



Hugbúnaðarverkefni 1 / Software Project 1

1. Introduction

HBV501G – Fall 2015

Matthias Book



HÁSKÓLI ÍSLANDS
VERKFRÆÐI- OG NÁTTÚRVÍSINDASVIÐ
ÍÐNAÐARVERKFRÆÐI-, VÉLAVERKFRÆÐI-
OG TÖLVUNARFRÆÐIDEILD

Matthias Book

- Dósent at Háskóli Íslands since October 2014
 - Office: Tæknigarður 208
 - Appointments by e-mail anytime
 - Contact: book@hi.is
- Research and teaching focus: Software Engineering
- Background
 - Studied Computer Science at Universities of Dortmund and Montana
 - Doctoral degree in Computer Science from University of Leipzig
 - Researcher and lecturer at Universities of Essen and Chemnitz
 - Research manager at software development company adesso

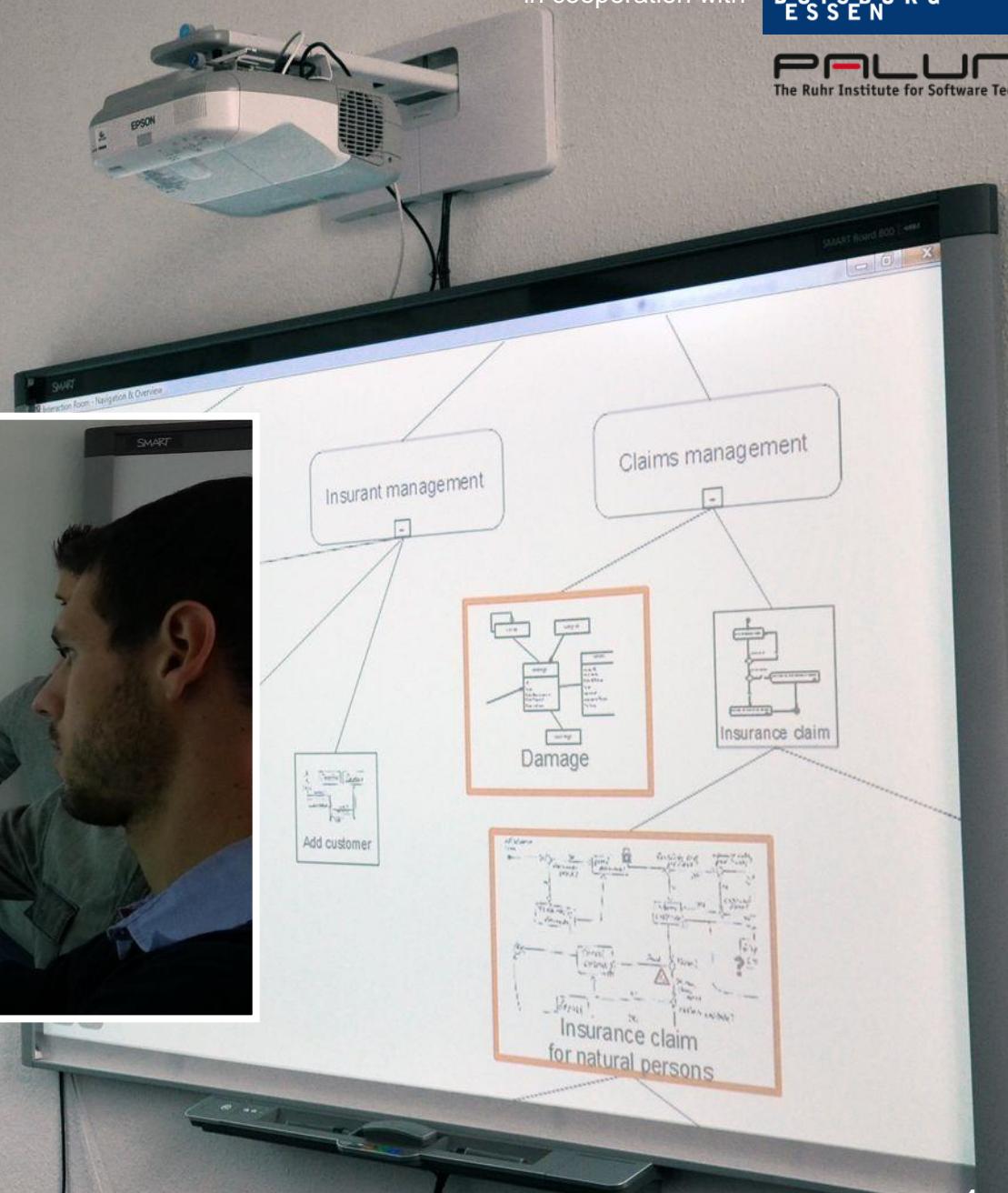
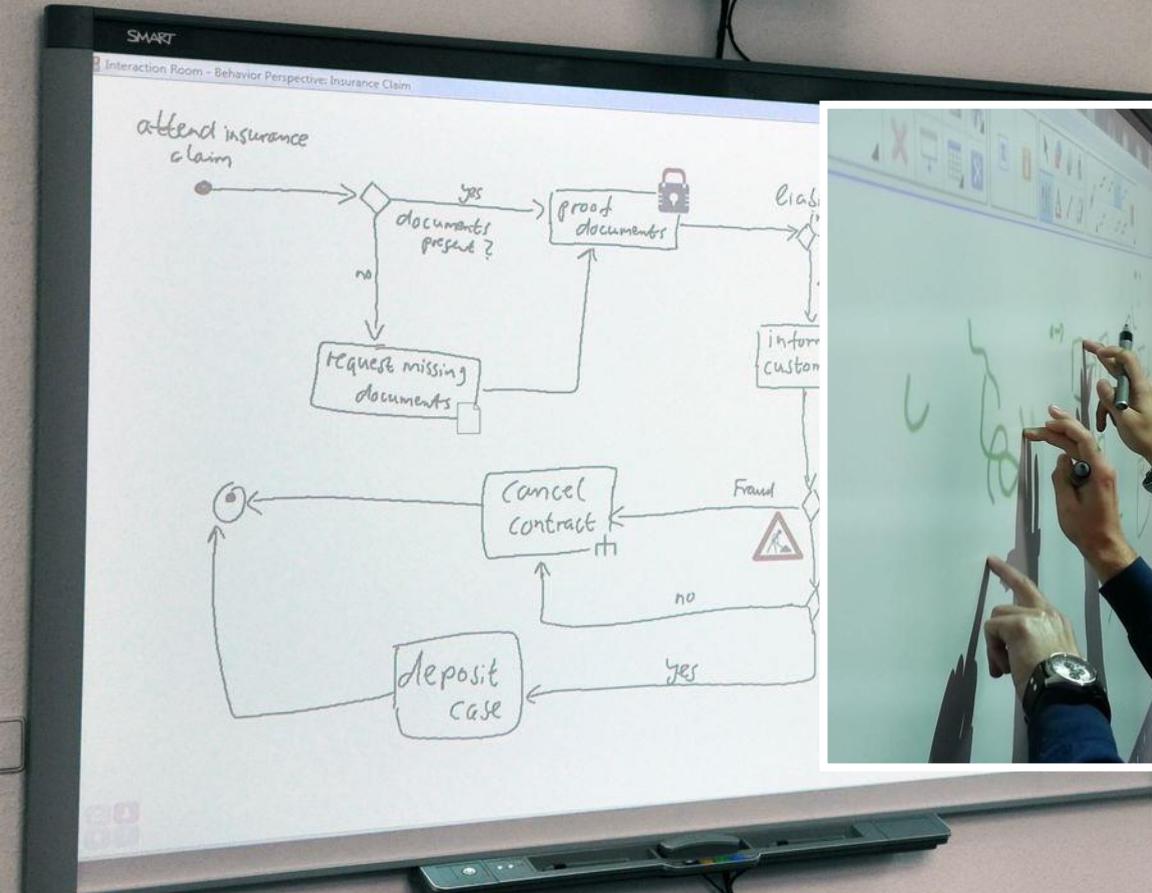


Research Interests

- **Interaction among software project stakeholders**
 - Facilitating effective communication between team members from heterogeneous backgrounds (business, technology, management...)
 - Identifying risks, uncertainties and value drivers early in a project, and focusing collaboration on these aspects rather than on business/technology trivia
- **Multi-modal interaction with large interactive displays**
 - Specifying and controlling user interactions with software systems through 2D and 3D gestures, voice commands etc.
- **Software engineering for mobile applications**
 - User experience, implementation challenges, ubiquitous computing
- **Software engineering in general**
- B.Sc., M.Sc., Ph.D. projects on these topics available – or suggest your own!

The Interaction Room

in cooperation with



Course Scope

- After a first taste of programming-in-the-large...
 - One simple approach to agile software development (HBV401G)
- ...now a more in-depth look at different software engineering paradigms:
 - **Plan-driven development and web engineering (HBV501G)**
 - Agile development and mobile software engineering (HBV601G)
- **Course aims:**
 - Learn about the different software engineering methods at your disposal for
 - requirements, estimation, architecture, design, integration, testing, management
 - Gather more practical experience with these approaches
 - in the context of two different technologies (web and mobile)
 - in the context of two different system landscapes (green-field and brown-field)
- Make well-informed method decisions and avoid pitfalls in your future industry projects



Are You in the Right Course?

- **Proper sequence:**

- HBV401G Software Development (prerequisite: HBV201G, TÖL101G, TÖL203G)
- **HBV501G Software Project 1** **(prerequisite: HBV401G)**
- HBV601G Software Project 2 (prerequisite: HBV501G)

- **Don't skip HBV401G before taking HBV501G!**

- Knowledge and project experience from HBV401G is expected
- HBV402G (Software Development A) is not equivalent preparation
- HBV501G and HBV601G are designed to be taken in sequence (continuous project)
- You won't have enough time in the spring semester to accomplish required project work for HBV401G and HBV601G in parallel



Recap: It All Starts With an Idea



A Typical Software Process



A Lose-Lose Situation



Matthias Book: Software Project 1



HÁSKÓLI ÍSLANDS



A Software Engineer's Most Important Skills

- Communication with people of different training, goals, expectations
- Communication on multiple levels of abstraction
- Creation and use of models and specifications
- Planning and coordinating work



Course Schedule

| Week | Thu: Team Consultations | Fri: Lectures | Sun: Assignments due | |
|------|-------------------------|----------------------------|---|----------|
| 1 | | Introduction | | |
| 2 | Phase 1: Inception | Requirements Engineering | Team Formation | (6 Sep) |
| 3 | Phase 1: Inception | Requirements Engineering | | |
| 4 | Project Management | Project Management | Project Vision, Development Plan (20 Sep) | |
| 5 | Phase 2: Elaboration | Requirements Engineering | | |
| 6 | Phase 2: Elaboration | Web Engineering | | |
| 7 | Phase 2: Elaboration | Web Engineering | | |
| 8 | Phase 2: Elaboration | Web Engineering | Software Architecture, Prototype (18 Oct) | |
| 9 | Phase 3: Construction | Object-Oriented Design | | |
| 10 | Phase 3: Construction | Object-Oriented Design | | |
| 11 | Phase 3: Construction | Object-Oriented Design | Design Model, Data Model | (8 Nov) |
| 12 | Phase 3: Construction | Plan-Driven Process Models | | |
| 13 | Phase 3: Construction | Plan-Driven Process Models | | |
| 14 | Project Presentations | Project Presentations | Beta Release, Presentation | (29 Nov) |



Course Structure

- **Lectures**
 - Friday 13:20-14:50, HT-103
- **Project team consultations**
 - Weekly meetings with tutors
 - to discuss requirements, design, questions
 - to present assignment deliverables
 - Thursday 08:30-11:30, V-152
 - Teams 1-10: 08:30-09:30
 - Teams 11-20: 09:30-10:30
 - Teams 21-30: 10:30-11:30
 - Please bring your laptop!
- **Project team meetings**
 - Scheduled freely among team members
- **Project teams**
 - 3-4 students per team
- **Project topics**
 - Your own project idea for a web application
 - e.g. a service, community, game, etc.
- **Basic architectural requirements**
 - Server-side back-end logic
 - developed in Java
 - Lightweight client-side front-end
 - running in a web browser



Some Project Ideas

- News aggregator
 - Educational game
 - Entertainment game
 - Consumer survey
 - Dictionary
 - Geo information
 - Recipe database
 - Image sharing
 - Playlist management
 - Scheduling
 - Community
 - Job market
 - Sports statistics
 - Workout tracker
 - Language learning
 - particularly: components for icelandiconline.is
 - Data tracking / survey creation / file mgmt.
- Project Spotlights from last semester:**
- 
-  **Corners**
 -  **FREGNIR**
 -  **SWAP**

Spotlight: Corners

Team:

- Edda Björk Konráðsdóttir
- Jóhanna Magnúsdóttir
- Kristín Fjóla Tómasdóttir
- Steinunn Friðgeirs dóttir

Categories

Math

Level 1

Level 2

Level 7

Level 8

Iceland

Paris

Bulgaria

Level 3

Level 4

Level 4

Level 5

Level 5

Blue

Purple

Navy

Orange

Yellow

Blue

Red

White

Red

White

Oh no! You lost!

Carl says:
Better luck next time!

Got it!

Swipe the background color in the middle to the matching color name in the corners!

4/9

Cyan

Olive

Magenta

Purple

Light gray

Cyan

Light gray

Cyan

White

1/9

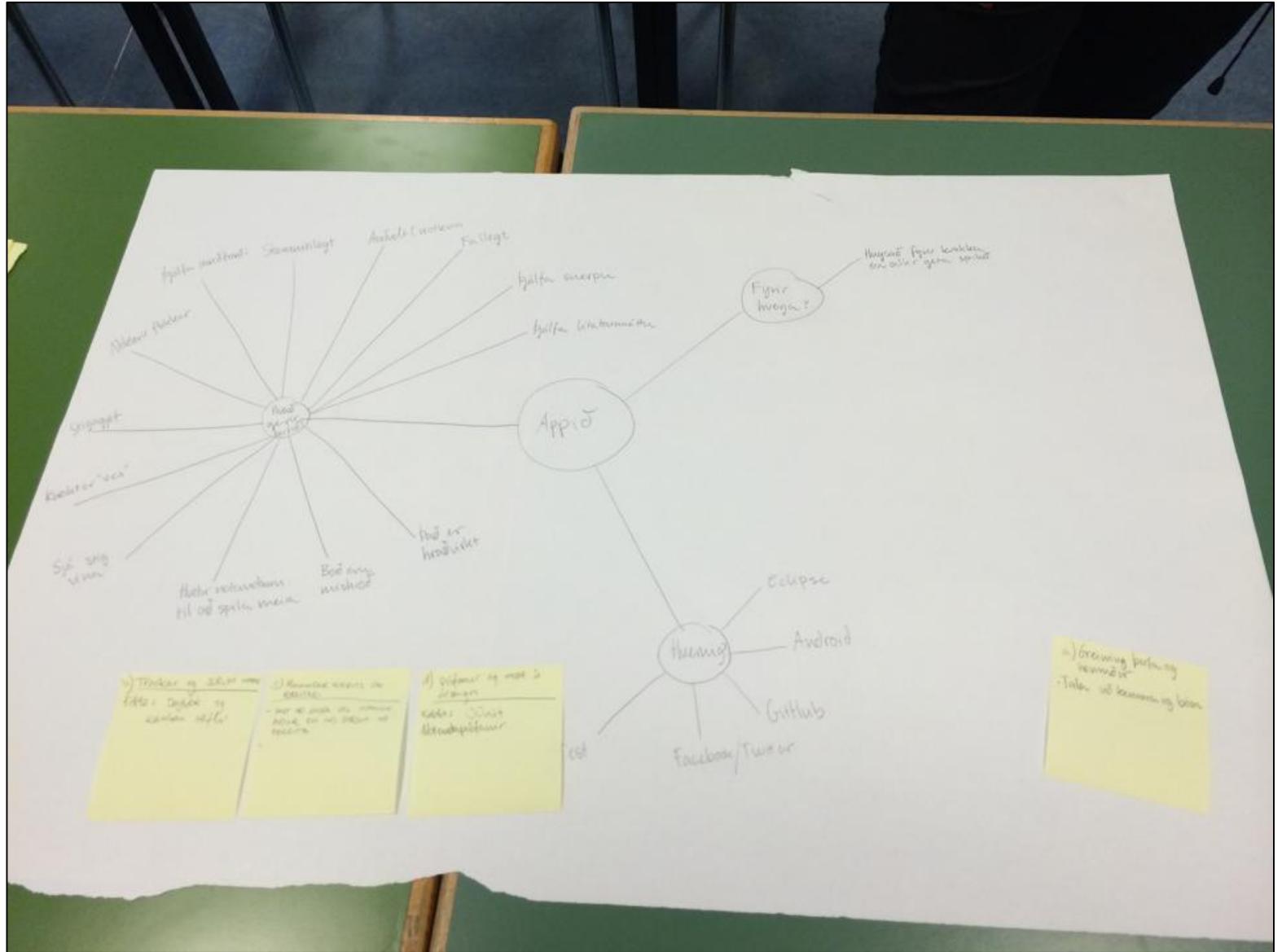
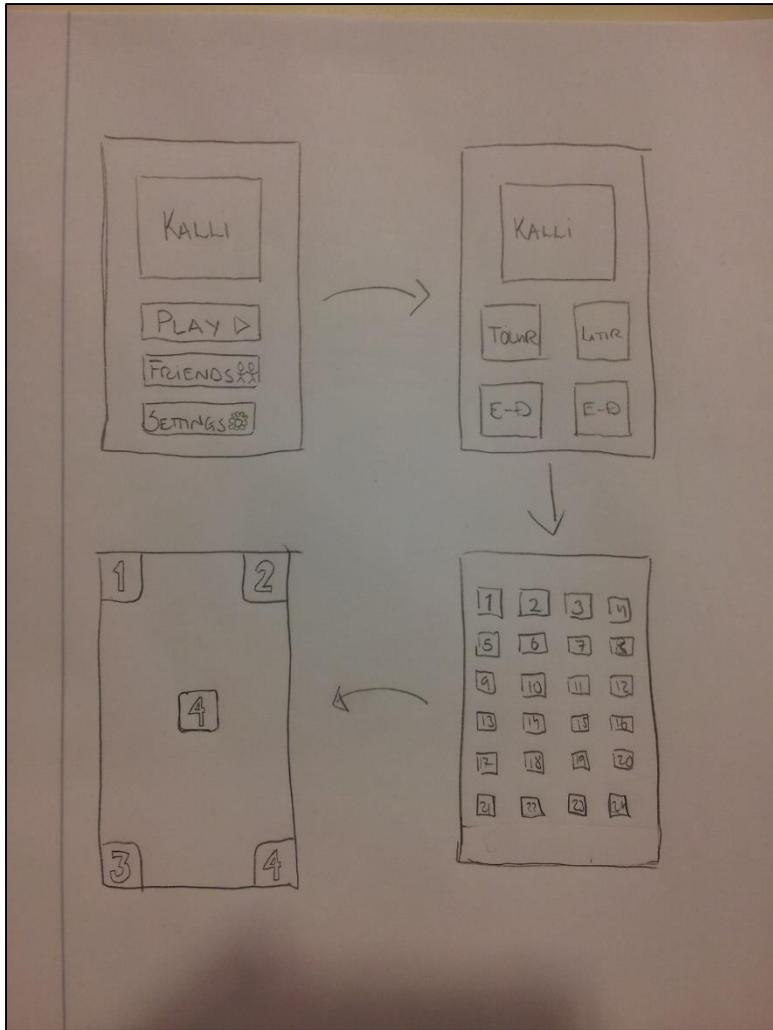
1/9

1/9

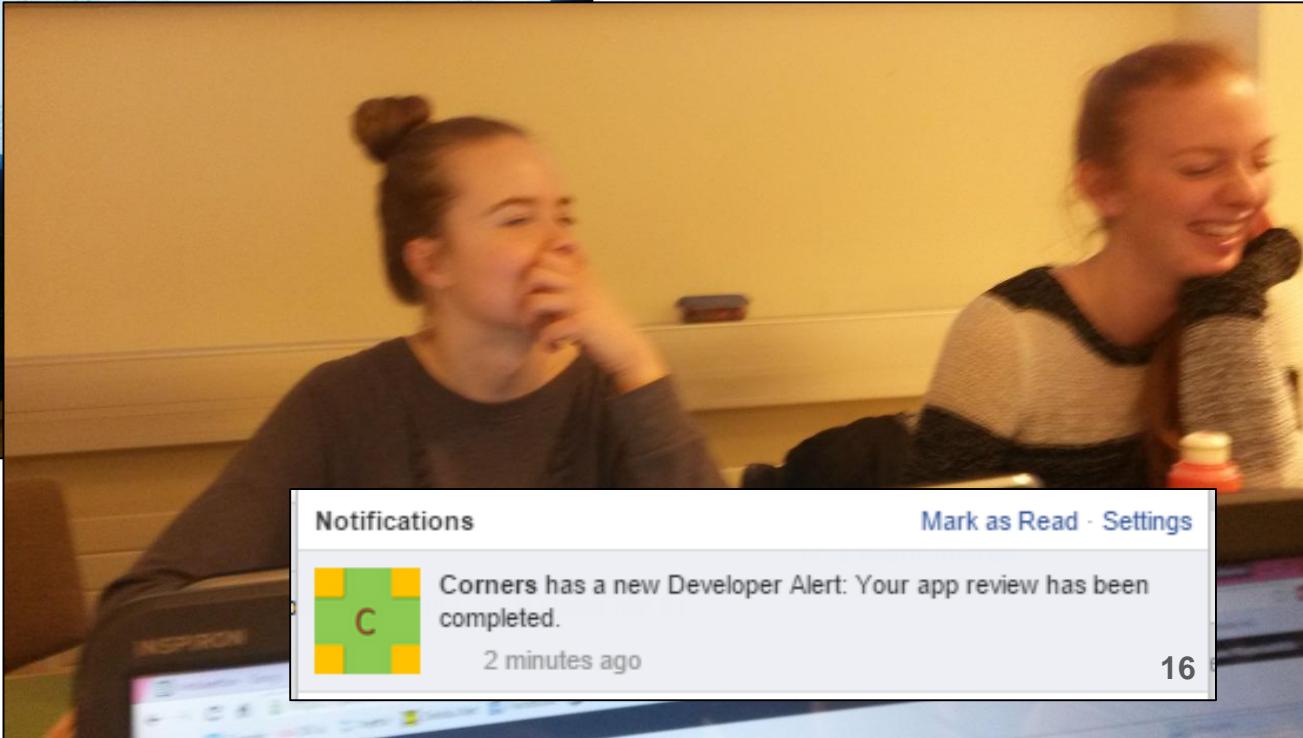
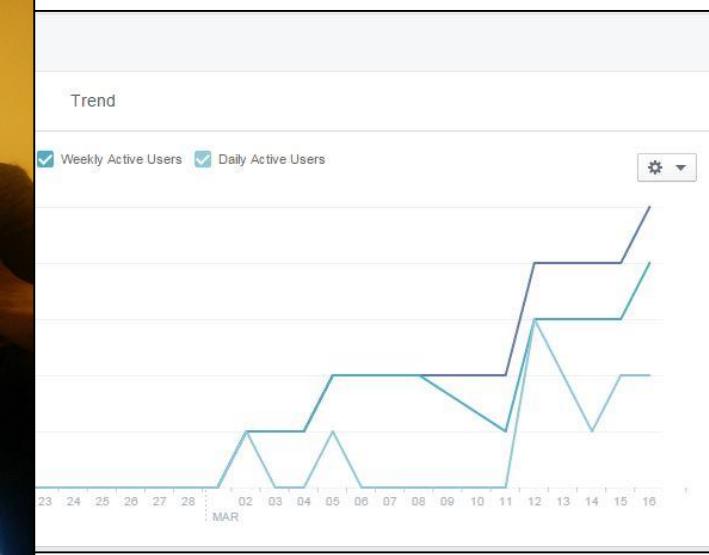
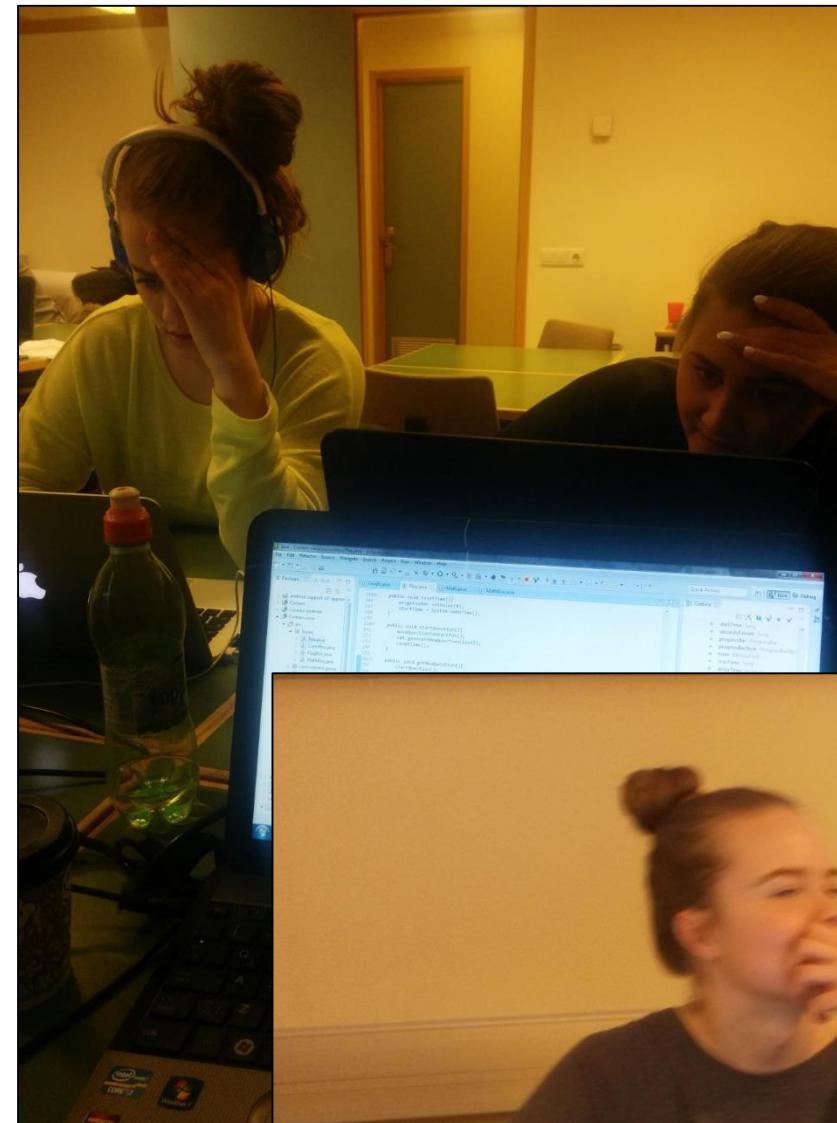
1/9

14

Spotlight: Corners



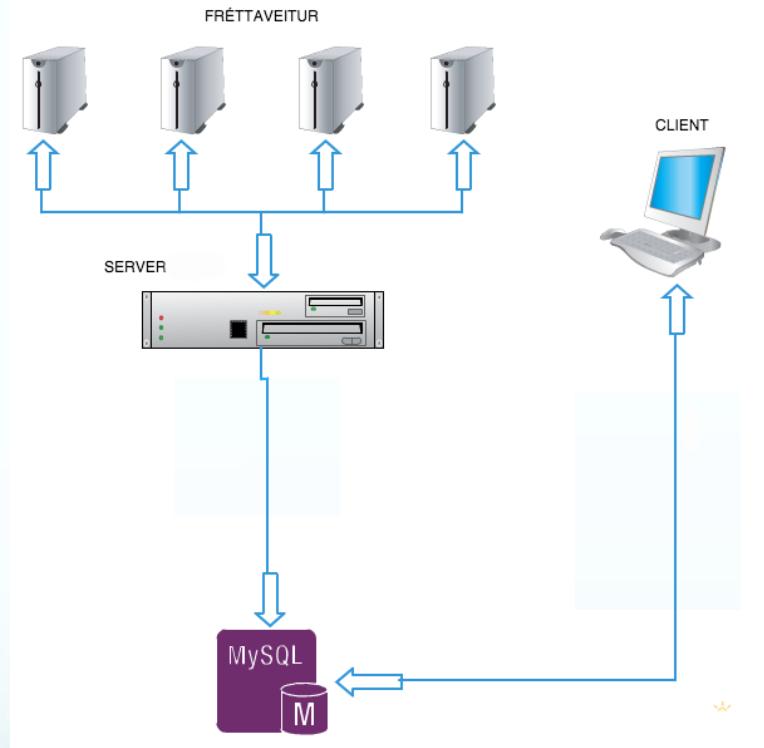
Spotlight: Corners





FREGNIR

Collecting news from different news media



OUR 2 BIGGEST CHALLENGES

- Amount of requests
- What is unique to a single news

Requests

- Collecting data every 2 minutes
- ~200 requests each time
 - ~6.000 requests each hour
 - ~140.000 requests each day
 - ~4.300.000 requests each month
- Hit rate limit

What is unique about single news

- Is the news already in the database?
How should the comparison be?
How should the equal method be implemented?
- What can you use?
 - link to the news
 - the title
 - the description
 - time
 - link to the photo
- Everything could change! → Duplicates news
- Solution:
 - complex connections between this items
 - comparison: old text similar to the new text

Most interesting about the class

- Freedom to do what you like to do, do something interesting
- Facing new problems and challenges and you have to find your own solutions

O

?

X

SWAP

A

!

□

Upphaf





Meteos

Play Game

High Score

How To Play

Ottar Company Production - 2004

High Scores

1. Ottar_Legend..... 56830
2. Arni_Vidskipta..... 43267
3. Hlif_Husmodir..... 21134
4. Hjortur_SickBoy 19547
5. Skossi&Hlynur..... 762

Go Back



Ok I got this, take me back

How to play.

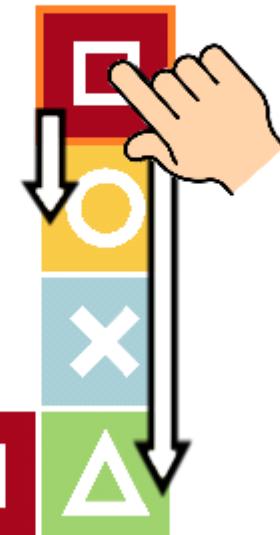
First of all, Meteos is about strategy and timing. You have to shoot the bad guys and get points without loosing badly.

Stack colored cubes together to get points and make it harder for the enemy as shown in the picture below

Next

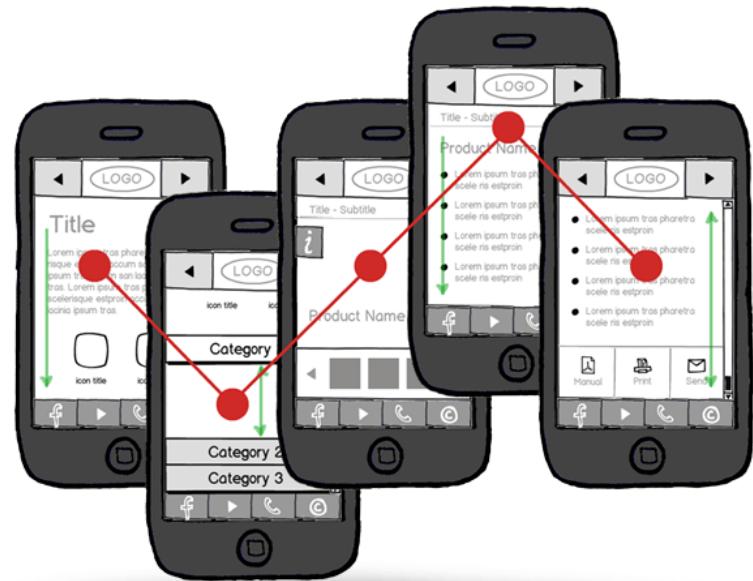


Drag blocks
up or down



NOT left and right

Notendaviðmót og útlit

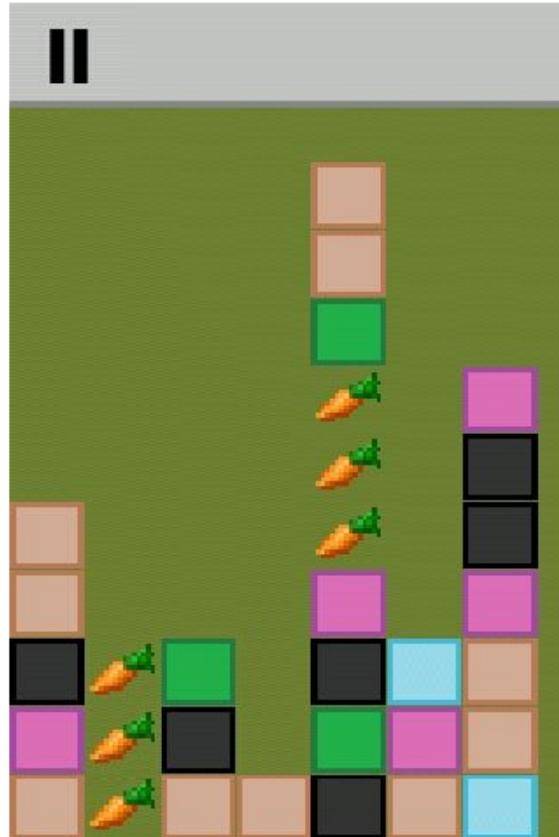


BLOKK GAME

Play Game

Scores

How to play



HOW TO PLAY

Click on buttons below
to get the desired info.

Objective

Controls

Info



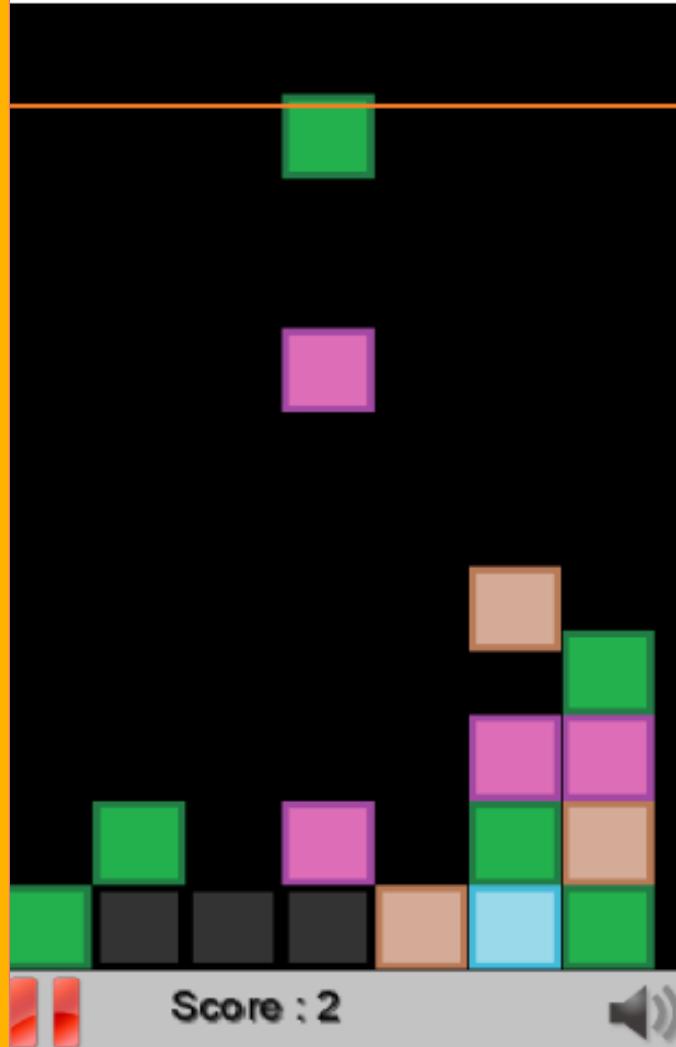
BLOKK GAME

Play Game

Scores

How to play

Store



GAME OVER

Sorry, no high score was made.

Your score was : 2

1st : 26000

2nd : 12000

3rd : 10000



STORE

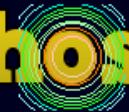
Your currency : 0

| | |
|--------------------|--------|
| Extra sound pack : | 5.000 |
| More blocks : | 20.000 |
| Slow motion mode : | 25.000 |
| Buggy mode : | 35.000 |
| Unlock credit list | 40.000 |
| Golden carrot : | 50.000 |

Unlock the whole game for \$0.99.
Coming soon.



Anymátigñhos

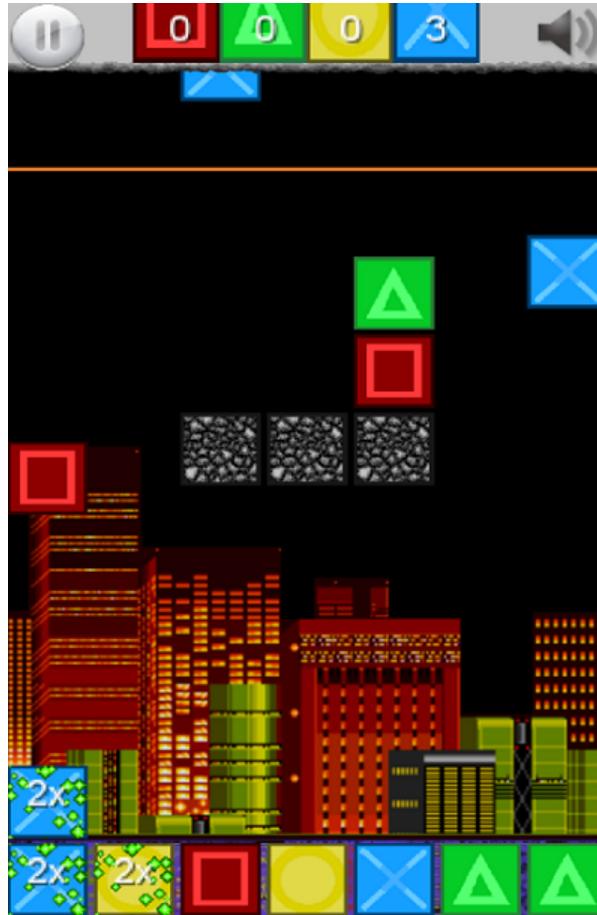


Play Game

Scores

How to play

Store



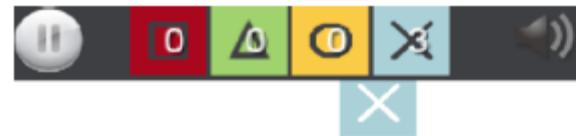
SWAP

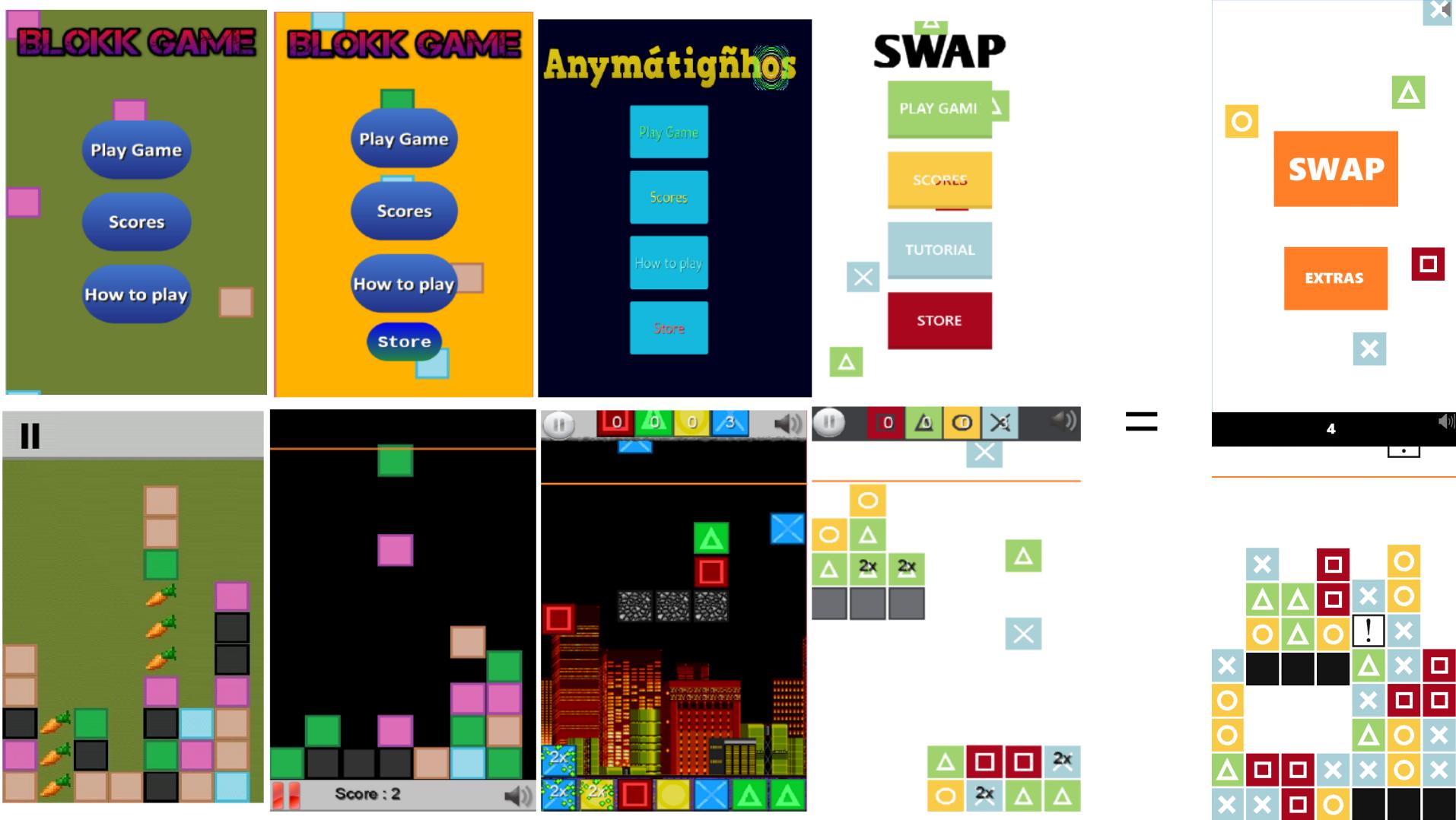
PLAY GAME

SCORES

TUTORIAL

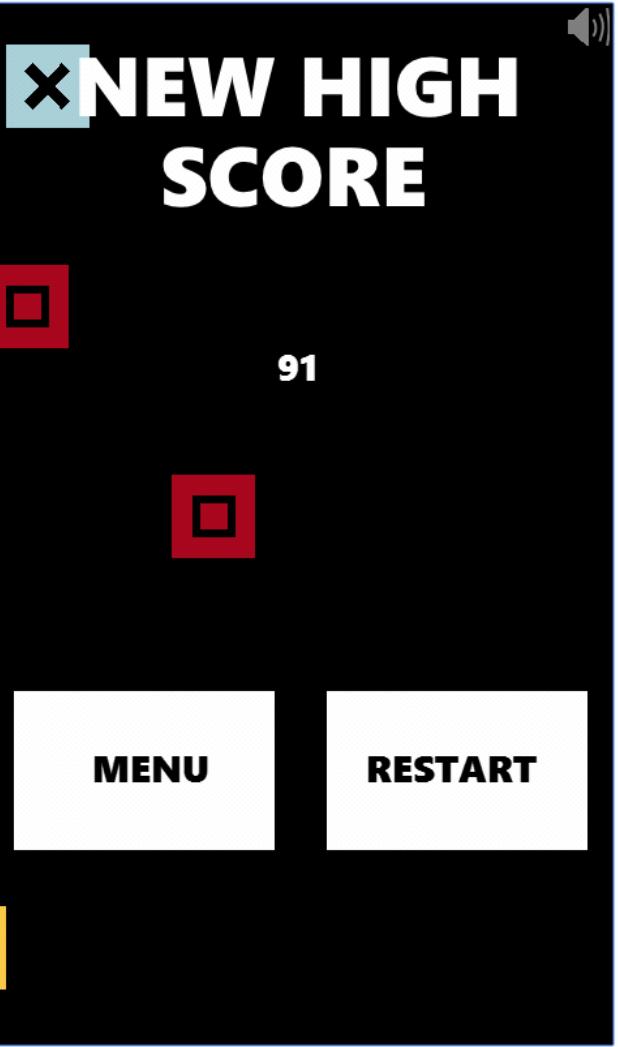
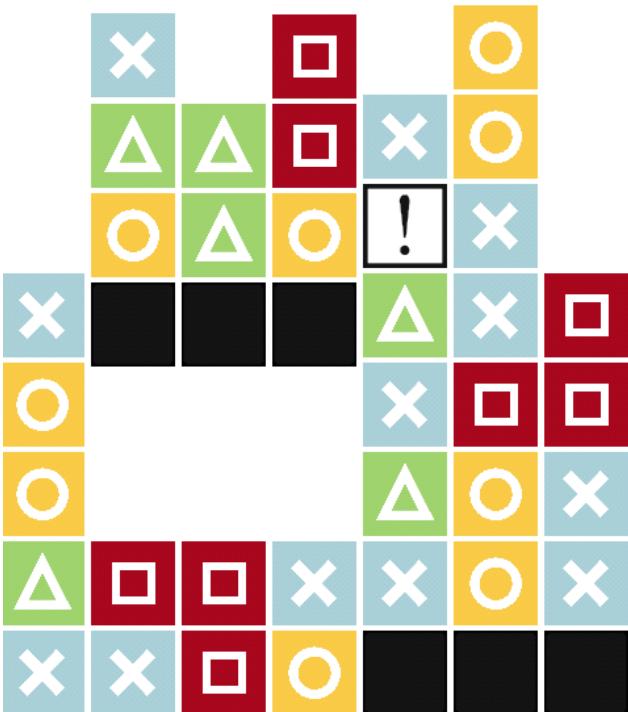
STORE



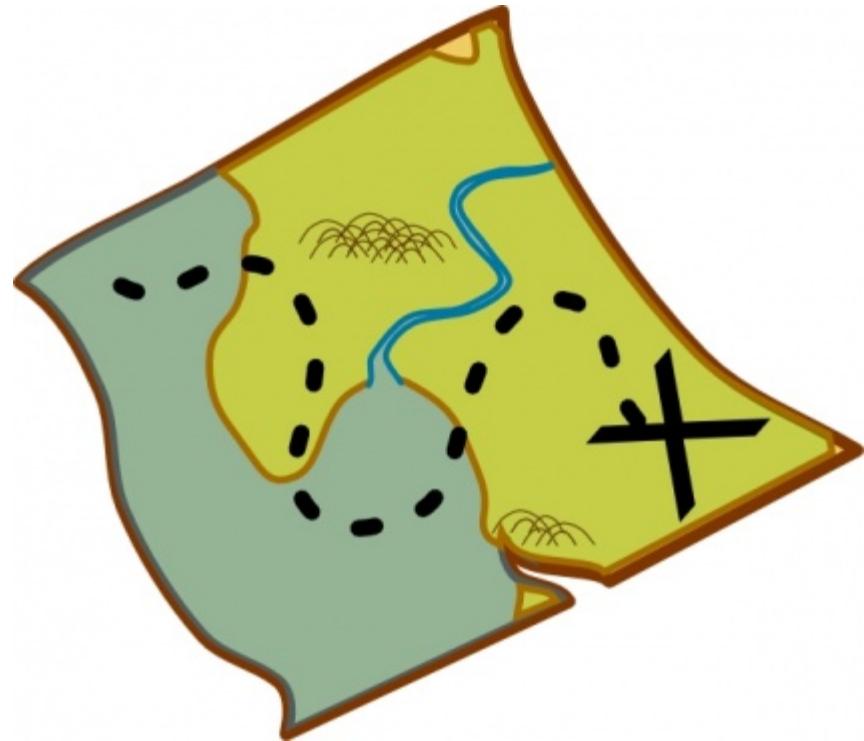




4

**SWAP****EXTRAS**

Ferðalagið





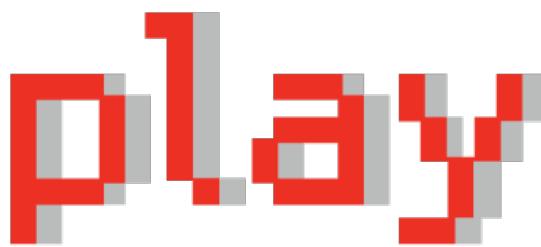
Hópefli í kjallaranum

*Léttöl



UT Messan

klak innovit



REYKJAVIK ■ APRIL 28-29th ■ 2015

Útgáfa





SWAP

Goon Squad Puzzle

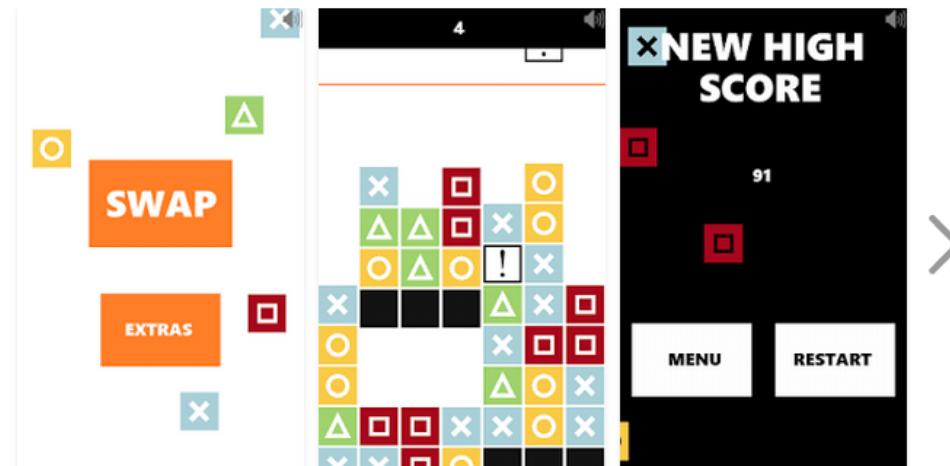
★★★★★ 7 •

PEGI 3

This app is compatible with your device.

Add to Wishlist

Install





"Dude, suckin' at something
is the first step to being
Sorta good at something."





Takk
fyrir

The Software Engineering Challenge

- You need to figure out requirements *and* solutions!
- This requires
 - Communication among your team members, distributing responsibility for components
 - A joint vision of the overall project, and each component's responsibilities
 - An understanding of which aspects of a component are public, and which are private
 - A joint specification of the interfaces between communicating components
 - Test drivers that your components can rely on while other components are still missing
 - Project management to ensure you deliver what you promised in time
- These are typical challenges that any large software project is grappling with.
- Lectures and team consultations will show you how to deal with them.



Team Formation

- **How:** Sign up on Doodle (<http://doodle.com/d77a698m4zvda936>)
 - Enter your name and HÍ e-mail address – example: “Jón Jónsson (jj1)”
 - Choose your desired team (max. 4 members per team; 3-4 recommended)
 - Note: Your team number determines your consultation time slot (see slide 12)
 - Ideally, form teams offline and let one person enter the names of all team members at once
 - If you can't find team members offline:
 - If you are early: Sign up for an empty group and wait for people to join
 - If you are late: Join a group that has only 1-2 members
 - If you can find only groups with 3 members:
Introduce yourself, ask what they are planning to build, and if it's ok to join them
 - If you are looking for teams or team members: Communicate through the course forum on Uglar
 - To change your group affiliation: Delete your previous Doodle entry and create a new one
 - Do not edit other people's Doodle entries! Resolve assignment conflicts by e-mail
- **When:** by next Wednesday (2 Sep)
 - So you can start working on your project idea together in the first consultation on Thursday
 - Team formation needs to be finalized by Sun 6 Sep at the latest



Grading

Assignments (combined weight: 50%)

- 4 team assignments
 - All team members co-author documents with project deliverables
 - Each team member must lead brief presentation of 1-2 deliverables to tutors
 - Focus: Don't just tell us what you did, but *why* you decided to do it this way!
 - Each team member's grade depends on
 - Quality of overall teamwork and product
 - Quality of indiv. deliverable presentations
 - Assessment of contribution by teammates
 - Assessment of competence by tutors
- Need passing grade on averaged assignments to be admitted to final exam

Final exam (weight: 50%)

- Focus: Understanding of software engineering concepts and methods
 - no extensive coding
 - maybe read/write small Java fragments
 - some UML modeling
- Scope: Lecture slides
 - Note: The soundtrack is relevant!
- Style: Written exam
 - Questions: Explain/argue/decide/analyze...
 - Answers: Short paragraphs of whole sentences – justify your opinions!
- Need passing grade on final exam to pass course



Course Support

- **Questions and feedback** welcome anytime! Preferably:
 - in class
 - in team consultations
 - in Uglá discussion forum
- **See Uglá for**
 - Slides (for download after each lecture)
 - Important course announcements
- **Contact information**
 - Matthias Book (book@hi.is)
 - Daníel Páll Jóhannsson (dpj4@hi.is)
 - Elsa Mjöll Bergsteinsdóttir (emb12@hi.is)



Suggested Literature

(for more in-depth reading – not required to buy for course)

- **Requirements Engineering**

- Karl Wiegers, Joy Beatty: Software Requirements. Microsoft Press, 2013

- **Object-oriented Analysis and Design**

- ❖ Craig Larman: Applying UML and Patterns. Prentice Hall, 2004
 - Eric Freeman, Bert Bates: Head First Design Patterns. O'Reilly, 2004

- **Java Web Application Development**

- Java Enterprise Edition Documentation. <https://docs.oracle.com/javaee/7/index.html>
 - Spring Framework Documentation. <http://spring.io/docs>
 - Nicholas S. Williams: Professional Java for Web Applications. Wrox, 2014

- **Unified Process**

- The Rational Unified Process. <http://sce.uhcl.edu/helm/rationalunifiedprocess/>
 - Per Kroll, Philippe Kruchten: The Rational Unified Process Made Easy – A Practitioner's Guide to the RUP. Addison-Wesley, 2003

Software Process Models

see also:

Pressman: Software Engineering – A Practitioner's Approach, Ch. 1 & 2



The Nature of Software

- Software comprises:
 - **Programs** (sets of computer instructions) that when executed provide desired features
 - **Data structures** that enable programs to adequately manipulate information
 - **Documentation** that describes the operation and use of programs
- Software vs. physical products
 - Software is developed or engineered, it is not manufactured.
 - Software does not “wear out”, but it can deteriorate over time.
 - Software is mostly custom-built (despite a trend toward component-/service-based systems)
- **Software is both a product and a vehicle that delivers a product.**



Types of Software

(Categories are not disjoint, but characterized by specific challenges)

- **System software**
 - Heavy interaction with hardware, no dedicated user interface (e.g. printer drivers, ...)
- **Application software**
 - Stand-alone software addressing particular need (e.g. spreadsheets, scientific computing...)
- **Distributed systems**
 - Code and data spread across multiple hardware devices (e.g. web applications, ...)
- **Embedded systems**
 - Highly specialized, highly hardware-dependent, real-time tasks (e.g. cruise control, ...)
- **Product-line software**
 - Many variation points to serve individual users' needs (e.g. accounting, ...)
- **Games**
 - Heavy focus on user experience, often with real-time aspects (e.g. arcade, first-person, ...)
- **Artificial intelligence systems**
 - Techniques such as machine learning, neural networks etc. (e.g. pattern recognition, ...)



Software as a Product and a Vehicle Delivering a Product

- There are many different [opinions on] things that software could and should do.
 - Developers need to understand the problem (**requirements**) before developing a solution.
- Application domains and technical toolsets are highly complex.
 - Smooth interaction of all software components requires careful consideration (**design**).
- Software errors have (possibly dramatic) consequences in real life.
 - Developers must ensure their software exhibits high quality (**testing**).
- Successful software will be used and built on for a long time.
 - Software should be built to be adaptable and extensible (**maintenance**).



Definitions of “Software Engineering”

- “The establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.”
 - Fritz Bauer, in: P. Naur, B. Randell (Eds.): Software Engineering: Report of a Conference. NATO Scientific Affairs Division, Bruxelles, 1969
- “The systematic approach to the development, operation, maintenance, and retirement of software.“
 - IEEE Standard Glossary of Software Engineering Terminology (IEEE Standard 729). IEEE Computer Society, 1983



Barton, Dijkstra, Wodon, Gill, Hume, Smith, Paul, Perlis, Randell at the NATO Software Engineering Conference, Garmisch, Germany, Oct 1968

The Essence of Engineering Practice

General Problem Solving Approach *(after George Polya, 1945)*

- Understand the problem
- Plan a solution
- Carry out the plan
- Examine result for accuracy

Software Engineering Activities

- Requirements Engineering
- Design and specification
- Code implementation
- Testing and quality assurance



The Essence of Engineering Practice

- **Understand the problem**

- Who has a stake in the solution?
- What are the unknowns?
- Can the problem be broken down?

- **Plan the solution**

- Have you seen similar problems before?
- Has a similar problem been solved?
- Can sub-problems be defined?
- Can you represent a solution in a manner that leads to effective implementation?

- **Carry out the plan**

- Does the solution conform to the plan?
- Is each component of the solution correct?

- **Examine the result**

- Is it possible to test each component of the solution?
- Does the solution produce results that conform to the required features and data?



Software Engineering Activities

▪ General Activities:

- Communication – understand what the stakeholders need
- Planning – consider how you will build a solution in the given time / budget
- Design – figure out how exactly the solution should work
- Construction – build the solution according to your design
- Deployment – roll out the solution for use by the stakeholders
- [Maintenance – all of the above, on a smaller, more detail-focused scale]

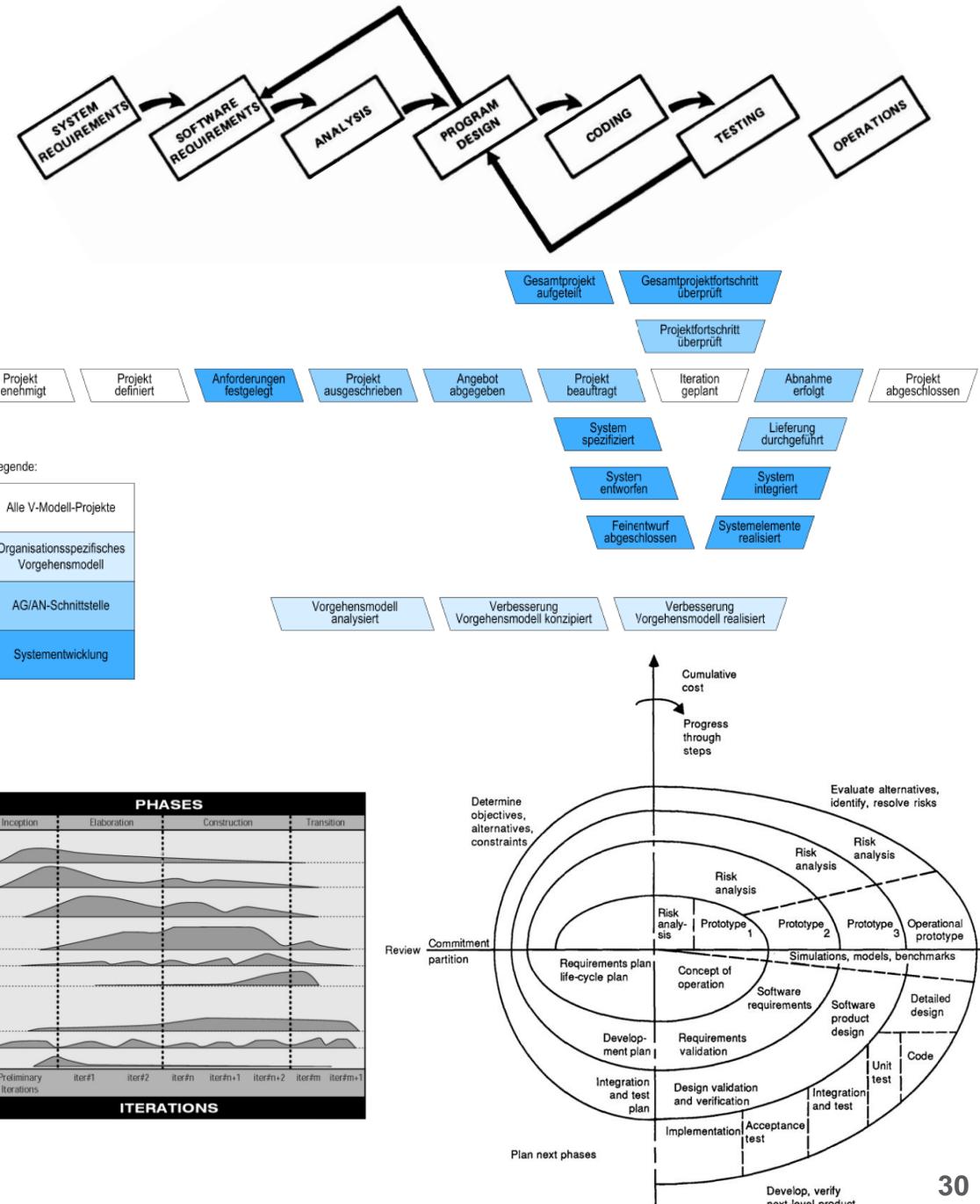
▪ Umbrella Activities:

- Quality assurance – make sure you build the right solution, the right way
- Project management – allocate resources, track progress
- Risk management – identify and eliminate risks
- Process improvement – measure/review how you are doing, learn, improve your methods



Software Process Models

- A **software process model** defines
 - how the various software engineering activities are structured in time
 - how they depend on each other through the creation and consumption of artifacts
- Large variety of models
 - Waterfall model, V model, spiral model, Unified Process (UP), agile models...
- Major differences
 - Structure vs. flexibility
 - Emphasis placed on different activities and artifacts



Issues with Sequential Process Models

- A sequential model (i.e. an extreme version of the “waterfall”) attempts to
 - identify all or most requirements at the start of the project
 - create a thorough design as a “blueprint” of the complete system before programming starts
 - defines a complete project schedule at the beginning of the project
- This assumes that
 - no requirements will change over the course of the project
 - all implementation details can be foreseen at the start of the project
 - all initial time and effort estimates are accurate
 - no unforeseen questions, uncertainties, conflicts etc. come up
 - developers have the discipline to follow all specifications and schedules precisely
- Obviously unrealistic: The larger the project, the less true these assumptions.
 - Caution: Nevertheless, it is very tempting to plan a project under these assumptions!
 - Double-check: Do your “iterations” actually mirror the steps of a strict waterfall model?



Iterative-Incremental Process Models



- Split the project into **iterations**
- Each iteration is a mini-project...
 - Understand requirements
 - Design, code and test software
- ...and produces an **increment**, i.e. an executable partial system
 - Basis for review and feedback
 - Foundation for next iteration

- **Benefits:**
 - Better productivity, lower defect rates
 - Earlier identification, mitigation of risks
 - Early visible progress
 - Early user engagement and feedback, i.e. closer alignment with user needs
 - Manageable complexity in each iteration
 - Lessons learned in one iteration can help to improve the next

In Defense of Plan-Driven Process Models

- Some agile proponents mistakenly equate all plan-driven models with strict sequential models (“if it’s not agile, it’s waterfall!”).
- Actually, all modern plan-driven models are iterative-incremental.
 - (if they are applied properly and do not get a “strict waterfall” superimposed on them)
- Difference is in...
 - the necessary amount of pre-planning
 - the necessary amount of structured design
 - the certainty of the overall target vision
- ...in each iteration, depending on
 - the criticality of the application domain (e.g. health/financial risks)
 - the complexity of the application domain, technology, existing system landscape

Plan-driven vs. Agile Iterative Development

Plan-driven Iterative Development

- Gain understanding through models and specifications
- Risk management through planning
- Aspiration for stable structures
- Optimization through planning
- Work on most risky features first
- Increments are partial systems
- Stable overall target vision
- Well-defined roles and responsibilities
- Discipline required to follow plans

Agile Iterative Development

- Gain understanding through communication and feedback
- Risk management through flexibility
- Acceptance of fluid structures
- Optimization through refactoring
- Work on most valuable features first
- Increments should be viable products
- Open overall target vision
- Self-organizing teams
- Discipline required to utilize freedom





Hugbúnaðarverkefni 1 / Software Project 1

2. Software Processes

HBV501G – Fall 2015

Matthias Book



HÁSKÓLI ÍSLANDS
VERKFRÆÐI- OG NÁTTÚRVÍSINDASVIÐ
ÍÐNAÐARVERKFRÆÐI-, VÉLAVERKFRÆÐI-
OG TÖLVUNARFRÆÐIDEILD

Team Consultations

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|---|
| E | x | | | | x | x | | | | | | x | | | | x | | | x | | x | | | | | x | | | x | | |
| D | | | x | | | x | | x | | | | x | | | | x | | x | x | | x | | | | x | | x | | | | |
| M | | x | x | | | | | | | | x | | | x | x | | | | | | | x | | x | | | x | | | | |
| # | 3 | 4 | 4 | 3 | 3 | 2 | 3 | 0 | 4 | 4 | 4 | 2 | 4 | 3 | 4 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 4 | 4 | 3 | 3 | 4 | 4 |

▪ Consultation time slots

- Thursday 08:30-11:30, V-152
 - Teams 1-10: 08:30-09:30
 - Teams 11-20: 09:30-10:30
 - Teams 21-30: 10:30-11:30
- Weekly meetings with tutors
 - to discuss requirements, design, questions
 - to present assignment deliverables

▪ Team and topic finalization

- Fix team members in Doodle by **Sun 6 Sep**
- Decide on a project idea within your team by **Wed 9 Sep**
- Attend consultations on **Thu 10 Sep**
 - to discuss idea with tutor
 - to define project scope



Recap: Grading

Assignments (combined weight: 50%)

- 4 team assignments
 - All team members co-author documents with project deliverables
 - Each team member must lead brief presentation of 1-2 deliverables to tutors
 - Focus: Don't just tell us what you did, but *why* you decided to do it this way!
 - Each team member's grade depends on
 - Quality of overall teamwork and product
 - Quality of indiv. deliverable presentations
 - Assessment of contribution by teammates
 - Assessment of competence by tutors
- Need passing grade on averaged assignments to be admitted to final exam

Final exam (weight: 50%)

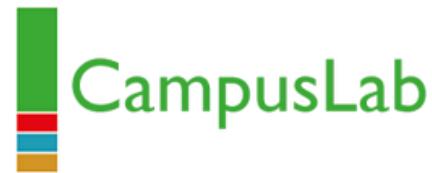
- Focus: Understanding of software engineering concepts and methods
 - no extensive coding
 - maybe read/write small Java fragments
 - some UML modeling
- Scope: Lecture slides
 - Note: The soundtrack is relevant!
- Style: Written exam
 - Questions: Explain/argue/decide/analyze...
 - Answers: Short paragraphs of whole sentences – justify your opinions!
- Need passing grade on final exam to pass course



Meta: Education vs. Practice

- Your projects in this course will likely be relatively simple
 - given the timeframe of three months until delivery
 - given the effort you will be able to invest beside your other coursework
 - given the synchronization issues of learning about methods in parallel to applying them
- In industrial practice, this simplicity would make them obvious candidates for
 - an agile development process
 - a client-side web technology
- However, our learning goal is to understand how to use
 - a plan-driven process
 - object-oriented design
- Initial experience with these is best gained using a lightweight project example.
 - (Plus, complexity of the technology should motivate some of the complexity of the process.)

Save the Date



UNIVERSITÄT
DUISBURG
ESSEN

PALUNO
The Ruhr Institute for Software Technology

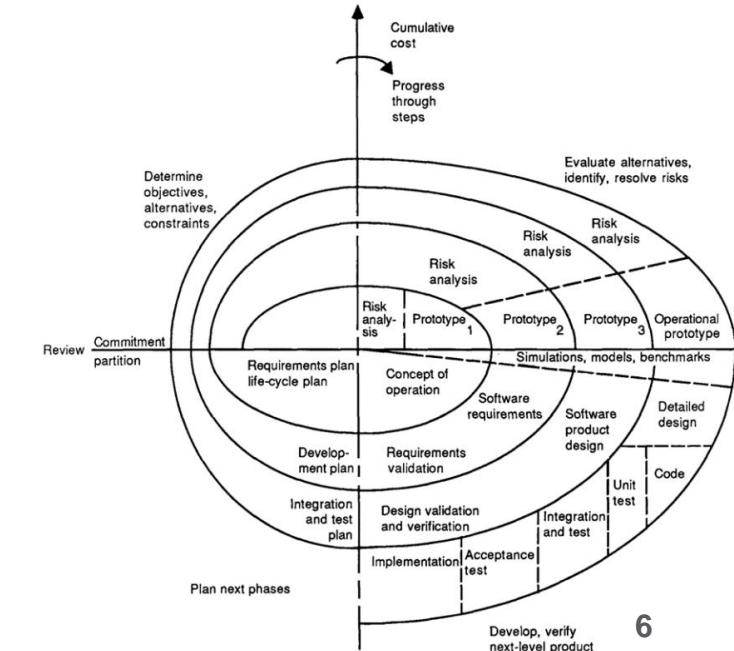
- Course unit on Project Management taught by guest lecturers Julia Hermann and Marc Hesenius
 - Research Associates at University of Duisburg-Essen, Germany
 - Tutors in CampusLab, a company providing training to IT professionals
- The Project Management unit will be taught in two blocks:
 - Thu 17 Sep 08:30-11:30 and
 - Fri 18 Sep **12:50-14:50**
- Please attend for the whole time, not just your team's timeslot!
 - Highly interactive teaching style and parallel group work
 - Not possible to join classes midway through



HÁSKÓLI ÍSLANDS

Recap: Software Process Models

- A software process model defines
 - how the various software engineering activities are structured in time
 - how they depend on each other through the creation and consumption of artifacts
- All modern plan-driven models are iterative-incremental
 - (if they are applied properly and do not get a “strict waterfall” superimposed on them)
- Difference is in...
 - the necessary amount of pre-planning
 - the necessary amount of structured design
 - the certainty of the overall target vision
- ...in each iteration, depending on
 - the criticality of the application domain (e.g. health/financial risks)
 - the complexity of the application domain, technology, existing system landscape



Recap: Plan-driven vs. Agile Iterative Development

Plan-driven Iterative Development

- Gain understanding through models and specifications
- Risk management
- Aspiration for stability
- Optimization throughout
- Work on most important features
- Increments are well-defined
- Stable overall tasks
- Well-defined roles and responsibilities
- Discipline required to follow plans

Agile Iterative Development

- Gain understanding through communication and feedback
- Through flexibility
- Structures
- Refactoring
- Feature prioritization
- Release features first
- Create viable products
- Vision

*In preparing for battle,
I have always found that plans are useless,
but planning indispensable.*

-- Dwight D. Eisenhower

The Unified Process

see also:

- Larman: Applying UML and Patterns, Ch. 2



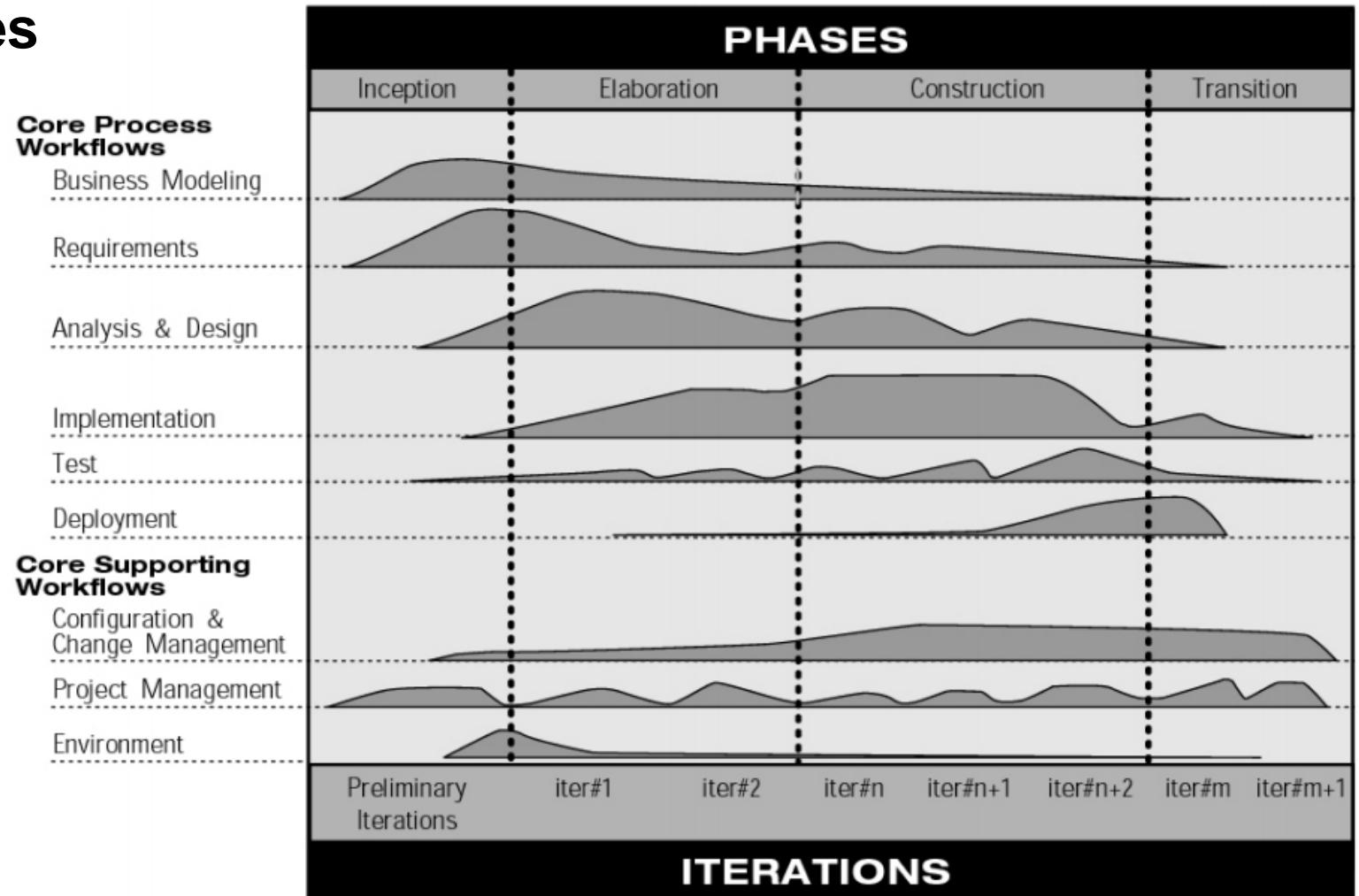
The Unified Process (UP)

- Developed by Ivar Jacobson, Grady Booch and James Rumbaugh in parallel with the Unified Modeling Language (UML); first published in 1999
- Defines a number of roles, workflows, activities and artifacts
 - See <http://sce.uhcl.edu/helm/rationalunifiedprocess/> for a browseable reference
 - Most of these are optional
 - UP needs to be tailored to individual project situations
 - Make deliberate decisions on which elements you need and why!
- Many refinements and variations proposed over time, e.g.:
 - **Rational Unified Process (RUP) – IBM's version**
 - Oracle Unified Method (OUM) – Oracle's version
 - Open Unified Process (OpenUP) – the Eclipse Foundation's version
 - Essential Unified Process (EssUP) – Jacobson's lightweight version



Unified Process Structure

- The UP defines four **phases** (inception, elaboration, construction, transition).
 - Each phase produces a **milestone** composed of a set of artifacts.
- Each phase consists of several **iterations**.
 - Each iteration produces an **increment**.
- Each iteration comprises **activities** from various **disciplines** (e.g. design, implementation).
 - Activities rely on and produce **artifacts**.



Unified Process Phases

1. Inception

- Approximate vision
- Business case
- Scope
- Vague estimates

Not a “requirements phase”, but feasibility analysis after which we can decide whether it makes sense to proceed with project

2. Elaboration

- Refined vision
- Identification of most requirements and scope
- More realistic estimates
- Iterative implementation of core architecture
- Resolution of high risks

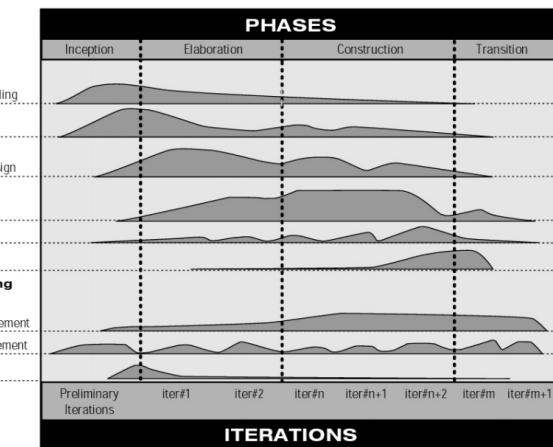
3. Construction

- Iterative implementation of lower-risk and other elements
- Preparation for deployment

4. Transition

- Beta tests
- Deployment

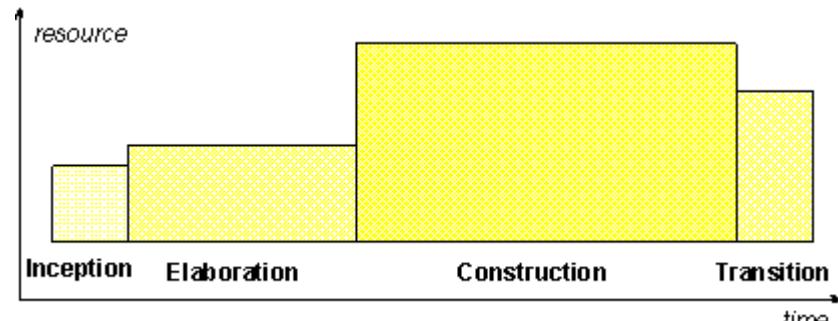
Not a “design phase”, but iterative development of core architecture and mitigation of high risks



Unified Process Phases

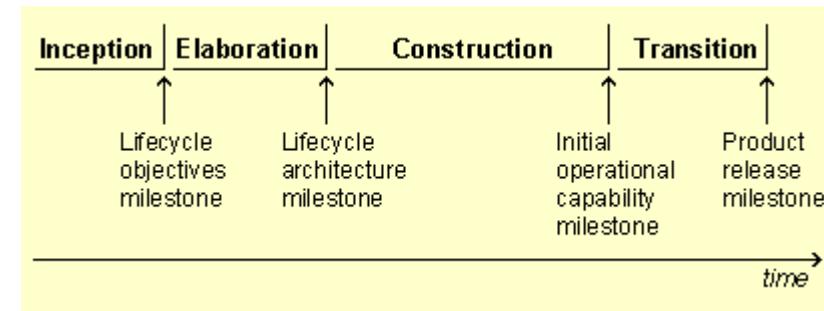
Time and Effort Distribution

- Inception and Transition typically take up to 10% each of time and effort
- Elaboration typically takes at most one third of time and effort
- Construction typically takes at least half of time and effort



Milestones

- Activities within phases overlap
- However, phases are separated by milestones to judge project feasibility and progress
- Milestones have defined deliverables and evaluation criteria



Inception: Primary Objectives

- Establish the project's **software scope and boundary conditions**, including
 - an operational vision
 - acceptance criteria
 - what is intended to be in the product and what is not
- Discriminate the **critical use cases** of the system
 - i.e. the primary scenarios of operation that will drive the major design trade-offs
- Identify (and maybe demonstrate) at least one **candidate architecture**
 - against the background of some of the primary scenarios
- Estimate the **overall cost and schedule** for the entire project
 - and more detailed estimates for the elaboration phase that will immediately follow
- Estimate **potential risks** (i.e. sources of unpredictability / uncertainty)
- Prepare the **supporting environment** for the project



Inception: Essential Activities

▪ Formulating the scope of the project.

- This involves capturing the context and the most important requirements and constraints to such an extent that you can derive acceptance criteria for the end product.

▪ Planning and preparing a business case.

- Evaluating alternatives for risk management, staffing, project plan, and cost/schedule/profitability trade-offs.

▪ Synthesizing a candidate architecture,

- evaluating trade-offs in design, and in make/buy/reuse, so that cost, schedule and resources can be estimated. The aim here is to demonstrate feasibility through some kind of proof of concept. This may take the form of a model which simulates what is required, or an initial prototype which explores what are considered to be the areas of high risk. *The prototyping effort during inception should be limited to gaining confidence that a solution is possible – the solution is realized during elaboration and construction.*

▪ Preparing the environment for the project,

- assessing the project and the organization, selecting tools, deciding which parts of the process to improve.



Inception: “Lifecycle Objectives” Milestone

Evaluation Criteria

- Stakeholder concurrence on scope definition and cost/schedule estimates
- Agreement that the right set of requirements have been captured and that there is a shared understanding of these requirements.
- Agreement that the cost/schedule estimates, priorities, risks, and development process are appropriate.
- All risks have been identified and a mitigation strategy exists for each.

Artifacts

- Vision
- Business Case
- Risk List
- Software Development Plan
- Iteration Plan
- Development Case
- Tools
- Glossary
- Use Case Model
- Project Repository
- Use Case Modeling Guidelines
- Domain Model
- Project-Specific Templates
- Prototypes

“Ensure that it makes sense to do this.”



Elaboration: Primary Objectives

- Establish **baseline architecture** addressing the architecturally significant scenarios
 - which typically expose the top technical risks of the project
 - Address all architecturally significant **risks** of the project
 - Produce an **executable architecture** and possibly exploratory, throw-away prototypes to mitigate specific risks such as
 - design/requirements trade-offs
 - component reuse
 - product feasibility
 - acceptance of investors, customers, and end-users
 - Demonstrate that the baseline architecture will support the **requirements** of the system
 - at a reasonable cost and in a reasonable time
 - Establish the **supporting environment** for the project
- Ensure that...
- the architecture, requirements and plans are stable enough
 - the risks sufficiently mitigated
- ...to predictably determine the **cost and schedule** for completion of development



Elaboration: Essential Activities

- **Defining, validating and baselining the architecture.**
- **Refining the vision**
 - based on new information obtained during the phase, establishing a solid understanding of the most critical use cases that drive the architectural and planning decisions.
- **Creating and baselining detailed iteration plans**
 - for the construction phase.
- **Refining the development case and putting the development environment in place,**
 - including the process, tools and automation support required to support the construction team.
- **Refining the architecture and selecting components.**
 - Potential components are evaluated and the make/buy/reuse decisions sufficiently understood to determine the construction phase cost and schedule with confidence.
 - The selected architectural components are integrated and assessed against the primary scenarios.
 - Lessons learned from these activities may well result in a redesign of the architecture, taking into consideration alternative designs or reconsideration of the requirements.



Elaboration: “Lifecycle Architecture” Milestone

Evaluation Criteria

- Product vision and requirements are stable.
- Architecture is stable.
- Key test and evaluation approaches are proven.
- Test and evaluation of executable prototypes have demonstrated that major risk elements have been addressed and credibly resolved.
- Iteration plans for Construction phase are of sufficient detail and fidelity to allow the work to proceed.
- Iteration plans for Construction phase are supported by credible estimates.
- All stakeholders agree that the current vision can be met if the current plan is executed to develop the complete system, in the context of the current architecture.
- Actual resource expenditure versus planned expenditure are acceptable.

Artifacts

- Prototypes
- Risk List
- Development Case
- Tools
- Software Architecture Document
- Design Model
- Data Model
- Implementation Model
- Vision
- Software Development Plan
- Design and Programming Guidelines
- Iteration Plan
- Use Case Model
- Supplementary Specifications
- Test Suite
- Test Automation Architecture
- Business Case
- Analysis Model
- Training Materials
- Project-Specific Templates

“Ensure that **you are really able** to do it.”



An Important Transition

- **During Inception and Elaboration**, the project still is a relatively light-and-fast, low-risk operation.
 - If the project fails to reach one of these milestones,
 - it may be aborted due to infeasibility
 - or it needs to be considerably re-thought.
- **During Construction and Transition**, the project becomes a high-cost, high-risk operation with substantial organizational inertia.
 - If the project fails to reach one of these milestones,
 - deployment may have to be postponed by at least one release.
- Note: The UP is not the *cause* for the high complexity. Rather, it provides the *structure* that makes such highly complex projects manageable at all.



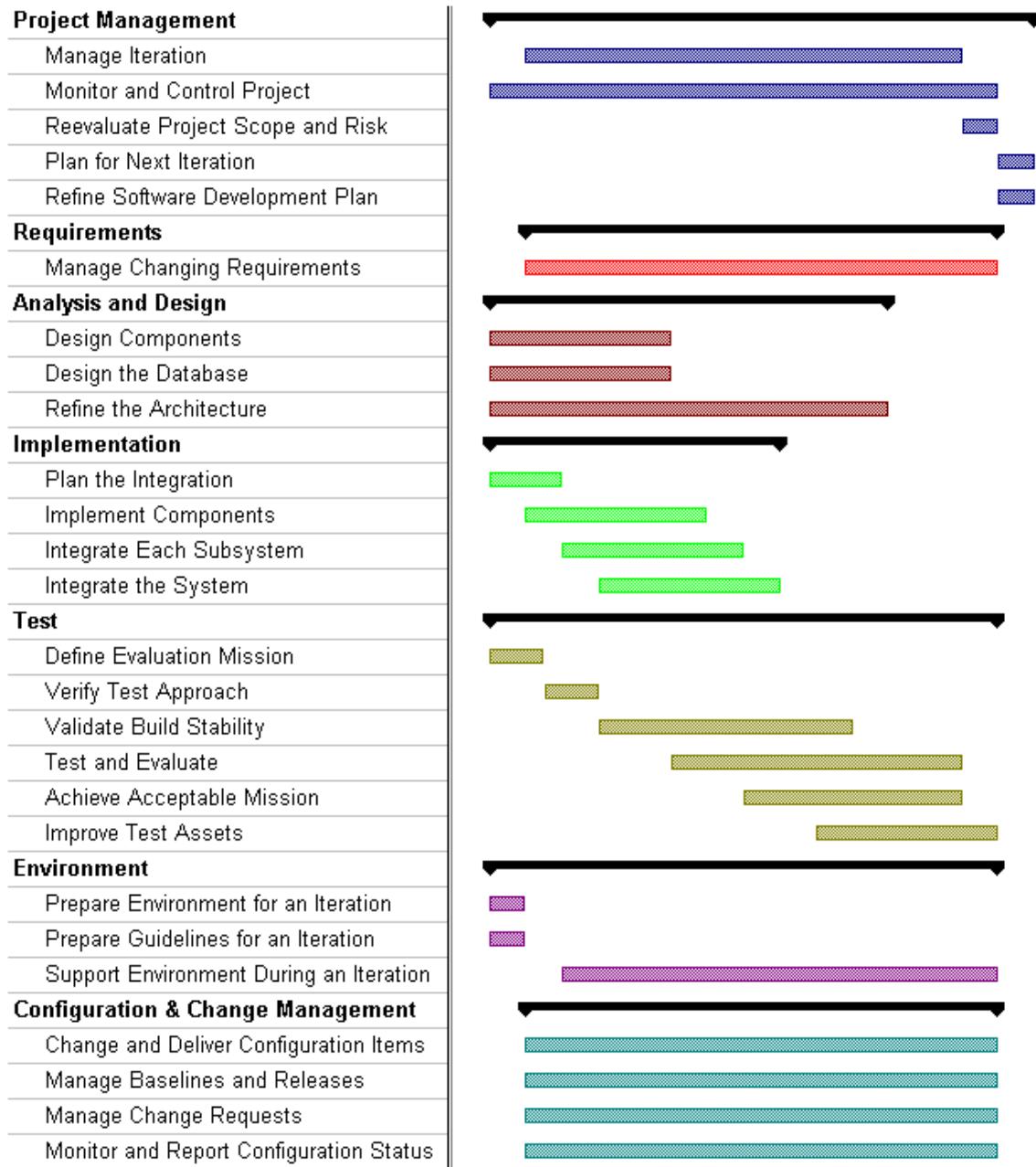
Construction: Primary Objectives

- Analyze, design, develop and test all required **functionality**.
- Achieve useful **versions** (alpha, beta, other test releases) of adequate quality.
- Minimize cost by optimizing **resources**, avoiding unnecessary scrap & rework.
- Achieve some degree of **parallelism** in the work of development teams.
 - Even on smaller projects, there are typically components that can be developed independently of one another, allowing for natural parallelism between teams (resources permitting).
 - Parallelism can accelerate the development activities significantly; but it also increases the complexity of resource management and workflow synchronization.
 - A robust architecture is essential if any significant parallelism is to be achieved.
- Decide if the software, sites, and users are all **ready** for application deployment.



Construction: Essntl. Activities

- Complete component **development and testing** against the defined evaluation criteria.
- **Resource management**, control and process optimization.
- **Assessment** of product releases against acceptance criteria for the vision.



Construction: “Initial Operational Capability” Milestone

Evaluation Criteria

- Is this **product release stable and mature** enough to be deployed in the user community?
- Are all the **stakeholders ready for the transition** into the user community?
- Are actual resource **expenditures** versus planned still **acceptable**?

Artifacts

- The System
- Deployment Plan
- Implementation Model
- Test Suite
- Training Materials
- Iteration Plan
- Design Model
- Development Case
- Tools
- Data Model
- Project-specific Templates
- Supplementary Specifications
- Use Case Model

“Ensure that it’s ready to beta-test.”



Transition: Primary Objectives

- **Beta-testing**
 - Validate the new system against user expectations
 - Parallel operation relative to a legacy system that shall be replaced
- **Tuning** activities such as bug-fixing, enhancement for performance and usability
- Achieving **user self-supportability**
- Assessment of the deployment baselines against the complete vision and the **acceptance criteria** for the product; achieving stakeholder concurrence that
 - deployment baselines are complete
 - deployment baselines are consistent with the evaluation criteria of the vision
- Executing **deployment** plans



Transition: Essential Activities

- **Deployment-specific engineering**
 - such as commercial packaging and production
- Converting operational **databases**
- **Testing** deliverable product at deployment site
- Finalizing end-user **support material**
- Creating a product **release**
- **Training** of users and maintainers
- Getting user **feedback**
- **Fine-tuning** the product based on feedback
- **Rollout** to marketing, distribution, sales forces



Transition: “Product Release” Milestone

“Ensure it’s **ready** for production use.”

Evaluation Criteria

- Is the **user satisfied**?
- Are actual resources **expenditures** versus planned expenditures **acceptable**?

Artifacts

- Product Build
- Release Notes
- Installation Artifacts
- Training Material
- End-User Support Material
- Test Suite
- Product Packaging





Hugbúnaðarverkefni 1 / Software Project 1

3. Inception Phase

HBV501G – Fall 2015

Matthias Book



HÁSKÓLI ÍSLANDS
VERKFRÆÐI- OG NÁTTÚRVÍSINDASVIÐ
ÍÐNAÐARVERKFRÆÐI-, VÉLAVERKFRÆÐI-
OG TÖLVUNARFRÆÐIDEILD

Team Consultations

▪ Tutor assignments

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| E | x | | | | x | x | | | | | | | | | | x | x | | x | | x | | | | | x | | x | | |
| D | | | x | | | | x | | x | | | x | x | | | | x | | | x | | | | | x | | x | | | |
| M | | x | x | | | | | x | | | | | | x | x | x | | | | | | x | x | x | | | x | | | |
| # | 3 | 4 | 4 | 3 | 3 | 2 | 3 | 4 | 4 | 0 | 0 | 2 | 4 | 3 | 4 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 3 | 4 | 4 | 3 | 3 | 4 | 4 |

▪ Time slots on Thursdays

- Teams 1-10: 08:30-09:30
- Teams 11-20: 09:30-10:30
- Teams 21-30: 10:30-11:30

▪ Seating arrangement in V-152

- Elsa's teams: first two rows
- Matthias' teams: middle two rows
- Daníel's teams: last two rows



Next Week

- Course unit on Project Management taught by guest lecturers Julia Hermann and Marc Hesenius
 - Research Associates at University of Duisburg-Essen, Germany
 - Tutors in CampusLab, a company providing training to IT professionals
- The Project Management unit will be taught in two blocks:
 - Thu 17 Sep 08:30-11:30 and
 - Fri 18 Sep **12:50-14:50**
- Please attend for the whole time, not just your team's timeslot!
 - Highly interactive teaching style and parallel group work
 - Not possible to join a day's class midway through



Recap: Unified Process Phases

1. Inception

- Approximate vision
- Business case
- Scope
- Vague estimates

Not a “requirements phase”, but feasibility analysis after which we can decide whether it makes sense to proceed with project

2. Elaboration

- Refined vision
- Identification of most requirements and scope
- More realistic estimates
- Iterative implementation of core architecture
- Resolution of high risks

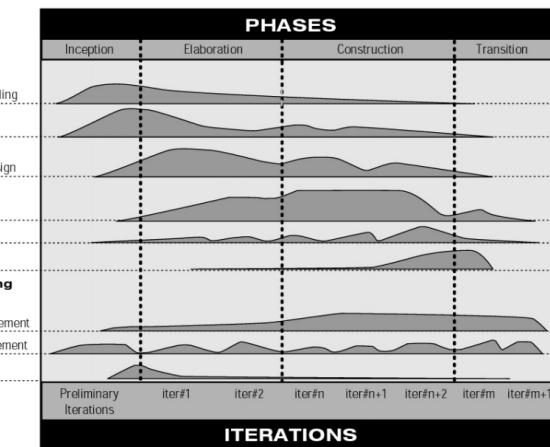
3. Construction

- Iterative implementation of lower-risk and other elements
- Preparation for deployment

4. Transition

- Beta tests
- Deployment

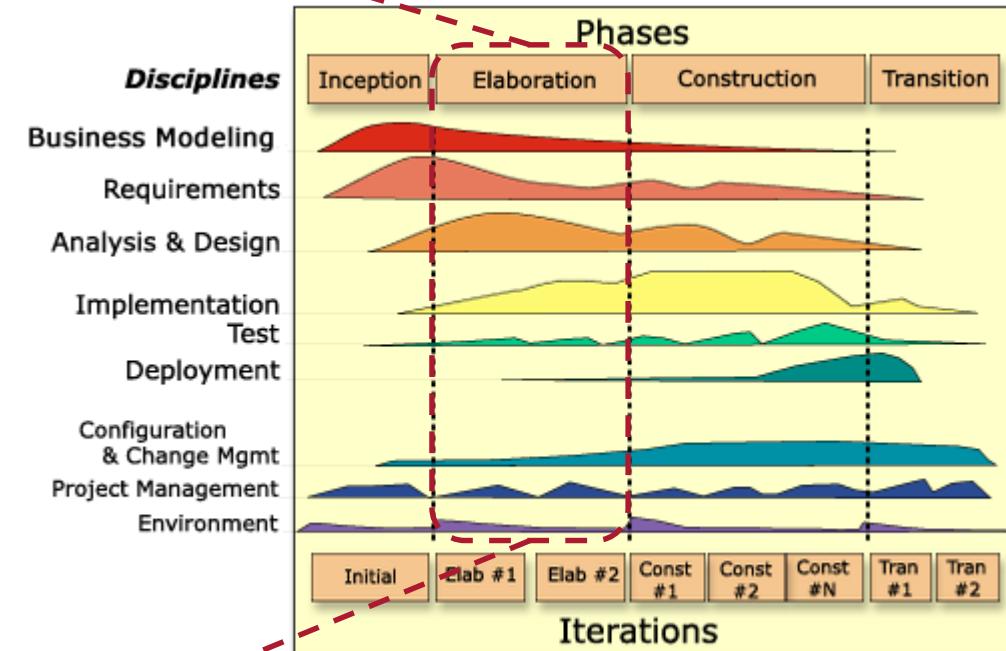
Not a “design phase”, but iterative development of core architecture and mitigation of high risks



Example: Definition of Activities and Artifacts in the RUP

| Project Management |
|---|
| Manage Iteration |
| Monitor and Control Project |
| Reevaluate Project Scope and Risk |
| Plan for Next Iteration |
| Refine Software Development Plan |
| Requirements |
| Analyze the Problem |
| Understand Stakeholder Needs |
| Define the System |
| Manage the Scope of the System |
| Refine the System Definition |
| Manage Changing Requirements |
| Analysis and Design |
| Define a Candidate Architecture |
| Analyze Behavior |
| Design Components |
| Design the Database |
| Refine the Architecture |
| Implementation |
| Structure the Implementation Model |
| Plan the Integration |
| Implement Components |
| Integrate Each Subsystem |
| Integrate the System |
| Test |
| Define Evaluation Mission |
| Verify Test Approach |
| Test and Evaluate |
| Achieve Acceptable Mission |
| Improve Test Assets |
| Environment |
| Prepare Environment for an Iteration |
| Prepare Guidelines for an Iteration |
| Support Environment During an Iteration |
| Configuration & Change Management |
| Change and Deliver Configuration Items |
| Manage Baselines and Releases |
| Manage Change Requests |
| Monitor and Report Configuration Status |

Phase: Elaboration



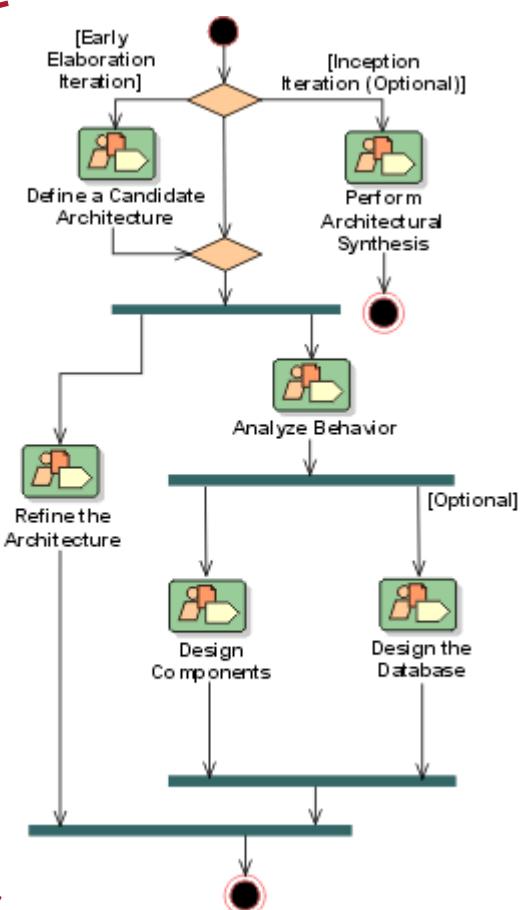
Rational Unified Process



Example: Definition of Activities and Artifacts in the RUP



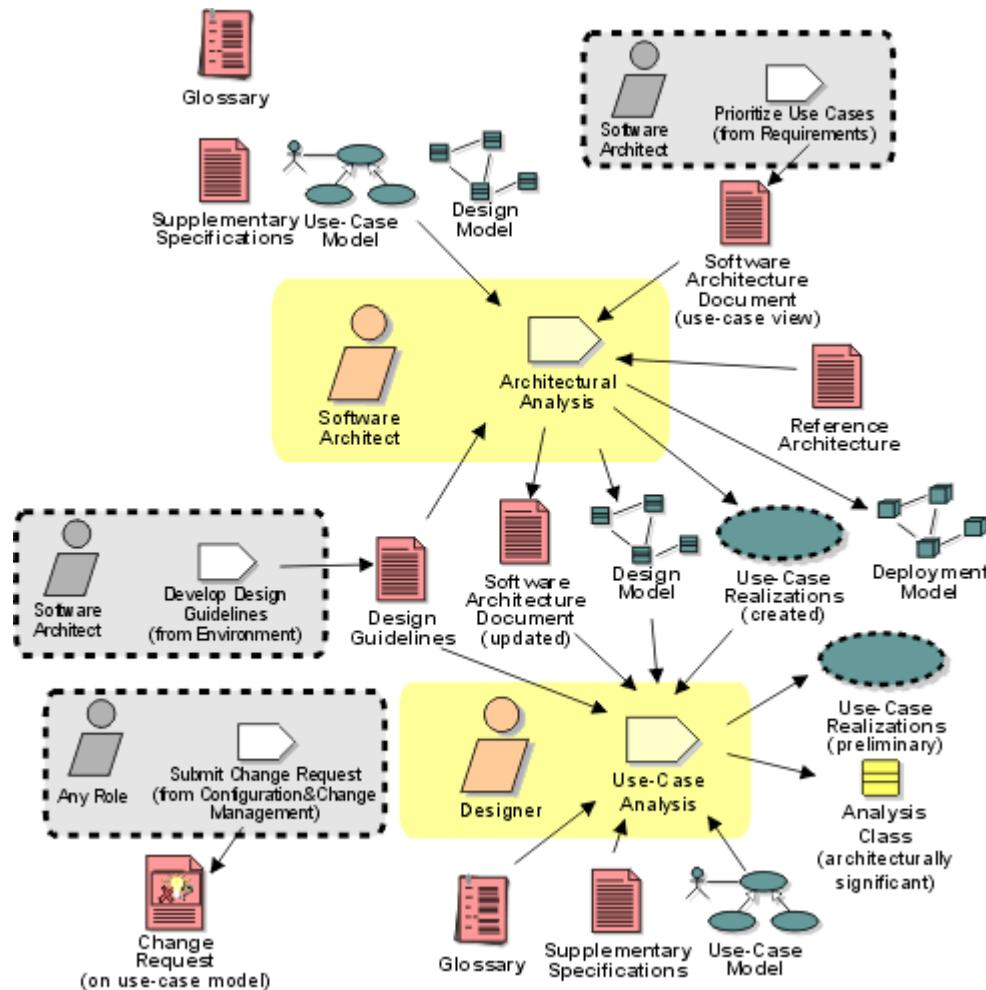
Phase: Elaboration



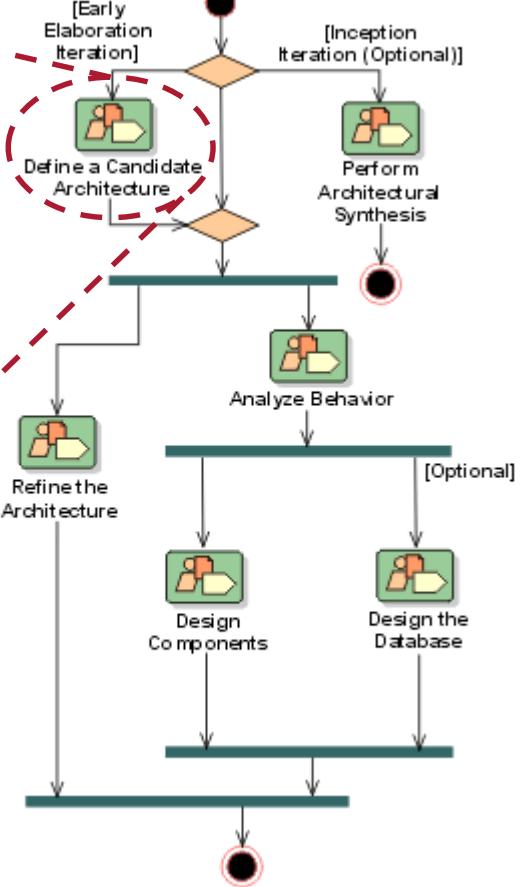
**Discipline Workflow:
Analysis & Design**



Example: Definition of Activities and Artifacts in the RUP



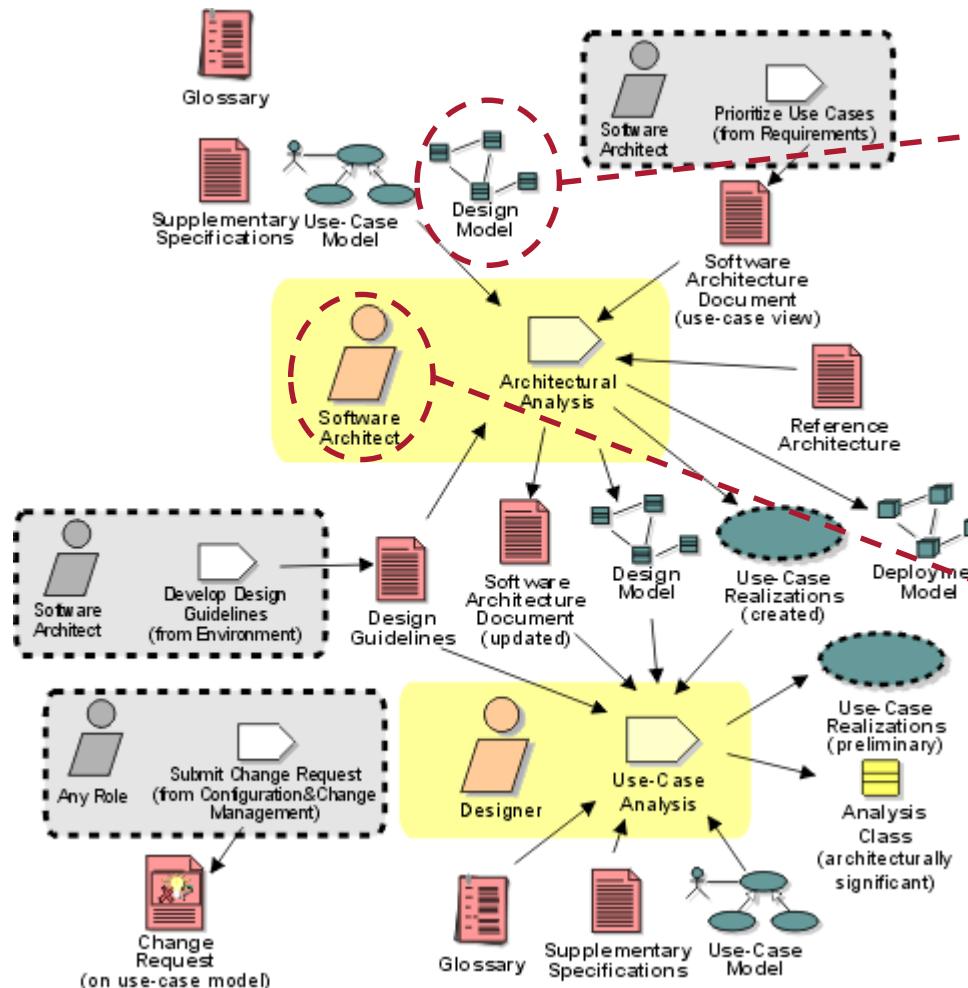
Workflow Detail:
Define a Candidate Architecture



Discipline Workflow:
Analysis & Design



Example: Definition of Activities and Artifacts in the RUP



Workflow Detail:
Define a Candidate Architecture

■ Artifact: Design Model

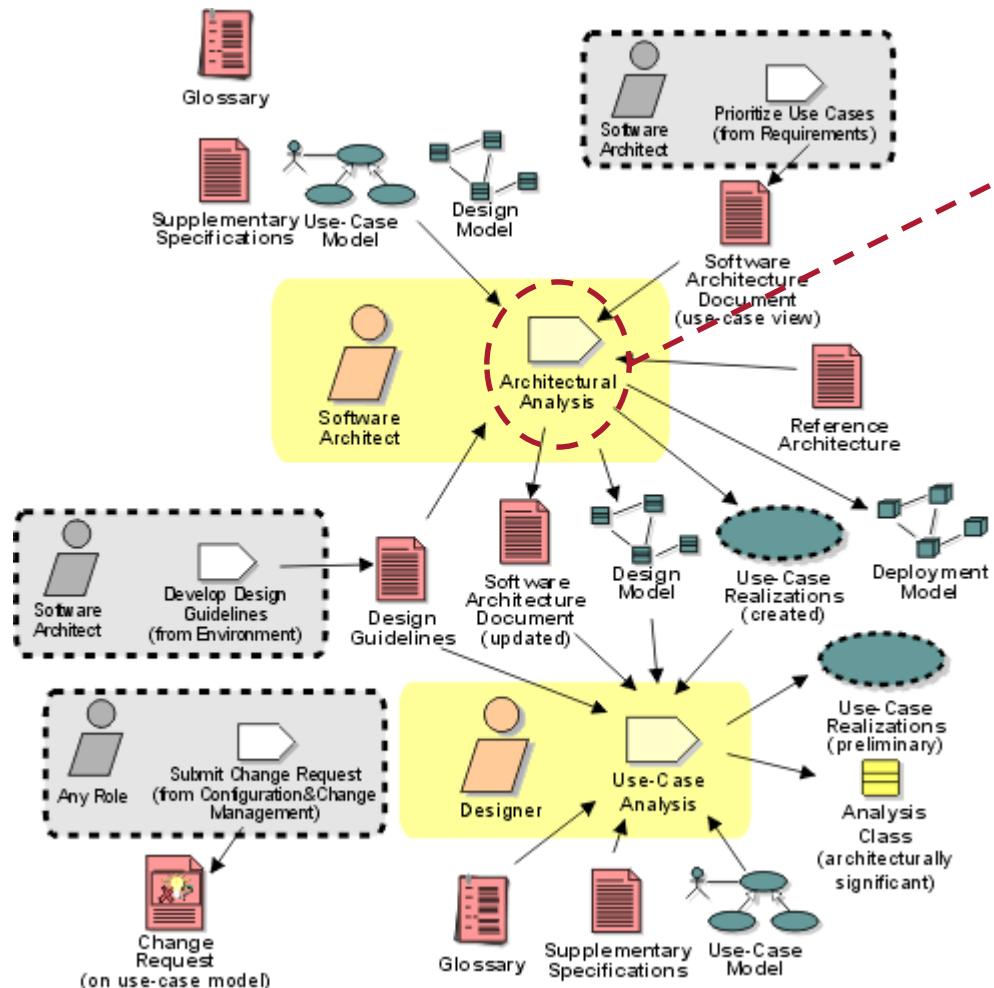
- UML Class Diagram
- UML Sequence Diagram
- UML Collaboration Diagram
- UML State Diagram
- ...

■ Role: Software Architect

- Responsibilities
- Staffing Guidelines



Example: Definition of Activities and Artifacts in the RUP

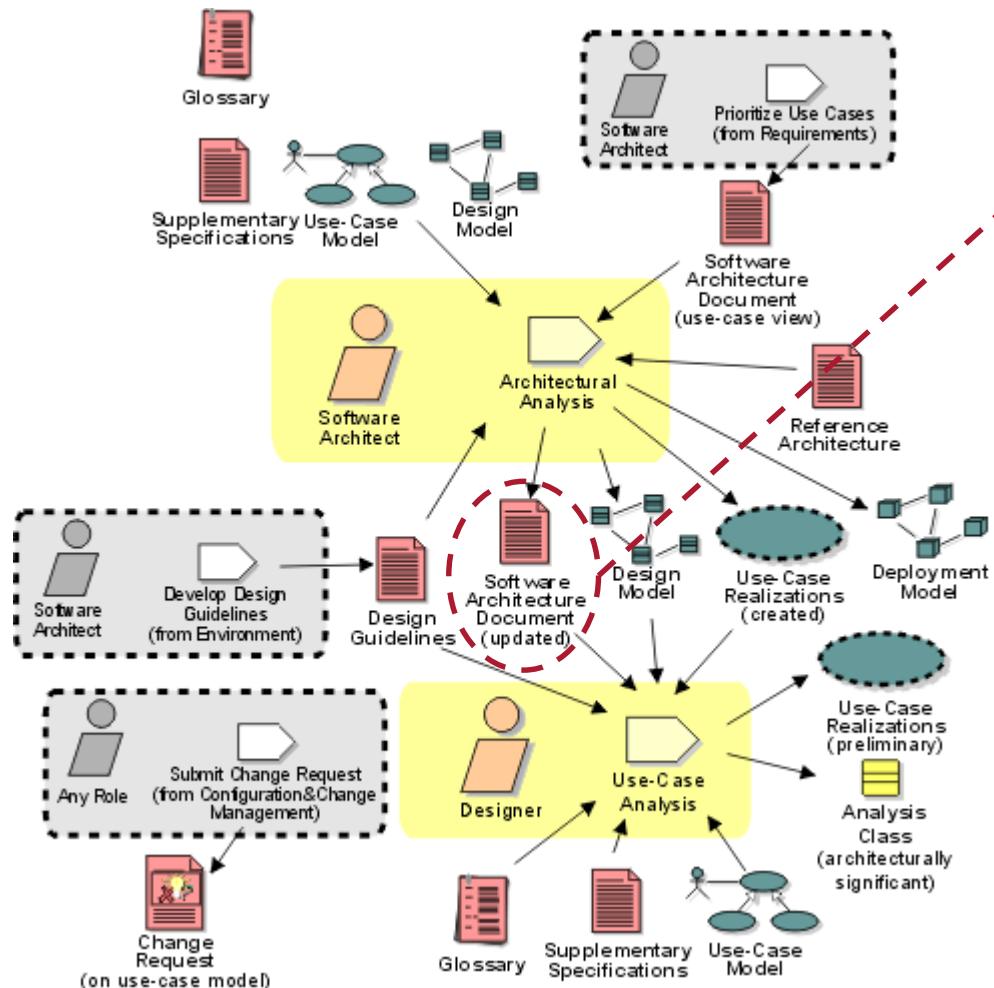


Workflow Detail:
Define a Candidate Architecture

■ Activity: Architectural Analysis

- Develop architecture overview
- Survey available assets
- Define high-level subsystem organization
- Identify key abstraction
- Develop high-level deployment model
- Identify analysis mechanisms
- Create use-case realizations
- Identify stereotypical interactions
- Review the results

Example: Definition of Activities and Artifacts in the RUP



Workflow Detail:
Define a Candidate Architecture

- **Artifact: Software Architecture Doc.**
 - Introduction
 - Architectural Representation
 - Architectural Goals and Constraints
 - Use-Case View
 - Logical View
 - Process View
 - Deployment View
 - Implementation View
 - Data View
 - Size, Performance
 - Quality

| Software Architecture Document | |
|--|--|
| 1. Introduction | (The introduction of the Software Architecture Document provides an overview of the entire Software Architecture Document, including its purpose, scope, audience, and structure of the Software Architecture Document.) |
| 2. Architectural Representation | (This section provides a comprehensive architectural overview of the system, using a mixture of different architectural notations to depict different aspects of the system. It is intended to capture and convey the overall architecture of the system.) |
| 3. Architectural Goals and Constraints | (This section defines the role of the Software Architecture Document in the overall project. It describes the goals and constraints of the system, including the requirements and constraints that must be met in order for the system to be successful.) |
| 4. Use-Case View | (This section provides a detailed description of the system's use cases, their interactions, and the system's behavior in response to those interactions.) |
| 5. Logical View | (This section provides a logical view of the system, showing how the system is organized into components and how those components interact with each other.) |
| 6. Process View | (This section provides a process view of the system, showing how the system is organized into processes and how those processes interact with each other.) |
| 7. Deployment View | (This section provides a deployment view of the system, showing how the system is deployed to different environments and how those environments interact with each other.) |
| 8. Implementation View | (This section provides an implementation view of the system, showing how the system is implemented and how it interacts with its environment.) |
| 9. Data View | (This section provides a data view of the system, showing how the system stores and retrieves data.) |
| 10. Size, Performance | (This section provides a size and performance view of the system, showing how large the system is and how well it performs.) |
| 11. Quality | (This section provides a quality view of the system, showing how well the system meets its quality requirements.) |



Tailoring the Unified Process

- Remember: Most elements of the UP are optional – tailor to project's needs!

For a lightweight project:

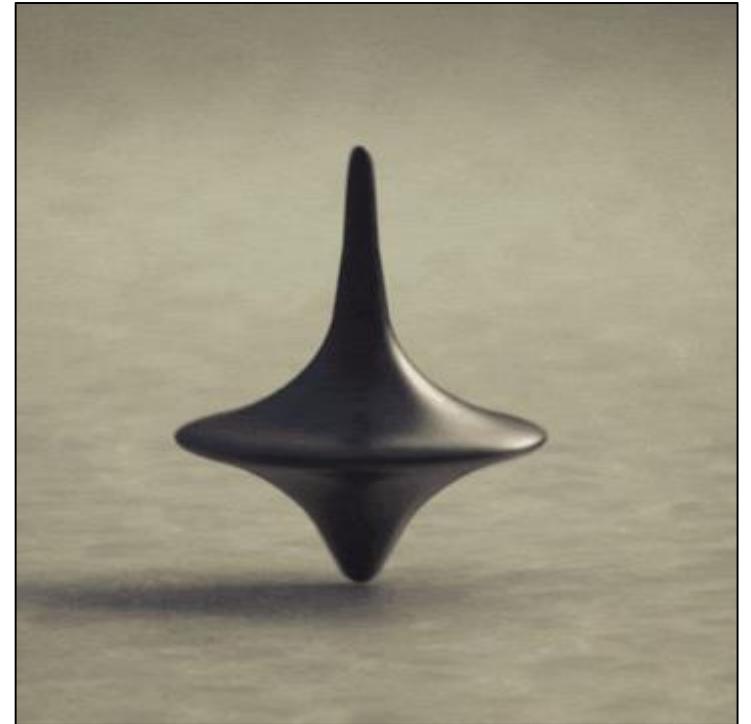
- Use only a small set of UP activities and artifacts
 - Avoid creating them unless they add value
- Don't try to complete all requirements and designs before implementation
 - In an iterative process, requirements will emerge based on feedback on increments
- Use the UML for visual modeling
 - Use simple notation and focus on those parts of the design that are not yet well understood
- Have a general Phase Plan covering just the milestones, and have a detailed Iteration Plan covering one iteration in advance
 - Detailed planning is done adaptively from iteration to iteration



Inception

see also:

- Larman: Applying UML and Patterns, Ch. 4
- Wiegers, Beatty: Software Requirements, Ch. 5



hellonama.tumblr.com



HÁSKÓLI ÍSLANDS

Matthias Book: Software Project 1

Recap: Primary Objectives of Inception Phase

- Establish the project's **software scope and boundary conditions**, including
 - an operational vision
 - acceptance criteria
 - what is intended to be in the product and what is not
- Discriminate the **critical use cases** of the system
 - i.e. the primary scenarios of operation that will drive the major design trade-offs
- Identify (and maybe demonstrate) at least one **candidate architecture**
 - against the background of some of the primary scenarios
- Estimate the **overall cost and schedule** for the entire project
 - and more detailed estimates for the elaboration phase that will immediately follow
- Estimate **potential risks** (i.e. sources of unpredictability / uncertainty)
- Prepare the **supporting environment** for the project

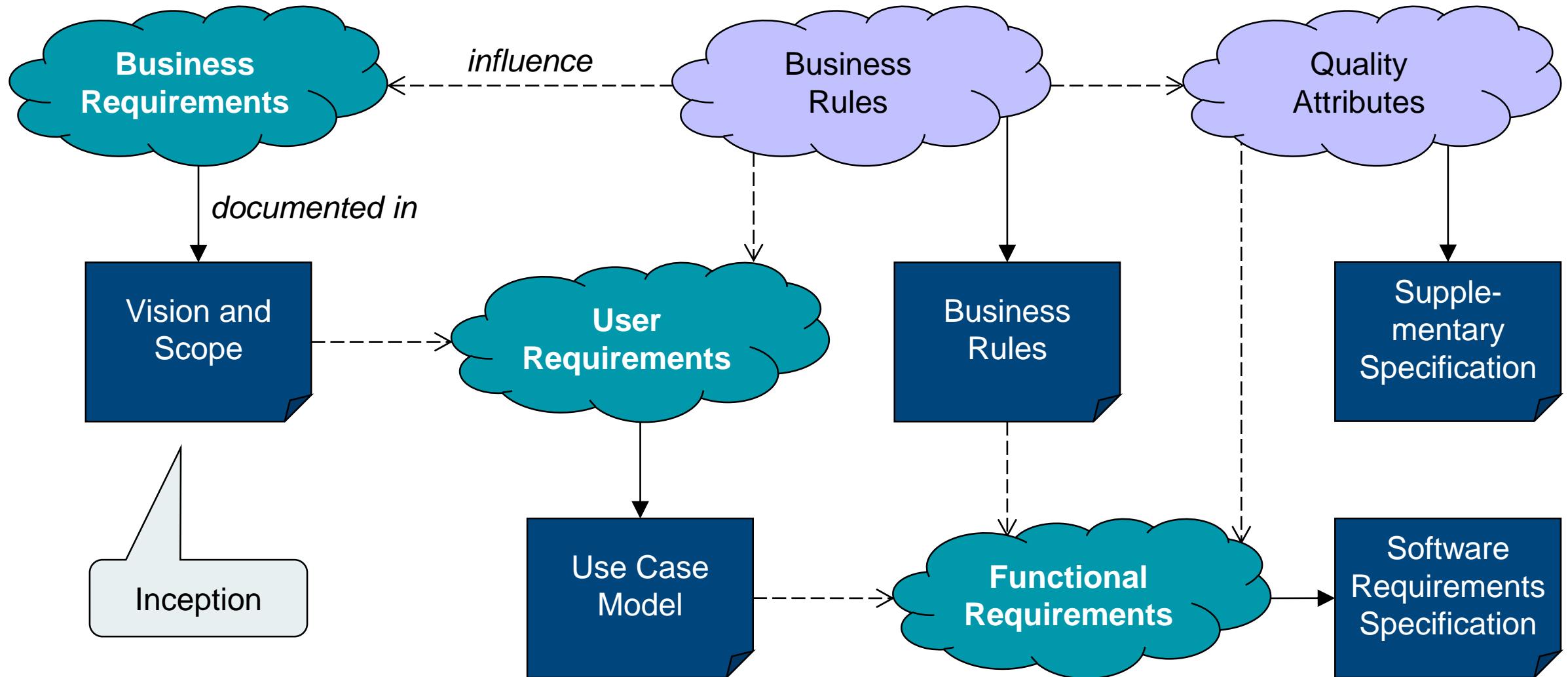


Inception in a Nutshell

- Goal: Envision the product vision, scope, and business case
 - **Do the stakeholders have basic agreement on the vision of the project, and is it worth investing in serious investigation?**
- It's not Inception if...
 - it takes more than very few weeks
 - there is an attempt to define most requirements
 - the architecture is being set in stone
 - there is no Vision and Scope document
 - estimates or plans are expected to be reliable
- Most requirements analysis occurs during Elaboration, in parallel with early production-quality programming and testing.



Requirements Levels



Business Requirements

- “Needs that lead to one or more projects to deliver desired business outcomes.”
- Come from funding sponsors, executives, marketing, product visionaries...
- Provide a reference for making decisions on requirements, designs, changes...
- Set context for / enable measurement of benefits the project hopes to achieve.

- **Product vision**

- What we ultimately want to accomplish to achieve the business objectives

Product
Vision

- **Project scope**

- What portion of the vision we will address in the upcoming project/release

Release 1
Scope

Release 2
Scope

Release 3
Scope

Release 4
Scope



Vision and Scope Document Template

1. Business requirements

1. Background
2. Business opportunity
3. Business objectives
4. Success metrics
5. Vision statement
6. Business risks
7. Business assumptions and dependencies

2. Scope and limitations

1. Major features
2. Scope of initial release
3. Scope of subsequent releases
4. Limitations and exclusions

3. Business context

1. Stakeholder profiles
2. Project priorities
3. Deployment considerations

- You can leave out sections that are not applicable to your project...
- ...but it's worth to consider carefully
 - if they are maybe relevant after all
 - how you would fill them
 - what that means for your project



Vision and Scope Document: 1. Business Requirements

- **1.1 Background (“what triggered this?”)**
 - Rationale and context for new product/system or changes made to an existing one
 - History or situation that led to decision to build this product/system
- **1.2 Business opportunity (“why does it look like we can be successful?”)**
 - Problem being solved / process being improved / market opportunity being targeted
 - Comparative evaluation of existing products/solutions and their inadequacy
 - Alignment with market trends, technology evolution, corporate strategy etc.
 - Needs of typical customers / users / target market
- **1.3 Business objectives (“what benefits do we expect out of this?”)**
 - Measurable, realistic goals that the product/solution is expected to help achieve
- **1.4 Success metrics (“how can we tell whether we are successful?”)**
 - Measurable, quantifiable criteria that indicate whether the project was worth undertaking



Vision and Scope Document: 1. Business Requirements

■ 1.5 Vision statement (“what will the product accomplish for whom?”)

- Summary of long-term purpose and intent of the product
- Reflection of the expectations of all stakeholders
- Idealistic – but grounded in realities of markets, technologies, architecture, resources etc.
- Template:
 - For [target customer]
 - Who [need/opportunity]
 - The [product name]
 - Is [product category]
 - That [major capabilities, key benefit, compelling reason to buy/use]
 - Unlike [primary competitive alternative, current system/process]
 - Our product [primary differentiation and advantage of new product]



Vision and Scope Document: 1. Business Requirements

■ 1.5 Vision statement (“what will the product accomplish for whom?”)

Example:

- **For** scientists
- **who** need to request containers of chemicals,
- **the** Chemical Tracking System
- **is** an information system
- **that** will provide a single point of access to the chemical stockroom and to vendors.
 - The system will store the location of every chemical container within the company, the quantity of material remaining in it, and the complete history of each container's locations and usage.
 - This system will save the company 25% on chemical costs in the first year of use by allowing the company to fully exploit chemicals that are already available within the company, dispose of fewer partially used or expired containers, and use a standard chemical purchasing process.
- **Unlike** the current manual ordering processes,
- **our product** will generate all reports required to comply with federal and state government regulations that require the reporting of chemical usage, storage, and disposal.

Vision and Scope Document: 1. Business Requirements

- **1.6 Business risks (“what could jeopardize the product’s success?”)**
 - Major business risks associated with developing – or not developing – the product
 - e.g. competition, timing, user acceptance, potential negative business impacts...
 - Not the same as project risks such as resource availability, technology factors etc.
 - Business risks can hit you even if your project runs smoothly from a technical perspective
 - Estimated likelihood and potential loss from each risk
 - Potential prevention/mitigation actions
- **1.7 Business assumptions/dependencies (“what are our plans based on?”)**
 - Assumptions / expectations / uncertainty about the development of external factors that form the basis for the preceding discussion of context, objectives, success metrics etc.
 - External factors that the execution / success of the project relies on (e.g. other projects’ deliverables, third party suppliers, legal regulations etc.)
 - Impacts of false assumptions and broken dependencies (these turn into risks!)



Vision and Scope Document: 2. Scope and Limitations

- **2.1 Major features (“what key things should the product be capable of?”)**
 - Major features or user capabilities (rule of thumb: the most important 10%)
 - especially those distinguishing it from previous / competing products
 - Describe e.g. as use cases
- **2.2 Scope of initial release (“what should be rolled out first?”)**
 - Beside features, scope can also comprise quality attributes (e.g. performance)
 - Focus on features that will provide the most value at the most acceptable cost to the broadest community in the earliest time frame
- **2.3 Scope of subsequent releases (“what can be rolled out later?”)**
 - Scope (and possibly rough timing) for later releases (more and more fuzzy towards future)
- **2.4 Limitations and exclusions (“what are we *not* going to do?”)**
 - Stakeholder expectations that will not be satisfied
 - Reasons for cutting aspects from the product scope



Vision and Scope Document: 3. Business Context

■ 3.1 Stakeholder profiles (“who has which interest in this project?”)

- People, groups, organizations that are
 - actively involved in the project
 - affected by its outcome
 - able to influence its outcome
- Profile should include
 - Major value or benefit the stakeholder will receive from the product
 - e.g. in terms of improved productivity, reduced rework and waste, cost savings, process optimization, automation, ability to perform new tasks, compliance with regulations, improved usability, etc.
 - Likely attitude toward the product
 - Major features and characteristics of interest
 - Known constraints to be accommodated
 - Possibly: names of key stakeholders who can serve as representatives / contacts



Vision and Scope Document: 3. Business Context

■ 3.2 Project priorities (“what room is there for compromise?”)

- Unexpected events may require changes to a project's scope, deadlines, resources, etc.
 - Example: If the product must be released a month ahead of schedule, can/should you...
 - ...defer certain requirements to a later release?
 - ...shorten planned system test cycle?
 - ...demand overtime from staff or hire additional contractors?
 - ...shift resources from other projects to help out?
- A general framework of priorities helps to decide how to best respond to such events.
 - Five dimensions that can be managed: **Features, Quality, Schedule, Cost, Staff**
 - **Constraint:** limiting factor that must be observed
 - **Driver:** significant success factor that can't be waived
 - **Degree of freedom:** some adjustment possible here
- (Not a substitute for talking to key stakeholders about best course of action when needed!)

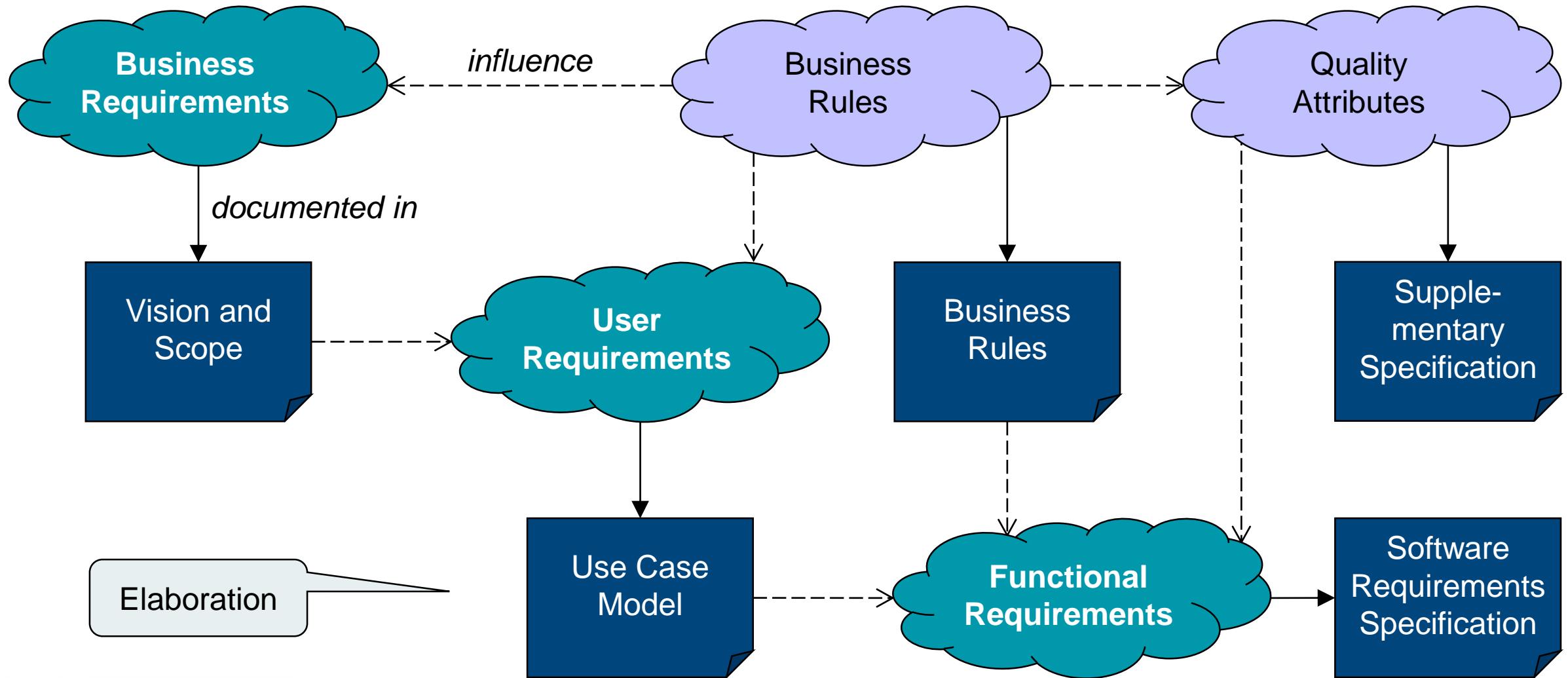
| <u>Example:</u> | Features | Quality | Schedule | Cost | Staff |
|--------------------------|----------|---------|----------|------|-------|
| Constraint | | | | X | X |
| Driver | | X | X | | |
| Degree of freedom | X | | | | |

Vision and Scope Document: 3. Business Context

- **3.3 Deployment considerations (“how will users get a hold of this?”)**
 - Not a technical deployment specification
 - Key requirements that need to be taken into account in this earliest of project stages to enable efficient deployment later on:
 - Access required by users
 - Distribution of users
 - Platforms employed by users
 - Infrastructure requirements
 - Training requirements
 - Anything affecting the preparation of successful roll-out of the product that is not a feature/characteristic of the software itself (and thus easily overlooked at first)



Requirements Levels



Types of Requirements

Functional Requirements

- Features
- Capabilities
- Business rules
- Security measures

Described in Use Case Model,
Business Rules, Software
Requirements Specification
documents

Described in Supplementary
Specification document

Quality (“non-functional”) Requirements

- Usability
 - Human factors, help, documentation
- Reliability
 - Failure rates, recoverability, predictability
- Performance
 - Response times, throughput, accuracy...
- Supportability
 - Adaptability, maintainability, internationalization...
- Constraints
 - Technology, external interfaces, operational setting, legal conditions...



Use Cases

see also:

- Larman: Applying UML and Patterns, Ch. 6



Use Cases

- In the Unified Process, we **discover and record functional requirements** by writing **text stories of an actor using a system to meet goals**.
 - **Simple example:** “Process Sale” use case (brief format)
 - A customer arrives at a checkout with items to purchase. The cashier uses the POS system to record each purchased item. The system presents a running total and line-item details. The customer enters payment information, which the system validates and records. The system updates inventory. The customer receives a receipt from the system and leaves with the items.
 - **Emphasis on users' goals and perspectives:**
 - Who is using the system?
 - What are their typical scenarios of use?
 - What are their goals?
- More user-centric than just coming up with a list of system features



Use Case Terminology

- **Actor**
 - Something with a behavior, e.g. a person (role), system or organization
 - **Primary actor:** Has user goals fulfilled through using services of system under development
 - Identify these to find user goals, which drive use cases
 - **Supporting actor:** Provides a service/information to the system under development
 - Identify these to clarify external interfaces and protocols
 - **Offstage actor:** Has interest in behavior of use case, but is not active in its execution
 - Identify these to ensure necessary interests are satisfied and conditions are met
- **Scenario** (or: use case instance)
 - A specific sequence of (inter)actions between actors and the system
 - One particular story of using a system, or one path through the use case
 - e.g. a successful path (success scenario) or a path involving an error/failure (alternate scenario)
- **Use case**
 - A collection of related success and failure scenarios describing an actor using a system to support a goal

Use Case Content

➤ Official definition in the RUP:

- A use case is a **set of scenarios** where each scenario is a **sequence of actions** a system performs that yields an **observable result of value** to a particular **actor**.

▪ Key writing challenge: Capturing the requirements precisely

- Understand and formulate the essence of the requirement
- Don't be too detailed, don't be too high-level
- Don't combine several alternatives in one scenario
- Don't forget to consider and describe alternate scenarios
- Don't interpolate what you're not sure about – ask!
- Don't preempt technical design decisions



Use Case Formats

- **Brief format**

- Terse one-paragraph summary, usually of the main success scenario
- Written during early requirements analysis to get a quick sense of subject and scope

- **Casual format**

- Multiple informal paragraphs that cover various scenarios
- Typically including at least the main success scenario and most relevant alternate scenarios
- Written during early requirements analysis to get a quick sense of subject and scope

- **Fully dressed format**

- All steps and variations described in detail, with supporting sections on preconditions etc.
- During first requirements workshop, about 10% of architecturally significant, high-value use cases are written out in this detail; rest may follow in further iterations as needed.

- **UML use case diagrams**

- Secondary, supplementary illustrations only – essential information is contained in the text!



Example: Casual Use Case Format



“Handle Returns” use case

■ Main success scenario

- A customer arrives at a checkout with items to return. The cashier uses the POS system to record each returned item. [...]

■ Alternate scenarios

- If the customer paid by credit, and the reimbursement transaction to their credit account is rejected, inform the customer and pay them with cash.
- If the item identifier is not found in the system, notify the Cashier and suggest manual entry of the identifier code (perhaps it is corrupted).
- If the system detects failure to communicate with the external accounting system, [...]



Fully Dressed Use Case Format Template

1. Use case name

- What are these scenarios about? (Start with a verb.)

2. Scope

- What is the system under design?

3. Level

- Is this use case describing a user's goal (top level) or a subfunction of some other use case?

4. Primary actor

- Who calls on the system to deliver its services?

5. Stakeholders and interests

- Who cares about this use case, and what do they want?

6. Preconditions

- What must be true on start (and worth telling the reader)?

7. Success guarantee

- What must be true on successful completion (and worth telling the reader)?

Fully Dressed Use Case Format Template

8. Main success scenario

- What is the typical, unconditional “happy path” scenario of success?

9. Extensions / alternate scenarios

- What are alternate scenarios of success or failure?

10. Special requirements

- What are related non-functional requirements?

11. Technology and data variations list

- What varying input/output methods and data formats should we be aware of?

12. Frequency of occurrence

- How often does this use case occur? (Influences investigation, timing, priority, testing...)

13. Miscellaneous / open issues

- What open issues are there?
- Template can be adapted to your project’s needs



Example: Fully Dressed Use Case Format



- **Name:** UC1: Process Sale
- **Scope:** NextGen POS application
- **Level:** User goal
- **Primary actor:** Cashier
- **Stakeholders and interests:**
 - **Cashier:** Wants accurate, fast entry, and no payment errors, as cash drawer shortages are deducted from salary.
 - **Salesperson:** Wants sales commissions updated.
 - **Customer:** Wants purchase and fast service with minimal effort. Wants easily visible display of entered items and prices. Wants proof of purchase to support returns.
 - **Manager:** Wants to be able to quickly perform override operations, and easily debug Cashier problems.
 - **Government Tax Agencies:** Want to collect tax from every sale. May be multiple agencies, such as national, state, and county.
- **Company:** Wants to accurately record transactions and satisfy customer interests. Wants to ensure that Payment Authorization Service payment receivables are recorded. Wants some fault tolerance to allow sales capture even if server components (e.g. remote credit validation) are unavailable. Wants automatic and fast update of accounting and inventory.
- **Payment Authorization Service:** Wants to receive digital authorization requests in the correct format and protocol. Wants to accurately account for their payables to the store.
- **Preconditions:** Cashier is identified and authenticated.
- **Success guarantee:**
 - Sale is saved.
 - Tax is correctly calculated.
 - Accounting and inventory are updated.
 - Commissions recorded.
 - Receipt is generated.
 - Payment authorization approvals are recorded.

Example: Fully Dressed Use Case Format



- **Main success scenario:**

1. Customer arrives at POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total. Price calculated from a set of price rules.

Cashier repeats steps 3-4 until indicates done.

5. System presents total with taxes calculated.
6. Cashier tells Customer the total, and asks for payment.
7. Customer pays and System handles payment.
8. System logs completed sale and sends sale and payment information to the external Accounting system (for accounting and commissions) and Inventory system (to update Inventory).
9. System presents receipt.
10. Customer leaves with receipt and goods (if any).

- **Alternate scenarios:**

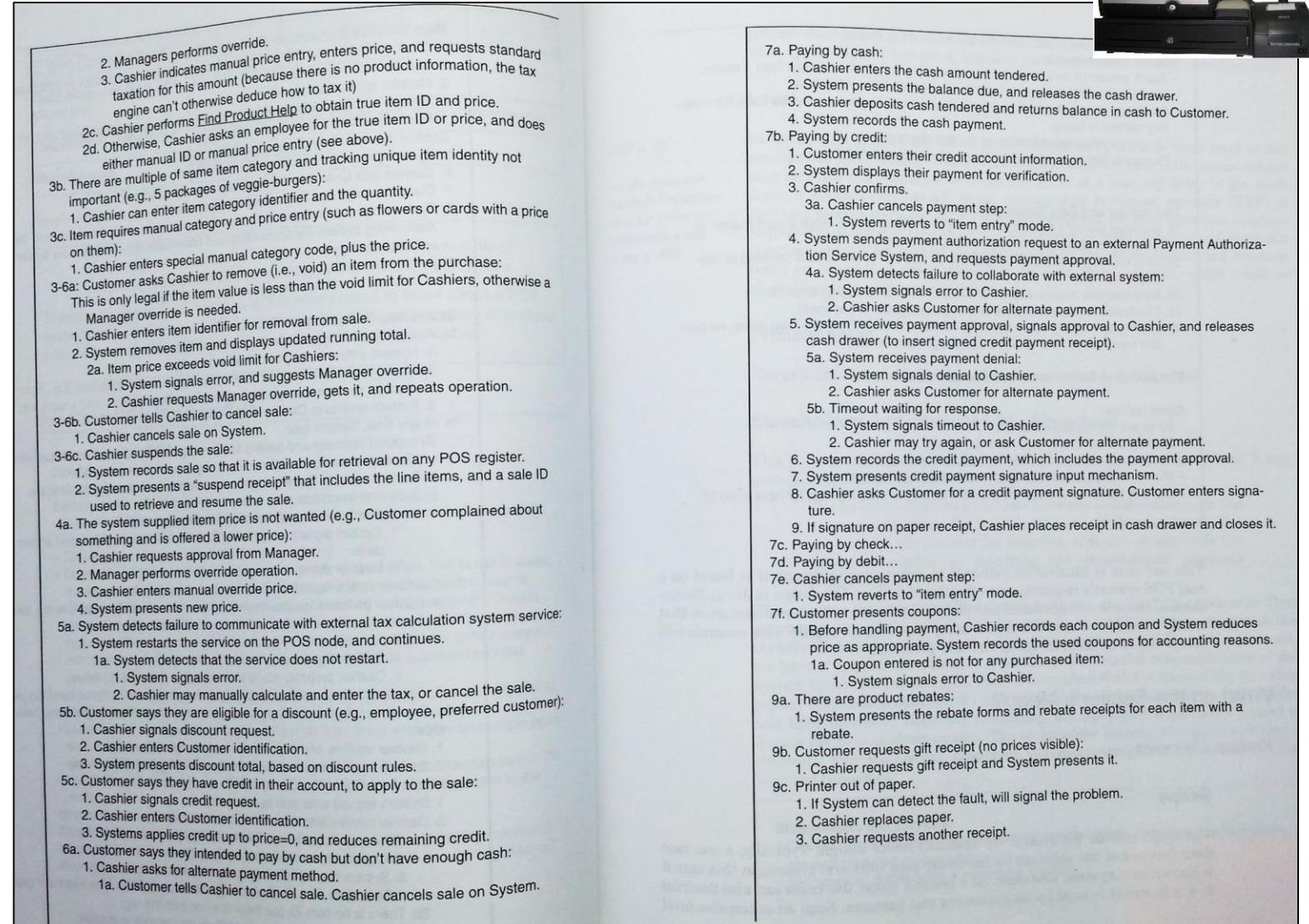
- (a) At any time, Manager requests an override operation:
 1. System enters Manager-authorized mode.
 2. Manager or Cashier performs one Manager-mode operation, e.g. cash balance change, resume a suspended sale on another register, void a sale, etc.
 3. System reverts to Cashier-authorized mode.
- (2-4) Customer tells Cashier they have a tax-exempt status (e.g. seniors, native peoples)
 1. Cashier verifies, and enters tax-exempt status.
 2. System records status (which it will use during tax calculations).
- (3a) Item requires manual category and price entry (e.g. flowers, gift cards).
 1. Cashier enters special manual category code, plus the price.
- (3b) Item requires weighing.
 1. Cashier places item on scale.
 2. System weighs item, records weight and calculates price.
- [...]

Example: Fully Dressed Use Case Format



■ Further extensions:

- System failure
- Resume suspended sale
- Invalid item ID
- Bundled items
- Remove item from sale
- Cancel sale
- Suspend sale
- Discount item
- Product rebate
- Apply customer credit
- Pay by cash
- Pay by credit card
- Pay (in part) with coupons
- [...]



Example: Fully Dressed Use Case Format



- **Special requirements:**
 - Touch screen UI on a large flat panel monitor. Text must be visible from 1 m.
 - Credit authorization response within 30 seconds 90% of the time.
 - Somehow, we want robust recovery when access to remote services such as the inventory system is failing.
- **Technology and data variations:**
 - (a) Manager override entered by swiping an override card through a card reader, or entering an authorization code via the keyboard.
 - (3a) Item identifier entered by car code laser scanner (if bar code is present) or keyboard.
 - (3b) Item identifier may be any UPC, EAN, JAN, or SKU coding scheme.
- (7a) Credit account information entered by card reader or keyboard.
- (7b) Credit payment signature captured on paper receipt. But within two years, we predict many customers will want digital signature capture.
- **Frequency of occurrence:** could be nearly continuous
- **Open issues:**
 - What are the tax law variations?
 - Explore the remote service recovery issue.
 - What customization is needed for different businesses?
 - Must a cashier take their cash drawer when they log out?
 - Can the customer directly use the card reader, or does the cashier have to do it?

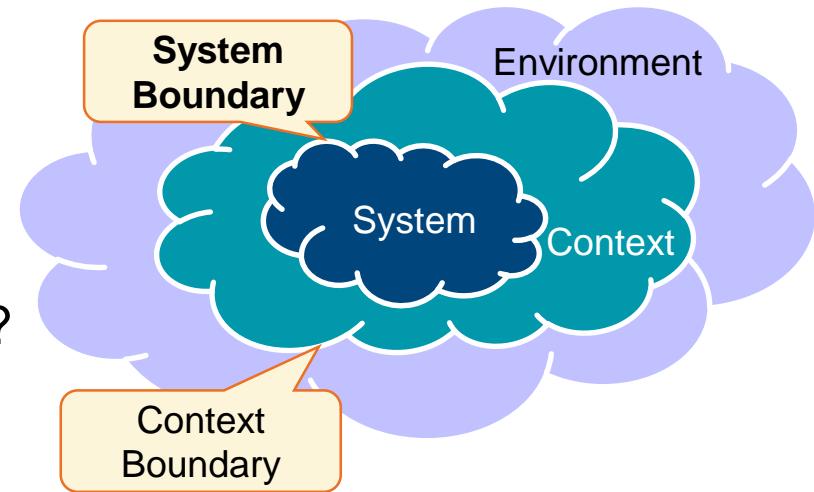
Finding Use Cases

Note: Not a linear process!

The steps influence each other and will be repeated in iterative-incremental fashion.

1. Choose the system boundary.

- Which aspects of the business domain will be covered/performed by your system?
- Which aspects of the business domain will influence/interact with your system?
- If you're not sure what's in the system, it may be easier to identify what is outside it.



2. Identify the primary actors.

- Who has goals fulfilled through using services of the system?
- Who performs business administrative tasks?
- Who performs technical administrative tasks?
- Beside human primary actors, are there external systems that use services of our system?
- Is “time” an actor because the system reacts to certain events (e.g. deadline expiry)?
- By whom is the system’s lifecycle handled (starting, stopping, restarting, error conditions...)?

Finding Use Cases

Note: Not a linear process!

The steps influence each other and will be repeated in iterative-incremental fashion.

3. Identify the goals for each primary actor.

- Note: Actors have goals and use software to help satisfy them.
- Don't ask people: "What are the system's tasks?" or "What do you do?"
 - This will just reflect current solutions and procedures, which may not be optimal.
- Rather, ask first: "Who uses the system and what are their goals?"
- Ask actors next: "What are your goals whose results have measurable value?"
 - This will open up the vision for new and improved solutions, focus on adding business value, get to the core of what stakeholders want from the system.
- Alternatively/additionally: Consider...
 - which real-life events can occur
 - which actors are involved in them
 - what purpose/goal they serve

4. Define use cases that satisfy user goals.

- Write one use case per goal (including its success and failure scenarios).
- Exception: Combine distinct "create/retrieve/update/delete" goals for certain data X into one "manage X" use case



Testing Use Cases' Validity

- **What is a valid use case?**

- Negotiate a Supplier Contract
- Handle Returns
- Log In
- Move Piece on Game Board

- One might say these are all use cases on different levels of detail.

- But what is helpful for the purpose of understanding what we need to build?

➤ **A valid use case describes scenarios which**

- Create results that are of **value** to a particular stakeholder
- Represent **elementary business processes**
- Are so **complex** that they require several pages to be described in fully dressed format

Testing Use Cases' Validity

■ Boss test

- Would your boss be happy if you told him you performed this use case all the time?
- Checks whether the use case describes actions that **create value**.

■ Size test

- Most use cases are on a **time scale** of a few minutes to an hour.
 - Neither something done in one small step, nor something taking several sessions or days to complete is likely a valid use case.
- Make sure it's not just one step of a more complex use case.
 - If the fully dressed format is less than one **page** long, it's probably not a use case.

■ EBP test

- Does the use case represent an elementary business process (EBP)?
 - A task performed by one person in one place at one time,
 - in response to a business event,
 - which adds measurable business value
 - and leaves the data in a consistent state.

■ What is a valid use case?

- ~~Negotiate a Supplier Contract~~
- Handle Returns
- ~~Log In~~
- ~~Move Piece on Game Board~~



Testing Use Cases' Validity

- **Do not take the tests too literally – use common sense!**
- A task requiring two or three people can still be a valid use case, even if it violates the strict EBP definition.
- A very complex or frequently recurring sub-task (e.g. “pay by credit card”) can be reasonably written in its own use case, so it can be referred to from several other use cases even if it violates the size or EBP test.
- A feature such as “authenticate user” may not pass the boss test, but still require careful analysis that warrants description as a use case of its own.
- *The purpose of use cases is to **understand** what we need to build.*



Reality Check

- **Caution:** Written specifications and models can easily give the **illusion** of correctness and completeness.
 - However, in some places, these specs are virtually guaranteed to be flawed, ambiguous, missing critical information, misunderstood by readers...
 - It is neither feasible nor economical to try to remedy this by a huge initial requirements and design effort.
 - Neither does it benefit a complex project to forgo requirements engineering altogether and rush straight to coding.
- **Use an iterative-incremental approach:** Key requirements, architectural baseline, more requirements, more implementation etc.
- **Strive for clear, precise language:** The writing process will prompt you to clarify your requirements.



Use Case Writing Guideline: Essential, UI-free Style

- **Describe the user's intentions and the system's responsibilities, rather than their concrete actions.**
- Do not include user interface details in the use case.
- Consider what is the “root goal” that the user is striving to accomplish.

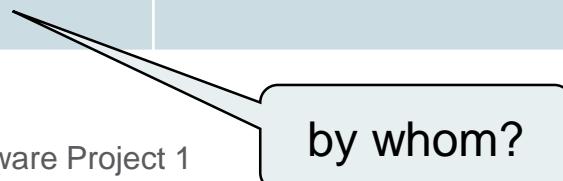
| Bad Example | Good Example | |
|--|---|---|
| <ol style="list-style-type: none">1. Administrator enters ID and password in dialog box (see Fig. 3).2. System authenticates Administrator.3. System displays “Edit Users” window (see. Fig. 4). | <ol style="list-style-type: none">1. Administrator identifies self.2. System authenticates identity. | <p>No technology presumptions: Could use password, RFID chip, biometrics, ...</p> |

- Helps to focus on the problem domain, rather than the solution domain
- Allows developers come up with new solutions and adapt to different platforms

Use Case Writing Guideline: Terse, Active Voice

- Write in crisp, terse style.
- Avoid “noise” words.
- Abbreviations are fine, but have a central place in the document to define them.
- Make sure you don’t eliminate relevant content, and dependencies are clear.
 - Under which conditions does something happen?
 - Where does data go? Who depends on certain data?
- Use active instead of passive voice.

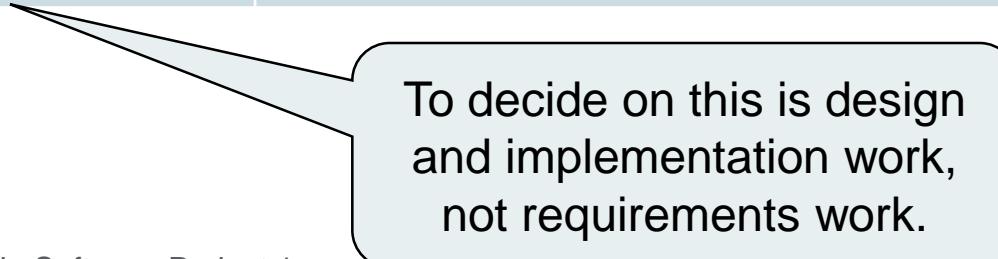
| Bad Example | Good Example |
|--|--|
| 1. The user's identity is authenticated. | 1. System authenticates user's identity. |



Use Case Writing Guideline: Black-Box Style

- Do not describe the internal workings of the system, its components, or design.
- Focus on describing the system's responsibilities.
 - Note: This corresponds to object-oriented thinking: Software components have responsibilities and collaborate with other components, but don't expose to each other how exactly they fulfill their responsibilities.
- Describe **what** the system will do, but **not how** it will do it.

| Bad Example | Good Example |
|--|---|
| <ol style="list-style-type: none">1. The system writes the sale to a database...2. The system generates an SQL INSERT statement for the sale... | <ol style="list-style-type: none">1. The system records the sale. |



To decide on this is design and implementation work, not requirements work.



Use Case Writing Guidelines: Actor-Goal Perspective

- Write requirements focusing on the users/actors of a system, asking about their goals and typical situations.
- Focus on understanding what the actor considers a valuable result.
- Avoid collecting plain “feature lists”, but consider who is using the system for what purpose.
 - Feature lists easily lead to “gold plating”, where every stakeholder tries to include their pet features, regardless of whether users need or want them.
- Focusing on the actors’ goals helps to
 - capture better what the system actually needs to accomplish
 - evaluate better how much priority should be assigned to a certain requirement



Team Assignment 1

- Two of the most important artefacts created in the Inception phase are:
 1. **The Vision and Scope document**
 - Describing what you want to build, what its key features/capabilities will be, who will use it...
 - Imagine having to convince your boss or an investor to provide funding for the project
 - What would they want to know to be convinced?
 2. **The initial Use Case document**
 - Describing the most important use cases of your product
 - i.e. the primary things that your system is supposed to be able to do
- Producing these documents and explaining the considerations that went into them will be your job in Team Assignment 1.



Team Assignment 1: Content

- By **Sun 20 Sep**, submit in Uglá:
 - A **Vision and Scope document** for your project
 - Following the template on slide 19 – most importantly, sections 1.5, 2.1 and 3.1
 - Use brief use case format for the 3-5 most important use cases in section 2.1
 - Try to come up with meaningful considerations for the other sections as well
 - (“Convince your boss” – there are e.g. business considerations for games, too!)
 - An initial **Use Case document** (text-only, no UML diagrams)
 - Describe the 2 most complex use cases mentioned in the Vision document in fully dressed format
 - Describe the other use cases mentioned in the Vision document in casual format
 - If you have further ideas for secondary / supplementary use cases, describe them in brief format
 - But no need for a complete list of use cases here – more requirements work will be done in Elaboration
- On **Thu 24 Sep**, present and **explain** your documents to your tutor:
 - What considerations are behind your vision and scope document? Why did you leave out certain sections of the template? How did you come up with your use cases? Why are these the most important ones? etc.



Team Assignment 1: Format

- The **Vision & Scope and Use Case documents** must
 - be produced by all team members together
 - be submitted as PDF documents **by Sun 20 Sep in Ugla**
 - contain your team number, the names and kennitölur of all team members
- Only the team member who will present must submit a PDF for the whole team.
 - Don't submit multiple versions – we'll just grade the first one we encounter!
- The **presentation**
 - should be given by one representative of the team (a different one for each assignment!)
 - should be based on the submitted document (don't prepare extra slides!)
 - should take around 5-10 minutes (plus some questions asked by the tutor)
- All team members receive the same grade, with some variation for the presenter



Gangi þér vel!

book@hi.is



HÁSKÓLI ÍSLANDS
VERKFRÆÐI- OG NÁTTÚRVÍSINDASVIÐ

IÐNAÐARVERKFRÆÐI-, VÉLAVERKFRÆÐI-
OG TÖLVUNARFRÆÐIDEILD





Project Management – Guest Lecture

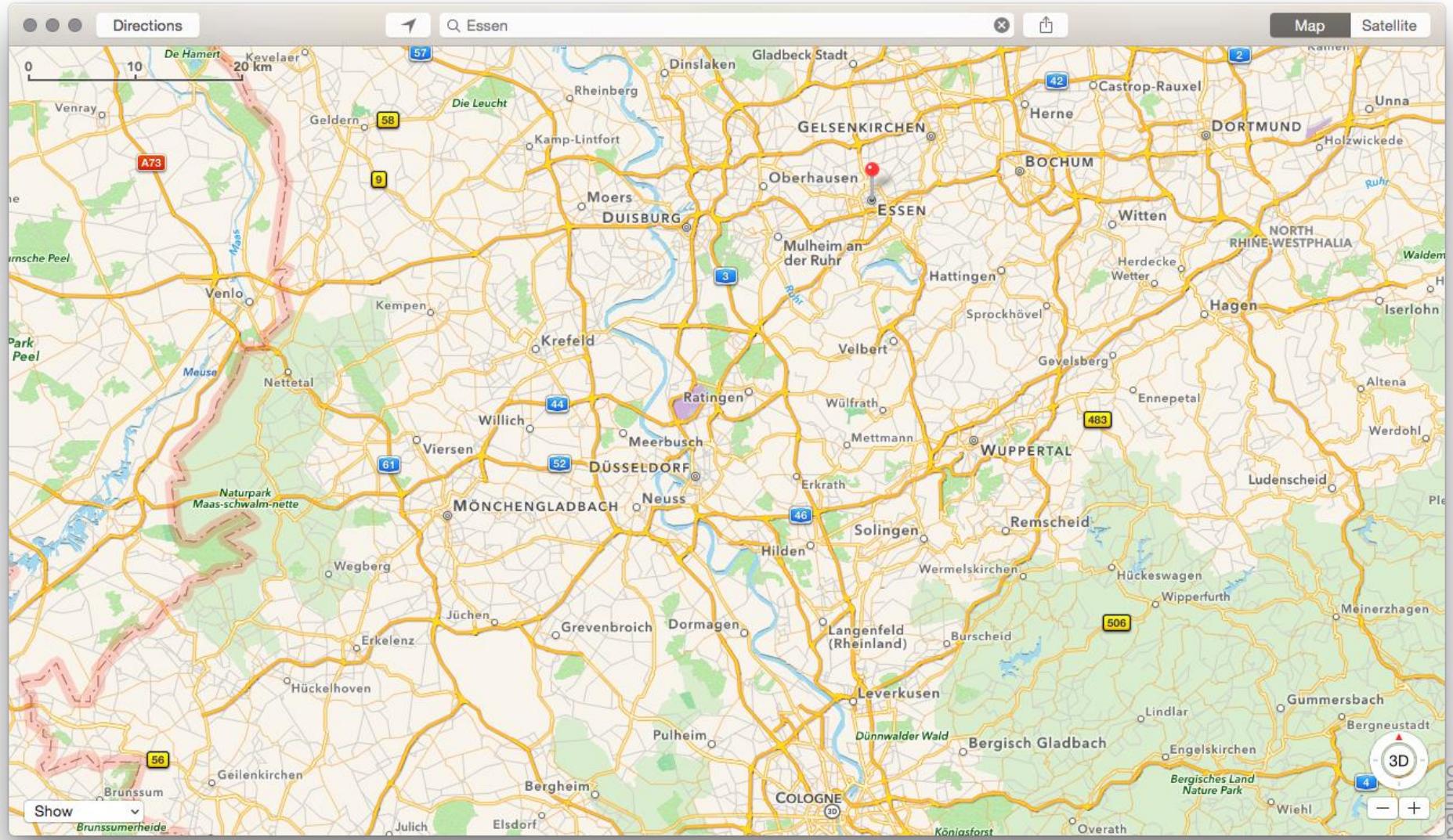
Julia Hermann, Marc Hesenius

paluno – The Ruhr Institute for Software Technology

- University of Duisburg-Essen
 - 12 departments
 - 39,000 students
 - One of the youngest and largest universities in Germany



paluno – The Ruhr Institute for Software Technology



paluno – The Ruhr Institute for Software Technology



Prof. Dr.
Frederik Ahlemann



Prof. Dr.
Stefan Eicker



Prof. Dr.
Michael Goedicke



Prof. Dr.
Volker Gruhn



Prof. Dr.
Maritta Heisel



Prof. Dr.
Thomas Herrmann



Prof. Dr.
Pedro J. Marrón



Prof. Dr.
Klaus Pohl

paluno – The Ruhr Institute for Software Technology

- Founded: Feb. 2010
- 8 Professors and ~100 Researchers
- Covers all aspects of software technology

- Prof. Dr. Volker Gruhn
 - Software Engineering, esp. mobile Applications
 - <http://se.paluno.uni-due.de>
- Research Focus and Projects:



Introduction

Marc Hesenius



- Head of *Mobile Interaction* Research Group
- Research Focus: Specification of Gesture-Based Interaction



Julia Hermann



- Research Focus: Bringing Social Aspects to Software Engineering
- Software affects many parts of today's life
- Ongoing digitalization will further increase societies dependency on technology – what does this mean for people without technical background?

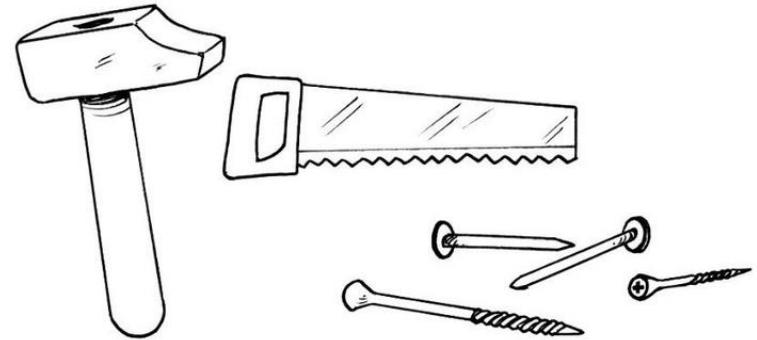
Our Lecture

Today

- Some foundations and exercises
- Different project and management aspects:
 - Tasks and work packages
 - Risk management
 - Planning and reporting
- Station-Based Learning:
Piazza

Tomorrow

- Practical part – just do it!



Station-based Learning

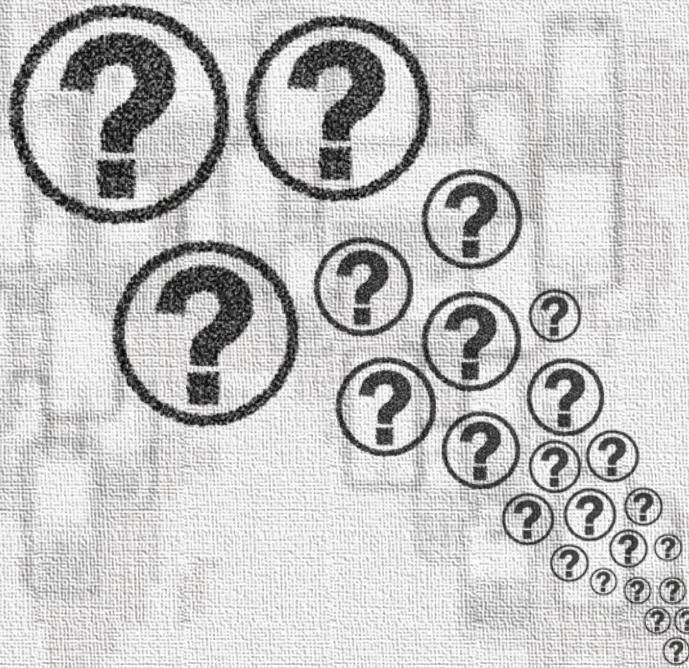
Piazza

- Form small groups (4-5 people) at the tables
- At each table, you will find envelopes with all materials – each envelope represents a station
- Each station contains a sheet of tasks and several content sheets – the tasks sheet explains everything
- After 30+10 minutes, switch to another table and work on the next station – we will announce all changes



Questions

Thank you
and enjoy!



Project Management

Worksheet Station 1: Tasks, subtasks and work packages

Exercise 1

One of your group reads the text aloud. Discuss the content and ask questions in your group. Did you understand everything? How does the content relate to your experience?

Exercise 2

Your best friends Peter and Mary are getting married and you, as a witness to their marriage, are responsible for organizing the wedding ceremony and feast.

1. Complete the work breakdown structure for the project "P&M's Wedding" on the following page. Add the necessary subtasks and work packages.

2. Analyze the tasks and look for dependencies between them. Define the dependency types you find.

Exercise 3

Think about the specifications you need from your "customers" Peter and Mary. What information do you need to successfully organize the wedding?

Identify three major milestones for the project "wedding". Sketch a milestone plan and determine the dependencies between the milestones.

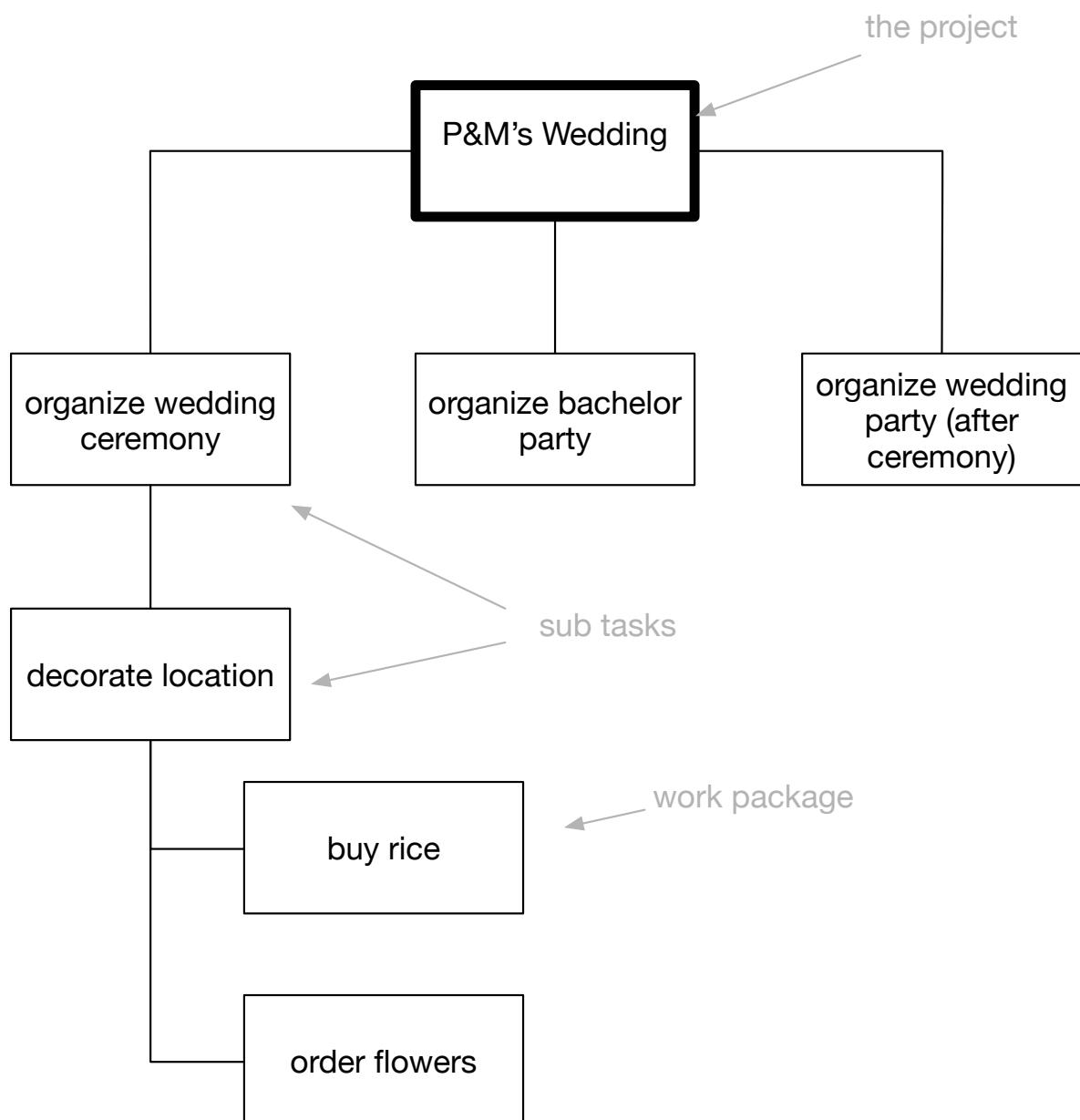
Exercise 4

Think about tasks for the project "PM-Wedding" that could use a "Definition of Done". Use the following guidelines to create a DoD.

| Example |
|---|
| Checkliste Definition of Done for: |
| Prepare Meeting |
| <input type="checkbox"/> Set time and date |
| <input type="checkbox"/> Set agenda |
| <input type="checkbox"/> Reserve meeting room |
| <input type="checkbox"/> Organise catering |
| <input type="checkbox"/> Invite participants |

Project Management

Work breakdown structure:



Milestones

- Milestones divide projects into timed phases and can also be used to coordinate sub-projects.
- A milestone is a special project event (e.g. completion of an important task like „first prototype implemented“, „customer accepts UI mockups“, or „release to manufacturing“)
- A milestone is at least described by the event's name. Typically, milestones have a final date to improve project planning and control.
- All milestones with their respective dates represent the project's roadmap

Subtasks and Work Packages

Projects are structured in Subtasks and work packages.

Subtasks are tasks defining enclosed project parts and can be further refined.

Work Packages are tasks that cannot be refined any further (smallest possible task).

Additional information can be added:

- Responsibility
- Results / Deliverables
- Dependencies to other tasks

Dividing projects into subtasks is a core element of project mgmt allowing collaborative working.

Beware of micro mgmt! Project managers and leaders define **WHAT** to do, now **HOW** to do things.

Self-organizing teams can take care of implementation details but need clear communication of project goals.

Definition of Done (DoD)

Short checklist defining when a work package / task / project is done.

The DoD is defined at project start but is subject to change over project lifetime when more requirements are elicited.

Sample content:

- Document reviewed by two independent persons
- Customer accepted changes
- File was published with open access

Defining a thorough DoD has several advantages:

- support communication between team members by defining clear criteria to finalize tasks
- ensure high quality standards for all results
- avoid unnecessary details and reduce effort

Project Management

Worksheet Station 2: Risk

Exercise 1

One of your group reads the text aloud. Discuss the content and ask questions in your group. Did you understand everything? How does the content relate to your experience?

Exercise 2

Your best friends Peter and Mary are getting married and you, as a witness to their marriage, are responsible for organizing the wedding ceremony and feast.

1. Identify eight potential risks for the project "PM-Wedding" and assign them to the risk categories displayed on posters.

| No. | Risk | Risk categories |
|-----|------|-----------------|
| 1 | | |
| 2 | | |
| 3 | | |
| 4 | | |
| 5 | | |
| 6 | | |
| 7 | | |
| 8 | | |

Project Management

2. Determine for each identified risk the probability and the potential level of damage. Use the following risk matrix:

| | | | |
|--------|-----|--------|-----------------|
| | | | |
| high | | | |
| medium | | | |
| low | low | medium | high |
| | | | level of damage |

Exercise 3

Detect in your group eight work packages. Prioritize them by entering them in the risk value matrix. Prioritize first by your own and discusses the result in your group.

| | | |
|------|-----|-------|
| | | |
| high | | |
| low | low | high |
| Risk | | Value |

Exercise 4

Discuss why tasks with high risk and high value on the risk value matrix must be implemented first and tasks with high value and low risk second. Is this a good practice? Can you think of alternatives?

Project Risks

Project risks are threads to a project's success. A risk analysis is conducted to identify and rate risks and plan proper countermeasures.

Typical categories for project risks:

- resource risks, e.g. employee illness or resignation, machine failure
- deadline risks, e.g. missing agreed dates
- acceptance risks, e.g. users reject the finally shipped product
- cost risks, e.g. customer bankruptcy
- technical risks, e.g. introduction of new technologies and methods
- quality risks, e.g. final product does not match expected quality

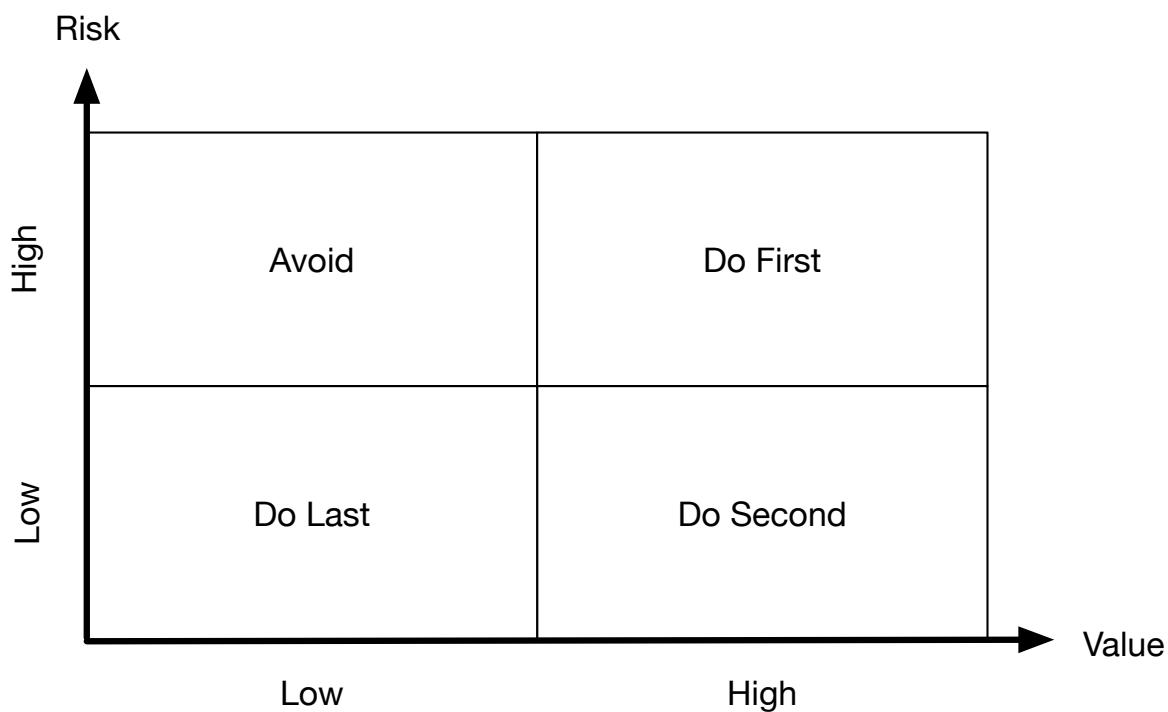
Identifying proper risks highly depends on experience and creativity. Risks are important to prioritize tasks and identify potential conflicts and problems.

Project Risk Rating

Two core questions help to rate project risks:

- How likely is the identified risk to occur (probability of occurrence)?
- How high is the potential damage?

A simple method to prioritize risks is the Risk Value Matrix, where value means the contribution to the project's goals:



Probability of occurrence and potential damage are either estimated or based on statistical data.

Countermeasures

Countermeasures depend on risk type and rating. Typical strategies are:

- risk avoidance by e.g. cutting project goals
- risk reduction by e.g. optimizing project goals
- risk transferal by e.g. moving them to the customer
- risk acceptance by e.g. agreeing to possible consequences (only for insignificant risks!)

Risks are a natural part of projects and cannot be avoided at all. It is crucial however to identify relevant risks as early as possible and act accordingly.

Beware! Risks might change during a project's lifetime, thus constant risk analysis and definition of countermeasures is a constant activity.

Project Management

Worksheet Station 3: Status reports and project planning

Exercise 1

One of your group reads the text aloud. Discuss the content and ask questions in your group. Did you understand everything? How does the content relate to your experience?

Exercise 2

Arrange each project situation in the table with color according to the RAG status and a category. Use the following explanation to determine the appropriate RAG status:

| Cost | <ul style="list-style-type: none"> ▪ EAC <= BAC ▪ EAC <= BAC + risk addition | <ul style="list-style-type: none"> ▪ EAC > BAC + risk addition | |
|----------------------|---|--|--|
| Time | <ul style="list-style-type: none"> ▪ Last milestone in time AND ▪ Future milestones expected to be in time | <ul style="list-style-type: none"> ▪ Last milestone in time but future milestones maybe not OR ▪ Last milestone delayed but future milestones expected to be in time | <ul style="list-style-type: none"> ▪ Neither the current nor future milestones are expected to be in time OR ▪ No statement possible |
| Quality | <ul style="list-style-type: none"> ▪ Internal and external validation successful OR ▪ Found errors removed | <ul style="list-style-type: none"> ▪ Various minor errors and exactly one major error detected OR ▪ Found errors removed | <ul style="list-style-type: none"> ▪ Multiple major errors found OR ▪ No testing was done OR ▪ No test plan exists |
| Customer Expectation | <ul style="list-style-type: none"> ▪ Customers feels answered AND ▪ Project lead can direct customer expectation AND ▪ Customer is satisfied | <ul style="list-style-type: none"> ▪ Final product will not satisfy all customer expectations OR ▪ Project lead cannot direct all customer expectations OR ▪ Customer already mentioned discontent with final product | <ul style="list-style-type: none"> ▪ Customer feels not included OR ▪ Project lead cannot direct customer expectation OR ▪ Customer already escalated project parts OR ▪ Escalation is foreseeable |

BAC (Budget at Completion) – Available project budget

EAC (Estimate at Completion) – Weekly estimation of necessary budget

| Situation | Category | RAG Status |
|--|----------|------------|
| The overall cost at the end of the project will be EUR 20.000 lower than planned. | | |
| A date has to be postponed by 2 weeks. All future milestones have to be shifted 2 weeks further. | | |
| The total cost was increased by 30% due to unsuccessful tests. | | |
| The customer's supervisor is very cooperative but expresses constant concern at all project manager's proposals. | | |

Exercise 3

Discuss why always two planning levels are necessary for complex projects: general and detailed planning. Why does not one of them suffice? Why is detailed planning a constant and recurring process and just considers a short period?

Project Planning

Project planning means to

- theoretically think through the whole project and identify missed tasks and work packages.
- determine task and work package dependencies and define work order

Planning typically involves two phases:

- general planning: subtasks, work packages, milestones
- detailed planning: prioritization of tasks, subtasks and work packages and derivation of work order

Completely planning a project thoroughly from beginning to end is typically not possible - aim to plan as precise as possible for near-term goals and be less specific for goals in the far future.

The RAG Status

To assess the project's current status, different aspects have to be taken into account: cost, time, quality, and customer expectations. A typical visualization is the RAG Status matching a traffic light:

- Red - critical issues: the project requires corrective actions that cannot be resolved by the project team and manager
- Amber - issues exist: project performance is affected, but the project team or manager can deal with them
- Green - no issues: the project is performing as expected and according to plan

The RAG status is a quick and simple overview for executives to identify projects needing further assistance.

Project Status Report

A weekly report containing the current project situation aiming to inform company executives about the project's current situation.

The status report contains:

- the project's RAG status
- a description of the current situation
- changes to risks
- next steps
- perspective on e.g. next milestones

Project: LiveKorr

PALUNO
The Ruhr Institute for Software Technology

| | | | | | | | |
|--|-------|------|-------|---|-------|----------------------|-------|
| Time | ○ ● ○ | Cost | ○ ● ○ | Quality | ○ ○ ● | Customer Expectation | ○ ○ ● |
| Situation | | | | Measures & Outlook | | | |
| <ul style="list-style-type: none">- Quality Assurance<ul style="list-style-type: none">- Tests yielded 72 tickets<ul style="list-style-type: none">- 58 minor issues- 12 test related- 2 major errors- UI issues<ul style="list-style-type: none">- Student interface lacks responsiveness- Buttons misplaced when rotating device on Android 4.4- Data Migration<ul style="list-style-type: none">- Work package planning has started- First conceptual draft available- Planned relocation of Dev-Team to new office will cause about 2 days delay- UI design for student overview delayed (P. Miller responsible but is involved in Project Sym4TK) | | | | <ul style="list-style-type: none">- Relocation will take place in week 12- Additional hardware will be delivered in week 11- refer to risk document (RD_150312) for risk avoidance strategy | | | |



Hugbúnaðarverkefni 1 / Software Project 1

5. Software Architecture

HBV501G – Fall 2015

Matthias Book



HÁSKÓLI ÍSLANDS
VERKFRÆÐI- OG NÁTTÚRVÍSINDASVIÐ
ÍÐNAÐARVERKFRÆÐI-, VÉLAVERKFRÆÐI-
OG TÖLVUNARFRÆÐIDEILD

What Makes a Great Software Engineer?

ACM Webcast, Fri 9 Oct, 18:00-19:00 GMT



- Paul Li and Andrew Ko discuss **what software engineers think are the attributes of software engineering expertise**, based on interviews spanning numerous divisions at Microsoft, including several interviews with software architects having over 25 years of experience. They discuss attributes spanning engineers' personality attributes, decision-making abilities, interactions with teammates, and software designs.
- **Paul Li** is a Senior Data Scientist at Microsoft's Windows and Devices Group, with a decade of research and practical experience at Avaya, ABB, IBM, and Microsoft.
- **Andrew Ko** is Associate Professor at the University of Washington Information School, with a research focus on human-computer interaction, computing education, and software engineering.

Registration: <http://event.on24.com/wcc/r/1054141/1119C0B0805E617FE139FC7DB74621CE>



HÁSKÓLI ÍSLANDS

Matthias Book: Software Project 1

Takeaway from Assignment 1

- Two of the most important artefacts created in the Inception phase are:
 1. **The Vision and Scope document**
 - Describing what you want to build, what its key features/capabilities will be, who will use it...
 - Imagine having to convince your boss or an investor to provide funding for the project
 - **Helps you to make architecture/design decisions later that are founded in a thorough understanding of application domain, stakeholder goals, and other relevant factors**
 2. **The initial Use Case document**
 - Describing the most important use cases of your product
 - i.e. the primary things that your system is supposed to be able to do
 - **Accumulates your detailed knowledge about how users will work with the system to accomplish their business goals, and how those usage scenarios should play out**



Elaboration

see also:

Larman: Applying UML and Patterns, Ch. 8



HÁSKÓLI ÍSLANDS

Matthias Book: Software Project 1



Recap: Unified Process Phases

1. Inception

- Approximate vision
- Business case
- Scope
- Vague estimates

Not a “requirements phase”, but feasibility analysis after which we can decide whether it makes sense to proceed with project

2. Elaboration

- Refined vision
- Identification of most requirements and scope
- More realistic estimates
- Iterative implementation of core architecture
- Resolution of high risks

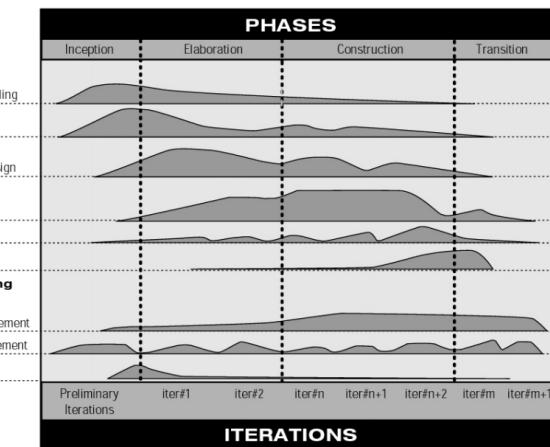
3. Construction

- Iterative implementation of lower-risk and other elements
- Preparation for deployment

4. Transition

- Beta tests
- Deployment

Not a “design phase”, but iterative development of core architecture and mitigation of high risks



Not an “implementation phase”, but incremental design and development of sets of features

Recap: Primary Objectives of Elaboration Phase

- Establish **baseline architecture** addressing the architecturally significant scenarios
 - which typically expose the top technical risks of the project
 - Address all architecturally significant **risks** of the project
 - Produce an **executable architecture** and possibly exploratory, throw-away prototypes to mitigate specific risks such as
 - design/requirements trade-offs
 - component reuse
 - product feasibility
 - acceptance of investors, customers, and end-users
 - Demonstrate that the baseline architecture will support the **requirements** of the system
 - at a reasonable cost and in a reasonable time
 - Establish the **supporting environment** for the project
- Ensure that...
- the architecture, requirements and plans are stable enough
 - the risks sufficiently mitigated
- ...to predictably determine the **cost and schedule** for completion of development



Elaboration in a Nutshell

- Goals:
 - Build the core architecture
 - Resolve the high-risk elements
 - Define most requirements
 - Estimate the overall schedule and resources
- Are the stakeholders confident that they have...
 - sufficient **understanding** of the domain
 - a realistic **solution** approach to the technical challenges
 - and the necessary **resources**
 - ...in order to begin investing major effort into the project
 - and get it done successfully?



Elaboration in a Nutshell

- It isn't Elaboration if...
 - it takes more than very few months
 - most requirements have already been defined
 - the architecture has already been defined
 - it produces only throw-away prototypes
 - it does not produce an executable architecture
 - there is an attempt to do a complete and careful design
 - there is no early and realistic testing
 - risks are not being addressed
- There may be considerable domain modeling that helps the team to understand the application domain.
- However, apart from the system architecture, detailed technical modeling of individual components won't occur until the iterations in the Construction phase.



Software Quality Attributes

see also:

Wiegers, Beatty: Software Requirements, Ch. 14

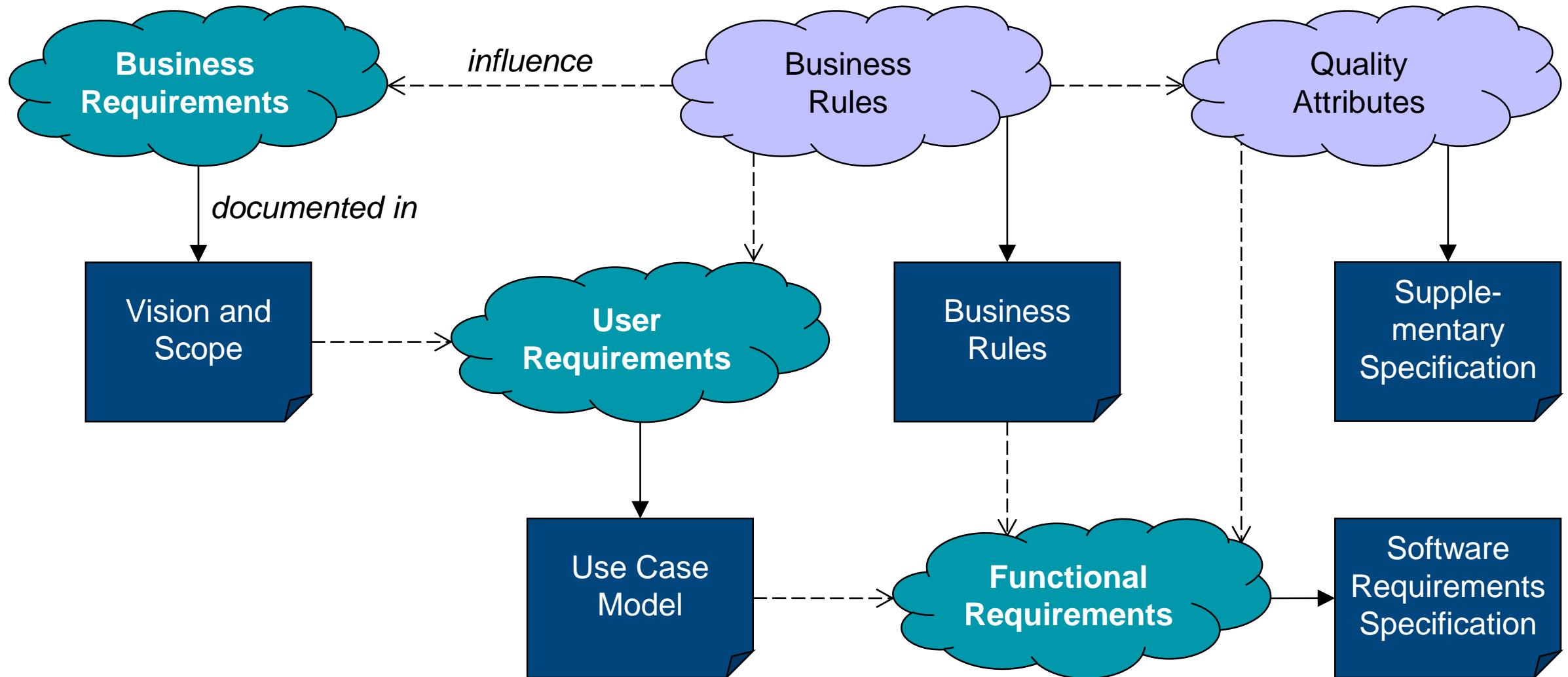


HÁSKÓLI ÍSLANDS

Matthias Book: Software Project 1



Recap: Requirements Levels



Software Quality Attributes

External Quality

- Availability
- Installability
- Integrity
- Interoperability
- Performance
- Reliability
- Robustness
- Safety
- Security
- Usability

Discernible through execution of software

Internal Quality

- Efficiency
- Modifiability
- Portability
- Reusability
- Scalability
- Verifiability

Discernible through working with code

-
- Typically, only a subset of these attributes is of particular relevance for a particular project.



External Quality Attributes

- **Availability**
 - The extent to which the system's services are available when and where they are needed
- **Installability**
 - How easy it is to correctly install, uninstall, and reinstall the application
- **Integrity**
 - The extent to which the system protects against data inaccuracy and loss
- **Interoperability**
 - How easily the system can interconnect and exchange data with other systems/components
- **Performance**
 - How quickly and predictably the system responds to user inputs or other events
- **Reliability**
 - How long the system runs before experiencing a failure



External Quality Attributes

- **Robustness**
 - How well the system responds to unexpected operating conditions
- **Safety**
 - How well the system protects against injury or damage
- **Security**
 - How well the system protects against unauthorized access to the application and its data
- **Usability**
 - How easy it is for people to learn, remember, and use the system



Internal Quality Attributes

- **Efficiency**
 - How efficiently the system uses computing resources
- **Modifiability**
 - How easy it is to maintain, change, enhance, and restructure the system
- **Portability**
 - How easily the system can be made to work in other operating environments
- **Reusability**
 - To what extent components can be used in other systems
- **Scalability**
 - How easily the system can grow to handle more users, transactions, servers, etc.
- **Verifiability**
 - How readily developers and testers can confirm that the software was implemented correctly



Detailed Dimensions of Quality Attributes

- Many of these attributes have several more detailed dimensions, e.g.:
- **Performance:**
 - Response time, throughput, data capacity, dynamic capacity, predictability, latency, behavior in degraded mode or overload conditions
- **Usability:**
 - Suitability for the task, suitability for learning, suitability for individualization, conformity with user expectations, self-descriptiveness, controllability, error tolerance
- **Modifiability:**
 - Maintainability, understandability, flexibility, extensibility, augmentability, supportability
- Specify precisely which ones are relevant in what way

Defining Relevant Quality Requirements

- A typical system can not (and does not need to) exhibit maximum possible quality across all attributes
 - Because of unavoidable design trade-offs
 - Because the effort would be prohibitively expensive
 - Need to focus on a set of relevant quality attributes, and reasonable requirements for them
1. Start with a broad list of quality attributes
 2. Reduce the list through discussion with stakeholders
 3. Prioritize remaining quality attributes
 4. Elicit specific expectations for each attribute from stakeholders
 5. Derive well-structured, measurable quality requirements from expectations



Caution

- Quality requirements often sound “fuzzy”
 - Formulate requirements precisely, and make them measurable
- It’s easy to write quality requirements and quantify them casually, but it can be very hard to actually implement a given quality requirement
 - Exaggerated quality demands can easily become major cost drivers
 - Make sure your quality requirements are necessary and realistic!
- Even quantified quality requirements are easily disregarded because there often is no immediately obvious place in the code to implement them (in contrast to functional requirements)
 - Often, you can’t precisely answer the question “where to implement this requirement?”
 - Be aware that many quality requirements have structural / architectural implications or need to be considered / implemented in many dispersed spots throughout the system
 - Examples: High availability, internationalization, security, controllability

Impact of Quality Requirements

- Quality requirements affect decisions / implementation in many different ways
 - They may also materialize in several aspects simultaneously (e.g. architectural + functional)
- **Functional requirements** are often influenced by
 - Installability, integrity, interoperability, reliability, robustness, safety, security, usability, verifiability
- **System architecture** is often influenced by
 - Availability, efficiency, modifiability, performance, reliability, scalability
- **Design constraints** are often influenced by
 - Interoperability, security, usability
- **Design guidelines** are often influenced by
 - Efficiency, modifiability, portability, reliability, reusability, scalability, verifiability, usability



Constraints

- Besides quality requirements, **restrictions on design, architecture or implementation choices** may be imposed by numerous external factors, e.g.
 - Prescribed technologies, tools, languages or databases
 - Expected operating environments or platforms
 - Required development conventions or standards
 - Backward compatibility with earlier products; potential forward compatibility
 - Limitations or compliance requirements with legal regulations or business rules
 - Hardware limitations such as memory/processor restrictions, power consumption, etc.
 - Available input modes on different devices
 - Existing interface conventions expected by users
 - Interfaces to existing systems, e.g. data formats and communications protocols
- Caution: Some stakeholder constraints are actually masked ideas for solutions
 - Check whether the constraint actually exists, or whether a different solution would also work

Software Architecture

see also:

Larman: Applying UML and Patterns, Ch. 13 & 33



HÁSKÓLI ÍSLANDS

Matthias Book: Software Project 1



Definition: Software Architecture

A software architecture is

- the set of significant **decisions about the organization** of a software system,
- the **selection of the structural elements** of which the system is composed,
 - and their **interfaces**,

together with

- their behavior, as specified in the **collaborations** among those elements,
- the **composition** of these structural and behavioral elements
 - into progressively larger subsystems,

and

- the architectural **style** that guides this organization
 - of the elements and their interfaces, their collaboration, and their composition.



Architectural Analysis

- How do we come up with a software architecture?
- **The essence of architectural analysis** is to
 - identify factors that should influence the architecture,
 - understand their variability and priority, and
 - resolve them by making architectural decisions.
- Challenge: It's difficult to...
 - Know what questions to ask
 - Weigh the trade-offs
 - Know the many ways to resolve an architecturally significant factor
 - Decide on the best way under the given circumstances



Examples of Issues to be Resolved at Architectural Level

- How do reliability and fault-tolerance requirements affects the design?
 - For what remote services will fail-over to local services be allowed? Why?
 - Do the local and remote services work exactly the same? What are the differences?
- How do the licensing costs of purchased subcomponents affect profitability?
 - Should we integrate a high-quality third-party persistence framework that will charge a fee on all transactions, go with a low-cost open-source alternative, or develop or own?
- How do the adaptability and configurability requirements affect the design?
 - What variations of business rules need to be reflected in the implementation?
 - What degree of evolution should be accommodated? How should variations be specified?
- How do availability and dependability requirements influence the effort?
 - Very strict availability requirements may induce huge costs for redundant hardware, hot-swap capability, failover mechanisms, backups, etc. Does this effort correlate with the risk?



Common Steps in Architectural Analysis

- Goal: Understand influence of architectural drivers, their priorities and variability

1. Identify and analyze **architectural drivers***, i.e. requirements that have an architectural impact

- Especially non-functional (quality) requirements
- But also functional requirements, esp. regarding expected variability or change
- Can be found e.g. in
 - Vision and Scope document
 - Business Rules document
 - Supplementary Specification document
 - Use Case document (sections on special requirements, technology variations, open issues in fully-dressed format)

2. Make **architectural decisions**

- Analyze alternatives
- Create solutions that address the impact:
 - Build a custom solution
 - Buy a third-party solution
 - Remove the requirement
 - Hire an expert
 - ...

3. Document decisions

- So people will not inadvertently undermine design decisions later
- or spend time pursuing rejected options



Describing Architectural Drivers

Architectural Drivers Table

- Part of the Supplementary Specification document in the UP:
 - Architectural driver
 - Quality scenario and measures
 - Variability
 - Current flexibility
 - Future evolution
 - Impact of driver (and its variability) on stakeholders, architecture, other drivers
 - Priority for success (high/medium/low)
 - Difficulty or risk (high/medium/low)

Quality Scenarios

- Short statements of the form “When [stimulus], [measurable response]”:
 - “When the completed sale is sent to the remote tax calculator, the result is returned within 2 seconds most of the time, measured in a production environment under average load.”
 - Still need to clarify what “most of the time” and “average load” mean exactly.
 - “When a bug report arrives, reply with a phone call within one working day.”
 - Note: Some constraints cannot be enforced, only supported.



Example: Architectural Drivers Table

| Driver | Quality Scenario | Variability | Impact | Priority | Difficulty |
|--------------------------------------|---|---|---|----------|------------|
| [...] | | | | | |
| Reliability – Recoverability | | | | | |
| Recovery from remote service failure | When a remote service fails, reestablish connectivity with it within one minute of its detected re-availability, under normal store load in a production environment. | <i>Current flexibility:</i> Expert says local client-side simplified services are acceptable (and desirable) until reconnection is possible. <i>Future evolution:</i> Within two years, some retailers are likely willing to pay for full local replication of remote service. | High impact on the large-scale design. Retailers dislike failure of remote services as it prevents them from using the POS system to make sales. | High | Medium |
| [...] | | | | | |



Example: Architectural Drivers Table

| Driver | Quality Scenario | Variability | Impact | Priority | Difficulty |
|-----------------------------------|---|---|---|----------|------------|
| [...] | | | | | |
| Constraint – Legal | | | | | |
| Current tax rules must be applied | When auditor evaluates system, 100% conformance will be found. When tax rules change, they will be operational within the period allowed by the government | <i>Current flexibility:</i> Conformance is mandatory. <i>Future evolution:</i> Tax rules can change almost weekly because of many rules and levels of government taxation (city, state...). | Failure to comply is a criminal offense. Difficult to write our own logic (and keep it up to date), but easy to buy a third-party component with maintenance contract. | High | Low |
| [...] | | | | | |



Pick Your Battles

- Quality scenarios are easy to write, but hard to implement and test!
- No point in listing many tall goals if they will not be implemented or tested
- Or worse, if they are implemented with high effort despite being unnecessary

➤ **Pick your battles:** What are the really critical make-or-break quality scenarios?

- Focus on defining, implementing and testing these
- Rather than distracting developers with a lot of requirements that are not followed through on or waste effort that could be used on other features
- Example pitfall: Avoid over-engineering – do not spend too much effort on “future-proofing” for evolution paths whose probability you cannot yet foresee.



Making Architectural Decisions

- Collecting and describing architectural drivers is comparatively easy.
- Addressing/resolving them in light of interdependencies, tradeoffs, interdependencies is much more difficult.
 - Highly dependent on application domain and technology
 - Business goals and stakeholder interests become central to technical decisions
 - Requires consideration of more large-scale/global goals and trade-offs than local-scale UI or OO design decisions
 - Requires knowledge and critical consideration of many areas:
 - Architectural styles and patterns, technologies, products, pitfalls, domain knowledge, trends...
- Hierarchy of goals to guide priority of architectural decisions:
 1. Inflexible constraints, e.g. safety and legal compliance – unavoidable
 2. Business goals, e.g. particular features, deadlines etc. – some flexibility
 3. All other goals (are often derived from business goals) – some leeway for interpretation

Documenting Architectural Decisions

- Architectural decisions should be documented in Technical Memos
 - part of the Unified Process' Software Architecture document

Technical Memo Template:

1. **Issue:** the internal or external quality attribute and requirement
2. **Solution summary:** one-sentence summary of architectural decision
3. **Factors:** the quality issue to be addressed
4. **Solution:** all structural/logical aspects required to implement the solution
5. **Motivation:** reasons for choosing the described solution
6. **Unresolved issues:** issues still to be clarified or solved in other ways
7. **Alternatives considered:** alternative solutions, reasons for their rejection



Example: Technical Memo

- **Issue:** Constraint – Legal: Tax rule compliance
- **Solution summary:** Purchase a tax calculator component.
- **Factors:** Current tax rules must be applied, by law.
- **Solution:**
 - Purchase a tax calculator with a licensing agreement to receive ongoing tax rule updates. Note that different calculators may be used at different installations.
- **Motivation:**
 - Time-to-market, correctness, low maintenance requirements, and happy developers (see alternatives). These products are costly, which affects our cost-containment and product pricing business goals, but the alternative is considered unacceptable.
- **Unresolved issues:**
 - What are the leading products and their qualities?
- **Alternatives considered:**
 - Building our own tax component is estimated to take too long, be error-prone, and create an ongoing costly and uninteresting (to company's developers) maintenance responsibility.



Basic Architectural Design Principles

- Principles of object-oriented design can be applied to architectural scale as well:
 - Low coupling
 - High cohesion
 - Protected variation
 - Separation of concerns
 - Localization of impact
- ...just with higher level of granularity (applications, subsystems, processes)
- Large body of architectural patterns – some very common ones:
 - Layering
 - Façade
 - Observer
 - Model-View-Controller



Layered Architectures

- Organize large-scale logical structure of a system into discrete layers of distinct, related responsibilities
- Have a clean, cohesive separation of concerns such that
 - “lower” layers provide more general services
 - “higher” layers are more application-specific
- Collaboration and coupling is from higher to lower layers, i.e. higher layers depend on functionality of lower layers
 - Strict layered architecture: A layer calls only features of layer below
 - Relaxed layered architecture: A layer can call different lower layers

Extreme example –
many systems are not
split in so many layers

UI

(e.g. address dialog)

Application

(e.g. ordering workflow)

Domain

(e.g. product inventory)

Business infrastructure

(e.g. currency conversion)

Technical services

(e.g. persistence)

Foundation

(e.g. database)



Benefits of Layered Architectures

- Separation of concerns
 - between high-level (application-specific) features and low-level (more generic) services
 - between user interface and application logic
- Reduced coupling and dependencies
- Improved cohesion and encapsulation
- Increased potential for reuse and replacement
- Increased potential for distribution of functionality across processes or devices



Model-View Separation Principle

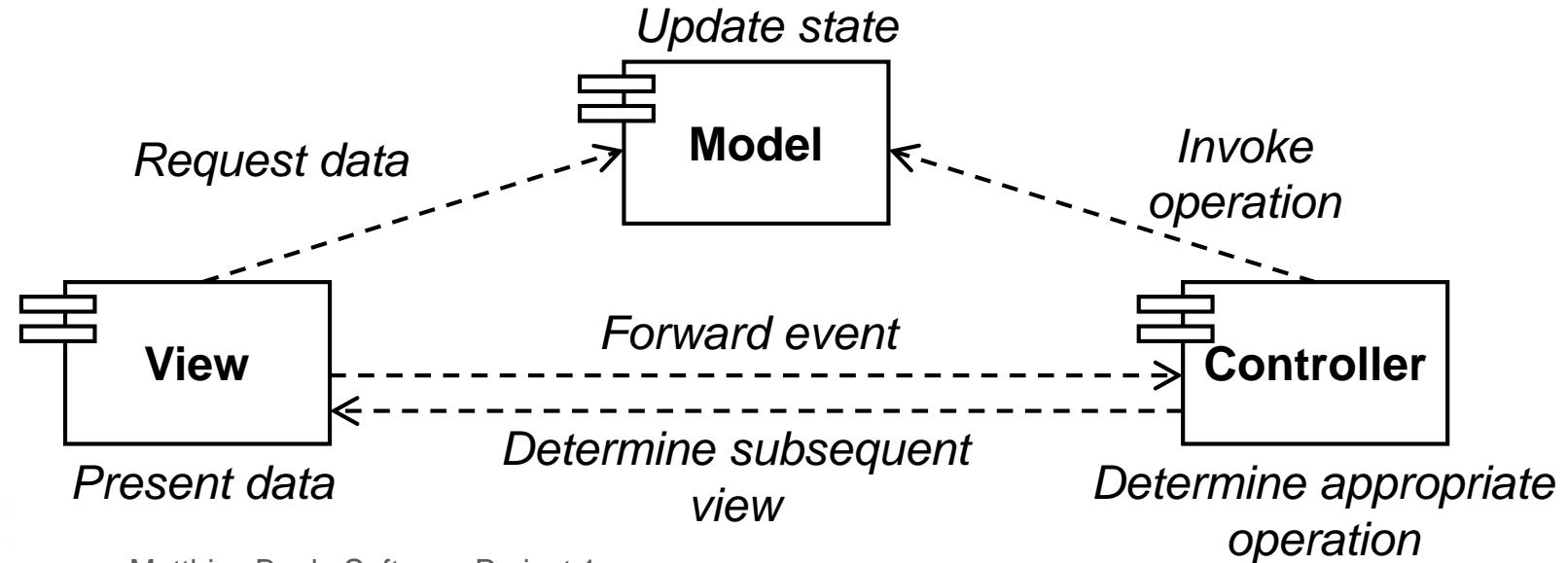
- The interface between user interface (UI) code and domain-specific code (whether implementing data structures or app logic) can easily become messy:
 - Both sides deal with very similar domain concepts, but represent them differently
 - Both sides need to be aware of business processes and rules, but to different degrees
 - Both sides could do the job of the other side, although not always as cleanly
 - Both sides could be decoupled from the other, but to which degree is it reasonable?
 - Both sides' dialog and control flows depend closely on each other
 - Without a clean separation, maintainability can become a nightmare
- Solution: Strict distinction of
 - **Model** components – comprising the business objects of the application domain
 - **View** components – comprising the user interface elements
 - **Controller** components – comprising the application logic that is in charge of the control flow

Model-View-Controller Pattern

- Strict distinction of
 - **Model** components – comprising the business objects of the application domain
 - **View** components – comprising the user interface elements
 - **Controller** components – comprising the application logic that is in charge of the control flow
- Implications and responsibilities:
 - A **business object** (e.g. Contract, Product, Customer, Sale...) will not call or rely on a user interface object (e.g. a window, web page, input field...).
 - It will only provide information to user interface objects when requested
 - A **user interface (UI)** object will not implement application logic (e.g. calculating tax).
 - It will only initialize UI objects with model data, receive UI events (e.g. mouse clicks), and delegate requests for functionality to appropriate controllers.
 - **Controller objects** receive messages from the UI, call the appropriate business functions, update the data model, and return control back to the user interface for the next interaction.

Model-View-Controller Pattern

- **Model:** Application state (data and operations working on them); usually subdivided again into the working data (in memory) and the persistence layer
- **View:** View(s) of a state; usually the user interface, but possibly other access services as well. *The view does not change the model!*
- **Controller:** Input interface; invokes particular operations on the model, based on events coming in from the view





Hugbúnaðarverkefni 1 / Software Project 1

6. Web Applications

HBV501G – Fall 2015

Matthias Book



HÁSKÓLI ÍSLANDS
VERKFRÆÐI- OG NÁTTÚRVÍSINDASVIÐ
ÍÐNAÐARVERKFRÆÐI-, VÉLAVERKFRÆÐI-
OG TÖLVUNARFRÆÐIDEILD

Miðmisseriskönnun

Evaluate this course on Uglá!
(30 Sep – 5 Oct)



HÁSKÓLI ÍSLANDS

Matthias Book: Software Project 1

Update: Assignments and Final Exam

Assignments (combined weight: 50%)

- 4 team assignments
 - Teams of 4: one presentation per member
 - Teams of 3: last pres. joint by all members
- All members receive the same grade
 - Can be above 10 for exceptional work
- Presenter receives bonus/malus
 - Added to or subtracted from team grade
- Grades above 10 included in averages
 - but capped at 10 for final course grade
- Need passing grade on averaged assignments to be admitted to final exam

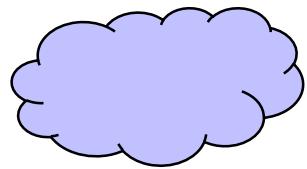
Final exam on 16 Dec (weight: 50%)

- Focus: Understanding of software engineering concepts and methods
 - maybe read/write small Java fragments
 - some UML modeling
- Scope: Lecture slides
 - Note: The soundtrack is relevant!
- Style: Written exam
 - Short paragraphs of whole sentences – justify your opinions!
- Allowed aids: Handwritten notes only
- Need passing grade on final exam and avg. assignments to pass course

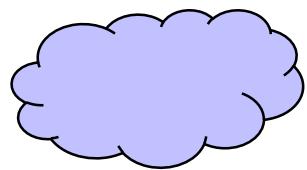


Software Architecture in Context

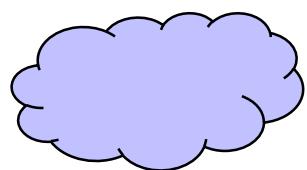
Business Domain



Business Requirements



Quality Attributes



Business Rules

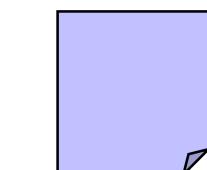
Requirements



Vision and Scope

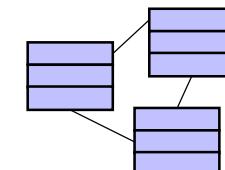


Supplementary Specification



Use Cases

Business Modeling

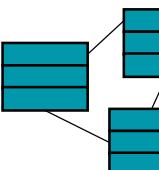


Domain Model

Architecture

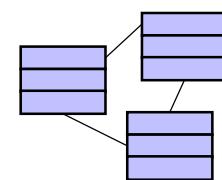


Software Architecture

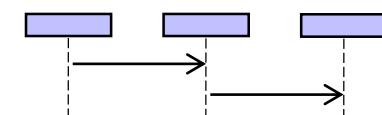


Package Diagrams

OO Design



Class Diagrams



Interaction Diagrams

Characteristics of Software Architecture

- Architectural concerns are especially **related to non-functional requirements**, but their resolution permeates the **implementation of the functional requirements**.
- Architectural concerns require **awareness of the business context, stakeholder goals**, as well as requirements' **variability** and **evolution**.
- Architectural concerns involve **system-level, large-scale issues** whose resolution usually involves **large-scale, fundamental decisions** that are extremely costly to change later on.
- Architectural decisions depend on the conception and **critical evaluation of alternative solutions**.
- Architectural decisions usually involve **interdependencies** and **tradeoffs**.

Iterative-Incremental Treatment of Software Architecture

■ Inception

- Candidate architectures are considered.
- If architectural feasibility of key requirements is questionable, a light proof-of-concept architecture may be implemented to determine feasibility.

■ Elaboration

- Architectural drivers (i.e. major architectural risks) are identified and resolved.
- An executable architecture that can serve as the backbone of further implementation is built.

■ Construction

- Business components are built and integrated with the core architecture.
- No major changes of the architecture should be necessary.

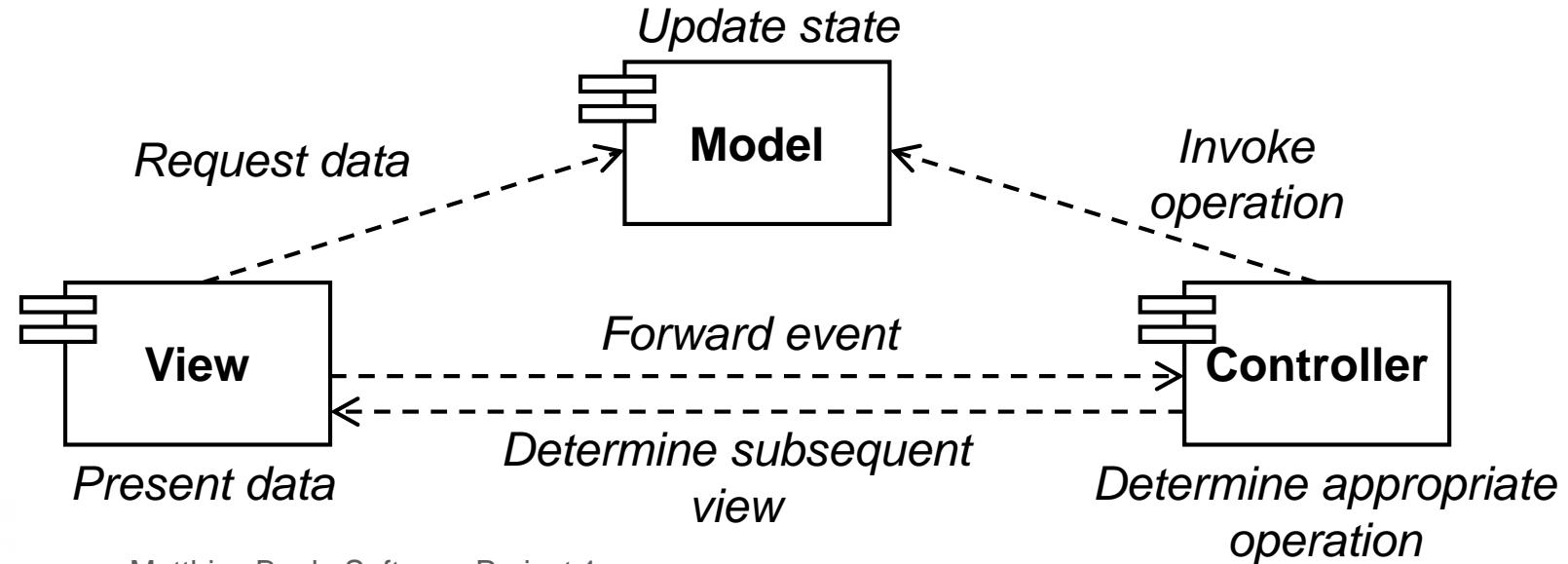
■ Transition

- The final architecture is documented as a basis for future maintenance and evolution.



Recap: Model-View-Controller (MVC) Pattern

- **Model:** Application state (data and operations working on them); usually subdivided again into the working data (in memory) and the persistence layer
- **View:** View(s) of a state; usually the user interface, but possibly other access services as well. The view does not change the model.
- **Controller:** Input interface; invokes particular operations on the model, based on events coming in from the view



Web Application Development with Spring Web MVC

see also:

- <http://docs.spring.io/spring-framework/docs/current/spring-framework-reference/html/mvc.html>, Sect. 21.3
- <http://spring.io/guides/gs/spring-boot/>
- <http://spring.io/guides/gs/serving-web-content/>



Web Applications

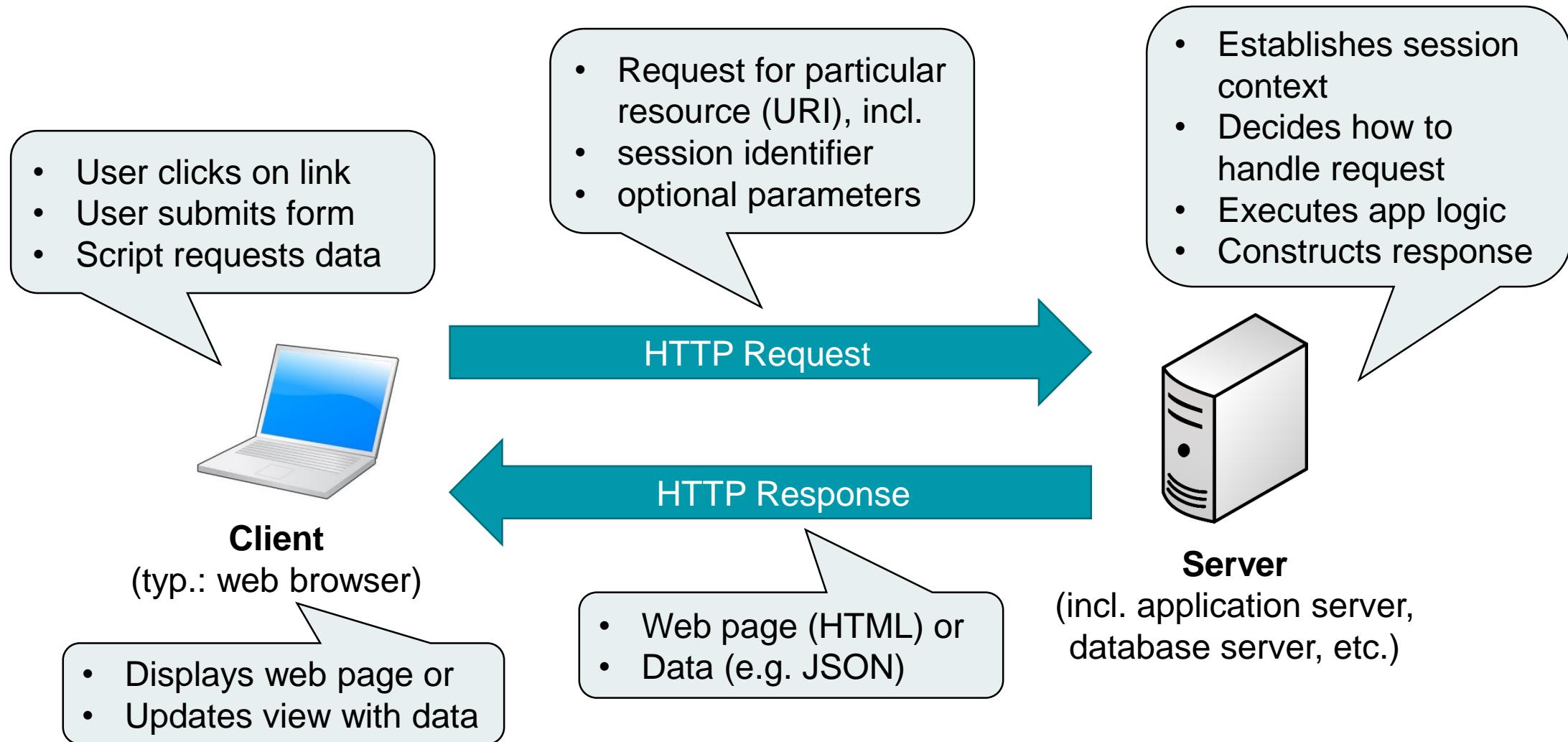
- In a web application, the architecture encompasses two communicating actors:
 - Client – usually, a web browser running on a desktop or mobile device
 - Server – a central host upon which the clients depend to provide data and/or computation

Common implementation options:

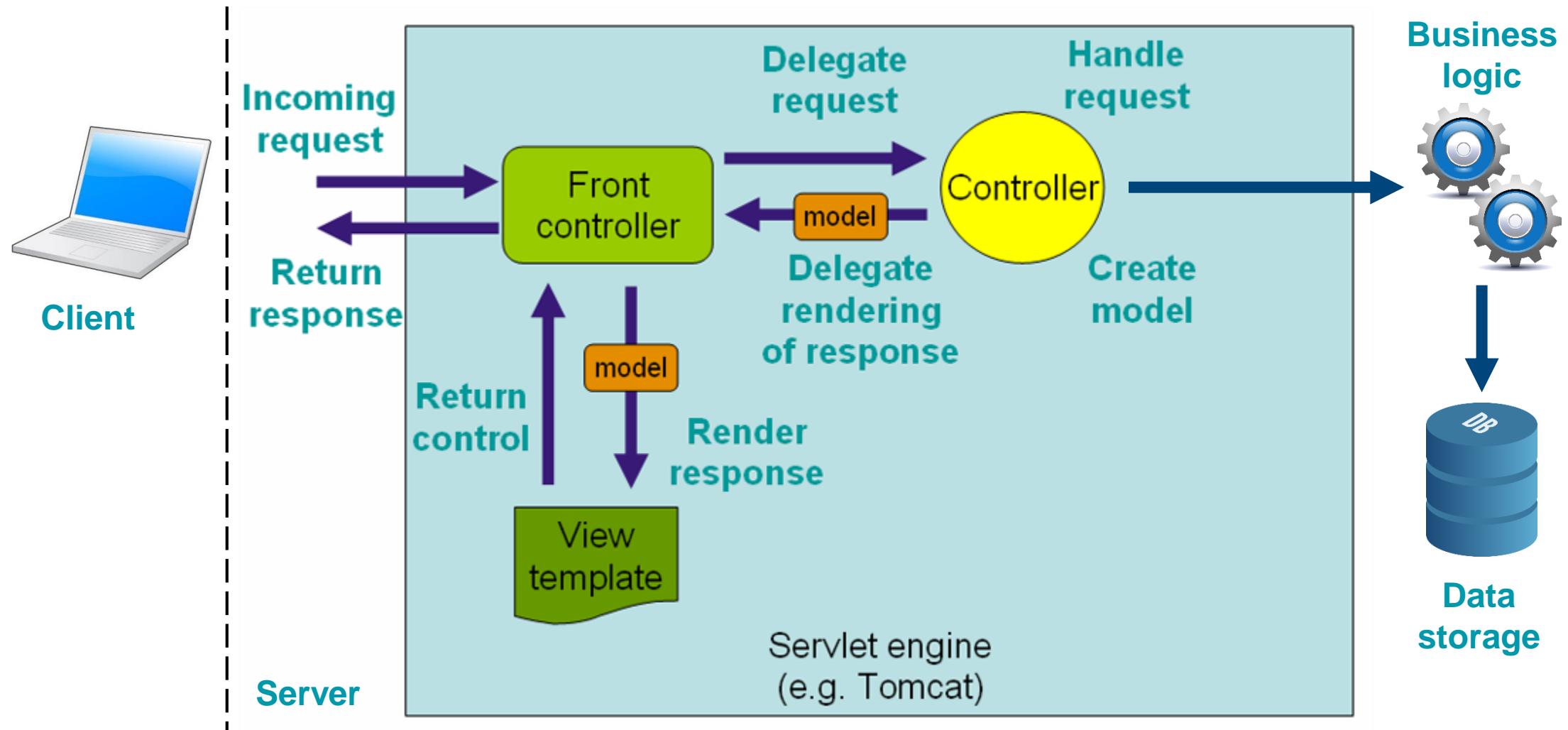
- **Client-side MVC**
 - Application's views are created on the client, dialog flow is controlled by the client
 - Server provides data and services (e.g. through RESTful services)
- **Server-side MVC**
 - Application's views are created on the server, dialog flow is controlled by the server
 - Client just displays the views and submits user inputs to server



HTTP Request/Response Cycle

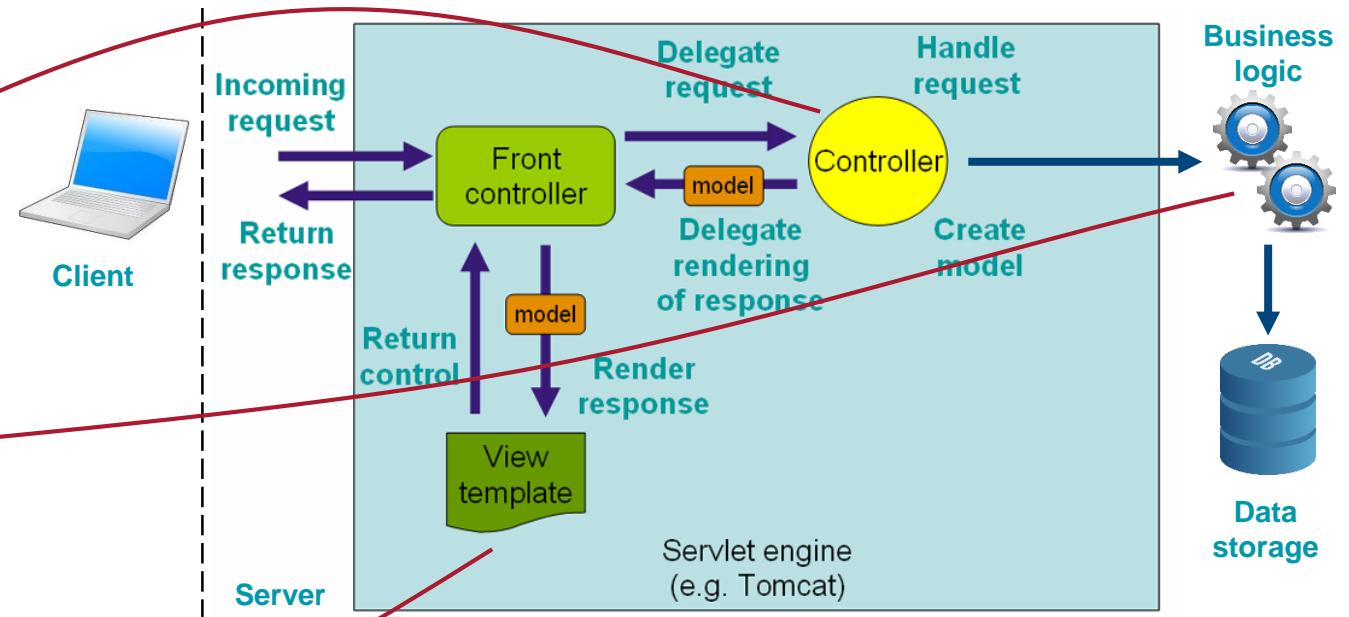
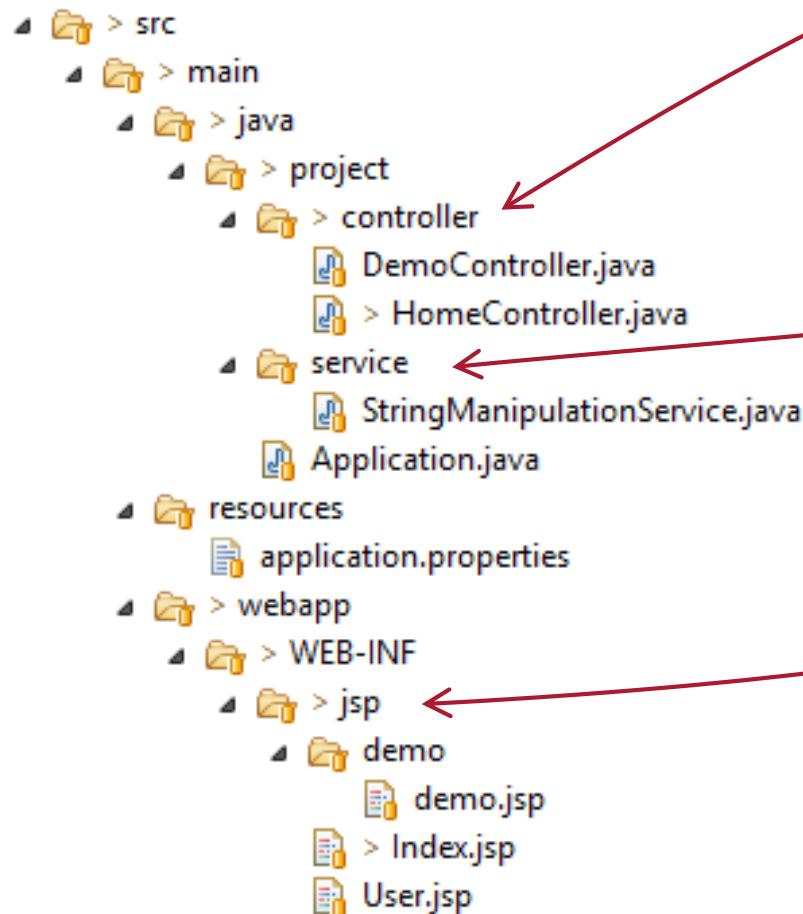


Spring Web MVC Framework



Spring Web MVC Framework

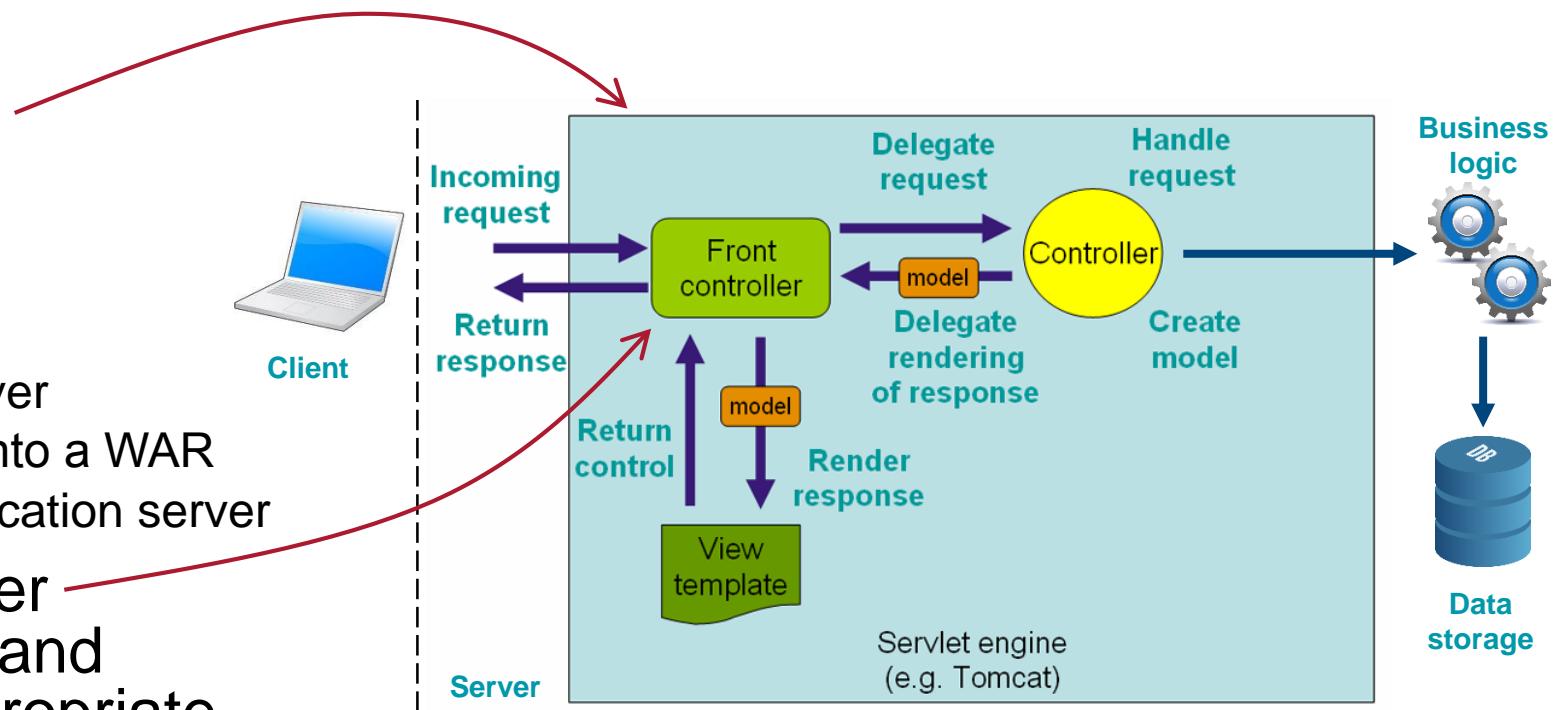
- Typical project structure:



- See “Development Environment” slides in Verkefni folder on Uglar for setup details

Setting up the Servlet Engine and Front Controller

- We need a servlet engine that will execute our web application
 - Usually, we'd have to
 - install an application server
 - package our app's files into a WAR
 - deploy WAR on the application server
- We need a Front Controller that will receive requests and distribute them to the appropriate business controller
 - Usually, we'd have to instantiate and configure a Spring DispatcherServlet
 - or write one ourselves from scratch
- Spring Web MVC and Spring Boot simplify these steps
 - Java annotations indicate our intentions without having to implement them



The Spring Boot Application Class

```
package project;  
// [import statements]  
  
@SpringBootApplication  
public class Application {  
    public static void main(String[] args) {  
        SpringApplication.run(Application.class, args);  
    }  
}
```

Implies various configuration and bootstrapping activities performed by the Spring Boot framework

Run the `main` method just like you would run any other Java program, and Spring will

- start an embedded application server
- deploy the web application

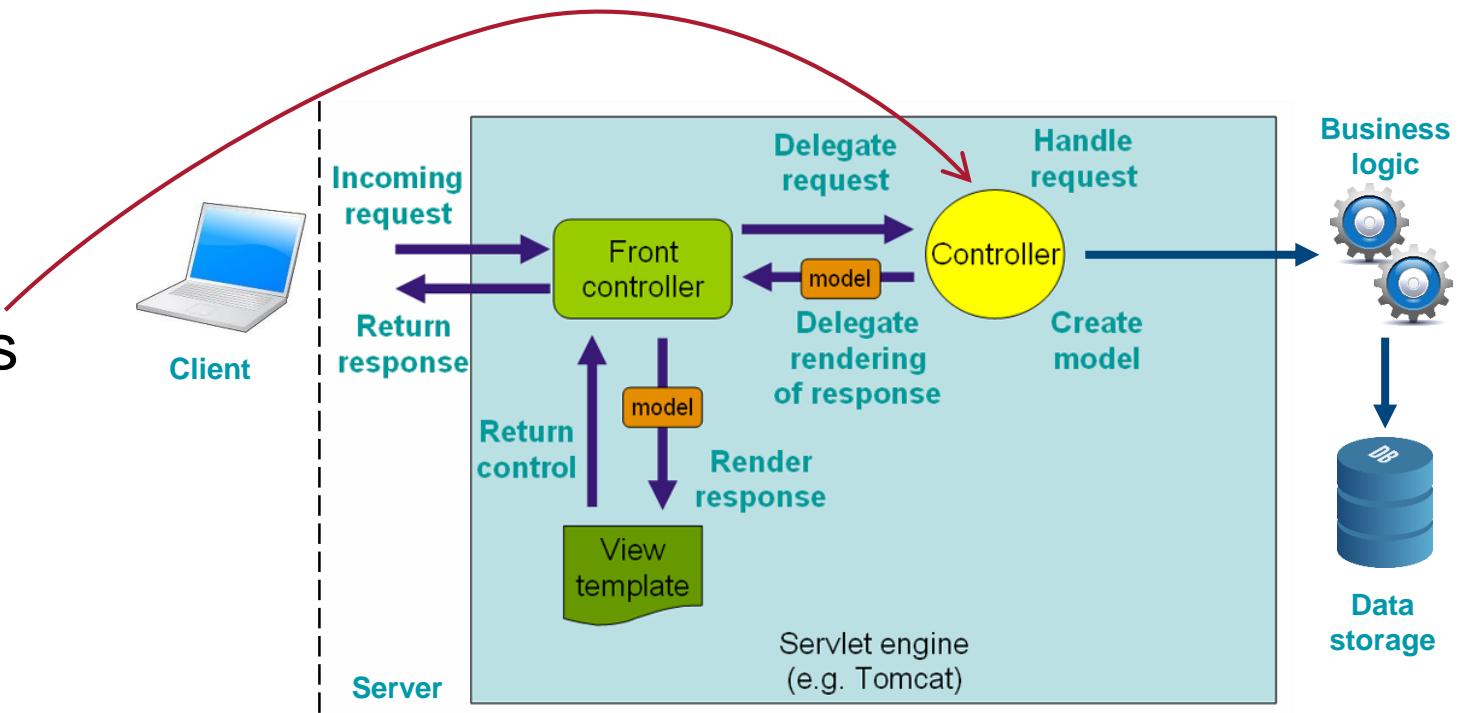
Application will then be accessible at `http://localhost:8080` through your browser



Handling Requests

- Clients can trigger different behaviors by requesting different URIs
- We need business controllers that will respond to specific requests

- Usually, we'd have to
 - Write the controller class
 - Configure the mapping from a particular request URI to a particular controller



- Spring Web MVC simplifies this
 - By providing the `@Controller` and `@RequestMapping` annotations
 - Letting us focus on implementing what the controller does, not how it is wired into the architecture

A Trivial Controller

```
package project.controller;  
// [import statements]  
  
@Controller  
public class HomeController {  
  
    @RequestMapping(value = "/", method = RequestMethod.GET)  
    public String home() {  
        // [invocations of application logic]  
        return "Index";  
    }  
}
```

Tells Spring framework to treat this class as an MVC controller

Path that this controller method should be mapped to

- here: <http://localhost:8080/> will invoke this method

HTTP request type that this controller shall react to

- Links are GET requests
- Form submissions can be GET or POST requests

Path to view template from which response shall be constructed

- here: Index.jsp

Application-specific logic typically invoked here:

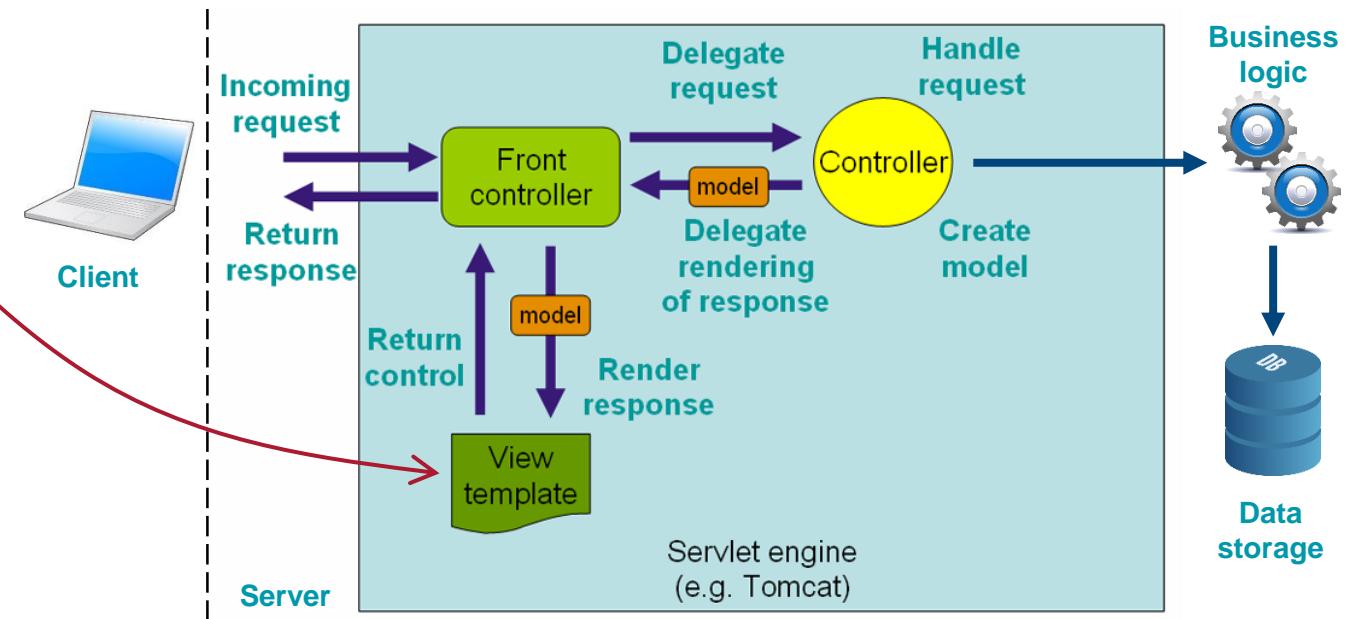
- Process request parameter
- Prepare model data to use in view



Constructing Responses

- We need view templates that define what our responses should look like

- Usually, we'd have to
 - Write a JavaServer Page (JSP) for each response page
 - Forward the HTTP Request to it



- Spring Web MVC simplifies this

- We still need to write the JSP
 - with support from various templating engines
- In the controller, we can simply name the JSP that should be displayed, without implementing the mechanics of request forwarding

A Static View

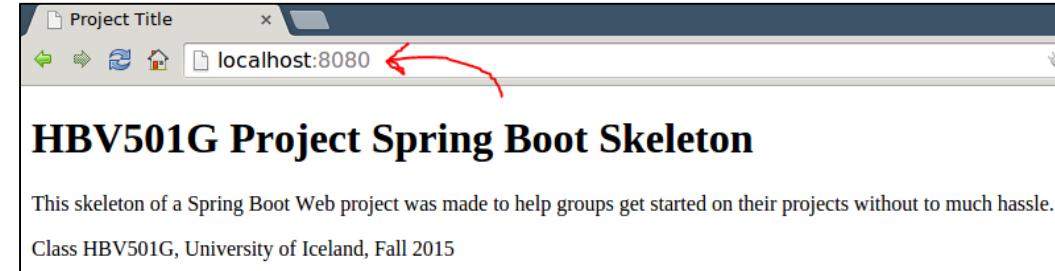
```
<!DOCTYPE html>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>

<html lang="en">
  <head>
    <title>Project Title</title>
  </head>
  <body>
    <h1>HBV501G Project Spring Boot Skeleton</h1>
    <p>This skeleton of a Spring Boot Web project was made to help groups...</p>
  </body>
</html>
```

Configure this location in
resources/application.properties

Stored on server in
main/webapp/WEB-INF/jsp/Index.jsp

HTML code of web
page to be delivered
back to client



Providing Data to the Server

- An HTTP request contains
 - The **address** of the desired resource (URI)
 - e.g. `http://localhost:8080/hello`
 - Used to
 - Identify the resource (file) to be returned to the client (in regular web servers)
 - Identify the endpoint (controller) to handle the request (in application servers)
 - **Optional parameters**
 - Either appended to the URI (in a GET request)
 - e.g. `http://localhost:8080/hello?id=1&message>Hello,+World!`
 - Or submitted in the HTTP request body (in a POST request)
 - Must be extracted and processed by the controller
- Different ways of creating these requests
 - **GET requests:** clicks on links (most common), form submissions, scripts
 - **POST requests:** form submissions (most common), scripts

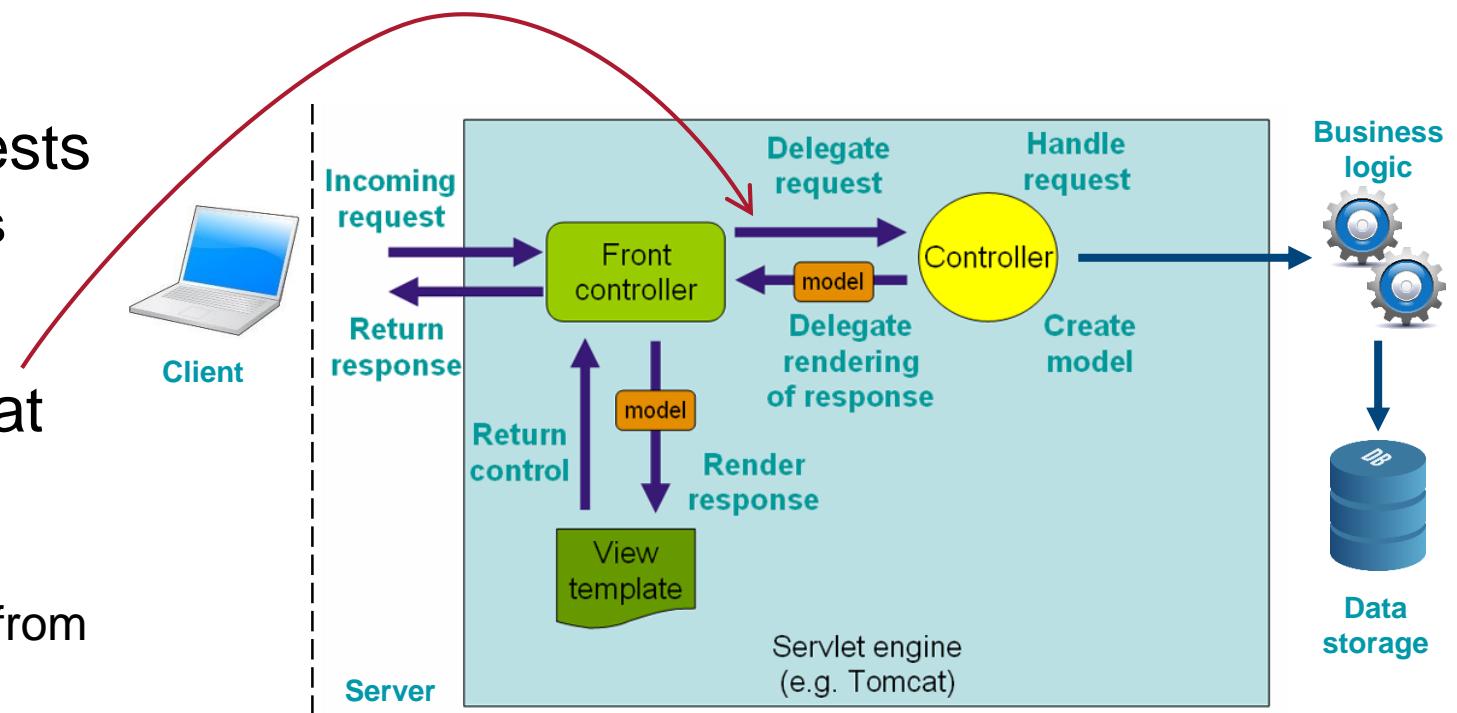
Form

Id:

Message:

Receiving Data from the Client

- Clients can add data to requests
 - through web forms or web links
- We want to work with data that client added to the request
 - Usually, we'd have to
 - extract individual parameters from URI or HTTP request body
- Spring Web MVC simplifies this
 - In the request handler, we can simply name the request parameters that should be extracted and provided as method parameters
 - Or we can interpret certain segments of the URI as path parameters



Request Parameters

```
package project.controller;  
// [import statements]  
@Controller  
public class HomeController {  
    @RequestMapping(value="/hello")  
    public String hello(  
        @RequestParam(value="name", required=false, defaultValue="User") String nafn,  
        Model model) {  
        // [application logic]  
        model.addAttribute("nom", nafn);  
        return "Greeting";  
    }  
}
```

Requesting
`http://localhost:8080/hello`
will invoke this method

Name of request parameter whose content shall be assigned to method parameter

- here: expected URL format:
`http://localhost:8080/hello?name=Jon`
- contents of name will be assigned to nafn

Value to assign if parameter is not specified in URL

Make data available in data model

- here: contents of local variable nafn will be available in data model under the label nom

For illustration only – typically, you would choose the same label for name, nafn and nom



Path Parameters

// [Controller declaration]

```
@RequestMapping(path="/owners/{ownerId}/pets/{petId}", method=RequestMethod.GET)
public String findPet(
    @PathVariable String ownerId, @PathVariable String petId, Model model) {
    Owner owner = ownerService.findOwner(ownerId);
    Pet pet = owner.getPet(petId);
    model.addAttribute("pet", pet);
    return "DisplayPet";
}
```

Template for extraction of path parameters from mapped URI

Method parameters with matching names

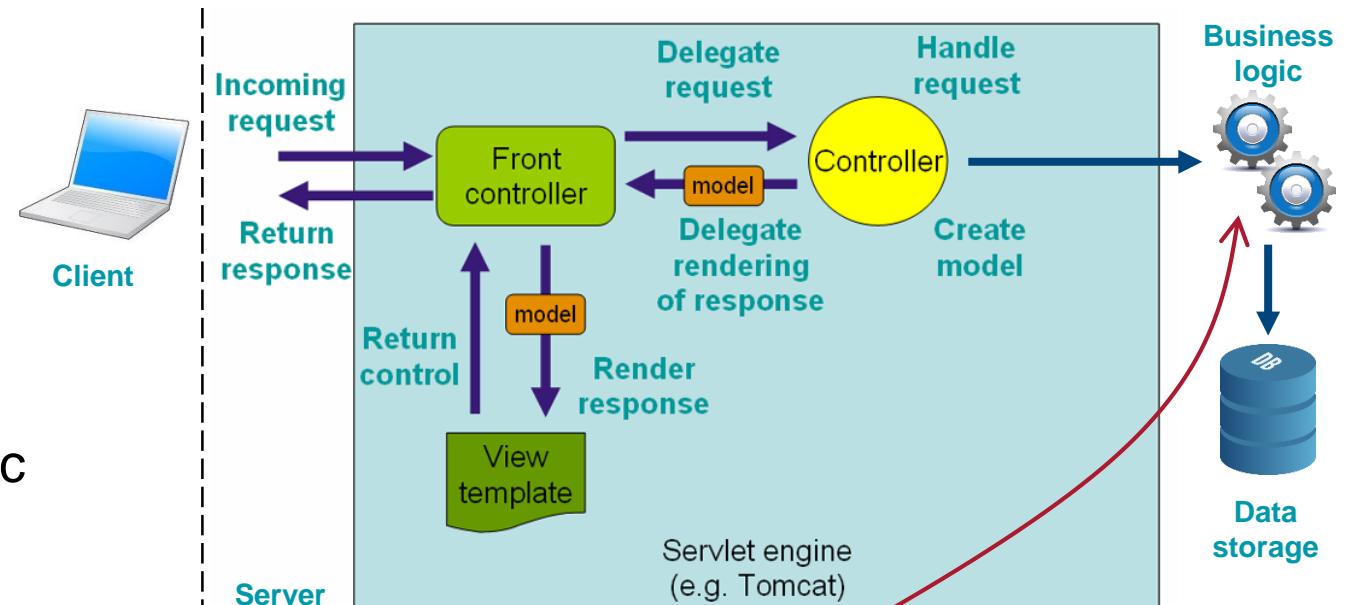
- Store the pet in the model
- Delegate to view that will display it

Invoking business logic functions



Invoking Business Logic

- Usually, the controller should not perform the actual business logic by itself, but deal only with
 - Collecting the necessary inputs from the request, session etc.
 - Calling the intended application logic
 - Preparing the model data required by the view
 - Determining the next view to be displayed

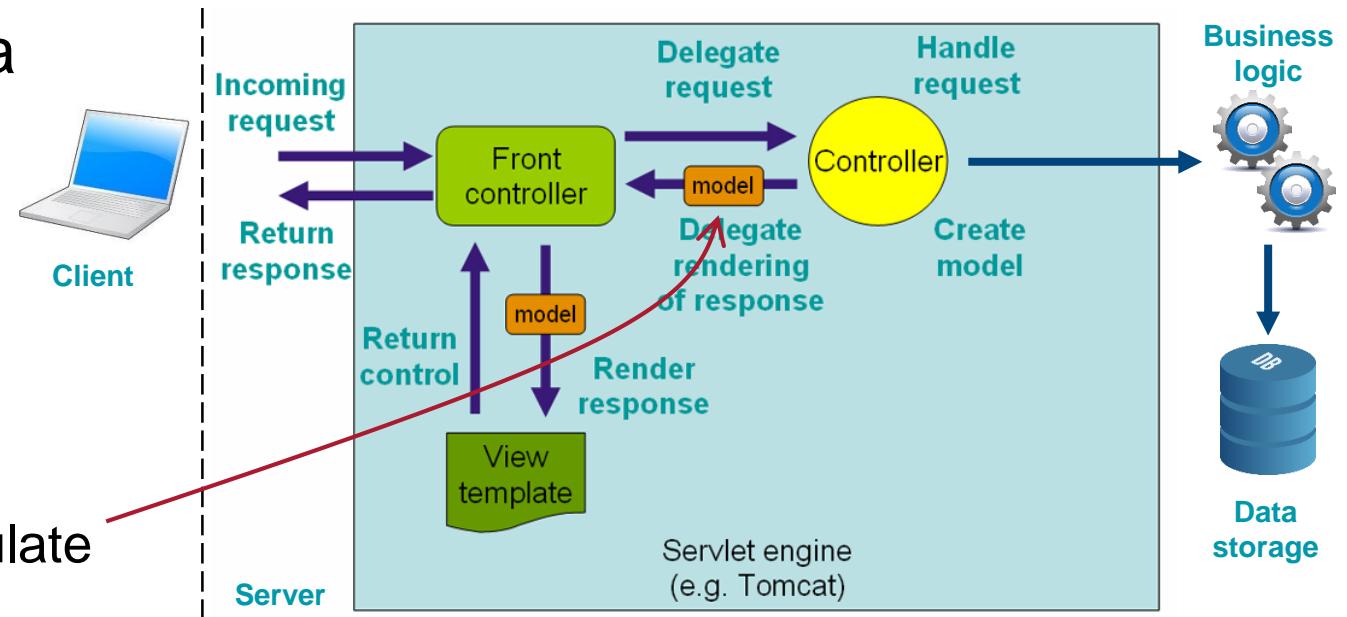


Incorporating Model Data into the View

- We want to include business data into the constructed responses

- Usually, we'd have to extract data from various sources in the view template

- Spring Web MVC simplifies this
 - In the request handler, we can populate a Model with all information that is required by the view to be constructed



Displaying Model Data

Stored on server in
main/webapp/WEB-INF/jsp/Greeting.jsp

```
<!DOCTYPE html>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>

<html lang="en">
  <head>
    <title>Greetings!</title>
  </head>
  <body>
    <p>Hello ${nom}!</p>
  </body>
</html>
```

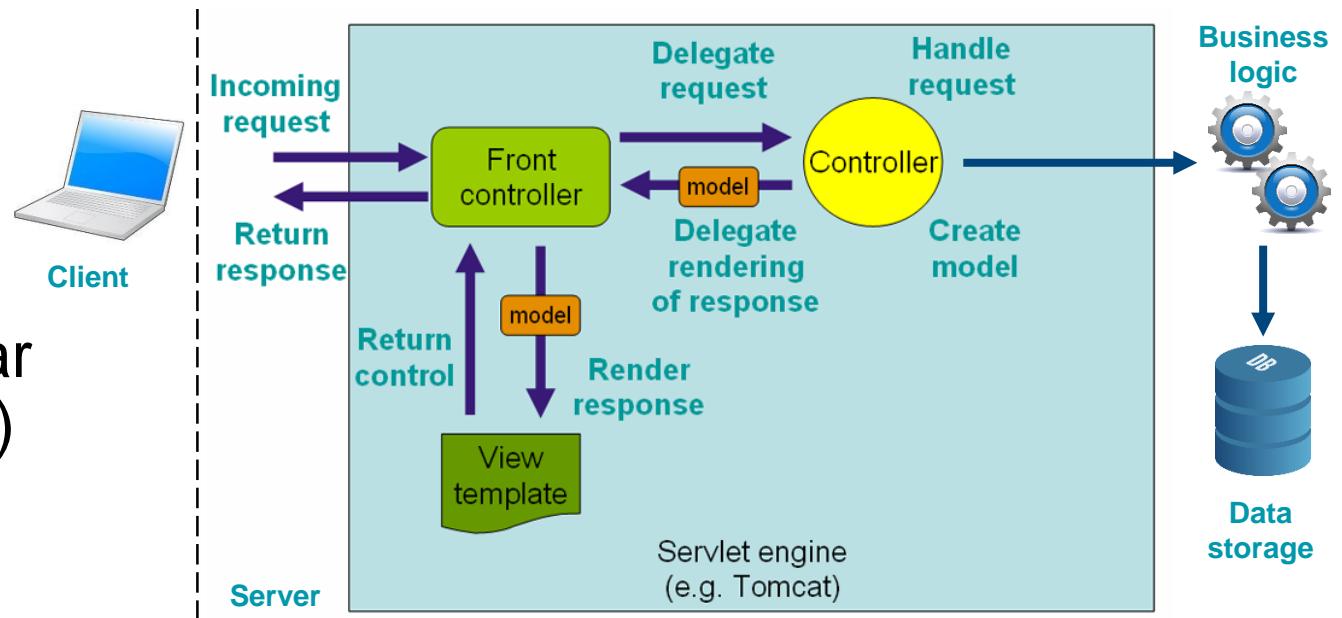
Integrate model contents into web page

- here: insert contents of model element nom



Handling More Complex Form Data

- Usually, we don't want to work on the level of individual request parameters
- Often, users are working with forms that correspond to particular data structures (business objects) on the server



- Spring Web MVC simplifies this
 - by allowing us to refer to JavaBeans implementing the business objects directly
 - in the formulation of the form
 - in the handling of the request

JavaBean for Storing Form Data

```
package hello;

public class UserInfo {
    private long id;
    private String email;

    public long getId() { return id; }
    public void setId(long id) { this.id = id; }

    public String getEmail() { return email; }
    public void setEmail(String email) { this.email = email; }
}
```

Simple JavaBean (i.e. a plain old Java object with private attributes and public getters and setters) describing the data we want to capture in the form

- here: information about a user, e.g. an ID and e-mail address



Controller Distinguishing Requests for Form and Result

```
package hello;  
// [import statements]  
@Controller  
public class UserInfoController {  
  
    @RequestMapping(value="/info", method=RequestMethod.GET)  
    public String userInfoForm(Model model) {  
        model.addAttribute("userinfo", new UserInfo());  
        return "InfoForm";  
    }  
  
    @RequestMapping(value="/info", method=RequestMethod.POST)  
    public String userInfoSubmit(@ModelAttribute UserInfo userInfo, Model model) {  
        model.addAttribute("userinfo", userInfo);  
        return "InfoResult";  
    }  
}
```

Under the URI /info, two different behaviors and resulting views shall be accessible:

- Upon a GET request (i.e. a regular link to the page), show the page InfoForm.jsp
- Upon a POST request (i.e. a submission of the form), show the page InfoResult.jsp

- Creates a new UserInfo object
- Adds it to the model
- Delegates to InfoForm.jsp

- Retrieves UserInfo object from request (with attributes populated through the form)
- Adds it to the model
- Delegates to InfoResult.jsp



Form View Template

Stored on server as
InfoForm.jsp

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
  <head>
    <title>Handling Form Submission</title>
    <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
  </head>
  <body>
    <h1>Form</h1>
    <form action="#" th:action="@{/info}" th:object="${userinfo}" method="post">
      <p>Id: <input type="text" th:field="*{id}" /></p>
      <p>e-Mail: <input type="text" th:field="*{email}" /></p>
      <p><input type="submit" value="Submit" /> <input type="reset" value="Reset" /></p>
    </form>
  </body>
</html>
```

Submit form data to controller at URL /info

Populate form with data from object stored under userinfo in model

Submit form data in HTTP POST request

Populate form fields with attributes id and email of userinfo object



Result View Template

Stored on server as
InfoResult.jsp

```
<!DOCTYPE HTML>
<html xmlns:th="http://www.thymeleaf.org">
    <head>
        <title>Handling Form Submission</title>
        <meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
    </head>
    <body>
        <h1>Result</h1>
        <p th:text="'Id: ' + ${userinfo.id}" />
        <p th:text="'e-Mail: ' + ${userinfo.content}" />
        <a href="/info">Return to form</a>
    </body>
</html>
```

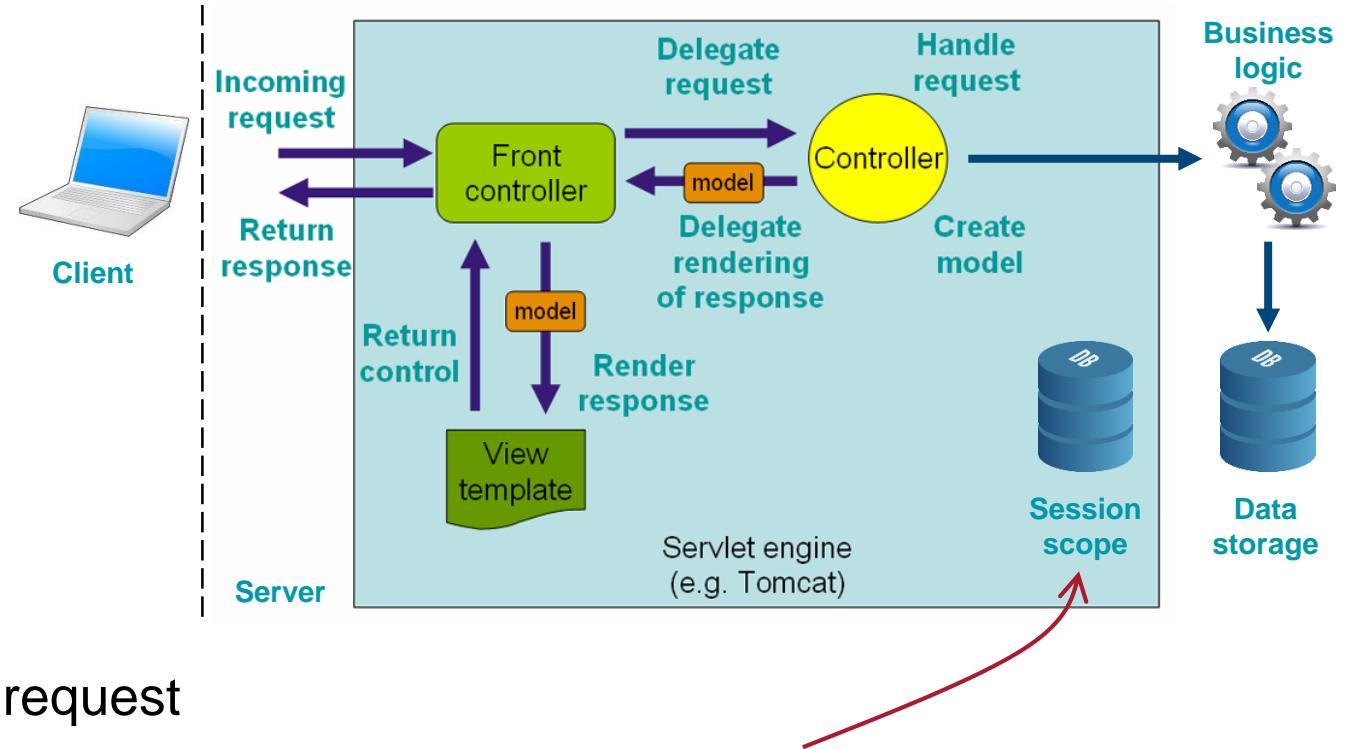
Get attributes id and content from object found under label userinfo in model, and integrate into HTML code

Trigger HTTP GET request



Preserving State Between Requests

- Often, we want to maintain a certain state on the server between requests
 - i.e. keep data on the server that is specific to a particular user's session
 - but should not be transferred back and forth with each request
- Tedious approach: We could
 - store such data in the database and retrieve it every time, based on some user ID transferred with each request
- The Java Servlet API simplifies this by providing a session scope that
 - is associated with a user's requests automatically
 - lets us store and retrieve objects by name
 - exists only for the duration of the user's session



Working with Session Attributes

```
@Controller  
public class GameStateController {  
    @RequestMapping(value="/update", method=RequestMethod.GET)  
    public String stateUpdate(HttpSession session, Model model) {  
        PlayerState ps = (PlayerState) session.getAttribute("playerstate");  
        // [application logic providing Score object]  
        session.setAttribute("myscore", score);  
        // [application logic]  
        session.removeAttribute("tempState");  
        model.addAttribute("userinfo", new UserInfo());  
        return "Game";  
    }  
}
```

Make HTTP session available in request handler

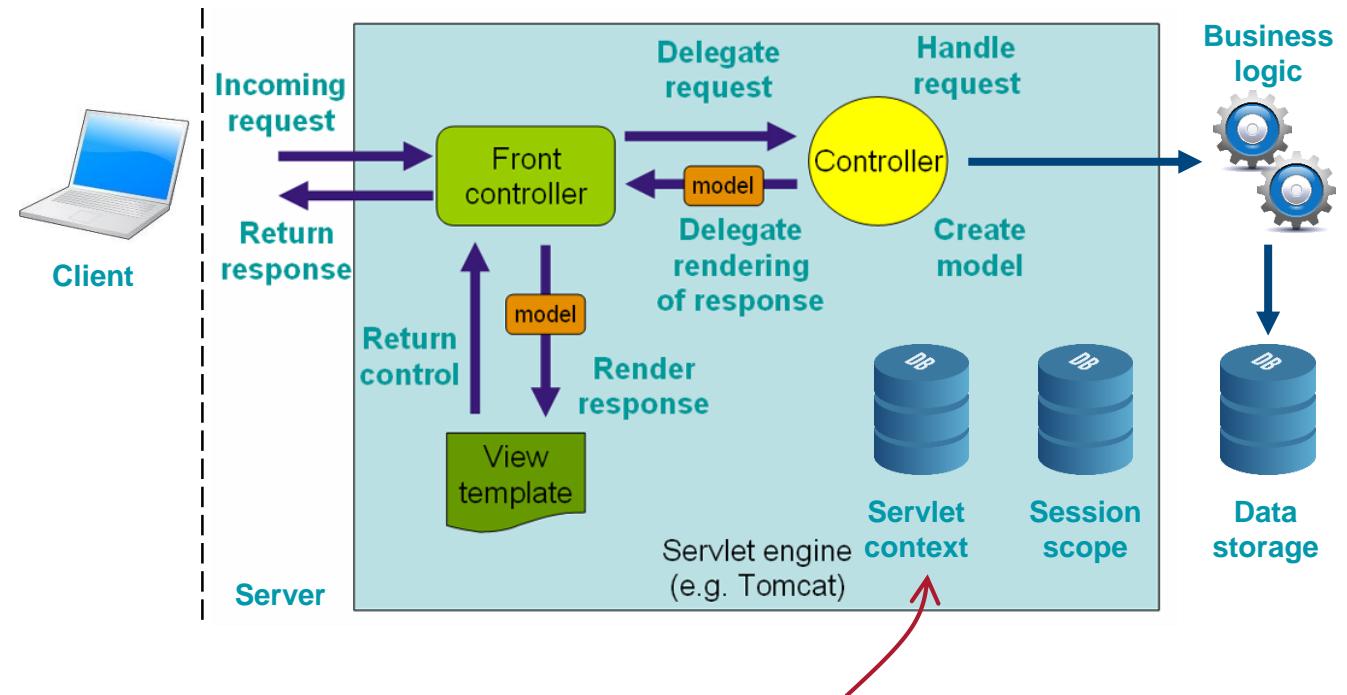
Retrieve object stored under given label from the session

Store given object under given label in the session

Remove object stored under given label from session

Sharing Data Across Sessions

- Sometimes, we want to make certain information available to all sessions of an application
- Tedious approach: We could
 - store such data in the database and retrieve it anytime we need it
 - but it is not object-oriented there
- The Java Servlet API simplifies this by providing an application scope that
 - is available in all sessions
 - lets us store and retrieve objects by name
 - exists as long as the application is deployed on the server and the server is running



Working with the Servlet Context

```
@Controller  
public class GameStateController {  
    @RequestMapping(value="/next", method=RequestMethod.GET)  
    public String nextRound(HttpServletRequest session, Model model) {  
        ServletContext context = session.getServletContext();  
        GameState gs = (GameState) context.getAttribute("gamestate");  
        // [application logic providing Score object]  
        context.addAttribute("hiscore", score);  
        // [application logic]  
        context.removeAttribute("someState");  
        return "Start";  
    }  
}
```

Make application context available in request handler

Retrieve object stored under given label from context

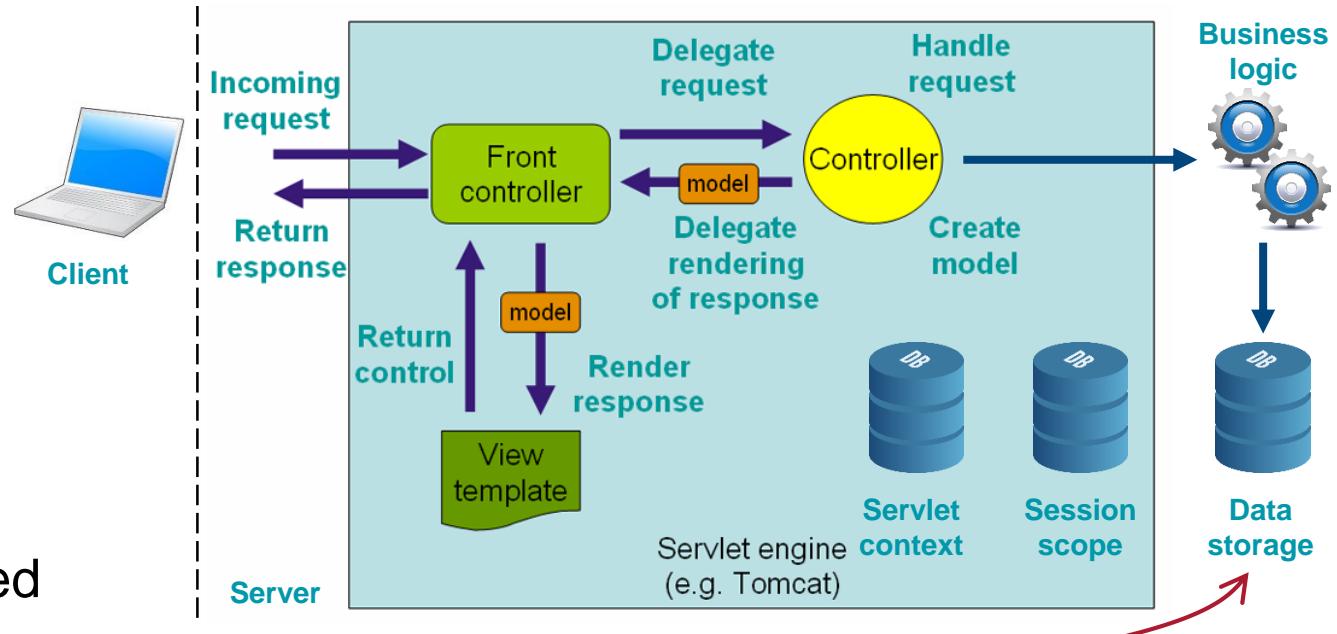
Store given object under given label in context

Remove object stored under given label from context



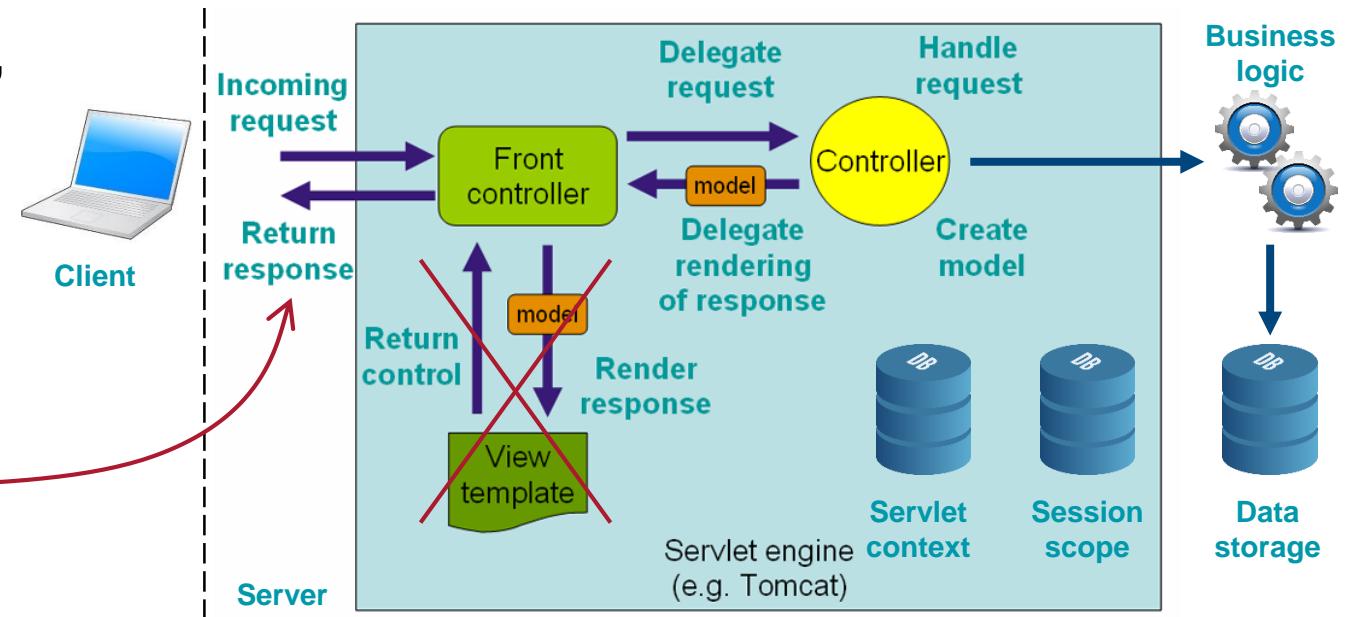
Persistent Data Storage

- Some data is not suitable for storage in the servlet engine's scopes, e.g.
 - data that shall be stored even when the server is down
 - data that is too large to be kept in memory
 - data that is most efficiently stored and retrieved in a non-object-oriented structure (e.g. relational data)
 - data that is retrieved from external sources
- For these purposes, a database or other data sources can be accessed from the business logic
 - Persistence frameworks can help with the mapping of objects to database structures



Returning Data Instead of a View

- When implementing a REST API, we don't need a view template to construct a web page
- Instead, the controller should create JSON or XML data that is returned directly in the HTTP Response



- Spring Boot simplifies this
 - by enabling us to bypass the view template
 - and automatically converting JavaBeans to a JSON representation

JavaBean Containing Data to Transfer

```
package hello;  
  
public class Greeting {  
  
    private final long id;  
    private final String content;  
  
    public Greeting(long id, String content) {  
        this.id = id;  
        this.content = content;  
    }  
  
    public long getId() { return id; }  
    public String getContent() { return content; }  
}
```

Plain old Java object with constructor and accessor methods) describing the data we want to transmit



REST Controller

```
package hello;  
// [import statements]  
  
@RestController  
public class GreetingController {  
  
    private static final String template = "Hello, %s!";  
    private final AtomicLong counter = new AtomicLong();  
  
    @RequestMapping("/greeting")  
    public Greeting greeting(@RequestParam(value="name", defaultValue="World") String name) {  
        return new Greeting(counter.incrementAndGet(),  
                           String.format(template, name));  
    }  
}
```

Indicates that this controller's request handlers will not determine a view template, but return data for inclusion in the HTTP response directly

Example request:
`http://localhost:8080/greeting?name=Jon`

Resulting HTTP response body:

```
{  
    "id": 1,  
    "content": "Hello, Jon!"  
}
```

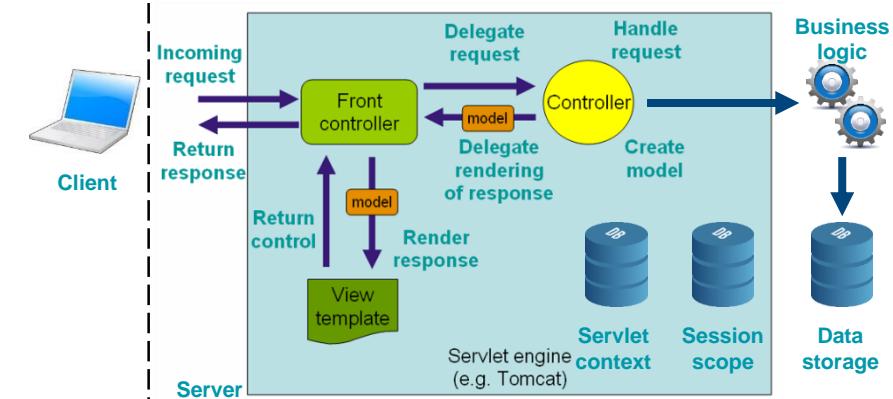
Request processing (URI mapping, parameter extraction, session access etc.) works as usual

- Construct and return the object to be sent to the client.
- Spring will use the Jackson library to automatically convert this object to JSON and include it in the response body.



Summary: Spring Web MVC Basics

- Build `@Controllers` to respond to different requests
- Use `@RequestMappings` to define which controller method should react to which URI
- Extract input from requests using
 - `@RequestParameters` for single request parameters
 - `@PathVariables` for parts of the URI path
 - `@ModelAttribute` for parameters describing objects
- Store and retrieve information spanning multiple requests in the `HttpSession`
- Store and retrieve information available to all sessions in the `ServletContext`
- Invoke application logic in regular Java classes from controller
- Store information to be provided to the next view in the Model
- Specify the next view in the return value of the request handler
- Construct views as JSPs that incorporate information from the Model
- Or use a `@RestController` to respond with JSON data instead of a view



Team Assignment 2

- Three of the most important artefacts created in the Elaboration phase are:
 1. **The domain model**
 - Describing the concepts and structures of the application domain your software works with
 2. **The Software Architecture document**
 - Describing the architectural drivers and the architectural decisions made to address them
 3. **The executable architecture**
 - Providing a technical backbone on which you can proceed to implement your business logic
- Producing these artefacts and explaining the considerations that went into them will be your job in Team Assignment 2.

Team Assignment 2: Content

- By **Sun 18 Oct**, submit in Uglá:
 - A **domain model** for your project, containing: (~50% of grade)
 - A **UML class diagram** covering all relevant objects of your application domain (~60%)
 - Which things, actors, concepts etc. are there, and how are they related with each other?
 - A **UML sequence or activity diagram** describing key processes in your application domain (~40%)
 - How do things work in your application domain? How does your software support that / integrate with that?
 - Note: This is not a model of the technical implementation yet, just a model of the real-world concepts and their relationships!
 - An initial **Software Architecture document**, containing: (~50%)
 - An **architectural drivers table**, following the template on slide 25 of lecture 5 (~33%)
 - Describe 3-5 architectural drivers that are relevant for your project (if you identify more (realistic ones), great!)
 - **Technical memos** documenting architectural decisions, following template on slide 30 of lect. 5 (~33%)
 - Describe decisions made to deal with 2-3 of the above-mentioned drivers (if you can describe more, great!)
 - A **UML package diagram** describing the key components of your architecture (~17%)
 - A **UML sequence diagram** describing the control flow between packages of your architecture (~17%)
- On **Thu 22 Oct**, demonstrate and explain your **executable architecture**:
 - Have a skeleton of your system running on your computer to illustrate that your components can exchange data with each other, and explain to your tutor how they do it
 - No need for any business logic or nice user interfaces; just have the technical components wired up

Team Assignment 2: Format

- All artefacts must be produced by all team members together.
- The **domain model** and **Software Architecture** document must
 - be submitted as **PDF documents by Sun 18 Oct in Ugla**
 - contain your team number, the names and kennitölur of all team members
- The code of the executable architecture does not need to be submitted.
- Only the team member who will present should submit the documents.
 - Don't submit multiple versions – we'll just grade the first one we encounter!
- The **demonstration of the executable architecture on Thu 22 Oct**
 - should be given by one representative of the team (a different one for each assignment!)
 - should be focused on the running system on your computer (don't prepare extra slides!)
 - should also refer to the UML diagrams in your software architecture document for support
 - will also involve answering questions about the submitted documents
 - should take around 10 minutes (plus some questions asked by the tutor)
- All team members receive the same grade, with some variation for the presenter



Gangi þér vel!

book@hi.is



HÁSKÓLI ÍSLANDS
VERKFRÆÐI- OG NÁTTÚRVÍSINDASVIÐ

IÐNAÐARVERKFRÆÐI-, VÉLAVERKFRÆÐI-
OG TÖLVUNARFRÆÐIDEILD





Hugbúnaðarverkefni 1 / Software Project 1

7. Domain Models

HBV501G – Fall 2015

Matthias Book



HÁSKÓLI ÍSLANDS
VERKFRÆÐI- OG NÁTTÚRVÍSINDASVIÐ
ÍÐNAÐARVERKFRÆÐI-, VÉLAVERKFRÆÐI-
OG TÖLVUNARFRÆÐIDEILD

Miðmisseriskönnun

Evaluation Results



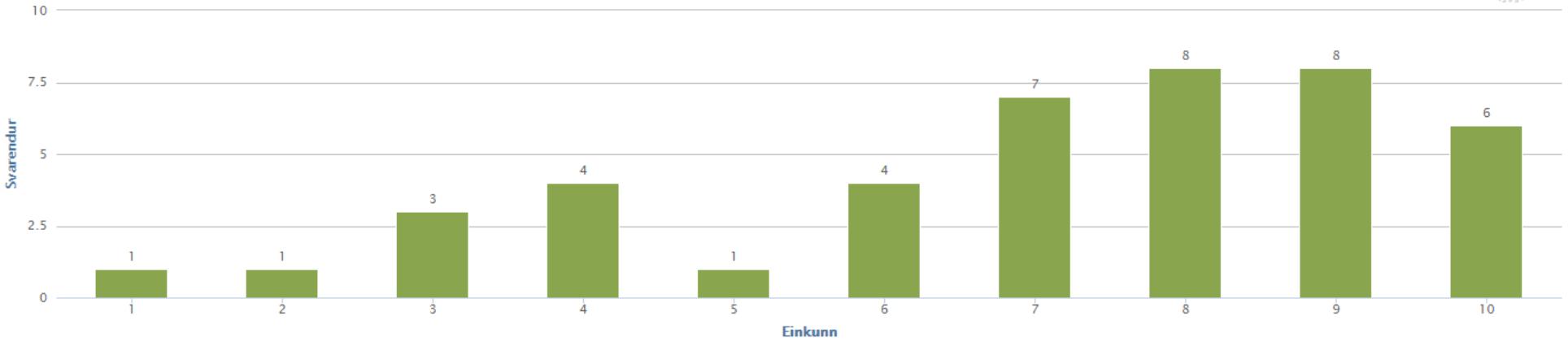
HÁSKÓLI ÍSLANDS

Matthias Book: Software Development

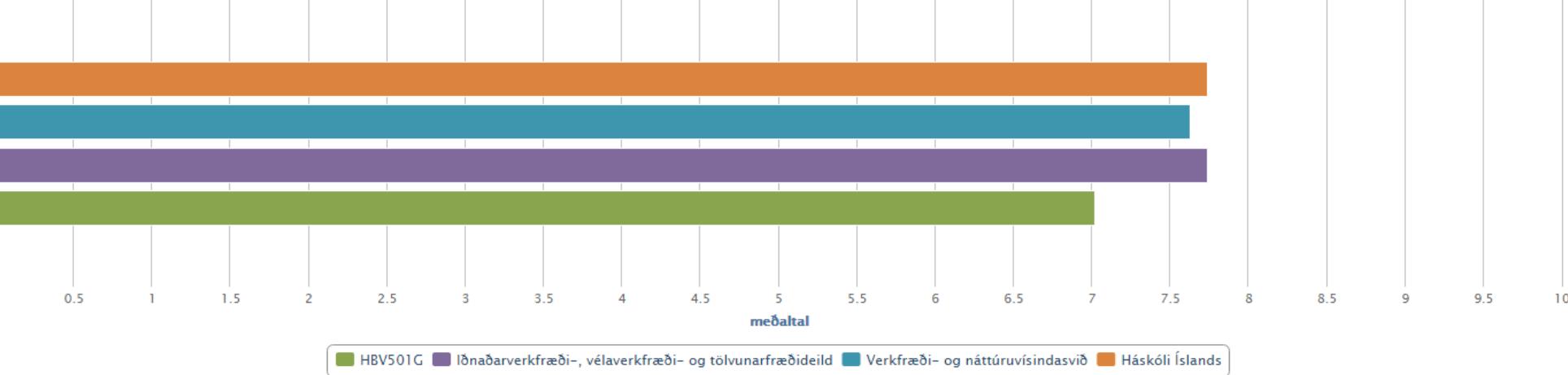
Grades



Gefðu námskeiðinu einkunn:
Dreifing einkunna



Gefðu námskeiðinu einkunn:
Samanburður



Participation:
43/94 (46%)



Answers to Open Questions (paraphrased)

What people like

- Free choice of project topics
- Clear slides
- Helpful tutors
- Guest lectures

- Clearer assignment requirements, more transparent grading
 - Striving to explain more, but ask if unsure
 - General rule: Substance will be rewarded

What people would like

- Free choice of “more modern” technology stack
 - OK, will relax client-side restrictions
 - But insist on all object-oriented code and Java-based server-side logic

- Stop planning and start programming
 - No, that’s the whole point of the course 😊
 - Document-heavy approach may be tedious when building a new gaming app
 - But essential to understand complex business information systems



What Makes a Great Software Engineer?

ACM Webcast, 18:00-19:00 GMT **tonight**



- Paul Li and Andrew Ko discuss **what software engineers think are the attributes of software engineering expertise**, based on interviews spanning numerous divisions at Microsoft, including several interviews with software architects having over 25 years of experience. They discuss attributes spanning engineers' personality attributes, decision-making abilities, interactions with teammates, and software designs.
- **Paul Li** is a Senior Data Scientist at Microsoft's Windows and Devices Group, with a decade of research and practical experience at Avaya, ABB, IBM, and Microsoft.
- **Andrew Ko** is Associate Professor at the University of Washington Information School, with a research focus on human-computer interaction, computing education, and software engineering.

Registration: <http://event.on24.com/wcc/r/1054141/1119C0B0805E617FE139FC7DB74621CE>



Domain Models

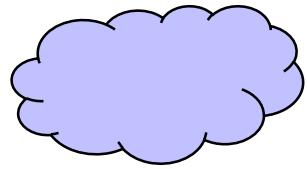
see also:

- Larman: Applying UML and Patterns, Ch. 9 & 32

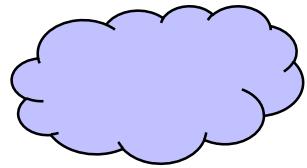


Recap: Inception and Elaboration Artifacts

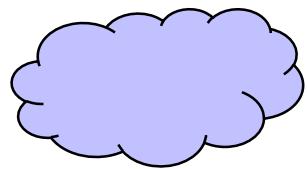
Business Domain



Business Requirements



Quality Attributes



Business Rules

Requirements



Vision and Scope

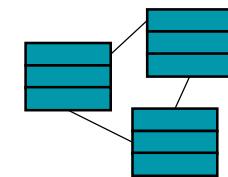


Supplementary Specification

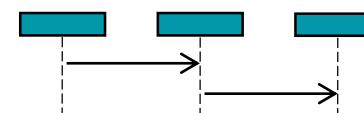


Use Cases

Domain Model

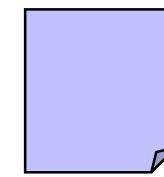


Structural Diagrams

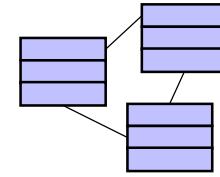


Interaction Diagrams

Architecture

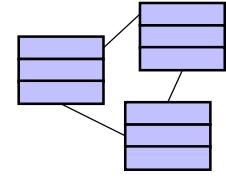


Software Architecture

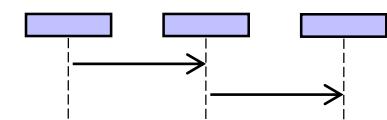


Package Diagrams

OO Design



Structural Diagrams



Interaction Diagrams

Goals of Domain Models

- Illustrates noteworthy concepts / terms / objects in an application domain
 - i.e. an object-oriented model of the real world
 - more precisely: the *segment* of the real world that is *relevant* to your system
 - Note: Do not try to model the real world in all its detail – time-consuming and largely irrelevant!
- Source of inspiration for (later) object-oriented design of application logic
 - i.e. shows which concepts, relationships, dynamics the system must support
 - even if not all domain model elements may have exact counterparts in the implementation
 - Note: Do not model implementation-specific classes (i.e. your system's technical design) here!

➤ **Goal: Understand the application domain,
so you can build software supporting it properly**



Creating a Domain Model

- A domain model is typically expressed as a UML class diagram showing
 - Conceptual classes (relevant things, concepts, ideas in the application domain)
 - Associations between conceptual classes
 - Attributes of conceptual classes (but no methods)
- UML sequence or activity diagrams can be added to understand the dynamics of the application domain
- **Approach:** For the requirements under design *in the current iteration*:
 1. Identify the conceptual classes
 2. Draw them as classes in a UML class diagram
 3. Add associations and attributes (but no methods)
 4. Draw UML sequence or activity diagrams to express dynamic aspects



Use a Pragmatic Modeling Approach

- Do not guess
 - Work together with domain experts, i.e. people who are regularly exposed to the concepts
- Do not spend more than a few hours on domain modelling in each iteration
 - Your goal is not to precisely document the whole world
 - But to understand what the relevant aspects for your system are, and how they work
- Do not use a modeling tool
 - Just use a whiteboard and take a picture when you are done
 - Much more intuitive and straightforward, especially for domain experts working with you
 - Domain model will ultimately be replaced by your system model, which is worth more preservation effort



Strategies for Identifying Conceptual Classes

- a) Extract noun phrases
 - e.g. in fully dressed use cases and the Vision and Scope document
- b) Use a category list
 - especially helpful for business information systems
- c) Reuse or modify existing models
 - something may be available from the client



Identifying Conceptual Classes by Extracting Nouns

- Example: Extraction from a use case's main success scenario

1. Customer arrives at POS checkout with goods and/or services to purchase
2. Cashier starts new sale
3. Cashier enters item identifier
4. System records sale line item and presents item description, price, and running total.
Price calculated from set of price rules.
5. System presents total with taxes calculated.
6. Cashier tells Customer the total, and asks for payment.



- Caution: Easy for a quick start, but needs refinement with common sense

- Not every noun is a conceptual class
- Some conceptual classes may not be mentioned explicitly
- Some nouns may refer to classes, some to attributes
- Natural language can be ambiguous

Identifying Conceptual Classes From Categories

| Conceptual Class Category | POS System Examples | Airline Examples | Game Examples |
|--|-----------------------------|----------------------|-------------------|
| Business transactions | Sale, Payment | Reservation | |
| Transaction line items | Sales Line Item | | |
| Products/services related to transactions/line items | Item | Flight, Seat, Meal | |
| Actors/roles of people related to transactions | Cashier, Customer | Passenger, Airline | Player |
| Place of transaction/service | Store | Airport, Plane, Seat | |
| Noteworthy events (often with assoc. time or place) | Sale, Payment | Flight | Game |
| Physical objects | Item, Register | Airplane | Board, Piece, Die |
| Descriptions of things (type information) | Product Description | Flight Description | |
| Catalogs | Product Catalog | Flight Catalog | |
| Containers of things | Store, Bin | Airplane | Board |
| Things in a container | Item | Passenger | Square, Piece |
| Collaborating systems | Credit Authorization System | Air Traffic Control | |
| Records of finance, work, transactions etc. | Receipt, Ledger | Maintenance Log | |
| Financial instruments | Cash, Check, Line of Credit | Ticket Credit | |

For inspiration only – you typically won't have classes for all of these



Describing the Application Domain (“Real World”)

- Use the existing terminology of the application domain, don't invent new terms
 - If there is a precise term for a concept, use it so users will know what you are talking about
 - Terms you choose will stick all the way to the implementation, user interface, documentation etc.
- Exclude irrelevant or out-of-scope aspects of the application domain
 - If your system's implementation doesn't need to “know” about it, don't include it in the model
 - You can include concepts that will become relevant in later iterations (so you're already aware of them), but you should not spend effort on describing them in detail yet
- Don't add concepts to the model if they don't exist in the application domain
 - Of course, you should model relevant non-physical concepts (e.g. Agreement, Route, Target...)
 - Don't add concepts based on your assumptions or opinions on how things should work
 - Don't add concepts that you need for technical reasons

- **How would you describe the structures and processes that your system supports**
- **to someone who just cares about what the system does, but not how it works?**
 - **to someone who shall implement the system using a technology unknown to you?**

Modeling Derived Information

- Should items containing derived data be modeled, or only “original” objects?
- Example:
 - A Receipt seems to be a noteworthy concept in a POS system, but then again it's just a report on a Sale and Payment, i.e. derived information. Should we include it in the model?
 - **Con:** Usually, a report comprises only information derived from other, original sources, so it should be sufficient to model the original concepts.
(The report is then considered as more of a “view” than an “object”.)
 - **Pro:** A report can play a particular role in business processes (a Receipt, for example, is necessary to return bought items). If this is a relevant use case for the system, the report should be included as a conceptual class.

➤ Guideline: When in doubt, include a concept...

- At least you're aware of it – but don't clutter up the domain model with things “just in case”
- ...and see later if you really need to implement it
 - Critically examine conceptual classes and see which ones your application logic depends on

Modeling Type Information

- Should type information (i.e. general characteristics of a group of objects) be explicitly modeled?
- Example:
 - A store sells items with some generic, some individual characteristics (e.g. custom stamps).
 - Generic characteristics of all stamps: Manufacturer, material, price...
 - Specific characteristics added upon ordering: Individual lettering
 - Is it sufficient to have just one Stamp class, or should we distinguish between GeneralStamp and OrderedStamp classes?
- Guiding questions:
 - What would happen if all individual stamp objects were deleted?
 - Would your system miss crucial information to sell new stamps?
 - What should happen if some general characteristic of stamps had to be changed?
 - Would you need to change information in all individual stamp objects as well?
 - Or would you want to prevent the individual stamp objects from being affected by changes?



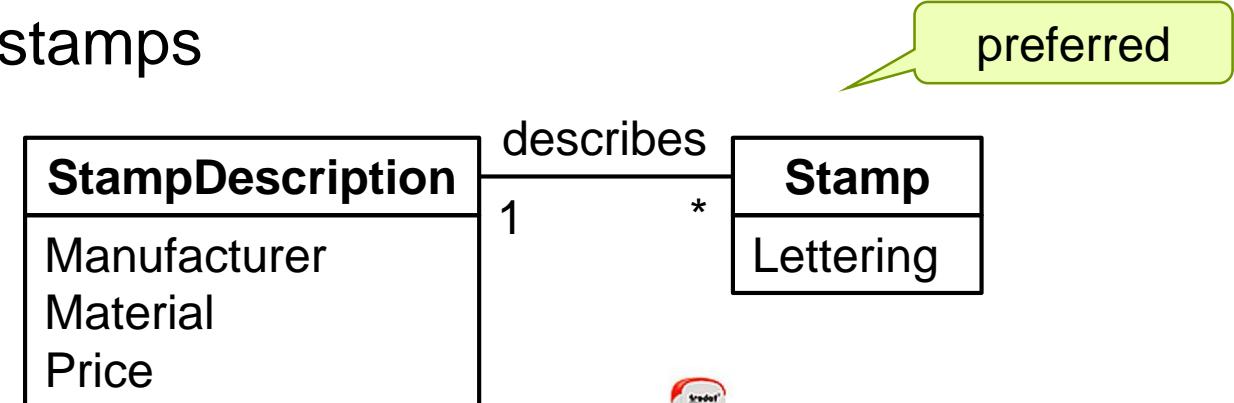
Include a separate general “type” class if you answer YES to any of these

Modeling Type Information

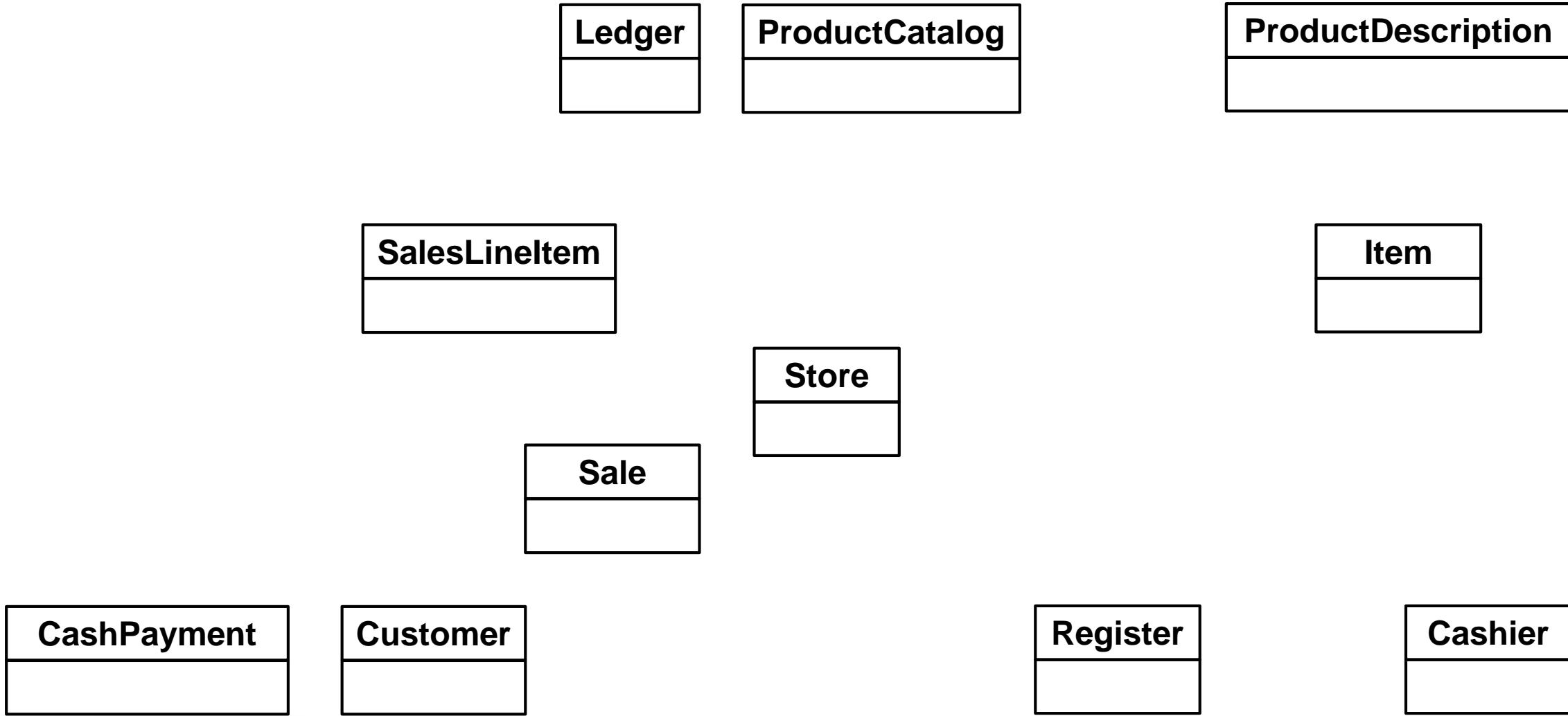
- Guidelines: Add a type class (e.g. ItemDescription for an Item) if
 - there needs to be a description of an item, independent of the current existence of any instances of that type of item
 - deleting instances of items would result in loss of information that needs to be maintained about general characteristics of that type of item
 - it avoids the storage of redundant or duplicated information in each instance of the item
- Example: Modeling customizable stamps

discouraged

| Stamp |
|--------------|
| Manufacturer |
| Material |
| Price |
| Lettering |



Example: Initial POS System Domain Model



Adding Associations

- An association between two classes indicates the presence of a **relevant, memorable relationship** between instances of those classes.
 - i.e.: Between which objects do we need a long- or short-term memory of a relationship? E.g.:
 - A SalesLineItem must be permanently associated with a Sale
 - A Piece must be permanently associated with a Player, and temporarily with a Square on a Board
 - But while a Cashier can look up ProductDescriptions, there is no need to remember which Cashier looked up which ProductDescriptions (i.e. no association)
- Associations can also indicate other common relationships
 - see suggestions on next slide
- Beyond that, be reluctant about adding too many associations
 - Focus on the main structural bonds, not everything that somehow has to do with something



Common Relationships Expressed by Associations

| Association Category | POS System Examples | Airline Examples | Game Examples |
|--|-----------------------------------|-------------------------------|-----------------|
| A is a transaction related to another transaction B | Cash Payment – Sale | Cancellation – Reservation | |
| A is a line item of a transaction B | Sales Line Item – Sale | | |
| A is a product for a transaction (or line item) B | Item – Sales Line Item | Flight – Reservation | |
| A is a role related to a transaction B | Customer – Payment | Passenger – Ticket | |
| A is a physical or logical part of B | Drawer – Register | Seat – Airplane | Square – Board |
| A is physically or logically contained in B | Item – Shelf | Passenger – Airplane | Square – Board |
| A is a description for (or: the type of) B | Item Description – Item | Flight Description – Flight | |
| A is known/recorded/reported/captured in B | Sale – Register | Reservation – Flight Manifest | Piece – Square |
| A is a member of B | Cashier – Store | Pilot – Airline | Player – Game |
| A is an organizational subunit of B | Department – Store | Maintenance – Airline | |
| A uses/manages/owns B | Cashier – Register | Pilot – Airplane | Player – Piece |
| A is next to B | Sales Line Item – Sales Line Item | City – City | Square – Square |

For inspiration only – you typically won't have associations for all of these



Using and Interpreting Associations in Domain Models

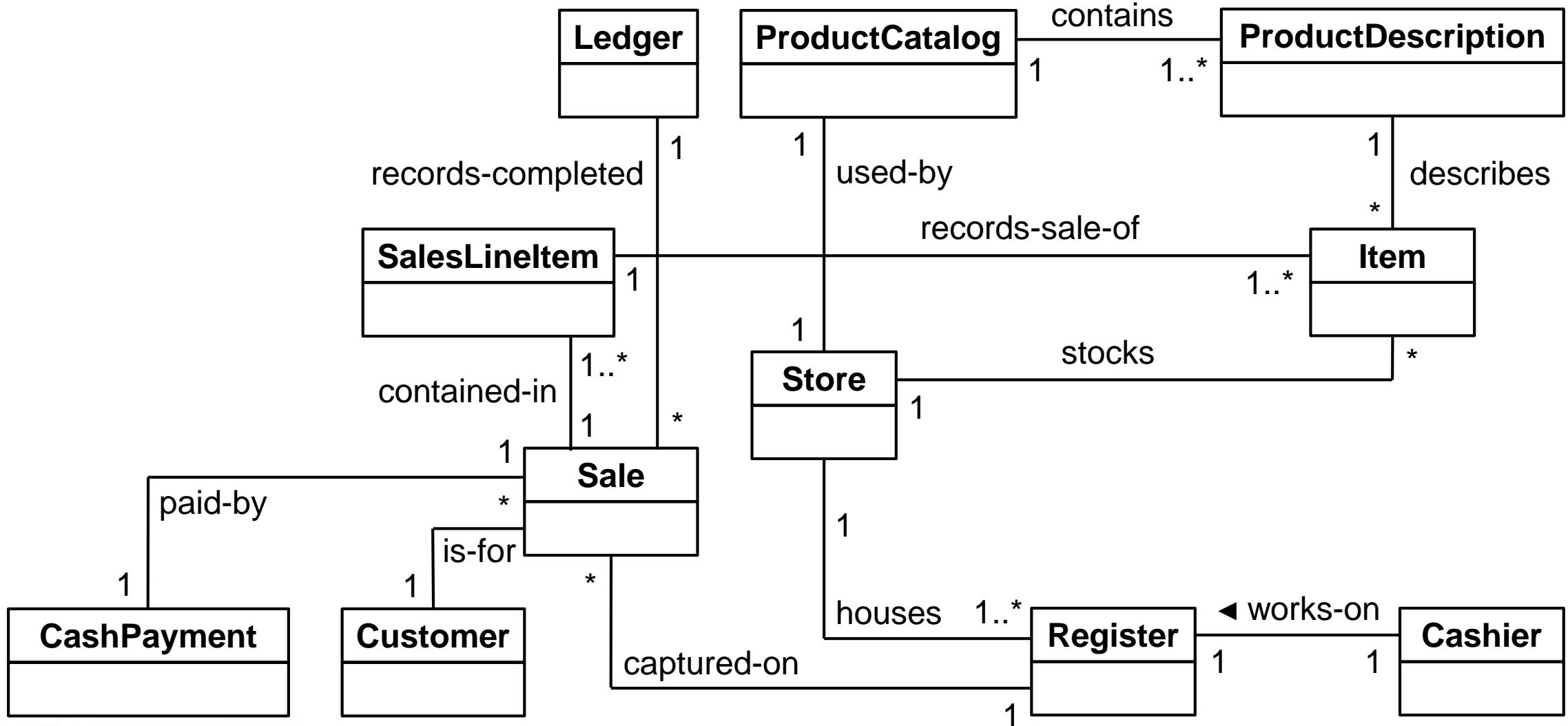
- Domain model associations are **bidirectional**
 - Arrow tips can be added for clarity, but don't prescribe references between software classes
- Associations should be **labeled** with verb phrases connecting the class names
 - e.g. Store stocks Item, Sale paid-by Cash Payment, Flight flies-to Airport, Piece is-on Square
- Ends of associations can be decorated with **multiplicities** indicating how many instances of a class can be associated with one instance of the other class at the same time
 - e.g. 0..1 (zero or one), * (zero or more), 1..* (one or more), 1 (exactly one), 1..42 (one to 42)
- **Multiple** associations can connect classes to symbolize different relationships
 - e.g. Flight flies-from Airport, Flight flies-to Airport
- An association can be **reflexive**, i.e. relating instances of the same class
 - e.g. Folder contains Folder



Associations in Object-Oriented Analysis vs. Design

- *In a domain model*, associations are NOT
 - Statements about data flows
 - Database foreign key relations
 - Instance variables
 - Object references in software
- Upon moving towards system models in object-oriented design...
 - Some domain model associations may survive in the technical model of the system and indicate data navigation, visibility, multiplicity etc.
 - Some domain model associations may be implemented differently/indirectly
 - Some domain model associations may be eliminated or replaced with other constructs
 - Some associations may be added to facilitate convenient data access
- ...but you shouldn't consider any of this yet when creating the domain model.

Example: POS Domain Model with Associations

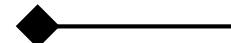


Aggregation and Composition

- **Aggregation** describes a “contains” relationship 

 - e.g. a Container holds Goods
 - But is only weakly defined in the UML, has no clear implications for the technical design, and is just as well expressed by a regular association

 - Guideline: Don't bother modeling it explicitly

- **Composition** describes a “consists of” relationship implying that 

 - A part instance (e.g. Square) belongs to only one composite instance (e.g. Board) at a time
 - Any part must belong to a composite at any time (“no loose Fingers”)
 - The composite is responsible for the creation and deletion of its parts, i.e.
 - upon its creation, a composite creates or requires its constituent parts
 - upon its deletion, a composite deletes its parts or attaches them to another composite
 - Operations applied to the composite propagate to / affect the parts
 - Most of these implications become only relevant in the technical design

 - Guideline: Can be defined in the domain model, but is not crucial – if in doubt, leave it out

Attributes in Domain Models

- An attribute is a logical data value of an object.
 - e.g. a Sale needs Date and Time attributes, a Product needs a Price attribute, etc.
- Guideline: Include attributes in the domain model that...
 - are needed to satisfy information requirements expressed in use cases
 - characterize individual instances of a class in a relevant way
- Attributes are typically primitive data types or trivial data structures
 - i.e. they are Boolean values, dates, numbers, characters, text, or addresses, phone numbers, colors, shapes, product codes, ISBNs, enumerated types (Size = {S, M , L}) etc.
 - But their data type is typically not specified in the model (unless it is non-obvious), and does not prescribe particular types to be used in the technical design
- Do not include attributes that are merely technical IDs or foreign keys
 - An ISBN is fine (part of the real world), a database ID is not (implementation detail)
- Do not include derived attributes, unless their omission would be confusing
 - e.g. no need for “total price” attribute of a Sale that can be derived from LineItems’ prices

Classes vs. Attributes

- When should a thing be modeled as a conceptual class, when as an attribute?
- Guidelines:
 - If we do not think of something as a number or text in the real world, it is probably a conceptual class, not an attribute.
 - Example: A Flight class' destination airport should not just be a string attribute ("KEF"), but an associated Airport class which may have attributes of its own
 - If equality is based on identity instead of value, it should be a conceptual class
 - Example: 1000 ISK and 1000 ISK are the same price (equality based on value → attribute), but Jón Jónsson and Jón Jónsson may be two distinct people (equality based on identity → class).
- Conceptual classes should always be related with an association, not expressed as an attribute in the domain model (*see next slide for exception*)
 - To facilitate richer description of the conceptual classes and associations
 - This does not prescribe a certain technical implementation.



Classes vs. Attributes

- Sometimes, an aspect that is originally considered to be of a primitive type (i.e. an attribute) may turn out to be better represented as a conceptual class, e.g. if:
 - It is composed of separate sections (that are relevant for the system to distinguish)
 - e.g. name → first name, middle name, last name, prefix, suffix, gender...
 - There are operations associated with it, such as parsing or validation
 - e.g. validate a credit card number, extract a birth date from a kennitala
 - It has other relevant attributes
 - e.g. a price may have a start and end date, be valid for a certain amount of items only, etc.
 - It is a quantity with a unit that is not fixed
 - e.g. payments in different currencies
- These kinds of conceptual classes may be specified as an attribute or an association in a domain model
 - Choose whichever way seems more natural

| Price |
|---------------|
| amount |
| validFromDate |
| validToDate |

1

| Item |
|---------------|
| name |
| price : Price |

| Item |
|------|
| name |

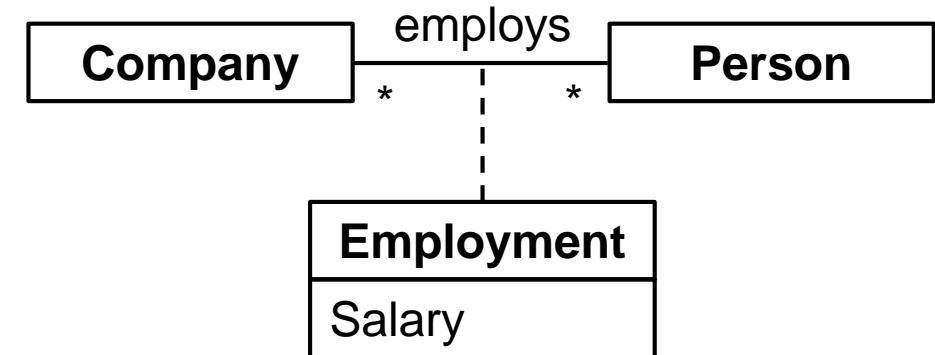
or

both ok

Association Classes

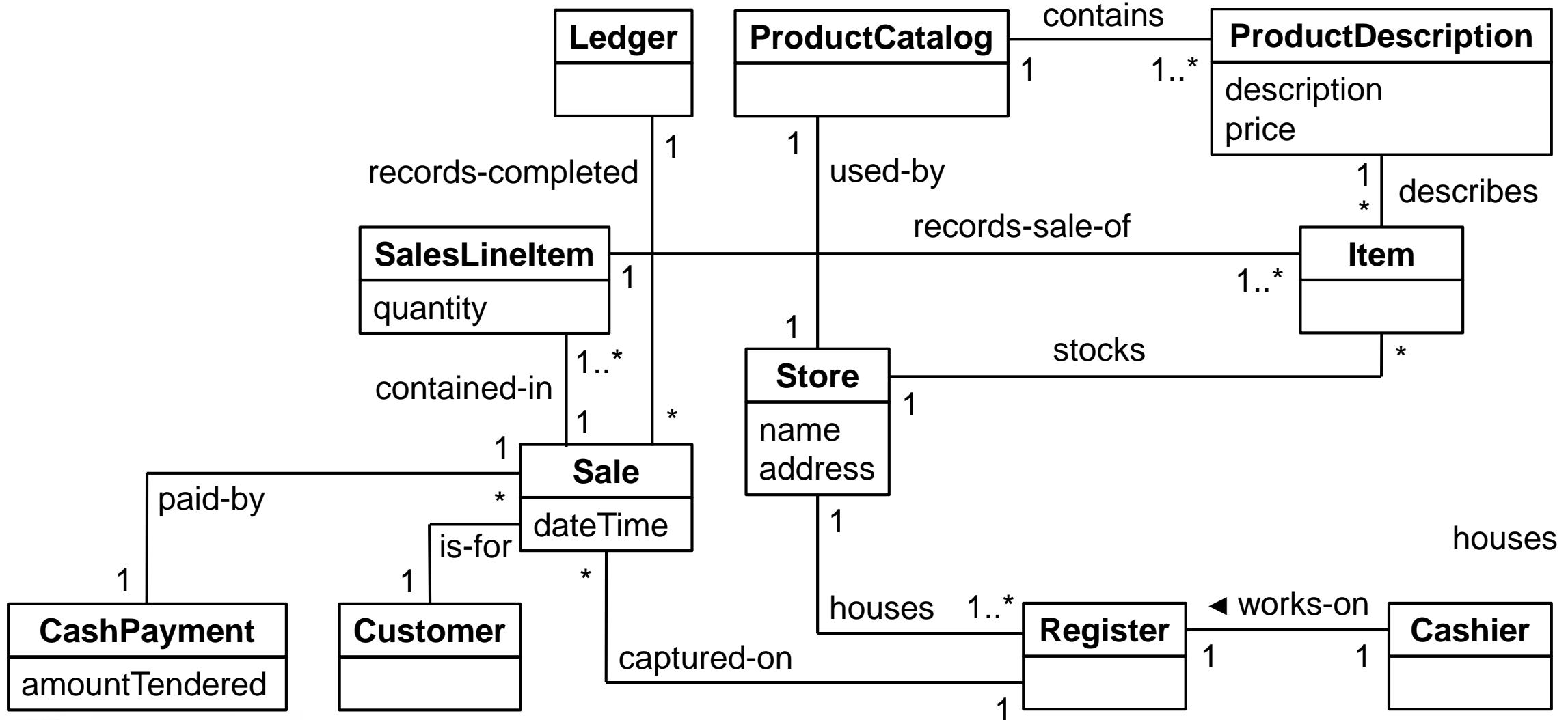
- Associations may have attributes just like classes do
 - Example: A person may have employment contracts with several companies, but the contract details are neither attributes of the person nor of the company
 - They are attributes of the association

➤ Model such attributes in **association classes**



- Guidelines for association classes:
 - Use whenever a attribute is related to an association, but not the classes it connects
 - There is a many-to-many association between two concepts, and information associated with the association itself
 - The lifetime of instances of the association class depend on the lifetime of the association

Example: POS Domain Model with Attributes



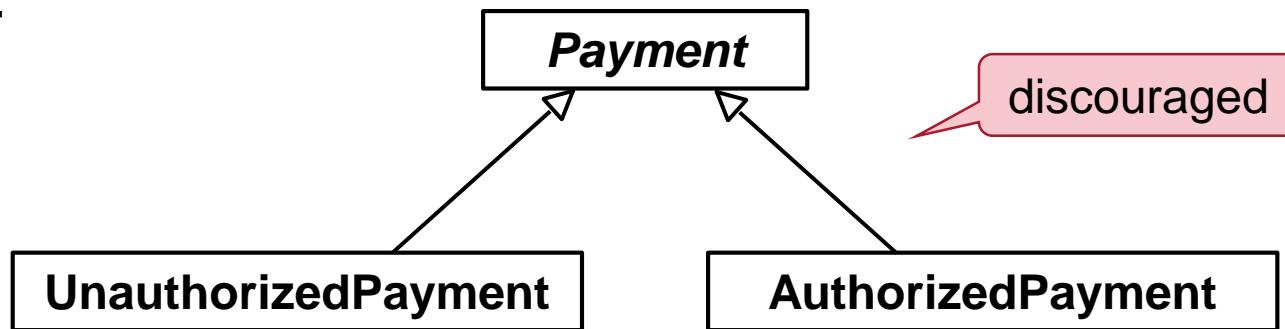
Generalization and Specialization

- Inheritance between super- and sub-classes can be used in domain models
 - Expressed in UML with inheritance generalization arrows (“extends”) 
- Define a subclass of a superclass if
 - The subclass has additional attributes and/or associations of interest
 - The subclass behaves/is operated on/handled/reacted to/manipulated differently than the superclass or other subclasses
- Define a superclass for subclasses if
 - Subclasses represent variations of a similar concept embodied in the superclass
 - All subclasses have the same attribute/association that can be factored out to the superclass
- Abstract classes can be used in domain models as well
 - For generic concepts, where concrete instances can exist only of subclasses
- Note: These domain model constructs do not prescribe implementation choices.

Modeling State Changes

- Do not reflect different states of a particular entity as subclasses of a superclass.

- Example:



- Instead:

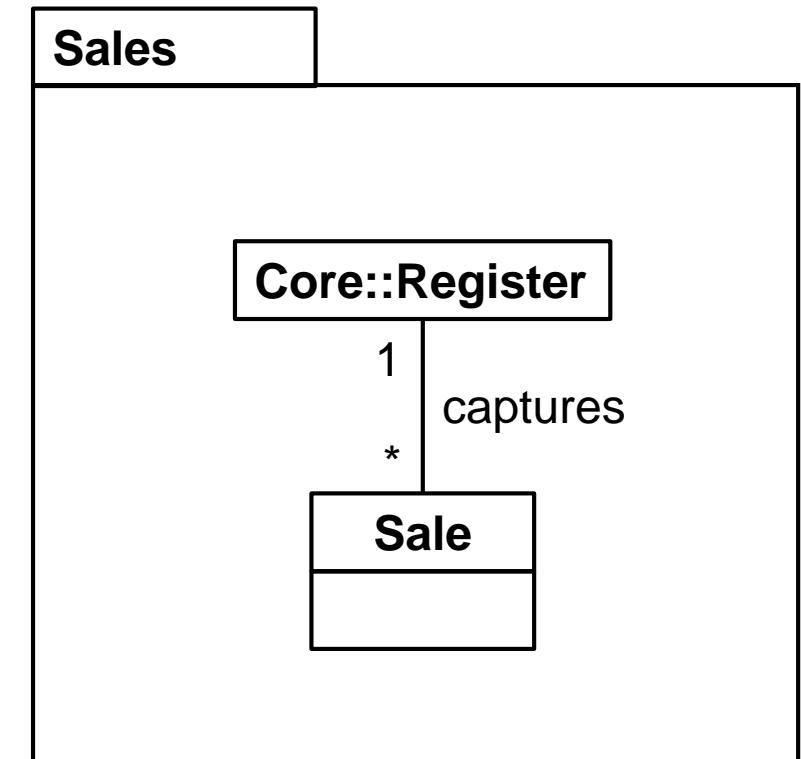
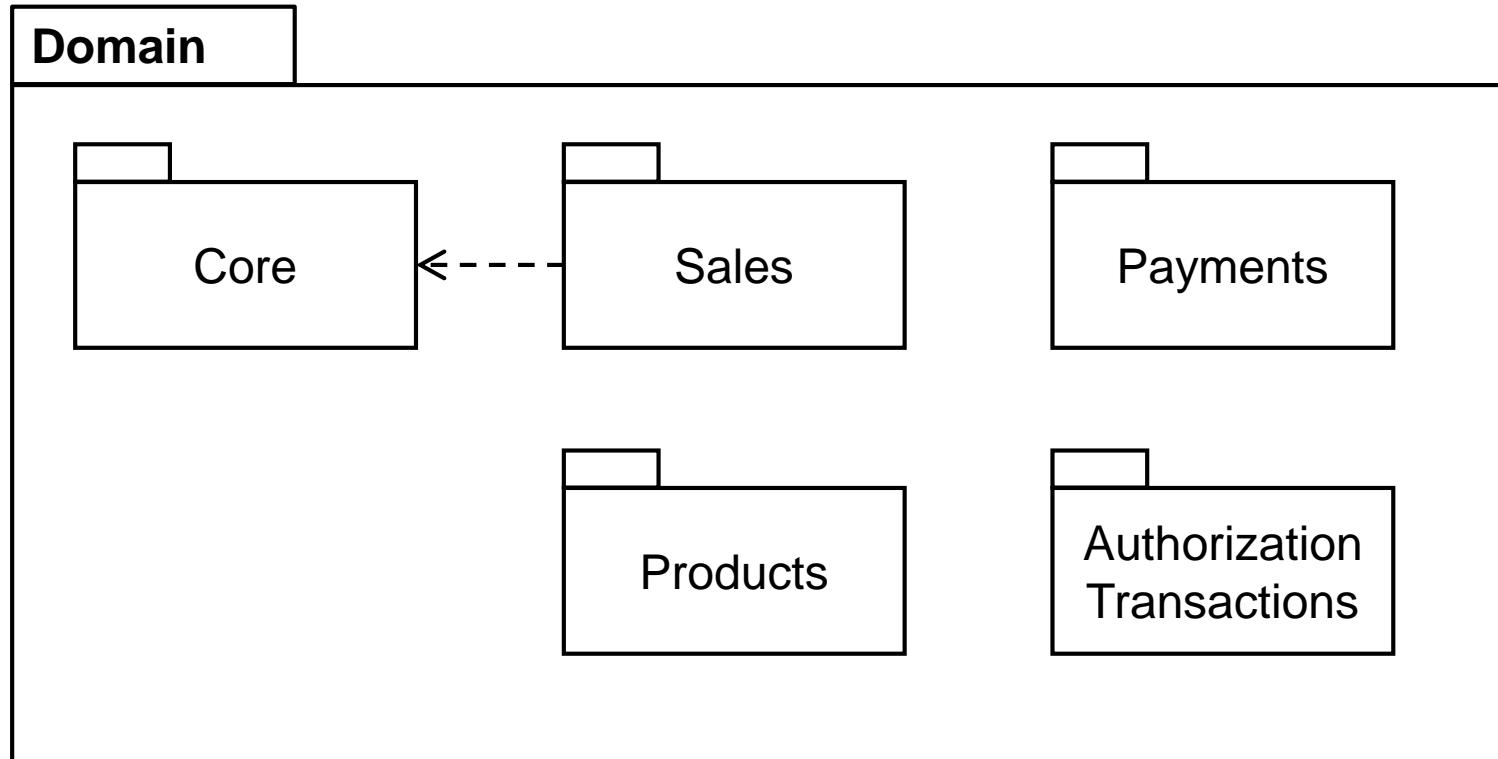
- Make the state explicit in an attribute (if it make sense to encode it in a single one)
- and/or (better): Draw a UML state diagram for the class to show under what circumstances it changes between which states

Partitioning Models with Packages

- A package is a set of classes sharing a similar purpose and namespace
 - e.g. UI, Claims Processing, Health Data...
 - Symbolized in UML by “folder” shape
- Packages can contain classes and other packages
 - A class or package is *owned* by the package in which it is defined
 - but can be *referenced* in other packages (path notation: *PackageName::ElementName*)
- References to classes in other packages create package dependencies
 - Symbolized in UML by dashed arrows (“depends-on”)
- **Guidelines: Place classes in same package that**
 - are in the same subject area, i.e. closely related by concept or purpose
 - are in a class hierarchy together
 - are strongly associated (e.g. by composition)
 - participate in the same use case



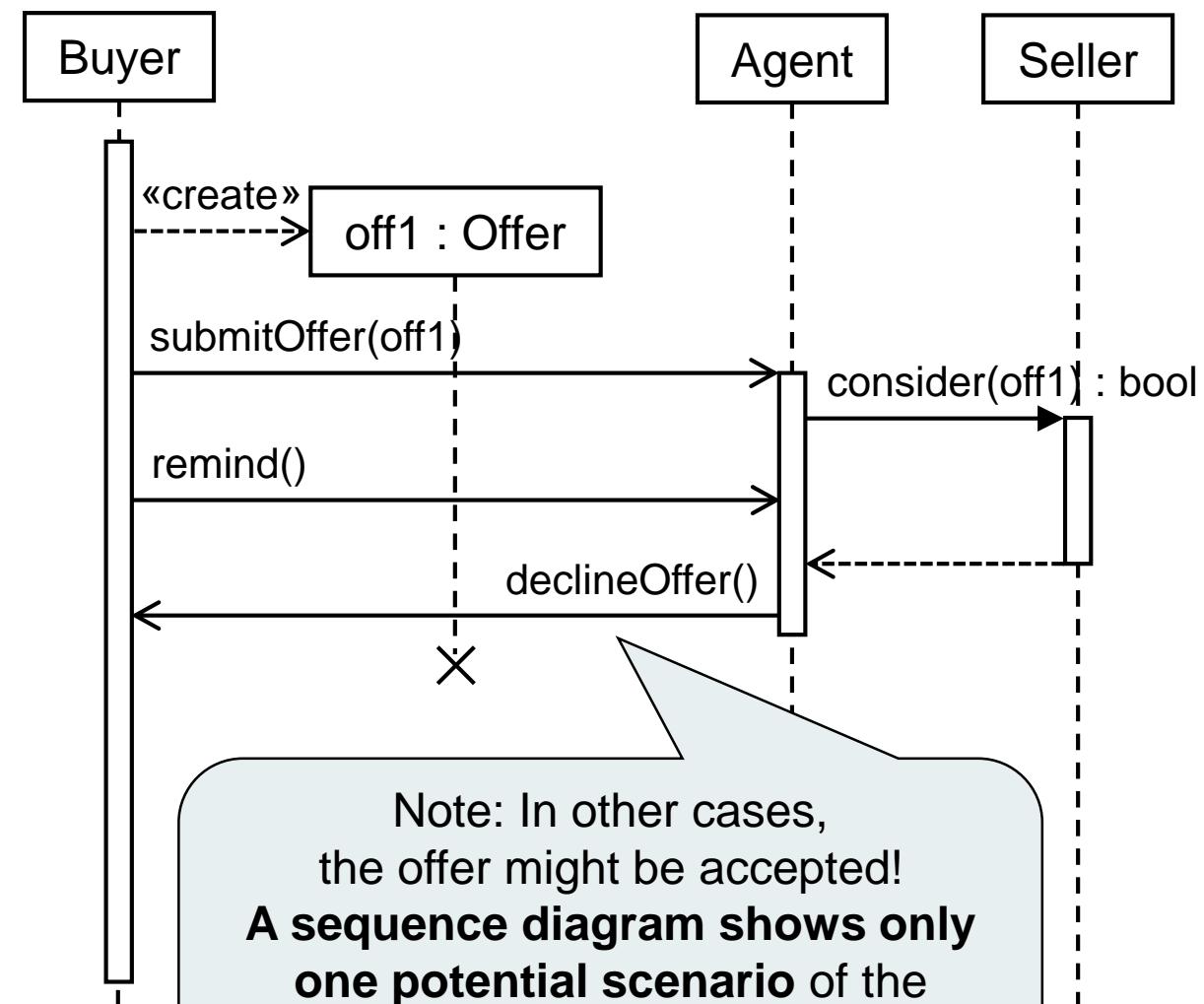
Example: POS Domain Model Packages



Recap: UML Sequence Diagrams

See HBV401G 2015
Lecture 11 for more

- **Participants** can be objects,  packages, components, other actors
- **Synchronous message** →
 - The message sender waits until the message receiver responds to the invocation.
- **Return message** ←
 - Control flow returns after invocation of synchronous message
- **Asynchronous message** →
 - The message sender does not wait for the receiver's response but remains in control after sending.



The System as Part of the Real World

- **Q:** How am I supposed to model just the “real world”, when my system is influencing / shaping / enabling that world?
- **A:** A domain model is not supposed to model the world *without your system*, but just *without knowledge of* your system’s *implementation details*.



The System as Part of the Real World

- No matter if you are
 - developing a system supporting existing processes,
 - building a system that will enable a completely new business model,
 - designing a game that has no real-life equivalent:
- **Model how things are supposed to work once your system is in place**
 - Only if you need to understand a very complex domain, it may help to model how things are currently done first – but don't spend too much effort before moving on to model future vision
- This may mean that
 - your system is a key actor (e.g. matching customers and advertisers)
 - your system is introducing new conceptual classes (e.g. Timeline, Post, Like)
- That's fine – just stay on conceptual level, but don't go to technical level yet
 - No software classes, web pages, database tables etc. in a domain model
 - For the domain model, it shouldn't matter if you'll build a Windows, Web or Android app



Iterative-Incremental Development of Domain Model

- Don't try to create “the one” correct and complete domain model early on
 - It will never be either correct or complete
 - It will cost more than you will gain from it
- Model only what you need to understand
 - To shape the overall architecture
 - To design and implement the features of the current iteration
- Have a relatively broad (but not too broad) domain model first, then refine details as your project progresses
- Work closely together with domain experts.
 - It's your responsibility to understand them, not vice versa!

Recap: Team Assignment 2

- Three of the most important artefacts created in the Elaboration phase are:
 1. **The domain model**
 - Describing the concepts and structures of the application domain your software works with
 2. **The Software Architecture document**
 - Describing the architectural drivers and the architectural decisions made to address them
 3. **The executable architecture**
 - Providing a technical backbone on which you can proceed to implement your business logic
- Producing these artefacts and explaining the considerations that went into them will be your job in Team Assignment 2.

Team Assignment 2: Content

- By **Sun 18 Oct**, submit in Uglá:
 - A **domain model** for your project, containing: (~50% of grade)
 - A **UML class diagram** covering all relevant objects of your application domain (~60%)
 - Which things, actors, concepts etc. are there, and how are they related with each other?
 - A **UML sequence or activity diagram** describing key processes in your application domain (~40%)
 - How do things work in your application domain? How does your software support that / integrate with that?
 - Note: This is not a model of the technical implementation yet, just a model of the real-world concepts and their relationships!
 - An initial **Software Architecture document**, containing: (~50%)
 - An **architectural drivers table**, following the template on slide 25 of lecture 5 (~33%)
 - Describe 3-5 architectural drivers that are relevant for your project (if you identify more (realistic ones), great!)
 - **Technical memos** documenting architectural decisions, following template on slide 30 of lect. 5 (~33%)
 - Describe decisions made to deal with 2-3 of the above-mentioned drivers (if you can describe more, great!)
 - A **UML package diagram** describing the key components of your architecture (~17%)
 - A **UML sequence diagram** describing the control flow between packages of your architecture (~17%)
- On **Thu 22 Oct**, demonstrate and explain your **executable architecture**:
 - Have a skeleton of your system running on your computer to illustrate that your components can exchange data with each other, and explain to your tutor how they do it
 - No need for any business logic or nice user interfaces; just have the technical components wired up

Team Assignment 2: Format

- All artefacts must be produced by all team members together.
- The **domain model** and **Software Architecture** document must
 - be submitted as **PDF documents by Sun 18 Oct in Ugla**
 - contain your team number, the names and kennitölur of all team members
- The code of the executable architecture does not need to be submitted.
- Only the team member who will present should submit the documents.
 - Don't submit multiple versions – we'll just grade the first one we encounter!
- The **demonstration of the executable architecture on Thu 22 Oct**
 - should be given by one representative of the team (a different one for each assignment!)
 - should be focused on the running system on your computer (don't prepare extra slides!)
 - should also refer to the UML diagrams in your software architecture document for support
 - will also involve answering questions about the submitted documents
 - should take around 10 minutes (plus some questions asked by the tutor)
- All team members receive the same grade, with some variation for the presenter



Team Assignment 2: Grading Criteria for Documents

▪ Domain Model

▪ Class diagram

- ✓ provides a clear and comprehensive overview of relevant domain concepts
- ✓ clearly illustrates relationships between domain concepts
- ✓ is syntactically correct UML

▪ Sequence/activity diagram

- ✓ clearly shows relevant dynamics in application domain
- ✓ uses participants found in class diagram
- ✓ is syntactically correct UML

▪ Software Architecture Document

▪ Architectural Drivers Table

- ✓ lists plausible quality requirements
- ✓ Quality scenarios are described precisely
- ✓ Plausible arguments for variability & impact

▪ Technical Memos

- ✓ describe plausible architectural decisions (in context of architectural drivers)
- ✓ Solution & Motivation sections are plausible
- ✓ **Package diagram** illustrates partitioning of high-level components of system clearly
- ✓ **Sequence diagram** illustrates main control flow between components clearly
- ✓ Architecture described by package and sequence diagrams makes sense and reflects decisions in technical memos



Team Assignment 2: Grading Criteria for Presentation

■ Executable Prototype

- ✓ works, can receive, process and output data
- ✓ Presenter can explain which components the prototype is composed of & how they interact
- ✓ Understanding of MVC pattern is reflected in prototype implementation and demonstration
- ✓ Any client-side components are structured clearly

■ Documents

- ✓ Presenter can use architectural diagrams to explain architecture
- ✓ Presenter can justify architectural decisions
- ✓ Presenter can correct any issues in submitted documents ad hoc

■ Overall

- ✓ Presenter shows initiative, explains things clearly, shows understanding of the approach



Gangi þér vel!

book@hi.is



HÁSKÓLI ÍSLANDS
VERKFRÆÐI- OG NÁTTÚRVÍSINDASVIÐ

IÐNAÐARVERKFRÆÐI-, VÉLAVERKFRÆÐI-
OG TÖLVUNARFRÆÐIDEILD





Hugbúnaðarverkefni 1 / Software Project 1

8. Persistence Layer

HBV501G – Fall 2015

Matthias Book



HÁSKÓLI ÍSLANDS
VERKFRÆÐI- OG NÁTTÚRVÍSINDASVIÐ
ÍÐNAÐARVERKFRÆÐI-, VÉLAVERKFRÆÐI-
OG TÖLVUNARFRÆÐIDEILD

Recap: Team Assignment 2

- Three of the most important artefacts created in the Elaboration phase are:
 1. **The domain model**
 - Describing the concepts and structures of the application domain your software works with
 2. **The Software Architecture document**
 - Describing the architectural drivers and the architectural decisions made to address them
 3. **The executable architecture**
 - Providing a technical backbone on which you can proceed to implement your business logic
- Producing these artefacts and explaining the considerations that went into them will be your job in Team Assignment 2.

Team Assignment 2: Content

- By **Sun 25 Oct**, submit in Uglá:
 - A **domain model** for your project, containing: (~50% of grade)
 - A **UML class diagram** covering all relevant objects of your application domain (~60%)
 - Which things, actors, concepts etc. are there, and how are they related with each other?
 - A **UML sequence or activity diagram** describing key processes in your application domain (~40%)
 - How do things work in your application domain? How does your software support that / integrate with that?
 - Note: This is not a model of the technical implementation yet, just a model of the real-world concepts and their relationships!
 - An initial **Software Architecture document**, containing: (~50%)
 - An **architectural drivers table**, following the template on slide 25 of lecture 5 (~33%)
 - Describe 3-5 architectural drivers that are relevant for your project (if you identify more (realistic ones), great!)
 - **Technical memos** documenting architectural decisions, following template on slide 30 of lect. 5 (~33%)
 - Describe decisions made to deal with 2-3 of the above-mentioned drivers (if you can describe more, great!)
 - A **UML package diagram** describing the key components of your architecture (~17%)
 - A **UML sequence diagram** describing the control flow between packages of your architecture (~17%)
- On **Thu 29 Oct**, demonstrate and explain your **executable architecture**:
 - Have a skeleton of your system running on your computer to illustrate that your components can exchange data with each other, and explain to your tutor how they do it
 - No need for any business logic or nice user interfaces; just have the technical components wired up

Team Assignment 2: Format

- All artefacts must be produced by all team members together.
- The **domain model** and **Software Architecture** document must
 - be submitted as **PDF documents by Sun 25 Oct in Ugla**
 - contain your team number, the names and kennitölur of all team members
- The code of the executable architecture does not need to be submitted.
- Only the team member who will present should submit the documents.
 - Don't submit multiple versions – we'll just grade the first one we encounter!
- The **demonstration of the executable architecture on Thu 29 Oct**
 - should be given by one representative of the team (a different one for each assignment!)
 - should be focused on the running system on your computer (don't prepare extra slides!)
 - should also refer to the UML diagrams in your software architecture document for support
 - will also involve answering questions about the submitted documents
 - should take around 10 minutes (plus some questions asked by the tutor)
- All team members receive the same grade, with some variation for the presenter



Team Assignment 2: Grading Criteria for Documents

▪ Domain Model

▪ Class diagram

- ✓ provides a clear and comprehensive overview of relevant domain concepts
- ✓ clearly illustrates relationships between domain concepts
- ✓ **does not contain technical details**
- ✓ is syntactically correct UML

▪ Sequence/activity diagram

- ✓ clearly shows relevant dynamics in application domain
- ✓ uses **active** participants found in class diagram
- ✓ is syntactically correct UML

▪ Software Architecture Document

▪ Architectural Drivers Table

- ✓ lists plausible quality requirements
- ✓ Quality scenarios are described precisely
- ✓ Plausible arguments for variability & impact

▪ Technical Memos

- ✓ describe plausible architectural decisions (in context of architectural drivers)
- ✓ Solution & Motivation sections are plausible
- ✓ **Package diagram** illustrates partitioning of high-level components of system clearly
- ✓ **Sequence diagram** illustrates main control flow between components clearly
- ✓ Architecture described by package and sequence diagrams makes sense and reflects decisions in technical memos



Team Assignment 2: Grading Criteria for Presentation

■ Executable Prototype

- ✓ works, can receive, process and output data
- ✓ Presenter can explain which components the prototype is composed of & how they interact
- ✓ Understanding of MVC pattern is reflected in prototype implementation and demonstration
- ✓ Extra points for including and demonstrating database access
- ✓ Any client-side components are structured clearly

■ Documents

- ✓ Presenter can use architectural diagrams to explain architecture
- ✓ Presenter can justify architectural decisions
- ✓ Presenter can correct any issues in submitted documents ad hoc

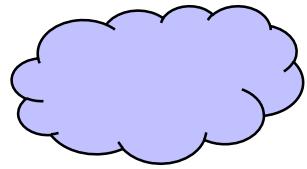
■ Overall

- ✓ Presenter shows initiative, explains things clearly, shows understanding of the approach

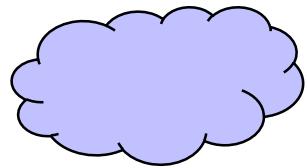


Recap: Inception and Elaboration Artifacts

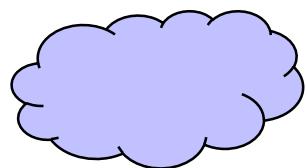
Business Domain



Business Requirements



Quality Attributes



Business Rules

Requirements



Vision and Scope

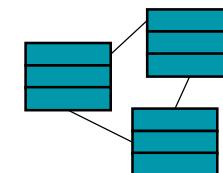


Supplementary Specification

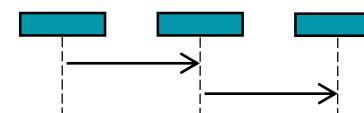


Use Cases

Domain Model



Structural Diagrams

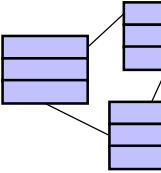


Interaction Diagrams

Architecture

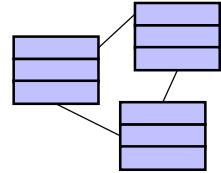


Software Architecture

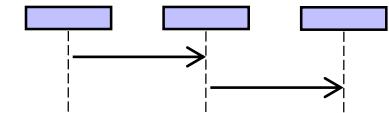


Package Diagrams

Design Model



Structural Diagrams



Interaction Diagrams

Domain Modeling Guidelines

- Focus on **what the purpose** of system is, **not how** it will be engineered
 - Imagine explaining to a non-technical person what they will be able to do with the system
 - Which concepts, roles, terms do you need to explain this? → Structural aspects of domain model
 - How do all these work together to accomplish the purpose? → Dynamic aspects of domain model
- **Domain models** do not involve methods, web pages, controllers, databases...
 - These go into the technically focused **design models** (after the domain is understood)
 - Methods, parameters, controllers etc. go into the class diagram of the OO design model
 - Web pages go into the navigational model of the web app, often coupled with UI mockups
 - Database structures go into an entity-relationship model, or are implied by the design model
- The design model is not a simple refinement of the domain model where you just add some technical detail.
 - It is an **independent model that describes an effective technical solution** of your requirements, **informed by the understanding of the application domain** that you gained from building the domain model.



UML Sequence Diagrams in Domain vs. Design Models

in Domain Models

- Sequence diagrams show how the active **components of the application domain collaborate** in the real world
 - Arrows are labeled with **actions or messages**, most intuitively from the actor's (i.e. arrow source's) perspective
 - Quite natural that interactions are **asynchronous**, and actors do not expect immediate responses before proceeding
- Classes representing **passive data entities** are usually **not shown**
 - unless they are key to understanding the interaction
- Illustration of **actors' behavior**
 - according to **use case**

not an exact prescription for

in Design Models

- Sequence diagrams show how the **classes of the system interact through calling each others' methods**
 - Arrows are labeled with **names of methods** being called (i.e. methods defined at the arrow's **destination**)
 - Typically, methods calls are **synchronous**, and an actor is waiting for a response before proceeding with something else
- Diagrams show **how data entities are processed** by active entities
 - e.g. their creation, destruction, and passing along as method parameters
- Specification of **classes' collaboration**
 - in an efficient **technical implementation**

Java Persistence API

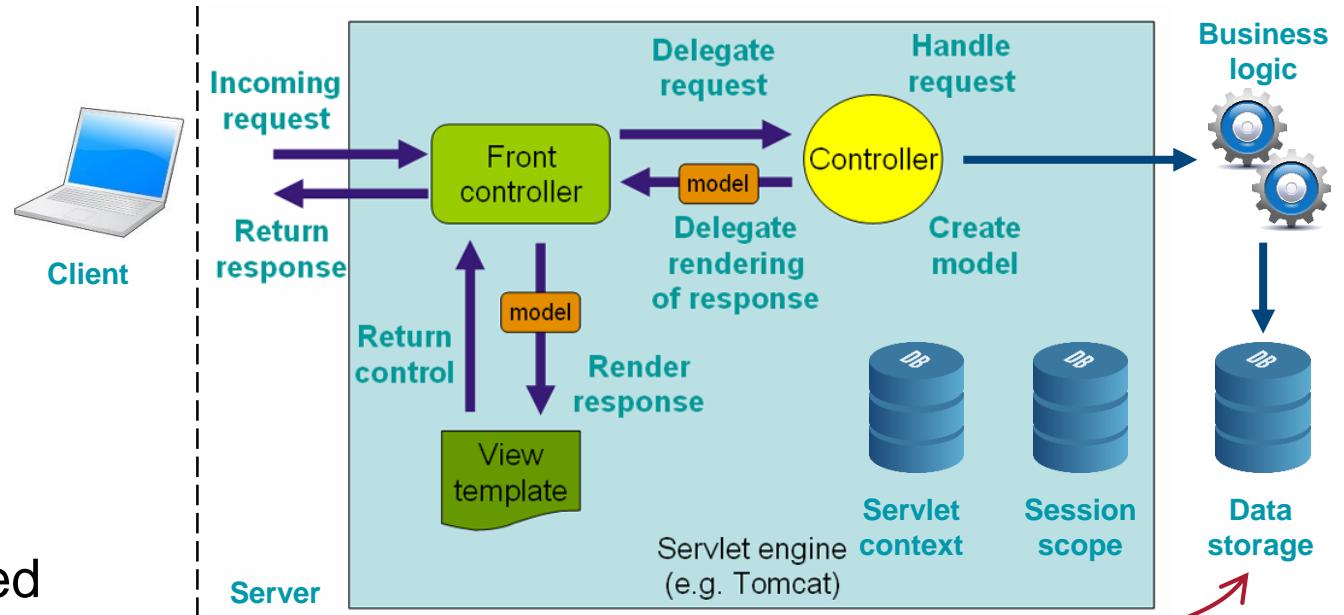
see also:

- Williams: Professional Java for Web Applications, Ch. 19-22, 24
- <http://docs.spring.io/spring-data/jpa/docs/current/reference/html/>



Recap: Persistent Data Storage

- Some data is not suitable for storage in the servlet engine's scopes, e.g.
 - data that shall be stored even when the server is down
 - data that is too large to be kept in memory
 - data that is most efficiently stored and retrieved in a non-object-oriented structure (e.g. relational data)
 - data that is retrieved from external sources
- For these purposes, a database or other data sources can be accessed from the business logic
 - Persistence frameworks can help with the mapping of objects to database structures



Recap: Object-Relational Mapping (ORM)

- All our object-oriented data structures exist in memory at run-time.
- However, we also need data structures outside our program...
 - to preserve information while the system is not running
 - to work with data structures that are larger than available memory
 - to exchange information with other (remote) systems
- There are a number of solutions for this
 - e.g. databases, XML, JSON, binary files...
- Unfortunately, many of them are not (or not fully) object-oriented :-(
- **Challenge: Object-Relational Mapping**
 - Transforming object-oriented data structures into a non-object-oriented persistent format



Motivation: Database Access without ORM

```
public Product getProduct(long id) throws SQLException {
```

Need to identify objects through their primary keys *and* OO references

```
    try (Connection c = this.getConnection();  
         PreparedStatement s = c.prepareStatement(  
             "SELECT * FROM dbo.Product WHERE productId = ?")) {
```

Need to deal with connection technicalities

```
        s.setLong(1, id);
```

```
        try (ResultSet r = s.executeQuery()) {
```

Need to map data types, object and table structures

```
            if (!r.next()) return null;
```

```
            Product p = new Product(id);
```

```
            p.setName(r.getString("Name"));
```

```
            p.setDatePosted(r.getObject("DatePosted", Instant.class));
```

Need to pick apart query results and piece together objects field by field

```
            p.setPrice(r.getDouble("Price"));
```

```
            // ...mapping a dozen more attributes...
```

Need to deal with linked entities (efficient retrieval, cascading deletions etc.)

```
            return p;
```

```
}
```

```
}
```



Need to maintain similarly complex code for creating and updating entities

- tedious
- ideal breeding ground for bugs

Database Access Implementation Options

- Previously, you may likely have done this manually
 - Connect to database
 - Formulate SQL statements
 - Map data back and forth between objects and relational structures
- In this class, we will see how to abstract from most of the technical steps
 - Just let the Java Persistence API (JPA) know which objects you want to be persistent...
 - ...and what information you want to retrieve from the database
 - Necessary database operations will be executed automatically
- Pros and Cons
 - Easy to use standard database operations without a lot of technical overhead
 - Not as visible what's going on behind the scenes
 - A manual implementation that does exactly what you need may be more efficient



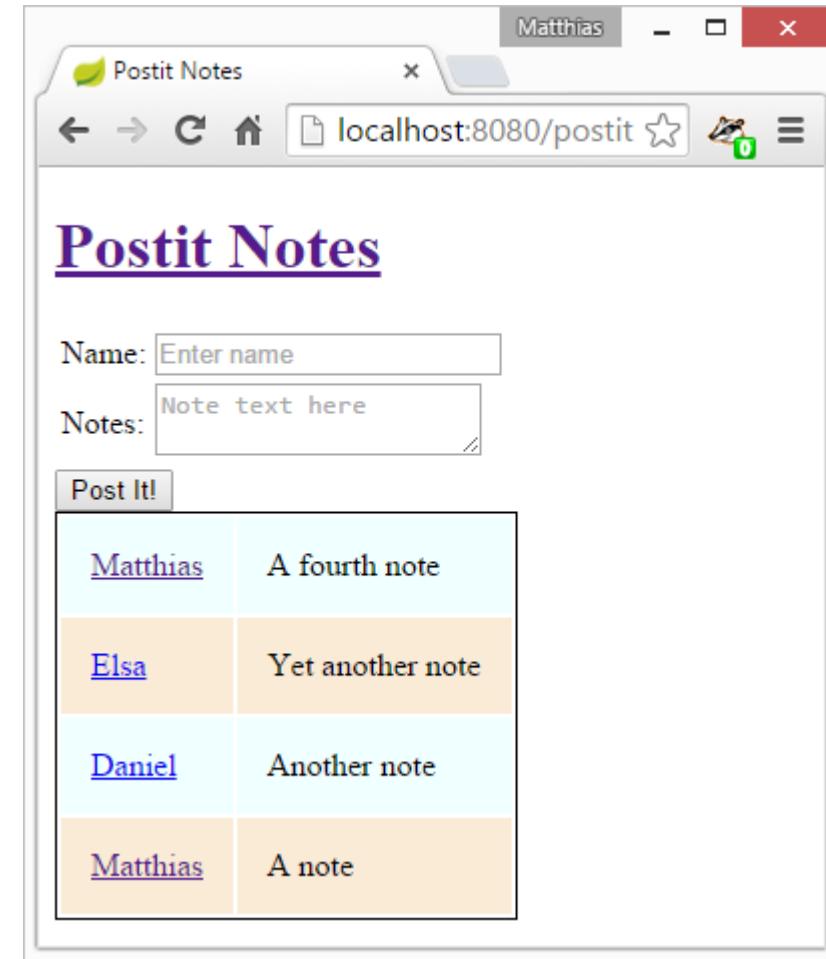
Technology Stack

- Structured Query Language (SQL) ($\rightarrow TÖL303G$)
 - Language for formulating queries against relational databases
- Java Database Connectivity (JDBC) ($\rightarrow Williams, Ch. 19$)
 - API for creating connections to a variety of databases (through appropriate JDBC drivers) and sending SQL queries to them
- Object-Relational Mapper (ORM) ($\rightarrow Williams, Ch. 19$)
 - Framework taking care of the mapping of object structures to relational structures, formulating suitable queries etc. Popular ORMs: Hibernate, MyBatis, EclipseLink...
- Java Persistence API (JPA) ($\rightarrow Williams, Ch. 20, 21$)
 - Additional layer abstracting from any particular ORM implementation
- Spring Data JPA ($\rightarrow Williams, Ch. 22, 24$)
 - Extension of JPA providing convenient methods for query formulation (among other things)



The Skeleton App's PostIt Demo

- The skeleton app now includes a small “PostIt” demo.
 - See “Persistence Layer” slides in Verkefni folder for setup instructions
- After starting your database server and running Application.main as usual, you can play with a demo of the persistence layer at <http://localhost:8080/postit>
 - Note that data you enter here remains persistent even after you restart the web application!



Postit View, Part 1: Data Entry Form

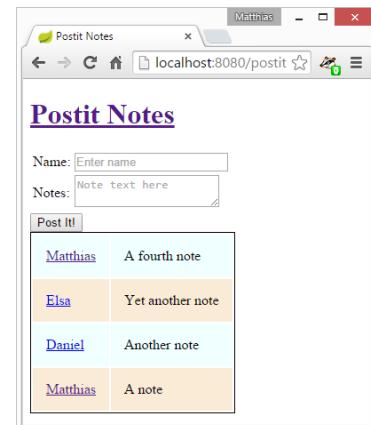
(PostitNotes.jsp)

```
<!DOCTYPE html>
<%@ taglib prefix="spring" uri="http://www.springframework.org/tags"%>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="sf" uri="http://www.springframework.org/tags/form" %>
<html lang="en">
  <head><!-- ... --></head>
  <body>
    <sf:form method="POST" commandName="postitNote" action="/postit">
      <table><tr>
        <td>Name:</td><td><sf:input path="name" type="text" placeholder="Enter name"/></td>
      </tr><tr>
        <td>Notes:</td><td><sf:textarea path="note" type="text" placeholder="Note text here"/></td>
      </tr></table>
      <input type="submit" value="Post It!"/>
    </sf:form>
    <!-- ... -->
```

Definitions of custom tags to be used in construction of HTML code (“smart” tags interpreted by server)

Name of @ModelAttribute in which we expect the data

Name of entity attribute in which to store the input



Request Handling, Part 1a: Submitting a New Postit (PostitNoteController)

```
@Controller  
public class PostitNoteController {  
    PostitNoteService postitNoteService;  
  
    @Autowired  
    public PostitNoteController(PostitNoteService postitNoteService) {  
        this.postitNoteService = postitNoteService;  
    }  
  
    @RequestMapping(value = "/postit", method = RequestMethod.POST)  
    public String postitNoteViewPost(@ModelAttribute("postitNote") PostitNote postitNote,  
                                    Model model) {  
        // ...
```

Obtain an instance of the service providing required business logic. Note:

- Dependency injection: Controller does not instantiate the class it depends on, but expects to receive it from outside
- `@Autowired` indicates that Spring takes care of instantiating and providing (injecting) the service, so we don't have to worry about when and where to do this



Persistent Data Entity (PostitNote)

```
@Entity  
@Table(name = "postitnote")  
public class PostitNote {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.IDENTITY)  
    private Long id;  
  
    private String name;  
    private String note;  
  
    public PostitNote() {  
    }  
}
```

Indicates that O/R mapping shall be performed for instances of this class

Optional: Name of DB table to use for entity (default: class name)

Indicates that the following attribute shall contain the primary key, and that it shall be populated with unique values

Required in order to create the object populated by the web form

```
public PostitNote(  
    String name, String note) {  
    this.name = name;  
    this.note = note;  
}  
  
public Long getId() {  
    return id;  
}  
  
public void setId(Long id) {  
    this.id = id;  
}  
  
// [...other getters & setters...]
```

Note: Initializing the id attribute is being taken care of automatically due to @GeneratedValue

All attributes with getters and setters will turn into DB table columns

Data Type Mappings

| Java types | SQL types (one of listed or equivalent, depending on DB) |
|--|--|
| short, Short | SMALLINT, INTEGER, BIGINT |
| int, Integer | INTEGER, BIGINT |
| long, Long, BigInteger | BIGINT |
| float, Float, double, Double, BigDecimal | DECIMAL |
| byte, Byte | BINARY, SMALLINT, INTEGER, BIGINT |
| char, Character | CHAR, VARCHAR, BINARY, SMALLINT, INTEGER, BIGINT |
| boolean, Boolean | BOOLEAN, BIT, SMALLINT, INTEGER, BIGINT, CHAR, VARCHAR |
| byte[], Byte[] | BINARY, VARBINARY |
| char[], Character[], String | CHAR, VARCHAR, BINARY, VARBINARY |
| Date, Calendar (with @Temporal annotation) | DATE, TIME, DATETIME |
| enum | SMALLINT, INTEGER, BIGINT, CHAR, VARCHAR |
| Serializable | VARBINARY (object stored in serialized form) |



Request Handling, Part 1b: Submitting a New Postit (PostitNoteController)

```
@RequestMapping(value = "/postit", method = RequestMethod.POST)
public String postitNoteViewPost(@ModelAttribute("postitNote") PostitNote postitNote,
                                 Model model) {
    postitNoteService.save(postitNote); Doing business with the postitNote received in the request (here: saving it)
    model.addAttribute("postitNote", new PostitNote()); Preparing a new, empty postitNote to populate the form with in the response
    model.addAttribute("postitNotes", postitNoteService.findAllReverseOrder());
    return "postitnotes/PostitNotes";
}
```



Declaration of Business Service (PostitNoteService)

```
import project.persistence.entities.PostitNote;  
import java.util.List;  
  
public interface PostitNoteService {  
  
    PostitNote save(PostitNote postitNote);  
    void delete(PostitNote postitNote);  
    List<PostitNote> findAll();  
    List<PostitNote> findAllReverseOrder();  
    PostitNote findOne(Long id);  
    List<PostitNote> findByName(String name);  
}
```

Declaration of various functionalities
that our business logic offers



Implementation of Postit Handling Services (PostitNoteServiceImplementation)

```
@Service
public class PostitNoteServiceImplementation
    implements PostitNoteService {
    PostitNoteRepository repository;

    @Autowired
    public PostitNoteServiceImplementation(
        PostitNoteRepository repository) {
        this.repository = repository;
    }

    @Override
    public PostitNote save(PostitNote postitNote) {
        return repository.save(postitNote);
    }

    @Override
    public void delete(PostitNote postitNote) {
        repository.delete(postitNote);
    }

    @Override
    public List<PostitNote> findAll() {
        return repository.findAll();
    }

    @Override
    public PostitNote findOne(Long id) {
        return repository.findOne(id);
    }

    @Override
    public List<PostitNote> findByName(String name) {
        return repository.findByName(name);
    }

    @Override
    public List<PostitNote> findAllReverseOrder() {
        List<PostitNote> postitNotes =
            repository.findAll();
        Collections.reverse(postitNotes);
        return postitNotes;
    }
}
```

Indicates this is a business logic implementation

Obtain data repository through dependency injection

Business logic implementation, heavily relying on repository in this case

This line does everything we had to code manually on slide 13!



Configuration of Data Repository (PostitNoteRepository)

```
public interface PostitNoteRepository extends JpaRepository<PostitNote, Long> {  
    PostitNote save(PostitNote postitNote);  
    void delete(PostitNote postitNote);  
    List<PostitNote> findAll();  
  
    @Query(value = "SELECT p FROM PostitNote p where length(p.name) >= 3")  
    List<PostitNote> findAllWithNameLongerThan3Chars();  
  
    List<PostitNote> findAllByIdDesc();  
    PostitNote findOne(Long id);  
    List<PostitNote> findByName(String name);  
}
```

Interface declaration
determines what functionality
the repository should offer

If you need particular kinds of queries,
you can

- define many typical ones by using appropriate keywords in your method names (see following slides)
- or use the `@Query` annotation to create individual queries

Note: Implementation of this interface will automatically be provided by the Java Persistence API (JPA)



Keywords in Query Method Names

| Keyword | Sample | Query snippet |
|--------------------|--|--|
| And | findByLastnameAndFirstname | ... where x.lastname = ?1 and x.firstname = ?2 |
| Or | findByLastnameOrFirstname | ... where x.lastname = ?1 or x.firstname = ?2 |
| Is, Equals | findByFirstname, findByFirstnameIs, findByFirstnameEquals | ... where x.firstname = 1? |
| Between | findByStartDateBetween | ... where x.startDate between 1? and ?2 |
| LessThan | findByAgeLessThan | ... where x.age < ?1 |
| LessThanEqual | findByAgeLessThanEqual | ... where x.age <= ?1 |
| GreaterThan | findByAgeGreaterThan | ... where x.age > ?1 |
| GreaterThanOrEqual | findByAgeGreaterThanOrEqual | ... where x.age >= ?1 |
| After | findByStartDateAfter | ... where x.startDate > ?1 |
| Before | findByStartDateBefore | ... where x.startDate < ?1 |
| IsNull | findByAgeIsNull | ... where x.age is null |
| IsNotNull, NotNull | findByAge(Is)NotNull | ... where x.age not null |



Keywords in Query Method Names

| Keyword | Sample | Query snippet |
|--------------|-------------------------------------|--|
| Like | findByFirstnameLike | ... where x.firstname like ?1 |
| NotLike | findByFirstnameNotLike | ... where x.firstname not like ?1 |
| StartingWith | findByFirstnameStartingWith | ... where x.firstname like ?1 (parameter bound with appended %) |
| EndingWith | findByFirstnameEndingWith | ... where x.firstname like ?1 (parameter bound with prepended %) |
| Containing | findByFirstnameContaining | ... where x.firstname like ?1 (parameter bound wrapped in %) |
| OrderBy | findByAgeOrderByLastnameDesc | ... where x.age = ?1 order by x.lastname desc |
| Not | findByLastnameNot | ... where x.lastname <> ?1 |
| In | findByAgeIn(Collection<Age> ages) | ... where x.age in ?1 |
| NotIn | findByAgeNotIn(Collection<Age> age) | ... where x.age not in ?1 |
| True | findByActiveTrue() | ... where x.active = true |
| False | findByActiveFalse() | ... where x.active = false |
| IgnoreCase | findByFirstnameIgnoreCase | ... where UPPER(x.firstname) = UPPER(?1) |



Main Class (Application)

Let JPA create implementations of
repository interfaces automatically

```
@SpringBootApplication
@EnableJpaRepositories
public class Application extends SpringBootServletInitializer {

    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder applicationBuilder) {
        return applicationBuilder.sources(Application.class);
    }

    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }
}
```



Recap: Request Handling, Part 1b: Submitting New Postit (PostitNoteController)

```
@RequestMapping(value = "/postit", method = RequestMethod.POST)
public String postitNoteViewPost(@ModelAttribute("postitNote") PostitNote postitNote,
                                 Model model) {

    postitNoteService.save(postitNote);

    model.addAttribute("postitNote", new PostitNote());

    model.addAttribute("postitNotes", postitNoteService.findAllReverseOrder());

    return "postitnotes/PostitNotes";

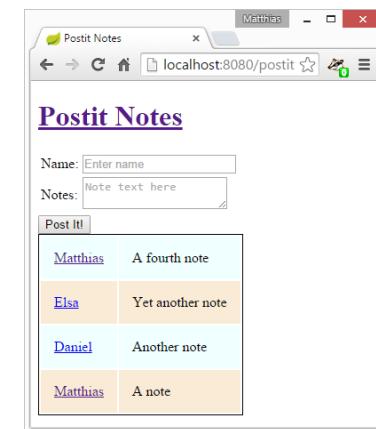
}
```



Postit View, Part 2: Postit List (PostitNotes.jsp)

```
<c:choose>
  <c:when test="${not empty postitNotes}">
    <table class="notes">
      <c:forEach var="postit" items="${postitNotes}">
        <tr>
          <td><a href="/postit/${postit.name}">${postit.name}</a></td>
          <td>${postit.note}</td>
        </tr>
      </c:forEach>
    </table>
  </c:when>
  <c:otherwise>
    <h3>No notes!</h3>
  </c:otherwise>
</c:choose>
```

Control tags evaluated on server
at time of HTML construction



Loop through postitNotes provided in Model, make each available as postit and display its name (as link) and note attributes

Display message instead of table if postitNotes is an empty List

Request Handling, Part 2: Fetching Lists of PostIts (PostitNoteController)

```
@RequestMapping(value = "/postit/{name}", method = RequestMethod.GET)
public String postitNoteGetNotesFromName(@PathVariable String name,
                                         Model model) {
    model.addAttribute("postitNotes", postitNoteService.findByName(name));
    model.addAttribute("postitNote", new PostitNote());
    return "postitnotes/PostitNotes";
}
```

If the URI contains a path variable, retrieve all Postits with that name

```
@RequestMapping(value = "/postit", method = RequestMethod.GET)
public String postitNoteViewGet(Model model) {
    model.addAttribute("postitNote", new PostitNote());
    model.addAttribute("postitNotes", postitNoteService.findAllReverseOrder());
    return "postitnotes/PostitNotes";
}
```

Otherwise, retrieve all Postits in reverse order



Configuring Database Access (application.properties)

Example shown for PostgreSQL; adapt accordingly for other database systems

[...]

How to find the database:

[protocol]:[driver]://[host]:[port]/[dbname]

spring.datasource.url=jdbc:postgresql://localhost:5432/HBV

spring.datasource.username=[your DB username]

spring.datasource.password=[your DB password]

spring.datasource.driver-class-name=org.postgresql.Driver

Authentication information

Driver for database connection

spring.jpa.hibernate.ddl-auto=update

Let JPA take care of creating the database tables and updating the schema when we change the structure of our data entity classes. Note:

- Structural changes may corrupt existing data
- If it seems JPA can't keep up with your changes, try running the application *once* with this parameter set to *create* instead of *update*
- For prototyping only – in a production environment, you'll want to do this manually

Project Structure

■ Controller Layer

- Request handlers

■ Persistence Layer

- Data entities
- Data repositories

■ Business Layer

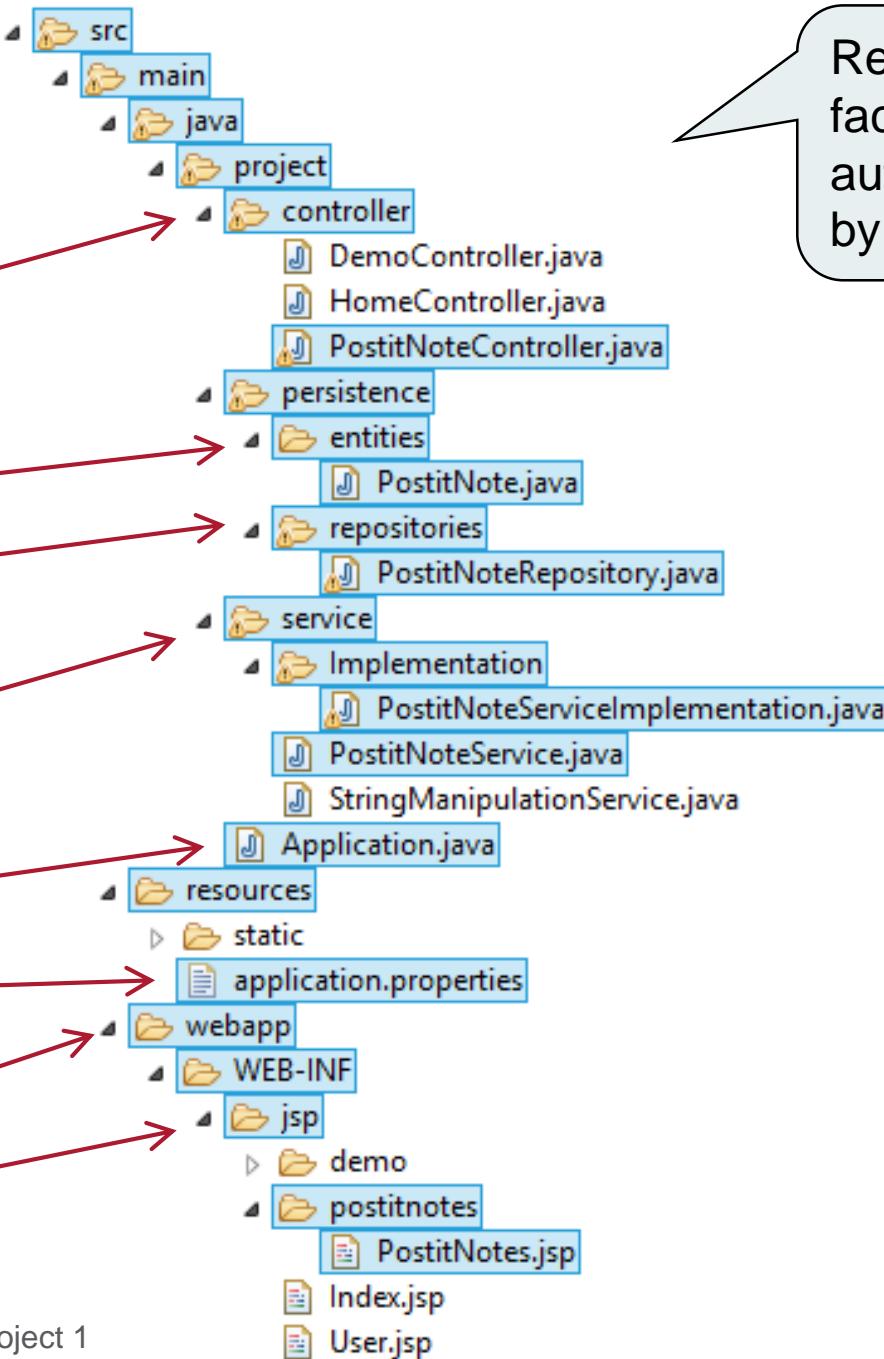
- Business logic classes

■ Configuration

- Spring Boot main class
- Configuration file

■ View Layer

- Static web content
- JavaServer Pages



Required structure to facilitate discovery and autowiring of the classes by the Spring framework

Beyond the Skeleton App: Mapping Complex Data Types

```
@Entity  
public class PostitNote {  
    @Id  
    @GeneratedValue(strategy =  
        GenerationType.IDENTITY)  
    private Long id;  
    private String name;  
    private String note;  
  
    public String getName() { return name; }  
    public void setName(String name) {  
        this.name = name; }  
    // [...other getters & setters...]  
}
```

- Recap: Automatic mappings to SQL types exist for common Java types:
 - Primitive types (int, float, boolean, ...)
 - Primitive type wrappers (Integer, ...)
 - Strings and arrays (char, byte[], ...)
 - Date, Time, Calendar
 - Enumerated properties (enum)
 - Any class implementing Serializable
- What about complex attributes?
 - Complex properties of the entity, e.g. a structured PhoneNumber
 - Embedded properties
 - References to other entities, e.g. the LineItems of a Receipt
 - Entity relationships



Embedded Properties

```
@Entity  
public class Person {  
    private long id;  
    private String firstName;  
    private Address address;  
  
    @Id  
    @GeneratedValue(strategy =  
        GenerationType.IDENTITY)  
    public long getID() {...}  
    public void setID(long id) {...}  
  
    @Embedded  
    public Address getAddress() {...}  
    public void setAddress(Address addr) {...}  
    // ...  
}
```

Indicates the fields of Address shall be incorporated directly into the Person table

```
@Embeddable  
public class Address {  
    private String street;  
    private String city;  
    private PhoneNo phoneNo;  
    public String getStreet() {...}  
    public void setStreet(String street) {...}  
    public String getStreet() {...}  
    public void setStreet(String street) {...}  
  
    @Embedded  
    public PhoneNo getPhoneNo() {...}  
    public void setPhoneNo(PhoneNo phNo) {...}  
    // ...  
}
```

Indicates that Address data is not stored in a table of its own, but embedded into another entity's table

Note: Therefore, no @Id here!

Regular POJO

Embeddable properties can contain embedded properties themselves

Entity Relationships: Bidirectionally Navigable One-to-Many Relationship

```
@Entity  
public class Receipt {  
    private long id;  
    private String buyer;  
    private Set<LineItem> lineItems =  
        new HashSet<>();  
  
    @Id  
    @ColumnName(name = "ReceiptId")  
    @GeneratedValue(strategy =  
        GenerationType.IDENTITY)  
    public long getId() {...}  
    public void setId(long id) {...}
```

Name of related entity's attribute that references this entity

Don't retrieve related entities from DB until someone accesses them in Java

```
        public String getBuyer() {...}  
        public void setBuyer(String buyer) {...}  
  
        @OneToMany(mappedBy = "receipt",  
            fetch = FetchType.LAZY,  
            cascade = CascadeType.ALL,  
            orphanRemoval = true)  
        public Set<LineItem> getLineItems()  
            {...}  
        public void setLineItems(  
            Set<LineItem> lineItems) {...}
```

Action on related entities when this one is deleted

Object-oriented declaration of the One-to-Many relation: A reference to a Set of the “many” objects



Entity Relationships: Bidirectionally Navigable Many-to-One Relationship

```
@Entity  
@Table(name = "Receipt_LineItem")  
public class LineItem {  
    private long id;  
    private String title;  
    private double price;  
    private Receipt receipt;  
  
    @Id  
    @ColumnName(name = "LineItemId")  
    @GeneratedValue(strategy =  
        GenerationType.IDENTITY)  
    public long getId() {...}  
    public void setId(long id) {...}
```

Reference to related entity

Retrieve related entity from DB immediately

There must be a related entity

@ManyToOne(fetch = FetchType.EAGER,
optional = false)

@JoinColumn(name = "ReceiptId")

Object-oriented declaration of the Many-to-One relation: A reference to the “one” object

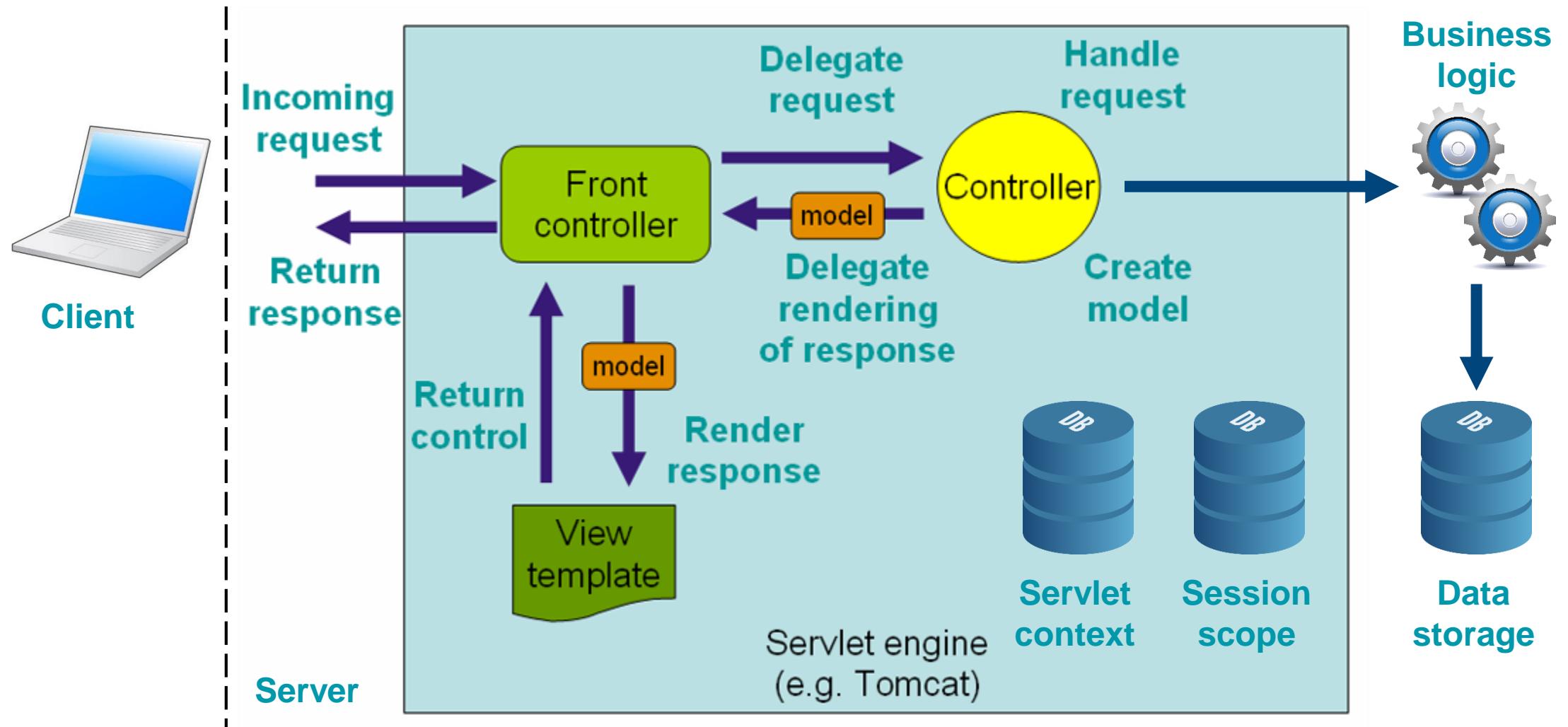
public String getTitle() {...}
public void setTitle(String title) {...}
public double getPrice() {...}
public void setPrice(double price) {...}

public Receipt getReceipt() {...}
public void setReceipt(Receipt receipt) {...}

}

Name of column in this table that will contain the other table's primary key

Summary





Hugbúnaðarverkefni 1 / Software Project 1

9. Behavior Modeling

HBV501G – Fall 2015

Matthias Book



HÁSKÓLI ÍSLANDS
VERKFRÆÐI- OG NÁTTÚRVÍSINDASVIÐ
ÍÐNAÐARVERKFRÆÐI-, VÉLAVERKFRÆÐI-
OG TÖLVUNARFRÆÐIDEILD

Recap: Mapping Complex Data Types

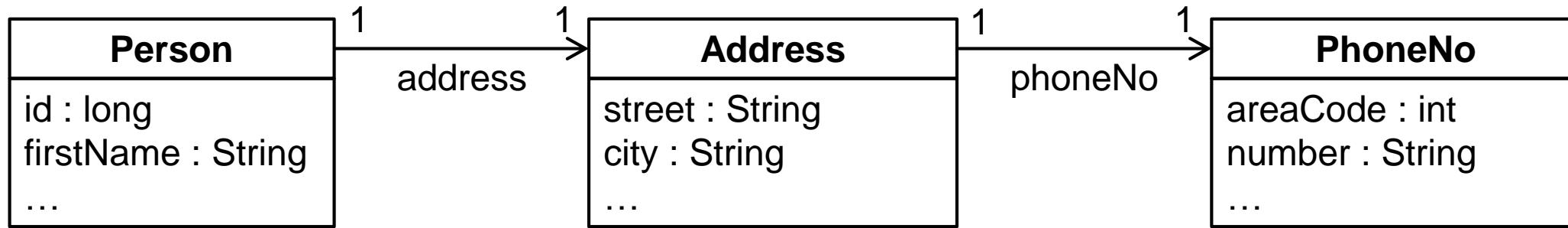
```
@Entity  
public class PostitNote {  
    @Id  
    @GeneratedValue(strategy =  
        GenerationType.IDENTITY)  
    private Long id;  
    private String name;  
    private String note;  
  
    public String getName() { return name; }  
    public void setName(String name) {  
        this.name = name; }  
    // [...other getters & setters...]  
}
```

- Recap: Automatic mappings to SQL types exist for common Java types:
 - Primitive types (int, float, boolean, ...)
 - Primitive type wrappers (Integer, ...)
 - Strings and arrays (char, byte[], ...)
 - Date, Time, Calendar
 - Enumerated properties (enum)
 - Any class implementing Serializable
- What about complex attributes?
 - Complex properties of the entity,
e.g. a structured PhoneNumber
 - Embedded properties
 - References to other entities,
e.g. the LineItems of a Receipt
 - Entity relationships



Composite One-to-One Relationships in UML

- Example: Composition of personal information



Composite One-to-One Relationships as Embedded Properties using JPA

```
@Entity  
public class Person {  
    private long id;  
    private String firstName;  
    private Address address;  
  
    @Id  
    @GeneratedValue(strategy =  
        GenerationType.IDENTITY)  
    public long getID() {...}  
    public void setID(long id) {...}  
  
    @Embedded  
    public Address getAddress() {...}  
    public void setAddress(Address addr) {...}  
    // ...  
}
```

```
@Embeddable  
public class Address {  
    private String street;  
    private String city;  
    private PhoneNo phoneNo;  
    public String getStreet() {...}  
    public void setStreet(String street) {...}  
    public String getCity() {...}  
    public void setCity(String city) {...}  
  
    @Embedded  
    public PhoneNo getPhoneNo() {...}  
    public void setPhoneNo(PhoneNo phNo) {...}  
    // ...  
}
```



Indicates the fields of Address shall be incorporated directly into the Person table

Indicates that Address data is not stored in a table of its own, but embedded into another entity's table

Note: Therefore, no @Id here!

Regular POJO

Embeddable properties can contain embedded properties themselves

Composite One-to-One Relationships in RDBMS

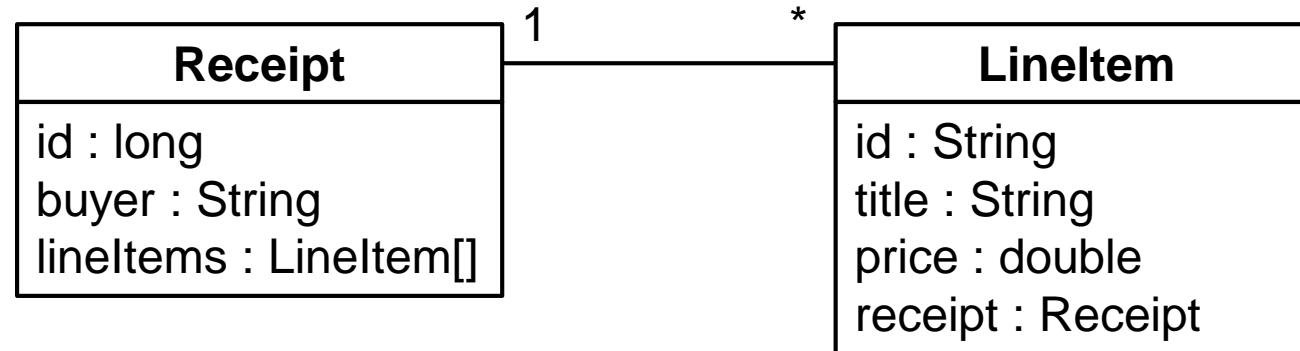
- Example: Resulting database table

Person

| id | firstName | street | city | countryCode | number | ... |
|-----------|------------------|---------------|-------------|--------------------|---------------|------------|
| 1 | Matthias | Dunhagi | Reykjavík | 354 | 525-4603 | ... |
| 2 | Christy | Nathan Road | Hong Kong | 852 | 3107-0566 | ... |
| ... | ... | ... | ... | ... | ... | ... |

One-to-Many Entity Relationships in UML

- Example: Aggregation of line items in a sales receipt



One-to-Many Entity Relationship using JPA

```
@Entity  
public class Receipt {  
    private long id;  
    private String buyer;  
    private Set<LineItem> lineItems =  
        new HashSet<>();  
  
    @Id  
    @ColumnName(name = "ReceiptId")  
    @GeneratedValue(strategy =  
        GenerationType.IDENTITY)  
    public long getId() {...}  
    public void setId(long id) {...}
```

Name of related entity's attribute that references this entity

Don't retrieve related entities from DB until someone accesses them in Java

```
    public String getBuyer() {...}  
    public void setBuyer(String buyer) {...}  
  
    @OneToMany(mappedBy = "receipt",  
        fetch = FetchType.LAZY,  
        cascade = CascadeType.ALL,  
        orphanRemoval = true)  
    public Set<LineItem> getLineItems()  
        {...}  
    public void setLineItems(  
        Set<LineItem> lineItems) {...}
```

Action on related entities when this one is deleted

Object-oriented declaration of the One-to-Many relation:
A reference to a Set of the “many” objects



One-to-Many Entity Relationship in RDBMS

- Example: Resulting database tables

Receipt

| ReceiptId | buyer |
|-----------|----------|
| 1 | Matthias |
| 2 | Christy |
| ... | ... |



Many-to-One Entity Relationship using JPA

```
@Entity  
@Table(name = "Receipt_LineItem")  
public class LineItem {  
    private long id;  
    private String title;  
    private double price;  
    private Receipt receipt;  
  
    @Id  
    @ColumnName(name = "LineItemId")  
    @GeneratedValue(strategy =  
        GenerationType.IDENTITY)  
    public long getId() {...}  
    public void setId(long id) {...}  
  
    Reference to related entity  
    Retrieve related entity from DB immediately  
    There must be a related entity  
    Name of column in this table that will contain the other table's primary key  
    Object-oriented declaration of the Many-to-One relation: A reference to the "one" object  
    public String getTitle() {...}  
    public void setTitle(String title) {...}  
    public double getPrice() {...}  
    public void setPrice(double price) {...}  
  
    @ManyToOne(fetch = FetchType.EAGER,  
        optional = false)  
    @JoinColumn(name = "ReceiptId")  
    public Receipt getReceipt() {...}  
    public void setReceipt(Receipt receipt)  
    {...}
```



One-to-Many Entity Relationship in RDBMS

- Example: Resulting database tables

Receipt

| ReceiptId | buyer |
|-----------|----------|
| 1 | Matthias |
| 2 | Christy |
| ... | ... |

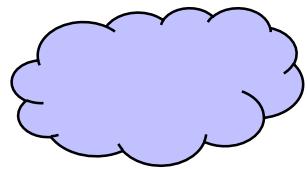
Receipt_LineItem

| LineItemId | title | Price | ReceiptID |
|------------|----------|-------|-----------|
| 1 | Textbook | 7000 | 1 |
| 2 | Headset | 3500 | 1 |
| 3 | Raincoat | 21000 | 2 |
| 4 | Movie | 3500 | 2 |
| ... | ... | ... | ... |

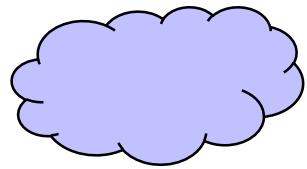


Recap: Inception and Elaboration Artifacts

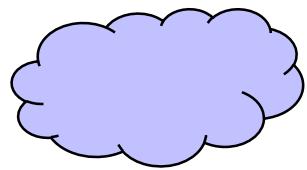
Business Domain



Business Requirements



Quality Attributes



Business Rules

Requirements



Vision and Scope

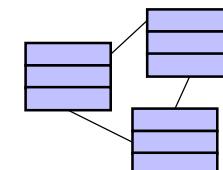


Supplementary Specification

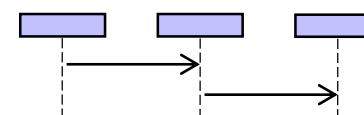


Use Cases

Domain Model



Structural Diagrams

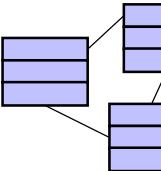


Behavioral Diagrams

Architecture

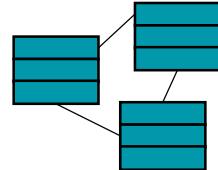


Software Architecture

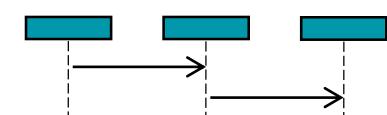


Package Diagrams

Design Model



Structural Diagrams



Behavioral Diagrams

UML Behavior Diagrams



see also:

- Larman: Applying UML and Patterns, Ch. 28 & 29
- Miles, Hamilton: Learning UML 2.0, Ch. 3 & 14

Recap: Sequence Diagrams

■ Synchronous message →

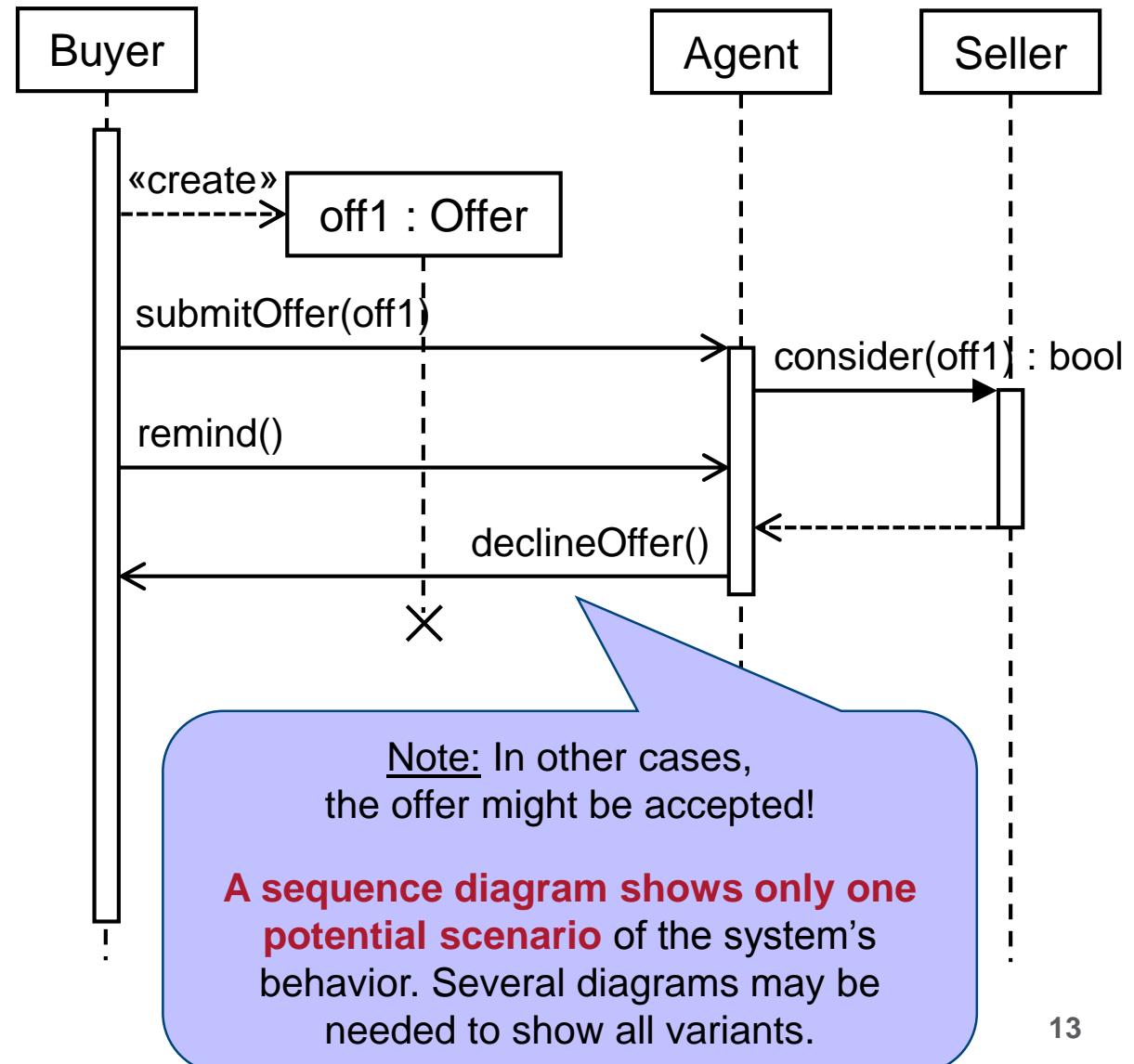
- The message sender waits until the message receiver responds to the invocation.
 - Java interpretation: Calling a method and waiting for it to terminate before control returns to the sender

■ Return message ←-----

- Control flow returns after invocation of synchronous message

■ Asynchronous message →

- The message sender does not wait for the receiver's response but remains in control after sending.
 - Java interpretation: Calling a method in a parallel thread



UML Diagram Types

■ Logical view

- What is the system made up of?
- How do parts interact with each other?
 - Class diagram
 - Object diagram
 - State machine diagram
 - Interaction diagram

■ Process view

- What processes exist in the domain?
- What must happen within the system?
 - Activity diagram

*(Structure following Kruchten's
4+1 View Model of Software Architecture)*

■ Development view

- How are the system's parts and layers organized?
 - Package diagram
 - Component diagram

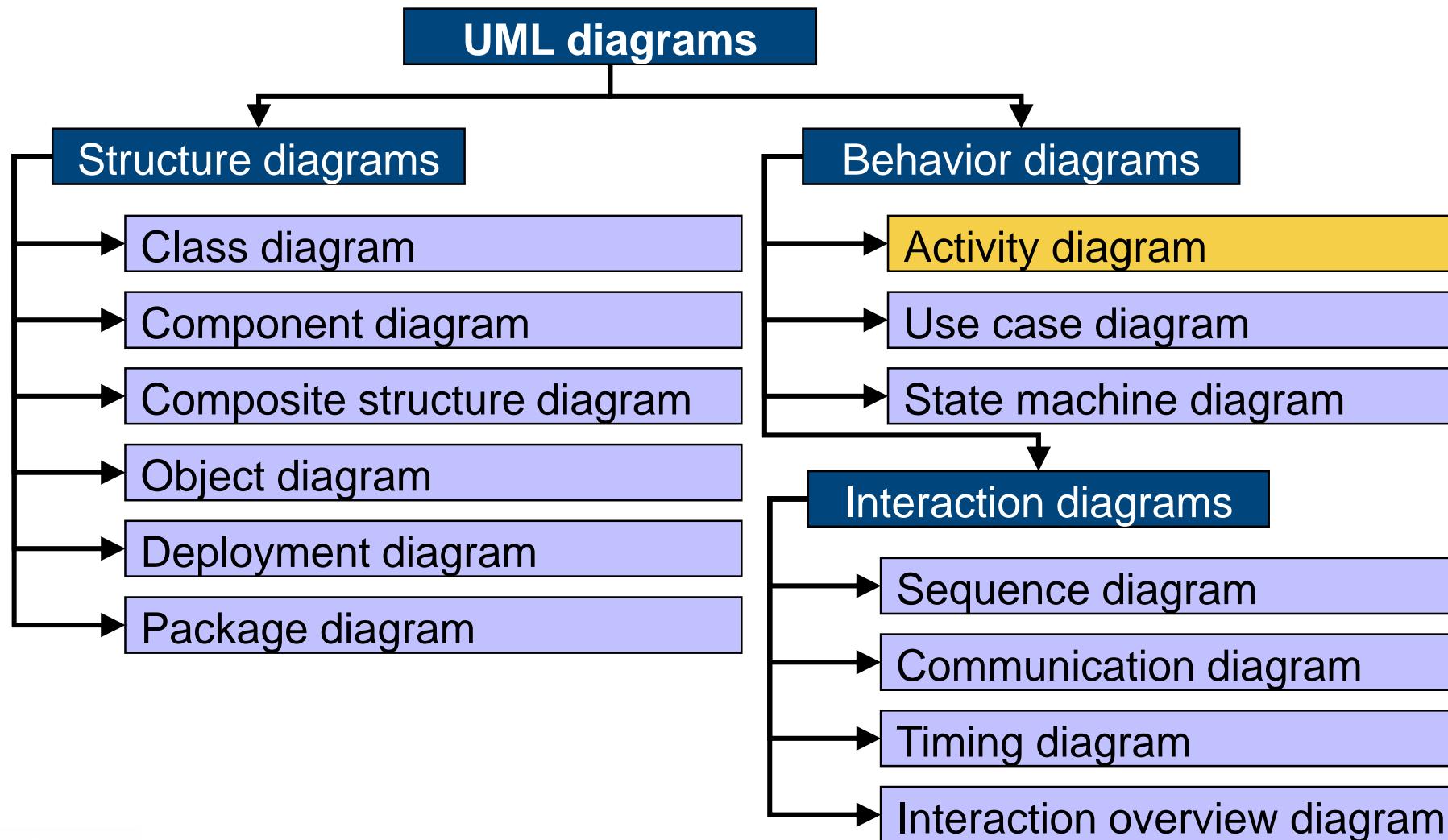
■ Physical view

- How do the abstract parts map to the deployed system?
 - Deployment diagram

■ Use case view

- What is the system supposed to do?
 - Use case diagram
 - Interaction overview diagram

UML Diagram Types



UML Activity Diagrams

■ Purpose

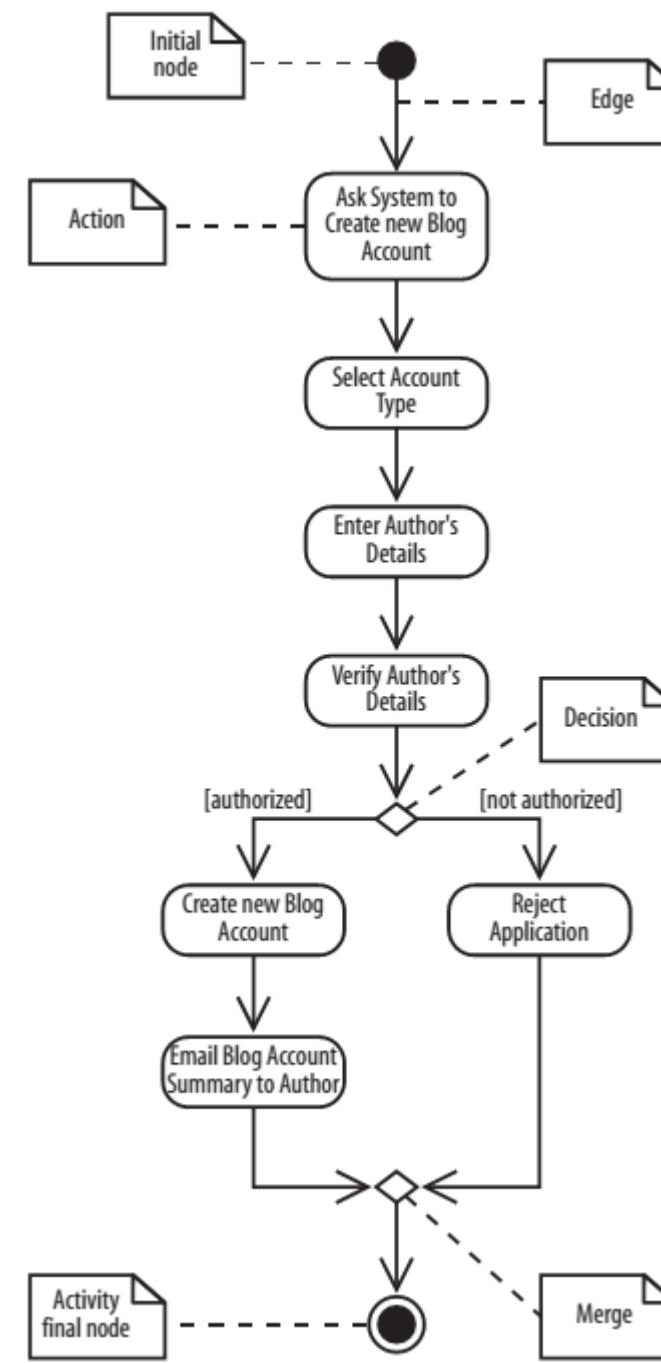
- Visualize the individual steps/actions that constitute a particular process/activity
- Domain model: “How does a process work?”
- Design model: “How does our system implement certain behavior?”

■ Features

- Visualizing processes with conditions, branches and loops
- Expressing concurrency and synchronization
- Visualizing data flows
- Hierarchical nesting



Example: Blog Account Creation



Basic Elements

Action

- An **action** is one step of the activity
 - Sum of all actions constitutes the **activity**



- Control flow



- Initial node



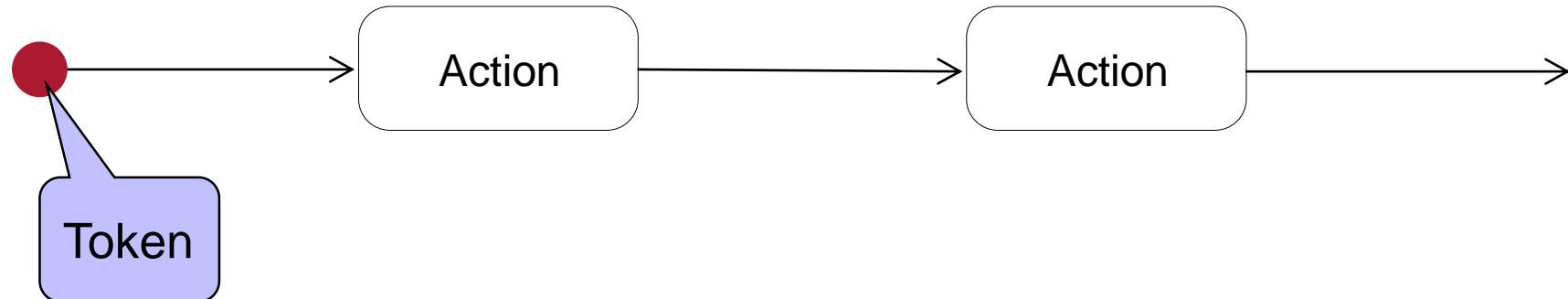
- Final node of a control flow branch



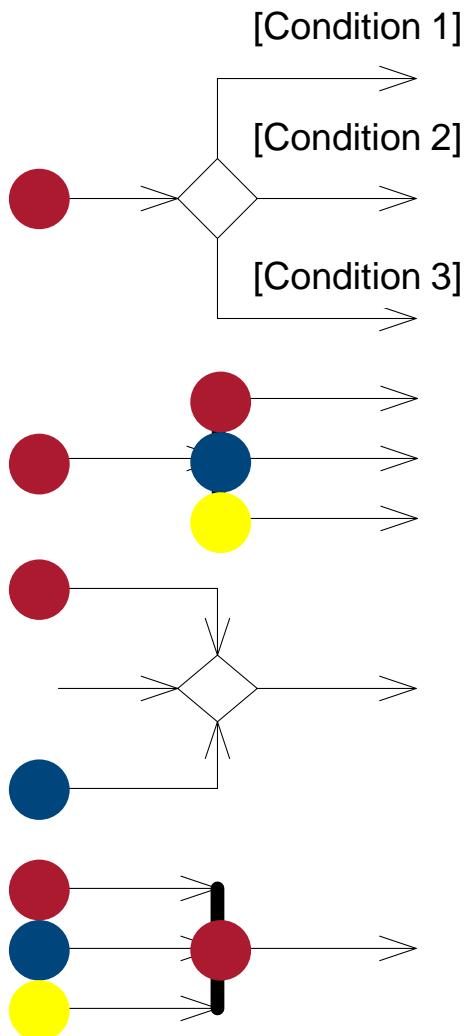
- Final node of activity

Token Concept

- Based on the concept of Petri nets (place/transition nets, P/T net)
- Control flow is simulated by tokens
- An action is executed when a token reaches its inbound edge
- When the action has been executed, a token leaves its outbound edge(s)

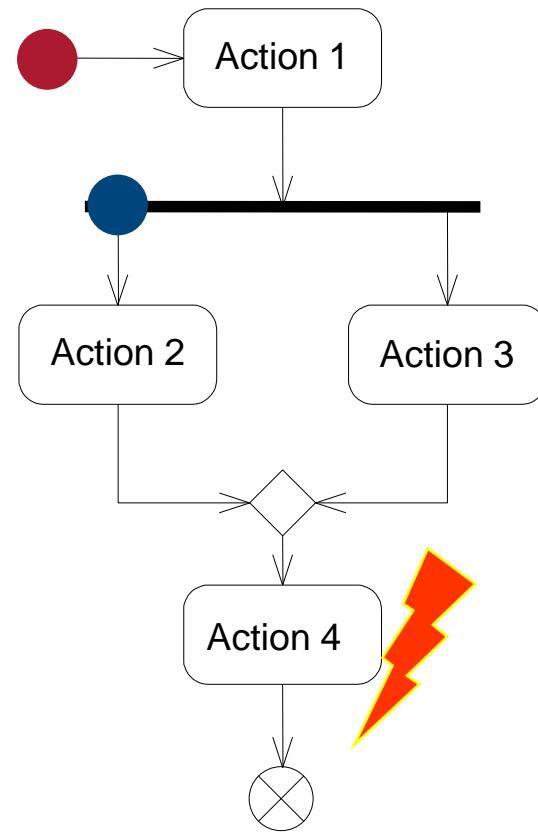
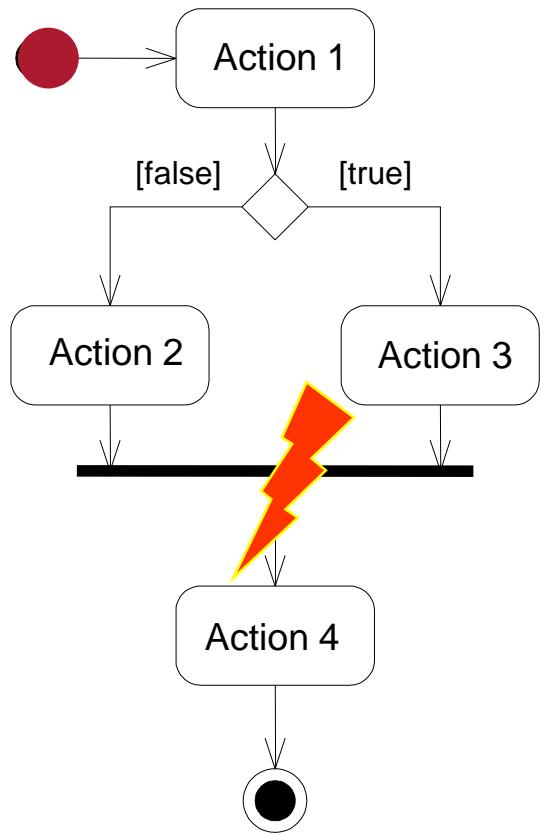


Token Concept for Control Flow



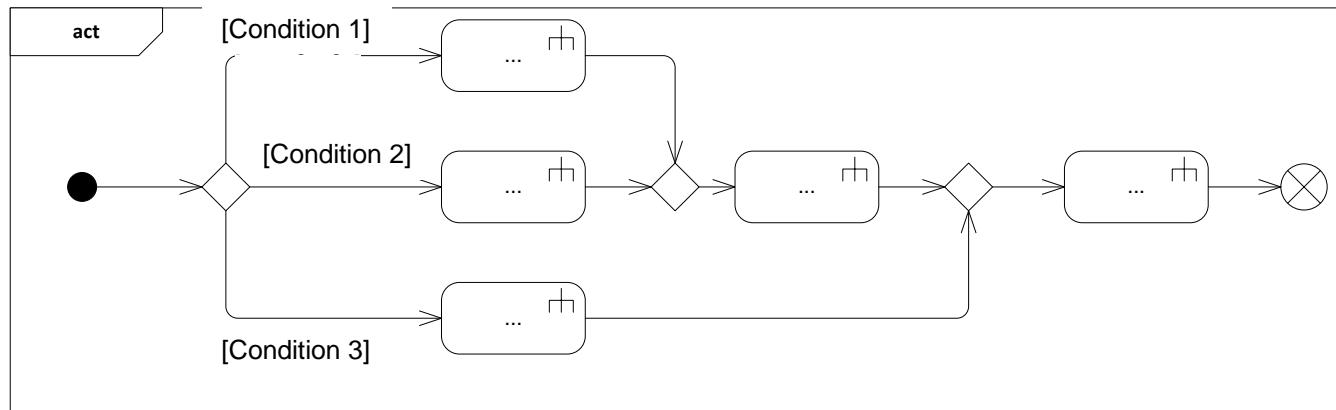
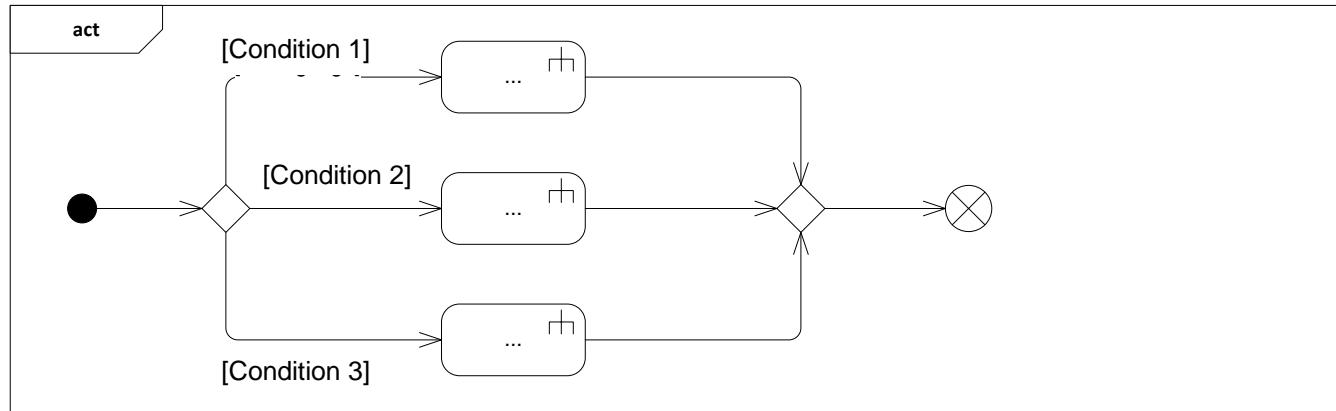
- **Decision nodes** create a token only on one outbound edge, depending on which condition is satisfied.
 - If several conditions are satisfied, the choice of outbound edge is undetermined.
- **Fork nodes** create a token on each outbound edge.
- **Merge nodes** create a token on the outbound edge whenever a token reaches the inbound edge.
- **Join nodes** create a token on the outbound edge only when a token has arrived on each inbound edge.

Modeling Errors



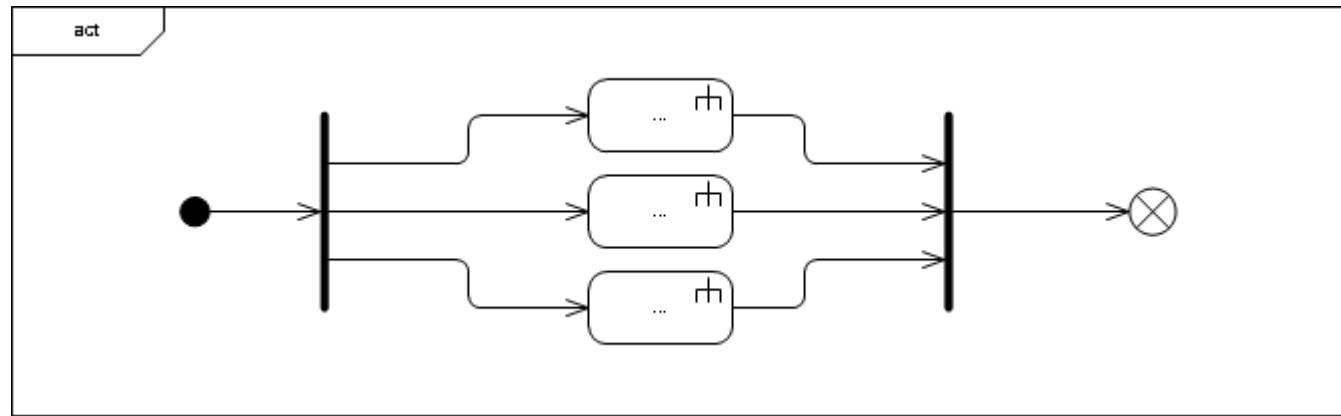
Modeling Rules

- Control flows separated by decision nodes must be recombined by merge nodes

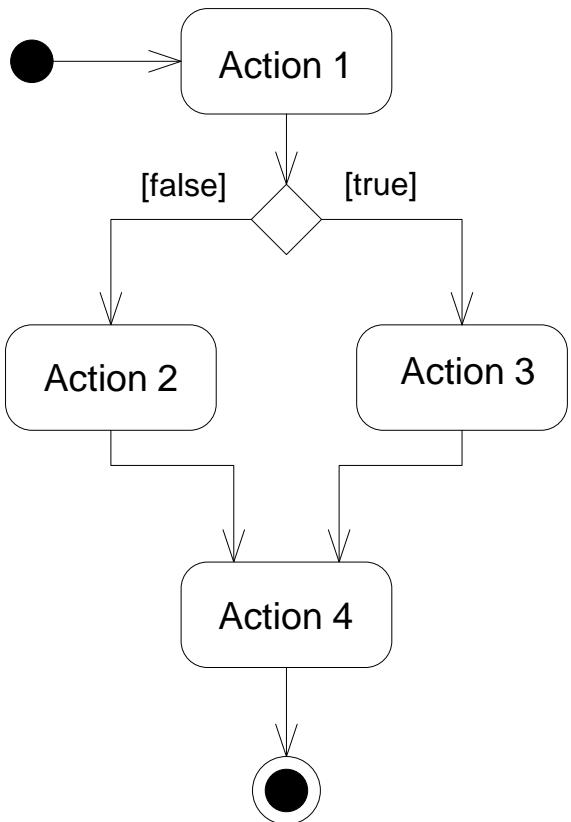


Modeling Rules

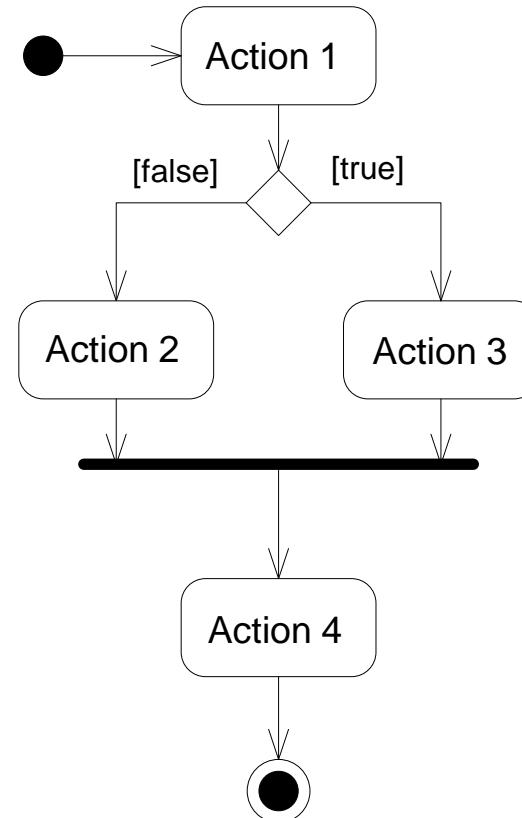
- Control flows split by fork nodes must be recombined by join nodes



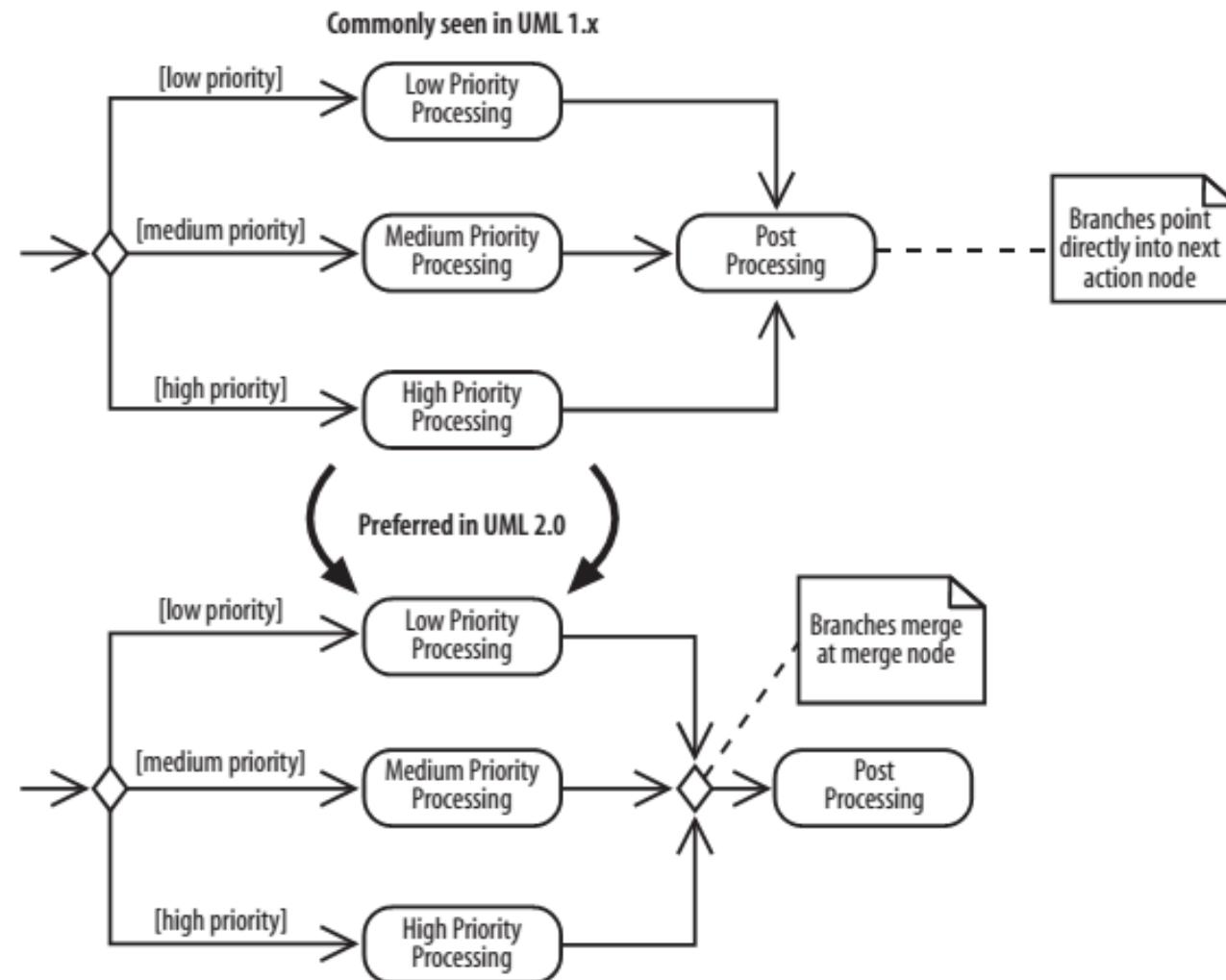
Modeling Errors



▲
≡

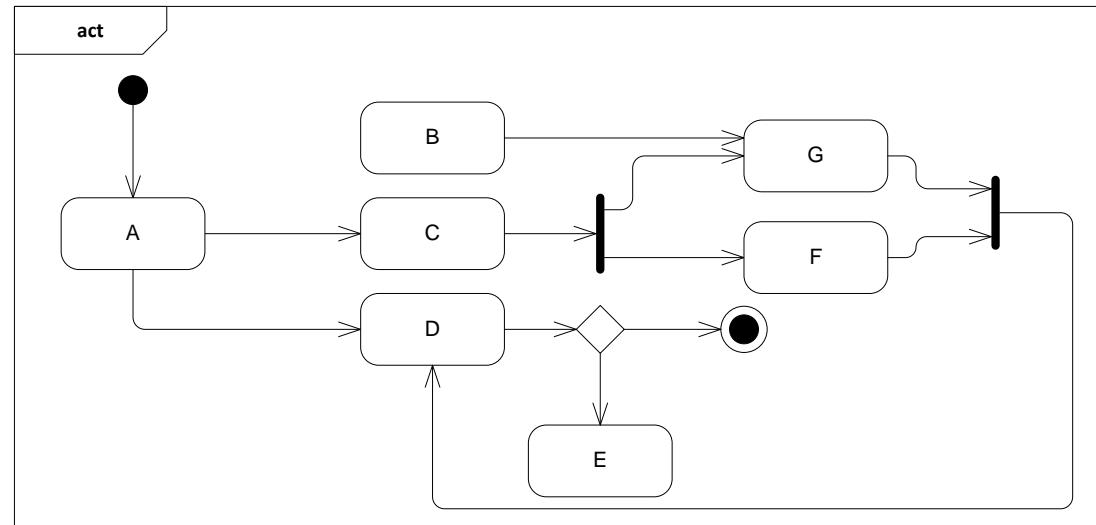


UML1.x vs. UML 2.0

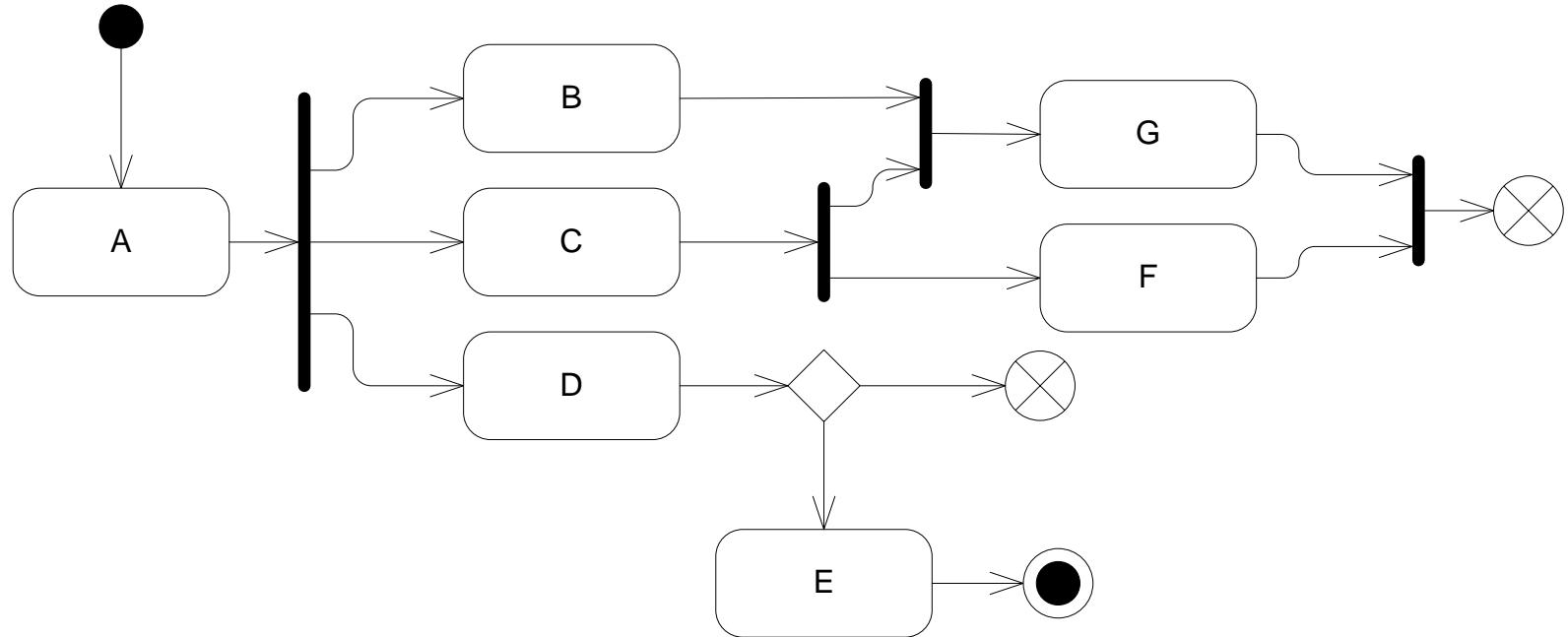


Modeling Rules

- An action must have exactly one inbound and one outbound node.
- An activity diagram must have exactly one initial node and at least one final node.



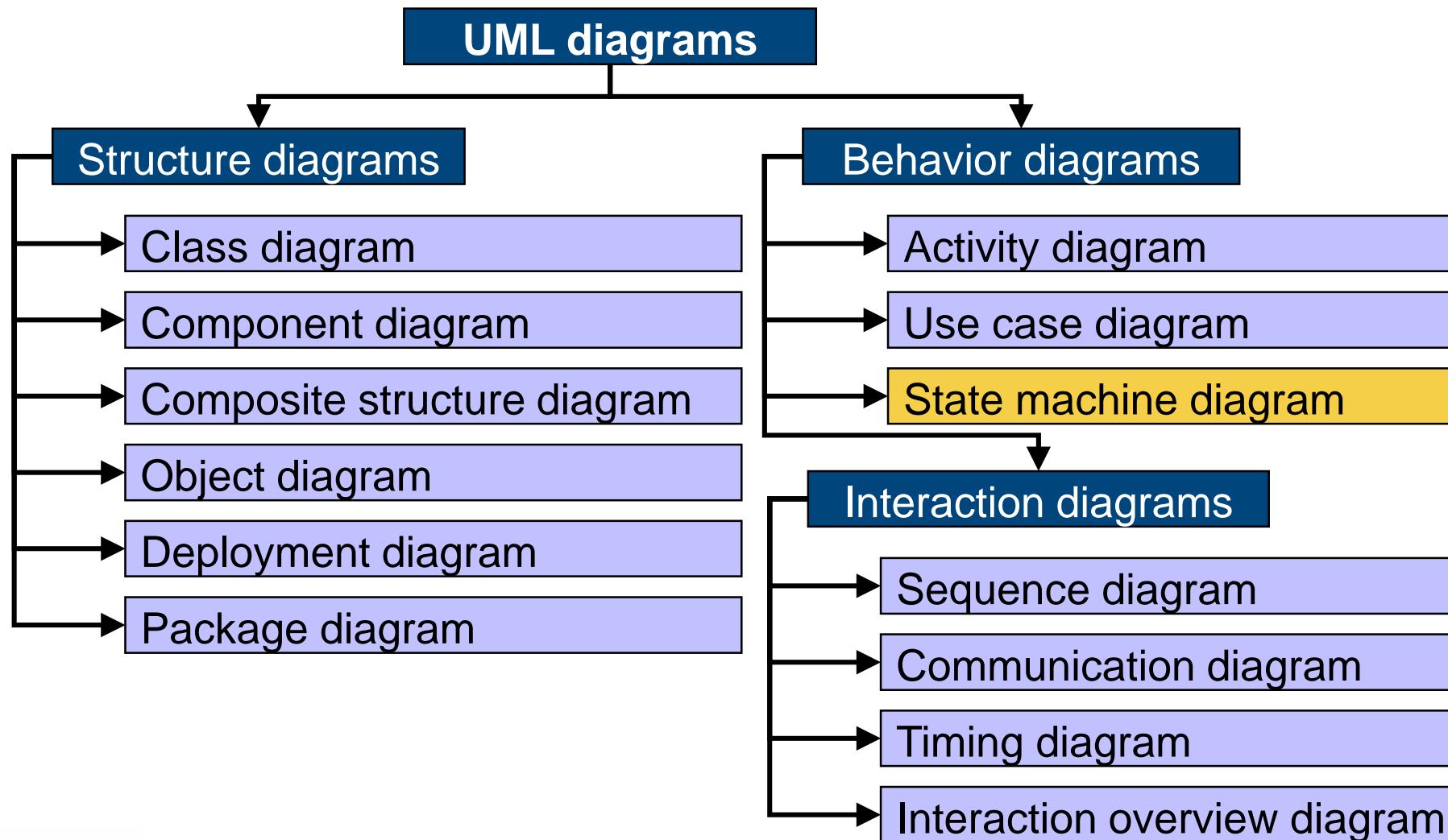
Exercise: Concurrency



(Note: Diagram violates some modeling rules)

1. Which actions are always executed?
2. Why may certain actions never be executed?
3. If G is executed, which activities will have been executed before?
4. Can F be executed before B?

UML Diagram Types



UML State Machine Diagrams

■ Purpose

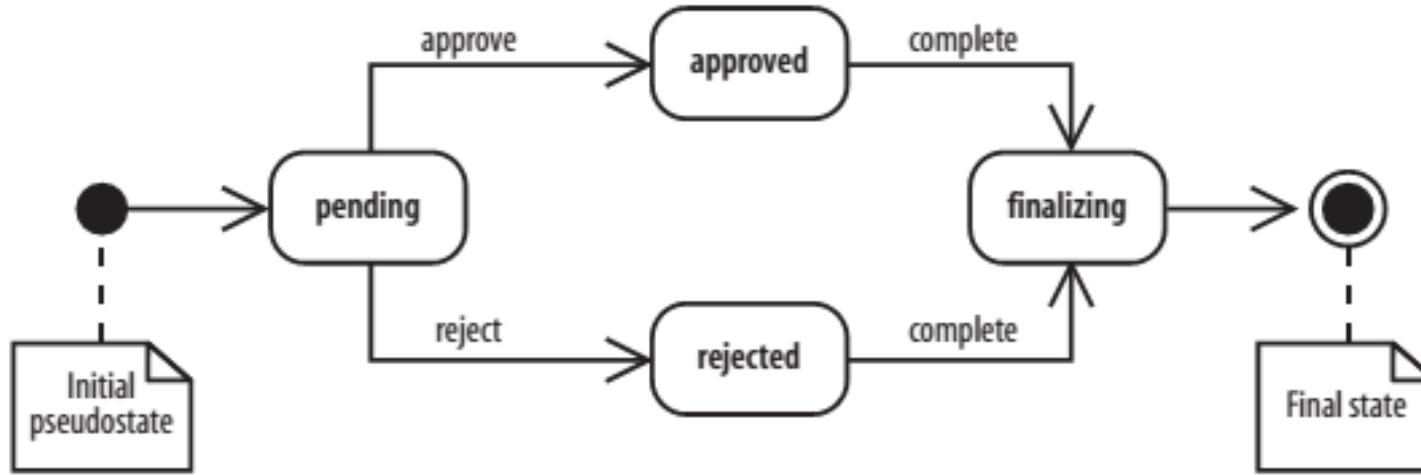
- Visualizing the states that an object, interface, component etc. can adopt, depending on certain events

■ Features

- Precise modeling of states, events, concurrency, conditions, input and output operations
- Hierarchical nesting



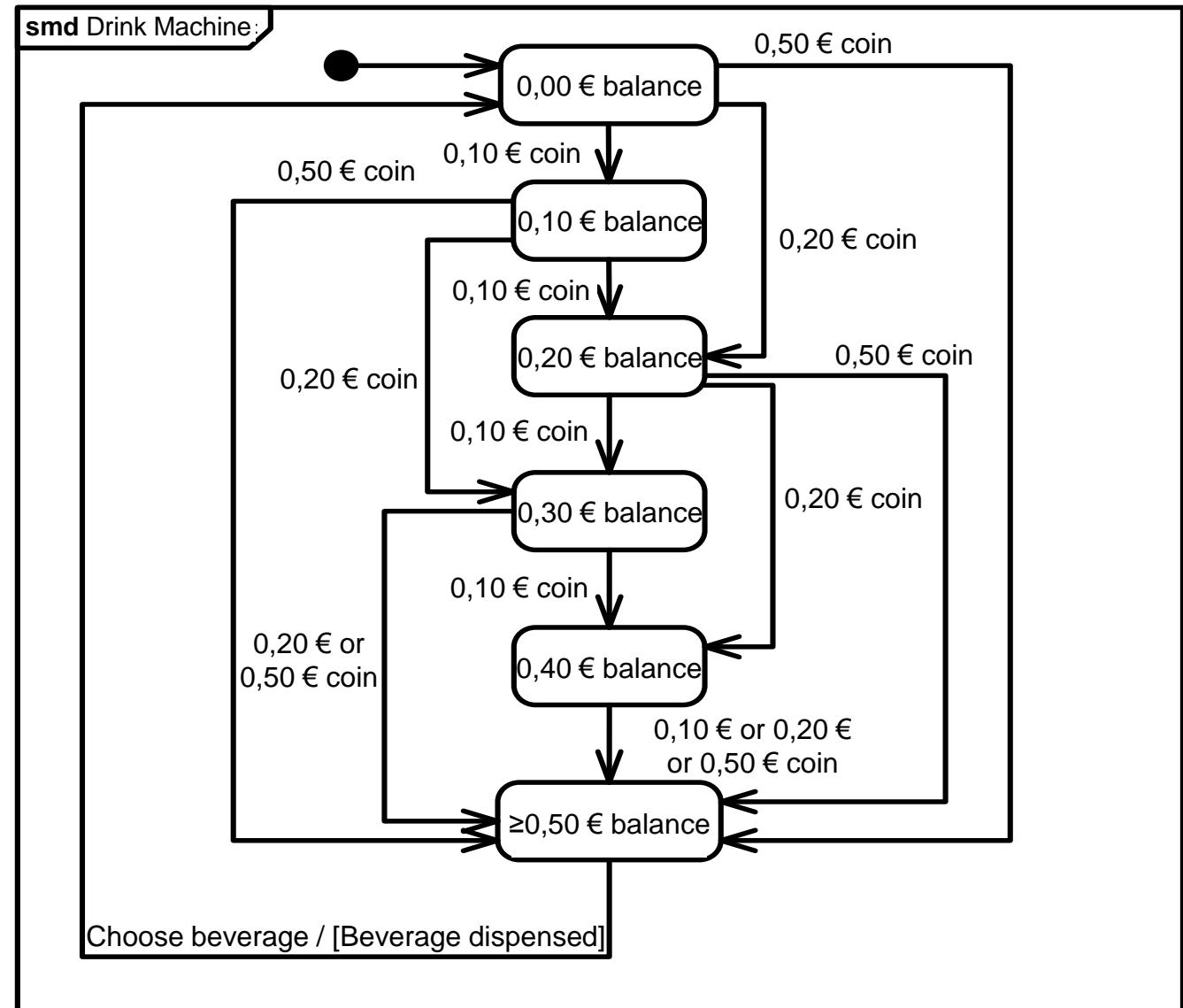
Example: Blog Account Creation



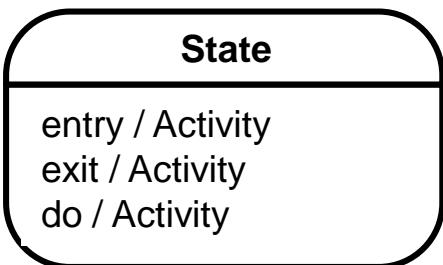
- At any point in time, the system is in exactly one state.
- A transition from the current to the next state occurs instantaneously
 - i.e. it doesn't take time

Example: Automatic Beverage Dispenser

- Alle beverages cost 0,50 €
- The machine accepts 0,10 €, 0,20 € and 0,50 € coins
- No change is given
- After paying at least 0,50 €, the user can choose a drink and receives it

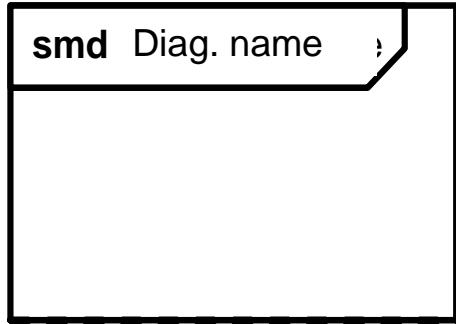


Notation Elements



- A **state** that the system can be it at one time
- Activities can be performed upon certain events:
 - *entry* → activity performed upon entering state
 - *exit* → activity performed upon leaving state
 - *do* → activity performed while in this state
- Initial pseudo state
- Final state
- Only one initial state allowed per diagram!

Notation Elements



Trigger [Guard] / Activity

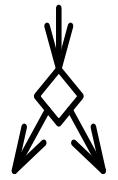


- State machine diagrams should be placed in a named frame
 - (smd / sm = state machine diagram)

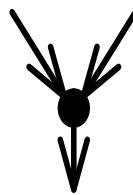
- **Triggers** are events prompting a state transition
- The specified **guard** is a condition that must be true for the transition to occur.
- A specified **activity** can be performed when the transition occurs.

Notation Elements

- **Pseudo states** are used to show complex relationships between states
- The system cannot be in a pseudo state
 - (i.e. passage through pseudo states occurs instantaneously)



- **Choice pseudo state:** Next transition depends on guard condition



- **Merge pseudo state:** Combining inbound transitions from alternate states



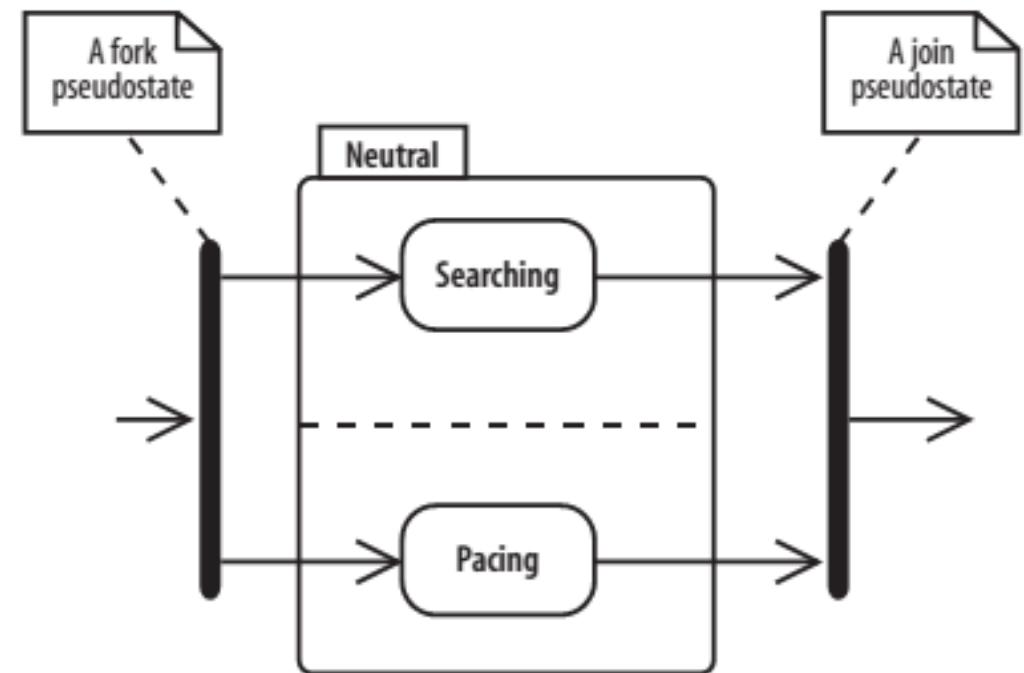
- **Fork pseudo state:** Transition into several concurrent states.



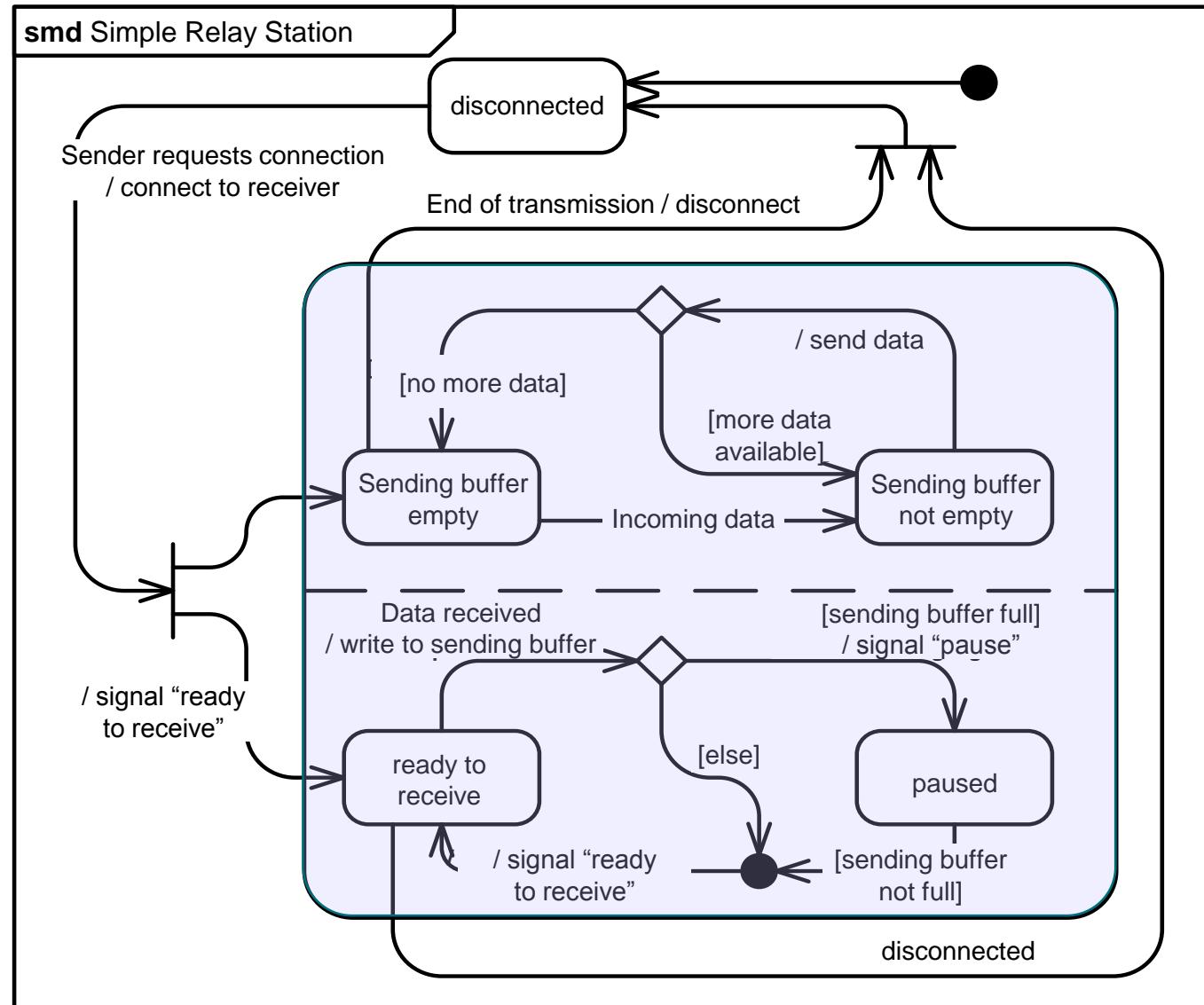
- **Join pseudo state:** Combining incoming transitions from concurrent states

Composite States

- Forks beg the question: How can a system be in two states at once?
- Solution: **Composite states** define regions in each of which the system can only be in one state at a time.



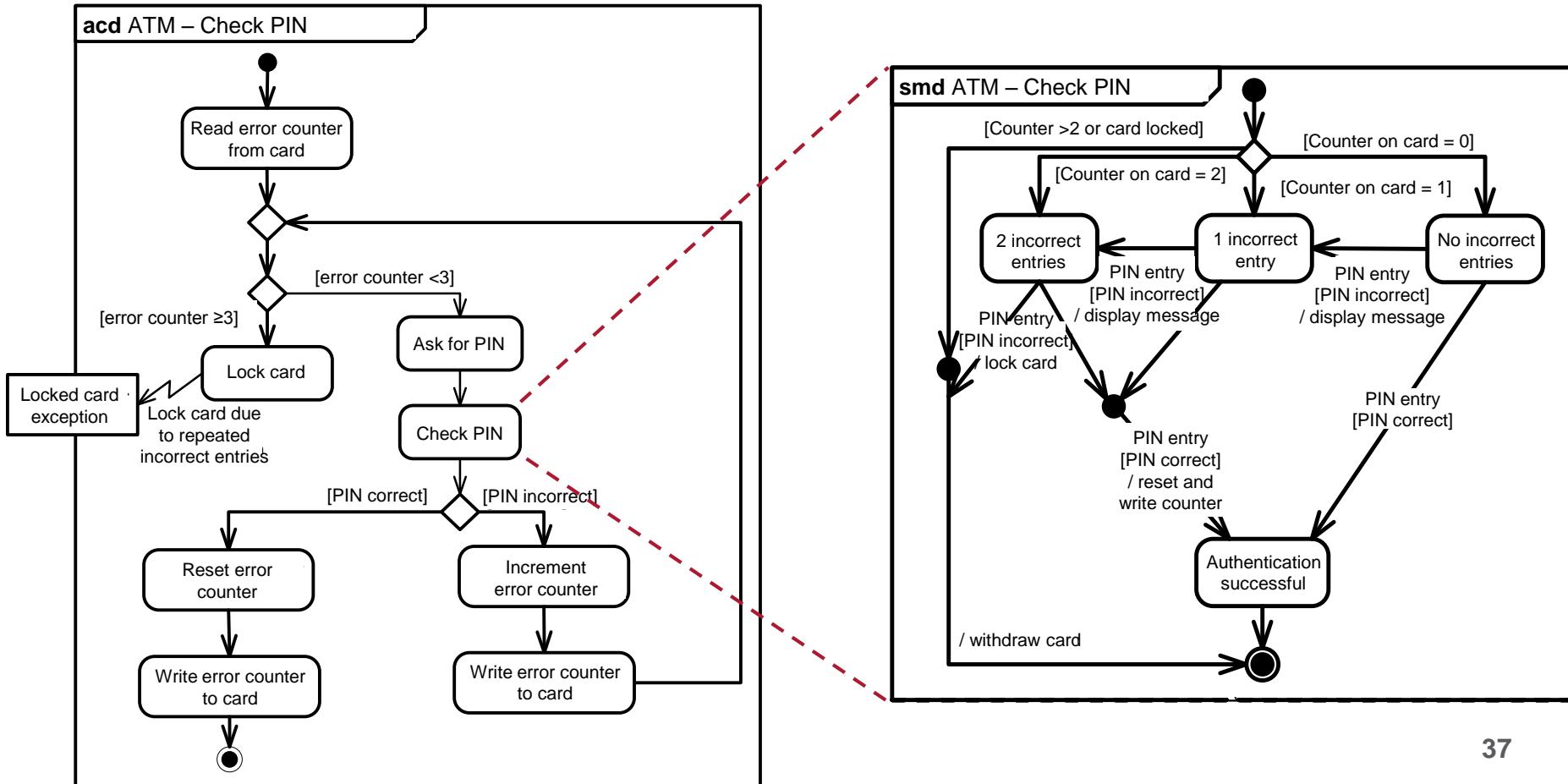
More Complex Example: A Relay Station



Activity Diagrams vs. State Machine Diagrams

- Activity diagrams describe what a component does
- State machine diagrams describe how a component changes

- Example: ATM checking a card holder's PIN
 - **acd:** Overall PIN checking process
 - **smd:** Keeping track of invalid entries



Team Assignment 3

- Having understood the application domain, the next step is designing the technical solutions that shall be implemented to satisfy the requirements.
 - This begins in the Elaboration phase and continues iteratively throughout the Construction phase, always focusing especially on the use cases to be implemented in the next iteration.

The **design model** includes

1. for the **structural** aspects: UML class and package diagrams
 - defining the classes your software system will be comprised of, and their relationships
 2. for the **behavioral** aspects: UML sequence, activity and/or state diagrams
 - explaining how classes work together, how objects or components change state, etc.
- Producing these artifacts and explaining the considerations that went into them will be your job in Team Assignment 3.



Team Assignment 3: Content

- By **Sun 15 Nov**, submit in Uglá:
 - **Structural UML diagrams** defining the structure of your project, including: (~50% of grade)
 - A **class diagram** showing the implementation classes your final system shall consist of
 - with attributes, data types, methods, relations, multiplicities
 - A **package diagram** showing the grouping of classes into packages
 - Relevant **behavioral UML diagrams** describing the dynamic aspects, e.g.: (~50% of grade)
 - One or more **sequence diagrams** detailing complex collaboration between certain classes
 - One or more **state machine diagrams** detailing state changes in a particular object or component
 - One or more **activity diagrams** detailing navigation or user interaction with the software
- On **Thu 19 Nov**, explain the diagrams of your design model to your tutor:
 - Why did you choose particular solutions? What design patterns did you use, if any?
 - How are your architectural decisions reflected in the structure?
 - Which aspects of the system's behavior are particularly complex?
 - Where do you see room for design improvement (if you had time after the semester)?

Team Assignment 3: Format

- All artifacts must be produced by all team members together.
- The **UML diagrams of the design model** must
 - be submitted in one **PDF document by Sun 15 Nov in Uglar**
 - contain your team number, the names and kennitölur of all team members
- Only the team member who will present should submit the documents.
 - Don't submit multiple versions – we'll just grade the first one we encounter!
- The presentation
 - should be given by one representative of the team (a different one for each assignment!)
 - should be based on the submitted document (don't prepare extra slides!)
 - should take around 5-10 minutes (plus some questions asked by the tutor)
- All team members receive the same grade, with some variation for the presenter



Team Assignment 3: Grading Criteria for Design Model

▪ Structural Diagrams

▪ General criteria

- ✓ syntactically correct UML

▪ Criteria for the class diagram

- ✓ provides a clear and comprehensive overview of technical implementation
- ✓ reflects object-oriented design principles
- ✓ Bonus: makes use of design patterns, where reasonable

▪ Criteria for the package diagram

- ✓ illustrates partitioning of high-level components of system clearly
- ✓ illustrates assignment of classes to packages

▪ Behavioral Diagrams

▪ General criteria

- ✓ Reasonable choice of diagram types and scopes to describe interesting aspects
- ✓ syntactically correct UML

▪ Criteria for any sequence diagrams

- ✓ clearly shows class collaboration (methods, relevant parameters in relevant scenarios)

▪ Criteria for any activity diagrams

- ✓ clearly shows processes occurring within the system or supported by the system

▪ Criteria for any state machine diagrams

- ✓ clearly shows changes in state of a particular object, component or system



Team Assignment 3: Grading Criteria for Presentation

■ Design Model

- ✓ Presenter can explain rationales for structuring the system in the shown way
- ✓ Presenter can explain how key aspects of the system work
- ✓ Presenter can use a combination of structural and behavioral diagrams to show how the parts of the system collaborate to satisfy the domain requirements

■ Overall

- ✓ Presenter shows initiative, explains things clearly, shows understanding of the approach



Gangi þér vel!

book@hi.is



HÁSKÓLI ÍSLANDS
VERKFRÆÐI- OG NÁTTÚRVÍSINDASVIÐ

IÐNAÐARVERKFRÆÐI-, VÉLAVERKFRÆÐI-
OG TÖLVUNARFRÆÐIDEILD





Hugbúnaðarverkefni 1 / Software Project 1

10. Design Models

HBV501G – Fall 2015

Matthias Book



HÁSKÓLI ÍSLANDS
VERKFRÆÐI- OG NÁTTÚRVÍSINDASVIÐ
ÍÐNAÐARVERKFRÆÐI-, VÉLAVERKFRÆÐI-
OG TÖLVUNARFRÆÐIDEILD

Design Models

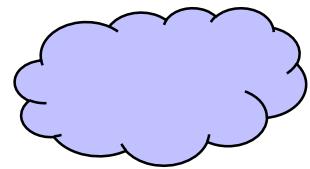
see also:

- Larman: Applying UML and Patterns, Ch. 17 & 25

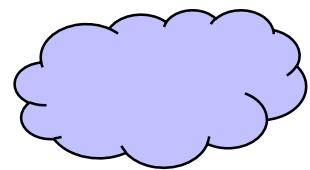


Recap: Inception and Elaboration Artifacts

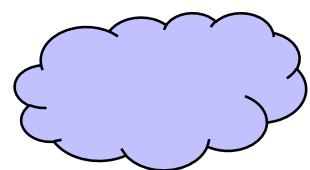
Business Domain



Business Requirements



Quality Attributes



Business Rules

Requirements



Vision and Scope

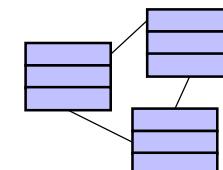


Supplementary Specification

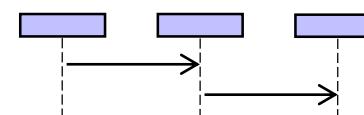


Use Cases

Domain Model



Structural Diagrams

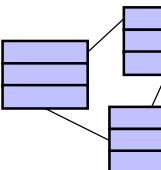


Behavioral Diagrams

Architecture

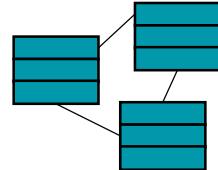


Software Architecture

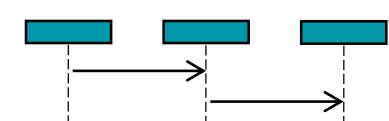


Package Diagrams

Design Model



Structural Diagrams



Behavioral Diagrams

Domain Models vs. Design Models

- We create **domain models** in order to **understand the application domain**
 - So we can talk to the business stakeholders about their requirements on their own terms
 - So we can introduce new team members to the purpose of the software before introducing them to its implementation
- Domain models illustrate business aspects / requirements / **problem domain**
- We create **design models** in order to express **how our solution shall work**
 - So we can talk to fellow developers
 - about what the best technical solutions to a given domain problem would be
 - about the concrete structures that we plan to implement
 - So we can introduce new team members to the internal structure/mechanics of the software after they have understood the business domain
- Design models illustrate technical aspects / implementation / **solution domain**



From Domain Models to Design Models

- Design models are not merely an incremental refinement of domain models.
- Actors in domain model may be mere data objects in design model
 - e.g. a Buyer driving a real-life sales process
- Passive objects in domain model may have responsibilities in design model
 - e.g. a Sale object calculating the total of its LineItems
- Some elements of domain model may not be part of design model
 - e.g. the User is still part of the whole system, but “in the flesh” rather than as an object
- Some necessary parts of the design model may not be in the domain model
 - e.g. objects describing types rather than instances; data structures; technical components...
- The design model is inspired by the domain model, but not derived from it.

Object-Oriented Design Heuristics

Guidelines / considerations for constructing the design model:

- A. Identifying the most suitable creator of objects of a particular class (→ L17.10)
- B. Identifying the most suitable objects to fulfill certain responsibilities (→ L17.11)
- C. Expressing support concepts not present in the application domain (→ L25.2)
- D. Keeping objects' responsibilities focused (→ L17.14)
- E. Keeping objects independent from other objects (→ L17.12)
- F. Protecting other objects from variation/instability within an object (→ L25.4)
- G. Expressing variations in object behavior (→ L25.1)
- H. Decoupling objects even if they need to work closely together (→ L25.3)
- I. Assigning responsibility for reacting to user input (→ L17.13)



A. Identifying Most Suitable Creators of Objects (Creator Principle)

■ Problem

- In the application domain, many objects might simply exist *a priori*
- But they need to be explicitly created in the software
- Sometimes, we may even need to create various representations of them
 - e.g. a general ProductDescription, and an actual Product that can be put in a user's Sale

■ Recommendation

- Class A should be created by class B if
 - B “contains” or compositely aggregates A
 - B records or closely uses A
 - B has initializing data that needs to be passed to A upon its instantiation
 - i.e. B is an “Expert” for A



A. Identifying Most Suitable Creators of Objects (Creator Principle)

▪ Contraindications

- If creation requires particular complexity, e.g. ...
 - recycling instances for performance reasons
 - conditionally creating instances from a family of related classes
- ...creation should be delegated to a Factory helper class

▪ Note

- Creators are often (but not always) identical with Experts.



B. Identifying Suitable Objects to Fulfill Responsibilities (Expert Principle)

■ Problem

- Domain and design objects are best understood as having certain responsibilities.
- Which objects should implement which responsibilities?
- May be hard to decide when several objects collaborate to fulfill a responsibility in the application domain

■ Recommendation

1. Clearly state what the responsibility to assign to some class is
2. Identify the class that has most of the information pertinent to that responsibility
 1. Look for such a class in the design model first
 2. If there is no such class, derive an appropriate class from the domain model
3. Assign the responsibility to that class



B. Identifying Suitable Objects to Fulfill Responsibilities (Expert Principle)

▪ Contraindications

- Expertise shouldn't be the only criterion in assigning responsibility to classes; coupling and cohesion need to be considered as well
 - e.g. a data object shouldn't be responsible for displaying / storing itself in the UI / DB

▪ Note

- Even objects that are “passive” in the application domain can serve active roles as Experts and Creators of classes in the technical design and implementation.



C. Expressing Support Concepts Not Found in Domain (Fabrication Principle)

■ Problem

- Sometimes, there is no suitable domain-model class to which a certain responsibility can be assigned in the design model
- e.g. responsibilities for organizing data into structures or for performing technical / support operations such as persistence

■ Recommendation

- Add classes to the design model as needed, even if they do not have a counterpart in the domain model, especially
 - technical classes such as PersistentStorage, Controller etc.
 - structural classes such as List, Map, Tree etc.
 - representational classes such as a ProductDescription for a Product
 - behavioral classes such as TableOfContentsGenerator

■ Contraindications

- Do not overdo behavioral decomposition – not every algorithm needs its own class; representational classes can also have behavioral responsibility



D. Keeping Objects' Responsibilities Focused (High Cohesion Principle)

■ Problem

- How to ensure that objects remain focused, understandable, reusable, maintainable
- and that they are stable and not constantly exposed to change?

■ Recommendation

- When assigning responsibilities to classes (based on other principles), ensure that the responsibilities of a class are strongly related to each other.

■ Contraindications

- Under some circumstances, it may be necessary to give up cohesion with regard to one functional aspect in order to gain it for another functional aspect
 - e.g. bundling all persistent storage access in one class (high cohesion wrt. storage), even if that class is used to store many different kinds of data (low cohesion wrt. data types)



E. Keeping Objects Independent from other Objects (Low Coupling Principle)

■ Problem

- How to ensure that objects are as independent as possibly from other objects, that they are largely unaffected by outside changes, and can be reused?

■ Recommendations

- When assigning responsibilities to classes (based on other principles), ensure that the class is connected to, has knowledge of, or depends on other elements as *little* as possible.
 - Try to avoid the following to the most reasonable degree possible:
 - Class X has an attribute referring to objects of class Y
 - Class X has a method referring to objects of class Y in parameters, local variables or return type
 - Class X calls a method of class Y
 - Class X is a direct or indirect subclass of class Y
 - Class X implements the interface Y



E. Keeping Objects Independent from other Objects (Low Coupling Principle)

▪ Recommendations (cont'd.)

➤ “Don’t talk to strangers” principle:

Within an object’s method, try to access only attributes and methods of:

- The object (`this`) itself
- A parameter of the method
- An attribute of the object (`this`)
- An element of a collection that is an attribute of the object (`this`)
- An object created within the method
- Avoid (within reason) long chains of reference resolutions
 - e.g. `obj.getX().getY().getZ().method()`

- ## ▪ Note: Relax – it's impossible to follow all these recommendations strictly
- Objects do need to talk to each other
 - Just make sure their communication is focused and reasonable

E. Keeping Objects Independent from other Objects (Low Coupling Principle)

▪ Contraindications

- Some degree of coupling is natural since objects are *supposed* to collaborate
- Keep coupling at a reasonable level
 - If coupling is too high, classes will be exposed to changes elsewhere
 - If coupling is too low, classes will be stuffed with responsibilities, violating High Cohesion principle
- Coupling to stable and pervasive elements (e.g. in the Java class libraries) is fine
 - e.g. long chains of method calls are common when working with streams

▪ Notes

- High coupling is not a problem per se – the problematic part is coupling to elements that are *unstable* (in their interface, behavior, presence...)
 - Avoid coupling to unstable elements
- But don't “future-proof” your design excessively or spend effort on minimizing coupling when there is no realistic stability threat

F. Protecting Other Objects from Variation/Instability (Protected Variations Principle)

■ Problem

- How to ensure that variations or instabilities within a class that we are designing does not have undesirable impact on other classes?

■ Recommendations

- Identify points of predicted variation or instability
- Create a stable interface around them that hides any internal changes

■ Contraindications

- It makes sense to protect external objects against known variation points in our object
- But consider if it's worth the effort to prophylactically protect against possible evolution points

■ Note

- This is a fundamental principle of software design (“information hiding” – D. Parnas, 1972)



G. Expressing Variations in Object Behavior (Polymorphism Principle)

■ Problem

- How to implement alternative behaviors depending on the “type” of a domain entity?
- How to exchange service implementations without having to inform the calling layer?

■ Recommendations

- Encapsulate alternative behaviors in different subclasses of the same superclass or interface
- Refer to the implementation only through the superclass/interface
- Do not make the type of a domain entity explicit in an attribute, and don't test for it and use conditional logic to perform different behaviors
 - “Type” is a major built-in concept of object-oriented languages – use it, don't simulate/rebuild it!
- Prefer using interfaces, esp. when you don't need to inherit existing implementations, or when you want to combine independent characteristics. Use abstract classes otherwise.

■ Notes

- A number of common design patterns rely on polymorphism, e.g. Adapter, Command, Composite, Proxy, State, Strategy



H. Decoupling Objects that Would Be Closely Coupled (Indirection Principle)

■ Problem

- How to avoid direct coupling between several classes that rely closely on each other, but should not or cannot communicate directly with each other?

■ Recommendation

- Assign responsibility for facilitating the communication to an intermediate object that abstracts away from the characteristics the objects should not expose to each other.
 - e.g. the JPA acts as an intermediary between business logic and persistence logic, allowing both layers to work closely together without having to be aware of each other's implementation details

■ Notes

- A number of common design patterns rely on indirection, e.g. Adapter, Bridge, Façade, Observer, Mediator



I. Assigning Responsibility for Reacting to User Input (Controller Principle)

■ Problem

- The user interface layer should only be concerned with displaying the user interface (UI).
- The business layer should only be concerned with executing business logic, without being aware of the UI.
- Who determines what happens upon certain UI events, and what to display upon certain business logic results?

■ Recommendations

- Have a dedicated control layer separating UI and business logic
- Have several controllers – use either of these approaches to structure them:
 - A **façade controller** handles everything related to a particular subsystem or component
 - A **use case controller** handles all aspects of a particular use case
- Let controllers receive and **prepare data** as necessary for processing by either layer
- Let controllers **delegate control** to business layer for actual processing
- Make sure you don't end up with one bloated (low cohesion) controller doing everything!



From Design Heuristics to Design Patterns

- The previously discussed principles are general guidelines for the construction of any object-oriented (or modular) system.
- Design patterns are proven ways of satisfying these principles while solving common design problems.



Design Patterns

see also:

- Larman: Applying UML and Patterns, Ch. 26
- Freeman, Robson: Head First Design Patterns



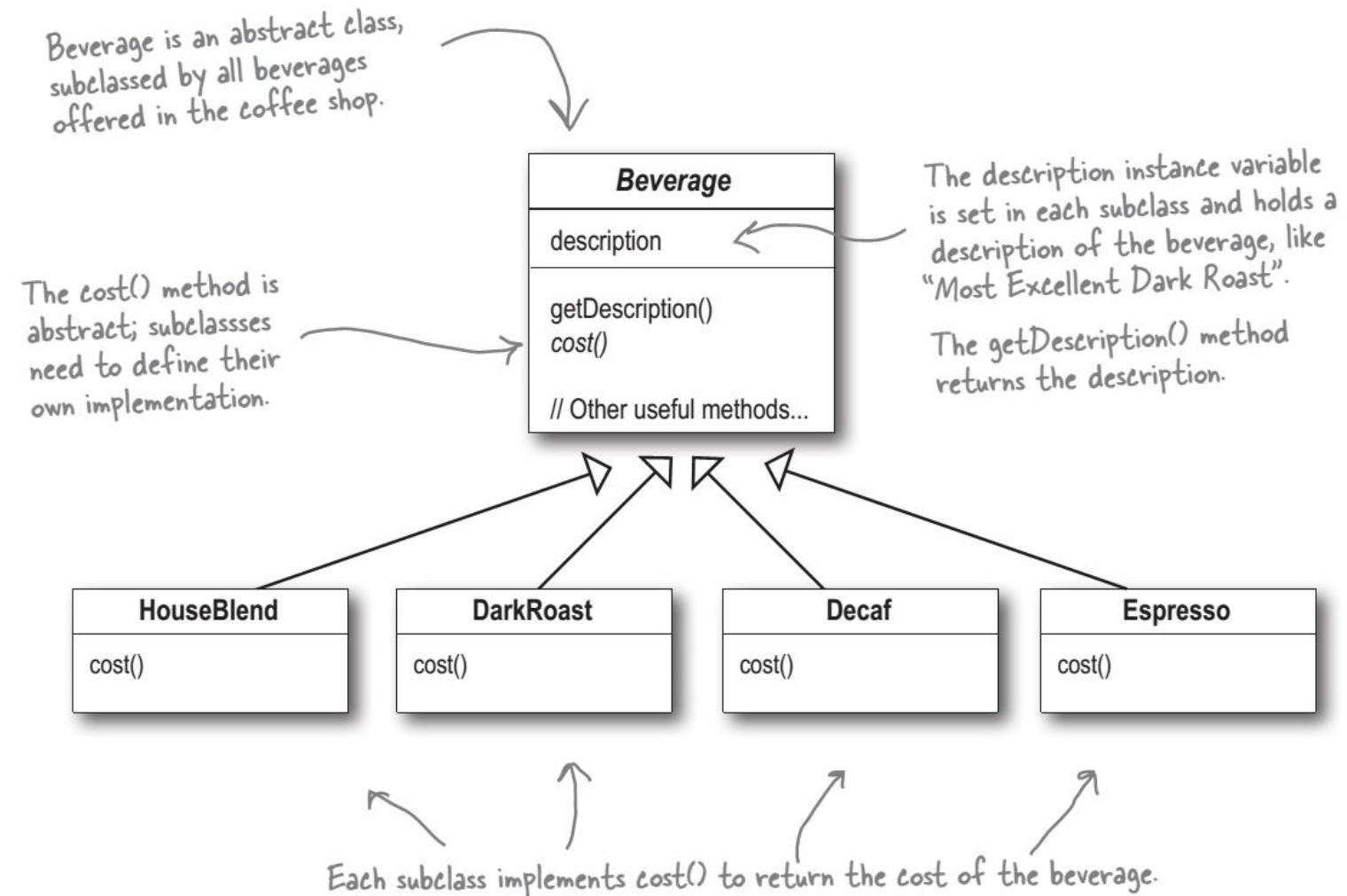
Recap: Design Patterns

- Solving new problems by applying solutions that have been proven in the past
 - How is the new problem similar to a previous problem?
 - How was the previous problem solved?
 - Is that solution generalizable?
 - How can the general solution be applied to the concrete problem?
- Description of the generic solutions as **patterns**
 - Originally a concept from the domain of architecture, only later applied to software
- Advantages:
 - **Pattern collections** (“pattern languages”) bundle comprehensive domain experience
 - **Explicit description** of characteristics and impacts of design decisions reduces risks
 - **Common vocabulary** simplifies communication and prevents misunderstandings

Decorator Pattern

Motivation: Typical Inheritance Solution

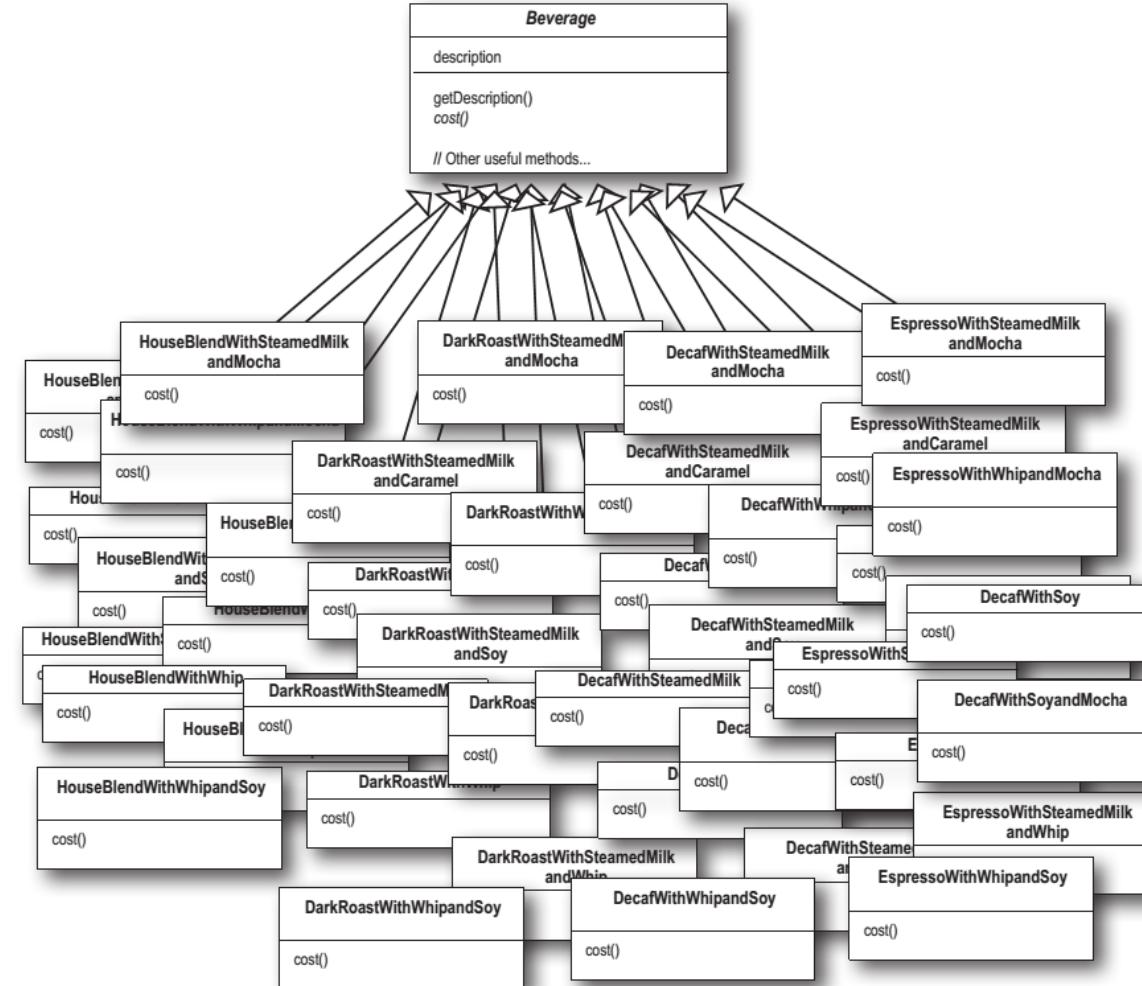
- We'd like to model all the different beverages sold in a cafeteria.
- For the four main beverages, this is easy:



Decorator Pattern

Motivation: Dealing with Variety through Inheritance

- Adding all the different condiments (whip cream, mocha, caramel...) makes things difficult.
- Naïve solution: Just adding subclasses for each variation would lead to a maintenance nightmare:



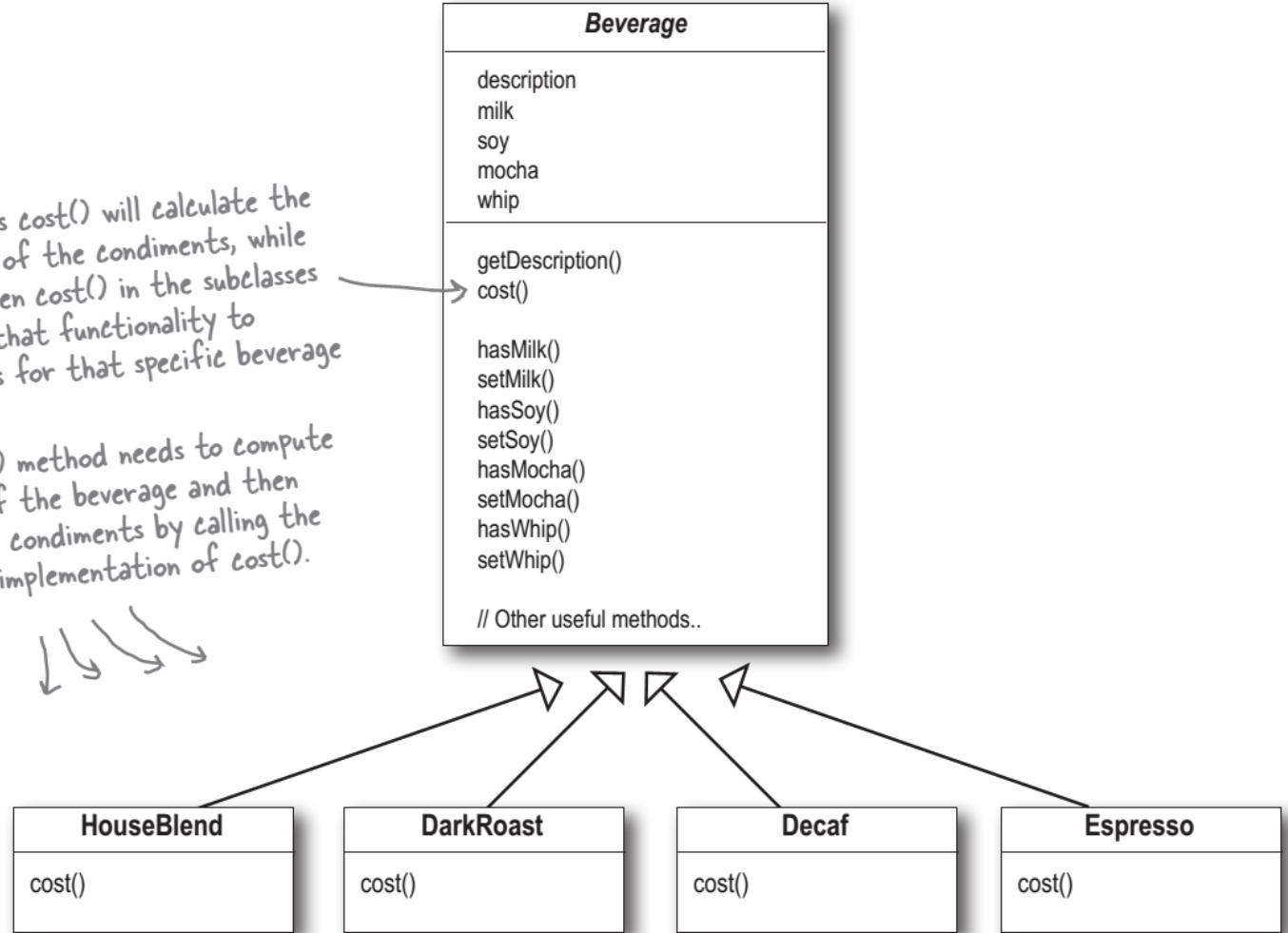
Decorator Pattern

Motivation: Dealing with Variety Through Configuration

- A better way might be to use subclasses only for our main beverages, and make the condiments configurable in the superclass.
- However, now `Beverage.cost()` needs to implement quite complex price calculations, and must be adapted everytime we change prices, introduce new condiments etc.

The superclass `cost()` will calculate the costs for all of the condiments, while the overridden `cost()` in the subclasses will extend that functionality to include costs for that specific beverage type.

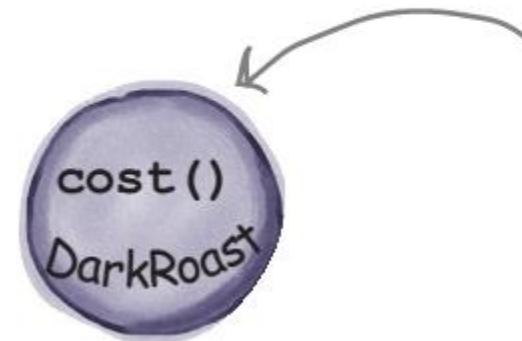
Each `cost()` method needs to compute the cost of the beverage and then add in the condiments by calling the superclass implementation of `cost()`.



Decorator Pattern

Solution: Extending an Object Without Modifying It

- ① We start with our **DarkRoast** object.

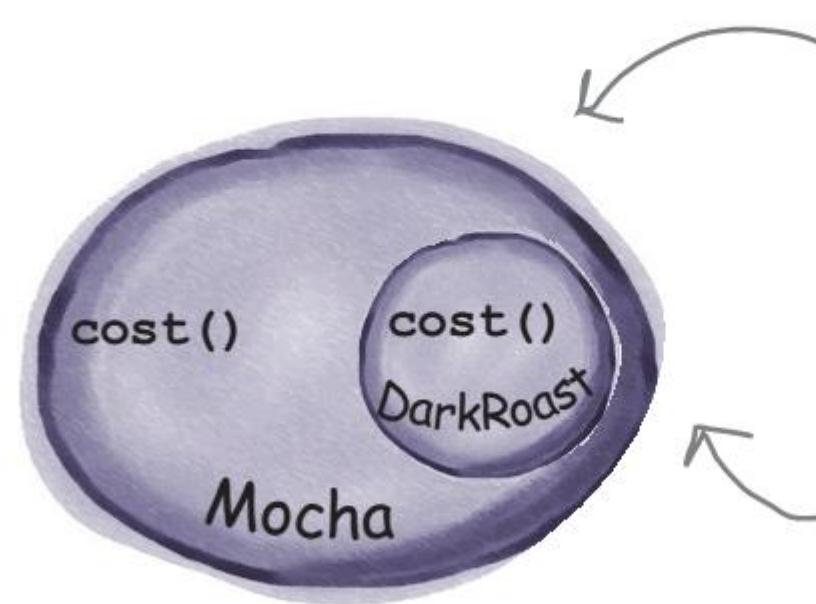


Remember that DarkRoast inherits from Beverage and has a cost() method that computes the cost of the drink.

Decorator Pattern

Solution: Extending an Object Without Modifying It

- ② The customer wants Mocha, so we create a Mocha object and wrap it around the DarkRoast.



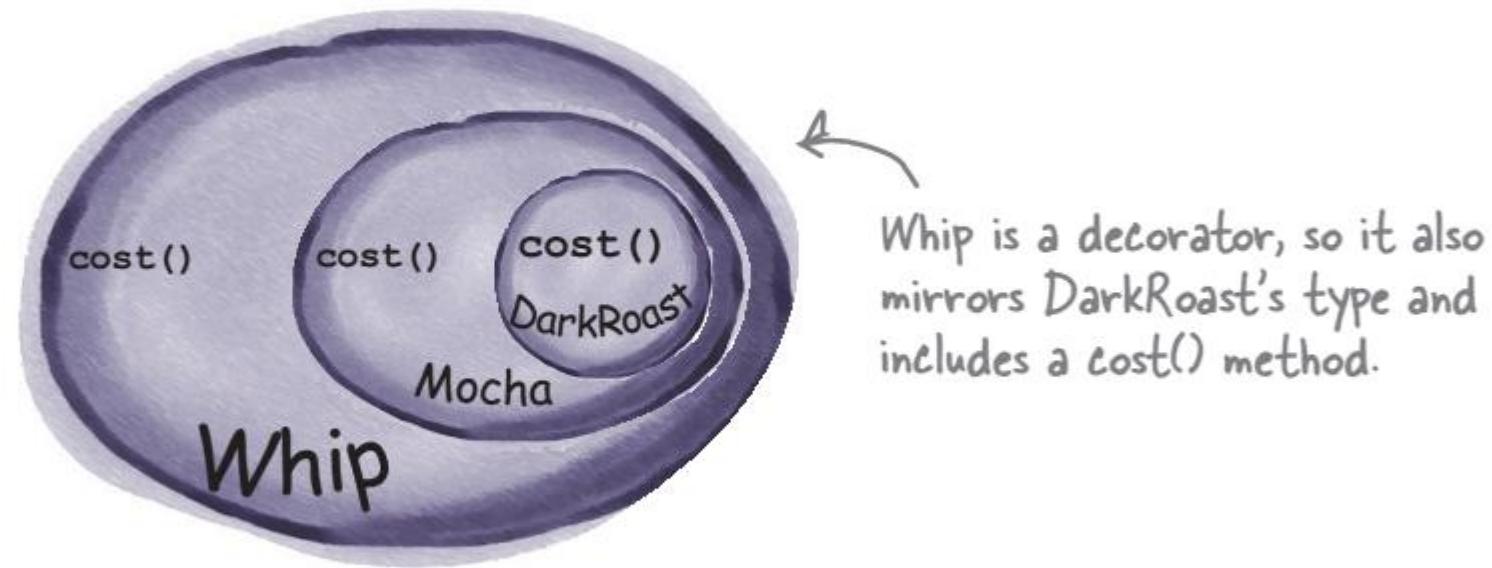
The Mocha object is a decorator. Its type mirrors the object it is decorating, in this case, a Beverage. (By "mirror," we mean it is the same type.)

So, Mocha has a `cost()` method too, and through polymorphism we can treat any Beverage wrapped in Mocha as a Beverage, too (because Mocha is a subtype of Beverage).

Decorator Pattern

Solution: Extending an Object Without Modifying It

- ③ The customer also wants Whip, so we create a Whip decorator and wrap Mocha with it.

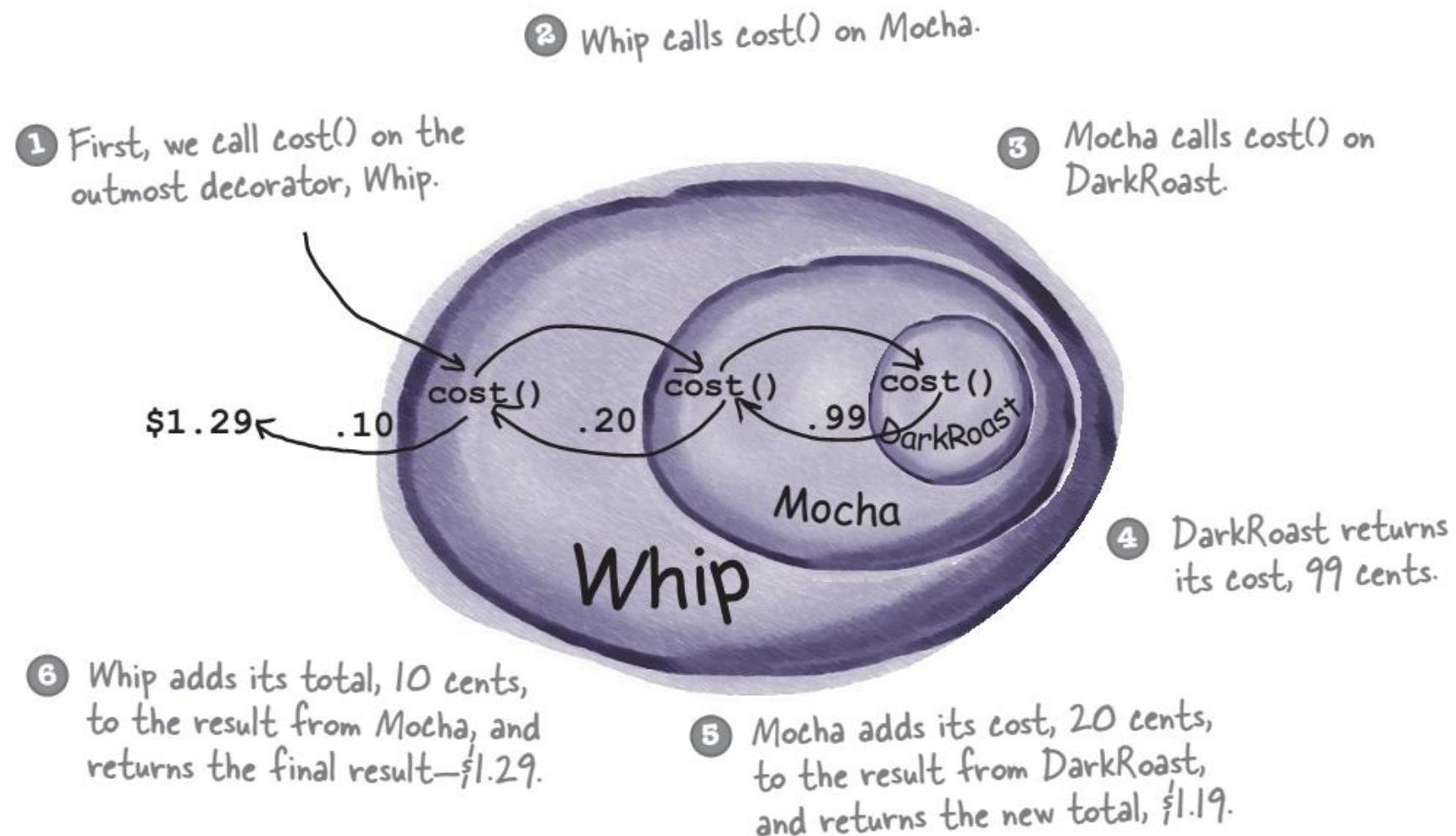


So, a DarkRoast wrapped in Mocha and Whip is still a Beverage and we can do anything with it we can do with a DarkRoast, including call its cost() method.



Decorator Pattern

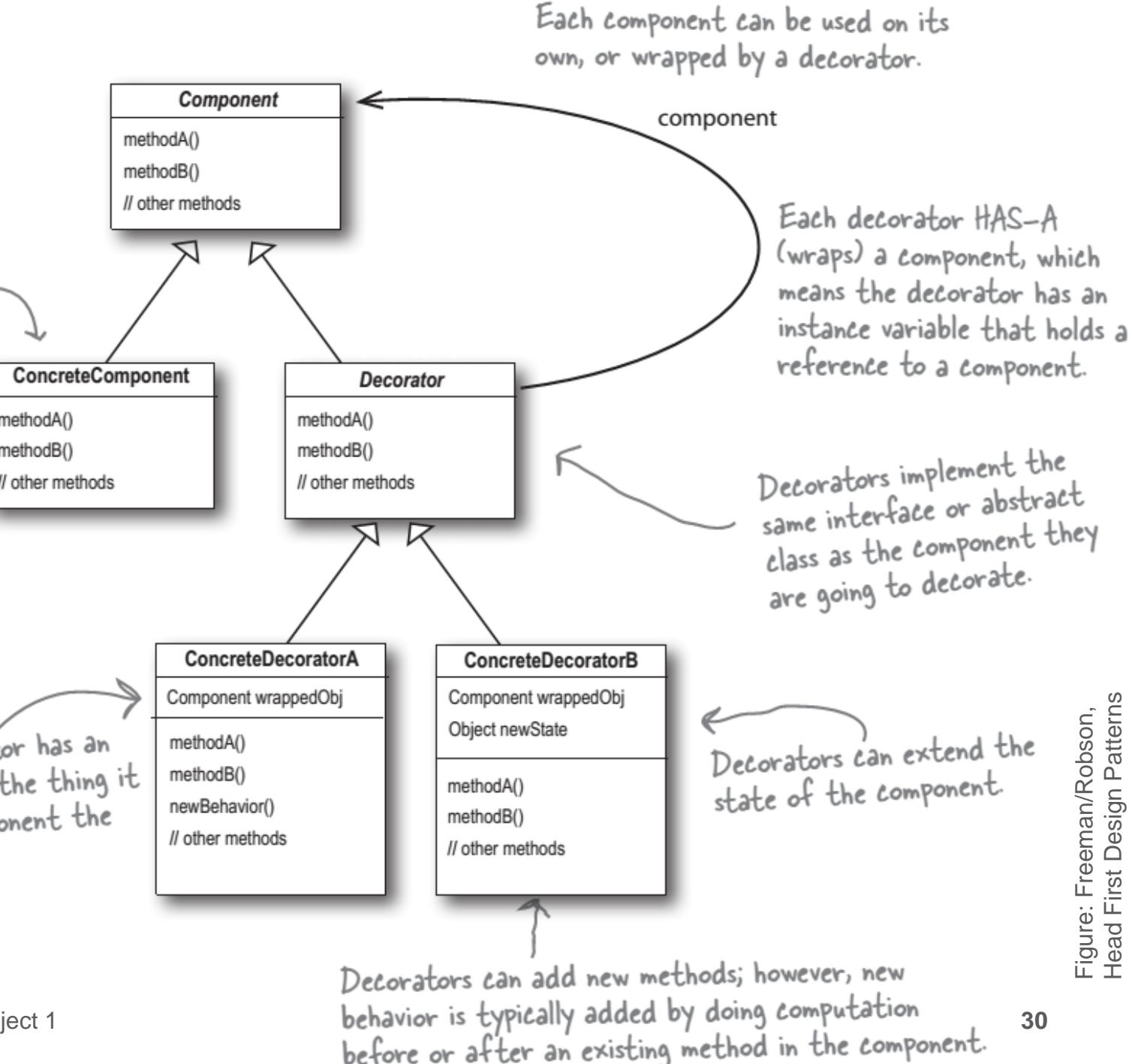
Solution: Extending an Object Without Modifying It



Decorator Pattern General Form

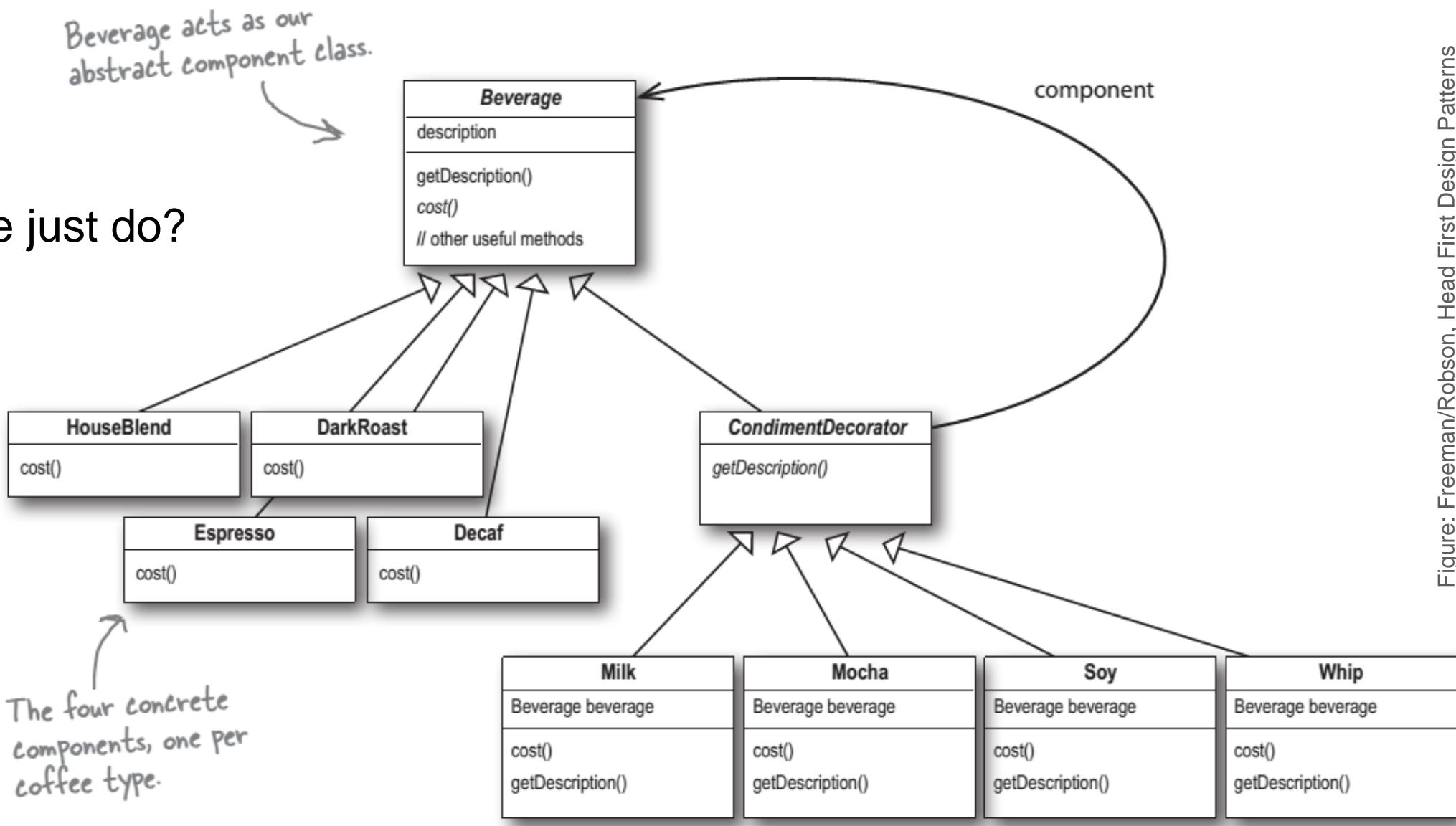
- The Decorator Pattern attaches additional responsibilities to an object dynamically.
- Decorators provide a flexible alternative to subclassing for extending functionality.

The ConcreteComponent is the object we're going to dynamically add new behavior to. It extends Component.



Decorator Example

- What did we just do?



Decorator Pattern Usage in Code

```
File Edit Window Help CloudsInMyCoffee  
% java StarbuzzCoffee  
Espresso $1.99  
Dark Roast Coffee, Mocha, Mocha, Whip $1.49  
House Blend Coffee, Soy, Mocha, Whip $1.34  
%
```

```
public static void main(String args[]) {  
    Beverage beverage = new Espresso();  
    System.out.println(beverage.getDescription()  
        + " $" + beverage.cost());
```

Order up an espresso, no condiments, and print its description and cost.

```
Beverage beverage2 = new DarkRoast();  
beverage2 = new Mocha(beverage2);  
beverage2 = new Mocha(beverage2);  
beverage2 = new Whip(beverage2);  
System.out.println(beverage2.getDescription()  
    + " $" + beverage2.cost());
```

Make a DarkRoast object.

Wrap it with a Mocha.

Wrap it in a second Mocha.

Wrap it in a Whip.

```
Beverage beverage3 = new HouseBlend();  
beverage3 = new Soy(beverage3);  
beverage3 = new Mocha(beverage3);  
beverage3 = new Whip(beverage3);  
System.out.println(beverage3.getDescription()  
    + " $" + beverage3.cost());
```

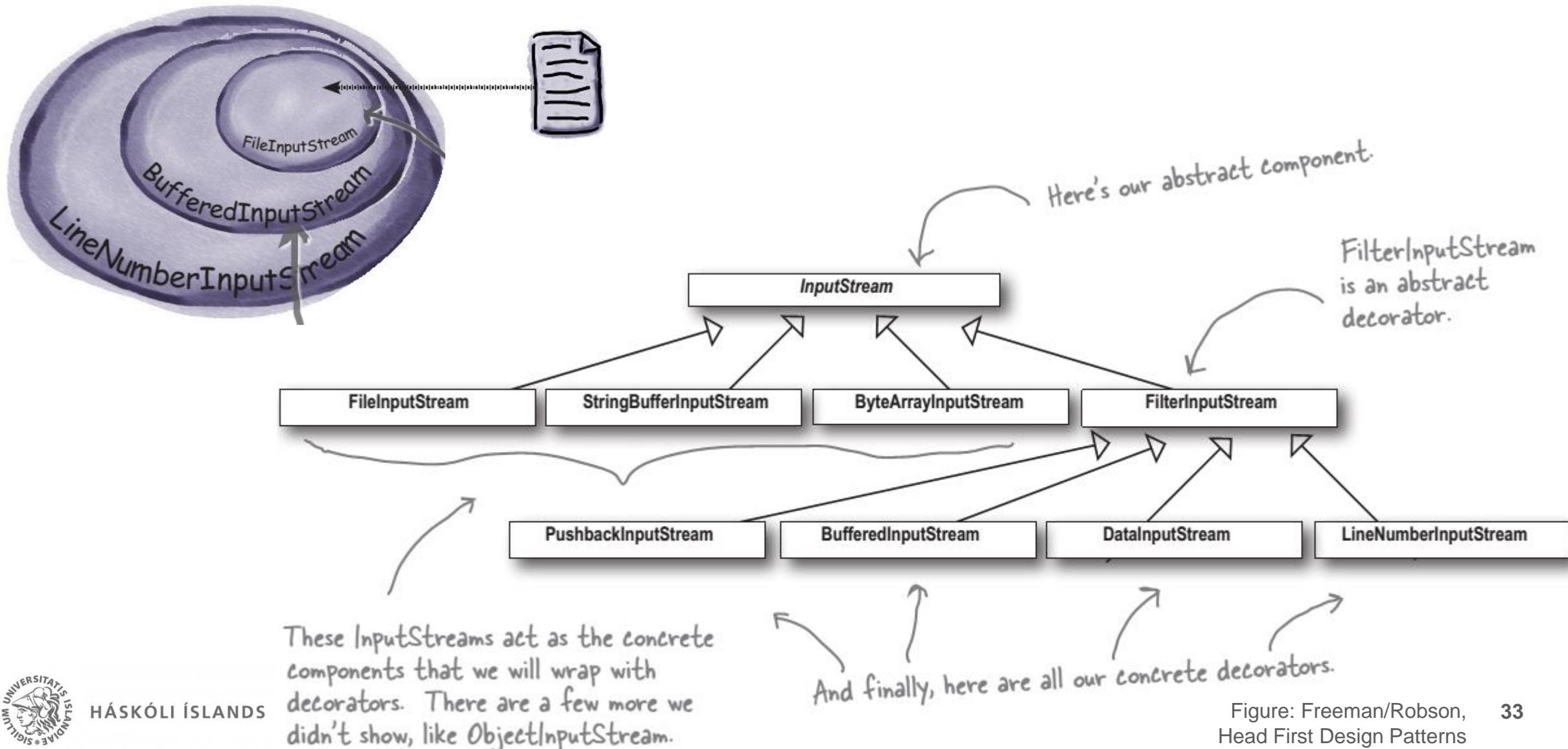
Finally, give us a HouseBlend with Soy, Mocha, and Whip.

```
}
```



Decorator Pattern

Another Example: Streams in Java



Decorator Pattern Benefits

- A decorator looks like a core object and can therefore “wrap around” it without other classes noticing that it’s there.
- This allows us to
 - Extend an object’s behavior
 - Without changing the object’s original implementation
 - And without influencing other classes who expected to work with the core object!
- A possible way to satisfy the Open-Closed Principle (\rightarrow HBV401G, Ch. 7)
- Usage recommendations
 - In many cases, simple inheritance or composition/delegation will be more straightforward
 - For more complex structures with many different variations or ongoing evolution, using the Decorator Pattern can be worthwhile



Command Pattern Motivation

- Imagine we are building a drawing program that lets the user draw and manipulate shapes in many ways
 - Drawing, moving, scaling, rotating, skewing, flipping, filling, smoothing, deleting... any shape
- The number of operations our software offers is likely to evolve over time
- We want to give the user the ability to undo as many actions as desired, i.e. to revert manipulated shapes into previous states
- Thoughts on naïve solutions:
 - Hardwiring all these operations into the software would be tedious but possible
 - Undoing an arbitrary amount of shape changes would require storing a lot of information about previous states of the drawing



Command Pattern Solution

- Turn the individual drawing commands into objects!

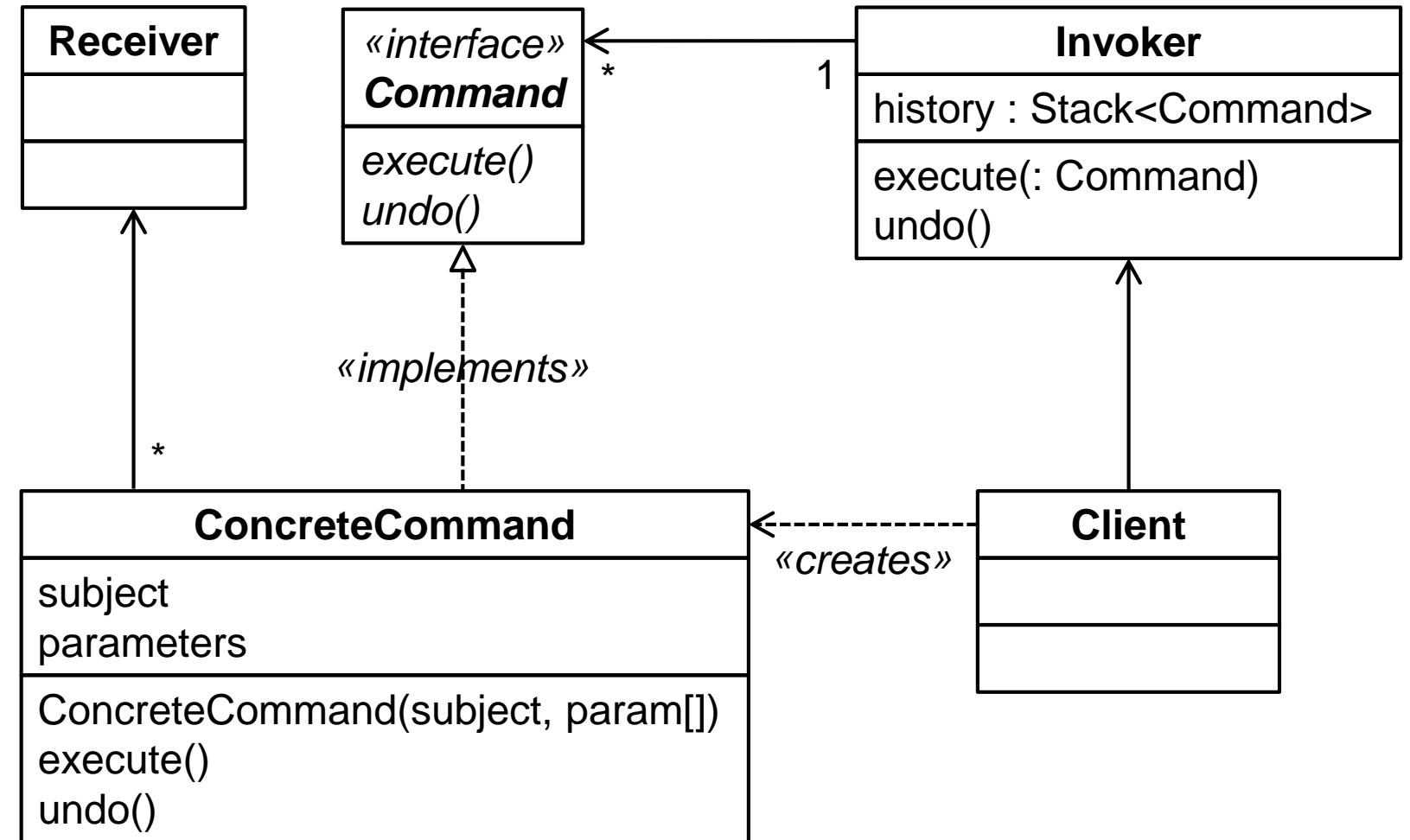
Benefits:

- Commands become conveniently configurable
 - We can even build commands out of commands (macros)
- The set of commands can be extended without changing the command invocation logic
 - e.g. when new features are added to our software
- We can treat commands as data
 - e.g. to keep a history of executed commands on a stack
- We can not just specify how to execute, but also how to revert each command
 - Allows us to implement “undo” functionality quite elegantly

Command Pattern

Generic Example: Command Execution and Reversion

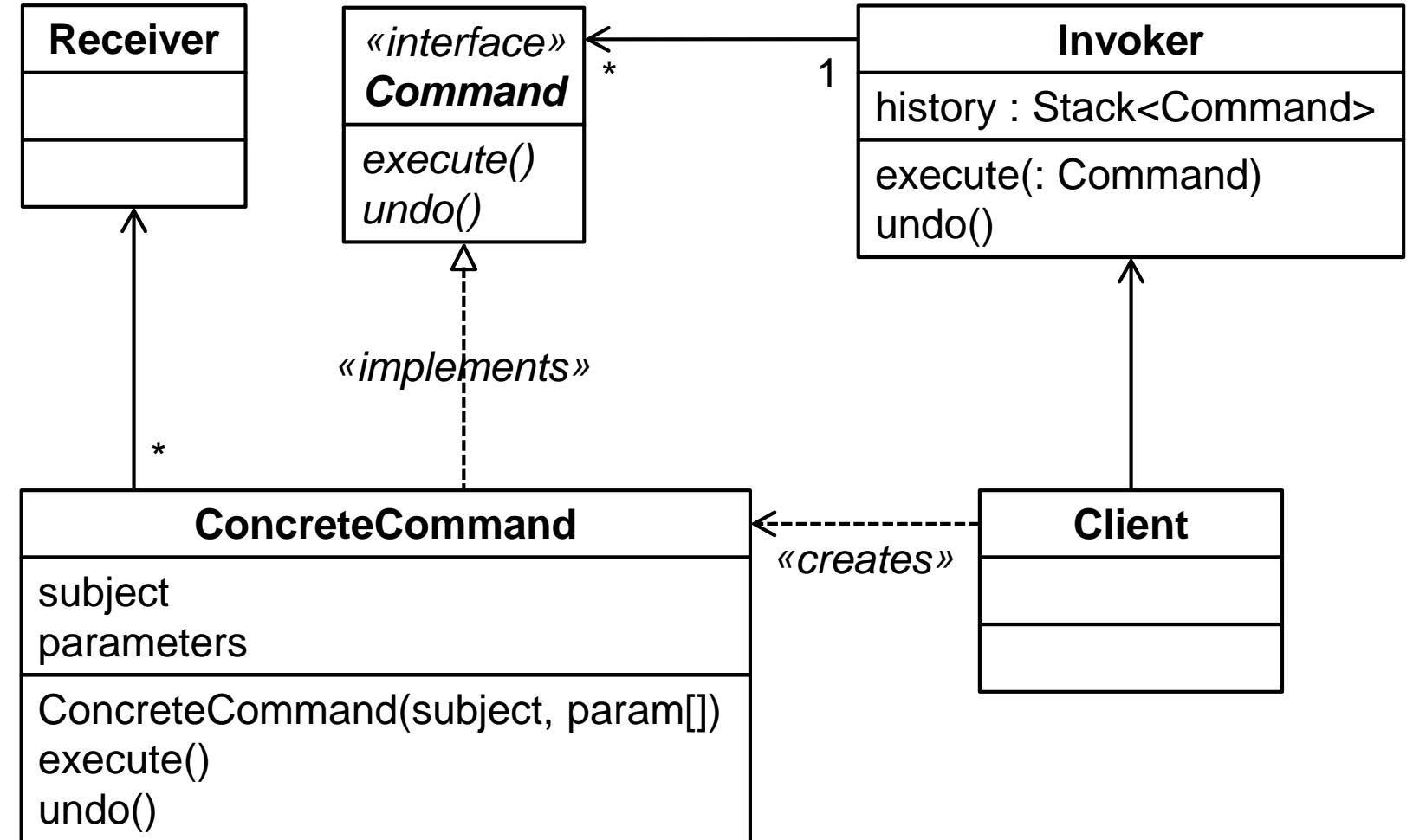
- ConcreteCommands
 - Know which Receiver(s) is/are able to perform an operation
 - here e.g.: a Rotator can rotate a shape
 - Expect a subject to work on...
 - here: the shape
 - ...and parameters describing the operation
 - here e.g.: degrees of rotation



Command Pattern

Generic Example: Command Execution and Reversion

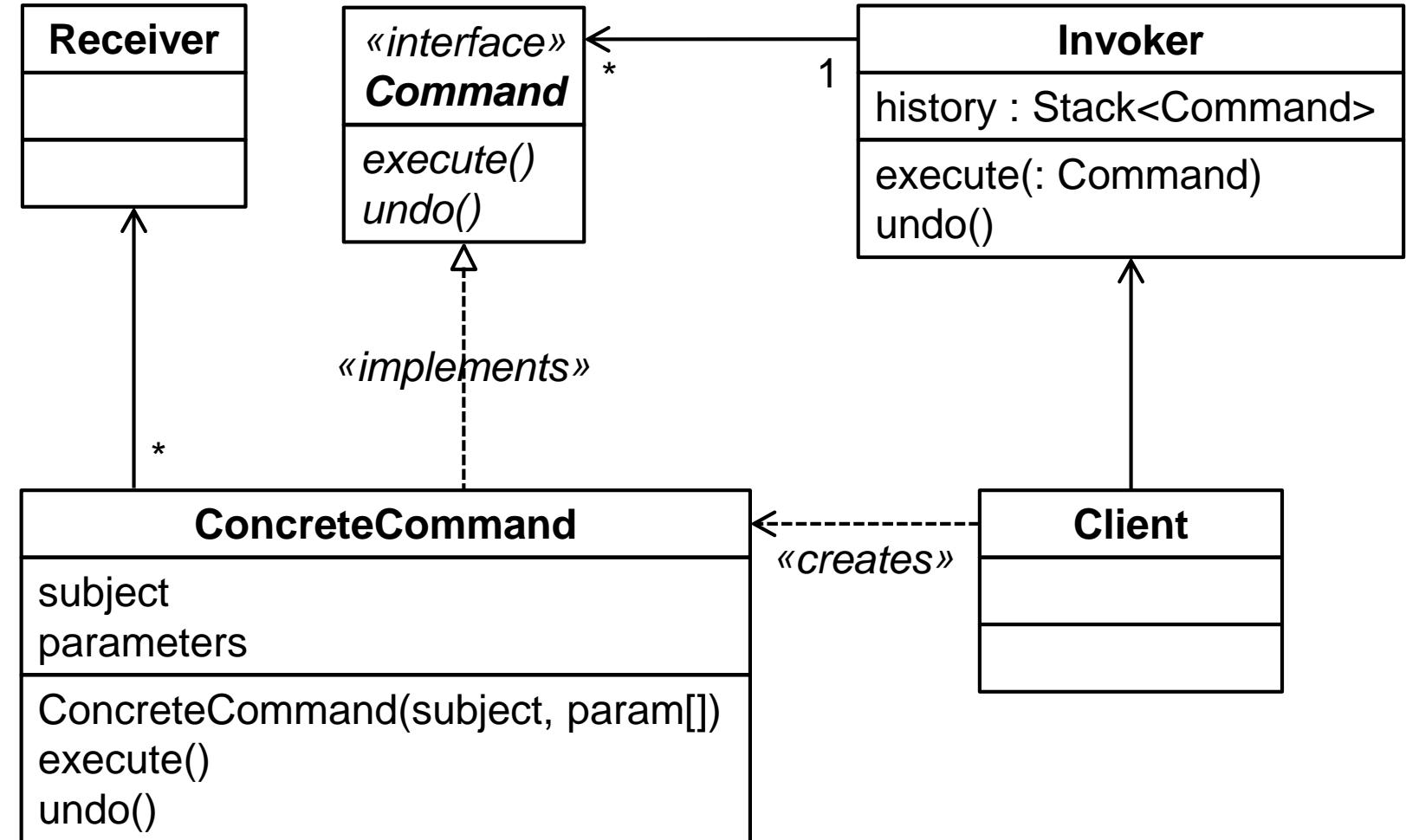
- To execute a command
 - Client instantiates and configures ConcreteCommand
 - Client asks Invoker to execute Command
 - Invoker calls Command's execute() method
 - Command calls the actual implementation of operation in Receiver
 - Invoker pushes Command on stack



Command Pattern

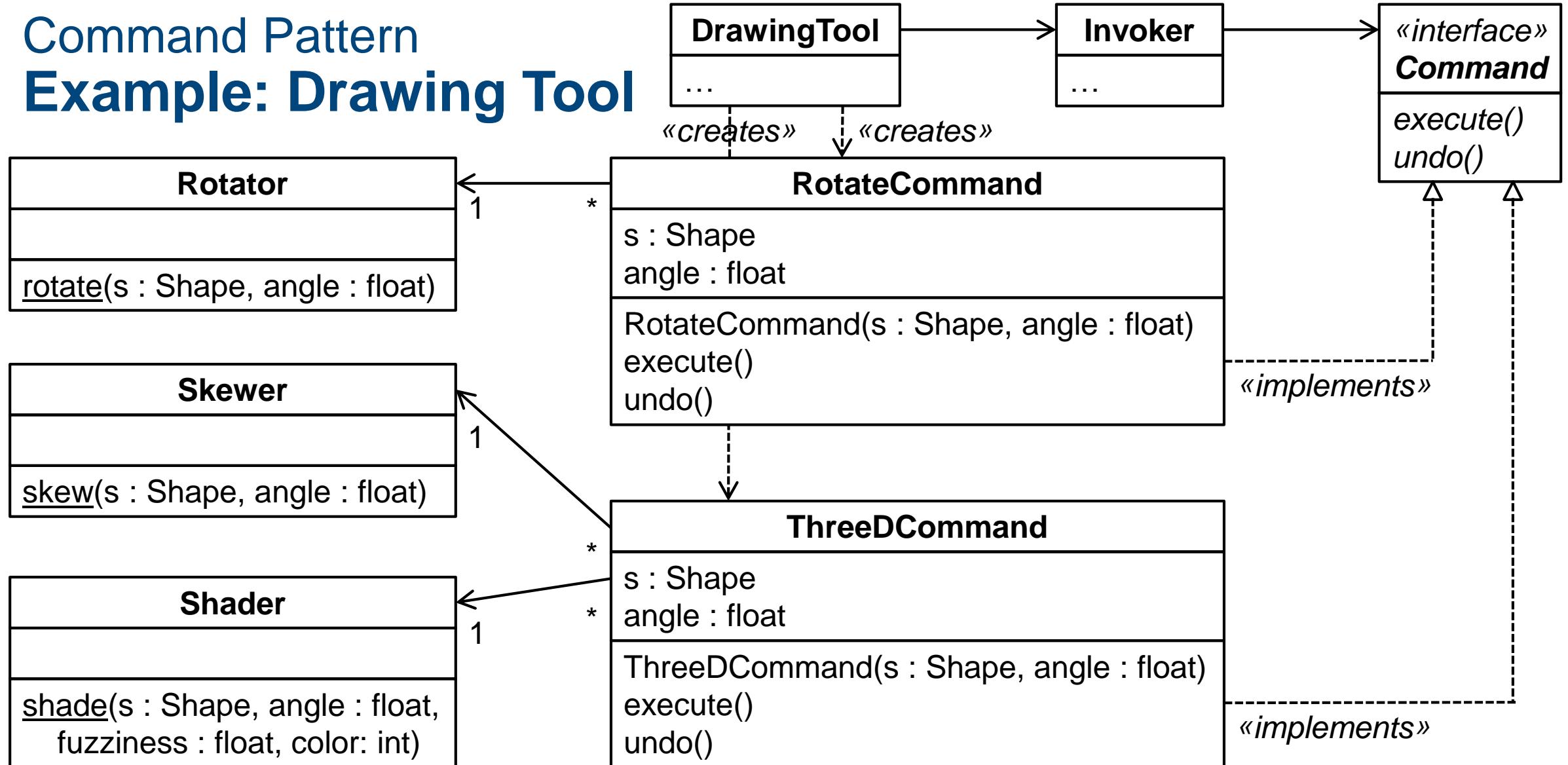
Generic Example: Command Execution and Reversion

- To undo a command
 - Client invokes `Invoker.undo()`
 - Invoker pops top Command off stack and invokes its `undo()` method
 - Undo method uses stored subject and parameter information to construct a call to Receiver with instructions that will negate previous operation's effect



Command Pattern

Example: Drawing Tool



Command Pattern

Example: Command Implementations

```
public class RotateCommand
    implements Command {
    private Shape s; private float angle;
    public RotateCommand(
        Shape s, float angle) {
        this.s = s; this.angle = angle;
    }
    public void execute() {
        Rotator.rotate(s, angle);
    }
    public void undo() {
        Rotator.rotate(s, -angle);
    }
}
```

```
public class ThreeDCommand
    implements Command {
    private Shape s; private float angle;
    public ThreeDCommand(
        Shape s, float angle) {...}
    public void execute() {
        Skewer.skew(s, angle);
        Shader.shade(s, 45, 10, 127);
    }
    public void undo() {
        Shader.clear(s);
        Skewer.skew(s, -angle);
    }
}
```



Command Pattern

Example: DrawingTool and Invoker Implementations

```
public class DrawingTool {  
    private Invoker invoker = new Invoker();  
  
    public testDriver() {  
        Shape s = new Shape(...);  
        // ...draw shape...  
        invoker.execute(  
            new RotateCommand(s, 90));  
        invoker.execute(  
            new ThreeDCommand(s, 20));  
        // ...draw shape...  
        invoker.undo();  
        invoker.undo();  
        // ...draw shape...  
    }  
}
```



```
import java.util.Stack;  
public class Invoker {  
    private Stack<Command> history  
        = new Stack<Command>();  
  
    public execute(Command c) {  
        c.execute();  
        history.push(c);  
    }  
  
    public void undo() {  
        Command c = history.pop();  
        c.undo();  
    }  
}
```

Command Pattern Benefits

- Allows our software to not just perform operations, but be aware of what it's doing, and e.g.
 - Undo operations
 - Facilitate end-user programming
 - Recording and bundling operations in macros
 - Choose between different appropriate operations
- **Usage recommendations**
 - In most cases, simple method calls will be more straightforward
 - Don't be tempted to build your own command language inside the programming language unless you have good reasons to do so (e.g. need for an undo feature)





Hugbúnaðarverkefni 1 / Software Project 1

11. Design Patterns

HBV501G – Fall 2015

Matthias Book



HÁSKÓLI ÍSLANDS
VERKFRÆÐI- OG NÁTTÚRVÍSINDASVIÐ
ÍÐNAÐARVERKFRÆÐI-, VÉLAVERKFRÆÐI-
OG TÖLVUNARFRÆÐIDEILD

Teaching Assistants Wanted

- I am looking for TAs for next semester's courses:
 - **HBV401G Software Development**
 - **HBV601G Software Project 2**
 - If you are interested or know someone who is, please contact me at **book@hi.is** with your CV and overview of grades
- **Tasks**
 - Advising student teams
 - Scoring assignments
 - **Opportunities**
 - Help to shape the course
 - Brush up your CV
 - **Requirements**
 - Successful completion of respective course
 - Ability to advise and evaluate student teams

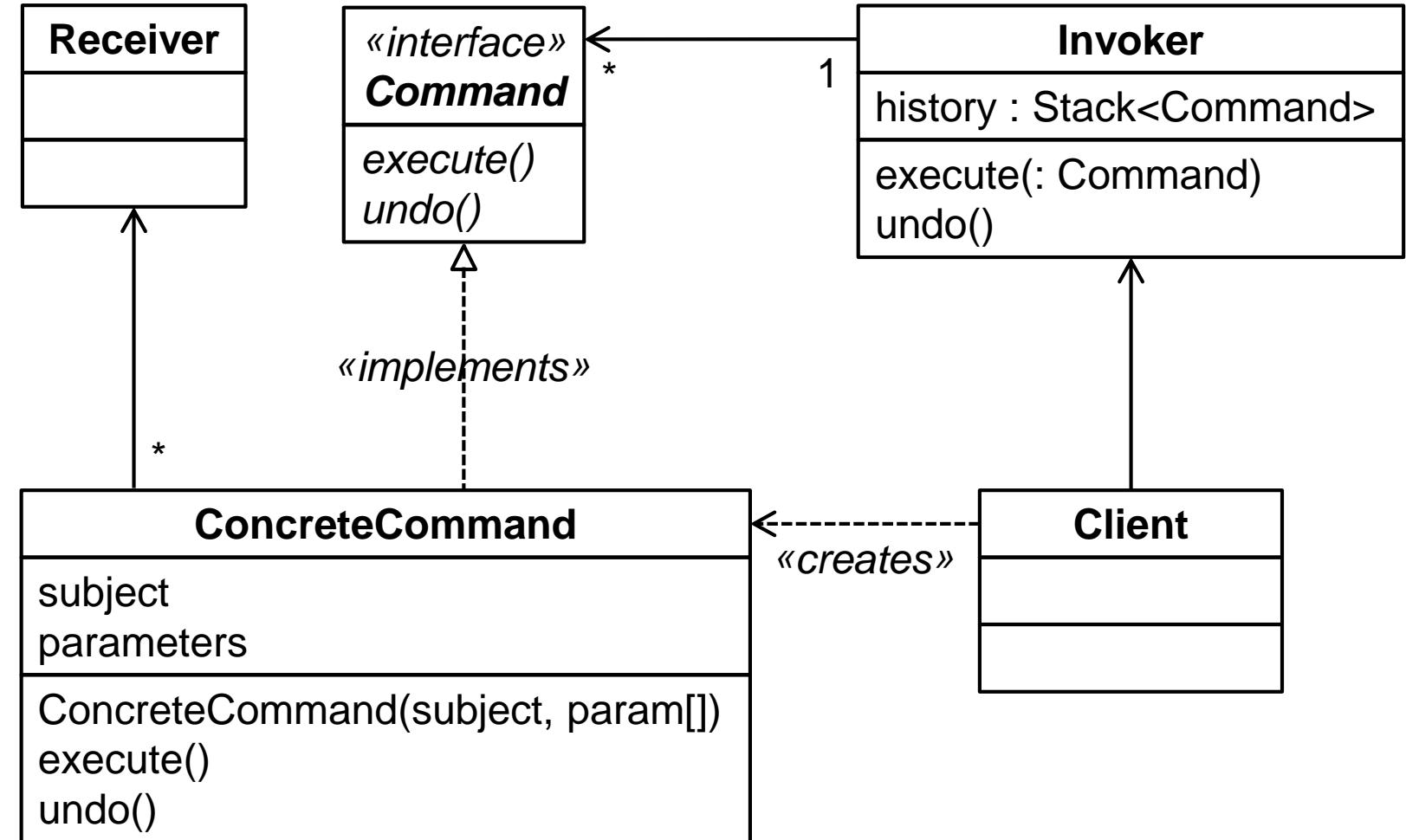


Recap: Design Patterns

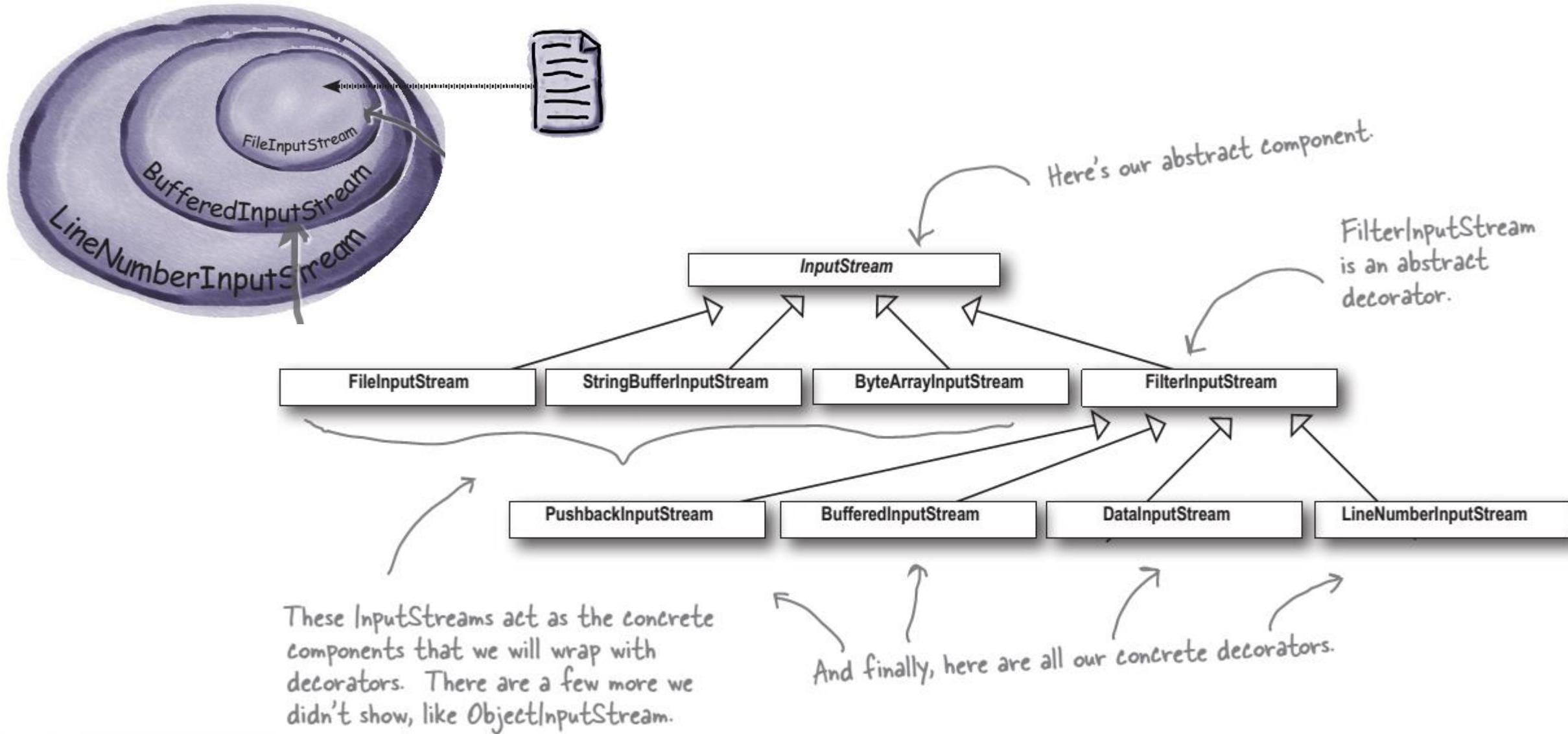
- Solving new problems by applying solutions that have been proven in the past
 - How is the new problem similar to a previous problem?
 - How was the previous problem solved?
 - Is that solution generalizable?
 - How can the general solution be applied to the concrete problem?
- Description of the generic solutions as **patterns**
 - Originally a concept from the domain of architecture, only later applied to software
- Advantages:
 - **Pattern collections** (“pattern languages”) bundle comprehensive domain experience
 - **Explicit description** of characteristics and impacts of design decisions reduces risks
 - **Common vocabulary** simplifies communication and prevents misunderstandings

Recap: Command Pattern

- To execute a command
 - Client instantiates and configures ConcreteCommand
 - Client asks Invoker to execute Command
 - Invoker calls Command's execute() method
 - Command calls the actual implementation of operation in Receiver
 - Invoker pushes Command on stack

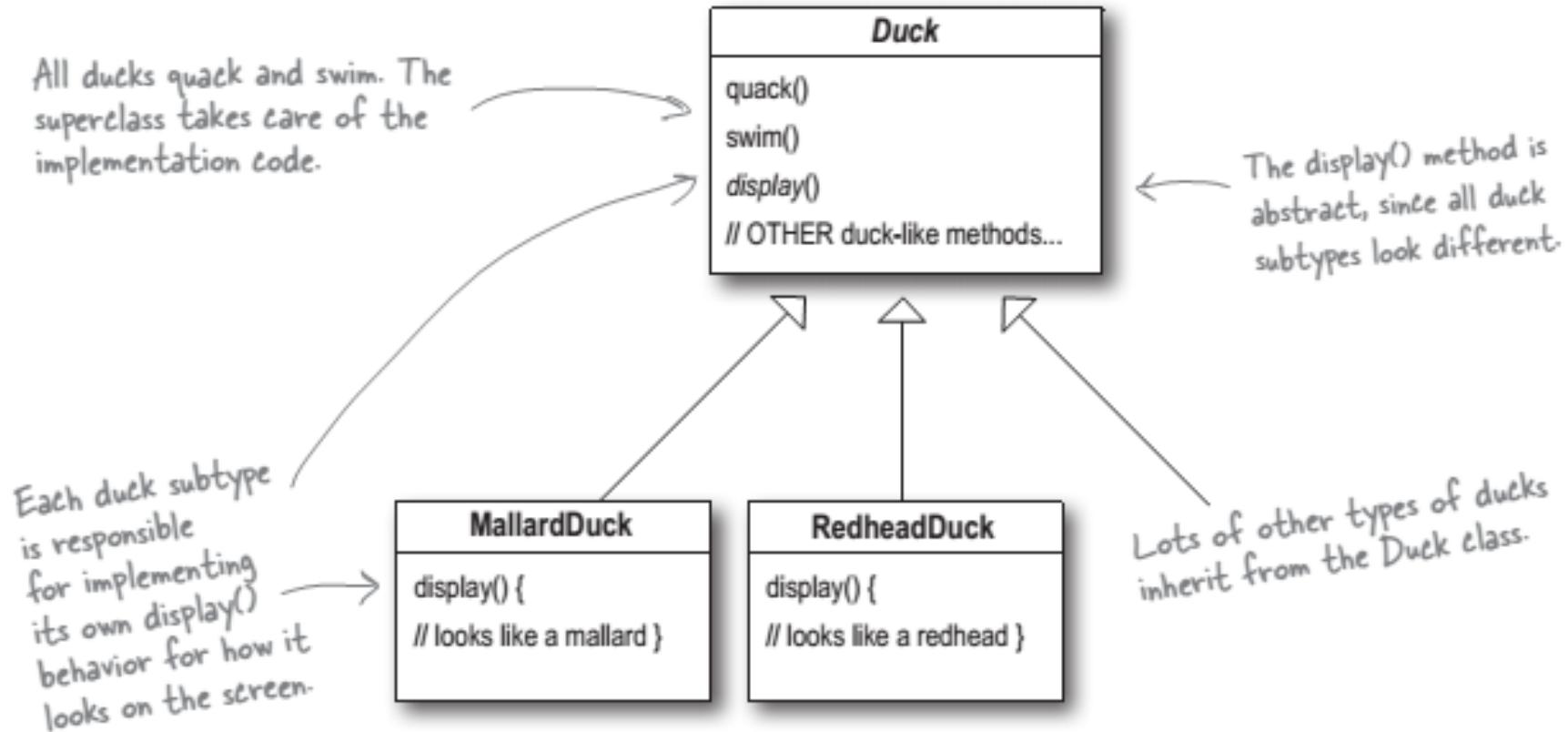


Recap: Decorator Pattern



Strategy Pattern Motivation: Modeling “Similar” Classes

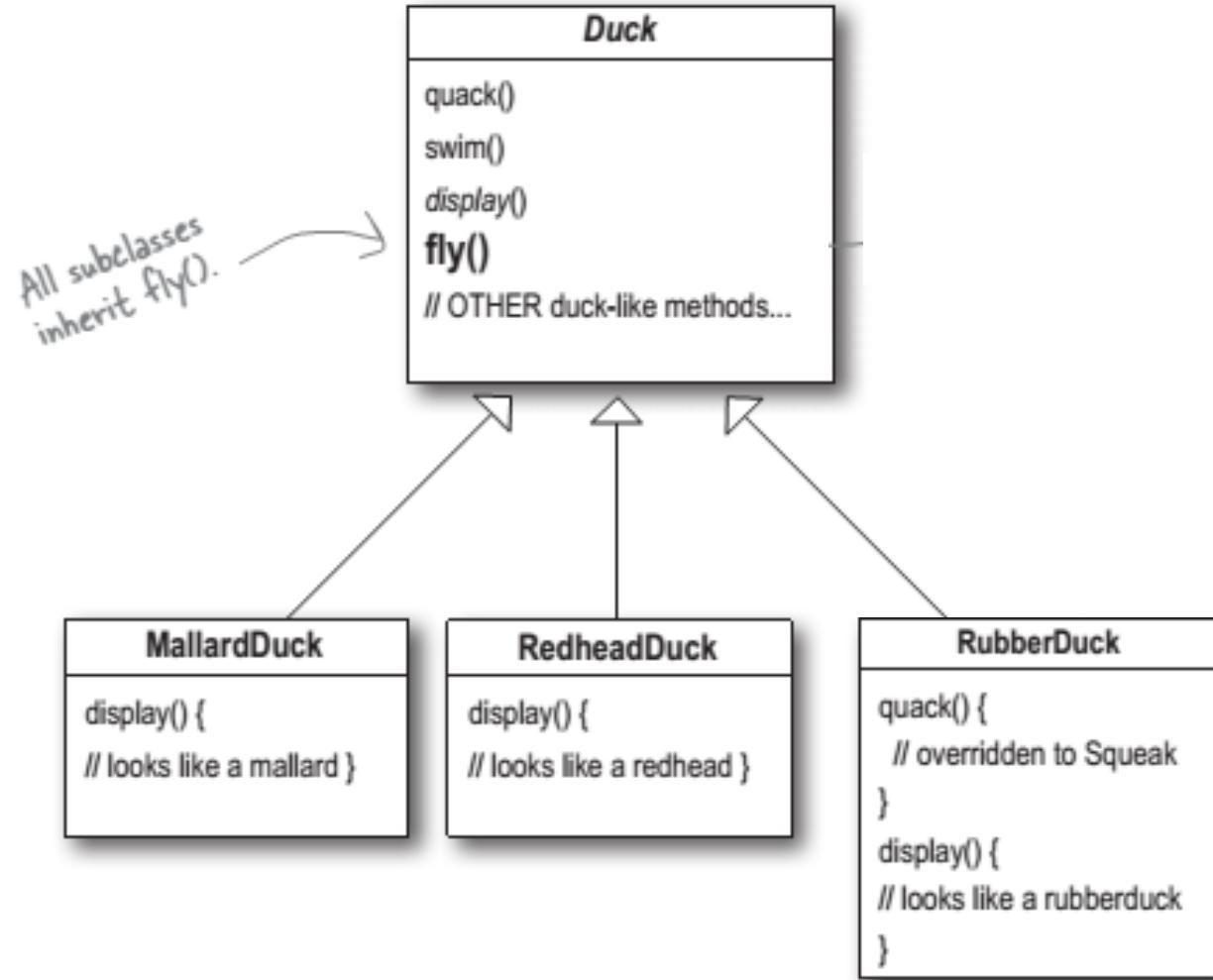
- Imagine we are supposed to simulate the behaviors of different kinds of ducks.
- Our usual strategy: Express generic and specific class properties with inheritance.



Strategy Pattern

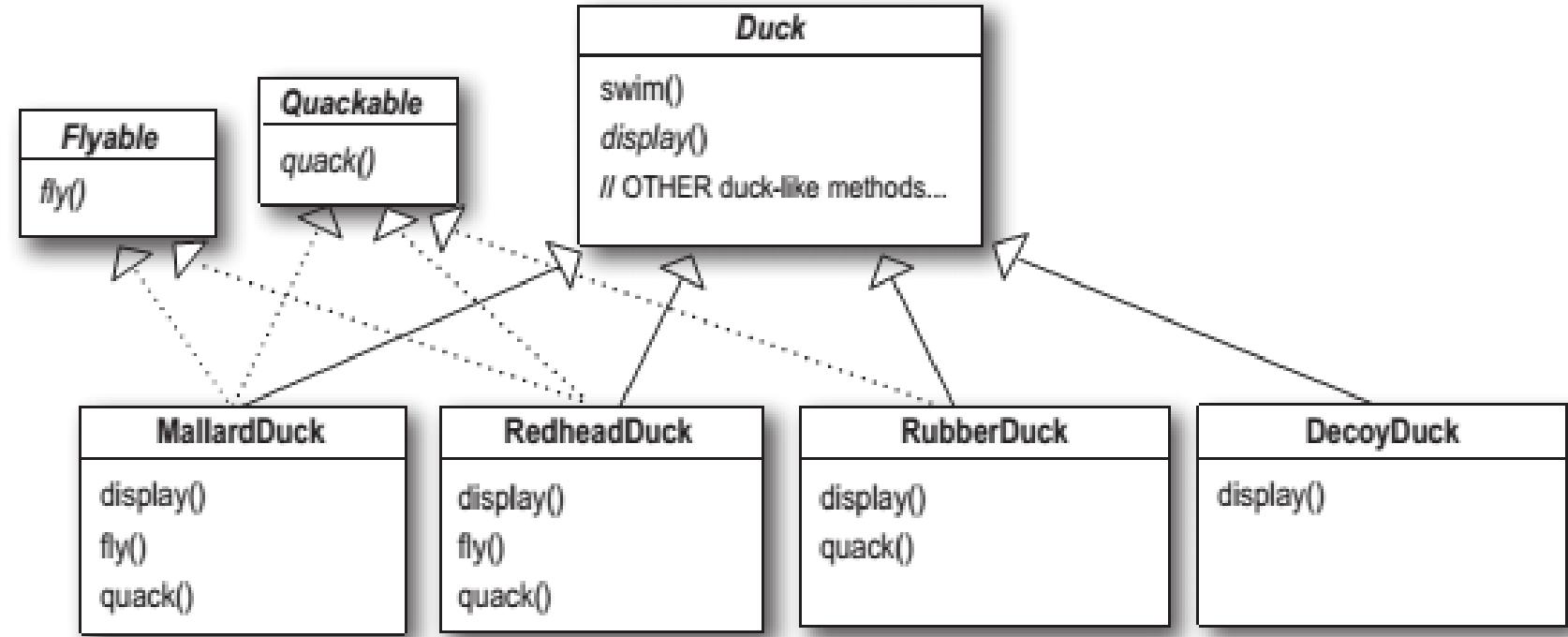
Motivation: Inheritance May Not Be Precise Enough

- If we want to add a new behavior to all subclasses, we just need to add it to the superclass.
 - here e.g.: giving all ducks the ability to fly
- But what if the behavior is not relevant for all the subclasses?
 - here e.g.: some ducks are unable to fly, some are unable to quack



Strategy Pattern Motivation: Interfaces as an Alternative?

- We could
 - pull the properties that are only supported by some subclasses out of the superclass, and
 - model them as interfaces that subclasses can elect to implement
- However, then
 - the subclasses cannot inherit the shared behavior
 - but all need to implement it themselves ☹



The Big Picture

Implementing Shared Behaviors

- **Where to implement shared behaviors to ensure low coupling of classes?**
- In our example:
 - All ducks can **swim**, and they all do it in the same way.
 - The one implementation is placed in the superclass, and inherited by all subclasses.
 - All ducks can be **displayed** in the simulator, but they all look differently.
 - The abstract method in the superclass ensures that all ducks will be displayable, but how they are displayed is implemented individually in each subclass.
 - All ducks can **quack**, but some do it in different ways.
 - The most common implementation is placed in the superclass and inherited by all subclasses, which can override it with specialized implementations if desired.
 - Some ducks can **fly**, but some can not (and should not look like they could).
 - Inheriting from a superclass would confer undesired abilities to some subclasses ☹
 - Implementing an interface would require re-implementing the ability in each eligible subclass ☹
 - Use composition and delegation instead of inheritance to add behavior to classes selectively.

The Big Picture

Encapsulation of Change as a Driver of Good Design

One of the most fundamental modular / object-oriented design principles:

Encapsulate what varies.

- If you can foresee that something will change...
 - an algorithm, a data structure, a technical component, a business process, etc.
 - ...encapsulate that part of your system
 - so other parts of the system will remain unaffected by any internal changes.
 - Various encapsulation techniques exist on different levels:
 - **Programming language:** Methods, classes, visibilities
 - **Class structure:** Abstract supertypes (i.e. abstract classes or interfaces), polymorphism
 - **Class collaboration:** Object-oriented design patterns
 - **System architecture:** Component-based design, layered architectures, communication protocols



The Big Picture

Encapsulation of Change as a Driver of Good Design

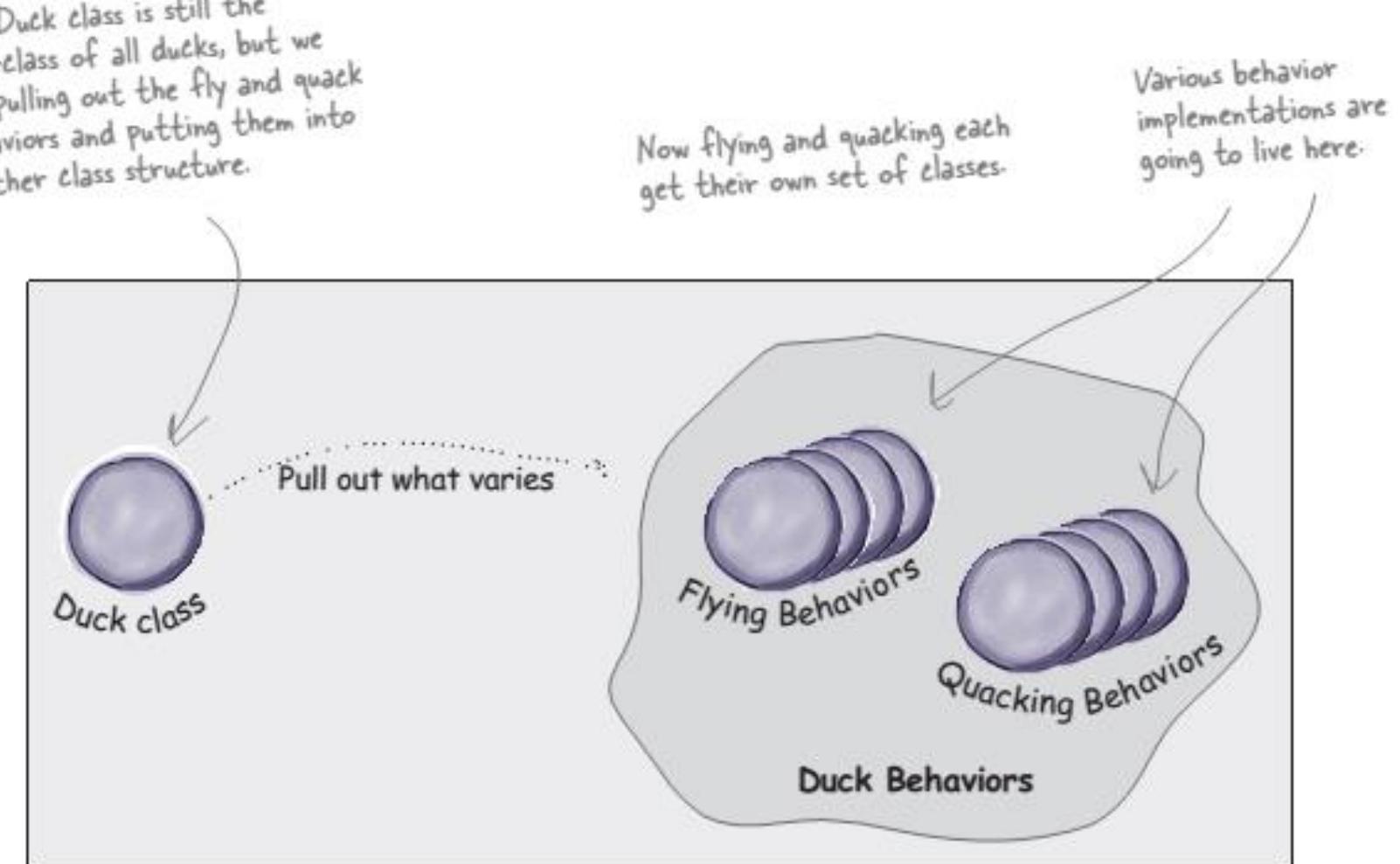
Two important corollaries to the encapsulation principle:

- **As a module user: Program to an interface, not an implementation.**
 - When you are working with something encapsulated, treat it as a black box and make no assumptions about how it works inside.
 - i.e. do not rely on knowledge/assumptions about things like data structures, sorting orders, side effects, thread-safety etc., as they may change (unless they are explicitly specified)
- **As a module provider:**
You can change an implementation, but never an interface.
 - Changing an interface will break any outside code relying on it.
 - Changes include adding abstract methods to supertypes (i.e. classes or Java interfaces), changing method signatures, and even changing a method's documentation if it promises certain properties such as sorting order, thread-safety etc.



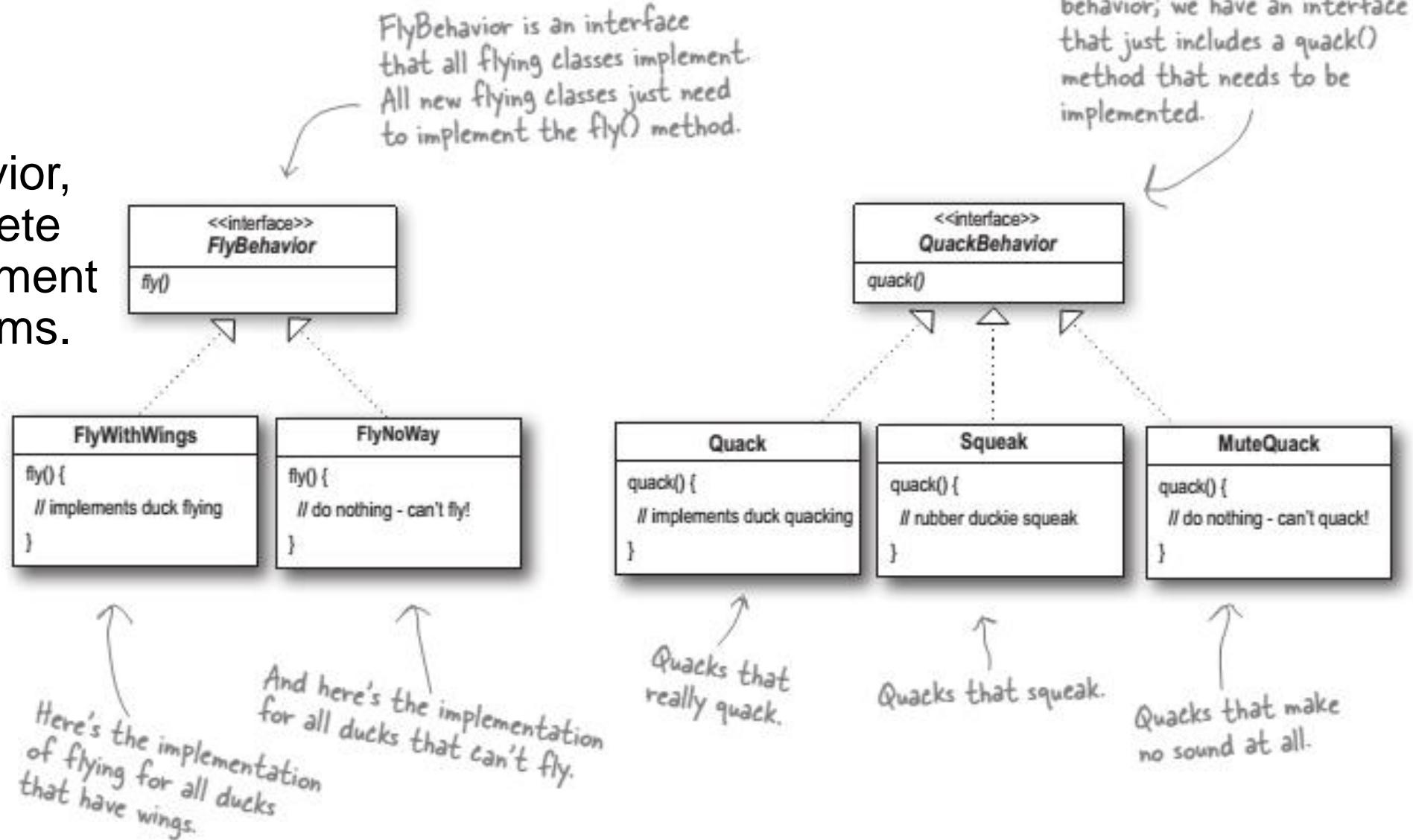
Strategy Pattern Solution: Encapsulating What Varies

- The behaviors that vary between duck types are pulled into individual classes encapsulating each behavior.



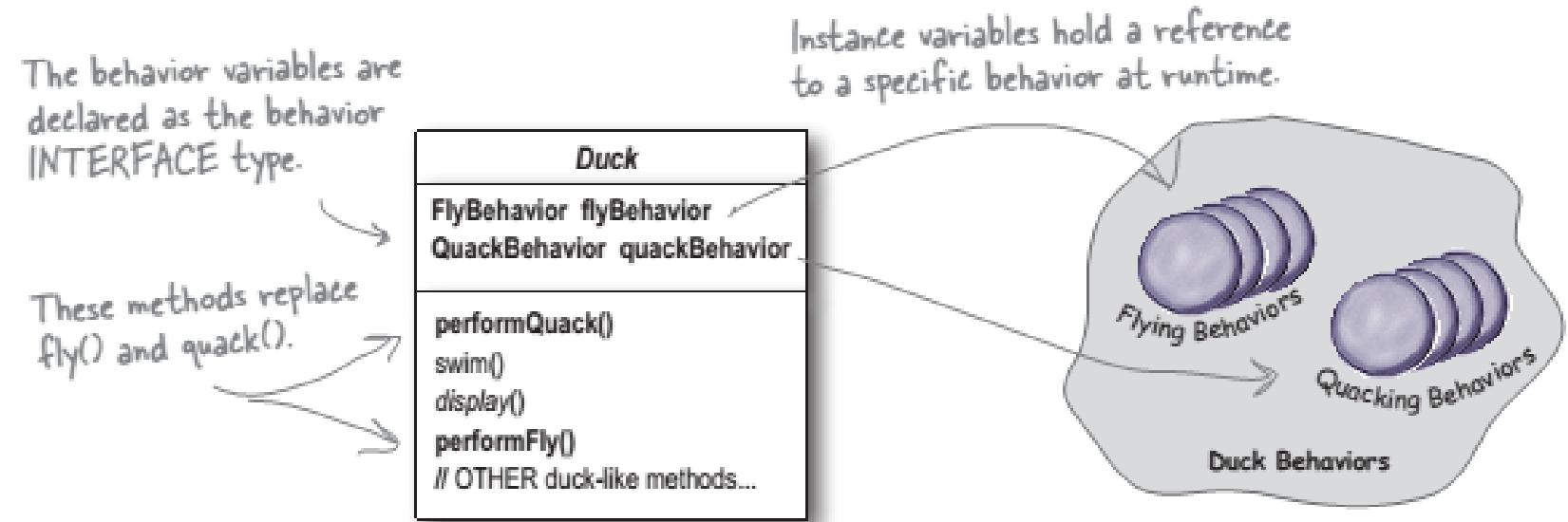
Strategy Pattern Solution: Enable Programming to Interfaces

- The interface describes the general behavior, and the concrete classes implement its specific forms.



Strategy Pattern Solution: Program to Interfaces

- Duck instances can now **delegate** the actual flying and quacking behavior to specialized objects implementing particular variations of that behavior.
- And we can even configure which behavior to use at runtime!



Strategy Pattern Example Implementation

- Implementation of specific behavior classes

```
public interface FlyBehavior {  
    public void fly();  
}
```

The interface that all flying behavior classes implement

```
public class FlyWithWings implements FlyBehavior {  
    public void fly() {  
        System.out.println("I'm flying!!");  
    }  
}
```

Flying behavior implementation for ducks that DO fly...

```
public class FlyNoWay implements FlyBehavior {  
    public void fly() {  
        System.out.println("I can't fly");  
    }  
}
```

Flying behavior implementation for ducks that do NOT fly (like rubber ducks and decoy ducks).

```
public interface QuackBehavior {  
    public void quack();  
}
```

```
public class Quack implements QuackBehavior {  
    public void quack() {  
        System.out.println("Quack");  
    }  
}
```

```
public class MuteQuack implements QuackBehavior {  
    public void quack() {  
        System.out.println("<< Silence >>");  
    }  
}
```

```
public class Squeak implements QuackBehavior {  
    public void quack() {  
        System.out.println("Squeak");  
    }  
}
```



Strategy Pattern Example Implement.

- Implementation of generic client class

```
public abstract class Duck {  
  
    FlyBehavior flyBehavior;  
    QuackBehavior quackBehavior;  
    public Duck() {  
    }  
  
    public void setFlyBehavior(FlyBehavior fb) {  
        flyBehavior = fb;  
    }  
  
    public void setQuackBehavior(QuackBehavior qb) {  
        quackBehavior = qb;  
    }  
  
    public abstract void display();  
  
    public void performFly() {  
        flyBehavior.fly();  
    }  
  
    public void performQuack() {  
        quackBehavior.quack();  
    }  
  
    public void swim() {  
        System.out.println("All ducks float, even decoys!");  
    }  
}
```

Declare two reference variables for the behavior interface types. All duck subclasses (in the same package) inherit these.

Delegate to the behavior class.



Strategy Pattern Example Implementation

- Implementation and configuration of a particular client type

```
public class MallardDuck extends Duck {
```

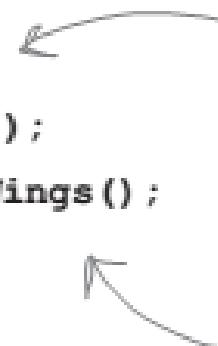
```
    public MallardDuck() {
```

```
        quackBehavior = new Quack();
```

```
        flyBehavior = new FlyWithWings();
```

```
}
```

Remember, MallardDuck inherits the quackBehavior and flyBehavior instance variables from class Duck.



A MallardDuck uses the Quack class to handle its quack, so when performQuack() is called, the responsibility for the quack is delegated to the Quack object and we get a real quack.

And it uses FlyWithWings as its FlyBehavior type.

```
    public void display() {
```

```
        System.out.println("I'm a real Mallard duck");
```

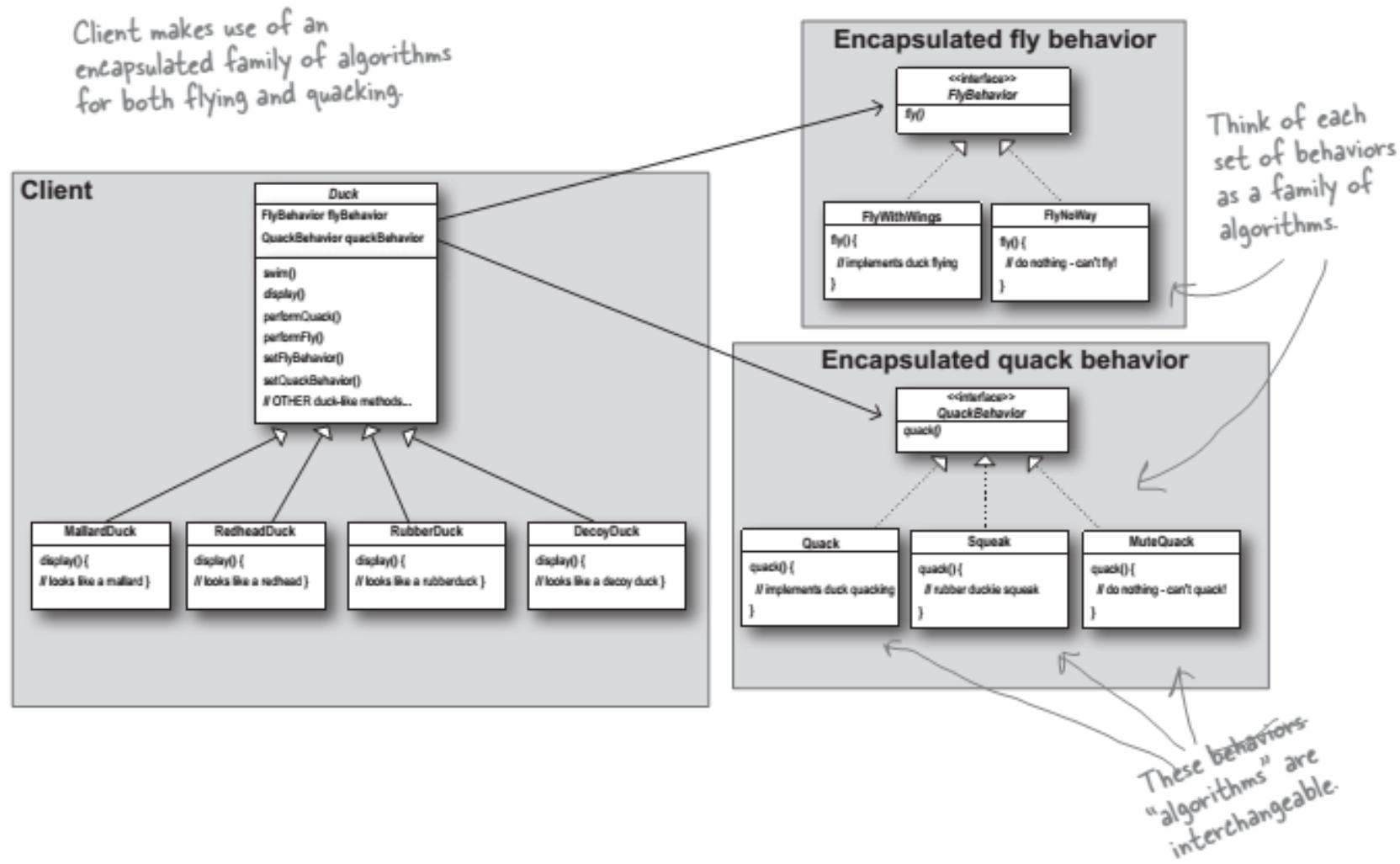
```
}
```

```
}
```



Strategy Pattern Summary

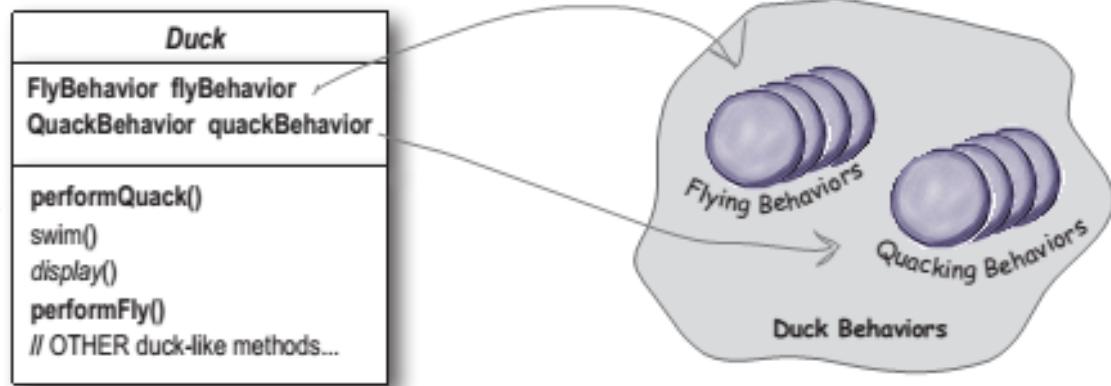
- The **Strategy pattern**
 - defines a family of algorithms,
 - encapsulates each one,
 - and makes them interchangeable.
- The Strategy pattern lets the algorithm vary independently from clients that use it.



Strategy or Decorator Pattern?

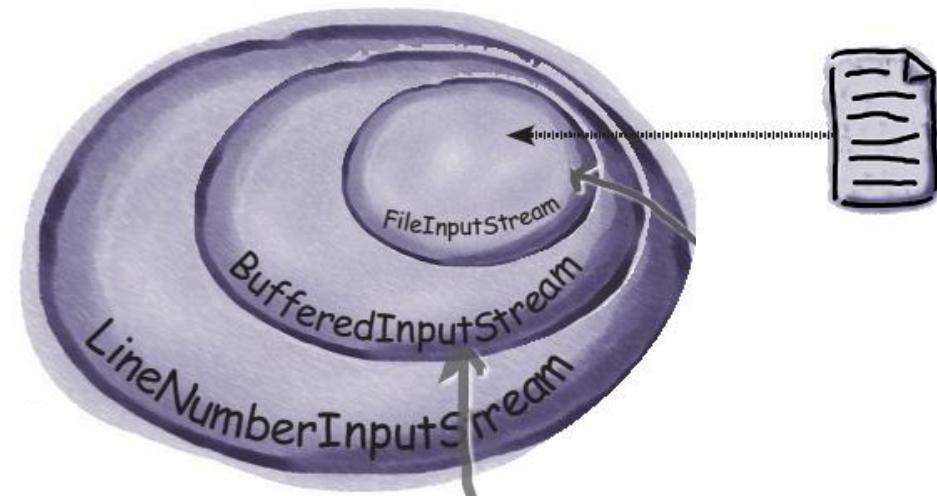
Strategy Pattern

- Useful when a class knows that it'll have to implement certain capabilities
- but wants to remain unaware of how those capabilities will be implemented



Decorator Pattern

- Useful when a class should not need to be aware that its behavior will be extended in an unknown variety of ways by other classes



State Pattern

Background: State in Software Systems

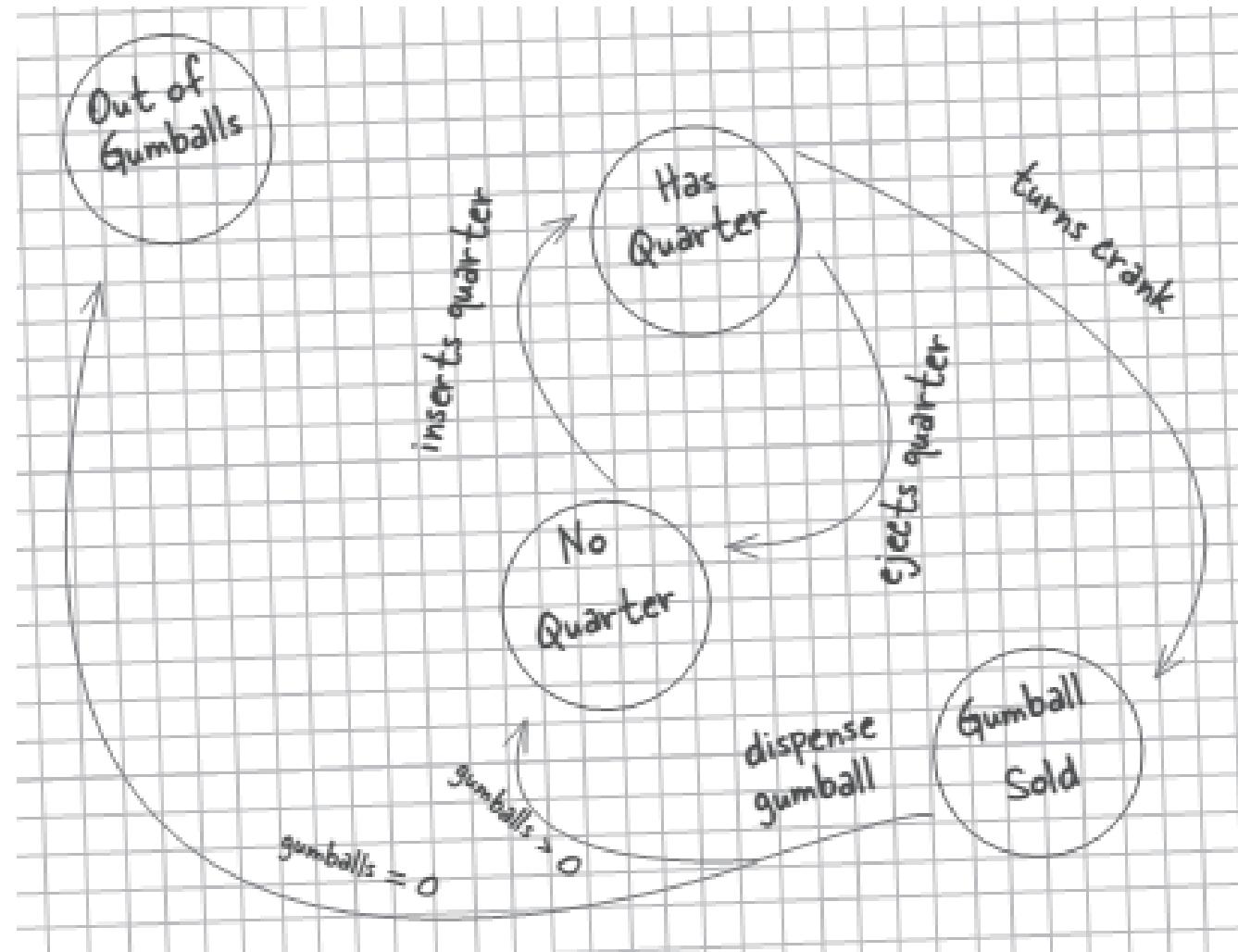
- State can be found on all levels of software systems, e.g.
 - the state of individual objects
 - the state of the whole system
 - the state of a particular business process
 - etc.
- Some states are easily represented in simple data structures
 - e.g. the score of a player in a game
- Some states permeate the system without being explicitly “stored” anywhere
 - e.g. the view that a user has most recently navigated to
- Some states are explicitly stored and influence a system’s or object’s behavior
 - e.g. whether a user is currently logged into a system, and which rights that user has
 - e.g. how a system is reacting to outside events under different conditions
 - e.g. which capabilities/behaviors certain entities exhibit in a game



State Pattern Motivation: Example



- Modeling the state transitions of a gumball machine



State Pattern Motivation: Impl.

Simple approach:

- Represent the states as integer values
- Represent the events as methods
- Implement actions and decide on subsequent states conditionally...

```
public class GumballMachine {  
  
    final static int SOLD_OUT = 0;  
    final static int NO_QUARTER = 1;  
    final static int HAS_QUARTER = 2;  
    final static int SOLD = 3;  
  
    int state = SOLD_OUT;  
    int count = 0;  
  
    public GumballMachine(int count) {  
        this.count = count;  
        if (count > 0) {  
            state = NO_QUARTER;  
        }  
    }  
    Now we start implementing  
    the actions as methods...  
  
    public void insertQuarter() {  
        if (state == HAS_QUARTER) {  
            System.out.println("You can't insert another quarter");  
        } else if (state == NO_QUARTER) {  
            state = HAS_QUARTER;  
            System.out.println("You inserted a quarter");  
        } else if (state == SOLD_OUT) {  
            System.out.println("You can't insert a quarter, the machine is sold out");  
        } else if (state == SOLD) {  
            System.out.println("Please wait, we're already giving you a gumball");  
        }  
    }  
    If the customer just bought a  
    gumball he needs to wait until the  
    transaction is complete before  
    inserting another quarter.  
}  
Here's the instance variable that is going  
to keep track of the current state we're  
in. We start in the SOLD_OUT state.  
We have a second instance variable that  
keeps track of the number of gumballs  
in the machine.  
The constructor takes an initial inventory  
of gumballs. If the inventory isn't zero,  
the machine enters state NO_QUARTER,  
meaning it is waiting for someone to  
insert a quarter, otherwise it stays in  
the SOLD_OUT state.  
When a quarter is inserted, if....  
...a quarter is already  
inserted we tell the  
customer...  
...otherwise we accept the  
quarter and transition to  
the HAS_QUARTER state.  
And if the machine is sold  
out, we reject the quarter.
```



State Pattern Motivation: Implementation

Simple approach:

- Represent the states as integer values
- Represent the events as methods
- Implement actions and decide on subsequent states conditionally...
- ...in a complex structure for each possible combination of state and event ☹

```
public void ejectQuarter() {  
    if (state == HAS_QUARTER) { ↗ Now, if the customer tries to remove the quarter...  
        System.out.println("Quarter returned");  
        state = NO_QUARTER;  
    } else if (state == NO_QUARTER) { ↗ If there is a quarter, we return it and go back to the NO_QUARTER state.  
        System.out.println("You haven't inserted a quarter");  
    } else if (state == SOLD) {  
        System.out.println("Sorry, you already turned the crank");  
    } else if (state == SOLD_OUT) {  
        System.out.println("You can't eject, you haven't inserted a quarter yet");  
    }  
}  
} ↗ You can't eject if the machine is sold out, it doesn't accept quarters!  
} ↗ Otherwise, if there isn't one we can't give it back.  
  
public void turnCrank() {  
    if (state == SOLD) {  
        System.out.println("Turning twice doesn't get you another gumball!");  
    } else if (state == NO_QUARTER) {  
        System.out.println("You turned but there's no quarter");  
    } else if (state == SOLD_OUT) {  
        System.out.println("You turned, but there are no gumballs");  
    } else if (state == HAS_QUARTER) {  
        System.out.println("You turned...");  
        state = SOLD;  
        dispense();  
    }  
} ↗ The customer tries to turn the crank...  
} ↗ Someone's trying to cheat the machine.  
} ↗ We need a quarter first.  
} ↗ We can't deliver gumballs; there are none.  
} ↗ Success! They get a gumball. Change the state to SOLD and call the machine's dispense() method.  
  
public void dispense() {  
    if (state == SOLD) {  
        System.out.println("A gumball comes rolling out the slot");  
        count = count - 1;  
        if (count == 0) {  
            System.out.println("Oops, out of gumballs!");  
            state = SOLD_OUT;  
        } else {  
            state = NO_QUARTER;  
        }  
    } else if (state == NO_QUARTER) {  
        System.out.println("You need to pay first");  
    } else if (state == SOLD_OUT) {  
        System.out.println("No gumball dispensed");  
    } else if (state == HAS_QUARTER) {  
        System.out.println("No gumball dispensed");  
    }  
} ↗ We're in the SOLD state; give 'em a gumball!  
} ↗ Here's where we handle the "out of gumballs" condition: If this was the last one, we set the machine's state to SOLD_OUT; otherwise, we're back to not having a quarter.  
} ↗ None of these should ever happen, but if they do, we give 'em an error, not a gumball.  
} ↗ // other methods here like toString() and refill()  
}
```



State Pattern Motivation: Dealing with Change

- If we want to change the structure of the state machine...
 - e.g. to add a state
 - like sometimes letting people win extra gumballs
 - e.g. to add a transition
 - e.g. to refill the machine when empty
- ...we need to change the implementation of each method
 - tedious and error-prone due to all the conditionals

```
final static int SOLD_OUT = 0;
final static int NO_QUARTER = 1;
final static int HAS_QUARTER = 2;
final static int SOLD = 3;

public void insertQuarter() {
    // insert quarter code here
}

public void ejectQuarter() {
    // eject quarter code here
}

public void turnCrank() {
    // turn crank code here
}

public void dispense() {
    // dispense code here
}
```

First, you'd have to add a new WINNER state here. That isn't too bad...

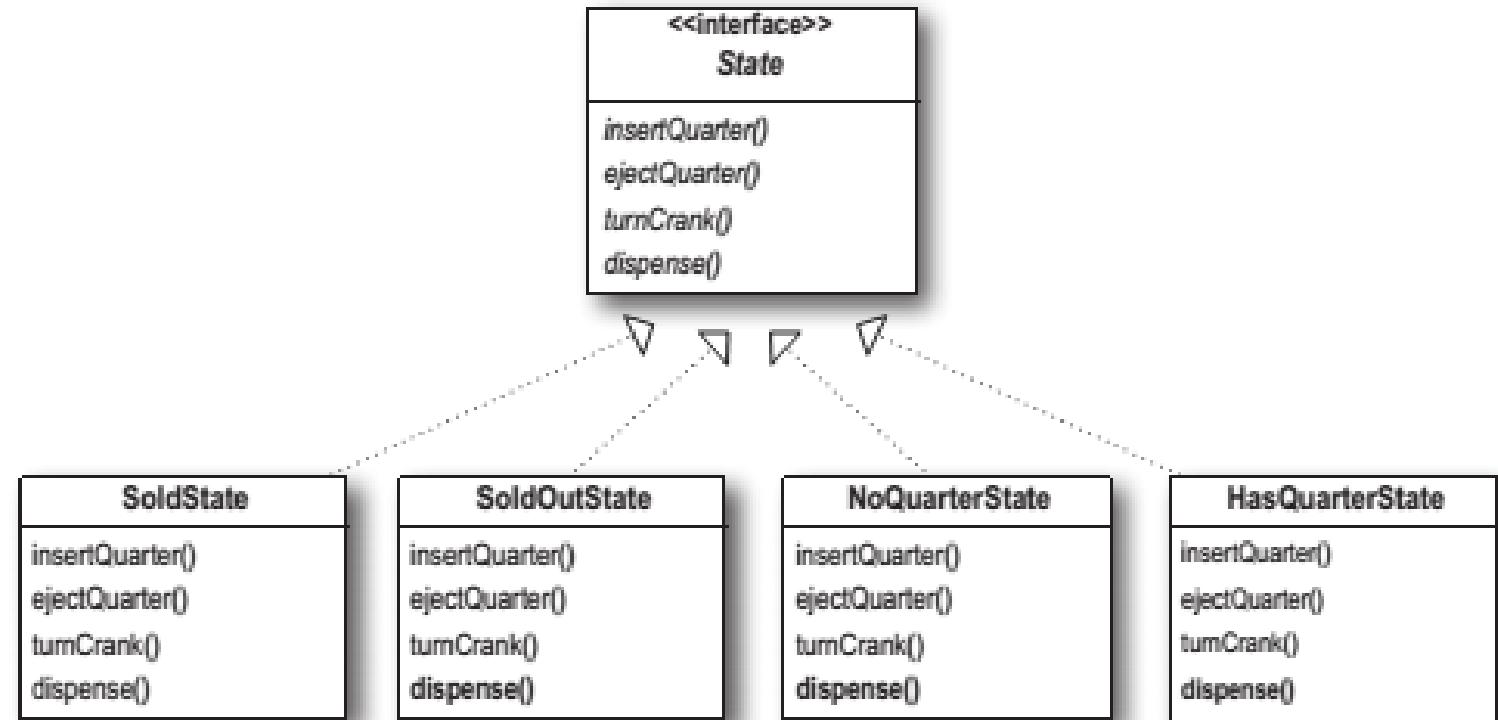
... but then, you'd have to add a new conditional in every single method to handle the WINNER state; that's a lot of code to modify.

turnCrank() will get especially messy, because you'd have to add code to check to see whether you've got a WINNER and then switch to either the WINNER state or the SOLD state.



State Pattern Solution: Modeling States as Classes

- Represent each state as a class (implementing a common State interface)
 - with methods representing the potential events
 - and method implementations determining what should happen upon a particular event in a particular state



State Pattern Example Implementation

- Integer state representations are replaced by State objects.
- State objects are created and initial state is determined.

```
public class GumballMachine {  
  
    State soldOutState;  
    State noQuarterState;  
    State hasQuarterState;  
    State soldState;  
  
    State state;  
    int count = 0;  
  
    public GumballMachine(int numberGumballs) {  
        soldOutState = new SoldOutState(this);  
        noQuarterState = new NoQuarterState(this);  
        hasQuarterState = new HasQuarterState(this);  
        soldState = new SoldState(this);  
  
        this.count = numberGumballs;  
        if (numberGumballs > 0) {  
            state = noQuarterState;  
        } else {  
            state = soldOutState;  
        }  
    }  
}
```

Here are all the States again...

...and the State instance variable.

The count instance variable holds the count of gumballs – initially the machine is empty.

Our constructor takes the initial number of gumballs and stores it in an instance variable.

It also creates the State instances, one of each.

If there are more than 0 gumballs we set the state to the NoQuarterState; otherwise, we start in the SoldOutState.

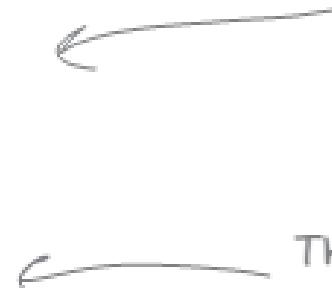


State Pattern Example Implementation

- The event methods of the Gumball-Machine don't do anything on their own, but delegate the decision on what to do upon an event to the current state.

```
public void insertQuarter() {  
    state.insertQuarter();  
}  
public void ejectQuarter() {  
    state.ejectQuarter();  
}  
public void turnCrank() {  
    state.turnCrank();  
    state.dispense();  
}  
  
void setState(State state) {  
    this.state = state;  
}  
  
void releaseBall() {  
    System.out.println("A gumball comes rolling out the slot...");  
    if (count != 0) {  
        count = count - 1;  
    }  
}  
// More methods here including getters for each State...  
}
```

Note that we don't need an action method for dispense() in GumballMachine because it's just an internal action; a user can't ask the machine to dispense directly. But we do call dispense() on the State object from the turnCrank() method.



This method allows other objects (like our State objects) to transition the machine to a different state.



The machine supports a releaseBall() helper method that releases the ball and decrements the count instance variable.



State Pattern Example Impl.

- Each State stores a reference to the GumballMachine that it describes.
- Each method contains logic reacting to the respective event and possibly changing its gumballMachine's state.
- (Other states are implemented analogously.)

```
public class SoldState implements State {  
    GumballMachine gumballMachine;  
    public SoldState(GumballMachine gm) { gumballMachine = gm; }  
  
    public void insertQuarter() {  
        System.out.println("Please wait, we're already giving you a gumball");  
    }  
  
    public void ejectQuarter() {  
        System.out.println("Sorry, you already turned the crank");  
    }  
  
    public void turnCrank() {  
        System.out.println("Turning twice doesn't get you another gumball!");  
    }  
  
    public void dispense() {  
        gumballMachine.releaseBall();  
        if (gumballMachine.getCount() > 0) {  
            gumballMachine.setState(gumballMachine.getNoQuarterState());  
        } else {  
            System.out.println("Oops, out of gumballs!");  
            gumballMachine.setState(gumballMachine.getSoldOutState());  
        }  
    }  
}
```

We're in the SoldState, which means the customer paid. So, we first need to ask the machine to release a gumball.

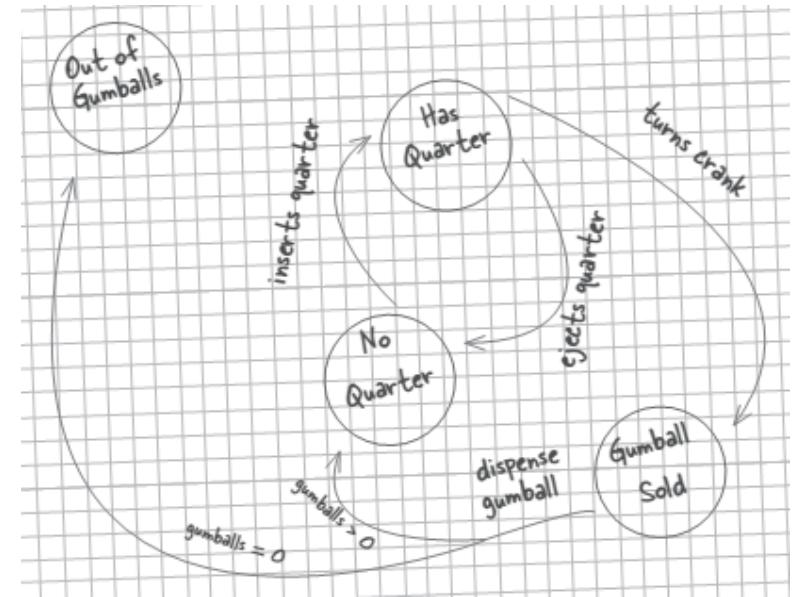
Then we ask the machine what the gumball count is, and either transition to the NoQuarterState or the SoldOutState.



State Pattern

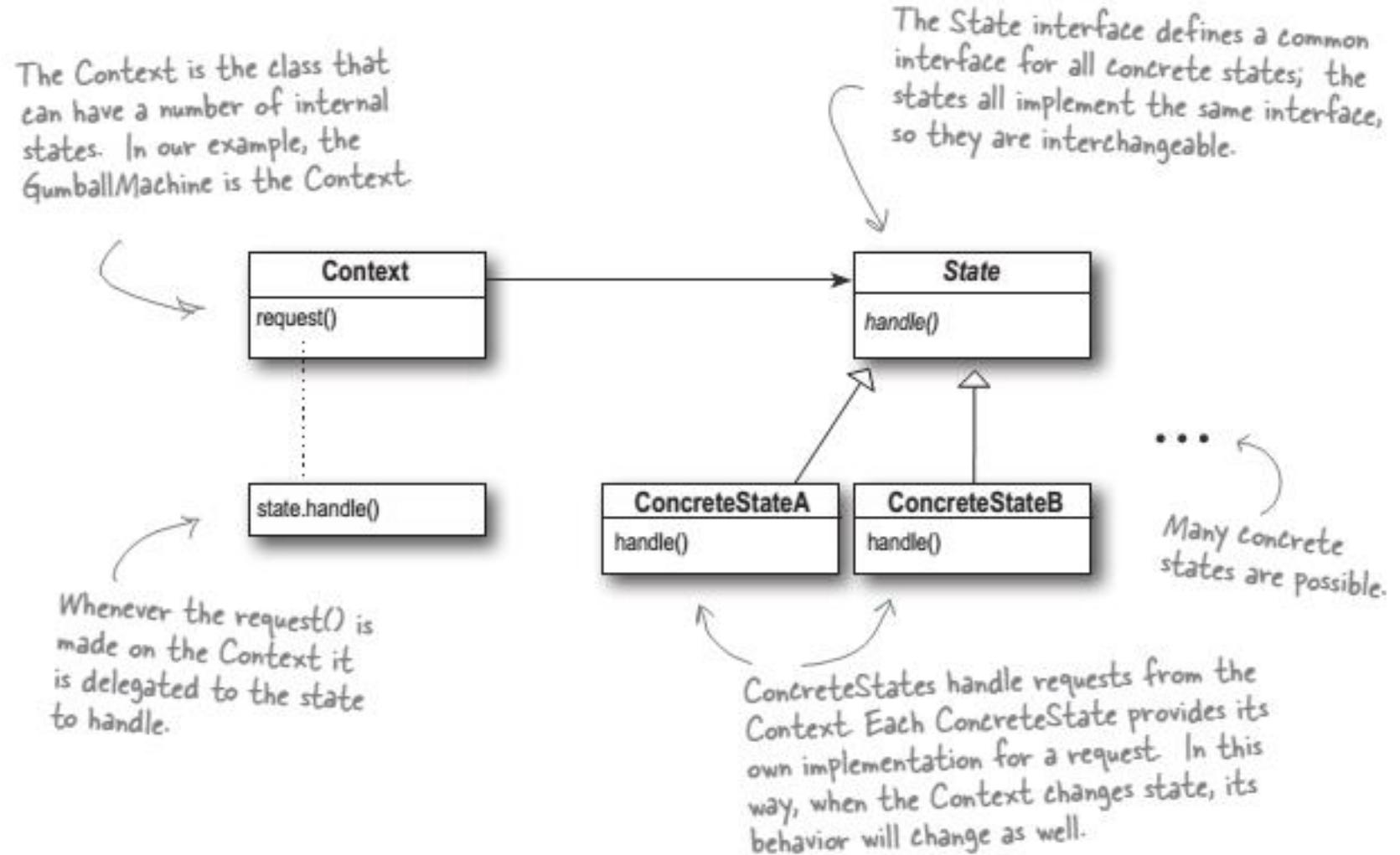
Protip: Dealing with Impossible Transitions

- A lot of the previous code was concerned with responding to events that
 - are not valid transitions in the business domain
 - but could be triggered by calls to the respective methods.
- In practice, you wouldn't write comments to stdout, but throw UnsupportedOperationExceptions when encountering events not valid in current state.
- An efficient way to implement this:
 - Instead of an `interface State`, use an `abstract class State` where all that each method does is `throw new UnsupportedOperationException()`.
 - In subclasses that `extend State`, you then need to override only methods for transitions that are actually allowed in the business domain, making the concrete state classes much smaller and more readable.



State Pattern Summary

- The **State pattern** allows an object to alter its behavior when its internal state changes.
- Notice: Structurally identical to the Strategy pattern!
 - with State as interchangeable algorithm
 - Key difference: State changes itself instead of being determined from outside



State Pattern

Discussion: State vs. Usability

- In many software systems, it is discouraged to have different states (“modes”) in which the application behaves differently, especially if it is not readily distinguishable which state the application is currently in.
 - e.g. having to switch an editor from “view mode” to “edit mode” in order to change a text
- Instead, users should usually be enabled to do whatever they like at any time, without the application getting in the way
 - e.g. in a modern editor, you can view and edit a document without having to switch modes
- Sometimes, working with states/modes can be helpful if they are intuitive though
 - e.g. using different “drawing tools” in a photo editing software
 - e.g. browsing a wiki page in “viewer” or “editor” mode, depending on login state and role
 - In games, certain player or environment states can be key features of gameplay
 - e.g. temporary superpowers etc.



Summary: Selected Design Patterns

Introduced in HBV401G

- Singleton
- Factory
- Observer
- Façade
- Adapter
- Proxy
- MVC

Introduced in this course

- Decorator (→ FR3)
 - Command (→ FR6)
 - Strategy (→ FR1)
 - State (→ FR10)
-
- Many more – see e.g.
 - Freeman, Robson: Head First Design Patterns, O'Reilly 2004
 - Gamma, Helm, Johnson, Vlissides: Design Patterns, Addison-Wesley 1994



Team Assignment 4

- In the Unified Process' Construction phase, implementation of the system proceeds incrementally.
- In each iteration,
 - A number of use cases are selected for implementation.
 - The technical design of the use cases is detailed.
 - The chosen design is implemented and tested.
- Team Assignment 4 focuses on
 - implementing the **final product**, and
 - explaining the **process** and
 - the **architecture and design decisions** that went into it.



Team Assignment 4: Content

- On **Thu 26 Nov**, demonstrate and explain your product in class:
 1. **Product:** What does your system do? Demonstrate the key use cases of your product.
 2. **Architecture:** How does your product work? Explain architecture & key design decisions.
 3. **Process:** How did you build the system? Relate and interpret challenges you faced.
 - Demonstrate #1 live; prepare a few slides for #2 and #3 (the order of #1–#3 is up to you).
- By **Sun 29 Nov**, submit in Uglá:
 - The **source code** of your final product, including everything required to build and run it, i.e.:
 - All server- and client-side code
 - SQL statements to create the required database schema and test data
 - unless auto-generated by the JPA
 - Any necessary instructions for building and running the project
 - e.g. Maven/Gradle scripts or manual instructions for obtaining 3rd party components (e.g. the database)
 - The **slides** of the presentation you gave in class



Team Assignment 4: Presentation Format

- The presentation on **Thu 26 Nov** should be given
 - in 2-person teams: by the member who has presented only one assignment yet
 - in 3-person teams: by all members together, with each member presenting one part
 - in 4-person teams: by the member who has not presented any assignment yet
- The presentation may take **max. 12 minutes** (plus 3 minutes for questions).
- All team members receive the same grade.
 - In 2- and 4-person teams: The presenter will receive a bonus/malus for parts #2 and #3.
- Presentations will be given in three rooms in parallel (9-10 teams per room).
 - Sign-up for presentation slots will open on Doodle next week.
- Please strive to be present for the whole time on Thursday (08:30-11:30)
 - so you can see other teams' work and learn from their experiences
 - so your classmates have an audience as well



Team Assignment 4: Artifact Format

- All artifacts must be produced by all team members together.
- Please submit in **by Sun 29 Nov** in Uglar
 - a ZIP file containing the **source code** and accompanying artifacts (see slide 34)
 - a PDF document with your **presentation slides**,
containing your team number, the names and kennitölur of all team members on title slide
- Only the presenting team member should submit the files.
 - If all members are presenting together, only one team representative should upload files.
 - Don't submit multiple versions – we'll just grade the first one we encounter!



Team Assignment 4: Grading Criteria

- **Overall**

- ✓ Presenter shows initiative, explains things clearly, shows understanding of the approach

1. Product Demonstration

- ✓ Software works stably
- ✓ Key use cases are working

- **Submitted code**

- Not graded explicitly, but checked for conformity with presented prototype
 - Exception: Complete mess may lead to malus points

2. Software Architecture and Design

- ✓ Architecture is described and illustrated clearly
- ✓ Key design decisions are described and illustrated clearly
- ✓ Room for improvement is discussed critically

3. Software Process

- ✓ Design and development process over the course of the semester is described clearly, in relation to RUP phases
- ✓ Handling of technical / methodical / collaboration challenges is discussed critically



Gangi þér vel!

book@hi.is



HÁSKÓLI ÍSLANDS
VERKFRÆÐI- OG NÁTTÚRVÍSINDASVIÐ

IÐNAÐARVERKFRÆÐI-, VÉLAVERKFRÆÐI-
OG TÖLVUNARFRÆÐIDEILD





Hugbúnaðarverkefni 1 / Software Project 1

12. Presentation Layer

HBV501G – Fall 2015

Matthias Book



HÁSKÓLI ÍSLANDS
VERKFRÆÐI- OG NÁTTÚRVÍSINDASVIÐ
ÍÐNAÐARVERKFRÆÐI-, VÉLAVERKFRÆÐI-
OG TÖLVUNARFRÆÐIDEILD

Kennslukönnun

Evaluate this course on Uglá!
(until 1 Dec)



Recap: Team Assignment 4: Content

- On **Thu 26 Nov**, demonstrate and explain your product for your classmates:
 1. **Product:** What does your system do? Demonstrate the key use cases of your product.
 2. **Architecture:** How does your product work? Explain architecture & key design decisions.
 3. **Process:** How did you build the system? Relate and interpret challenges you faced.
 - Demonstrate #1 live; prepare a few slides for #2 and #3 (the order of #1–#3 is up to you).
- By **Sun 29 Nov**, submit in Uglá:
 - The **source code** of your final product, including everything required to build and run it, i.e.:
 - All server- and client-side code
 - SQL statements to create the required database schema and test data
 - unless auto-generated by the JPA
 - Any necessary instructions for building and running the project
 - e.g. Maven/Gradle scripts or manual instructions for obtaining 3rd party components (e.g. the database)
 - The **slides** of the presentation you gave in class

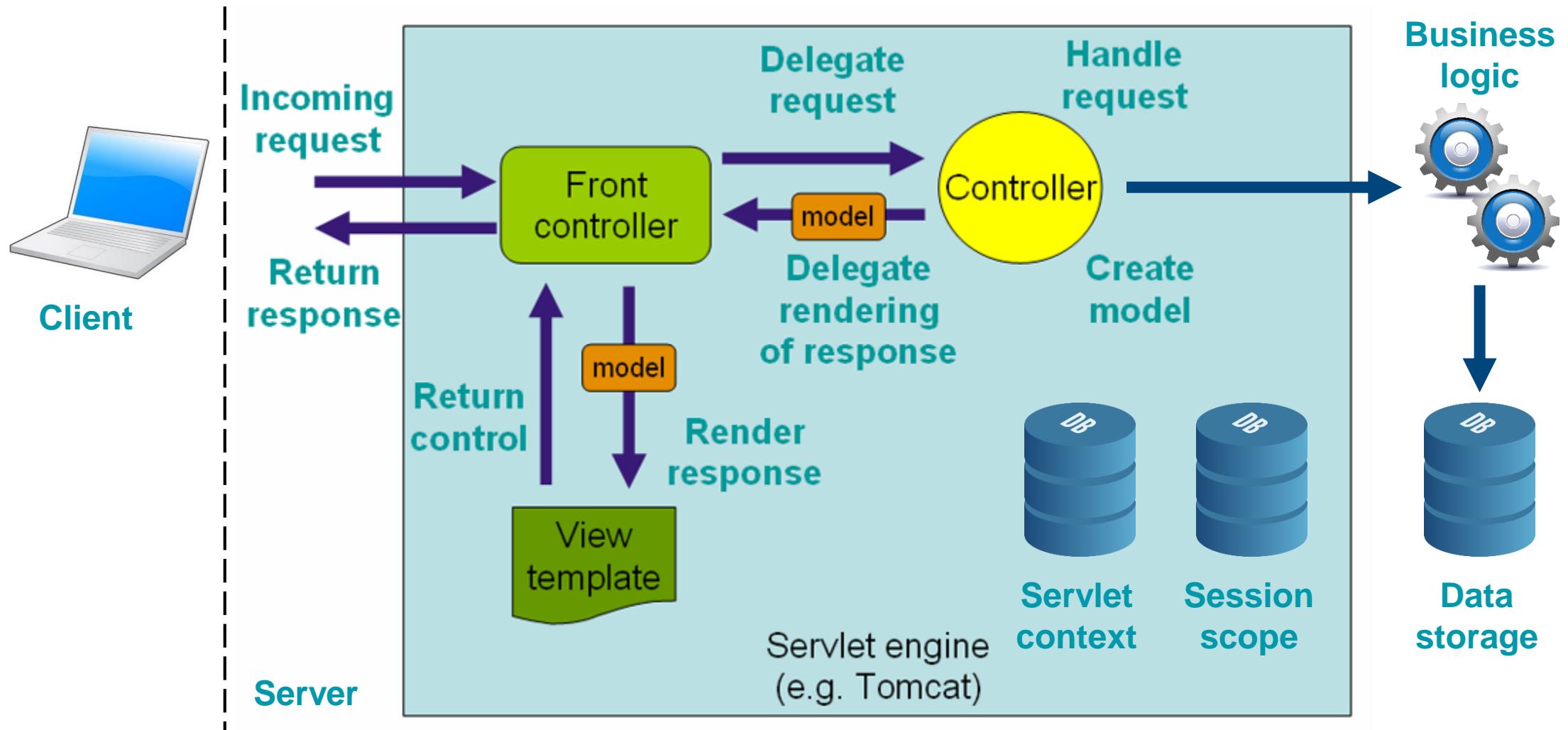


Team Assignment 4: Presentations

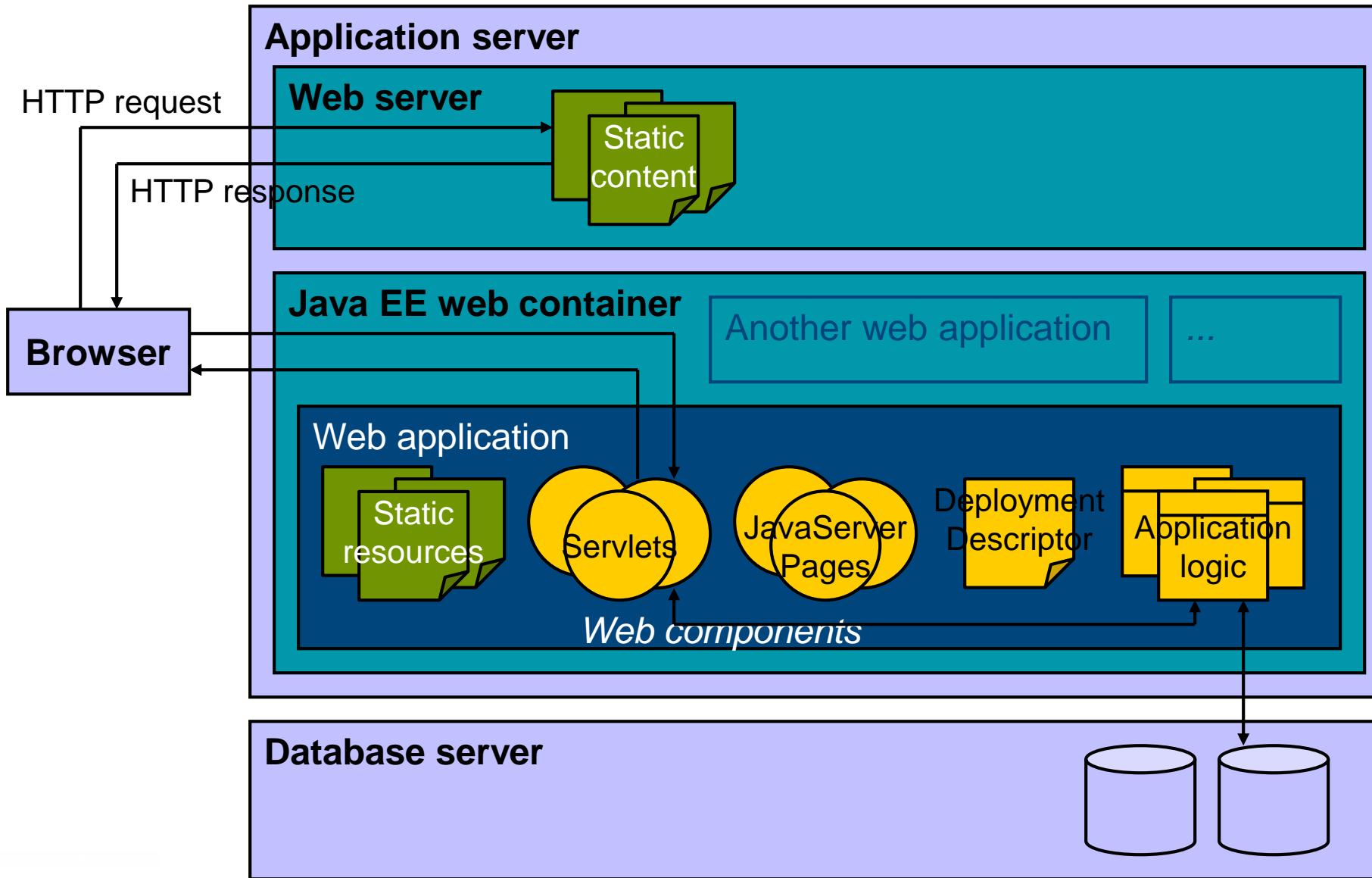
- The presentation on **Thu 26 Nov** should be given
 - in 2-person teams: by the member who has presented only one assignment yet
 - in 3-person teams: by all members together, with each member presenting one part
 - in 4-person teams: by the member who has not presented any assignment yet
- The presentation may take **max. 12 minutes** (plus 3 minutes for questions).
- All team members receive the same grade.
 - In 2- and 4-person teams: The presenter will receive a bonus/malus for parts #2 and #3.
- **Presentations will be given in three rooms in VR-II in parallel:**
 - **Elsa's teams: V02-138 / Daniel's teams: V02-147 / Matthias' teams: V02-152**
- **Your presentation will be in your team's usual consultation timeslot.**
- Please strive to be present for the whole time on Thursday (08:30-11:30)
 - so you can see other teams' work and learn from their experiences
 - so your classmates have an audience as well



Recap: Spring Web MVC Framework



Server-Side Web Applications: Under the Hood



Java Servlets

see also:

- Williams: Professional Java for Web Applications, Ch. 3



Example: Time-of-Day Servlet

```
import javax.servlet.*; import javax.servlet.http.*; import java.util.*; import java.io.*;  
  
public class TimeOfDayServlet extends HttpServlet {  
    protected void doGet(HttpServletRequest request, HttpServletResponse response)  
        throws ServletException, IOException {  
        response.setContentType("text/html");  
        PrintWriter out = response.getWriter();  
        out.println("<html>");  
        out.println("<body>");  
        out.println("<p>Time: " + new Date() + "</p>");  
        out.println("</body>");  
        out.println("</html>");  
    }  
}
```

Inherit methods for initialization etc.

protected void doGet(HttpServletRequest request, HttpServletResponse response)

Process GET requests

throws ServletException, IOException {

Set content type of ensuing output

response.setContentType("text/html");

Write HTML code to output stream



Deployment Descriptor (web.xml)

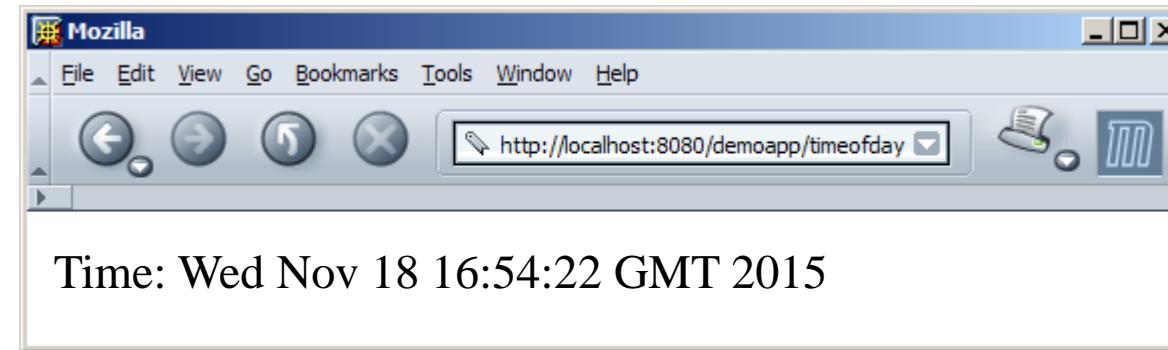
```
<?xml version="1.0" encoding="UTF-8"?>  
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"  
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
         xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee  
                           http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd" version="3.1">  
  
    <display-name>Demo Application</display-name>           → For display in management tools  
    <servlet>  
        <servlet-name>TimeOfDayServlet</servlet-name>          → For references in the deployment descriptor  
        <servlet-class>TimeOfDayServlet</servlet-class>  
    </servlet>  
    <servlet-mapping>  
        <servlet-name>TimeOfDayServlet</servlet-name>  
        <url-pattern>/timeofday</url-pattern>                  → Component alias (path to servlet in URL)  
    </servlet-mapping>  
</web-app>
```



Invoking the Servlet

- `http://localhost:8080/demoapp/timeofday`

Context root Component alias



Servlet Lifecycle

- Within the servlet's lifecycle, the web container calls the following methods of the servlet that should be overridden as needed:

- Servlet initialization:

```
public void init() throws ServletException
```

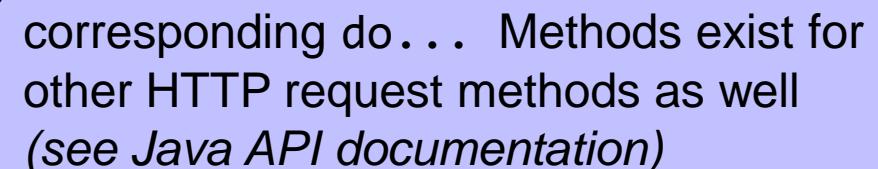
- Request handling (multi-threaded):

```
protected void doGet(  
    HttpServletRequest req, HttpServletResponse resp)  
    throws ServletException, java.io.IOException
```

```
protected void doPost(  
    HttpServletRequest req, HttpServletResponse resp)  
    throws ServletException, java.io.IOException
```

- Servlet destruction:

```
public void destroy()
```



corresponding do... Methods exist for other HTTP request methods as well (see Java API documentation)

HTTP Requests

- Clients (e.g. web browsers) can sent HTTP requests using a number of methods
 - most commonly: GET and POST (RESTful applications often also use PUT, DELETE etc.)
 - In either case, URL contains the address of the web component that shall receive the request
- **GET method**
 - Request parameters are included visibly in the URL
 - Advantages:
 - Parameters are included when using browser's "Back" button
 - Parameters are included in bookmarks
 - Disadvantages:
 - Sensitive data may be included in bookmarks or server logfiles
 - Length of URL is limited, encoding of non-ASCII characters is required
 - suitable for moderate amounts of non-sensitive text parameters
- **POST method**
 - Request parameters are included "invisibly" in HTTP request body
 - Advantages: more secure, can transfer binary data of any size
 - Disadvantages: Browser's "Back" button, bookmarks etc. may not work



Example: HTML Login Form

...

```
<form action="/demoapp/login" method="get">  
    Name: <input type=text name="username"><br/>  
    Password: <input type=password name="passwd"><br/>  
    <input type=submit name="login" value="Enter">  
</form>
```

...

- URL created upon form submission:

`http://localhost/demoapp/login?username=mbook&passwd=mySecretPassword
&login=Enter`

- Server-side web container will forward this request to the servlet with the alias login in the web application demoapp



Handling a Request

- doGet/doPost receives an HttpServletRequest object with data contained in request from client
- Accessing request parameters in HttpServletRequest:

```
public String getParameter(String name)
```

- returns the value of the first parameter with the given name
- or `null` if the parameter is not present in the request

```
public String[] getParameterValues(String name)
```

- returns the values of all parameters with the given name
- or `null` if the parameter is not present in the request
 - typically used for check boxes, multiple selection fields

```
public Enumeration getParameterNames()
```

- Returns a list of names of all parameters in the request
- or `null` if no parameters are present in the request

- Binary data can be read from the request body using streams (see Java API)



Producing a Response

- doGet/doPost receives an HttpServletResponse object with data on response to be returned to client
- Outputting text to the HttpServletResponse:

```
public void setContentType(String type)
```

- announces content type of the following output (usually text/html)
- Encoding can be added optionally, e.g. ; charset=UTF-8
- Needs to be called before any invocations of getWriter or getOutputStream!

```
public java.io.PrintWriter getWriter()
```

```
public ServletOutputStream getOutputStream()
```

- returns a Writer or OutputStream that can be used to send text or binary data to client
- Only one of the methods may be called, and only once per request!
- HTML code is sent to client using print methods of the Writer

- Further methods to return binary data to client exist (see Java API)



Example: Request Handling

```
import javax.servlet.*; import javax.servlet.http.*;
import java.util.*; import java.io.*;

public class LoginServlet extends HttpServlet {
    protected void doGet(
        HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        String username = request.getParameter("username");
        String passwd = request.getParameter("passwd");
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><body>");
        out.print("<p>Hi " + username + ", your password is ");
        out.print(passwd.equals("mySecretPassword") ? "correct" : "incorrect");
        out.println("</p>");
        out.println("</body></html>");
    }
}
```



Session Management

see also:

- Williams: Professional Java for Web Applications, Ch. 5



Motivation: Sessions and State

- HTTP is a stateless protocol: The server usually has no way of telling whether any two requests were sent by the same client.
- Problem: Often, the application needs to know which client sent which request (**session**) and with which data structures this user is currently working (**state**).
- Solution: The client must identify itself in each request.
 - It must be ensured that a unique client ID is sent along with each request.
 - Manual implementation undesirable: high technical effort and complexity → error-prone

Session Management: Solutions in Java

- The web container automatically takes care of
 - assigning session IDs to clients
 - transmitting them with each request
 - making the respective client-specific state available on the server
- **Default:** Transmitting the session ID in an HTTP cookie
 - Cookie: small data entity provided by the server that is stored on the client and transmitted to the server with each request
 - Advantage: Transmission of session ID without any developer involvement (automatic mechanism between web container and web browser)
 - Disadvantage: Not all clients can handle cookies; some might have them disabled
- **Fallback:** Appending the session ID to each request URL
 - `http://localhost/webapp/servlet;jsessionid=XYZ?param=val`
 - Advantage: Independent of client capabilities
 - Disadvantage: Developer must ensure that the web container can perform URL rewriting (automatic inclusion of session ID into any links provided on the application's web pages)



URL Rewriting in Servlets

- Problem when performing URL rewriting:
 - Session IDs are assigned at runtime by the web container
 - URLs are implemented in servlets and JavaServer Pages at design time by the developer
 - How to get session IDs into URLs?
- Solution in servlets: Use encodeURL method of HttpServletResponse
 - public String encodeURL(String url)
 - Inserts jsessionid into URL if necessary
 - Should be used for any URLs in HTML code generated by servlets
 - Example:

```
out.print("<a href=' " + response.encodeURL("/app/servlet?param=val") + "'>");
```



URL Rewriting in JSPs

- Problem when performing URL rewriting:
 - Session IDs are assigned at runtime by the web container
 - URLs are implemented in servlets and JavaServer Pages at design time by the developer
 - How to get session IDs into URLs?
- Solution in JSPs: Use url tag of Java Standard Tag Library
 - Can be used to construct URLs anywhere in HTML code
 - Inserts jsessionid into URL if necessary
 - Example:

```
<a href="  
    <c:url value="/shop">  
        <c:param name="action" value="addToCart" />  
    </c:url>  
>Add to cart</a>
```



Session Management in Servlets

- The current client's session can be created or retrieved using methods of `HttpServletRequest`:

```
public HttpSession getSession()
```

```
public HttpSession getSession(boolean create)
```

- assigns a new session to the client from which the request came that we are currently handling, if it does not yet have one or if `create==true`
- returns the `HttpSession` object in which all data associated with this client is stored
- All servlets and JavaServerPages have access to the session object and can exchange client-specific data through it.
- The session object exists until
 - it is explicitly destroyed using its `invalidate` method, or
 - no more requests come in from the client for a particular amount of time ("session timeout")



Accessing Data in Sessions

- Methods for accessing data in the session (which is essentially a Map):

```
public void setAttribute(String name, Object attrb)
```

- stores the object attrb under the key name in the session

```
public Object getAttribute(String name)
```

- returns the object stored under the key name in the session

```
public void removeAttribute(String name)
```

- Removes the object stored under the key name from the session

- Methods for controlling the session's lifecycle:

```
public void setMaxInactiveInterval(int interval)
```

- determines the inactivity interval (in seconds) after which the session is invalidated

```
public void invalidate()
```

- marks the session ID as invalid, removes object references from the session

- More methods for accessing session meta data → see Java API



Orientation

- If you are using the Spring Web MVC framework and Spring Boot, you don't need to implement any servlets yourself, but should use the much more convenient techniques introduced in Lecture 6 instead:
 - @Controllers instead of servlets
 - @RequestMappings etc. instead of deployment descriptors
 - @RequestParameters instead of HttpServletRequest objects
 - etc.
- The preceding slides are intended to explain the mechanisms that are going on under the hood (and may be more visible when you use other frameworks).
- In the following slides, we will focus on parts that the Spring framework does not take off your shoulders: Creating the user interface on the server side with JavaServer Pages.



JavaServer Pages

see also:

- Williams: Professional Java for Web Applications, Ch. 4 & 6



Motivation: JavaServer Pages

- Servlets are simple Java classes. Implications:
 - Accessing other classes, implementing logic is easy
 - Construction and output of HTML code is tedious
 - In larger projects, we need:
 - better separation of application and presentation layer
 - more convenient hypertext construction
 - Solution: **JavaServer Pages (JSPs)**
 - HTML pages with integrated dynamic elements
 - Easy implementation of HTML code
 - Accessing classes, implementing logic is more tedious
- Recommendation:
- Servlets for invoking application logic (usually encapsulated by a framework like Spring)
 - JSPs for presenting user interface



Example: Time-of-Day JSP (showtime.jsp)

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ page import="java.util.Date" %>

<html>
  <body>
    <p>Time: <%= new Date() %></p>
  </body>
</html>
```

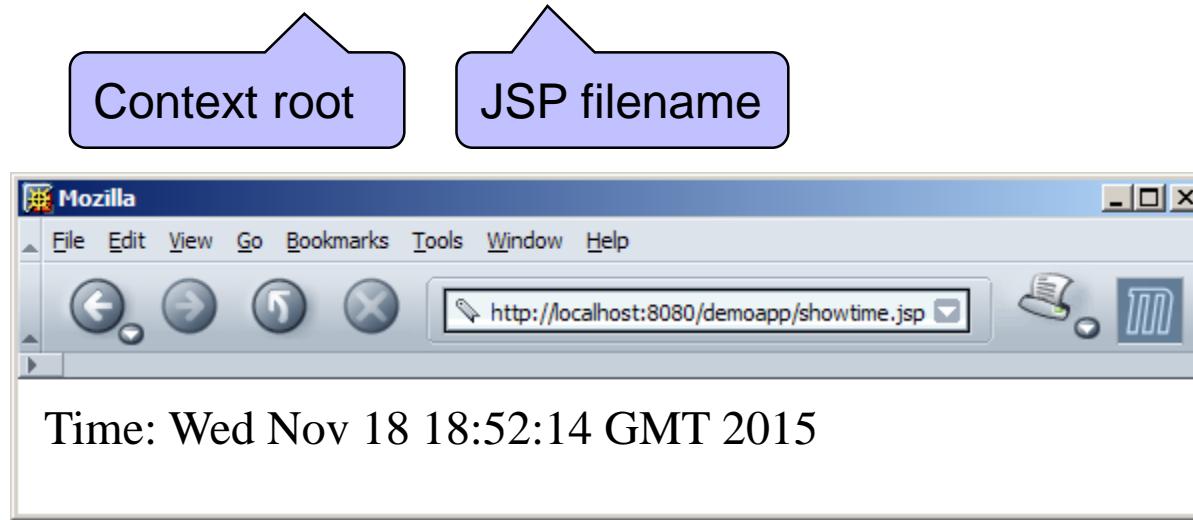
Static text: included directly
in HTML page

Script element: Content is
evaluated at runtime and
integrated into HTML page



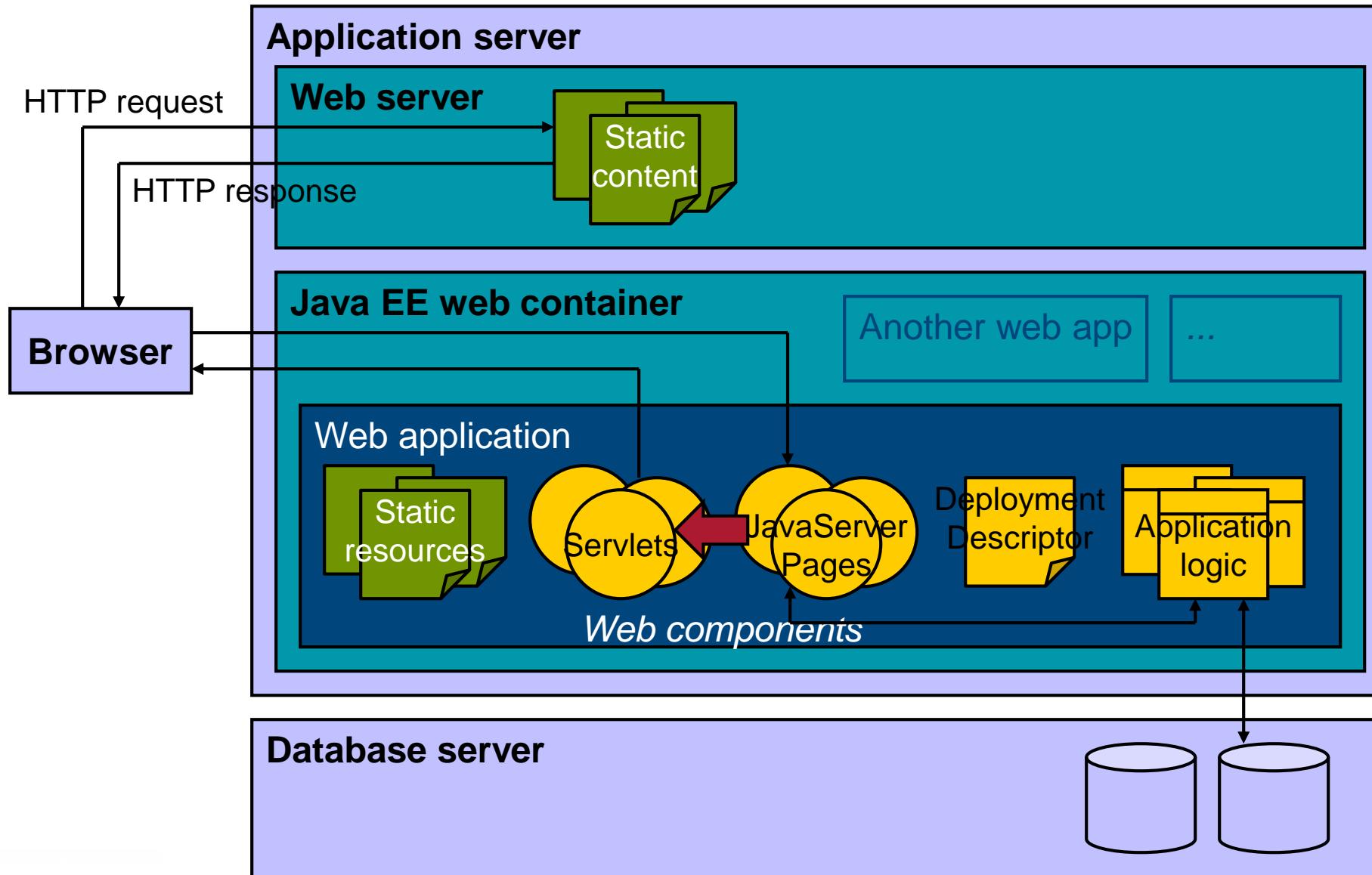
Invoking the JSP

`http://localhost:8080/demoapp/showtime.jsp`



- Upon first invocation, a servlet is generated from the JSP
 - may take slightly longer
- Subsequent invocations are sent directly to servlet
 - transparent for developer and user

Dynamic Servlet Generation from JSPs



Integration of Java Statements into JSPs (*discouraged*)

- Importing classes with the page directive:

```
<%@page import="package1.class, package2.*, ..." %>
```

- makes given classes from given packages available in JSP

- Integrating Java statements with Scriptlets:

```
<% Java statements %>
```

- Executes Java code in brackets at runtime

- Java control structures can be applied to static HTML code:

```
<% if (request.getParameter("passwd").equals("mySecretPassword")) { %>
    <p>Password correct!</p>
<% } else { %>
    <p>Password incorrect!</p>
<% } %>
```

- Integrating Java expressions into JSPs: `<%= Java expression %>`

- Evaluates expression in brackets
- Converts the result to a string
- Includes the string into the HTML code



Implicit Objects in Java Fragments (*discouraged*)

- The following objects are available in Java fragments (statements or expressions) in any JSP, without having to be explicitly imported:
 - `HttpServletRequest request`: current request
 - `HttpServletResponse response`: current response
 - `PageContext pageContext`: page-specific data store
 - `HttpSession session`: current user's session (data store)
 - `ServletContext application`: application-wide data store
 - `JspWriter out`: text output stream to client
 - `ServletConfig config`: web application configuration (data store)
- Problem with Java code built into JSP:
No separation of application logic and presentation
 - **Avoid – use Expression Language and Custom Tags instead!**

Excursion: JavaBeans

- Accessing data model is easiest when entities follow **JavaBeans** convention:
- JavaBeans are regular classes adhering to this structure:
 - Have a constructor without parameters:
`public BeanName()`
 - Have a get method for each readable property:
`public PropertyType getPropertyname()`
 - Have a set method for each writable property:
`public void setPropertyName(PropertyType property)`
- Rules for properties:
 - usually attributes of the JavaBean class, but
 - can also be calculated in get methods
 - can also be combined with other values in set methods
 - *.PropertyType* can be a primitive type, reference type or array



Expression Language (EL)

- Usage scenarios
 - in static HTML code (will be evaluated and incorporated as string into HTML code)
 - in attributes of JSP elements (such as actions and custom tags)
- Syntax: $\${expression}$
 - Operators: common operators for comparison, arithmetics etc.
 - Operands: Literals, JavaBeans, implicit objects, methods
 - Expression will be evaluated to a typed value (primitive or reference type)
 - When used in static text, automatic conversion to String
- Accessing structured objects:

| ■ EL syntax | Type of <i>a</i> | Semantics of <i>b</i> | equiv. Java syntax |
|---------------------------|-------------------------|------------------------------|-----------------------------|
| ■ $a.b$ | Array | Index | $a[b]$ or $a["b"]$ |
| ■ $a[b]$ | List | Index | $a.get(b)$ |
| ■ $a[b]$ | Map | Key | $a.get(b)$ |
| ■ $a[\textcolor{red}{b}]$ | JavaBean | Property name | $a.get\textcolor{red}{B}()$ |



Accessing JavaBean Properties

`beanInstance.propertyName`

- looks for the JavaBean *beanInstance* in the page, request, session and application scope (in this order) and returns the value of that bean's property called *propertyName*

`pageScope.beanInstance.propertyName`

`requestScope.beanInstance.propertyName`

`sessionScope.beanInstance.propertyName`

`applicationScope.beanInstance.propertyName`

- retrieves the JavaBean *beanInstance* from the given scope and return the value of that bean's property called *propertyName*



Implicit Objects in EL

- The following objects are available in EL expressions in any JSP, without having to be imported explicitly:

`pageScope.obj, requestScope.obj, sessionScope.obj, applicationScope.obj`

- delivers the object *obj* in the page, request, session or application scope

`initParam.parameterName`

- delivers the value of the initialization parameter called *parameterName*

`param.parameterName`

- delivers the value of the first request parameter called *parameterName*

`paramValues.parameterName`

- delivers an array with the values of all request parameters called *parameterName*

`cookie.cookieName`

- delivers the value stored in the cookie *cookieName*



Example: LoginServlet Writes Data

```
public class LoginServlet extends HttpServlet {  
    public void doGet(HttpServletRequest request, HttpServletResponse response)  
        throws ServletException, IOException {  
        HttpSession session = request.getSession();  
        String username = request.getParameter("username");  
        String passwd = request.getParameter("passwd");  
        User u = new User(username, passwd);  
        session.setAttribute("user", u);  
        PrintWriter out = response.getWriter();  
        out.println("<html><body>");  
        out.println("<p>User " + username + " with password " + passwd + " logged in.</p>");  
        out.print("<p><a href='" + response.encodeURL("menu.jsp?lang=is") + "'>íslenska</a> ");  
        out.println("<a href='" + response.encodeURL("menu.jsp?lang=en") + "'>english</a></p>");  
        out.println("</body></html>");  
        out.close();  
    }  
}
```

Place a new user
bean in session scope



Example: menu.jsp Reads Data

```
<html>
  <body>
    <p>Welcome, ${sessionScope.user.username}!</p>
    <p>Your password is ${user.passwd}.</p>
    <p>Chosen language: ${param.lang} </p>
  </body>
</html>
```

Get username
property of user
bean from session

Access passwd
property without
specifying scope of
user bean

Read request
parameter lang



Tag Libraries

see also:

- Williams: Professional Java for Web Applications, Ch. 7



JSP Standard Tag Library and Custom Tags

- Tag libraries define additional tags that can be incorporated into the HTML code to generate markup or implement control structures
 - Syntax: `<prefix:tag ...>...</prefix:tag>`
 - Developers can build „taglibs“ with individual custom tags and use them in their JSPs
 - Many template engines are based on custom tags
- The **JSP Standard Tag Library (JSTL)** contains a variety of tags for control flow, XML processing, internationalization, SQL queries, string manipulation etc.
 - in the libraries core, xml, fmt, sql, functions
 - Imported into JSP with the taglib directive:
`<%@taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>`
 - Imports the JSTL Core taglib
 - Tags of this library have the prefix c

Conditional Evaluation: if Tag

- Syntax:

```
<c:if test="${expression}">  
    Element content  
</c:if>
```

- If *expression* is evaluated to `true`, the *Element content* is evaluated and incorporated into the output.

- Example:

```
<c:if test="${user.passwd == 'mySecretPassword'}">  
    Password correct!  
</c:if>
```

- Will integrate the string `Password correct!` into the output if the property `passwd` of the `user` object has the value `mySecretPassword`; otherwise, no output is created.



Conditional Evaluation: choose Tag

```
<c:choose>
  <c:when test="${expr1}>
    Element Content1
  </c:when>
  <c:when test="${expr2}>
    Element Content2
  </c:when>
  <c:otherwise>
    Element Content3
  </c:otherwise>
</c:choose>
```

- the *Element Content* of the first when block whose *expr* can be evaluated to **true** will be evaluated and incorporated into the output
- If no *expr* can be evaluated to **true**, the *Element Content* of the otherwise block will be evaluated and incorporated into the output



Iteration: forEach Tag

- Syntax:

```
<c:forEach var="element" items="${collection}">  
    Content using ${element}  
</c:forEach>
```

- Evaluates the *Content* for each element of the *collection*
- The current element of each iteration is available in the variable *element*

- Treatment of various types of collections:

- Array of primitive or reference types
 - primitive types are packed in instances of wrapper classes (`double` → `Double` etc.)
- Implementations of Collection or Map interfaces
 - Map elements are placed in `Map.Entry` instances
- Implementations of Enumeration or Iterator interfaces
 - possible only once per JSP due to lack of access to reset mechanism
- String of comma-separated substrings



Example: LoginServlet

```
public class LoginServlet extends HttpServlet {  
    public void doGet(HttpServletRequest request, HttpServletResponse response)  
        throws ServletException, IOException {  
        ...  
        Vector u = new Vector();  
        u.add(new User("admin", "guru"));  
        u.add(new User(username, passwd));  
        session.setAttribute("list", u);  
        ...  
    }  
}
```



Example: menu.jsp

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
  <body>
    <c:forEach var="user" items="${list}">
      <p>User ${user.username} has password ${user.passwd}.</p>
    </c:forEach>
    ...
    <p>Chosen language:<br/>
    <c:choose>
      <c:when test="${param.lang == 'is'}">íslenska</c:when>
      <c:when test="${param.lang == 'en'}">english</c:when>
      <c:otherwise>undefined</c:otherwise>
    </c:choose>
    </p>
  </body>
</html>
```

Elements of list are subsequently read into user, and tag content built with that variable value

Evaluation depends on value of request parameter lang



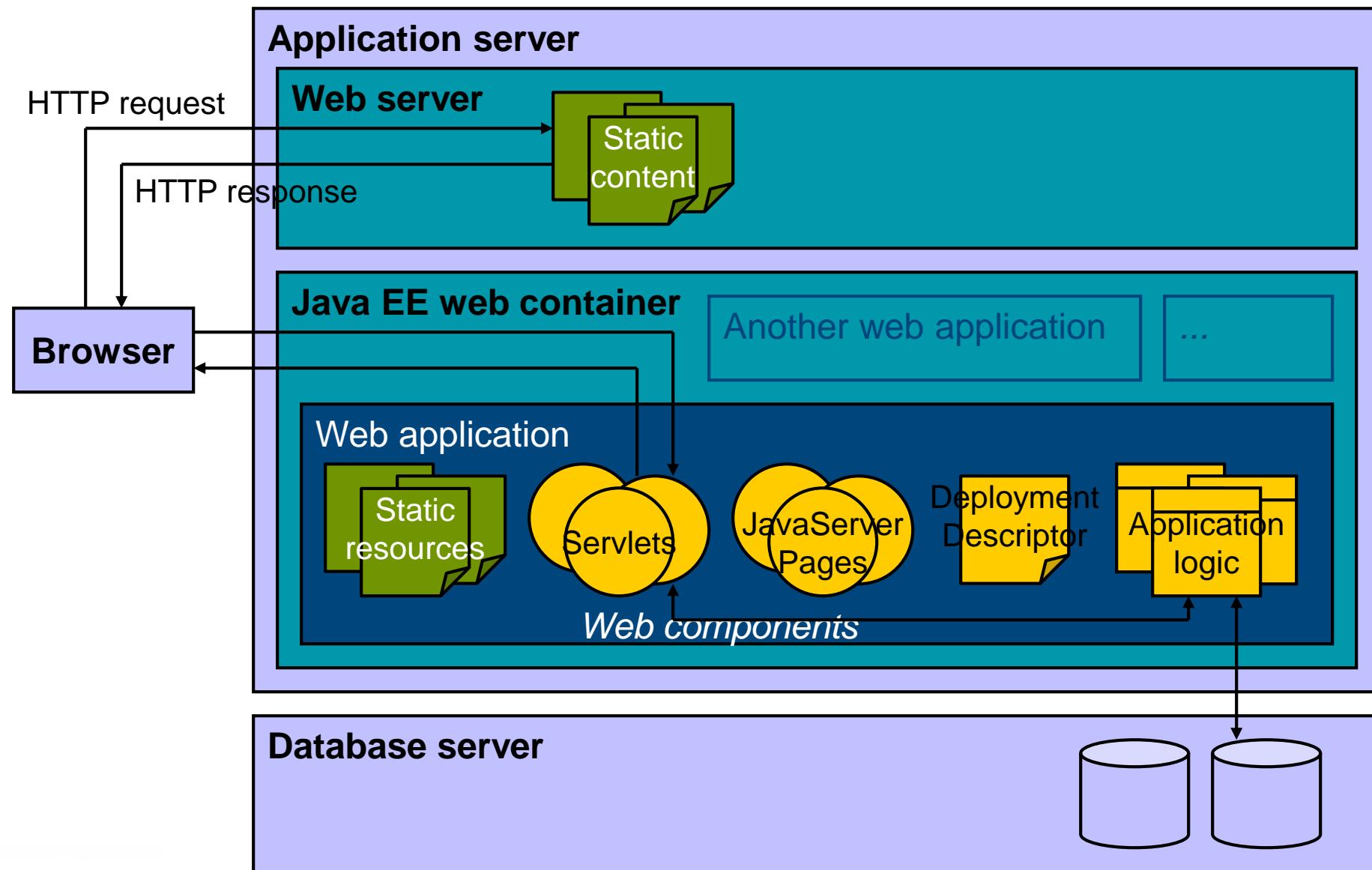
Summary: Elements of JSPs

- Static text
 - Text with HTML markup that is sent to output (HTTP response) unchanged
- Directives: <%@ ... %>
 - Commands that are interpreted at the time of generation of the servlet from the JSP, but which do not create any output
- Scripting Elements: <% ... %>, <%= ... %>
 - Java source code fragments that are executed at runtime, and whose output is incorporated into the response (*discouraged*)
- Expressions: \${...}
 - Expressions enabling simple access to objects' properties
- Actions: <jsp:...>...</jsp:...>
 - Statements that influence the behavior of the JSP at runtime
- Tags: <prefix:tag ...>...</prefix:tag>
 - Statements (self-defined or provided by the JSTL or frameworks) that influence the behavior of the JSP at runtime and may create output sent to the response

Note: Any JSP “behavior” occurs at the time of creating the HTTP response (i.e. usually creating an HTML page) on the server. It is **not** behavior executed in the client’s browser!



Summary: Components of Server-Side Web Applications





Hugbúnaðarverkefni 1 / Software Project 1

13. Final Exam Preparation

HBV501G – Fall 2015

Matthias Book



HÁSKÓLI ÍSLANDS
VERKFRAÐI- OG NÁTTÚRVÍSINDASVIÐ
ÍÐNAÐARVERKFRAÐI-, VÉLAVERKFRAÐI-
OG TÖLVUNARFRÆÐIDEILD

Lessons Learned



HÁSKÓLI ÍSLANDS

Lessons Learned

Last semester's closing questions:

- What lessons have you learned from the project experience?
- What might prevent you from following through on your good intentions in future projects?
- And how would you avoid those obstacles?

Retrospective on this semester:

- Which things worked better than last semester?
- Did you manage to follow through on your good intentions?
- What prevented you from doing so?
- How would you avoid that in the future?



Recap: Education vs. Practice

- Projects in this course are relatively simple
 - given the timeframe of three months until delivery
 - given the effort you could invest beside your other coursework
 - given the synchronization issues of learning about methods in parallel to applying them
- In industrial practice, this simplicity would make them obvious candidates for
 - an agile development process
 - a client-side web technology
- However, our learning goal is to understand how to use
 - a plan-driven process
 - object-oriented design
- Initial experience with these is best gained using a lightweight project example.
 - (Plus, complexity of the technology should motivate some of the complexity of the process.)

Course Review

In the small course projects, all the RUP and UML artifacts may seem to get in the way. **In large projects, only they will help you to keep the project under control!**

Classes

- Software Process Models
- Rational Unified Process
- Use Cases
- Project Management
- Domain Models
- Software Architecture
- Design Models
- Design Patterns
- Web Technologies

Assignments

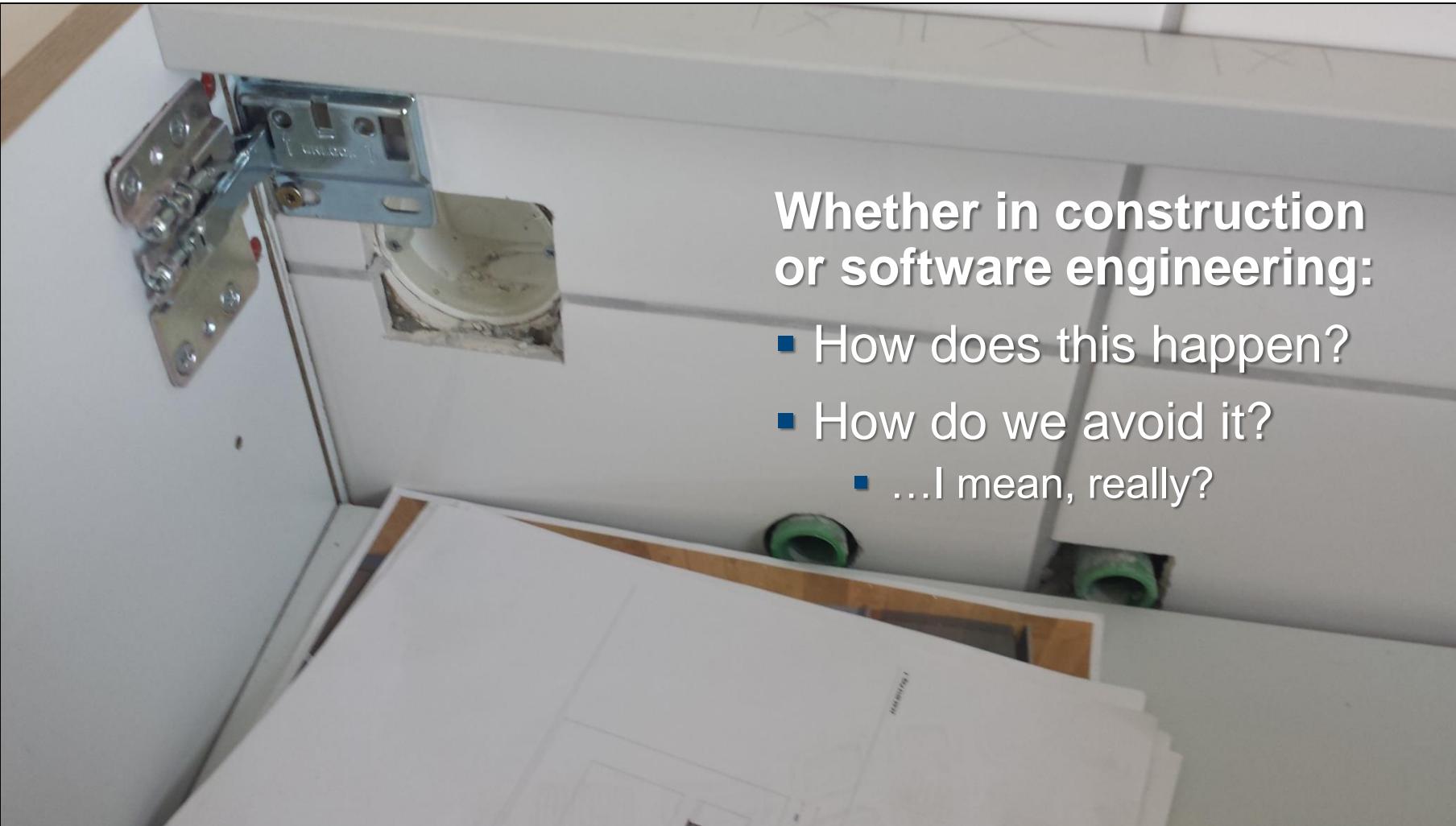
- Vision and Use Cases
- Domain Model and Software Architecture
- Design Model
- Final Product

Project

- [Choosing model]
- Understanding project scope
- Understanding domain concepts
- Understanding solution options
- Elaborating solutions
- Implementing code



What's Wrong With This Picture?



The Nature of Software Development

**“Because software is embodied knowledge,
and that knowledge is initially dispersed, tacit, latent, and incomplete,
software development is a social learning process.”**

Howard Baetjer, Jr.: Software as Capital. IEEE Computer Society Press, 1998



Project Grades and Final Exam



HÁSKÓLI ÍSLANDS

Recap: Assignment Grading

- 4 team assignments
 - All team members co-author documents with project deliverables
 - Each team member must lead brief presentation of 1-2 deliverables to tutors
 - Focus: Don't just tell us what you did, but *why* you decided to do it this way!
 - Each team member's grade depends on
 - Quality of overall teamwork and product
 - Quality of indiv. deliverable presentations
 - Assessment of contrib. by teammates
 - Assessment of competence by tutors
- All members receive the same grade
 - Can be above 10 for exceptional work
- Presenter receives bonus/malus
 - Added to or subtracted from team grade
- Grades above 10 included in averages
 - but capped at 10 for final course grade
- Averaged assignments contribute 50% to final course grade
 - Need passing grade on averaged assignments to be admitted to final exam



Bonus Points

- Each team member of an n -person team can award up to $n-1$ bonus points to their teammates.
- Bonus points can be split and distributed freely among team members.
 - Example: In a 4-person team, each member could distribute up to 3 bonus points. Alice chooses to nominate Bob for 1.5 points and Charlie for 0.5 bonus points.
- Bonus points are on same scale as grades and applied to overall (50%) project grade...
 - if you give someone 1 bonus point, you indicate their contribution should be graded 1.0 higher
- ...but they are a relative measure and will be normalized across the team
 - if you all give each other 1 bonus point, you won't boost everyone's grade by 1.0, but all stay at 0
- Bonus points should reflect a fair assessment of your own and your teammates' contribution.
 - Ultimate decision of how bonus point nominations are counted remains with tutors.
- Submit your bonus point nominations as assignment comments in Uglar by **Sun 6 Dec.**

Final Exam

- **Time:** Wed 16 Dec, 09:00-12:00
- **Focus:** Understanding of software engineering concepts and methods
- **Scope:** Lecture slides
(i.e. contents of Námsefni folder)
 - incl. worksheets on project management
 - Note: The soundtrack is relevant!
- **Style:** Written exam
 - Write in provided answer book
 - Mark answer book only with your exam number, *not* your name
- **Weight:** 50% of final course grade
 - must pass assignments and final exam to pass course

- **Tools:**
 - Any handwritten material allowed
 - i.e. blank sheets of paper with your ink
 - no photocopied notes
 - no margin notes in printed lecture slides
 - Dictionary allowed (as a book)
 - No electronic devices allowed
- **Questions:**
 - Explain / argue / discuss / calculate...
 - No optional questions
 - But answers that exceed expectations can make up for deficiencies elsewhere
- **Answers:**
 - in English
 - short paragraphs of whole sentences
 - possibly small code fragments / models



Sample Questions

- The following review questions are representative of the types of questions that could be on an exam.
- The answers to some of the following review questions can be found immediately on the lecture slides.
 - Caution: An open-book exam will obviously contain very few of these!
- Most questions require more individual reasoning and/or application of learned knowledge.
 - Be precise, justify your answers.

■ **Read exam questions carefully**

- Are you asked to explain or give an example or both?
- Are you asked to argue for or discuss an issue?
- Are you asked to explain or discuss an issue, or both?
- How many concepts are you asked to consider?

■ **Answer precisely**

- i.e. brief, focused, comprehensive



Types of Questions

- “**Explain...**”
 - Give an explanation of what something is or how something works in general.
 - Note: A concrete example is not a substitute for a general explanation.
- “**Argue...**”
 - Make up your mind about a certain issue.
 - Present arguments for your opinion on the issue.
- “**Discuss...**”
 - Consider both sides of an issue.
 - Present arguments for both sides.
 - You can state your own opinion, but should present counter-arguments as well.
- “**Suggest...**”
 - Come up with a solution for an issue.
 - Briefly explain what should be done.
- “**Give an example...**”
 - Give a brief example that illustrates the considered concept.
 - When giving examples for several distinct concepts, make sure the examples highlight the distinguishing characteristics.
- “**Point out issues...**”
 - Examine a given artifact, and explain what is wrong with it.
 - Don’t just say what is wrong, but why it is.
- “**Draw...**”
 - Draw a UML diagram reflecting the relevant aspects of the given scenario.
 - Pay attention to proper syntax!
- “**Implement...**”
 - Write/modify Java code. For modifications, place question sheet into answer book.
 - Pay attention to proper syntax!



Sample Questions: Software Process Models

- Explain how software can be both a product and a vehicle that delivers a product, and argue what that means for its creation process.
- Explain the difference between sequential, iterative-incremental, agile and plan-driven process models.
- Discuss how Eisenhower's statement “I have always found that plans are useless, but planning is indispensable” could be interpreted in the context of software engineering.

Sample Questions: The Unified Process

- Explain the relationship of phases and activities in the Unified Process.
- Explain how the phases of the Unified Process differ from the phases of the waterfall model.
- Explain how the goals of the Inception and Elaboration phase in the Unified Process differ.



Sample Questions: Project Vision and Use Cases

- Give an example of a vision statement that could appear in a RUP Vision and Scope Document.
- Point out issues with the following use cases: [...]
- Explain the difference between use cases and user stories.
- Discuss if user interface descriptions should be part of a use case.



Sample Questions: Project Management

- Name two categories of project risks, and give two examples for each of them.
- Explain how a risk-value matrix helps to prioritize risks.



Sample Questions: Software Architecture

- Explain the difference between internal and external quality attributes, and give two examples of each.
- Explain how quality requirements can influence the design and implementation of a system, and why they may require trade-offs.
- Argue which of the following quality scenarios have no architectural impact: [...]

Sample Questions: Domain Models

- Discuss advantages and limitations of using category lists to identify conceptual classes.
- Assume that an online store sells a variety of items that users can customize (e.g. t-shirts in different sizes; coffee mugs with custom lettering etc.). Draw a UML class diagram (as part of a domain model) showing how these items are listed in the store's inventory and in a customer's order.
- Explain the different meanings of arrows in UML sequence diagrams in a domain model vs. in a design model.

Sample Questions: Behavior Modeling

- Explain the difference between a decision node and a fork node in a UML activity diagram.
- Point out the modeling errors in the following UML activity diagram: [...]
- Draw a UML state machine diagram modeling the following scenario: [...]



Sample Questions: Design Models and Patterns

- Explain the issue addressed by the Creator and Expert Principles, and the solution strategy recommended by them.
- Assume you are asked to build a solution for the following scenario: [...] Draw a UML class diagram (as part of a design model) illustrating your solution, using a suitable design pattern.
- Explain which design pattern has been used in the following model, and argue whether you would have made the same choice: [...]

Sample Questions: Web Applications

- Modify the following implementation of a Java class so it can function as a controller in the Spring Web MVC framework: [...]
- Implement the entity classes that are needed to let the Java Persistence API handle the storage and retrieval of the following objects: [...]
- Modify the following implementation of a JavaServer Page so that it displays the property name of the object user that is stored in the current session, as well as the request parameter language: [...]
- Argue which HTTP request method you would use in order to transfer the following data from a client to a server: [...]

Kennslukönnun

Evaluate this course on Uglar!
(until 1 December)



Special Thanks

Elsa Mjöll Bergsteinsdóttir
Daníel Páll Jóhannsson



HÁSKÓLI ÍSLANDS



Teaching Assistants Wanted

- I am looking for TAs for next semester's courses:
 - **HBV401G Software Development**
 - **HBV601G Software Project 2**
 - If you are interested or know someone who is, please contact me at **book@hi.is** with your CV and overview of grades
- **Tasks**
 - Advising student teams
 - Scoring assignments
 - **Opportunities**
 - Help to shape the course
 - Brush up your CV
 - **Requirements**
 - Successful completion of respective course
 - Ability to advise and evaluate student teams



Seminar / Project / Thesis Topics Available

- **Interaction among software project stakeholders**

- Facilitating effective communication between team members from heterogeneous backgrounds (business, technology, management...)
- Identifying risks, uncertainties and value drivers early in a project, and focusing collaboration on these aspects rather than on business/technology trivia

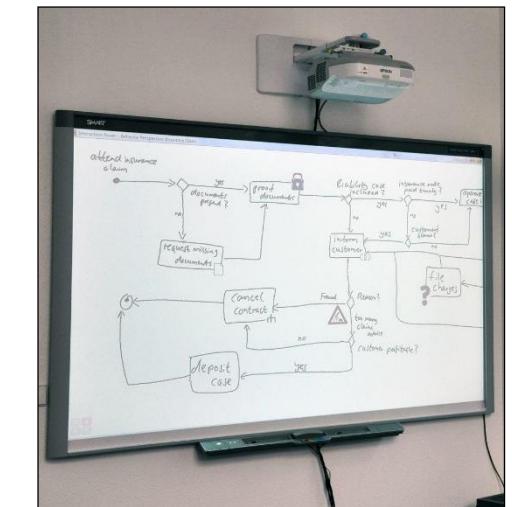
- **Multi-modal interaction with large interactive displays**

- Specifying and controlling user interactions with software systems through 2D and 3D gestures, voice commands etc.

- **Software engineering for mobile applications**

- User experience, implementation challenges, ubiquitous computing

- **Intl. collaboration with Univ. of Duisburg-Essen, Germany**



- If you are interested or know someone who is, please contact me at **book@hi.is**

Keep in Touch!

- For questions, projects, events, guest talks, student jobs, collaboration...



book@hi.is



<http://is.linkedin.com/in/matthiasbook>

Takk fyrir!

book@hi.is



HÁSKÓLI ÍSLANDS
VERKFRÆÐI- OG NÁTTÚRUVÍSINDASVIÐ

IÐNAÐARVERKFRÆÐI-, VÉLAVERKFRÆÐI-
OG TÖLVUNARFRÆÐIDEILD

