. 53*





p. 57

Wie kann ein gegebenes Sequenz Diagram interpretiert warden?

Fowler, M. (2003) *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. 3rd edition. Boston, Mass.: Addison-Wesley Professional.
Available at: https://martinfowler.com/books/uml.html.

Structural models

Structural models (Summerville)

- *Structural models* of software display the organization of a system in terms of the components that make up that system and their relationships.
- *Structural models* may be static models, which show the structure of the system design, or dynamic models, which show the organization of the system when it is executing.
- You create structural models of a system when you are discussing and designing the system architecture.

Class Diagrams

Class diagrams

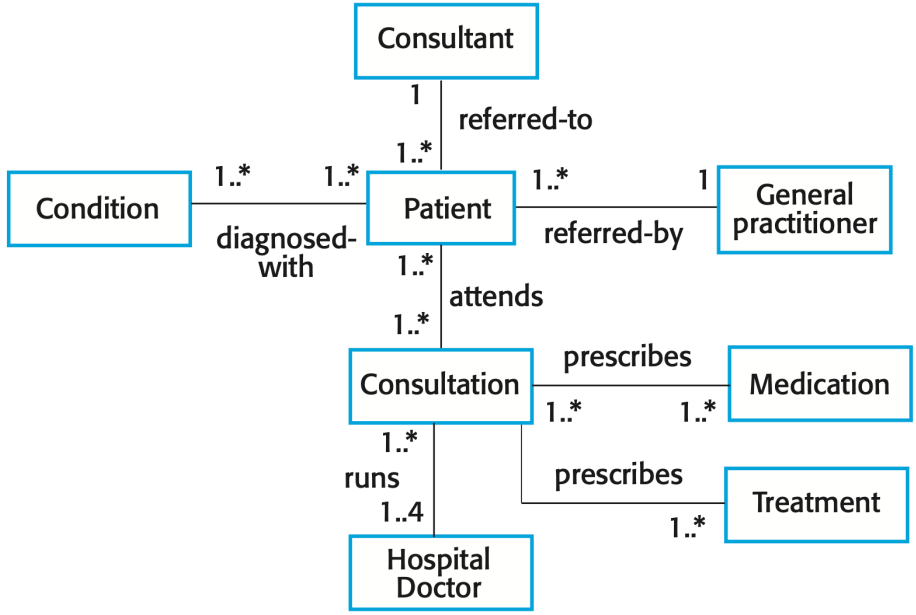
- *Class diagrams* are used when developing an object-oriented system model to show the classes in a system and the associations between these classes.

- An object class can be thought of as a general definition of one kind of system object.

- An association is a link between classes that indicates that there is some relationship between these classes.

- When you are developing models during the early stages of the software engineering process, objects represent something in the real world, such as a patient, a prescription, doctor, etc.

Classes and associations in the MHC-PMS



Wie können die verschiedenen Arten von Beziehungen gelesen und verbalisiert warden?

Key points

- A model is an abstract view of a system that ignores system details. Complementary system models can be developed to show the system's context, interactions, structure and behavior.
- Context models show how a system that is being modeled is positioned in an environment with other systems and processes.
- Use case diagrams and sequence diagrams are used to describe the interactions between users and systems in the system being designed. Use cases describe interactions between a system and external actors; sequence diagrams add more information to these by showing interactions between system objects.
- Structural models show the organization and architecture of a system. Class diagrams are used to define the static structure of classes in a system and their associations.

System Modeling (a broader view)

Generalization

- In modeling systems, it is often useful to examine the classes in a system to see if there is scope for generalization. If changes are proposed, then you do not have to look at all classes in the system to see if they are affected by the change.

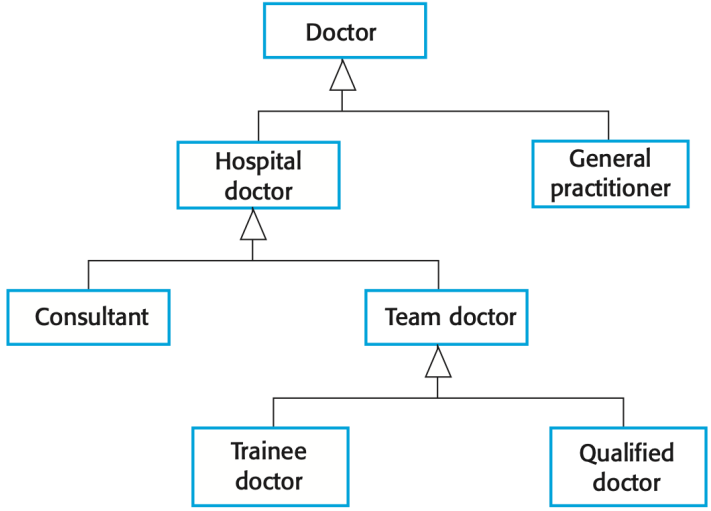
- In object-oriented languages, such as Java, generalization is implemented using the class inheritance mechanisms built into the language.

- In a generalization, the attributes and operations associated with higher-level classes are also associated with the lower-level classes.

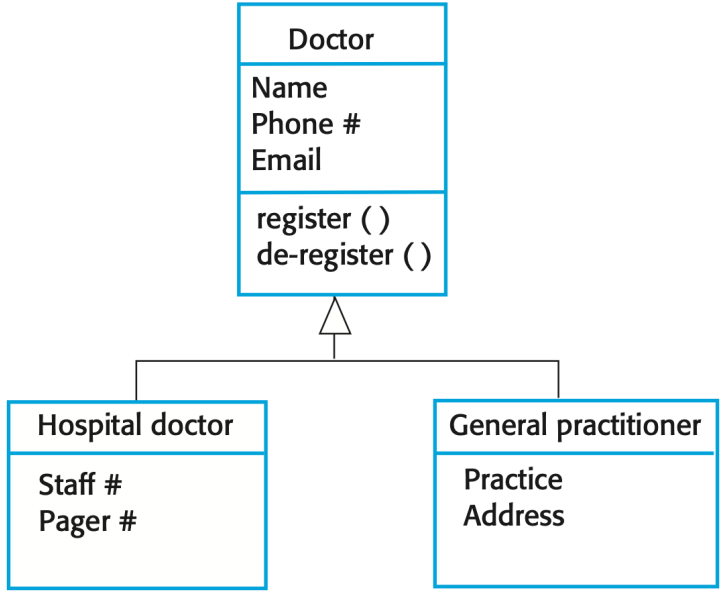
- The lower-level classes are subclasses inherit the attributes and operations from their superclasses. These lower-level classes then add more specific attributes and operations.

Wird in der realen Umsetzung über Vererbung realisiert.

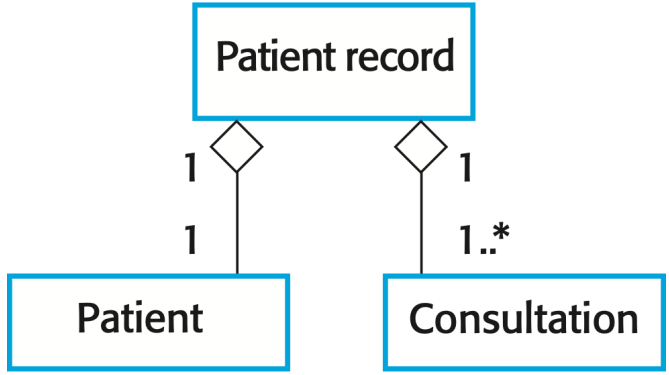
Class Diagrams show Inheritance / Generalization Hierarchies



A generalization hierarchy with added detail



And aggregation associations...



Aggregation & Composition

Figure 5.3. Aggregation

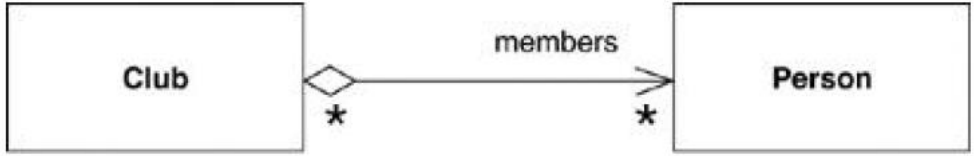
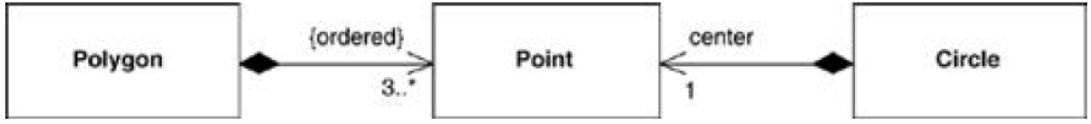


Figure 5.4. Composition



Aggregation more loose, exist outside of association
Composition: does not exist without association or at least doesn't make sense

Behavioral models

Behavioral models

- *Behavioral models* are models of the dynamic behavior of a system as it is executing. They show what happens or what is supposed to happen when a system responds to a stimulus from its environment.

- You can think of these stimuli as being of two types:
  - **Data -** Some data arrives that has to be processed by the system.
  - **Events** - Some event happens that triggers system processing. Events may have associated data, although this is not always the case.

Behavioral models

Data-driven modeling

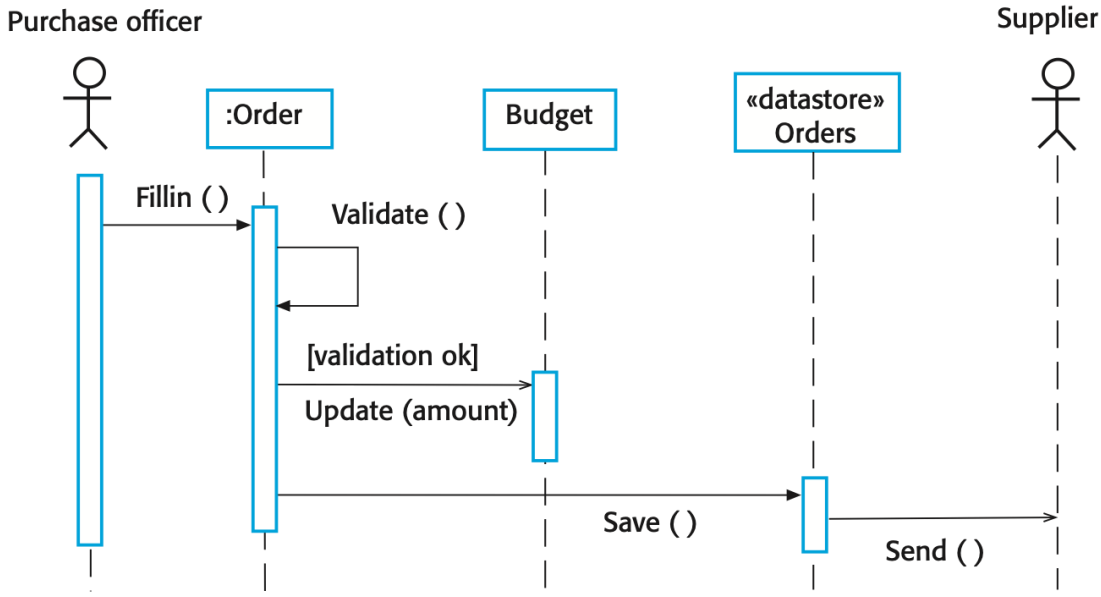
- Many business systems are data-processing systems that are primarily driven by data. They are controlled by the data input to the system, with relatively little external event processing.

- Data-driven models show the sequence of actions involved in processing input data and generating an associated output.

- They are particularly useful during the analysis of requirements as they can be used to show end-to-end processing in a system.

Was wäre die verbale Skizze eines daten-getriebenen Modelles (was keine Insulinpumpe ist)?

Sequence Diagram - Ex: Order processing



Verbalisierung davon?

Behavioral models

Event-driven modeling

- Real-time systems are often event-driven, with minimal data processing. For example, a landline phone switching system responds to events such as ‘receiver off hook’ by generating a dial tone.

- Event-driven modeling shows how a system responds to external and internal events.

- It is based on the assumption that a system has a finite number of states and that events (stimuli) may cause a transition from one state to another.

Was wäre die verbale Skizze eines event-getriebenen Modelles (was keine Microwelle st)?

Other Diagrams

Context models

Process perspective

Context Model

And Additionally: Context models

- *Context models* are used to illustrate the operational context of a system - they show what lies outside the system boundaries.
- Social and organisational concerns may affect the decision on where to position system boundaries.
- *Architectural models* show the system and its relationship with other systems.

Process Model

Process Models / Perspective

- *Context models* simply show the other systems in the environment, not how the system being developed is used in that environment.
- *Process models* reveal how the system being developed is used in broader business processes.
- UML activity diagrams may be used to define business process models.

Interaction Model

Interaction models

- *Modeling user interaction* is important as it helps to identify user requirements.
- *Modeling system-to-system interaction* highlights the communication problems that may arise.
- *Modeling component interaction* helps us understand if a proposed system structure is likely to deliver the required system performance and dependability.
- **Use case diagrams** and **sequence diagrams** may be used for *interaction modeling*.

Key points

- *Behavioral models* are used to describe the dynamic behavior of an executing system. This behavior can be modelled from the perspective of the data processed by the system, or by the events that stimulate responses from a system.

- *Activity diagrams* may be used to model the processing of data, where each activity represents one process step.

- *State diagrams* are used to model a *system's behavior* in response to internal or external events.

An Introduction to ~~Scrum~~ Agile Software Development, mostly Scrum

Presentation based on Scrum Intro by Mike Cohn

Alexander Kramer

Incremental / Iterative / Agile?

- Waterfall / Big Design Upfront (BUP):
  neither incremental or iterative

- Incremental: Split into features that are built one by one, but plan everything up-front
- Iterative: feedback cycles in the process
- Agile: Incremental + Iterative,
  adheres to agile values.

Scrum

Scrum: Characteristics

- Self-organizing teams
- Product progresses in a series of fixed-length “**sprints**”
- Requirements are captured as items in a list of “**product backlog**” => do one thing at a time
- No specific engineering practices prescribed
- Uses generative rules to create an agile environment for delivering projects
- One of the “**agile processes**”

- Transparency and Retrospectives

Scrum: Rhythm

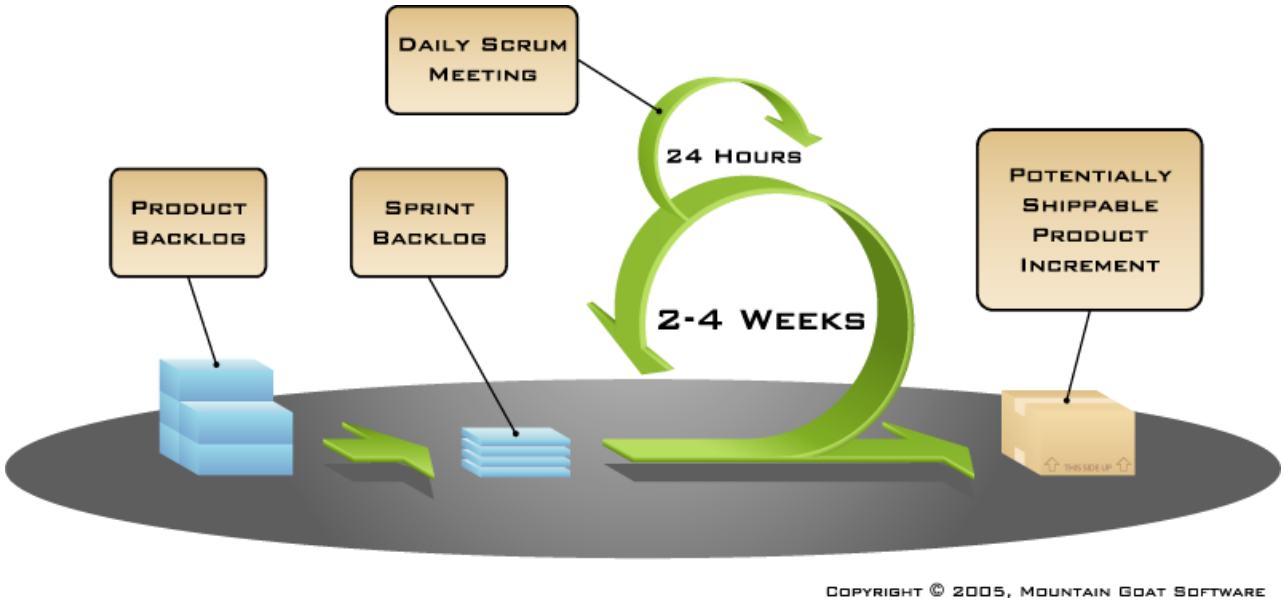


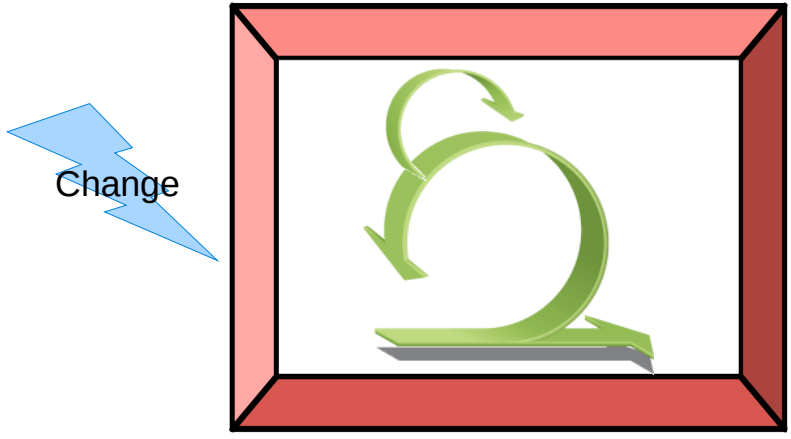
Image available at www.mountaingoatsoftware.com/scrum

Scrum: Sprints

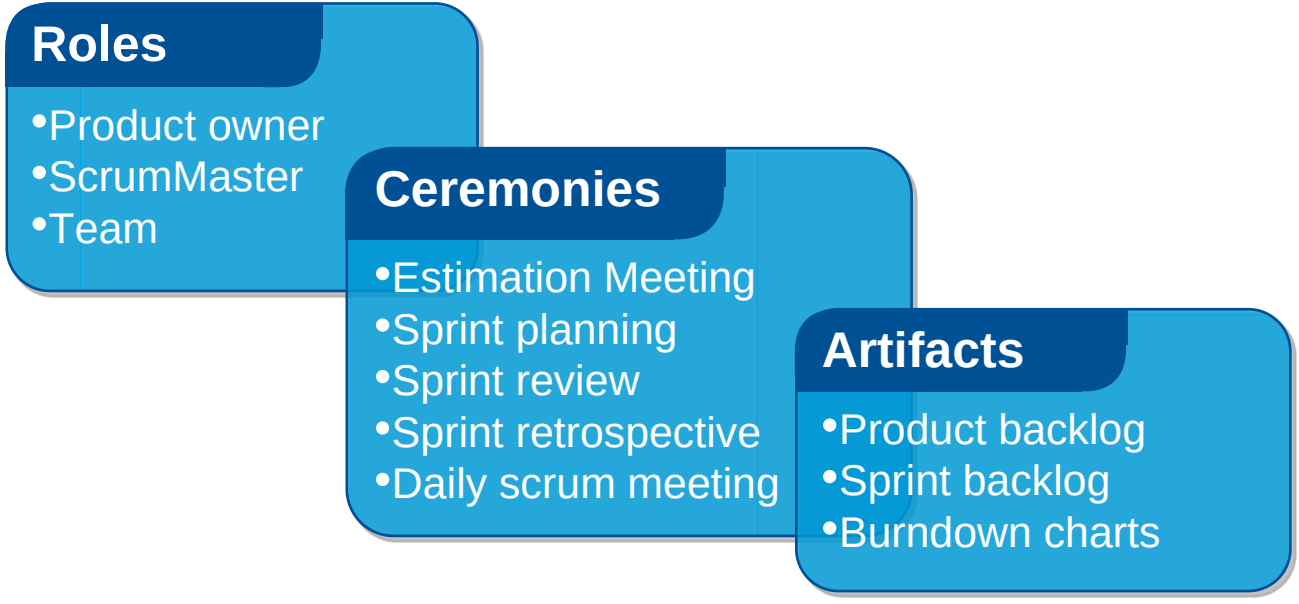
- Scrum projects make progress in a series of “**sprints**”
  - Analogous to *Extreme Programming iterations*
- Typical duration is 2–4 weeks or a calendar month at most
- A constant duration leads to a better rhythm
- Product is *designed*, *coded*, and *tested* during the sprint

No changes during a sprint

- Plan sprint durations around how long you can commit to keeping change out of the sprint



Scrum Framework



Scrum: Artefacts

Product backlog

- The requirements
- A list of all desired work on the project
- Ideally expressed such that each item has value to the users or customers of the product
- Prioritised by the *product owner*
- Reprioritised at the start of each sprint

This is the product backlog

A sample product backlog

Backlog item	Estimate
Allow a guest to make a reservation	3
As a guest, I want to cancel a reservation.	5
As a guest, I want to change the dates of a reservation.	3
As a hotel employee, I can run RevPAR reports (revenue-per-available-room)	8
Improve exception handling	8
...	30
...	50

INVEST Model

User Stories should follow the INVEST model:

- Independent
- Negotiable
- Valuable
- Estimatable
- Small
- Testable

Was bedeuten die einzelnen Punkte?

Make the User Stories testable

Scenario 1: Title

Given [context]

and [some more context]

When [event]

Then [outcome]

and [some more outcome]

Scenario 1: Account balance is negative

Given the account's balance is below 0, *when* the account owner attempts to withdraw money, *then* the bank will deny it

source: http://www.codesqueeze.com/the-easy-way-to-writing-good-user-stories

The sprint goal

- A short statement of what the work will be focused on during the sprint

Life Sciences

Support features necessary for population genetics studies.

Database Application

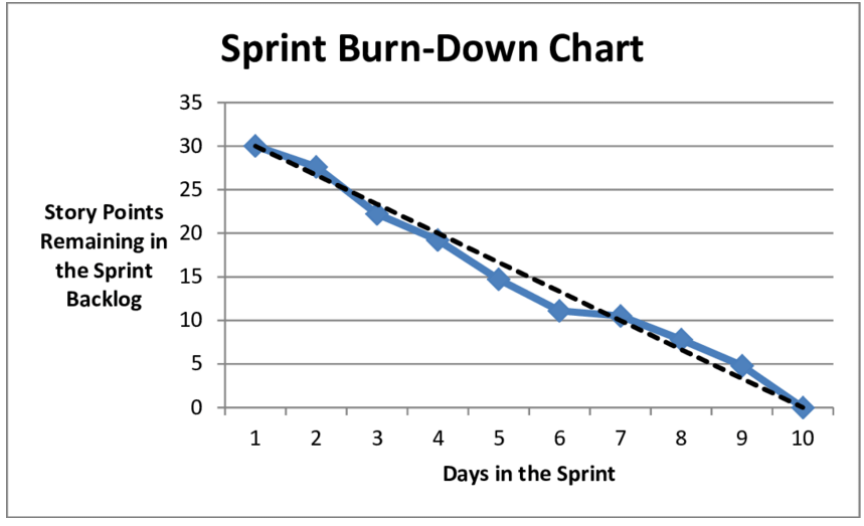
Make the application run on SQL Server in addition to Oracle.

Financial services

Support more technical indicators than company ABC with real-time, streaming data.

Burn-Down Chart

- Shows the remaining Ideal Story Points vs. Actual Story Points over Time



Scalability

- Typical individual team is 7 ± 2 people
  - Scalability comes from teams of teams
- Factors in scaling
  - Type of application
  - Team size
  - Team dispersion
  - Project duration
- Scrum has been used on multiple 500+ person projects

Scrum – The Roles

Scrum Framework

Roles

- Product owner
- ScrumMaster
- Team

Ceremonies

- Estimation Meeting
- Sprint planning
- Sprint review
- Sprint retrospective
- Daily scrum meeting

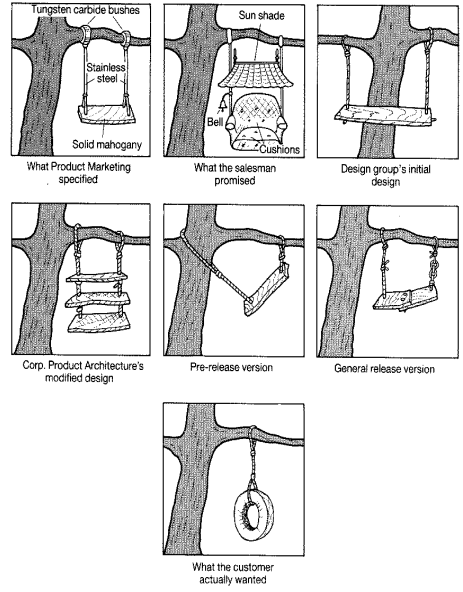
Artifacts

- Product backlog
- Sprint backlog
- Burndown charts

Welche Aufgaben haben die einzelnen Rollen?

Requirements Analysis

Info 3



Requirements engineering

- Is the **process** of establishing the *services* that the customer requires from a system and the *constraints* under which it operates and is developed.

- The **requirements** themselves are the descriptions of the *system services* and *constraints* that are generated during the *requirements engineering* process.

Types of requirement

- **User requirements (Lastenheft)**
  - Statements in natural language plus diagrams of the services the system provides and its operational constraints.
  - May be the basis for a bid for a contract.

- **System requirements (Pflichtenheft)**
  - A structured document setting out detailed descriptions of the system's functions, services and operational constraints.
  - Defines what should be implemented so may be part of a contract between client and contractor.

Functional and non-functional requirements

- **Functional requirements**
  - Statements of services the system should provide, how the system should react to particular inputs and how the system should behave in particular situations.
  - May state what the system should not do.

- **Non-functional requirements**
  - Constraints on the services or functions offered by the system such as timing constraints, constraints on the development process, standards, etc.
  - Often apply to the system as a whole rather than individual features or services.

- **(Domain requirements)**
  - Constraints on the system from the domain of operation

Beispiele dafür?

Functional requirements for the MHC-PMS

- A user shall be able to search the appointments lists for all clinics.
- The system shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day.
- Each staff member using the system shall be uniquely identified by his or her 8-digit employee number.

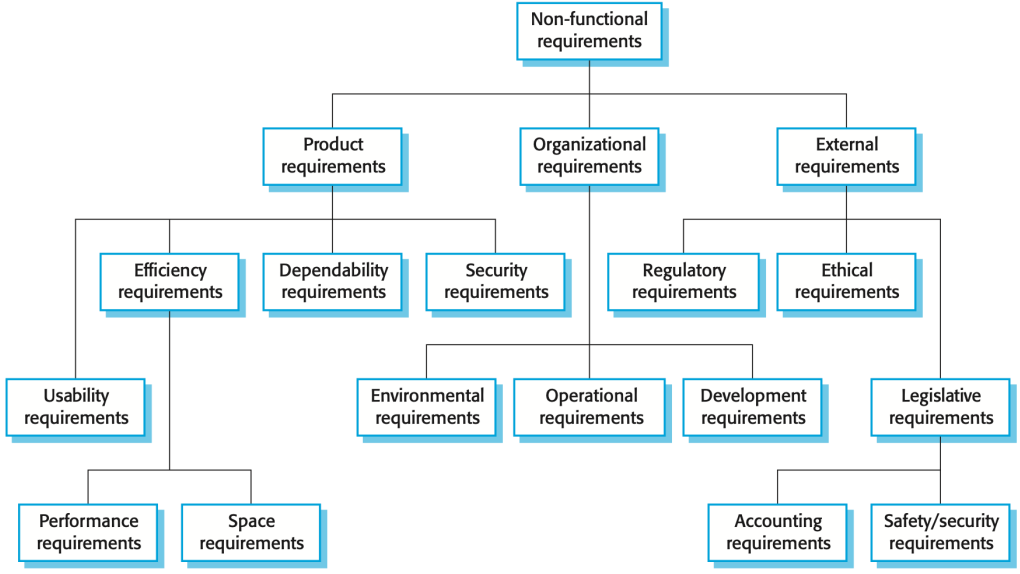
Examples of non-functional requirements in the MHC-PMS

Product requirement
The MHC-PMS shall be available to all clinics during normal working hours (Mon–Fri, 08.30–17.30). Downtime within normal working hours shall not exceed five seconds in any one day.

Organizational requirement
Users of the MHC-PMS system shall authenticate themselves using their health authority identity card.

External requirement
The system shall implement patient privacy provisions as set out in HStan-03-2006-priv.

Classification of non-functional requirements



Erklärung von min 2 davon mit Beispiel?

Metrics for specifying nonfunctional requirements

Property	Measure
Speed	Processed transactions/second User/event response time Screen refresh time
Size	Mbytes Number of ROM chips
Ease of use	Training time Number of help frames
Reliability	Mean time to failure Probability of unavailability Rate of failure occurrence Availability
Robustness	Time to restart after failure Percentage of events causing failure Probability of data corruption on failure
Portability	Percentage of target dependent statements Number of target systems

Was wäre ein Beispiel für eine Metrik einer nicht-funktionalen Anforderung?

Requirements completeness and consistency

In principle, requirements should be both *complete and consistent*.

Complete

- They should include descriptions of all facilities required.

Consistent

- There should be no conflicts or contradictions in the descriptions of the system facilities.

But: In practice, it is impossible to produce a complete and consistent requirements document.

- vs. Agile Manifesto: “*Customer collaboration over contract negotiation*”

How to find Requirements?

- Who are the Stakeholders (Interessensgruppen)?

- Two Approaches:
  - Interviews
  - **Ethnography**
    - Observation
    - Interviews
    - Surveys

Interviews in practice

- Types of interview
  - **Closed interviews** based on pre-determined list of questions
  - **Open interviews** where various issues are explored with stakeholders.

- Normally a mix of closed and open-ended interviewing.

- Interviews are good for getting an overall understanding of what stakeholders do and how they might interact with the system.
  - Effective interviewing
    - Be open-minded, avoid pre-conceived ideas about the requirements and are willing to listen to stakeholders.
    - Prompt the interviewee to get discussions going using a springboard question, a requirements proposal, or by working together on a prototype system.

Ethnography

- A social scientist spends a considerable time observing and analysing how people actually work.
- People do not have to explain or articulate their work.
- Social and organisational factors of importance may be observed.
- Ethnographic studies have shown that work is usually richer and more complex than suggested by simple system models.

Scope of ethnography

- **Requirements** that are derived from the way that people **actually work** rather than the way in which process definitions suggest that they ought to work.

- Requirements that are derived from **cooperation** and awareness of **other people's activities**.
  - Awareness of what other people are doing leads to changes in the ways in which we do things.

- Ethnography is effective for understanding **existing processes** but cannot identify new features that should be added to a system.

Requirements checking

- **Validity -** Does the system provide the functions which best support the customer's needs?
- **Consistency -** Are there any requirements conflicts?
- **Completeness -** Are all functions required by the customer included?
- **Realism** - Can the requirements be implemented given available budget and technology?
- **Verifiability -** Can the requirements be checked?

Requirements validation techniques

- **Requirements reviews**
  - Systematic manual analysis of the requirements.
- **Prototyping**
  - Using an executable model of the system to check requirements.
- **Test-case generation**
  - Developing tests for requirements to check testability.

Design Patterns

Alexander Kramer

Observer Pattern

- The observer pattern is a software design pattern in which an object, called the subject, **maintains a list** of its dependents, **called observers**, and **notifies them automatically** of any state changes, usually by **calling one of their methods.**

Composite

- Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Welcher Zusammenhang besteht zwischen Composite Pattern und Django?

https://www.youtube.com/watch?v=ZCNQ7xsed58

Info3 POEAA

Pattern of Enterprise Application Architecture

Alexander Kramer

Wie kann man POEAA erklären?

CQRS (Command Query Responsibility Segregation)

- This pattern separates data reading **(Query)** from data writing **(Command)**.
- This allows for better scalability and performance, as each model can be optimized specifically for its task.

Welchen Vorteil hat die Trennung von einem System für data reading und einem System für data writing?

OOD – Object Oriented Design

Info 3

Alexander Kramer

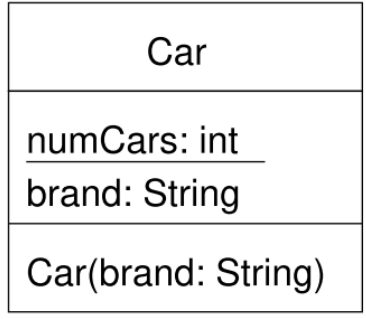
Example of Bad Design

```
class Car {
   private AirConditon airCondition ;
   private int getTempFromSensor ()
   {
     // ...
   }
   int getTemperature ()
   {
      int t = getTempFromSensor ();
      if (t > 30)
      {
          airCondition.setOn(true);
     }
     return t;
   }
}
```

- Principle of Single Responsibility: A unit of code (class, method, etc.) should be only responsible for one part of the program's functionality
- Alternative: there should be only one possible reason to modify the unit
- *"Do one thing, do it well"* (Ken Thompson)

Static Attributes - UML

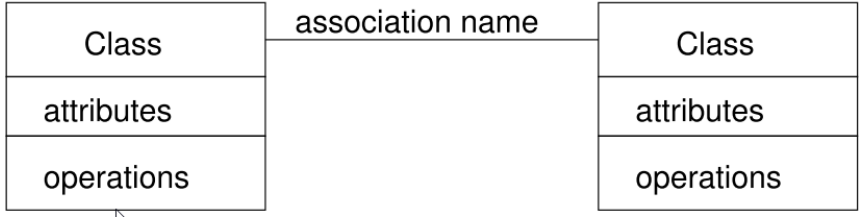
In UML , static attributes are underlinded



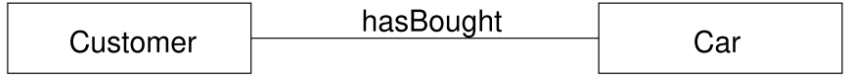
Associations (1)

An association represents a **hasA** relation between objects of two, usually different classes.

General form:



Example:



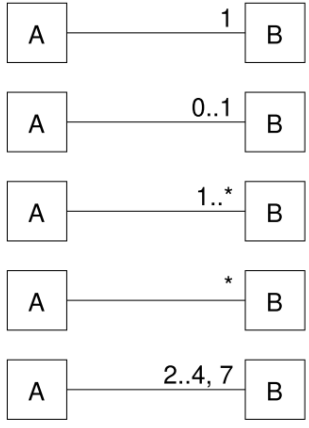
Associations (2)

- Associations are another way to represent attributes, especially when the type of the attribute is a class.
- The following two ways of modeling the relation between Customer and Car have the same meaning, if we ignore the association name.

Multiplicities

- Represents the number of objects involved in the relation.

Every object of A has



exactly one object of B

zero or one object of B

one or more objects of B

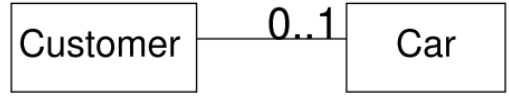
any number (maybe zero) of objects of B

2 to 4 or 7 objects of B

Multiplicities - Examples

- Represents the number of objects involved in the relation.

```
class Customer {
  Car hasBought;
}
```




```
class Customer {
  List<Car> hasBought;
}
```

Aggregation

Special form of association. An aggregation describes a ***whole-part*** relation.

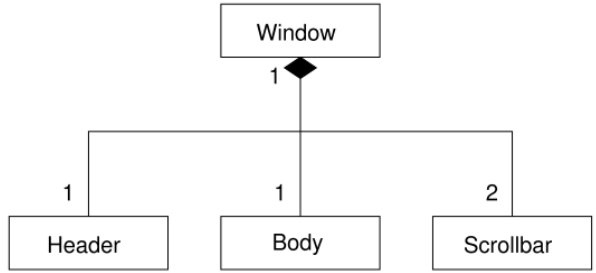
Example:

- An exam has 10 questions.
- A university has several faculties.

Composition

Strong aggregation: existence of the part is dependant of the whole.

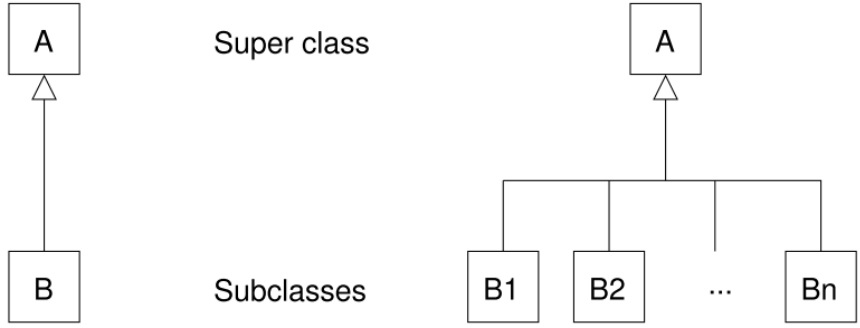
Example:



Unterschied zwischen Aggreagation und composition? Mit Beispiel?

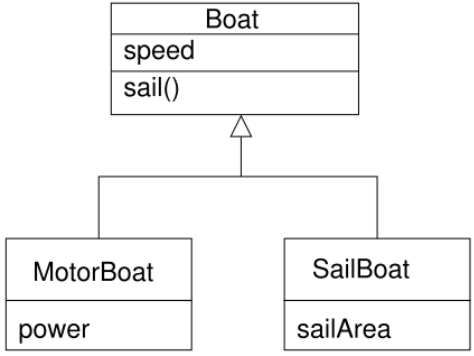
Inheritence

- Every object of a *subclass* is also an object of a *superclass*.
- The inheritance relation is transitive. That is, if **B extends A** and **C extends B**, then **C** is also a **subclass of A**.
- **“Extend”** or **“specialisation of”** simply means that objects of the *subclass* can do everything that objects of the *superclass* can, oftentimes even more.



Inheritence - Substitution

- When ever an object of a class A is expected, you can also use an object of a subclass of A.




```
Boat boat;
boat = new MotorBoat(); // OK
```

Und warum?

Inheritence – Implicit Inheritance

- In Java, every class is a direct or indirect subclass of **Object**.

```
class A {
    // class definition
}
```

Is the same as:

```
class A extends Object {
    // class definition
}
```

That is why

- methods like **public String toString()** and **public int hashCode()**, which are defined in **Object**, are available in every Java class
- the method **public boolean equals(Object o)** accepts any object of any class as parameter

Inheritence – What If…

```
public class Object {
    public String toString () { // (1)
        // original method
    }
}

public class Boat {
    @Override // optional
    public String toString () {  // (2)
        return "Boat{" + " speed=" +
        speed + '}';
    }
}
Object o = new Boat (42);
o.toString();
```

- In the last line, **o** is declared to be of type **Object**.
- An object of type **Boat** is created and assigned to it.
- This is legitimate because **Boat** is a subclass of **Object**

- **But which implementation is invoked? (1) or (2)?**

OOD – Object Oriented Design

Info 3 – Static Type and Dynamic Type

Alexander Kramer

Static Type

- Preciously we have studied the *dynamic type* of an object
- In Java, an **object** also has a *static type*, which is the type of the *declaration* of a variable
- It is called *static type*, because it does not change at runtime

```
Boat boat = new MotorBoat();
// static type: Boat
// dynamic type: MotorBoat
boat.sail(); // implementation of dynamic type is used
```

- The **static** type is known at compile time. It is used by the compiler to check if a method or an attribute is available

- The **dynamic** type may change at run time. It is used by *Java / C# / etc* to find an implementation in case of method overriding

Example

```
Boat boat = new MotorBoat ();
// static type: Boat
// dynamic type: MotorBoat
boat.sail(); // implementation of dynamic type is used
Object o = boat; //OK , for Object is superclass of Boat
// static type of o: Object
// dynamic type of o: MotorBoat
o.sail (); // Illegal, for sail () not defined in Object
```

- **Object o** means something like “I want to use variable o” as an *object of* class **Object**
- **.sail()** is in Boat defined, not in Object**,** so isn’t visible during runtime.

Down Casting

```
Boat boat = new MotorBoat();
// static type: Boat
// dynamic type: MotorBoat
boat.sail(); // implementation of dynamic type is used
Object o = boat; //OK , for Object is superclass of Boat
// static type of o: Object
// dynamic type of o: MotorBoat
o.sail (); // Illegal, for sail() not defined in Object
```

- **Object o** means something like “I want to use variable o” as an *object of* class **Object**
- **Remember:** *static type* determines if a method or an attribute is available

```
Boat b = (Boat)o; // same as object o, but static type is
                  // Boat
b.sail();         // ok
MotorBoat mBoat = (MotorBoat)o;
mBoat.sail(); // ok
// same as object o, but static type is MotorBoat
```

- This means we sometimes need to **“cast”** an object down to another type
- “’o’ has the properties of a ‘Boat’ (“I promise!!!”), so can be assigned to an variable which is defined for that properies (Boat b).”
- Same with “o” cast as “MotorBoat”, cause it was created with the dynamic type “MotorBoat”.

Type Casting in General

```
// Java
int    a = 3;
double d = a; // implicit casting
String s = d + ""; // implicit casting
int    b = (int)d; // explicit casting
```

- Type casting: “changing the type of an expression” [wikipedia]
- Usually, implicit casting is also called type ***coercion***
- Usually from a smaller data type to a larger data type
- Other way around with implicit casting (last line)
- Double -> Floating Point data type

Type Casting in General

```
# Python
# Integer
x = int(1)
y = int(2.8)
z = int("3")

# Floats
x = float(1)
y = float(2.8)
z = float("3")
w = float("4.2")

# Strings
x = str("s1")
y = str(2)
z = str(3.0)
```

- Behaviour depends on the language
- Be aware of data lost (casting from float to int usually doesn’t involve rounding, just cutting of the numbers after the decimal point)

Abstract Classes

```
abstract class Figure {
  abstract public double area();
}
```

Abstract Method

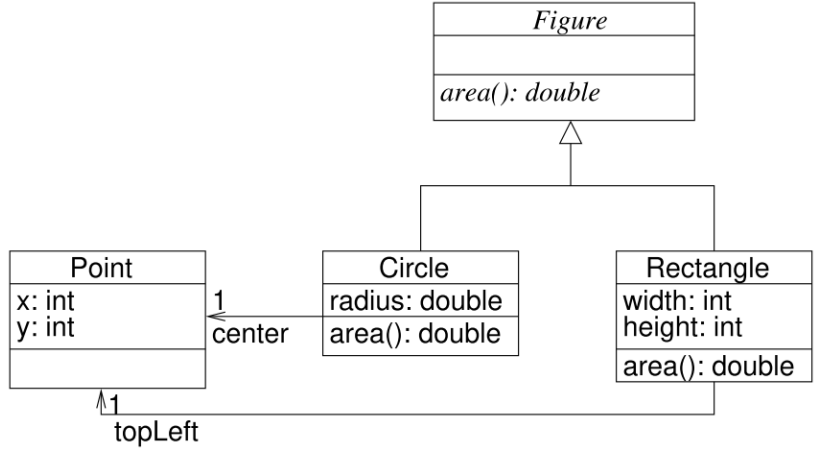
- First, we declare the method area to be abstract by adding the keyword **abstract** in its signature
- An abstract method must not have an implementation (line 2)

Abstract Class

- If a class contains at least one abstract method, then the class must be abstract too
- An abstract class must not contain any constructor, since it would not make sense to create an object as long as some method cannot be executed

Abstract Classes

- In the UML, names of **abstract classes** and **abstract methods** are written in *italic*



Interfaces (1)

- In object oriented programming, inheritance is used for two reasons
  - For code reuse
  - To model an “is-a” relation: **B extends A** expresses “An object of B is an object of A”

- Examples: a *circle* is a *figure*, a *bird* is an *animal*

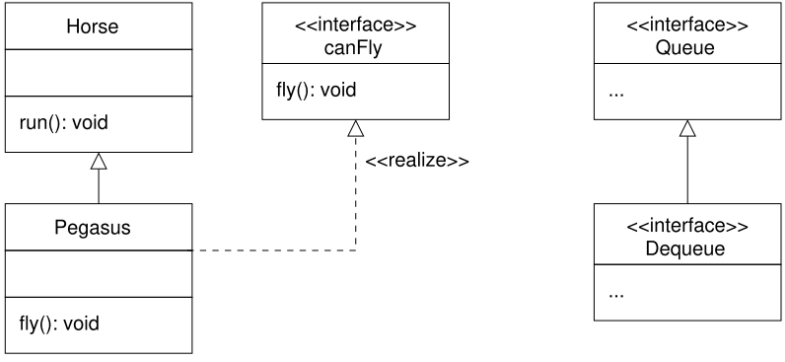
Interfaces (1)

- In object oriented programming, inheritance is used for two reasons
  - For code reuse
  - To model an “is-a” relation: **B extends A** expresses “An object of B is an object of A”
- Examples: a *circle* is a *figure*, a *bird* is an *animal*

- Although Pegasus is a horse and is also a bird, **class Pegasus extends Bird, Horse** is unfortunately not allowed in Java and C#

- Instead, we need to use **interfaces** to express this multiple **“is-a”** relation

Interfaces in UML



- The **≪interface≫** *stereotype* identifies interfaces
- The **≪realize≫** stereotype marks the implements relation in Java
- The **extends** relation between interfaces is presented in the same way as **extends** between classes

Visibility Modifiers

- In **Java** there are four possible visibilities for classes, attributes, and methods:
  - **public**: visible everywhere (+ in the UML)
  - **protected**: visible only in the class and all subclasses (# in the UML)
  - **private**: visible only in the class (- in the UML)
- none visibility modifier: visible only in the package(~ in the UML)

- The same goes for **C#** (with the addition of '*internal',* seen only within the package*)*

- In **Python** everything is basically *public*, but you you can tell the IDE that you *intend* to use methods private
  - *Protected -> single underscore, i.e.* `def _paint():`
  - *Privat -> double underscore, i.e.* `def __paint():`

Modifier final

- Usage (1): When applied to an attribute or a variable, modifier **final** makes it **immutable**
- After the initialization, attribute/variable cannot be assigned another value

```
final int i = 3;      // first assignment OK
i = i + 1            // error
```

In Python there is no native „final“-keyword, but can be reproduced using the typing-package for using strong-typing in Python.

```
from typing import Final
PI: Final[float] = 3.14
```

Modifier final

```
class Person {
   private final Person mother; // birth mother
   private String name;

   public Person(Person mother) {
         this.mother = mother;
   }

   public void setMother(String name) {
      mother.name = name;
      // changes the value of attribute
      // “name” of  “mother” -> OK
   }
}
```

- Note that final only disables **assignment**. In other words, it only prevents the attribute / variable from getting another value
- But it **does not** prevents the object from being modified

Modifier final

- When applied to a method, modifier final prevents the method from being overridden (in subclasses)
  - Example: **public final native Class<?> getClass();**
  - Keyword **native**: this method is not implemented in Java, but is based on other implementations
- When applied to a class, modifier **final** prevents the class from being extended (having subclasses)
  - Example: **public final class String**

Refactoring

Alexander Kramer

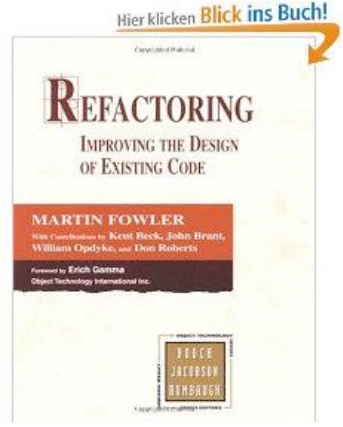
Comic (c) dilbert.com

Refactoring

- “Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior.”
- Martin Fowler, http://www.refactoring.com/

Catalog:

- http://www.refactoring.com/catalog/index.html



1999

Good Design = Easy to Change

- If the system design does not accommodate a new Requirement:
  - Refactor to make the change easy
  - Implement the easy change

Example for Refactorings

1. Add Parameter
2. Extract Method
3. Rename Method/Variable
4. Inline Method
5. Extract Class
6. Move Method/Field
7. Replace Conditional with Polymorphism
8. Decompose Conditional
9. Introduce Explaining Variable
10. Replace Magic Numbers with Named Constants
11. Encapsulate Collection
12. Generalize Type
13. Simplify Method Calls
14. Use Dependency Injection
15. Eliminate Redundant Code
16. Use Design Patterns

Kann man mindestens zwei Beispiele für Refactoring erklären? Und fünf benennen? Problem, Lösung, Vorgehensweise.

Refactoring Tips

- Before you start refactoring, check that you have a **solid suite of tests**. These tests must be *self-checking*.

- Refactoring changes the programs in **small steps**. If you make a mistake, it is easy to find the bug.

- Any fool can write code that a computer can understand. Good programmers write **code that humans can understand**.

- When you feel the need to write a comment, first try to refactor the code so that any comment becomes superfluous.

Was ist self-checking?

Tests

Alexander Kramer

Testing

Why We Test

- Tests give us confidence that the code is ***probably*** correct
- Tests make it possible to find bugs before they are built into the production system
- Tests make change of our software system possible
- Tests as documentation
- Tests as acceptance criteria

Testing

Why We Test

- Tests give us confidence that the code is ***probably*** correct
- Tests make it possible to find bugs before they are built into the production system
- Tests make change of our software system possible
- Tests as documentation
- Tests as acceptance criteria

- Note, however, that tests cannot guarantee the correctness of our software

- *Testing can only show the presence of errors, not their absence*

(Dijkstra, 1972)

Testing

Types of Tests: Black Box vs. White Box

- **White box tests**: testers can access the implementation, read documentation or code
- **Black box tests**: testers can not access the above, but only observe the output
- **Gray box tests**: testers can read the documentation, but not the code

Testing

Types of Testing: Unit vs. Integration vs. System Tests

- **Unit tests:** testing a unit of code, usually a method or a class
- **Component tests:** testing components, which consist of several classes
- **Integration tests:** testing the interaction of several components
- **System tests:** testing the correctness of the system as a whole

Testing

Unit Tests

- testing a unit of code, usually a method or a class
- Mostly down to a comparison of expected outcome to observed outcome
- Self-validating test (knows if there was success or not)
- Bundled into test suits (sum of all test i.e. of a class)

- Can be part of *continues development*

Testing

Unit Tests - Java

```
@BeforeAll
static void setUp()
{
   helloWorld = new HelloWorld();
   LinkedList<String> ll = new LinkedList<String>();
}
```

Vorbereiten eines Test-Sets, z.B. Instanziieren einer Klasse, Aufbau einer Kommunikationsverbindung etc.

```
@BeforeEach
static void setUp()
{
   ll.clear(); // Löschen aller Elemente
   ll.add("FirstElement");
}
```

Vorbereiten vor jedem einzelnen Test-Sets, z.B. Zurücksetzen der Testdaten

```
@AfterAll
static void setUp()
{
   // OutStream os = new OutStream();
   // os.open();
   // After all  the Tests:
   os.close();
}
```

Nacharbeiten nach allen Tests: gracefully beenden von Streams, löschen von angelegten Dateien etc.

```
@AfterEach
static void setUp()
{
 // OutStream os = new OutStream();
   // os.open();
   // After all  the Tests:
   os.close();
}
```

Nacharbeiten nach jedem einzelnen Test, wenn z.b. Testinhalt verschiedene Verbindungsaufbaus für Streams is.

Testing

Unit Tests - Java

- Most common *assert*-methods in Java:
  - assertEquals(expected, actual):
  - assertNotEquals(expected, actual)
  - assertTrue(condition):
  - assertFalse(condition):
  - assertNull(object):
  - assertArrayEquals(expectedArray, actualArray):
  - assertSame(expected, actual): // compares references
  - fail(): //forced abort, can be used to test if a part of a code was reached even if it shoulnd be reached

Testing

Properties of Desirable Tests: FIRST

- **F**ast
- **I**solated
  - A test should concern only a small part of code, and is isolated from the rest of the system
  - Tests should be isolated from each other. The order of their execution should not matter
- **R**epeatable
  - The result of the test should be the same, independently of the time
- **S**elf-Validating
  - No other, external verification should be necessary
- Unit tests: **T**imely
  - Unit tests should be written together with the code that should be tested

Testing

Boundary Value Analysis

- Data at the “border” of the *domain of definition* often cause problems
  - Example: write a method to return a random integer between **m** and **n** (both included)

What is domain of definition?

Testing

Guidelines: Examples

- Implement unit tests to trigger every error message
- Test methods with data outside its domain of definition
- Test methods with collections (*lists*, *sets*, etc.) as input parameters with empty or one-element collections
- Use collections of different sizes in tests

Testing

Test Coverage

What do we need to test?

- We need both **positive** and **negative** tests
- **Each class** needs to be tested
- **Each method** needs to be tested
- **Each if or else** branch needs to be tested
- **Each state** needs to be tested
- **Each statement** needs to be tested
- **Each potential exception** needs to be tested
- etc.

- Criteria like these are called *test coverage criteria*

Define test coverage?

Testing

Test Driven Development: Basic Idea

- Write code to satisfy tests

- How to TDD: repeat until product is done
  - Write a failing test
  - Minimal modification of your code to make all tests pass
  - Refactor and test if necessary

- This cycle is also known as the red-green-cycle

Clean Code &
Working with Legacy Code

Alexander Kramer

Clean Code

- DRY: Don’t repeat yourself!
- KISS: Keep it simple, stupid!
- KISS: Keep it simple and stupid!
  - “One of my most productive days was throwing away 1000 lines of code” [Ken Thompsen]
- YAGNT: You ain’t gonna need it
  - Don’t be too eager to improve/optimize your code
  - Make your code functionally correct first, then improve it

Clean Code

Software Metrics

Gives hints to software quality: quantifying some aspects of the software

- LOC: Line of code, KLOC: Kilo line of code (without comments)
- Depth of inheritance hierarchy
- Length of invocation chain
- Depth of nested **ifs**

- **Fog Index**: average length of words in the documentation. The larger the Fog Index, the harder it is to understand the documentation

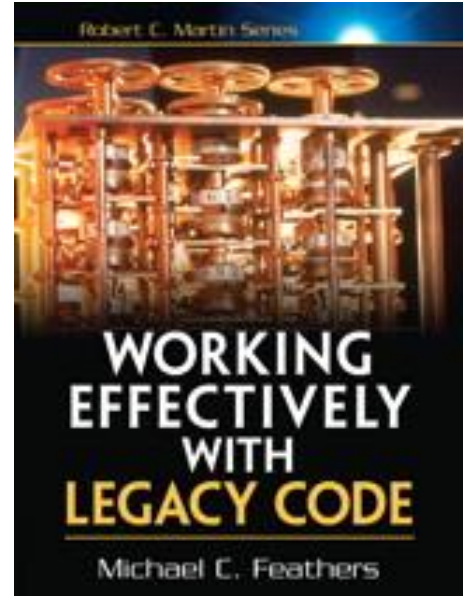
- Cyclomatic complexity

Seam

- “A seam is a place where you can alter behaviour in your program without editing in that place”.

- “Enabling Point: Every seam has an enabling point, a place where you can make the decision to use one behaviour or another.”

  (Michael Feathers: Working Effectively with Legacy Code, p. 36)



Seam Types / Feathers

- Preprocessing Seams
- Link Seams
- Object Seams

Erklärung und Beispiel?

…. Stubs and Mocks

- **Stubs** provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test.

- **Mocks**: objects pre-programmed with **expectations** which form a specification of the calls they are expected to receive.

Continuous Integration & Delivery (CI & CD)

Continuous Integration

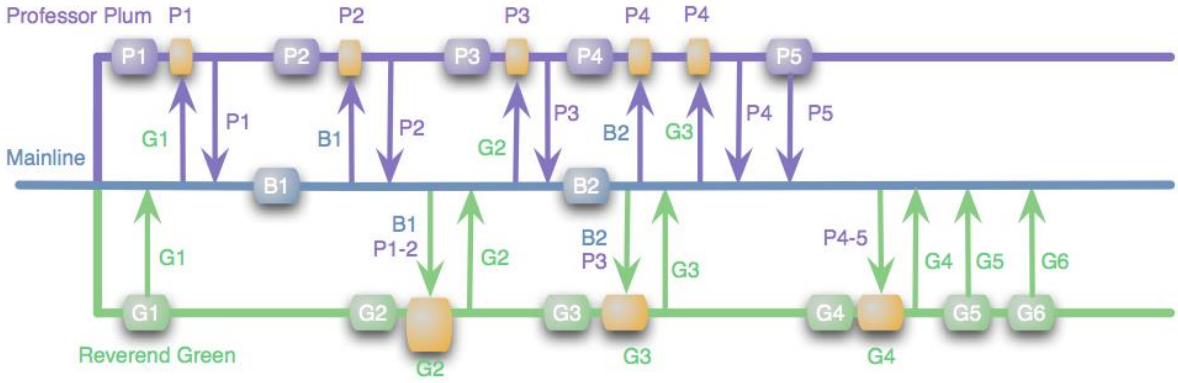
Definition

- *“Continuous Integration is a software development **practice** where members of a team **integrate their work frequently**, usually each person integrates **at least daily** - leading to multiple integrations per day. Each integration is **verified by an automated build** (including test) to **detect** integration **errors as quickly as possible.**”*

Martin Fowler, http://martinfowler.com/articles/continuousIntegration.html

CI Practices 1/4

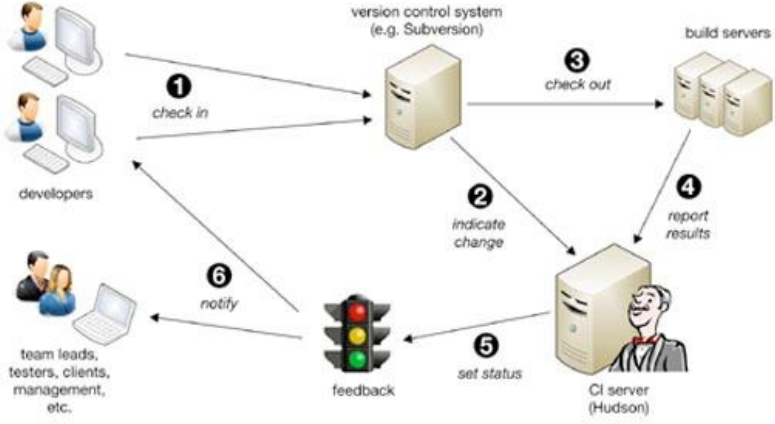
- Maintain a Single Source Repository.
- Everyone Commits to the Mainline Every Day
- Run Tests locally before committing



Jez Humble, http://www.slideshare.net/jezhumble/continuous-delivery-5359386, Graphic by Martin Fowler

CI Practices 2/4

- Automate the Build (Dependency Management)
- Make Your Build Self-Testing (Test Automation)
- Every Commit Should Build the Mainline on an Integration Machine => CI Server



http://www.methodsandtools.com/tools/tools.php?hudson

Deployment

Deployment

- “Deployment: taking the components that make up a release (typically a specific version of an application) and getting them correctly set up in an infrastructure environment so that the release is accessible to (end) users.”

  Andrew Phillips: Deployment is the new Build, http://devopsdays.org/events/2011-boston/proposals/deployment_is_the_new_build/, Slide 20

DEV

TEST

PRD

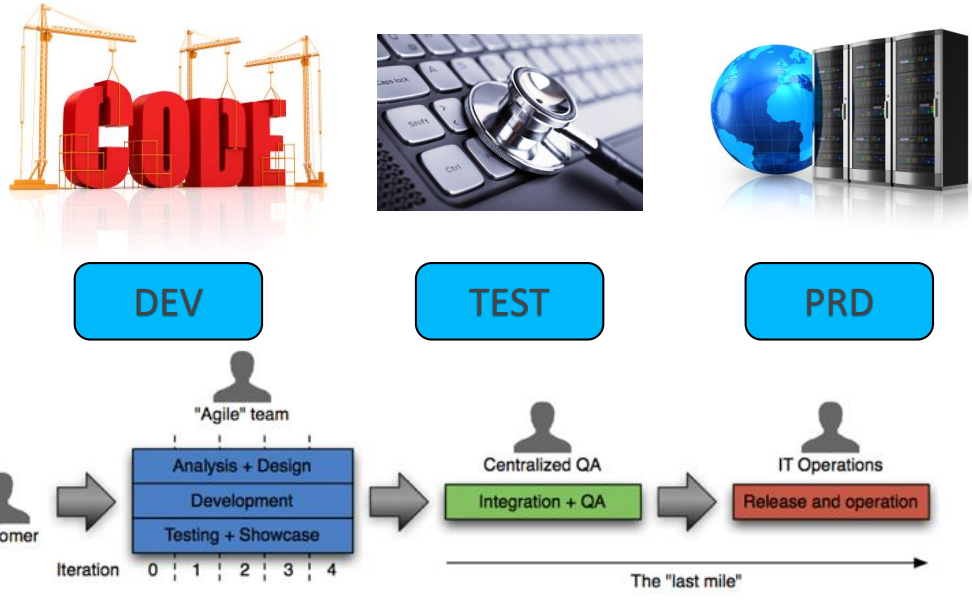
Continuous Integration

Continuous Deployment

Continuous Delivery

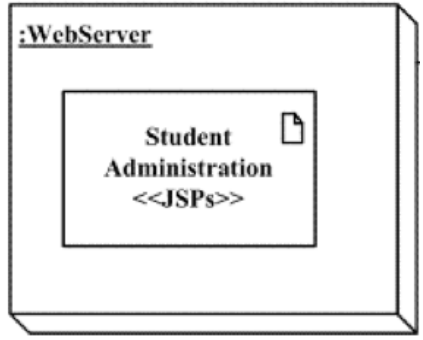
Why is Continuous Delivery difficult?

QA & OPS are not part of the agile process



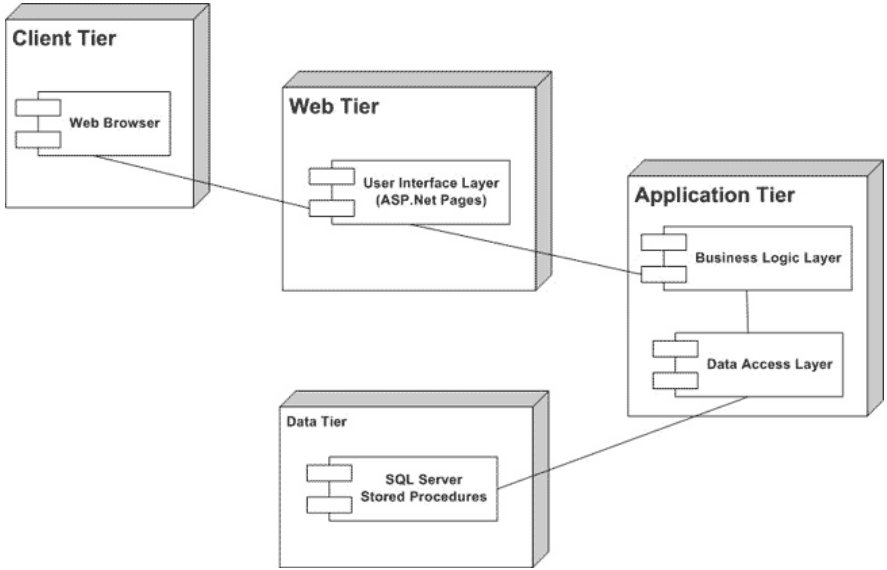
Jez Humble: “Agile 101”
http://www.slideshare.net/jezhumble/continuous-delivery-5359386

Automatic Deployment – What's the big deal?



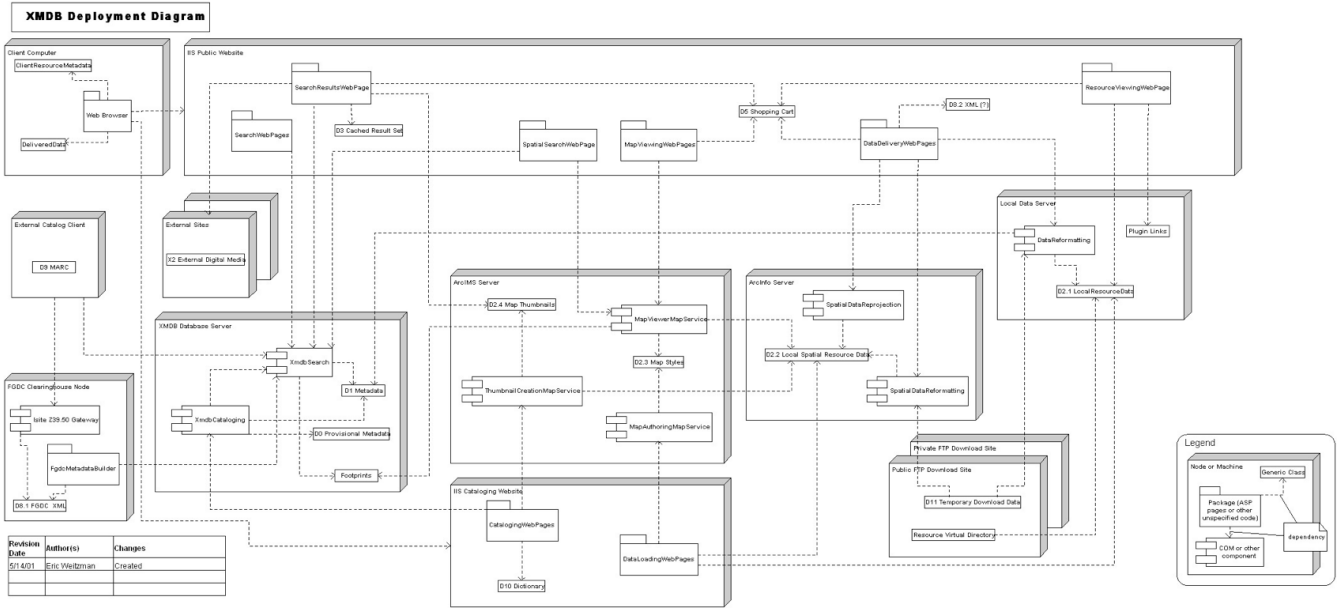
http://www.agilemodeling.com/style/deploymentDiagram.htm

3-Tier Architecture



http://www.education.vic.gov.au/devreskit/appdev/standards/documentation/documentation-standards-details.htm

Real Life Deployment Diagram



http://proceedings.esri.com/library/userconf/proc01/professional/papers/pap595/p595.htm

Snowflake Servers

Foto by Doug Waldron http://www.flickr.com/photos/dougww/5259100253/

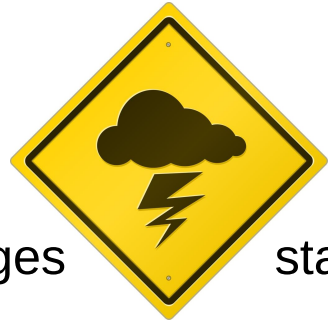
Big Ball of Mud

https://de.wikipedia.org/wiki/Big_Ball_of_Mud

Foto by Moff http://www.flickr.com/photos/moff/3262796959

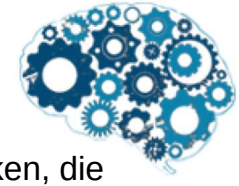
Conflicting Goals!

development vs. operations

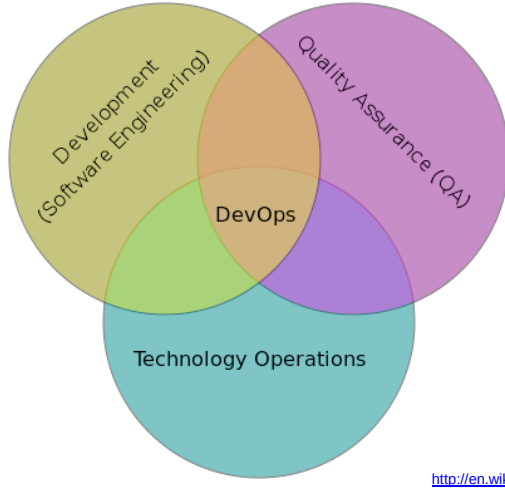


implement changes stability of the platform

Buzzword: Dev-ops



- Es handelt sich um eine Kollaborationskultur und eine Reihe von Praktiken, die darauf abzielen, die Kluft zwischen Softwareentwicklung und IT-Betrieb zu überbrücken.



http://en.wikipedia.org/wiki/File:Devops.svg

Dev-ops: Change of Culture

- developers and operations work together:
- ops are involved in development,
  - ops have requirements, too: stability, maintainability, monitorabilty...

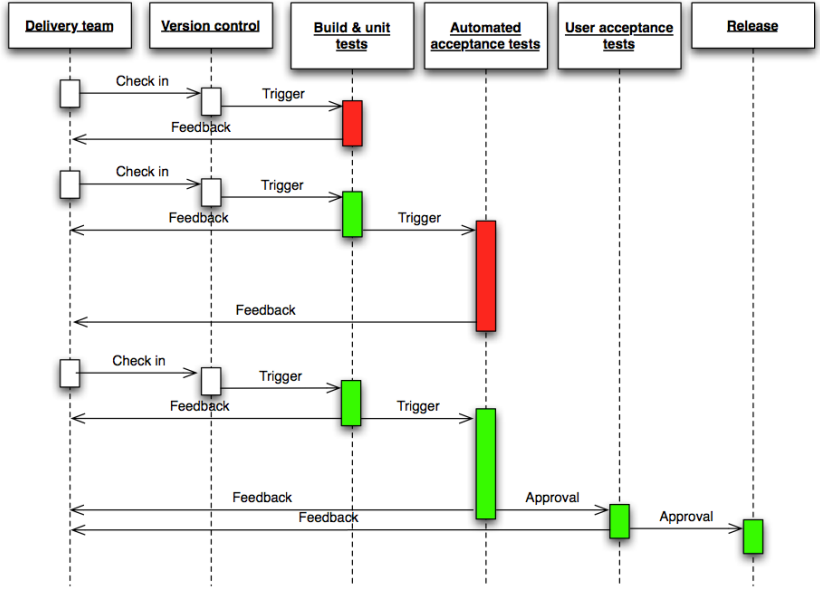
- devs rotate through ops & carry pagers
- one team: together responsible for whole platform: features & stability

Dev-ops: Automation

- Deployment Pipeline: Automate and Test everything:
  - Tests
  - Deployments: Testing every single part of the release process
  - Database Migrations

- Infrastructure as Code

Deployment Pipeline



Jez Humble, http://www.slideshare.net/jezhumble/continuous-delivery-5359386

Why were we doing all this again?

Warum will man schnelles Deployment haben?

Princiles behind the Agile Manifesto

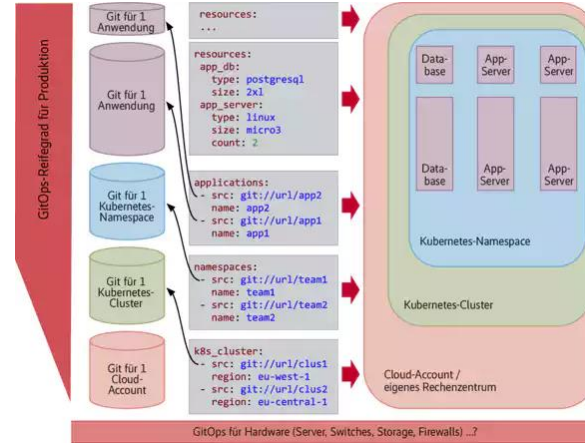
- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.

- http://agilemanifesto.org/principles.html

Current Tools

GitOps

- Zielzustand (“Was”) beschreiben - getrennt vom “Wie”
- Agenten stellen diesen Zustand ohne menschliches Eingreifen her
- GitOps nutzt Git-Repositories als Single Source of Truth für die Bereitstellung von Infrastructure as Code (IaC).
- *IaC* bezieht sich auf das Verwalten und Provisionieren von Infrastruktur durch Code statt durch manuelle Prozesse.



Schapiro, S. (2021) ‘Finger weg: GitOps läutet die Ära des automatisierten IT-Betriebs ein’, *iX*, 17 March, pp. 42–47.
Available at: https://www.heise.de/select/ix/2021/4/2032116550453239806 (Accessed: 16 December 2024).

Summary

Continuous... What?

- **Continuous Integration:**
  Integrate Source Code Continuously. (No Branches.)
- **Continuous Deployment:**
  Deploy Every Good Build to Production.
- **Continuous Delivery:**
  Push-Button Deployment to Production

[Humble, Farley 2010]