The Agile Manifesto

- "We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
  - Individuals and interactions over processes and tools
  - Working software over comprehensive documentation
  - Customer collaboration over contract negotiation
  - Responding to change over following a plan.
- That is, while there is value in the items on the right, we value the items on the left more."

The Agile Manifesto - Principles

- Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
- Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
- Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

The Agile Manifesto - Principles

- Business people and developers must work together daily throughout the project.
- Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
- The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.

The Agile Manifesto - Principles

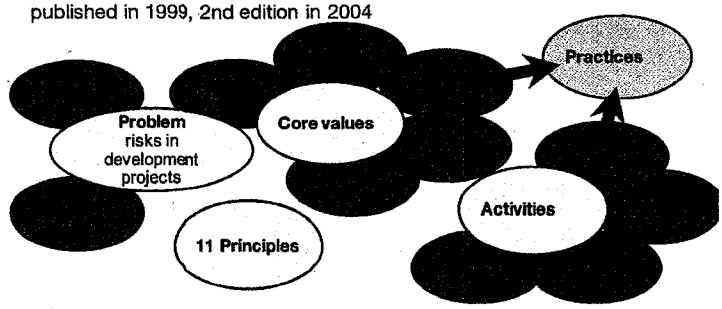
- Working software is the primary measure of progress.
- Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

The Agile Manifesto - Principles

- Continuous attention to technical excellence and good design enhances agility.
- Simplicity - the art of maximizing the amount of work not done - is essential.
- The best architectures, requirements, and designs emerge from self-organizing teams.
- At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

eXtreme Programming (XP)

- Kent Beck
  - Other contributors: Ward Cunningham, Ron Jefferies
- Chrysler Comprehensive Compensation System (C3), 1996
- "eXtreme Programming explained - embrace change", published in 1999, 2nd edition in 2004



The Values

- Communication
- Simplicity
- Feedback
- Courage
- Respect

The Values of XP: communication

- Everyone is part of the team and we communicate face to face daily.
- We will work together on everything from requirements to code.
- We will create the best solution to our problem that we can together.

http://www.extremeprogramming.org/values.html

The Values of XP: simplicity

- We will do what is needed and asked for, but no more.
- This will maximize the value created for the investment made to date.
- We will take small simple steps to our goal and mitigate failures as they happen.
- We will create something we are proud of and maintain it long term for reasonable costs.

http://www.extremeprogramming.org/values.html

The Values of XP: feedback

- We will take every iteration **commitment** seriously by delivering **working software.**
- We **demonstrate** our software early and often then listen carefully and make any changes needed.
- We will talk about the project and **adapt our process** to it, not the other way around.

http://www.extremeprogramming.org/values.html

The Values of XP: courage

- We will **tell the truth about progress** and estimates.
- We **don't document excuses** for failure because we plan to succeed.
- We don't fear anything because **no one ever works alone.**
- We will **adapt to changes** when ever they happen

http://www.extremeprogramming.org/values.html

The Values of XP: respect (new in V2)

- Everyone gives and feels the respect they deserve as a valued team member.
- Everyone contributes value even if it's simply enthusiasm.
- Developers respect the expertise of the customers and vice versa.
- Management respects our right to accept responsibility and receive authority over our own work.

http://www.extremeprogramming.org/values.html

XP: basic principles

zh aw School of Engineering

Fundamental principles

- Rapid feedback
- Assume simplicity
- Incremental change
- Embracing change
- Quality work

Other principles (contd)

- Teach learning
- Small initial investment
- Play to win
- Concrete experiments
- Open, honest communication
- Work with people's instincts, not against them
- Accepted responsibility
- Local adaptation
- Travel light
- Honest measurement

XP Practices (v1)

zh aw School of Engineering

- The Planning Game
- Small releases
- Metaphor
- Simple design
- Testing
- Refactoring
- Pair programming
- Collective ownership
- Continuous integration
- 40 hours week
- On-site customer
- Coding standards

XP Practices (v1)

zh aw School of Engineering

- The Planning Game
- Small releases
- Metaphor
- Simple design
- Testing
- Refactoring
- Pair programming
- Collective ownership
- Continuous integration
- 40 hours week
- On-site customer
- Coding standards

- Balance between **business** and **technical** considerations
- **Business** people decide about:
- Scope + Priority + Composition of releases + Dates of releases
- **Technical** people decice about:
- Estimates + Consequences + Process + Detailed Scheduling

XP Practices (v1)

zh aw School of Engineering

- The Planning Game
- Small releases
- Metaphor
- Simple design
- Testing
- Refactoring
- Pair programming
- Collective ownership
- Continuous integration
- 40 hours week
- On-site customer
- Coding standards

- Every release should be as small as possible, containing the **most valuable business requirements.**
- The release has to make sense as a whole (no half-working features).
- Better to release once a month than twice a year.

XP Practices (v1)

zh aw School of Engineering

- The Planning Game
- Small releases
- Metaphor
- Simple design
- Testing
- Refactoring
- Pair programming
- Collective ownership
- Continuous integration
- 40 hours week
- On-site customer
- Coding standards

- Everybody on the team needs to have a **"common understanding"** for the system.
- Everybody on the team needs to have a **"shared vocabulary"**.
- This applies to **technical** and **non-technical** people.
- What are the basic elements of the system and what are their relationships?

XP Practices (v1)

zh aw School of Engineering

- The Planning Game
- Small releases
- Metaphor
- Simple design
- Testing
- Refactoring
- Pair programming
- Collective ownership
- Continuous integration
- 40 hours week
- On-site customer
- Coding standards

- The **right design** for a software system is one that:
- Runs all tests.
- Has no duplicated logic.
- Has the fewest possible classes and methods.
- "Put in what need when you need it".
- **Emergent, growing design**; no big design upfront (through refactoring)

XP Practices (v1)

zh aw School of Engineering

The Planning Game	Pair programming
Small releases	Collective ownership
Metaphor	Continuous integration
Simple design	40 hours week
Testing	On-site customer
Refactoring	Coding standards

- Any program feature without an **automated test** simply doesn't exist.
- The tests become part of the system.
- The tests allow the system to **accept change**.
- **Development cycle:**
- Listen (requirements)
- Test (write first)
- Code (simplest)
- Design (refactor)

XP Practices (v1)

zh aw School of Engineering

The Planning Game	Pair programming
Small releases	Collective ownership
Metaphor	Continuous integration
Simple design	40 hours week
Testing	On-site customer
Refactoring	Coding standards

- When implementing a feature, ask yourself if there is a way to **improve** the existing source code, so that implementing the feature is easier.
- Automated tests provide a **safety-net** for refactoring without fear.

XP Practices (v1)

zh aw School of Engineering

The Planning Game	Pair programming
Small releases	Collective ownership
Metaphor	Continuous integration
Simple design	40 hours week
Testing	On-site customer
Refactoring	Coding standards

- All production code is written by **two people** looking at **one screen**, with one keyboard and one mouse.
- **Two roles**. The programmer on the keyboard focuses on the current method. The other programmer thinks about the broader context (refactoring, etc.)
- **Pairs change frequently**.

XP Practices (v1)

zh aw School of Engineering

The Planning Game	Pair programming
Small releases	Collective ownership
Metaphor	Continuous integration
Simple design	40 hours week
Testing	On-site customer
Refactoring	Coding standards

- **Anybody** who sees an **opportunity** to **add value** to **any portion** of the code is required to do so at any time. TTL.
- Everybody **takes responsibility** for the whole of the system. Not everyone knows every part equally well, but everyone knows something about every part.

XP Practices (v1)

zh aw School of Engineering

The Planning Game	Pair programming
Small releases	Collective ownership
Metaphor	Continuous integration
Simple design	40 hours week
Testing	On-site customer
Refactoring	Coding standards

- Code is integrated and tested **at least once a day** (sometimes more).
- **Build process must me automated**, on a dedicated machine.
- Automated **tests** are run and make it possible to identify problems early.

XP Practices (v1)

zh aw School of Engineering

The Planning Game	Pair programming
Small releases	Collective ownership
Metaphor	Continuous integration
Simple design	40 hours week
Testing	On-site customer
Refactoring	Coding standards

- **Sustainable** development. Effort should be **spread out** evenly.
- Extended periods of overtime have a negative impact on productivity.
- Goal: be **fresh** every morning, be tired and **satisfied** every evening.
- Not being in front of a computer does not mean forgetting about the system... taking a step back often leads to "Aha!" moments.

XP Practices (v1)

School of Engineering

The Planning Game	Pair programming
Small releases	Collective ownership
Metaphor	Continuous integration
Simple design	40 hours week
Testing	On-site customer
Refactoring	Coding standards

- A real customer must be physically with the team, available to answer their questions.
- Real customer = user who will use the system.
- The real customer does not work on the project 100% of his time, but needs to be "there" to answer questions rapidly.
- The real customer also help with prioritization.

XP Practices (v1)

School of Engineering

The Planning Game	Pair programming
Small releases	Collective ownership
Metaphor	Continuous integration
Simple design	40 hours week
Testing	On-site customer
Refactoring	Coding standards

- **Collective ownership** + **constant refactoring** means that coding practices must be unified in the team.

XP (V2) : 13 primary practices

School of Engineering

Sit Together	Quarterly Cycle
Whole Team	Slack
Informative Workspace	Ten-Minute Build
Energized Work	Continuous Integration
Pair Programming	Test-First Programming
Stories	Incremental Design
Weekly Cycle	

XP (V2): 11 corollary practices

School of Engineering

Real Customer Involvement	Code and Tests
Incremental Deployment	Single Code Base
Team Continuity	Daily Deployment
Shrinking Teams	Negotiated Scope Contract
Root-Cause Analysis	Pay-Per-Use
Shared Code	

What is a User Story?

- A concise, written description of a piece of functionality that will be valuable to a user (or customer) of the software.

As a [user role], I want to [goal] so that [benefit]

User Story Cards have 3 parts

- **Card** - A written description of the user story for planning purposes and as a reminder
- **Conversation** - A section for capturing further information about the user story and details of any conversations
- **Confirmation** - A section to convey what tests will be carried out to confirm the user story is complete and working as expected

Techniques for gathering stories

Different techniques for gathering stories:

- Questionnaires
- Interviews
- Workshops

INVESTing in good stories

Good stories focus on six attributes:

- **I**ndependent
- **N**egotiable
- **V**aluable to users or customers
- **E**stimatable
- **S**mall
- **T**estable

The acronym INVEST stands for these six attributes.

Independent

- Dependenices lead to problems estimating and prioritizing
- Can ideally select a story to work on without pulling in 18 other stories

Example: Dependant story

- Example of dependant stories:
  - A company can pay for a job posting with a Visa card
  - A company can pay for a job posting with a MasterCard
  - A company can pay for a job posting with an American Express card
- How to fix these stories:
  - Combine the dependent stories into one larger but independent story
  - Find a different way of splitting the stories

Negotiable

- Stories are not contracts, they are reminders for the developer and the customer
- Leave or imply some flexibility
- Mistakenly associate the extra detail with extra precision
- Details that were already discussed may always be noted down as tests (back of card)

Negotiable: Example tests

Example of tests for the Credit Card story:

- Test with Visa, MasterCard and American Express (pass). Test with Diner's Club (fail).
- Test with good, bad and missing card ID numbers.
- Test with expired cards.
- Test with over $100 and under $100.

Valuable

- Stories should be valuable to users or customers, not developers
- Rewrite developer stories to reflect value to users or customers

Valuable: Example stories

- Avoid stories that are only valued by developers:
  - All connections to the database are through a connection pool.
  - All error handling and logging is done through a set of common classes.
- Better variations of these stories could be the following:
  - Up to fifty users should be able to use the application with a five-user database license.
  - All errors are presented to the user and logged in a consistent manner.

Estimatable

- Because plans are based on user stories, we need to be able to estimate them
- Reasons why stories could be difficult to estimate:
  - Developers lack domain knowledge.
  - Developers lack technical knowledge.
  - The story is too big.
- Solutions:
  - Clarify with customer
  - Spike
  - Split up story

Scalable (or Small)

- User stories should not be so big as to become impossible to plan/task/prioritize with a certain level of certainty.
- Epics typically fall into one of two categories:
  - Complex stories are intrinsically large
  - Compound stories are multiple stories in one

Testable

School of Engineering

- Stories need to be testable
- Successfully tested means successfully developed
- Tests should be automated

INVEST

School of Engineering

- **I**ndependent
- **N**egotiable
- **V**aluable to users or customers
- **E**stimatable
- **S**mall
- **T**estable

Summary

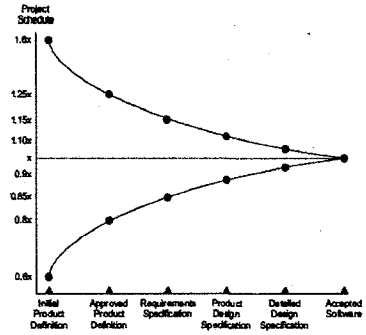
School of Engineering

- User Stories combine written and verbal communications, supported with a picture where possible.
- User Stories should describe features that are of value to the user, written in a user's language.
- User Stories detail just enough information and no more.
- Details are deferred and captured through collaboration just in time for development.
- Test cases should be written before development, when the User Story is written.
- User Stories need to be the right size for planning.

Cone of Uncertainty

School of Engineering

- Ranges of uncertainty in sequential development



An Agile Approach

School of Engineering

- Agile Manifesto values
  - Individuals and interaction over processes and tools
    ... because great software is made by great individuals
  - Working software over comprehensive documetation
    ... because that makes it possible to get feedback early
  - Customer collaboration over contract negotiation
    ...because agile teams would like all parties to the project to be working toward the same set of goals
  - Responding to change over following a plan
    ... because their ultimate focus is on delivering as much value as possible to the project's customer

An Agile Approach to Planning

School of Engineering

- Key idea: A project rapidly and reliably generates a flow of useful new capabilities and new knowledge
- New capabilities are delivered in the product
- New knowledge is used to make the product the best it can be
  - New product knowledge helps us know more about what the product should be
  - New project knowledge is information about the team, the technologies in use, the risk, and so on

An Agile Approach to Planning

School of Engineering

- Key idea: A project rapidly and reliably generates a flow of useful new capabilities and new knowledge
- Failing to plan to aquire new knowledge leads to plans built on the assumption that we know everything necessary to create an accurate plan.
- Is this assumption true in the world of software development?

Summary

School of Engineering

- Agile teams work together but include roles filled by specific individuals
  - Product owner, Customer, Users, developers, and managers are roles on an agile project
- Agile teams work in short, timeboxed iterations that deliver a working product by the end of each iteration
- Projects should be viewed as rapidly and reliably generating a flow of useful new capabilities and new knowledge, rather than just the execution of a series of steps
- Levels of planning: release, iteration, and daily

Story Points

School of Engineering

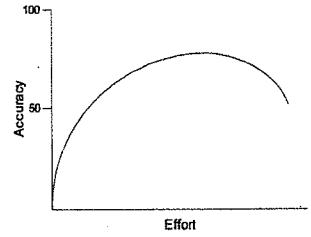
- Number of story points associated with a story represents the overall size of the story
- No set formula for defining the size of a story
- Story points estimates the amount of effort involved in developing the feature, the complexity of developing it, the risk inherent in it, and so on
- Story points are a relative measure of the complexity of a user story

Velocity

- Velocity is a measure of a team's rate of progress
- Velocity is calculated by summing the number of story points assigned to each user story that the team completed during the iteration
- Best guess is, that the team will complete similar amouts of story points per iteration

Techniques for Estimating

- The more effort we put in something, the better the result.
  - Right?
- Agile teams choose to be closer to the left of the Effort-Accuracy-Graph



Select an Iteration Length

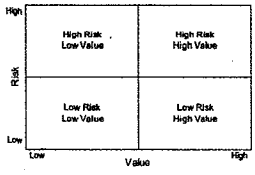
- Most agile teams work in iteration of two to four weeks
- Factors affecting the iteration length are:
  - The length of the release being worked on
  - The amount of uncertainty
  - The ease of getting feedback
  - How long priorities can remain unchanged
  - Willingness to go without feedback
  - The overhead of iterating
  - A feeling of urgency is maintained

Estimate velocity

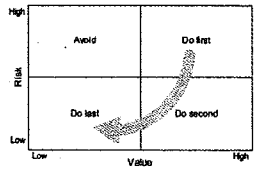
- Use historical values
  - Is the technology, domain, team, product owner, tools etc. the same?
- Run an iteration (or two or three)
- Make a forecast
  - Estimate the number of hours that each person will be available to work on the project each day.
  - Determine the total number of hours that will be spent on the project during the iteration. (→ Focus factor)
  - Arbitrarily and somewhat randomly select stories and expand them into their constituent tasks. Repeat until you have identified enough tasks to fill the number of hours in the iteration.

Risk

- The four quadrants of the risk–value relationship.

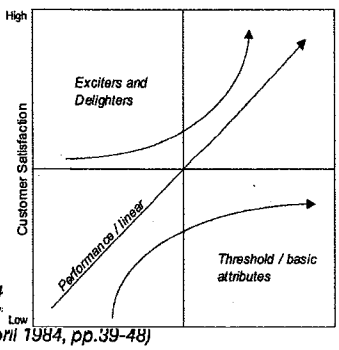


- Combining risk and value in prioritizing features.



Prioritizing Desirability

- Kano Model of Customer Satisfaction
  - Threshold, or must-have features
  - Linear features : "the more, the better"
  - Exciters and delighters: provide great customer satisfaction, price premium



(Kano Noriaki, Nobuhiko Seraku, Fumio Tal "Attractive Quality and Must-Be Quality," Japanese Society for Quality Control, April 1984, pp.39-48)

Summary

- Because there is rarely enough time to do everything, the product owner needs to prioritize what is worked on first.
- Four primary factors:
  - The financial value of having the features
  - The cost of developing (and perhaps supporting) the new features
  - The amount and significance of learning and new knowledge created by developing the features
  - The amount of risk removed by developing the features
- These factors are combined by thinking first of the value and cost of the theme.
- Doing so sorts the themes into an initial order. Themes can then be moved forward or back in this order based on the other factors.

Scrum in (less than) 100 words

- Scrum is an agile process that allows us to focus on delivering the highest business value in the shortest time.
- It allows us to rapidly and repeatedly inspect actual working software (every two weeks to one month).
- The business sets the priorities. Teams self-organize to determine the best way to deliver the highest priority features.
- Every two weeks to a month anyone can see real working software and decide to release it as is or continue to enhance it for another sprint.

Characteristics

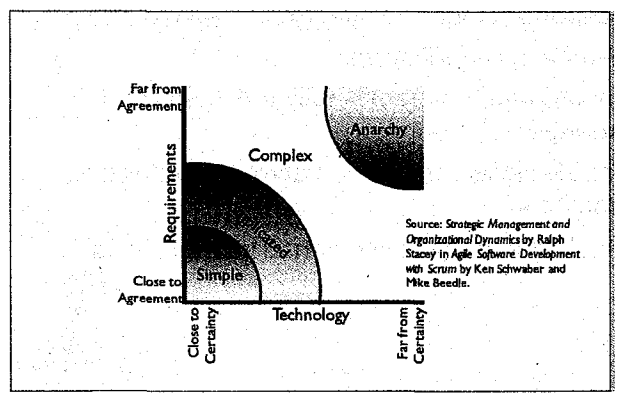
- Self-organizing teams
- Product progresses in a series of month-long "sprints"
- Requirements are captured as items in a list of "product backlog"
- No specific engineering practices prescribed
- Uses generative rules to create an agile environment for delivering projects
- One of the "agile processes"

The Agile Manifesto – a statement of values

Individuals and interactions	over	Process and tools
Working software	over	Comprehensive documentation
Customer collaboration	over	Contract negotiation
Responding to change	over	Following a plan

Source: www.agilemanifesto.org

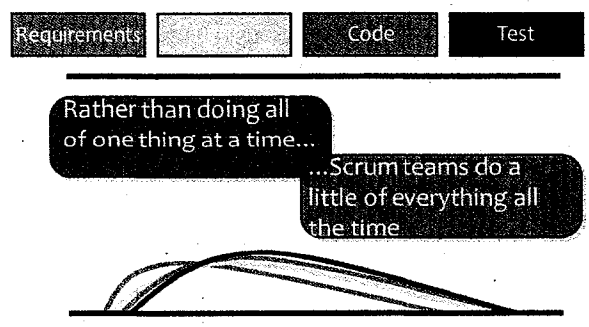
Project noise level



Project noise category

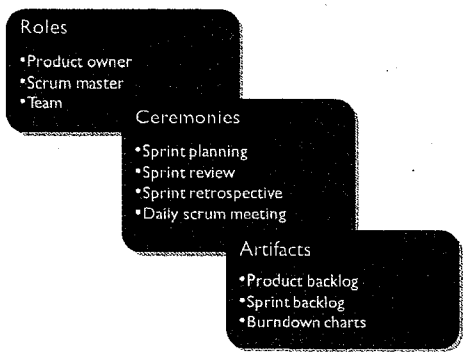
- Process Control Theory: Two approaches to control and manage processes
  - Defined process control model: Simple processes with unobtrusive noise, repeatable
  - Empirical process control model: Complex, noisy processes, not repeatable
- Noise category indicates what process should be used
- Attention: Almost no system development process is so simple, has so little noise, for the defined process control model to be appropriate!
- Empirical processes are managed through frequent inspection and adaptive control

Sequential vs. overlapping development



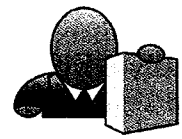
Source: Takeuchi; Nonaka: The New New Product Development Game. Harvard Business Review, January 1986.

Scrum Ingredients



Product owner

- Define the features of the product
- Decide on release date and content
- Be responsible for the profitability of the product (ROI)
- Prioritize features according to market value
- Adjust features and priority every iteration, as needed
- Accept or reject work results

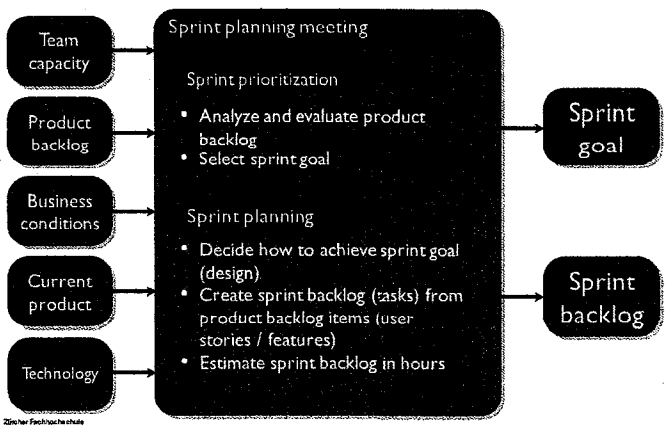


Scrum Master

- Responsible for enacting Scrum values and practices
- Removes impediments
- Ensure that the team is fully functional and productive
- Enable close cooperation across all roles and functions
- Shield the team from external interferences

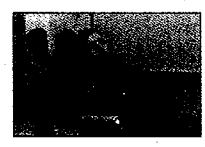
Team

- Typically 5-9 people
- Cross-functional:
  - Programmers, testers, user experience designers, etc.
- Members should be full-time
  - May be exceptions (e.g., database administrator)
- Teams are self-organizing
  - Ideally, no titles but rarely a possibility
- Membership should change only between sprints



Sprint Review

- Team presents what it accomplished during the sprint
- Typically takes the form of a demo of new features or underlying architecture
- Informal
  - 2-hour prep time rule
  - No slides
- Whole team participates
- Invite the world



Sprint retrospective

- Periodically take a look at what is and is not working
- Typically 15–30 minutes
- Done after every sprint
- Whole team participates
  - ScrumMaster
  - Product owner
  - Team
  - Possibly customers and others

Daily Scrum

- Parameters
  - Daily
  - 15-minutes
  - Stand-up

- Not for problem solving
  - Whole world is invited
  - Only team members, ScrumMaster, product owner can talk (→ chickens and pigs)
- Helps avoid other unnecessary meetings

Daily Scrum

Everyone answers three questions:

1. What did you do yesterday?
2. What will you do today?
3. Is anything in your way?

- These are not status for the ScrumMaster
- They are commitments in front of peers

Summary

- Is result-oriented
- Is commitment-driven
- Is value-focused
- Empowers and respects teams

The Sprint

School of Engineering

All Sprints have consistent duration

Starts right after the previous one

Scope is negotiated constantly throughout

- Between Development Team and Product Owner
- This recognizes uncertainty even within the Sprint

Mechanics of Sprint Review

School of Engineering

Product Owner Shares

- What was done
- What wasn't
- State of the Product Backlog
- Likely Release projections

Development Team Shares

- The actual Increment of software
- What happened in the Sprint
- How problems were addressed and the effect on the increment

Everyone

- Provides and hears feedback

This is *not* an excuse for self-congratulations

This *is* an inspect and adapt opportunity

A Typical Sprint Retrospective Model

School of Engineering

What worked well?

What will we commit to doing in the next Sprint?

Scrum Team members make actionable commitments

What is Definition of Done?

School of Engineering

"Complete as mutually agreed to by all parties and conforming to an organization's standards, conventions, and guidelines. When something is reported as "done" at the Daily Scrum or demonstrated as "done" at the Sprint review meeting, it must conform to this agreed definition." – Agile Project Management with SCRUM

So Definition of Done is...

Product quality standard?

A list of activities for producing potential shippable product increment?

Anything else?

Problems of without defining "DONE"

School of Engineering

Technical debts – "Pay me now or pay me later"

The undone work accumulate in a nonlinear manner

Illusion of progress (velocity) and unpredictable ship date

Team over-commit the amount of work they can do in a sprint

The product owner surprises with the delivery during the sprint review session

How to define "DONE"?

School of Engineering

The definition of done should be agreed to by the team, written down and rigorously followed

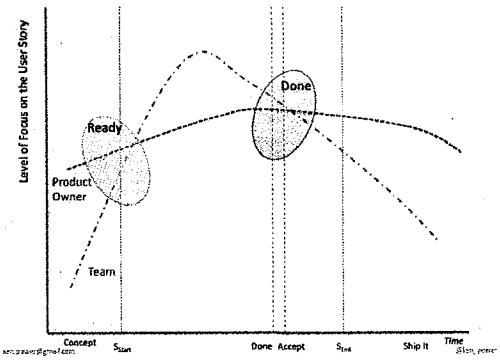
- Definition of done reflect the maturity/experience level of the team

The initial definition of done can be derived/tailored from organization standard.

Involve product owner (client) – "Dancing with client"

Evolve definition of done by inspect and adapt.

Ready? Done?



Definition of Ready for a User Story

- User Story defined
- User Story Acceptance Criteria defined
- User Story dependencies identified
- User Story sized by Delivery Team
- Scrum Team accepts User Experience artefacts
- Performance criteria identified, where appropriate
- Person who will accept the User Story is identified
- Team has a good idea what it will mean to Demo the User Story

Definition of Ready for a Sprint

- The Sprint Backlog is prioritized
- The Spring Backlog contains all defects, User Stories and other work that the team is committing to
- No hidden work
- All team members have calculated their capacity for the Sprint
- Fulltime on project = X hours per day
- All User Stories meet Definition of Ready

Productive Self Organization

Requires skill

- In the domain at hand
- In the constraints of the framework
- In the software development craft

Skills needed in software teams using Scrum

- Scrum itself
- The business domain
- Useful technologies
- Practices of software craftsmanship
- The science of user experience
- Languages and frameworks
- Levels of testing
- Mastery of development tools
- Build and Deploy Automation
- Emerging architecture or design
- Many, many more

2 Parts of the Scrum Discussion

People Practices	Engineering Practices
Planning	Design
Empiricism	Coding
Collaboration	Testing
Self-organization	Automation
Leadership	Deploying
Communication	User experience
Transparency	Emergent Architecture

Fewer Specialists

- In a multi-disciplinary Development Team of appropriate size, specialization is simply not possible
- Task pairing and sharing grows everyone
- Focus shifts from fulfillment of individual duties to the overall success of the team

Self-Organizing Development Teams

zh aw School of Engineering

- The Development Team determines
- Who does what & when
- Who is needed on the team and not
- When it needs help resolving Impediments
- Needed process improvements
- Technology practices
- Their own Scrum Master

If you think good architecture is expensive, try bad architecture

Brian Foote and Joseph Yoder: "Big Ball of Mud" http://www.laputan.org/mud/

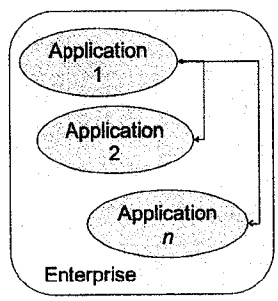
Software Architecture

- **Describes a solution that fits**
  - fulfills functional and non-functional requirements
  - can be realized (political and organizational)
  - can be implemented (e.g. proof through prototypes)
- **Software Architecture**
  - process: design a solution
  - product: models, documentation, prototypes
  - means: eases implementation of larger system

What "Design Decisions"?

- **Structural**
  - "The architectural elements should be organized and composed exactly like this ..."
- **Behavioral**
  - "Data processing, storage, and visualization will be performed in strict sequence."
- **Interaction**
  - "Communication among all system elements will occur only using event notifications."
- **Non-functional**
  - "The system's dependability will be ensured by replicated processing modules."

Kind of Software Architecture



- **Enterprise architecture**
  "... defines way how an enterprise uses many applications"
  - metaphor: *city planning*

- **Application architecture**
  "... defines the pieces that compose an application"
  - metaphor: *building architecture*

Technical Adequacy

Software architecture accomplishes:

- **Fulfillment of functional requirements**
- **Fulfillment of non-functional requirements**
  - Fulfillment of security requirements
  - Fulfillment of data management requirements
  - Fulfillment of integration requirements

And also...

- Maintainability requirements
- Extendibility requirements

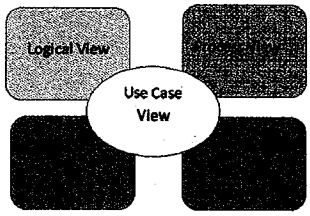
Organizational Aspects

Software architecture interrelates with

- **Team**
  - Complexity
    - Team size: huge software architecture ç è small team?
  - Project type
    - Simple projects with low uncertainty
    - Simple projects with high uncertainty
    - Complex projects with low uncertainty
    - Complex projects with high uncertainty
- **Project organization (Conway's law, 1968)**
  - If you have two groups then chances are high that you get a client-server solution...

4+1 Architectural *Views* Philippe Kruchten

- **Kruchten**
  - Need for several views
  - Requirements tie everything together
    - 4+1 **view**



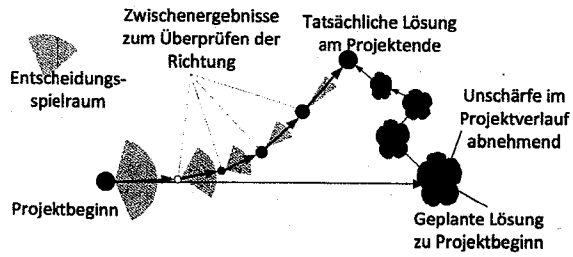
Why do we need a Software Architecture?

- **Even though every stakeholder wants to do her/his best**
  - overall results are bad (too often!)
- **Some reasons:**
  - overall structure inadequate
  - interfaces to the outside inappropriate
  - interfaces between parts don't fit
  - functionality falls in "gap" between the components
- **Architecture can help**

Why is architecture hard?

- **Multi-dimensional decisions are hard**
  - architectural decisions
    - have strong, overarching impact
    - define system- and project structure
- **All factors are interdependent**
  - changeability impacts performance
  - security impacts adaptability
  - everything impacts cost
- **Requirements change continuously**
  - "change is the only constant"
  - architecture guides change
    - framework for change

Solution as Moving Target



Starke: Effektive Software Architekturen

Why architecture (nevertheless)?

- **Communication among stakeholders**
  - basis for discussions and decisions
  - common understanding about the system
  - get consensus
- **document design decisions**
  - guidelines for implementation
  - basis for evolution and iterations
  - helps structuring team and planning
  - checkpoints for goal achievement
- **abstraction of the system (as built)**
  - for building product lines and homogeneous systems
  - for outsourcing or acquiring parts

Architecture as process "architecting"

- **architecting is *design work***
  - but: architecture is not an art, but engineering
- **architecture is about *abstraction***
  - major task: remove needless details, simplify!
    - only abstraction allows understanding of an architecture resp. a system
  - architecture consists of models
    - model = simplified view of reality
- **architecture is about *simplification***
  - needed for architecture talent

Overall goals of architecture and design

- **We should design our systems in a way that**
  - components can be exchanged with new ones easily (***maintainability***)
  - new requirements can be met (***flexibility***)
  - individual components might be used in other systems (***reusability***)
  - the structure of the system is presented understandably (***readability***, ***understandability***)
  - development can be split into team(s) quality attributes [Bass2003]
- **and most important:**
  - we get ***simplest solution*** that works

Some Typical Misconceptions

- **Architecture is the same as design**
  - "Architect's focus is on the boundaries and interfaces"
- **Architecture is about infrastructure**
  - "Frameworks, application servers, and databases from a minor part of the problem space only"
- **Architecture solves technical problems**
  - "Chances are your biggest problem isn't technical"
- **Architecture is rigid and fixed (up front)**
  - "Understand the impact of change"
  - "Start with a walking skeleton"
  - "Great software is not built, it is grown"
- **Architecture is pure science or pure art**
  - requires both and more

Tasks of an Architect (I)

- **Decide (under uncertainty, but decide)**
  - "Use uncertainty as a driver", "Architectural tradeoffs"
- **Document (adequately)**
  - "Communication is king, clarity and leadership its servants"
  - "Record your rationale"
- **Proof feasibility**
  - "One line of working code is worth 500 lines of specification"
  - "Try before choosing"
- **Program**
  - "Architects must be hands on"
  - "Before anything an architect is a developer"
  - "If you design it, you should be able to code it"
  - J. Coplien: Architect always/also implements

Tasks of an Architect (II)

- **Communicate**
  - "Stand up", "Talk the talk"
  - "Learn a new language"
- **Negotiate (with stakeholders)**
  - "Seek the value in requested capabilities"
  - "You're negotiating more than you think"
- **Simplify**
  - "Simplify essential complexity; reduce accidental complexity"
  - "Simplicity before generality, use before reuse"
  - "Make sure the simple stuff is simple"
- **Standardize**
  - "Reduce the entropy"

Tasks of an Architect (3)

- **Listen**
  - "Hear the stakeholder's concerns"
- **Observe**
  - "Don't control, but observe"
  - "Get the 1000-foot view"
- **Think (about the future)**
  - "Everything will ultimately fail"
  - "Focus on application support and maintenance"
  - "Your system is legacy; design for it"
  - "You can't provide future-proof solutions"
- **Lead**
  - "Give developers autonomy"
  - "There is no 'I' in architecture"

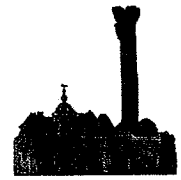
Top 5 mistakes (from IBM Architectural Thinking)

- **Believing the requirements**
- **Being seduced by the technology**
- **Majoring on your strengths and neglecting other areas**
- **Not stopping designers from designing**
- **Thinking you can do it all yourself**

Requirements for an architect

- **Knowledge and Experience in Architecture**
- **Technical breadth and technical understanding**
- **Disciplined, methodical working**
- **Experience with the whole Software Life Cycle**
- **leadership qualities**
- **ability to communicate**

Architect's Anti-Behavior

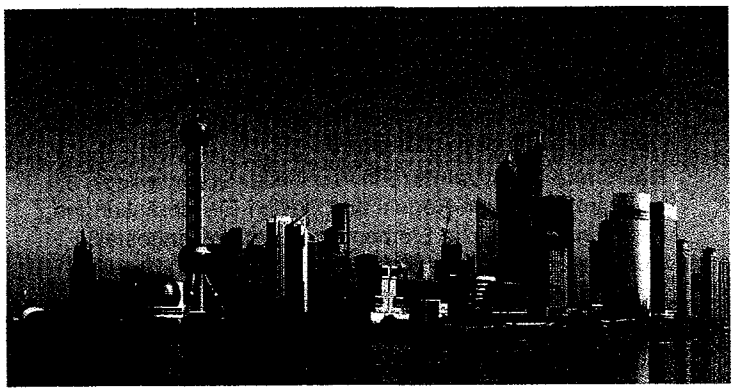


- **design in an ivory tower**
- **design by committee**
- **design by power point**
- **re-invent the wheel**
- **over engineering**
- **under engineering**
- **premature framework design**
  From [97-Things 2009]
  - "Challenge assumptions – especially your own"
  - "Pattern pathology"
  - "Don't be clever"

What an Architect shouldn't do?

- **Marchitecture**
  - do what "market leader companies" propose
- **Design for own *résumé***
  - use most current technology for the sake of it
- **Start with a framework**
  - before understanding requirements
- **Forget to simplify**
- **Add too many layers**

...



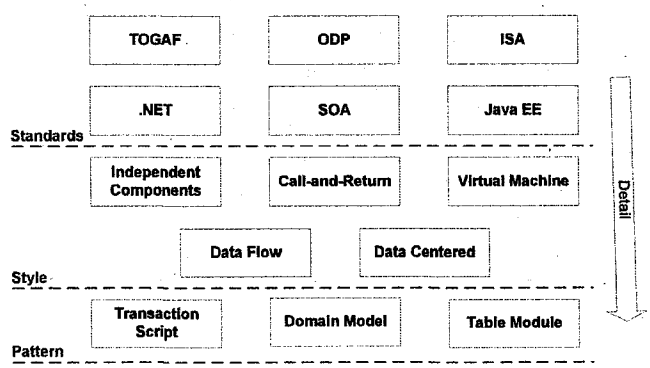
Inhalt

1. Einführung: Positionierung Standard – Stil - Pattern
2. Definition: Was ist ein architektonischer Stil
3. Independent Components
4. Call & Return
5. Virtual Machine
6. Data Flow & Data Centered
7. Beispiel

Einführung: Positionierung Standard – Stil - Pattern

Manhattan oder nicht

Standard – Stil - Pattern



Standard – Stil - Pattern

Das Vorgehen zur Abbildung von Anforderungen auf ein Zielsystem geht über drei Ebenen:

1. **Architektur-Standards:** Auswahl und Anwendung des Architekturstandards
2. **Architektur-Stil:** Auswahl und Anwendung der Kombination der verschiedenen Architektur-Stilelemente
3. **Pattern:** Auswahl und Anwendung der passenden Entwurfsmuster

Definition: Was ist ein architektonischer Stil

Der architektonischer Stil charakterisiert eine Klasse / Familie von Systemen .

Vollständige Definition

Perry / Wolf:

Falls Architektur ein formales Arrangement von architektonischen Elementen ist, so wird eine architektonischer Stil dadurch definiert, dass er Elemente und formale Aspekte verschiedener spezifischer Architekturen abstrahiert. Ein architektonischer Stil ist weniger eingeschränkt und weniger vollständig als eine definierte Architektur.

Die 4 Elemente eines Stils 1/2

- **Vocabulary:** Ein Vokabular von Design-Elementen bestehend aus Komponenten und Verbindungselementen. Komponenten können beispielsweise Server, Parser, Datenbanken und Filter sein. Verbindungselemente können beispielsweise RPC (Remote Procedure Call), Data Streams oder Sockets sein.
- **Rules and Constraints:** Ein Architektur Stil unterliegt bestimmten Design-Regeln und Design-Einschränkungen, die definieren welche Elemente wie zusammengestellt werden können. Diese Regeln vermeiden grundlegende Fehler in einem System Design.

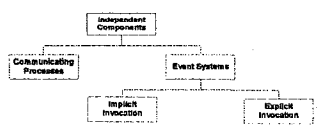
Die 4 Elemente eines Stils 1/2

- **Semantics:** Die Bedeutung der verschiedenen Design-Elemente des Stils ist eindeutig. Somit können semantische Regeln zur Interpretation einer Komposition von Elementen herangezogen werden.
- **Analytics:** Analytische Instrument zur Prüfung des Vokabulars, der Regeln und Einschränkungen eines bestimmten Architektur Stils, respektive zur Analyse des konkret realisierten Systems. Mit diesen Instrumenten können Kriterien wie die „Schedulability" eines Real-Time Systems oder auch „Deadlock Prevention" eines Message-Passing Systems

Übersicht Stile

- **Independent Components:** Unabhängig ablaufende Elemente, die über Nachrichten miteinander interagieren
- **Call-and-Return:** Definiert durch einen fixen Kommunikations-Mechanismus zwischen aufrufendem und aufgerufenen Element
- **Virtual Machine:** Erlaubt die Realisierung portabler und interpretierbarer Systeme
- **Data Flow:** System ist eine Abfolge von Datenbezogenen Transformationen
- **Data Centered:** Zentrale Aufgabe ist Zugriff und Aktualisierung von Daten eines Repositories

Independent Components



Unabhängig ablaufende Elemente, die über Nachrichten miteinander interagieren

Communicating Processes

- **Definition:** Besteht aus einer beliebigen Anzahl miteinander kommunizierender Elemente. Die Synchronisation erfolgt ausschliesslich über die Kommunikationskanäle zwischen den Elementen und kann sowohl synchron als auch asynchron erfolgen
- **Bemerkung:** CSP (Communicating Sequential Processes) von Tony Hoar
- **Beispiel:** Das Springer-Problem

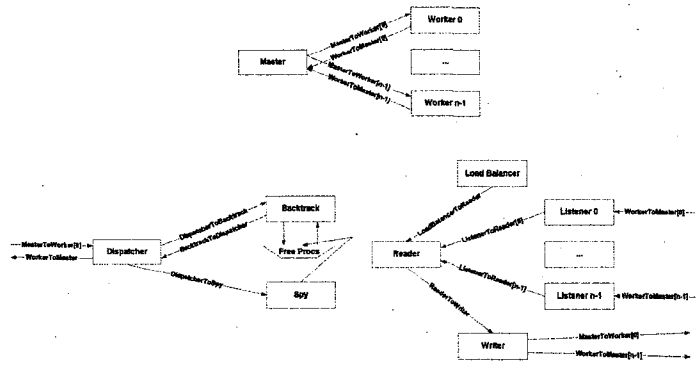
Communicating Processes - Beispiel

Springer-Problem

Ein Springer soll nacheinander alle Felder eines Schachbrettes beliebiger Breite (resp. Länge) unter Einhalten der Schachregeln besuchen ohne auch nur ein einziges Feld zweimal zu betreten. Diese Aufgabenstellung wurde vom Basler Mathematiker Leonhard Euler vor über 200 Jahren in seinem Buch "Memoires de Berlin" zum ersten Mal beschrieben. Eine Überprüfung aller möglichen Züge, die so genannte erschöpfende Suche, ist aufgrund der Anzahl Möglichkeiten (Dimension: n*n ergibt n! Möglichkeiten) zu aufwendig.

Methode: Backtracking

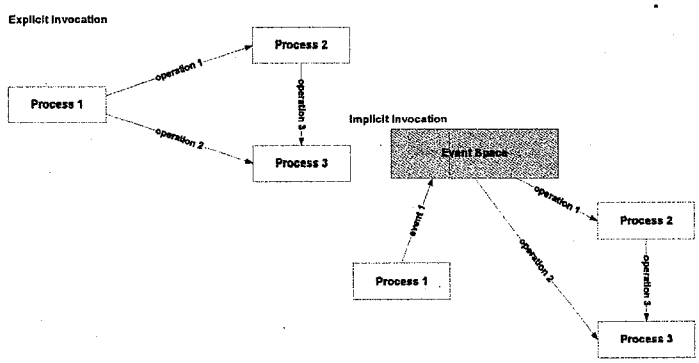
Communicating Processes - Beispiel



Event Systems

- **Event Systems:** Dieser Ereignisgesteuerte Architektur Stil definiert Architekturen, deren Prozesse über Events miteinander kommunizieren.
  - **Implicit Invocation:** Sämtliche Events werden an einen so genannten Event Space gesendet, der dann die Verteilung der Events an die betroffenen Prozesse übernimmt.
  - **Explicit Invocation:** Die Events werden direkt zwischen den einzelnen Prozessen ausgetauscht.
- **Bemerkung:** Publisher / Subscriber – Kommunikation. Zentral ist Trennung der Interaktion von den Elementen, Auflösung statischer Abhängigkeiten und mögliche Parallelisierung
- **Beispiel:** MVC / Message Broker

Event Systems



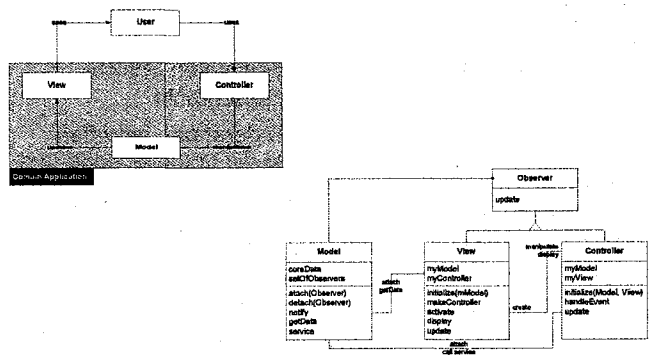
Event Systems - Beispiel

MVC

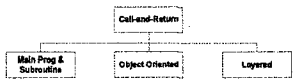
Als Event System: Events, die vom Controller (Mouse Move, Mouse Over, etc) erzeugt werden. Auf diese Events reagiert eines oder mehrere Model Komponenten, die alle Views (dargestellte User Interface Komponenten) nachführen.

- **Model:** Das Herzstück der Applikation. Es enthält die Daten und die Zustände, die ein System zu einem bestimmten Zeitpunkt darstellen. Sobald Änderungen am Model erfolgen, werden sämtliche Views, die ein Model ganz oder teilweise darstellen nachgeführt.
- **View:** Das User Interface, welches die Informationen über ein Model für die Benützer / Benützerinnen der Applikation darstellt.
- **Controller:** Die Steuerung der Interaktion durch den User erfolgt durch den Controller. Der Controller akzeptiert User Inputs als Events

Event Systems - Beispiel



Call & Return

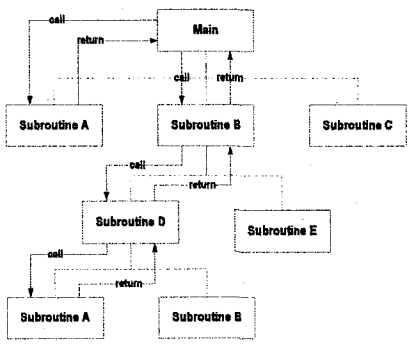


Fixer Kommunikations-Mechanismus zwischen aufrufendem Element und aufgerufenen Element

Main Program & Subroutine

- **Definition:** Repräsentiert das klassische Programmierparadigma der SASD mit eindeutigem Kontrollfluss, der einen Baum bestehend aus Hauptprogrammen und Unterprogrammen, wobei die Kontrolle dem gerade ausführenden Teil des Systems übertragen wird, durchläuft
- **Bemerkung:** SASD (Structured Analysis - Structured Design) von Demarco und Yourdon
- **Beispiel:** RPC

Main Program & Subroutine

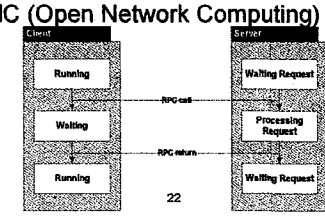


Main Program & Subroutine - Beispiel

Remote-Procedure-Call

RPC ist ein Protokoll, das die Implementierung verteilter Anwendungen vereinfacht. Ein Programm kann eine Funktion eines Programms, das auf einem anderen Rechner läuft, nutzen, ohne sich um die zu Grunde liegenden Netzwerkdetails kümmern zu müssen.

Bemerkung: RPC ist als RFC 1831 definiert, der den so genannten ONC (Open Network Computing) RPC beschreibt



Layered Architecture

Schichtenmodelle sind hierarchisch gegliedert und teilen die Aufgaben eines Gesamtsystems in Schichten auf. Das Spektrum der Elemente, die einer bestimmten Schicht zugeordnet werden können, wird eingeschränkt.

Stärke: Lokalität der Änderungen. Änderungen betreffen maximal die geänderte Schicht sowie die untenliegende und die oben liegende Schicht. Falls an den Schnittstellen zwischen den Schichten nichts geändert wird, so ist sogar lediglich eine Schicht betroffen.

- **Typical Layering:** GUI – Business Layer – Data Access Layer – Data Layer.
- **Bemerkung:** Begriff wurde in der Telekommunikation durch das OSI-Modell etabliert

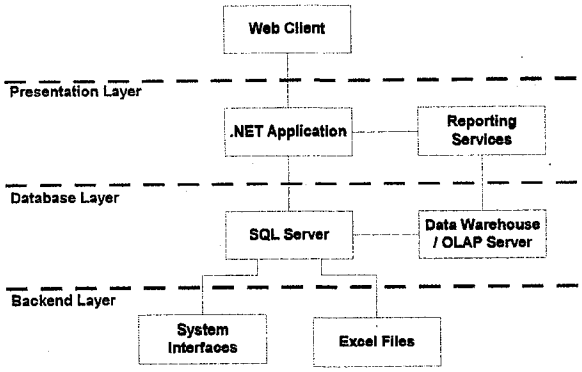
Layered Architecture - Beispiel

Risk Management System

Bewertung und Isolation operativer Risiken durch Konsolidierung von Produktions- und Vertriebsdaten.

- Der **Presentation Layer** enthält die .NET Applikation sowie die Reporting Service Komponente zur Definition, Abfrage und Anzeige und Aufbereitung von Reports.
- Der **Database Layer** besteht aus einem MS SQL Server, der die operativen Daten aus dem Backend Layer täglich bezieht und dem Data Warehouse, welches die täglichen Daten konsolidiert und als OLAP-Qubes aufbereitet.
- Der **Backend Layer** enthält die Daten, die von den operativen System als Excel Files oder aber als Host Files bereitgestellt werden.

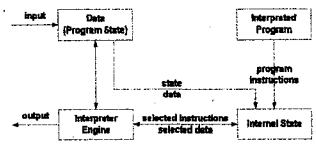
Layered Architecture - Beispiel



Virtual Machine

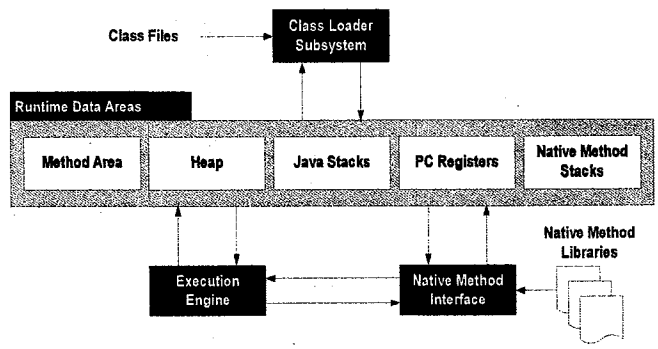
Erlaubt die Realisierung portabler und interpretierbarer Systeme.

Interpreter



- **Definition Interpreter:** Eine abstrakte Maschine – also ein hypothetischer Computer mit den definierten Elementen für deren inneren Aufbau
- **Bemerkung:** Java & C# Runtime sind Interpreter
- **Beispiel:** Java VM

Interpreter - Beispiel



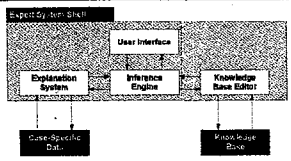
Interpreter – Beispiel Java VM 1/2

- **Class Loader Subsystem:** Diese Komponente lädt Typen (Classes und Interfaces) und prüft die Korrektheit aller importierten Klassen
  - Loading: auffinden und importieren der binären Daten eines Typs
  - Linking: Prüfung der importierten Typen und optionale Auflösung symbolischer Referenzen
  - "Initialization" Initialisierung der Klassen Variablen
- **Execution Engine:** Das Ausführen der einzelnen Instruktionen (Bytecode mit Opcode und optionalen N Operanden)
- **Runtime Data Area:** Daten, Bytecodes, Objekte, Rückgabewerte, lokale Variablen und Zwischenresultate
- **Method Area:** Alle Klassen eines Java Programms
- **Heap:** Instanzierte Objekte des Programms.

Interpreter – Beispiel Java VM 2/2

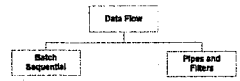
- **PC Register:** Jedem Java Thread wird ein eigenes Program Counter Register und ein eigener Java Stack zugewiesen
- **Java Stack:** Als Stack Frames organisiert, der den Status eines Methodenaufrufs hält. Push / Pop Mechanismus für zwischenresultate
- **Native Method Stacks:** Der Status einer Native Method wird im Native Method Stack gespeichert
- **Native Method Interface:** Diese Schnittstelle erlaubt den Aufruf von Klassen und Methoden, die in Maschinen-sprache in den Native Method Libraries gespeichert sind

Rule Based System



- **Definition Interpreter:** Generalisierung des Interpreter. Zentrale Eigenschaft ist die Trennung zwischen der Maschine an sich und den Regeln, die die Maschine beschreiben.
- **Beispiel:** Inference Engine
  - **Knowlege Base Editor:** Verwaltung der Wissensbasis
  - **Explanation System:** "Erklärung" der Schlussfolgerungen (Basis Case-Specific Data) - gefundene Expertise in Kontext zum Verständnis

Data Flow und Data Centered



Datenorientierte Stile

Data Flow

Der Data Flow Architektur Stil umfasst Architekturen, die ein System als eine Abfolge von Transformationen auf sequentiellem Dateninput modellieren. Aktiviert und gesteuert wird das System durch vorliegende Daten. Diese Daten werden Schritt für Schritt durch das System geleitet, werden dabei verändert und schliesslich als ausgehende Daten bereitgestellt.

- **Batch Sequential:** Die Transformationen auf den Daten erfolgen durch voneinander unabhängige Elemente dieses Architektur Stils. Dies bedeutet, dass die Daten als ganzes eingelesen werden, um dann transformiert und anschliessend wieder als Ganzes abgelegt zu werden.
- **Pipes and Filters:** Die Transformationen auf den Daten erfolgen lokal (Filters) mittels Streaming (Pipes). Dies bedeutet, dass ein eingehender Data Stream laufend in einen ausgehenden Data Stream umgewandelt wird.

Data Centered

Der Datenzentrierte Architektur Stil umfasst Architekturen, die Systeme beschreiben, deren zentrale Funktion der Zugriff auf Daten eines Repositories ist.

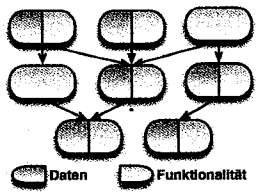
- **Datenbank:** Passives Repository
- **Blackboard:** Aktives Repository
- **Bemerkung 1:** Ein Blackboard System sendet Nachrichten über geänderte Daten an interessierte Subscriber des Systems. Dieser Mechanismus ist sehr flexibel und skalierbar, da Blackboard Systems die Datenhaltung von den angeschlossenen Clients trennen.
- **Bemerkung 2:** Die Trennung zwischen passiven Datenzentrierten Systemen und aktiven Datenzentrierten Systemen ist fliessend, da heute jede Datenbank mit Triggern sehr einfach in ein Blackboard System umgewandelt werden kann

Beispiel

Siehe Script

Das Pattern

"An object model of the domain that incorporates both behavior and data."



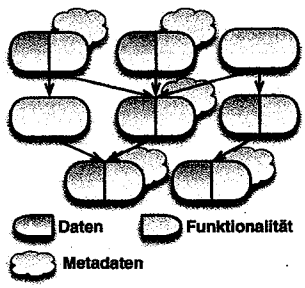
Anwendung von bewährten OO-Techniken für die Implementation der Geschäftslogik:
Kapselung, Vererbung, Polymorphie, Design Patterns ...

OO verspricht:
- bessere Skalierung bei zunehmender Komplexität
- bessere Erweiterbarkeit
- bessere Wartbarkeit
- weniger Redundanz

Zwei typische Herausforderungen

- **Peristence-Ignorance**
  - Testbarkeit
  - Wiederverwendbarkeit
  - Freiheit im OO-Design
- ORM mit JPA bietet einen Lösungsansatz
- **Zentralisierung der Validierungs-Logik**
  - Validierung auf verschiedenen Ebenen (UI, Business-Logik, DB)
- Bean Validation bietet einen Lösungsansatz

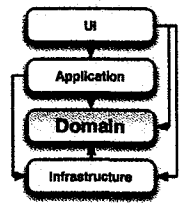
Klassen + Metadaten



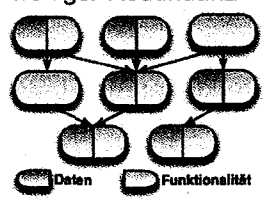
- Klassen werden mit Metadaten angereichert
- Metadaten sind verantwortlich für einen speziellen (infrastruktur) Concern
- Metadaten werden zu Laufzeit von einer Runtime konsumiert
- Kern-Logik bleibt entkoppelt von den Metadaten
- Klassisch in XML
- Seit JDK 5 als annotations
  - First level language construct, weniger Artefakte
  - Aber: Abweichung von POJO

Ausgangslage Domain Model

Anwendung von bewährten OO-Techniken für die Implementation der Geschäftslogik:
Kapselung, Vererbung, Polymorphie, Design Patterns ...



OO verspricht:
- bessere Skalierung bei zunehmender Komplexität
- bessere Erweiterbarkeit
- bessere Wartbarkeit
- weniger Redundanz



Der O/R-Mismatch

- Technische Ausprägungen des O/R-Mismatch:
- Typen-Systeme
  - Null
  - Datum/Zeit
- Abbildung von Beziehungen
  - Richtung
  - Mehrfachbeziehungen
- Vererbung
- Identität
  - Objekte haben eine implizite Identität
  - Relationen haben eine explizite Identität (Primary Key)
- Transaktionen

Der O/R-Mismatch

	DB	OO
Designziele	• Speichern und Abfragen von Daten • Speicherung unabhängig von konkreten Business-Problemen	• Vereinigung von Zustand und Verhalten • Kapselung, Modularisierung, Abstraktion • Modellierung konkreter Business-Probleme
Architektur-ansätze	• Client/Server: verteiltes System	• Objekte sind lokal und nicht verteilt
Abfrage/Zugriff	• Deklarative Abfragesprache • Beziehungen zwischen Daten müssen nicht explizit definiert sein	• Imperative Navigation entlang von Referenzen • Beziehungen zwischen Objekten müssen explizit definiert sein

Versprechen von O/R-Mapping

- Die Applikation wird von der DB entkoppelt
  - Applikationsentwickler muss kein SQL beherrschen
  - Das relationale Modell der Datenbank hat keinen Einfluss auf das OO-Design
- Automatische Persistenz
  - Automatisierte Abbildung der Objekte in die relationalen Strukturen
  - Der Applikationsentwickler muss sich nicht um die "low-level"-Details des CLI (z.B. connections) kümmern.
- Transparente Persistenz / Persistence Ignorance
  - Die Klassen des Domain-Models wissen nicht dass sie persistiert und geladen werden können und haben keine Abhängigkeit zur Persistenz-Infrastruktur.

Versprechen von O/R-Mapping

- Abstraktion
  - Abstraktion ist eines der wichtigsten Werkzeuge um Komplexität zu bewältigen
- Separation of Concerns
  - Bei der Applikationsentwicklung kann man sich ausschliesslich auf die Geschäftsprobleme konzentrieren
  - Infrastruktur-Probleme können separat gelöst werden und beeinflussen das Design und die Implementation der Geschäftslogik nicht.

Versprechen von O/R-Mapping Frameworks

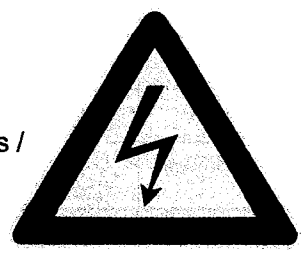
- Umsetzung der Versprechen von O/R-Mapping
- Einsatz von bewährten Patterns und Konzepten
- Einsparungen von viel Code
  - Manuell codierter DataAccess-Layer kann einen grossen Code-Anteil ausmachen
  - Referenzbeispiele zeigen Einsparungen um Faktor 7
- Strukturierung / Layerung des Codes vorgegeben

Pitfalls von O/R-Mapping Frameworks

- O/R-Mapping-Frameworks sind komplex und bieten viel Funktionalität.
  - Hibernate Core: 765k LOC, 206 Personen-Jahre, $11 Mio (Source: Ohloh.net)
- O/R-Mapping-Frameworks sind keine Rapid-Application-Development-Tools
- Der Einsatz eines O/R-Mapping-Frameworks hat Einfluss auf die Architektur und den Design der gesamten Applikation
- Die zugrundeliegenden Konzepte sollten verstanden sein.
- Gründliche Auseinandersetzung ist Voraussetzung für den erfolgreichen Einsatz eines O/R-Mapping-Frameworks

Konzeptionelle Probleme beim O/R-Mapping

- Folgende konzeptionelle Probleme sollten beim O/R-Mapping beachtet werden:
- Location Transparency
- Optimiertes SQL
- Mengen-Manipulationen
- Relationale Suche / Reports / OLAP



Location Transparency

- Alle Daten immer verfügbar
- Für die Applikation sollte es keinen Unterschied machen, ob die sich Daten im lokalen Speicher oder auf der entfernten Datenbank befinden.
- Wieviele Daten werden geladen?
  - Zuviele: Speicher, Bandbreite -> Ladezeit!
  - Zuwenige: Konstantes Nachladen -> Ladezeit!
  - Stichworte: Lazy-Loading / Eager-Loading

Dynamisch Generiertes SQL

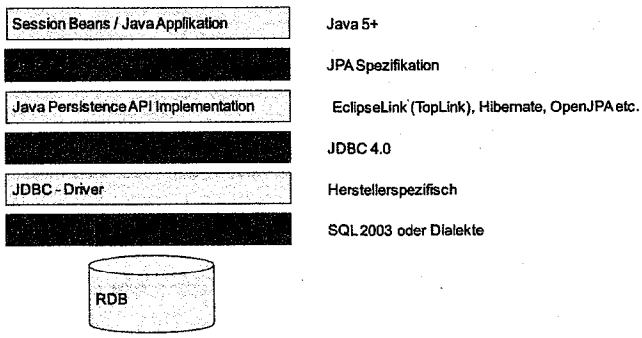
- SQL ist nicht so performant wie handgeschriebens, getuntes SQL
  - Dynamisch generiertes SQL muss auch nicht gewartet werden!
  - Ausgereifte Frameworks generieren gut optimiertes SQL
- Performance
  - Stored Procedures sind nicht performanter als dynamisches SQL!

Relationale Suche / Reports

Probleme:

- Karthesisches Produkt
  - In der DB sehr effizient umgesetzt
  - In der OO-Welt keine Entsprechung
- Die Struktur des Resultats der Anfrage existiert nicht in der OO-Welt
  - Es gibt keinen Typ mit den Attributen (Vorname, Nachname, Produktname)
  - Die relationale Welt erlaubt flexible (untypisierte) Sichten auf die Daten (Deklarative ad-hoc formulierte Anfragen)
  - Resultat einer Anfrage kann völlig entkoppelt sein von der Struktur wie die Daten gespeichert werden.
  - Die OO-Welt verlangt definierte, stark typisierte Strukturen
- Das ‚Flachdrücken' / ‚Denormalisieren' muss in der OO-Welt manuell ausgeführt werden
- Alle beteiligten Objekte müssen geladen werden (Performance!)

Technologiestack



Entity Überblick

- Eine Entity ist persistierbar. Der Zustand kann in einer Datenbank abgespeichert und später wieder darauf zugegriffen werden
- Wie jedes andere Java Objekt hat eine Entity eine Objektidentität. Zusätzlich besitzt sie eine Datenbankidentität (Primary Key)
- In Zusammenhang mit der Datenbank werden die Entities transaktional verwendet. Die Erstellung, Änderung und das Löschen wird in einer Transaktion durchgeführt

Entity Metadata

- Kennzeichnung mit Annotation @Entity oder Mapping mit XML
- Klasse kann Basisklasse oder abgeleitet sein
- Klasse kann abstrakt oder konkret sein
- Serialisierbarkeit ist bezüglich Persistenz nicht erforderlich
- Anforderungen:
  - Standardkonstrukutor muss vorhanden sein.
  - Klasse darf nicht final, kein Interface und keine Enumeration sein und keine final-Methoden enthalten
  - Felder müssen private oder protected sein.
  - Zugriff von Clients auf Felder nur über get/set- oder Business-Methoden erlaubt.
  - Jede Entity muss einen Primärschlüssel (@Id) haben
- **Configuration by Exception / Conventions over Configuration**

Entity Manager, Beispiel

- Entity Manager erstellen

```
EntityManagerFactory emf =
Persistence.createEntityManagerFactory("jpa.emp.2011");
EntityManager em = emf.createEntityManager();
```

- Entity persistieren

```
Employee emp = new Employee(158);
em.persist(emp);
```

- Entitiy suchen

```
Employee emp = em.find(Employee.class, 158);
```

Entity Manager, Beispiel

- Entity verändern

```
em.getTransaction().begin();
emp.setSalary(emp.getSalary() + 1000);
em.getTransaction().commit();
```

- Entity löschen

```
em.getTransaction().begin();
em.remove(emp);
em.getTransaction().commit();
```

- Queries

```
Query q = em.createQuery("SELECT e FROM Employee e");
Collection emps = q.getResultList();
```


Persitence Unit

- Eine Persistence Unit ist eine logische Einheit von Entities. Sie wird beschrieben durch:
  - Einen Namen
  - Die zu dieser Unit gehörenden Entity-Klassen
  - Angaben zum Persistence Provider
  - Angabe zum Transaktionstyp
  - Angaben zur Datenquelle
  - Weitere Properties
  - Namen von XML O/R-Mapping Files
- Technisch wird die Beschreibung einer Persistence Unit in der Datei META-INF/persistence.xml abgelegt.
- Persistence Archive = JAR

Access Typ

- Für das Persistenz-Framework existieren zwei Zugriffspunkte auf die Daten einer Klasse

```
// Field Access
@Entity public class Employee {
  @Id private int id;
}

//Property Access
@Entity public class Employee {
  protected int id;
  @Id public int getId() {
    return id;
  }
}
```


Persistente Datentypen

- Erlaubt:
  - Alle primitiven Typen, String
  - Alle Wrapperklassen und serialisierbaren Typen (z.B. Integer, BigDecimal, Date, Calendar)
  - byte[], Byte[], char[], Character[]
  - Enumerations
  - Beliebige weitere Entity-Klassen
  - Collections von Entities, welche als Collection<>, List<>, Set<> oder Map<> deklariert sind.
- Nicht erlaubt:
  - Alle Arten von Arrays ausser die obgenannten
  - Collections von etwas anderem als Entities, also z.B. Wrapperklassen und andere serialiserbare Typen.

Lazy Fetching und Large Objects

- Lazy Field Loading

```
@Basic(fetch = FetchType.LAZY)
private String comments;
```

- Large Objects

```
@Basic(fetch = FetchType.LAZY)
@Lob
private byte[] picture;
```


Enumerations

- Enumerations können persistent sein. In der Datenbank wird entweder der Ordinalwert (Position) oder der Stringwert (Name der Konstante) abgelegt.

```
// Variante Ordinal
@Enumerated(EnumType.ORDINAL)
   protected MessageStatus status;

// Variante String
@Enumerated(EnumType.STRING)
protected MessageStatus status;
```

- **Vorsicht bei Änderungen an der Enumeration**

Transiente Attribute

- Attribute können von der Persistierung ausgeschlossen werden
- Entweder mittels transient:

```
transient private String translatedName;
```

- oder wenn das Attribut serialisiert werden soll mittels Annotation:

```
@Transient
private String translatedName;
```


Entity Identity - Primärschlüssel

- Jede Entity-Klasse muss einen mit @Id bezeichneten Primärschlüssel besitzen.
- Eine Id kann von folgenden Typen sein:
  - Primitive Java Typen: byte, int, short, long, char
  - Wrapper Klassen: Byte, Integer, Short, Long, Character
  - Array von primitiven Typen oder Wrapper Klassen
  - java.lang.String
  - java.math.BigInteger
  - Zeittypen: java.util.Date, java.sql.Date
- Floating Point Typen sind ebenfalls erlaubt, aber sind aufgrund der möglichen Rundungsfehler nicht zu empfehlen

Primärschlüssel Generierung

- Primärschlüssel können in Zusammenarbeit mit der Datenbank generiert werden. Beispiel:

```
@Entity public class Employee {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  public Integer id;
}
```

- Stragegien sind Identity, Table, Sequence und Auto

Eine Parent-Child Beziehung

```
@Entity
public class Employee {
   …
   @ManyToOne
   private Department department;
   …
```

Mapping der many-to-one Beziehung

```
@Entity
public class Department {
   …
   @OneToMany
   @JoinColumn (name="department_id")
   private Set<Employee> employees =
              new HashSet<Employee>();
   …
```

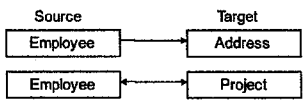
Mapping der one-to-many Beziehung
- Field/Property muss ein Interface sein
- Achtung:
  - Unidirektionales one-to-many ohne Beziehungstabelle wird erst ab JPA2 unterstützt

```
@Entity
public class Department {
   …
   @OneToMany (mappedBy = "department")
   private Set<Employee> employees =
              new HashSet<Employee>();
   …
```

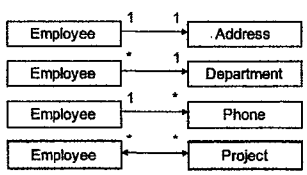
Mapping der bidirektionalen Beziehung
- JPA muss wissen, dass nur ein Foreign-Key für beide Richtungen existiert.

Collection Types

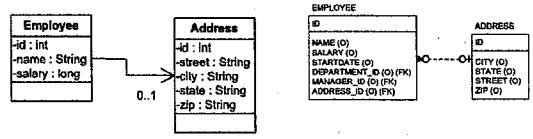
- Richtung
  - Unidirektional
  - Bidirektional



- Kardinalität
  - One-to-one
  - Many-to-one
  - One-to-many
  - many-to-many



one-to-one, unidirektional



Employee

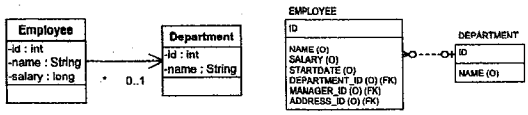
```
@OneToOne
private Address address;
```

entspricht:

```
@OneToOne
@JoinColumn(name="address_id", referencedColumnName = "id")
private Address address;
```

(JPA2 unterstützt auch one-to-one mit einer zusätzlichen Zwischentabelle)

many-to-one, unidirektional

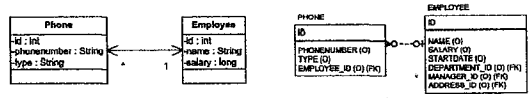


Employee

```
@ManyToOne
private Department department;
```

(JPA2 unterstützt auch many-to-one mit einer zusätzlichen Zwischentabelle)

one-to-many, bidirektional



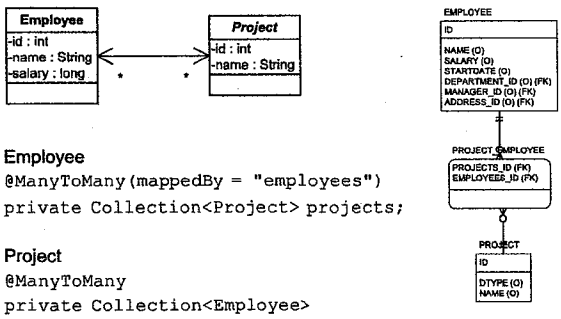
Phone

```
@ManyToOne(optional = false)
private Employee employee;
```

Employee

```
@OneToMany(mappedBy = "employee")
private Collection<Phone> phones;
```


many-to-many, bidirektional



Employee

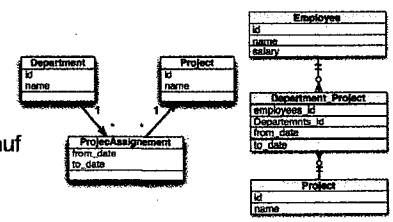
```
@ManyToMany(mappedBy = "employees")
private Collection<Project> projects;
```

Project

```
@ManyToMany
private Collection<Employee>
employees;
```


Many-To-Many Beziehungen

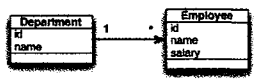
- „If you think that two objects share a simple many-to-many relationship, you haven't looked closely enough at the domain. There is a third object waiting to be discovered with attributes and a life cycle all its own." - Dierk König
- Oft sind weitere Daten auf der Zwischentabelle nötig
- Üblicherweise mappt man dann die Zwischentabelle auf eine eigene Entity

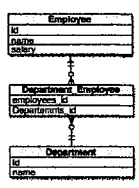


one-to-many, unidirektional

- Bei einer unidirektionalen one-to-many Beziehungen fehlt das mappedBy Element und das Target hat keine Rückbeziehung
- JPA verwendet in diesen Fällen ebenfalls eine Beziehungstabelle

```
@OneToMany
private Set<Employee> employees = new
HashSet<Employee>();
```





- JPA 2 spezifiziert die unidirektionale one-to-many Beziehung ohne Zwischentabelle.

```
@OneToMany
@JoinColumn (name="department_id")
private Set<Employee> employees =     new HashSet<Employee>();
```


Bidirektionale Beziehungen

- JPA verändert die Java-Semantik nicht!
- D.h. der korrekte Unterhalt von bidirektionalen Beziehungen ist Sache der Applikation!

```
Department taxes = new Department();
Employee john = new Employee();
taxes.getEmployees().add(john);
john.setDepartment(taxes);
```

Bidirektionale Beziehungen

School of Engineering

- **Best Practice: Convenience Methoden auf den Entities:**

```
@Entity
public class Department {
   ...
   @OneToMany
   private List<Employee> employees = new ArrayList<Employee>();

   public void addEmployee(Employee employee){
      if (employee == null)
        throw new IllegalArgumentException("Null employee");
      if (employee.getDepartment() != null)
          employee.getDepartment().getEmployees().remove(employee);

      getEmployees().add(employee);
      employee.setDepartment(this);
   }
   ...
}
```

Analog: removeEmployee() sowie Methoden auf Employee.

Verwendung von Collections

School of Engineering

- `java.util.Set`
  - **Eindeutig** (`Object.equals()`)
  - `@OneToMany`
    `private Set<Phone> phones;`
- `java.util.List`
  - **geordnet, kann sortiert werden**
  - `@OneToMany`
    `@OrderBy("phonenumber ASC")`
    `private List<Phone> phones;`
- `java.util.Map`
  - **Key/Value Paare**
  - `@OneToMany`
    `@MapKey(name = "phonenumber")`
    `private Map<String, Phone>`
    `phones;`

JPA 2:
Persistenter Index

```
@OneToMany
@OrderColumn(name="index")
private List<Phone> phones;
```


Lazy- und Eager-Loading

School of Engineering

Beziehungen werden transparent (nach)geladen:

```
EntityManager em = ...
Department foreignAffairs = em.find(Department.class,
123);
foreignAffairs.getEmployees().iterator().next();
```

- **Default bei one-to-one und many-to-one**
  - `FetchType.EAGER`
- **Default bei one-to-many und many-to-many**
  - `FetchType.LAZY`
- **Defaultverhalten kann übersteuert werden. z.B.**
  - `@OneToMany(fetch = FetchType.EAGER)`
    `private Set<Phone> phones;`

Speichern und Löschen von Beziehungen

School of Engineering

```
Department taxes = new Department();
Employee john = new Employee();
taxes.addEmployee(john);
Employee jane = new Employee();
taxes.addEmployee(jane);

em.persist(taxes);
em.persist(john);
em.persist(jane);
em.flush();
```

- **Jede Entity hat einen eigenen, unabhängigen Lifecycle!**
- **IllegalStateException wenn vergessen wird, eine assoziierte Entity zu persistieren.**

```
for (Employee empl :
taxes.getEmployees()){
    em.remove(empl);
}
em.remove(taxes);
em.flush();
```

- **Delete all entities individually**

Transitive Persistenz

School of Engineering

- **Persistenz wird von JPA propagiert auf assoziierte Entities.**

```
CascadeType {ALL, PERSIST, MERGE, REMOVE, REFRESH, DETACH};
```

```
@OneToMany (mappedBy = "department",
          cascade = CascadeType.ALL)
private Set<Employee> employees =
new HashSet<Employee>();
```

- **Kaskadierung wird auf der Assoziation konfiguriert**

```
Department taxes = new Department();
Employee john = new Employee();
taxes.addEmployee(john);
Employee jane = new Employee();
taxes.addEmployee(jane);

em.persist(taxes);
em.flush();
```

- **Speichern eines Parents speichert auch alle Kinder**

```
em.delete(taxes);
em.flush();
```

- **Löschen eines Parents löscht auch alle Kinder.**

Orphan Deletion

School of Engineering

```
Phone phone1 = …
Employee john = em.find(Employee.class, 123);
john.getPhones().remove(phone1);
em.flush();
```

- **Child wird nicht gelöscht!**

Entfernen eines Kindes aus der Collection des Parents setzt nur den Foreign Key auf der Kind-Tabelle auf NULL.
- **FK Constraint Verletzung möglich**
- **Das Kind ist nun "orphaned"**

- **In JPA 1 muss das Löschen von Orphans explizit in der Applikation erfolgen.**
- **JPA 2 unterstützt das automatische Löschen von Orphans**

```
@OneToMany(cascade=ALL, mappedBy="customer", orphanRemoval="true)
public Set<Order> getOrders() { return orders; }
```

Kontext Typen

- PersistenceContextType.TRANSACTION
  - Standard im Java EE Umfeld
  - Lesender und schreibender Zugriff nur innerhalb der Transaktion.
  - Gelesene Objekte sind nach der Transaktion im Zustand detached
  - Wiedereinkopplung in eine Transaktion mit merge()
- PersistenceContextType.EXTENDED
  - Standard im Java SE Umfeld
  - Alle Objekte sind lesend und schreibend zugreifbar
  - Modifikationen finden lokal statt
  - Effekt von persist(), remove() usw. wird aufbewahrt
  - Propagation von Efffekten und Änderungen in die DB aber nur, wenn nachträglich begin()/commit() ausgeführt wird

Objektverwaltung

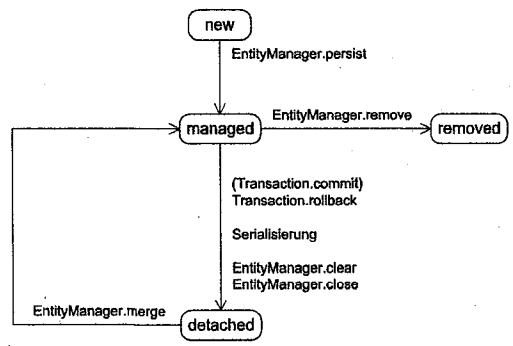
- Der Transfer von Objekten von und zur Datenbank erfolgt automatisch:
  so spät wie möglich → Lazy Access
- Der Transfer von Objekten von und zur Datenbank kann manuell erzwungen werden → synchron zum Aufruf
- Selbstverständlich gilt ein Transaktionsmodell: Der Zugriff auf Objekte erfolgt ab Beginn der Transaktion, die Synchronisation mit der Datenbank wird spätestens beim Commit abgeschlossen und unterliegt der ACID-Regel (*Atomicity, Consistency, Isolation* und *Durability*)
- Auf Objekte kann auch ausserhalb von Transaktionen zugegriffen werden, jedoch ohne Konsistenz- und Synchronisationsgarantie

Objektzustand

Objekte haben vier mögliche Zustände:

- **New**
  - Objekt ist neu erzeugt, hat noch keinen Zusammenhang mit der Datenbank und noch keine gültige ID.
- **Managed**
  - Das Objekt hat eine Entsprechung in der Datenbank. Änderungen werden vom Entity Manager automatisch getracked und mit der DB abgeglichen.
- **Detached**
  - Das Objekt hat eine Entsprechung in der Datenbank, wurde aber abgekoppelt. Der Zustand wird nicht mehr automatisch abgeglichen mit der Datenbank.
- **Removed**
  - Das Objekt existiert noch, ist aber zum Löschen markiert.

Zustände und Übergänge



Entity persistieren

- Mit persist() wird eine neue Entity vom EntityManager verwaltet

```
Department dept = em.find(Department.class, deptId);
Employee emp = new Employee();
emp.setId(empId);
emp.setName(empName);
emp.setDepartment(dept);
dept.getEmployees().add(emp);
em.persist(emp);
```

- Die Methode contains() kann geprüft werden ob eine Entity managed ist

Objekt ID

- Ein neues Objekt bekommt erst eine ID, wenn es das erste Mal physisch in die Datenbank transportiert wird

```
Employee emp = new Employee();
em.persist(emp);
System.out.println(emp.getId()); // null

em.flush(); oder em.getTransaction().commit();
System.out.println(emp.getId()); // gültige ID, z.B. 1
```

Kaskadierte Persistenz (1)

- Kaskadierte Persistenz heisst: Alle von einem persistenten Objekt aus erreichbaren Objekte sind ebenfalls persistent

```
Employee employee = new Employee();
em.persist(emp);
Address address = new Address();
employee.setAddress(address);
```

- Die Kaskadierung muss deklariert werden:
  - PERSIST
  - MERGE
  - REMOVE
  - REFRESH
  - ALL

Kaskadierte Persistenz (2)

- Die Kaskadierung kann für das Erstellen und das Löschen der Persistenz separat eingestellt werden

```
public class Employee {
  @OneToOne( cascade={CascadeType.PERSIST,
                      CascadeType.REMOVE } )
  private Address address;
  ...
}
```

- Die Kaskadierung bezieht sich nun auf die persist() und die remove()-Methode.

Orphan Removal (JPA 2.0)

- Sollen abhängige Kindelemente bei to-many Beziehungen ebenfalls gelöscht werden, kann dies seit JPA 2.0 ebenfalls deklariert werden:

```
@OneToMany(cascade=ALL,
           mappedBy="customer",
           orphanRemoval=true)
private Set<Order> orders;
```


Entity suchen

- Mit find() kann eine Entity über ihren Primary Key gefunden werden
- Die gefunden Entity wird kommt automatisch in den Zustand managed
- Da find() über den Primary Key sucht, kann diese Methode vom Persistence Provider optimiert werden und unter Umständen einen Datenbankzugriff vermieden werden
- Soll eine one-to-one oder many-to-one Reference auf eine bestehende Entity gebildet werden, kann getReference() verwendet werden um das vollständige Laden der Target-Entity zu verhindern

Einlesen

- Der Objektzustand wird beim ersten Zugriff auf das Objekt eingelesen.
- Wenn FetchType.EAGER gesetzt ist, werden referenzierte Objekte ebenfalls mitgeladen.
- Wenn FetchType.LAZY gesetzt ist, werden referenzierte Objekte beim ersten Gebrauch eingelesen.
- Der Objektzustand wird nie automatisch aufgefrischt, nur via die EntityManager.refresh()-Methode.
- Eine neue Transaktion führt nicht automatisch zum erneuten Einlesen bestehender Objekte.

Entity löschen

- Mit remove() wird dem EntityManager mitgeteilt, dass diese Entity gelöscht werden kann

```
Employee emp = em.find(Employee.class, empId);
em.remove(emp);
```

Objektzustand nach Commit

- Wenn der Persistence Context EXTENDED ist, bleibt ein Objekt im Zustand managed nach dem Commit.
  - Änderungen nach dem Commit werden aufbewahrt und im Rahmen der nächsten Transaktion in die Datenbank übernommen.
- Wenn der Persistence Context TRANSACTION ist, geht ein Objekt in den Zustand detached über nach dem Commit.
  - Änderungen müssen mit EntityManager.merge() innerhalb der nächsten Transaktion wieder eingekoppelt werden.

Objektzustand nach Rollback

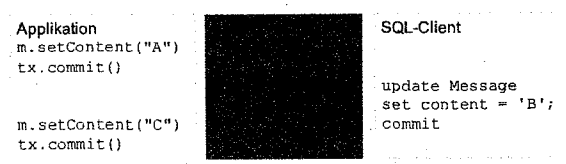
- Nach einem Rollback ist jedes noch vorhandene Objekt im Zustand detached. Die Belegung der Felder wird durch den Rollback nicht geändert, jedoch der Zustand in der Datenbank
- Vorsicht vor möglichen Inkonsistenzen
- Objekte via Methoden des Entity Manager neu laden: find(), getReference(), createQuery() usw.

Persistence Context aufräumen

- Ab und zu kann es vorkommen, dass der Persistence Context gelöscht werden soll
- Dies kann mit der Methode clear() des EntityManager erreicht werden
- Alle Entities kommen in den Zustand detached
- Vorsicht: enthält der Persistence Context Änderungen welche noch nicht mit commit() gespeichert wurden, gehen diese verloren

Synchronisation mit der Datenbank

- Das Zurückschreiben findet zum Commit-Zeitpunkt oder explizit mit der flush() Methode statt.
- Das Zurückschreiben beinhaltet kein Refresh allfälliger Änderungen in der Datenbank in der Zwischenzeit -> Lost Update!
- Das Zurückschreiben betrifft nur Änderungen, nicht ungeänderte Daten.



Foreign Key Constraints

- Änderungsoperationen erzeugen oft Konflikte mit Fremdschlüssel-bedingungen, wenn Datensätze in der falschen Reihenfolge gelöscht werden.
- Oracle erlaubt die Constraints DEFERABLE zu definieren, damit diese erst zum Commit Zeitpunkt geprüft werden:

```
ALTER TABLE EMPLOYEE ADD CONSTRAINT EMPLOYEE_ADDRESS_FK
  FOREIGN KEY (ADDRESS_ID)
  REFERENCES ADDRESS(ID)
  DEFERRABLE INITIALLY DEFERRED;
```


Entity über die ID laden

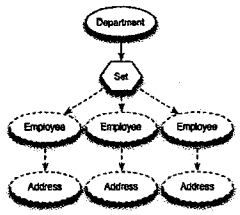
- Der EntityManager stellt zwei Möglichkeiten zur Verfügung: find() und getReference()

```
EntityManager em = …
Integer id = 1234;
Employee john = em.find(Employee.class, id);
```

- Falls die Entität bereits im Persistence Context geladen ist, so wird kein DB-Query ausgeführt.
- find(): Wenn sich die Entity noch nicht im Persistence Context befindet, so wird sie von der DB geladen.
  - Resultat is null, falls die Entity nicht existiert
- getReference(): Wenn sich die Entität noch nicht im Persistence Context befindet, so wird ein Proxy zurückgegeben. Es wird vorerst kein DB-Query ausgeführt. Dieses erfolgt erst wenn auf die Entität zugegriffen wird.
  - EntityNotFoundException erst beim Zugriff, falls die Entity nicht existiert.

Navigation des Objektgraphen

- Ausgehend von einer Entity kann ein Objektgraph traversiert werden. Dabei lätdt JPA transparent alle notwendigen Daten von der DB.
  - Dieses Feature wird "Lazy Loading" genannt
  - Die Entities müssen persistent und der EntityManager muss offen sein
  - Dies ist ein mächtiges Feature, birgt aber auch Gefahren



Queries mit JPQL

- JPQL ist eine mächtige Abfragesprache basierend auf dem Entitätenmodell:
  - Stark an SQL angelehnt
  - Unabhängig von der darunterliegenden Datenbank
  - Abfragen basieren auf dem Klassenmodell (Entitäten), nicht auf dem Datenmodell (Tabellen)
  - Unterstützt OO-Konstrukte wie Vererbung, Polymorphismus und Pfadausdrücke

```
String queryString =
"select e.address from Employee e where e.mainProject.name = 'JPA Kurs'";
Query query = em.createQuery(queryString);
List<Address> users = query.getResultList();
```


Verwendung von JPQL

- Typischerweise wird JPQL verwendet um Entities zu laden. JPQL unterstützt aber auch andere Szenarien:
  - Abfrage von skalaren Werten (Projektionen oder Aggregationen)
  - Bulk Updates und Deletes
  - Reporting Queries: Rückgabe von Daten-Tupels, Nutzung von Gruppierungs- und Aggregationsfunktionen der DB
  - Constructor Expressions: Abfüllen von beliebigen Objekten (nicht notwendigerweise Entities)
- JPQL kann entweder in Dynamischen Queries oder in Named Queries verwendet werden.

JPQL Sprach-Features

- JPQL ist eine sehr mächtige und flexible Abfragesprache. Hier nur einige Features:
- JOINS und Subqueries (IN, EXISTS)
- Aggregatsfunktionen (AVG, COUNT, MIN, MAX, SUM)
- GROUP BY und HAVING
- Funktionen (LOWER, ABS, TRIM ...)
- LIKE
- Collection-Abfragen: IS EMPTY, MEMBER
- ANY, ALL, SOME,

Fetching & Lazy Loading

- Die Idee von Lazy Loading ist es, die Daten erst dann von der DB zu laden, wenn sie auch wirklich in der Applikation benötigt werden.
  - Das Laden sollte für den Client transparent sein
  - Dem Programmierer wird viel Arbeit erspart
- Nachteile:
  - Traversieren eines Objekt-Graphen kann in vielen einzelnen DB-Queries resultieren
  - "N+1 select problem": Für eine Parent-Child Beziehung wird für jedes Kind ein eigenes DB-Query abgesetzt
- Das Gegenteil von Lazy Loading ist Eager Loading

Vorteile und Nachteile von JPQL

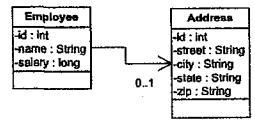
- Vorteile
  - Sehr mächtig und flexibel
  - Stark an SQL angelehnt
- Nachteile
  - JPQL ist eine embedded Language die in Java mittels Strings verwendet wird.
    - Keine Überprüfung beim Kompilieren, keine Typ-Sicherheit
  - Flexible Komposition eines Queries ist nicht elegant möglich (String-Manipulation)
  - Für nicht-triviale Anwendungen ist SQL Knowhow und Verständnis des Datenmodels ist notwendig

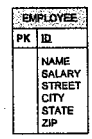
Criteria API in JPA 2

- Mit der Criteria API wird in JPA 2 eine Objekt-Orientierte Schnittstelle zum programmatischen Erstellen von Queries standardisiert.
- Vorteile:
  - Dynamisches Erstellen von Queries (Komposition)
  - Keine String-Manipulation notwendig
  - OO-Konstrukte zum Erstellen komplexer Queries
  - Gewisse Typsicherheit

Embedded Objects

- Komposition: Mutterobjekt mit eingebetteten Objekten
- Eingebettete Objekte haben keine eigene Identität
- Mutterobjekt und eingebettete sind in derselben Tabelle abgelegt





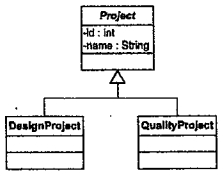
Embedded Objects, Beispiel

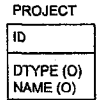
```
@Embeddable
public class Address {
  private String street;
  private String city;
  private String state;
  private String zip;
}

@Entity
public class Employee {
  @Id private int id;
  private String name;
  private long salary;
  @Embedded private Address address;
}
```


Vererbung

- Vererbungshierarchien können problemlos verwendet und abgebildet werden.
- Klassen können abstrakt oder konkret sein.
- Alle Klassen in der Vererbungshierarchie müssen den Primärschlüssel der Basisklasse verwenden (erben).
- Es gibt vier Mappingstrategien auf die Datenbank:
  - Eine einzige Tabelle für die gesamte Verbungshierarchie
  - Eine Tabelle für jede konkrete Klasse
  - Eine Tabelle für jede Klasse
  - Mapped Superclass

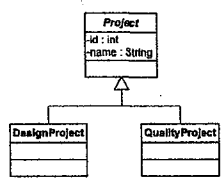
SINGLE_TABLE

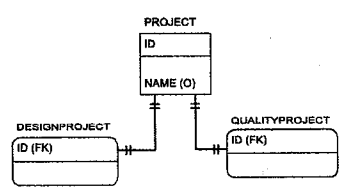





```
@Entity
@Inheritance public class Project

@Entity public class DesignProject extends Project

@Entity public class QualityProject extends Project
```

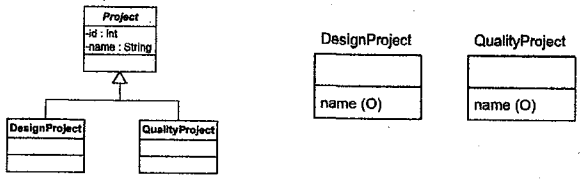

JOINED






```
@Entity @Inheritance(strategy=InheritanceType.JOINED)
public class Project

@Entity public class DesignProject extends Project

@Entity public class QualityProject extends Project
```

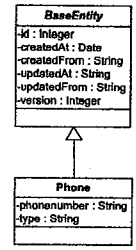
TABLE_PER_CLASS (Optional)




```
@Entity
@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS)
public class Project

@Entity public class DesignProject extends Project

@Entity public class QualityProject extends Project
```


MAPPED_SUPERCLASS

```
// Klasse BaseEntity
@MappedSuperclass
public abstract class BaseEntity {
  @Id @GeneratedValue
  protected Integer id;
  protected Integer version;
  protected Timestamp createdAt;
  protected String createdFrom;
  protected Timestamp updatedAt;
  protected String updatedFrom;

// Klasse Phone
@Entity
public class Phone extends BaseEntity
```



Was sind Bedingungen?

- Anforderungen
  - Restriktionen an Bean, Feld oder Property
  - z.B. Not Null, 0 - 10, gültige E-Mail-Adresse etc.
- Nutzen
  - User über Fehler informieren
  - Sicherstellen, dass eine Komponente richtig funktioniert
  - Ungültige Daten in der Datenbank verhindern

Probleme

- Duplizierung
  - Mehrfache Deklaration der selben Bedingung
  - Doppelter Code
  - Risiko der Inkonsistenz
- Überprüfung zur Laufzeit
  - Nicht alle Bedingungen können überall geprüft werden
  - Unterschiedliche Semantik

Die Lösung

- Einheitliche Form
  - Eine Sprache für alle
  - Basierend auf dem Domain Modell (JavaBeans)
- Einheitliche Validierung
  - Ein Framework
  - Eine Implementierung
- Brücke zu anderen Technologien
  - API um auf Bedingungen zuzugreifen

Annotations

- Deklaratation
  - Auf Ebene Bean, Feld oder Getter
- Eigenschaften
  - Message
  - Groups
  - Spezifische Parameter
- Deklarationen werden vererbt
  - Klasse
  - Interface

Mögliche Bedingungen

- Vordefinierte
  - @Null / @NotNull
  - @AssertTrue / @AssertFalse
  - @Min / @Max / @Size / @Digits
  - @Past / @Future
  - @Patterns / @Pattern
- Eigene, z.B.
  - @Email
  - @CreditCard
  - @Zugnummer
  - ...

Constraint Metatdata Request API

- Bietet Zugriff die Metadaten
  - Z.B. Liste aller Bedingungen einer
- Nützlich für Schnittstellen zu anderen Technologien
  - Persistence (DDL)
  - Presentation layer (Javascript™ programming language)
- Tools

SOLID auf einen Blick

- **Single Responsibility Principle (SRP):** Jedes Modul soll genau eine einzige Verantwortung übernehmen.
- **Open Closed Principle (OCP):** Offen für Erweiterung, geschlossen für Änderung
- **Liskov Substitution Principle (LSP):** Eine Instanz einer Klasse soll durch eine Instanz einer abgeleiteten Klasse ersetzt werden können, ohne dass sich das Verhalten ändert.
- **Interface Segregation Principle (ISP):** Clients sollen nur von Schnittstellen abhängig sein, die sie auch tatsächlich benutzen
- **Dependency Inversion Principle (DIP):** Umkehr der Abhängigkeiten - Schnittstellen statt konkrete Klassen

Vollständige Definition SRP

Uncle Bob: Die Stärke der wechselseitigen Beziehung der einzelnen Bestandteile eines Modules / einer Klasse untereinander

Das Prinzip der **einzigen Verantwortung** bedeutet, dass jedes Modul – jede Klasse – soll **genau eine Verantwortung** übernehmen.

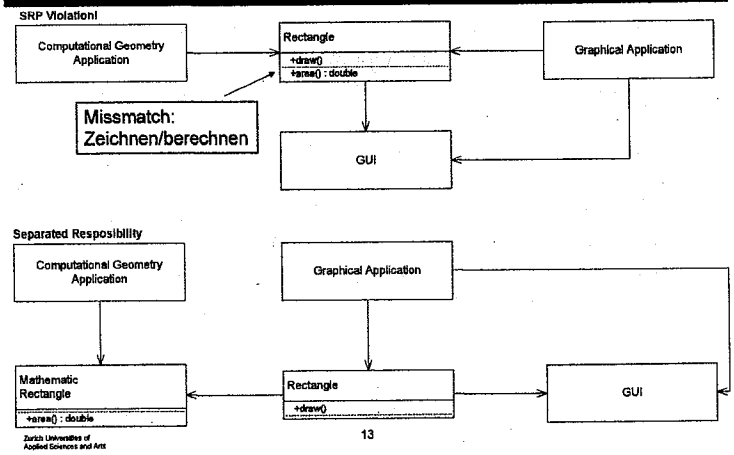
Verantwortung wird in diesem Fall über den Grund für eine Änderung definiert. Falls es mehr als einen Grund gibt, eine Klasse zu verändern, so verletzt die Klasse bereits dieses Prinzip. Hat also mehr als eine einzige Verantwortung.

Lose Kopplung

Um lose Kopplung zu erreichen, sind Module oder Klassen so zu gestalten, dass sie im Minimum die drei Kohäsionsarten Funktionale, Sequentielle und Informationsfluss Kohäsion erfüllen.

- **Funktionale Kohäsion:** Lediglich eine einzige Variable wird überhaupt verändert
- **Sequentielle Kohäsion:** Es werden zwei Variablen verändert, aber beide Modifikationen resultieren lediglich in einen Output.
- **Informationsfluss Kohäsion:** Die Änderung mehrerer Attribute wird von einem einzigen Input veranlasst

Beispiel SRP



Vollständige Definition OCP

Betrand Meyer: Module sollten sowohl offen, als auch geschlossen sein

Eine Klasse muss zwei primäre Eigenschalten aufweisen:

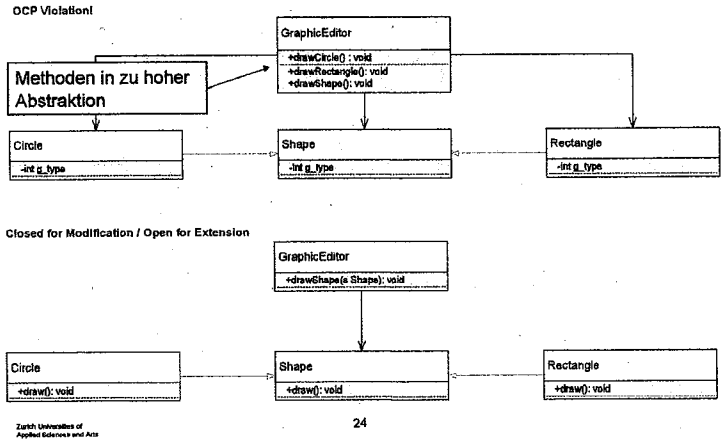
- **Offen für Erweiterungen:** Das Verhalten einer Klasse kann erweitert werden. Diese Eigenschaft erlaubt das nachträgliche verändern der Verhaltens oder das Hinzufügen von neuen Eigenschaften, falls entsprechende Anforderungen dies verlangen.
- **Geschlossen für Änderungen (JDK 9: sofern sie von aussen erreichbar ist):** Es können keine Änderungen an der Klasse durchgeführt werden. Es ist nicht erlaubt, an die Sourcen der Klasse zu verändern.

Abstraktion

Definition: Unter Abstraktion versteht man Verallgemeinerung, das Absehen vom Besonderen und Einzelnen, das Loslösen vom Dinglichen. Abstraktion ist das Gegenteil von Konkretisierung

- **Klassifizierung:** Charakterisierung der Gemeinsamkeiten von Individuen durch Typen oder Klassen
- **Komposition:** Zusammensetzen einer Menge zusammenhängender Individuen zu einem Ganzen
- **Generalisierung**: Verallgemeinerung der Merkmale einer Menge ähnlicher Typen
- **Benutzung**: Nutzung von Leistungen Dritter durch ein Individuum zwecks Erbringung eigener, höherwertiger Leistungen

Beispiel OCP



Vollständige Definition LSP

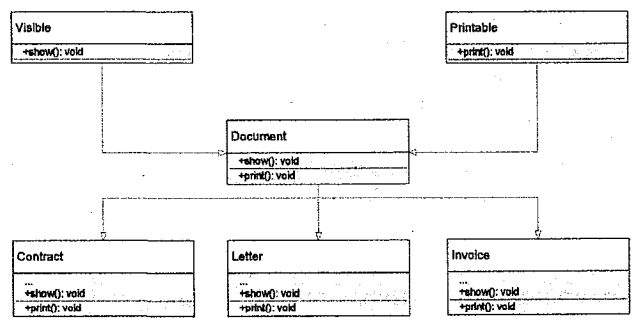
Barbara Liskov: Es sei p(x) eine beweisbare Eigenschaft von Objekten x des Typs T. Dann soll p(x) auch für Objekte y des Typs S wahr sein, wenn S ein Subtyp von T ist.

Das Liskovsche Ersetzbarkeitsprinzip (Substitutionsprinzip) schreibt vor, dass eine Instanz einer Klasse durch eine Instanz einer abgeleiteten Klasse ersetzt werden können soll, ohne dass sich das Verhalten ändert. Im Klartext heisst das, dass sich Subtypen einer bestimmten Klasse ähnlich verhalten müssen, damit eine Anwendung stabil bleibt.

Konsequenzen

- **Vorbedingungen:** Eine Unterklasse kann die Vorbedingungen für eine Operation, die durch die Oberklasse definiert wird, einhalten oder abschwächen. Sie darf die Vorbedingungen aber nicht verschärfen.
- **Nachbedingungen:** Eine Unterklasse kann die Nachbedingungen für eine Operation, die durch eine Oberklasse definiert werden, einhalten oder erweitern. Sie darf die Nachbedingungen aber nicht einschränken.
- **Invarianten:** Eine Unterklasse muss dafür sorgen, dass die für die Oberklasse definierten Invarianten immer gelten.

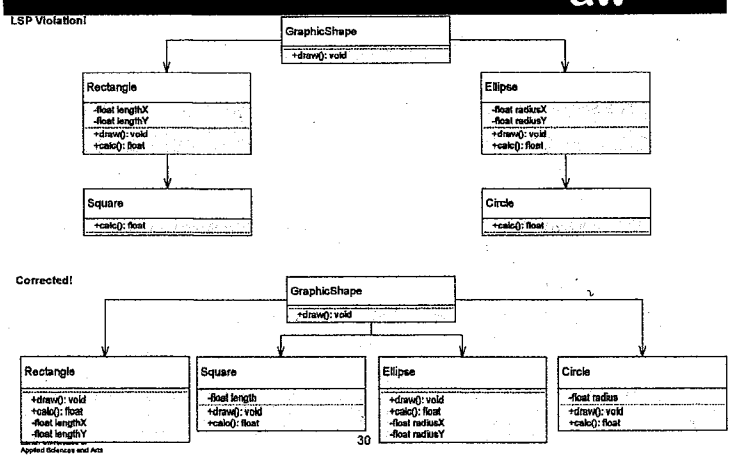
Vererbung



Polymorphismus

- **Dynamische Polymorphie:** Einer Variablen können Objekte verschiedenen Typen zugeordnet werden. Der Typ der Variablen fungiert dabei als Schnittstelle. Implementieren die verschiedenen Objekte dieselbe Methode – jedoch mit verschiedenem Verhalten – so kann zur Laufzeit (abhängig vom Typ des Objektes) die entsprechende Methode aufgerufen werden.
- **Statische Polymorphie (oder Überladung):** Der Aufruf einer Operation wird aufgrund eines konkreten Typs einer Variable oder einer Konstanten auf eine bestimmte Methode abgebildet. Die Auswahl des Aufrufs einer Methode erfolgt dann beispielsweise aufgrund des Parametertyps oder der Anzahl der Parameter.

Beispiel LSP

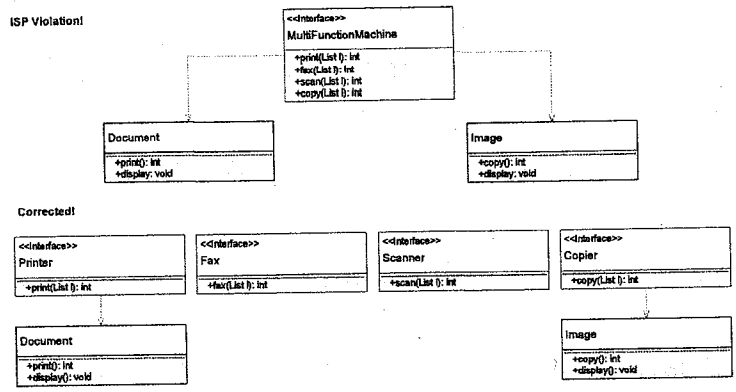


Vollständige Definition ISP

Uncle Bob: Die Nutzer einer Klasse / eines Objektes sollten nicht von Schnittstellen abhängig sein, die sie nicht selbst verwenden

Zur Einhaltung des Prinzips wird jede Schnittstelle wird darauf hin geprüft, ob all ihre Methoden von einem bestimmten Client verwendet werden. Falls dies nicht der Fall, ist zu prüfen, ob eine Trennung der Schnittstelle in mehrere einzelne Schnittstellen sinnvoll ist.

Beispiel ISP



Vollständige Definition DIP

Uncle Bob:

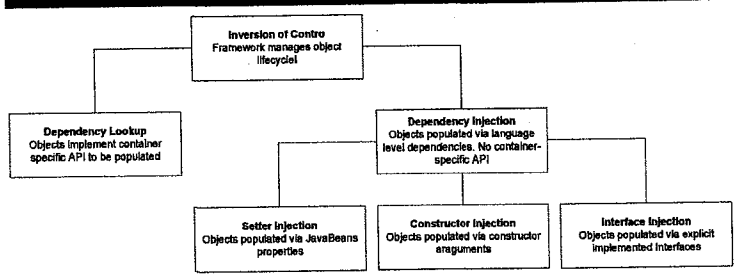
Module höherer Ebenen sollten nicht von Modulen niedriger Ebenen abhängen.
Sie sollten beide von Abstraktionen abhängen.

Abstraktionen sollten nicht von Details abhängen.

Details sollten von Abstraktionen abhängen

DIP wird als wichtigstes Design-Prinzip für die Entkopplung von Komponenten eines Objektorientierten Software-Systems angesehen. Es formuliert, wie Objekte über andere abhängige Objekte Bescheid wissen sollten. Und zwar lediglich über eine Abstraktion.

Dependency Injections



Dependency Injection (DI) bedeutet, dass Abhängigkeiten zwischen Klassen oder Packages von einer externen Instanz zugewiesen werden.

Beispiel DIP