Introduction to Software Engineering

12 Project Management

Stephan Krusche

26 July 2022
Technical University of Munich
https://ase.in.tum.de

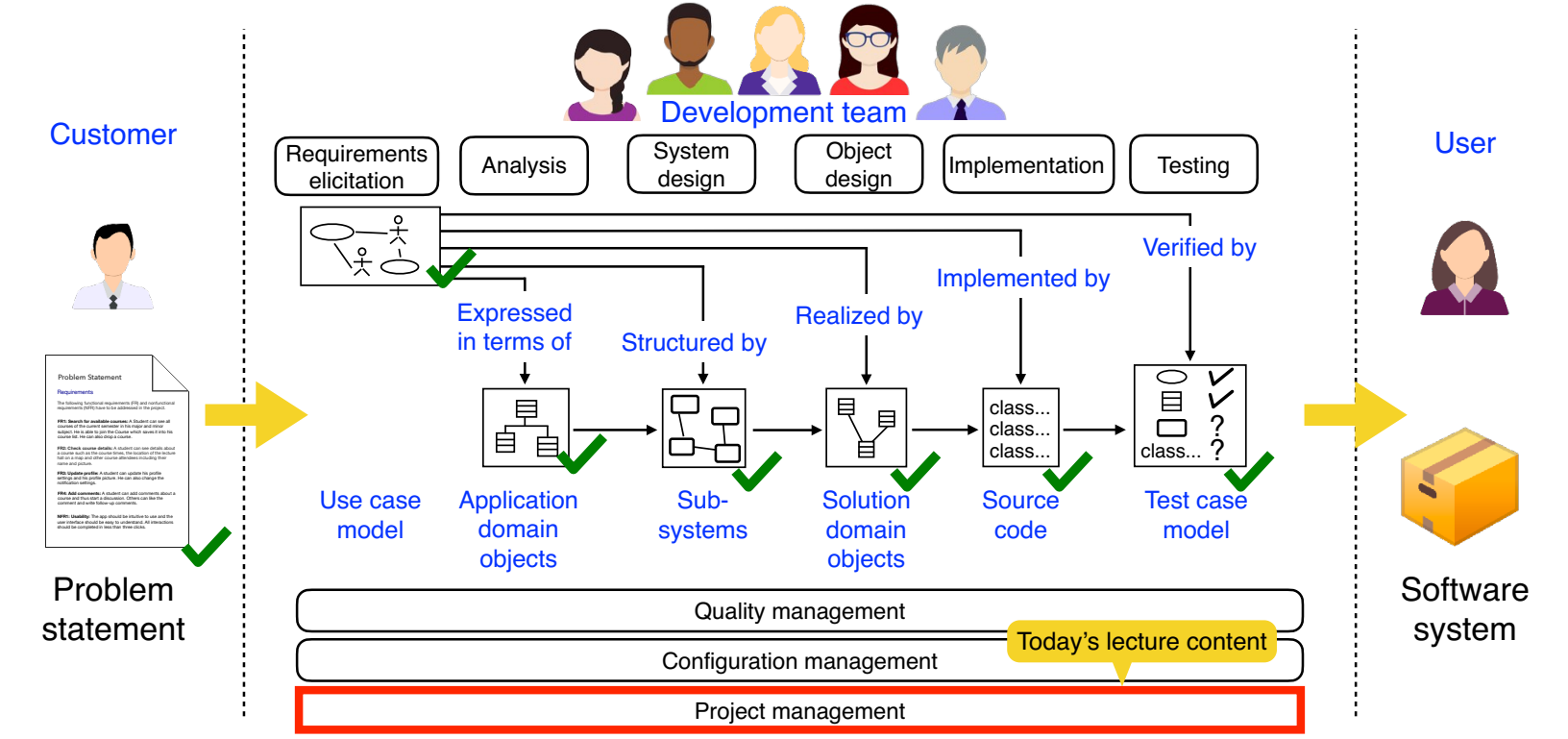
Roadmap of the Lecture

- **Context and assumptions**
  - We completed all software development lifecycle activities
  - You understand Scrum, UML diagrams, JavaFX, Gradle, REST, MVC, and patterns
- **Learning goals: at the end of this lecture you are able to**
  - Explain the differences between responsibility, authority, accountability and delegation
  - Differentiate project organization forms
  - Differentiate communication events from communication mechanisms
  - Understand communication activities to start a project

Course schedule (Garching)

#	Date	Subject
1	26.04.22	Introduction
2	03.05.22	Model-based Software Engineering
3	10.05.22	Requirements Analysis
4	17.05.22	System Design I
5	24.05.22	System Design II
6	31.05.22	Object Design I
	07.06.22	**Holiday (no lecture, no tutor groups)**
7	14.06.22	Object Design II
8	21.06.22	Testing
	28.06.22	**Guest Lecture SAP (no tutor groups)**
9	05.07.22	Software Lifecycle Modeling
10	12.07.22	Software Configuration Management
11	19.07.22	Software Quality Management
12	26.07.22	Project Management

Overview of model based software engineering



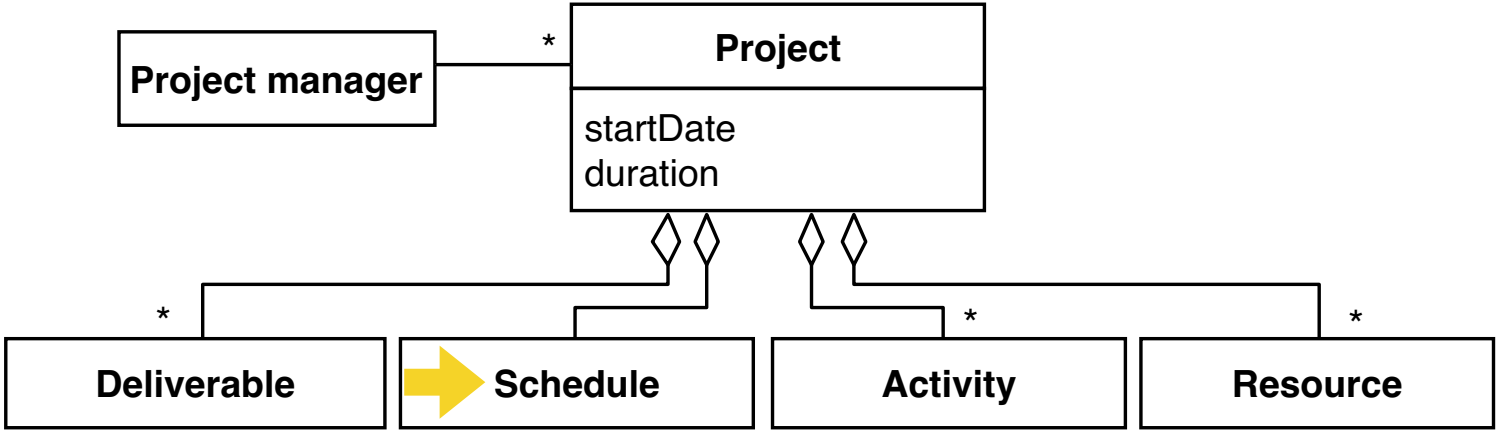
Outline

- **Project management**
- Work breakdown structure
- Organization forms
- Communication
- Course review

Definition: **project**

- Unique, temporary undertaking (endeavor), limited in time, with a clear goal and a specific budget, requiring a concerted effort to create a product, service or result
- Consists of
  - A start date and duration
  - A set of deliverables for a client
  - A schedule
  - All technical and managerial activities required to produce and deliver the deliverables
  - Resources consumed by the activities
- Managed by a project manager who
  - Administers the resources
  - Maintains accountability
  - Makes sure the project goals are met

Modeling a project: **initial** object model

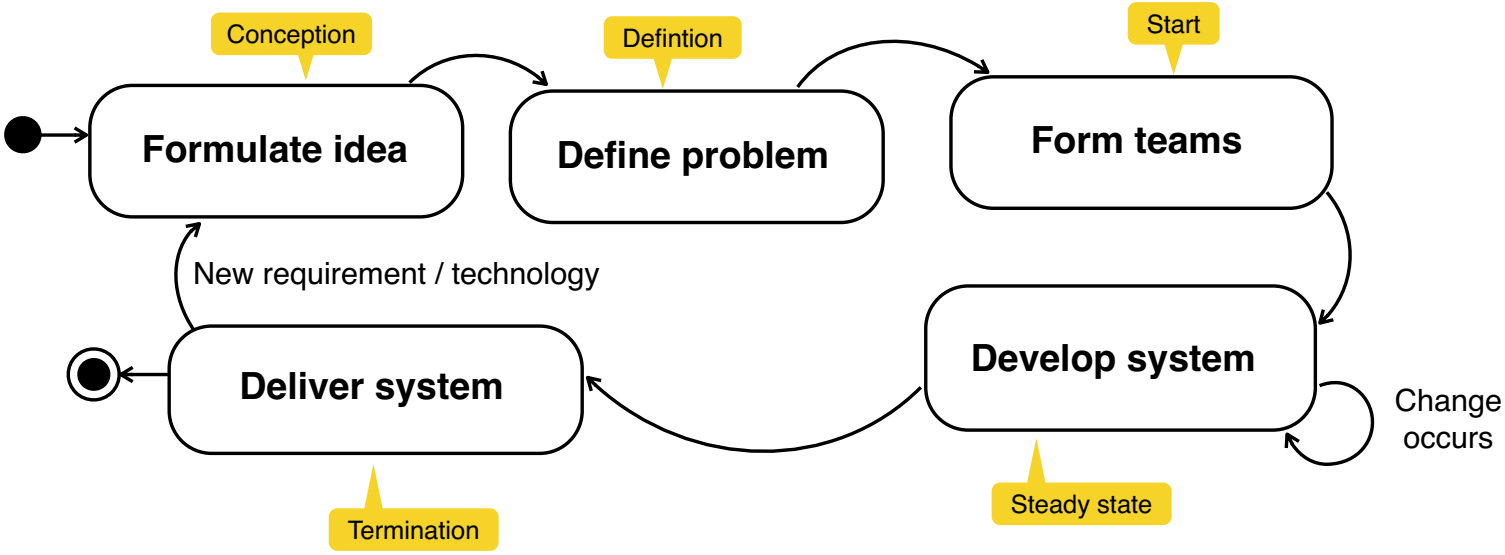


Simple dynamic model of a project (schedule)

Every project has at least 5 states

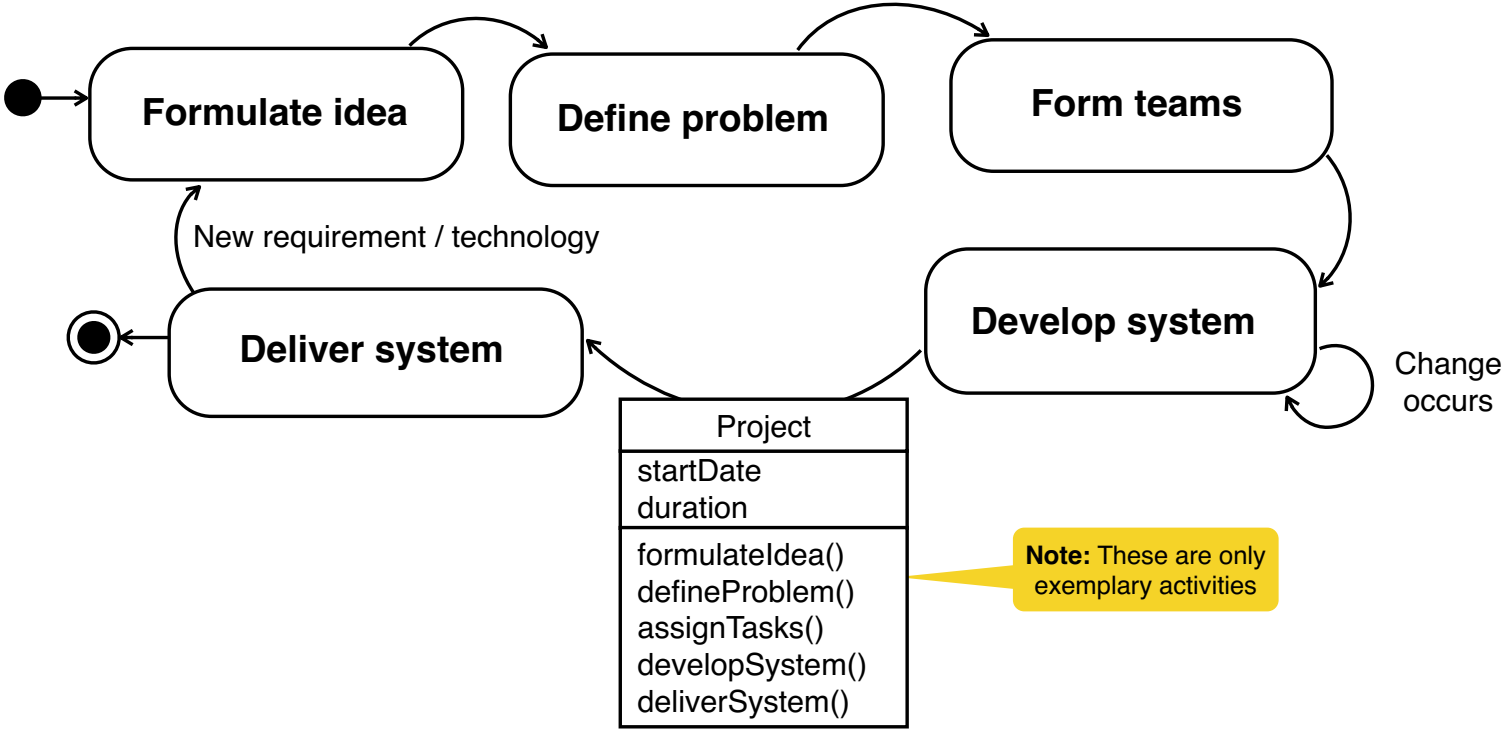
1. **Conception:** the idea is born
2. **Definition:** a plan is developed
3. **Start:** teams are formed
4. **Steady state:** the work is done
5. **Termination:** the project is finished

Modeling a project: **dynamic model**

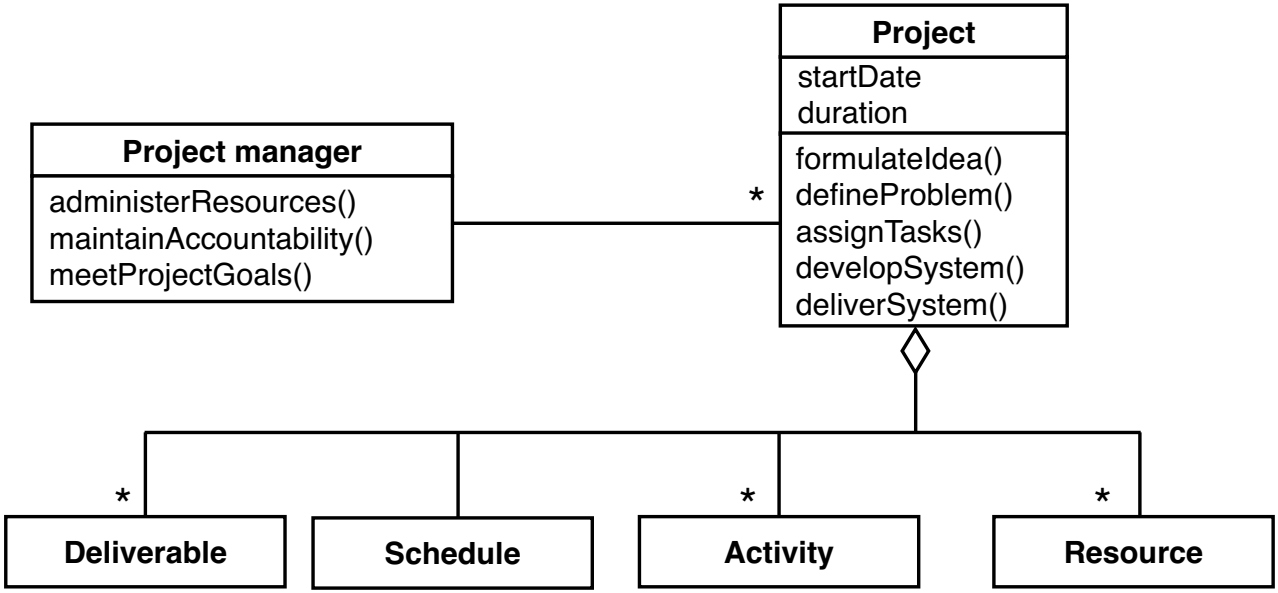


Note: this is an informal model

Modeling a project: **dynamic model**



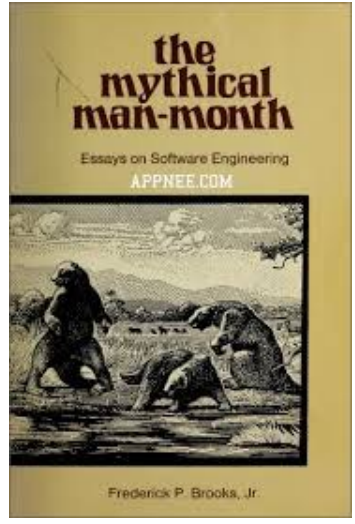
Modeling a project: **refined** object model



Laws of project management

- Projects progress quickly until they are 90% complete
  - Then they remain at 90% complete forever
- If project content is allowed to change freely, the rate of change will exceed the rate of progress
- Project teams detest progress reporting because it manifests their lack of progress
- Murphy's law
  - "When things are going well, something will go wrong"
  - "When things just can't get worse, they will"
  - "When things appear to be going better, you have overlooked something"

"Adding manpower to a late software project makes it later"
-- Frederick Brooks



Typical project management issues

- How should the project be organized?
- Who should be part of it?
- How do we break down the overall work to be done?
- How do we schedule the work?
- What are the deliverables?
- Who should do what? —> **Roles**

Role

Defines a set of **responsibilities**: duties or tasks a person is assigned to do

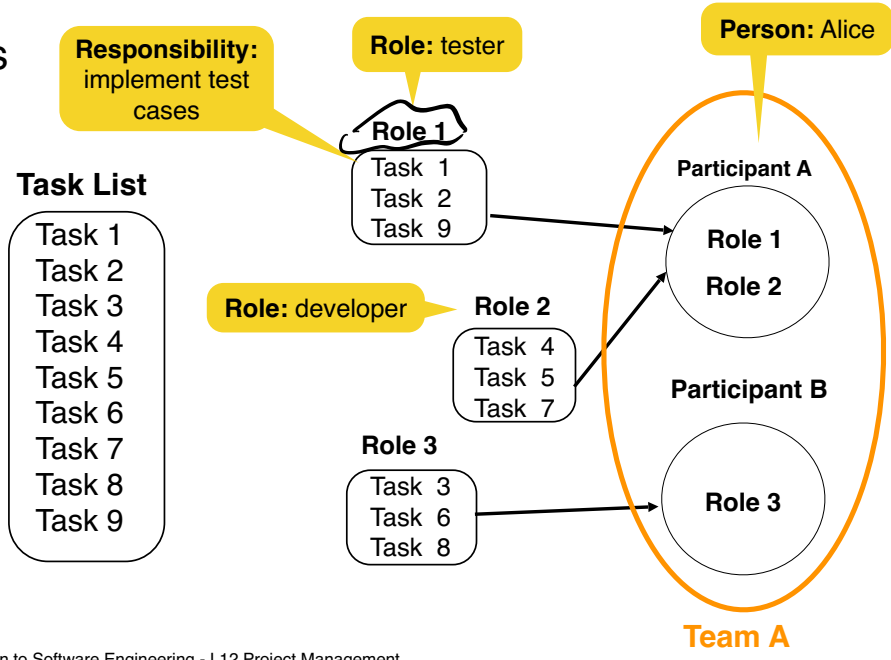
Examples of roles and corresponding responsibilities

- **Project manager**
  - Administer the resources
  - Make sure the project goals are met
- **Analyst**
  - Analyse the application domain
  - Create a taxonomy of the domain abstractions
- **System architect**
  - Decompose the system into subsystems
  - Choose a software architectural style
- **Tester:** design and implement tests

Roles and responsibilities

- Responsibilities (e.g. in the form of specific tasks) are assigned to roles
- Roles are assigned to people
- People are assigned to teams

Task
↓
Role
↓
person
↓
Team



Assignment of roles to participants

- **One to one:** ideal but rare
- **Many to few**
  - Each project member assumes several "hats"
  - Danger of over-commitment
- **Many to “too many"**
  - Some people don't have significant roles
  - Lack of accountability
  - Losing touch with project
- Problems in role assignments
  - **Incompetence:** the wrong person fills the wrong role
  - **Useless role:** the role exists only to minimize damage control
  - **Increase of bureaucracy:** the role swells unnecessarily

Examples for refactored solutions

- **Dealing with incompetence:** do not promote your most brilliant engineer to management 能力浪费
- **Dealing with useless roles:** put individuals to work in their core competencies
- **Dealing with increased bureaucracy**
  - Improve estimation
  - Don’t wait until the last minute

Key concepts for mapping roles to people

- **Authority:** the ability to make binding decisions between people and roles
- **Responsibility:** the commitment of a role to achieve specific results
- **Accountability:** tracking a performance of a task to a specific person
- **Delegation:** binding a task assigned to one person to another person

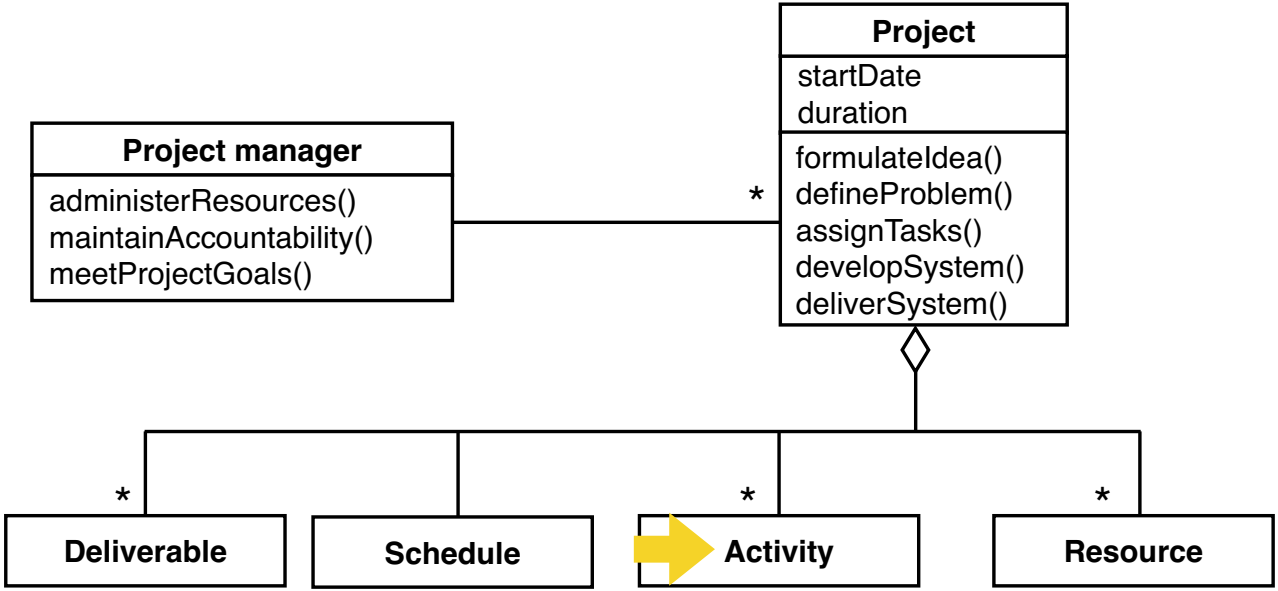
Delegation in project management

- Binding a task assigned to one person to another person
- 3 main reasons for delegation
  1) **Time management:** free yourself up for other tasks
  2) **Expertise:** the most qualified person makes the decision
  3) **Training:** develop another person's ability to handle additional assignments

➡ **You can delegate work, but you cannot delegate responsibility**

➡ **You can only share responsibility**

Review: refined object model of a project

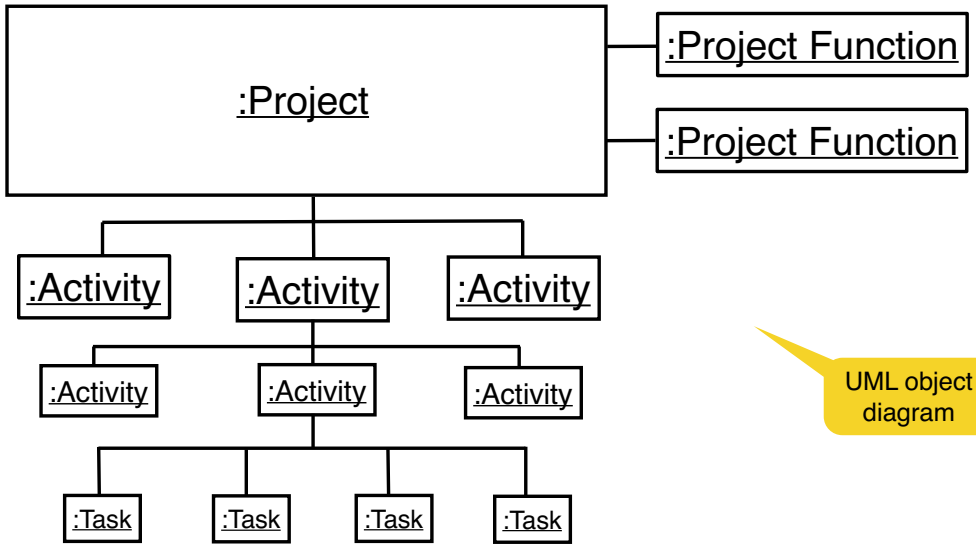


Outline

- Project management
- **Work breakdown structure** 拆分工作
- Organization forms
- Communication
- Course review

Example of a project's work breakdown structure

A project includes project functions, activities and tasks



Activities

- Major work that culminates in a project milestone
  - A project milestone is a scheduled event used to visualize/measure progress
  - A project milestone is visible to the customer
  - A project milestone usually produces a baseline
- Can have internal checkpoints (not externally visible)
- Allow to separate concerns
- There is often a precedence relation
  - Example: “activity A1 must be finished before activity A2 can start”

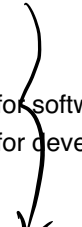
Review: examples of activities in a software project

- Requirements elicitation
- Analysis
- System design
- Rationale management
- Software configuration management
- Object design
- Implementation
- Testing

Some of these activities span the duration of a project —> **project functions**

Project function

- An activity that spans the entire duration of a software project
- Examples of project functions include [1]
  - Project management
  - Software configuration management
  - Quality management
  - Continuous integration
  - Release management
- Sometimes, project functions are also called cross development processes or integral processes [2]



[1] IEEE 1058-1998: Standard for software project management plans
[2] IEEE 1074-2006: Standard for developing a software project life cycle process

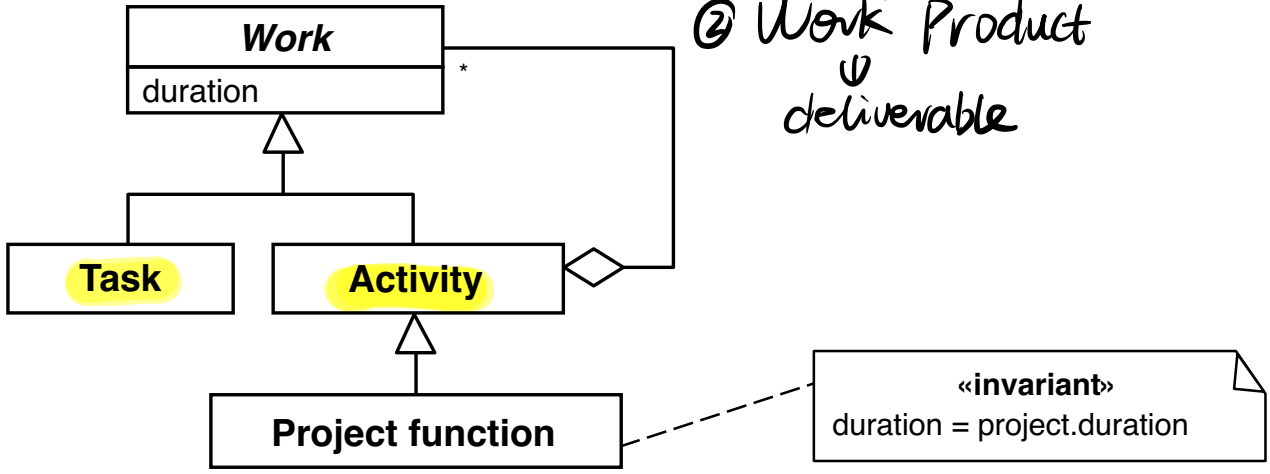
Task

- Describes the smallest amount of work monitored by the project manager
- Typically less than 2-4 working days effort
- Associated with
  - Role
  - Work package
  - Work product
  - Start date
  - Duration
  - Required resources

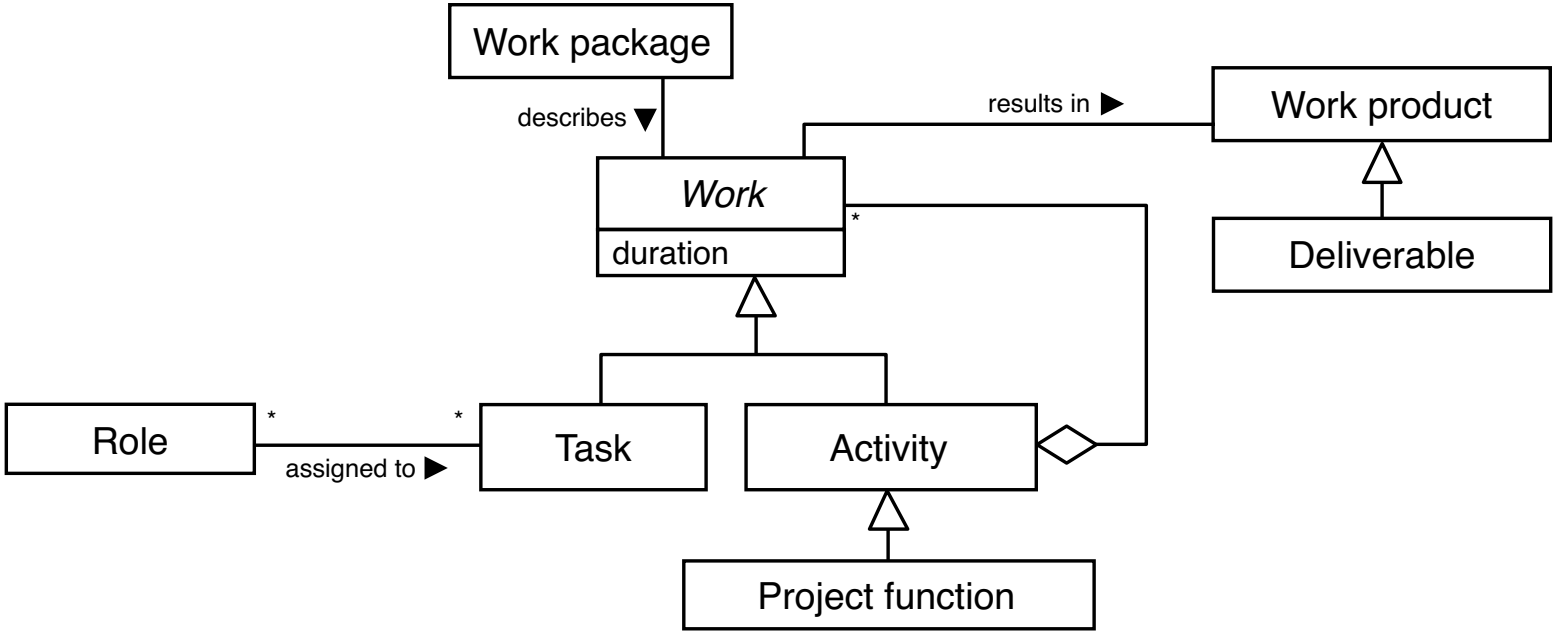
Work

- Activities are often grouped into **higher-level** activities, e.g.
  - Phase 1, phase 2, …., phase n
  - Step 1, step 2, …., step n
- Work: A task or an activity that contains other tasks and lower-level activities

① Work Package
[task or activity]
② Work Product
deliverable



Model of a work breakdown structure



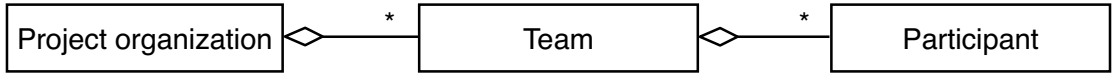
Work breakdown structure: the aggregation of the work to be performed in a project
Often called **WBS** (in traditional projects) or **epics** (in agile projects)

Outline

- Project management
- Work breakdown structure
- **Organization forms**
- Communication
- Course review

Project organization

- Defines the relationships among resources (in particular participants) in a project
- A project organization should define
  - Who decides what (decision structure)
  - Who reports their status to whom (reporting structure)
  - Who communicates with whom (communication structure)
- 3 types
  - Functional organization
  - Project-based organization
  - Matrix organization



Functional organization (按功能)

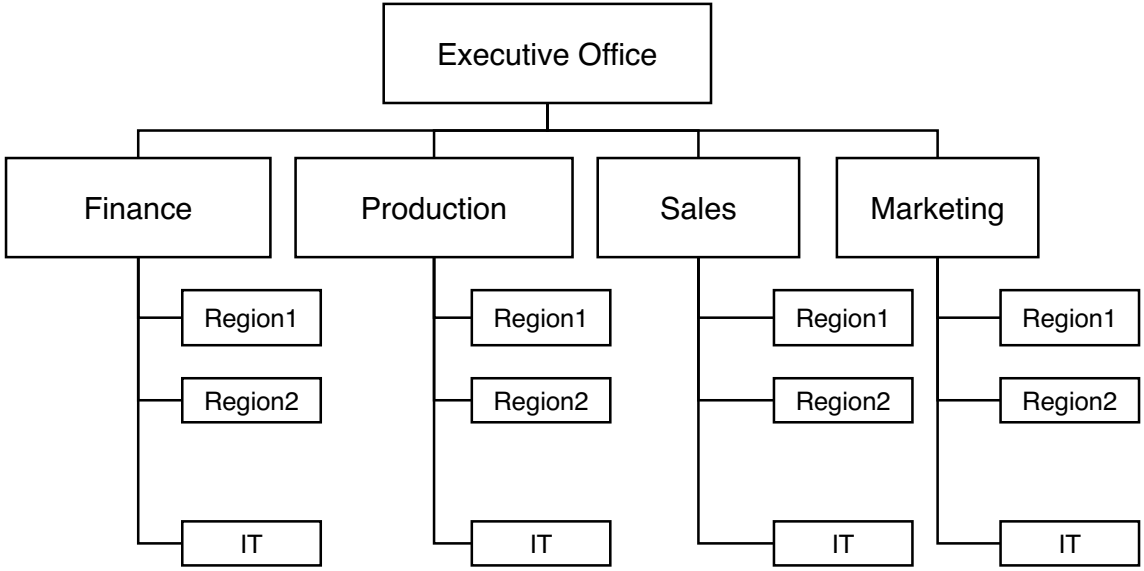
- People are grouped into departments, each of which addresses one activity ("function")
- Examples of departments

  - In traditional companies: finance, production, sales, marketing
  - In software companies additionally: analysis, design, integration, testing, delivery
- Properties of functional organizations
  - Projects are pipelined through the departments
    - Example: the project starts in research, moves to development, then moves to production
  - Different departments often address identical needs
    - Example: configuration management, IT infrastructure

one thing.

  - Only few participants are completely involved in a single project

Example of a functional organization



Also called **line organization**

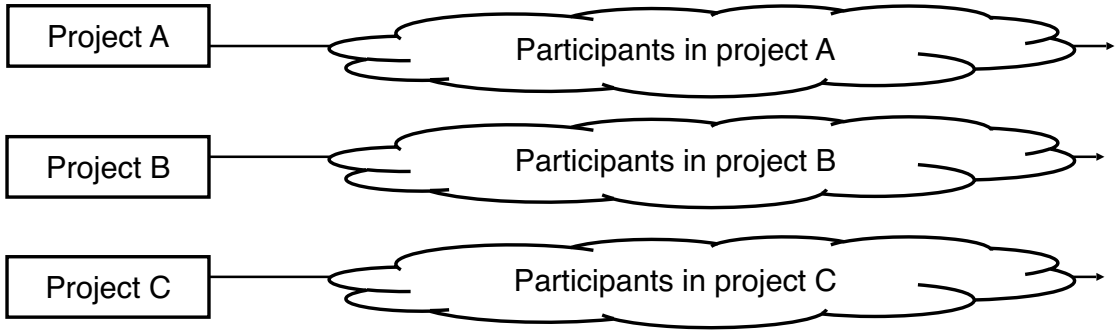
Properties of functional organizations

- **Advantage**
  - \+ Members of a department have a good understanding of the functional area they support
- **Disadvantages**
  - \- It is difficult to make major investments in equipment and facilities
  - \- High chance of work duplication or overlap among departments

Project-based organization

- People are assigned to one of the several project in the organization, each of which has a problem to be solved in a certain time within a given budget
- Key properties of project-based organizations
  - Teams are assembled when a project is created
  - Each project has a project manager
  - A participant is involved only in a single project
  - Teams are disassembled when the project terminates

Example of project-based organization



Properties of project-based organizations

- **Advantages**
  - \+ Responsive to new requirements
    (the project is newly established and can be tailored around the problem)
  - \+ New people familiar with the problem or with special capabilities can be hired
  - \+ There is no idle time for the project members
- **Disadvantages**
  - \- Teams cannot be assembled rapidly: often difficult to manage the staffing/hiring process (flat staffing vs. gradual staffing)
  - \- Roles and responsibilities need to be defined at the beginning of each project (because there are no predefined departments as in a functional organization)

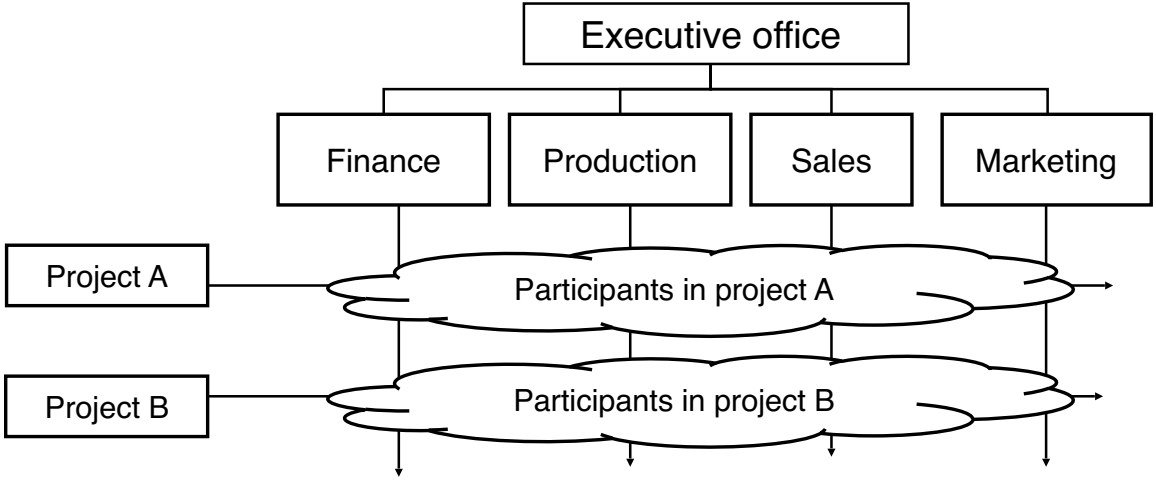
flexible (对比流水线)　challenging

When to use which organization type?

- **Functional organization**
  - Projects with high degree of certainty, stability, uniformity and repetition
  - Requires little communication
  - Role definitions are clear
  - The more people on a project, the more the need for a formal structure
- **Project-based organization**
  - Project has high degree of uncertainty
  - Open communication is needed among participants
  - Roles are defined on project basis
  - Requirements are likely to change during the project
  - A new technology that could affect the outcome may appear during the project

Matrix organization

- People from different departments of a functional organization are assigned to work on one or more projects
- Project manager and participants are usually assigned to a project with less than 100 % of their time



Properties of matrix organizations

- **Advantages**
  - \+ Teams for projects can be assembled rapidly from the departments
  - \+ Expertise can be applied to different projects as needed
  - \+ Consistent reporting and decision procedures can be used for projects of the same type
- **Disadvantages**
  - \- Team members are often not familiar with each other
  - \- Team members have different working styles

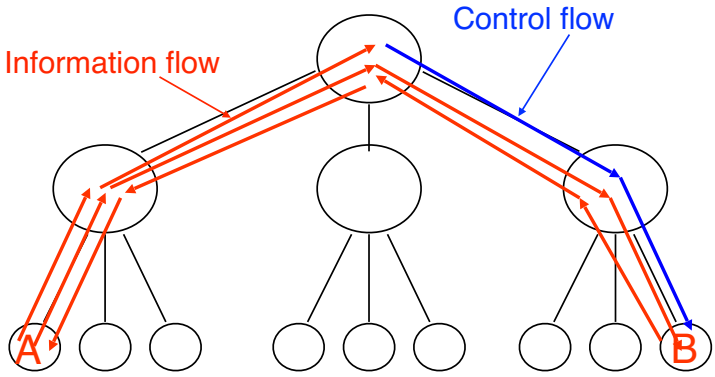
Challenges in matrix organizations

- Team members working on multiple projects have competing demands for their time
- Multiple work procedures and reporting systems are used by different team members
- **Double boss problem:** team members must respond to two different bosses with different focuses
  - **Focus of the department manager**: assignments to different projects, performance appraisal
  - **Focus of the project manager**: work assignments to project members, support of the project team, deliver project in time and within budget
- Department and project interests might be in conflict with each other

Project organization structures

- A project organization has at least 3 structures that model the relationships between people
  1) **Decision structure** models the control flow: who decides what?
  2) **Reporting structure:** who reports their status to whom?
  3) **Communication structure** models the information flow: who facilitates communication with whom?

Example of information and control flow in a line organization



Chief executive

First level manager

Project members

A wants to talk to B: complicated communication flow

A wants to make sure B makes a certain change: complicated decision flow

Information and control flow along hierarchical boundaries

Observations on project management structures

- Information flow in a hierarchical project organization does not work well with unexpected changes
- The manager is not necessarily always right and might even misunderstand communication requests
- Improving information flow through **non**-hierarchical project organizations
  - \+ Cut down bureaucracy (direct communication is possible)
  - \+ Reduce development time
  - \+ Better communication between multiple teams
  - \+ Decisions are expected to be made at each level
  - – Hard to manage (who is in control in case of conflicts?)

Outline

- Project management
- Work breakdown structure
- Organization forms
- **Communication**
- Course review

Communication skills

- Project managers **and** software engineers need to acquire several skills
  - **Collaboration:** negotiate requirements with the client and with members from your team and other teams
  - **Presentation:** present a major part of the system during a review
  - **Technical writing:** write a part of the proposal, or a part of the project documentation
  - **Management:** facilitate a team meeting, find compromises, negotiate between conflicting demands

→ In large system development efforts, you will spend more time communicating than coding

→ **Technology manager**

Communication skills

- Clear and accurate communication is critical for the success of a project
- Modes of communication (also called communication events)
  - Planned communication
  - Event-driven communication
- Difference between communication events and communication mechanisms

Communication event vs. mechanism

Communication event: information exchange with defined objectives & scope

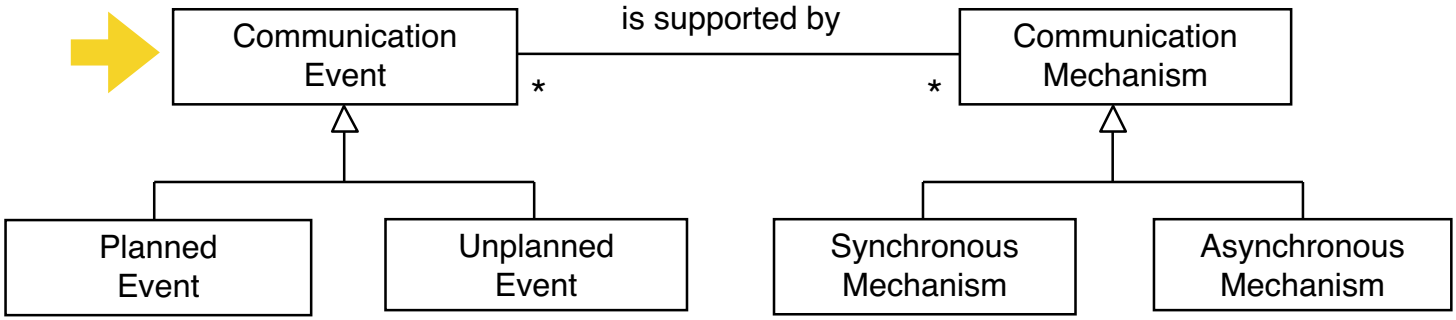
- **Scheduled:** planned communication
  Examples: review, meeting
- **Unscheduled:** event-driven communication
  Examples: request for change, clarification, bug report

Communication mechanism: tool or procedure that can be used to deal with a communication event

- **Synchronous:** same time
- **Asynchronous:** different time pull request email

Another distinction can be made: **formal** vs. **informal** communication

Modeling communication



Communication events (examples)

- **Problem definition:** focus on scope
  - **Objective:** present goals, requirements and constraints
  - Example: client presentation
  - Usually scheduled at the beginning of a project
- **Project review:** focus on system models
  - **Objective:** assess status and review the system model
  - Example: analysis review, system design review
  - Scheduled after each project milestone
- **Client review:** focus on requirements
  - **Objective:** brief the client, agree on requirements changes
  - Example: requirements review, prototype review
  - The first client review is usually scheduled after the analysis phase

Communication mechanisms (synchronous examples)

- **Informal meeting**
  - Example: meeting at the coffee machine, hallway meeting
  - **Supports**: unplanned conversations, request for clarification, request for change
  + Cheap and effective for resolving simple problems
  – Information loss, misunderstandings are frequent
- **Formal meeting**
  - Example: face to face, telephone conference tool, video conference tool
  - **Supports**: planned conversations, client review, project review, status review, brainstorming, issue resolution
  + Effective for issue resolutions and consensus building
  – High cost (people, resources)

Communication mechanisms (asynchronous examples)

- **E-Mail**
  - **Supports**: release, change request, brainstorming
  - \+ Ideal for planned and formal communication and announcements
  - – E-mail taken out of context can be misunderstood, sent to the wrong person or lost
- **Chats**
  - **Supports**: release, change request, brainstorming
  - \+ Suited for discussion among people who share a common interest; cheap (shareware available)
  - – Rather informal
- **Wikis**
  - **Supports**: release, change request, inspections
  - \+ Documents contain links to other documents
  - – Does not easily support rapidly evolving documents

Summary

- Projects are concerted efforts towards a goal within a limited time
- Project participants are organized in terms of teams, roles, control and communication relationships
- An individual can fill more than one role
- Work is organized in terms of activities and tasks
  - Tasks are assigned to roles
  - Tasks produce work products
- 3 project organization forms: functional, project-based, matrix
- Communication is critical: formal vs. informal, mechanisms vs. events

Literature

- F. P. Brooks: The Mythical Man Month: Anniversary Edition: Essays on Software Engineering. Addison-Wesley, Reading, MA, 1995
  - Also available as PDF File: https://www.researchgate.net/publication/220689892_The_Mythical_Man-Month_Essays_on_Software_Engineering
- G. M. Weinberg: The Psychology of Computer Programming, Van Nostrand, New York, 1971
- D. J. Paulish: Architecture-centric Software Project Management, SEI Series in Software Engineering, Addison-Wesley, 2001
- E. Raymond: The Cathedral and the bazaar, 1998 http://manybooks.net/titles/raymondericother05cathedralandbazaar.html

Introduction to Software Engineering

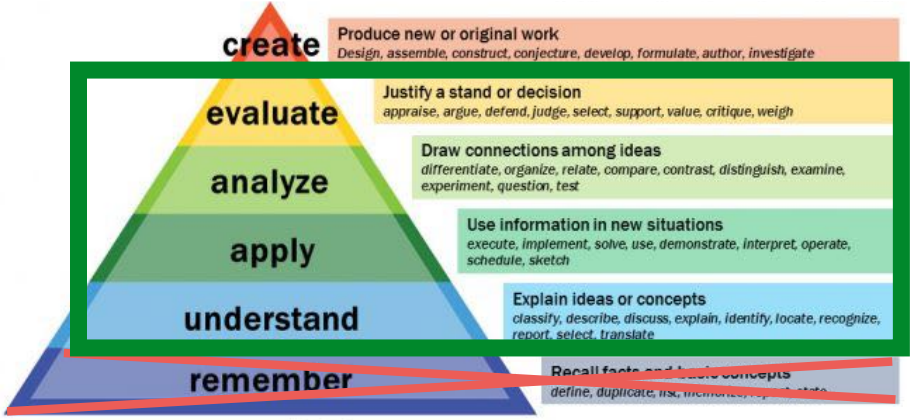
12 Course Review

Stephan Krusche

26 July 2022
Technical University of Munich
https://ase.in.tum.de

Final exam first lecture -- information

- The exercises focus on **understanding** and **problem solving**
- You **cannot** pass the GOE just with "learning by heart"
- Make sure that you are able to apply software engineering concepts to problem statements
- Review the learning goals of each lecture



Final thoughts and suggestions for practicing

- **Important:** if a topic is not mentioned in the course review, it can still be part of the graded online exercise
- Review the Graded Online Exercise Tutorial
- Review the Exam Mode Students' Guide
- Review the exercises: in-class, group work, team work and homework
- Repeat all the quizzes on Artemis using the **practice** mode

Course topics

1. **Software engineering as a problem solving activity**
2. Abstraction and modeling
3. Requirements analysis
4. System design and architectural patterns
5. Object design and design patterns
6. Testing
7. Software lifecycle modeling
8. Software configuration and release management
9. Software quality management
10. Project management

Why is software development difficult?

- The problem is usually ambiguous (e.g. impossible trident)
- Requirements are usually unclear and change when they become clearer
- The problem domain (also application domain) is complex and so is the solution domain
- The development process is difficult to manage
- Software is a discrete system
  - Continuous systems have no hidden surprises
  - Discrete systems can have hidden surprises! (Parnas)

David Lorge Parnas - an early pioneer in software engineering who formulated in 1972 the concepts of modularity and information hiding which are the foundation of object oriented methodologies

Software engineering: a problem solving activity

- **Analysis:** understand the nature of the problem and break the problem into pieces
- **Synthesis:** put the pieces together into a larger structure

divide and conquer

➡ Techniques, methodologies and tools

Techniques, methodologies and tools

- **Techniques**
  - Formal procedures for producing results using some well defined notation
  - Example: recipe, quick sort algorithm
- **Methodologies**
  - Collection of techniques applied across software development and unified by a philosophical approach
  - Examples: cookbook, object oriented analysis and design, functional decomposition
- **Tools**
  - Instruments or automated systems to accomplish a technique
  - Examples
    - Compiler, editor, debugger
    - Integrated development environment (IDE)
    - Modeling editors, computer aided software engineering (CASE)

Course topics

1. Software engineering as a problem solving activity
2. **Abstraction and modeling**
3. Requirements analysis
4. System design and architectural patterns
5. Object design and design patterns
6. Testing
7. Software lifecycle modeling
8. Software configuration and release management
9. Software quality management
10. Project management

Abstraction

- Allows us to ignore unessential details
  1. Abstraction is a **thought process** (activity) where ideas are distanced from objects
  2. Abstraction is the **result** (entity) of a thought process

➡ Abstractions can be expressed with a model

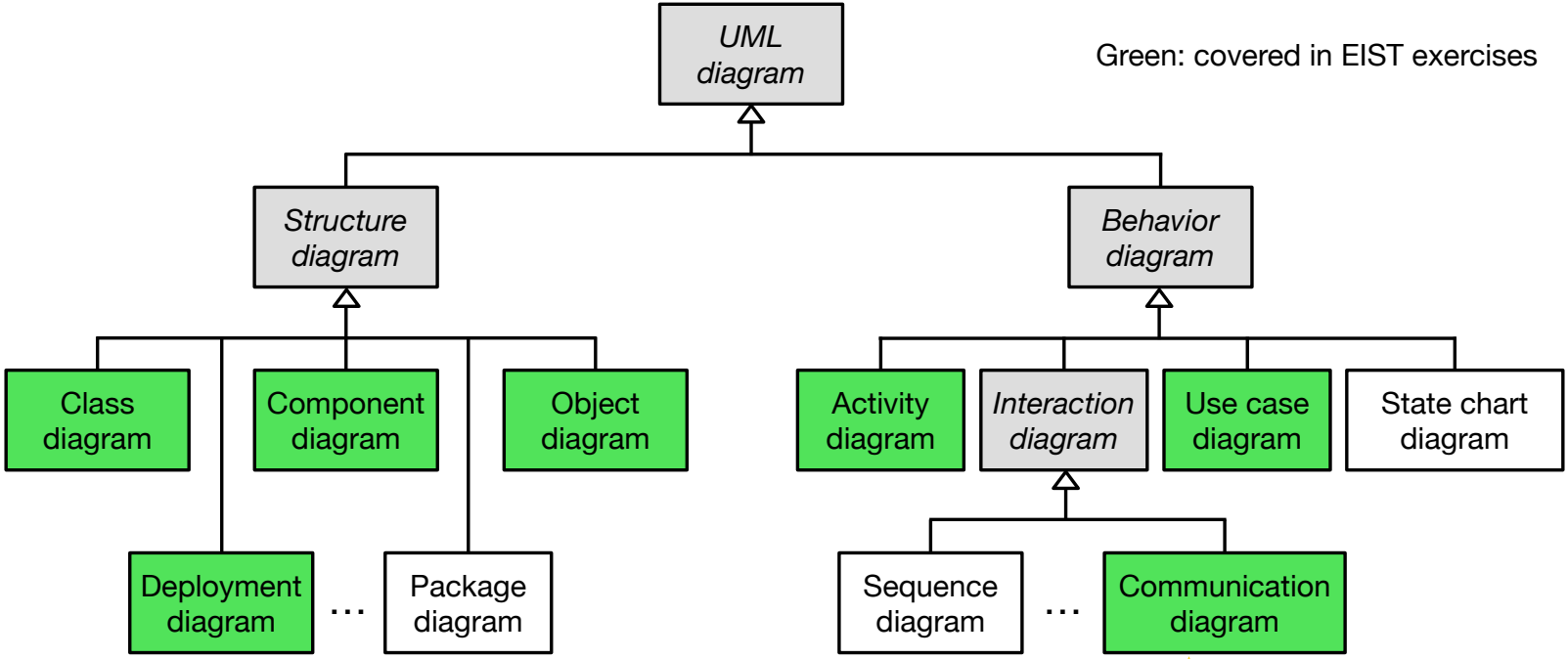
Models to describe software systems

- Object model: what is the structure of the system?
- Functional model: what are the functions of the system?
- Dynamic model: how does the system react to external events?

- **System model:** object model + functional model + dynamic model

- Object model: class diagrams, object diagrams, communication diagrams, deployment diagrams
- Functional model: scenarios, use case diagrams
- Dynamic model: communication diagrams, activity diagrams

Overview of UML diagrams

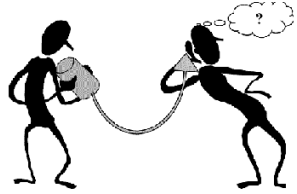


Why do we use UML?

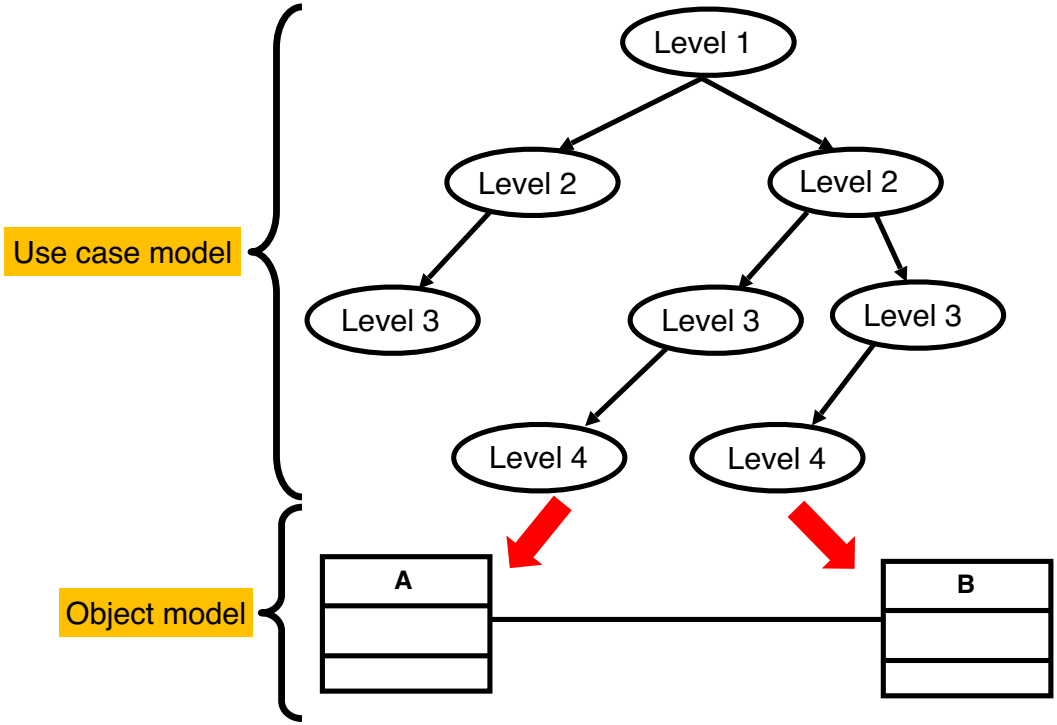
It reduces complexity by **focusing** on abstractions

It can be seen as a high level "programming language" enabling the **generation** of source **code**

It is a means of **communication** between people involved in a software project



Model-based software engineering approach



Top level use cases ("business processes")

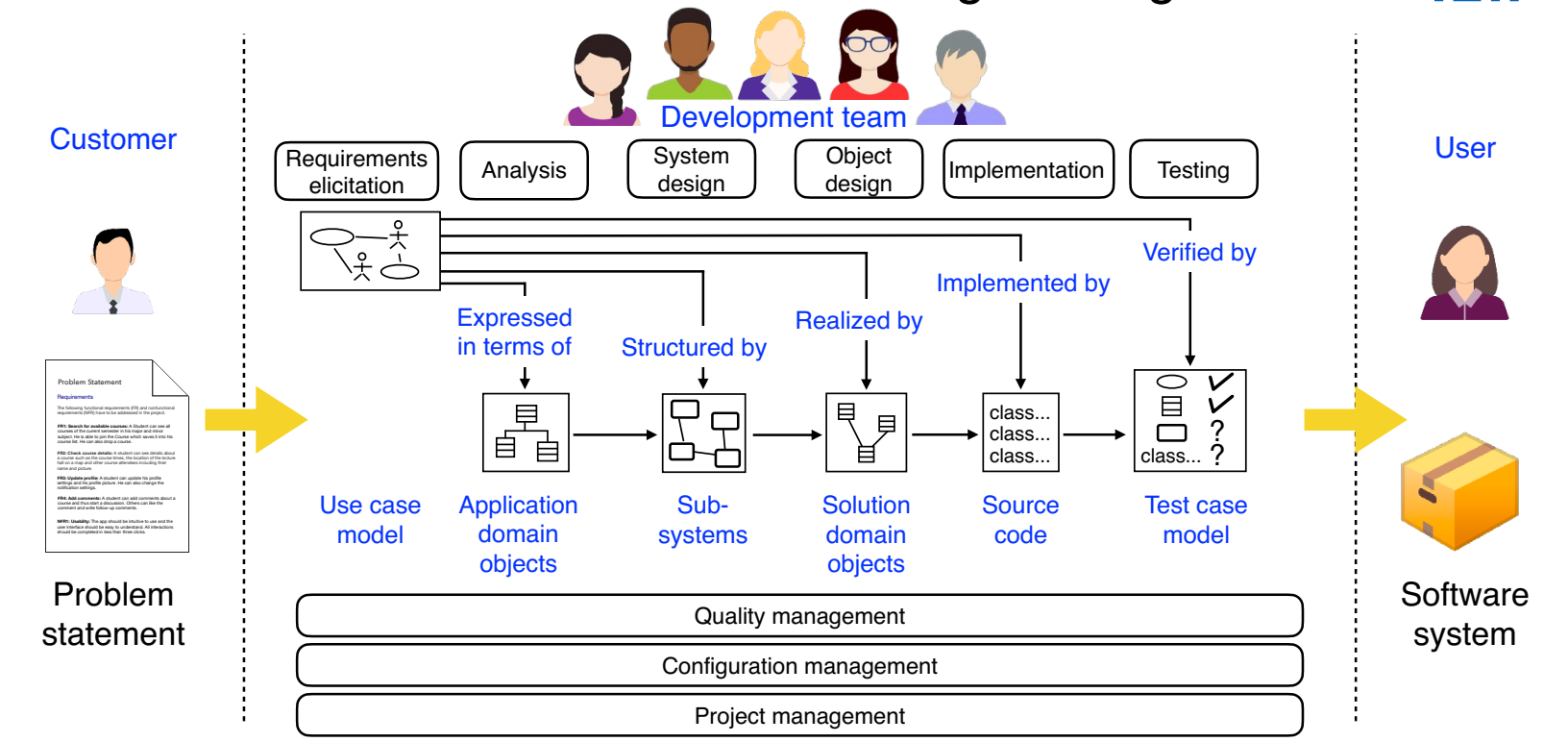
Level 2 use cases

Level 3 use cases

System functions

A and B are called participating objects

Overview of model based software engineering

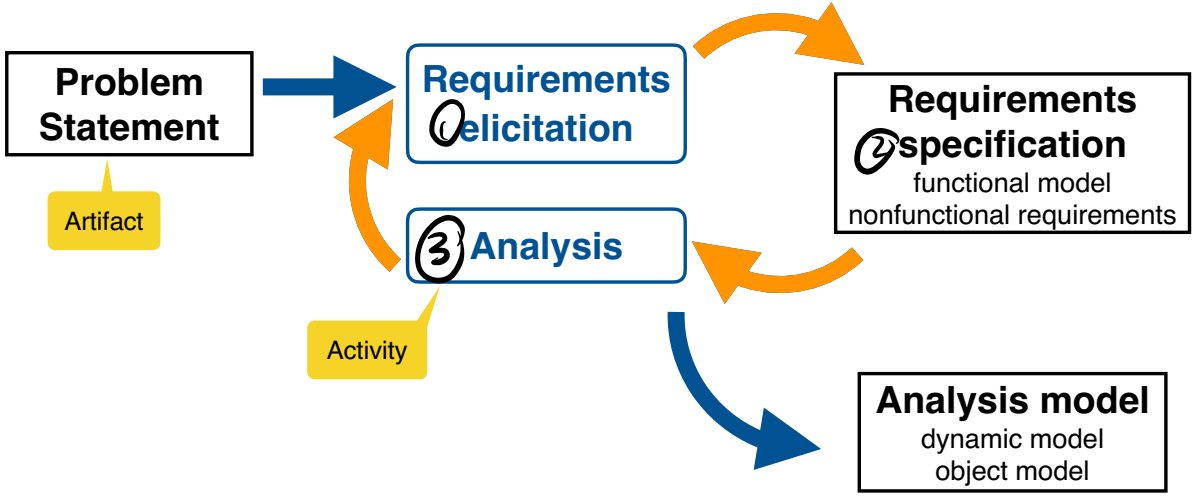


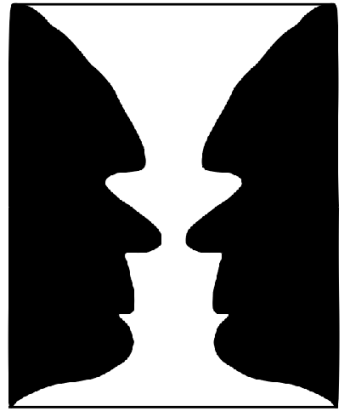
Course topics

1. Software engineering as a problem solving activity
2. Abstraction and modeling
3. **Requirements analysis**
4. System design and architectural patterns
5. Object design and design patterns
6. Testing
7. Software lifecycle modeling
8. Software configuration and release management
9. Software quality management
10. Project management

Overview: requirements engineering

- **Requirements elicitation:** describe the purpose of the system
- **Analysis:** create a model of the system, which is correct, complete, consistent, and verifiable





An optical illusion

Ambiguity in drawings; we need to be consistent in our models

Types of requirements

- **F**unctionality: what is the software supposed to do?
  - External interfaces (—> actors): interaction with people, hardware, other software

Functional requirements

- Quality requirements
  - **U**sability
  - **R**eliability
  - **P**erformance
  - **S**upportability

+ measurable !

- Constraints (pseudo requirements)
  - Required standards, operating environment, etc.

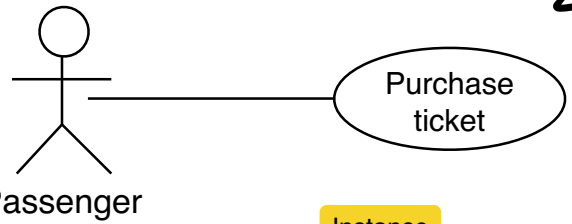
Nonfunctional requirements

➡ **FURPS** is an acronym representing a model for classifying software attributes (functional and nonfunctional requirements)

Scenario example (natural language) ⇒ Problem Statement

Joe wants to take the subway from Munich Marienplatz to Garching Forschungszentrum and selects a single day ticket for Munich Zone M-2. The ticket machine displays a price of 10,10€. Joe inserts a 20€ bill. The ticket machine returns 9,90€ and prints the single day ticket. Joe takes the change of 9,90€ and the ticket and goes to the U6.

Scenario example (formalized)



1) Name
2) Participating actors
3) Flow of events

Instance

1) Name: Purchase ticket

2) Participating actors:

Joe: Passenger

Instance

3) Flow of events

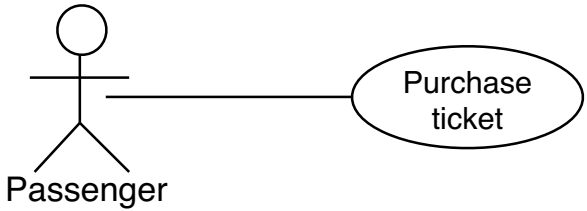
Actor step

1. Joe wants to take the subway from Munich Marienplatz to Garching Forschungszentrum and selects a single day ticket for Munich Zone M-2

System step (indented)

2. The ticket machine displays a price of 10,10€
3. Joe inserts a 20€ bill
4. The ticket machine returns 9,90€
5. The ticket machine prints the single day ticket
6. Joe takes the change of 9,90€ and the ticket and goes to the U6

Textual use case description: example



1) Name
2) Participating actors
3) Flow of events
4) Entry conditions
5) Exit conditions
6) Special requirements

1) Name: Purchase ticket

2) Participating actors: Passenger

Abstract version of the previous scenario

3) Flow of events

1. The passenger selects the number of zones to be traveled
2. The ticket machine displays the amount due
3. The passenger inserts at least the amount due
4. The ticket machine returns change
5. The ticket machine issues the ticket

4) Entry conditions

- The passenger stands in front of the ticket machine
- The passenger has sufficient money to purchase a ticket

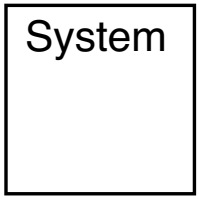
5) Exit conditions

- The passenger has the ticket

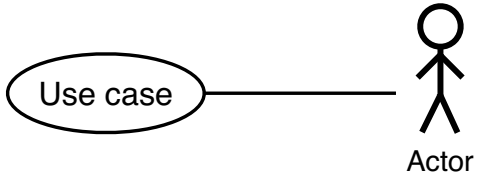
6) Special requirements

- The ticket machine is connected to a power source

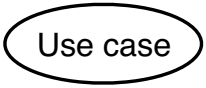
UML use case diagram: overview of all elements



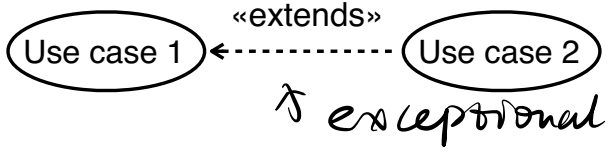
System boundary



Association

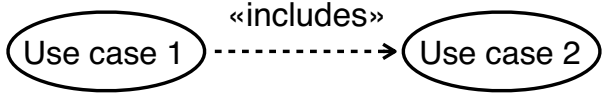


Use case



Extension

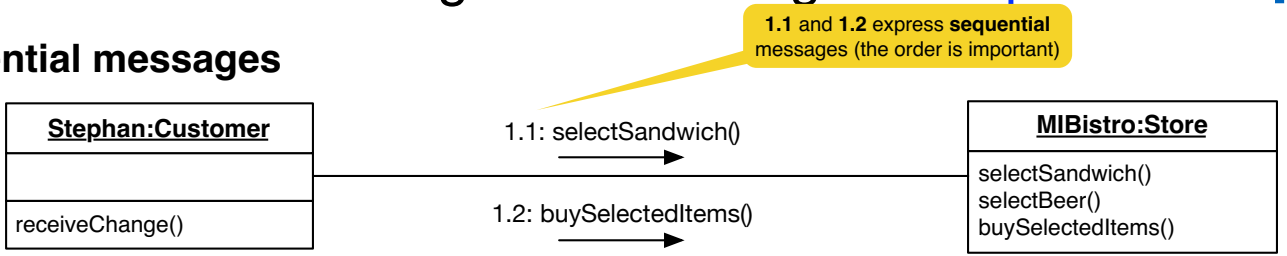
Actor



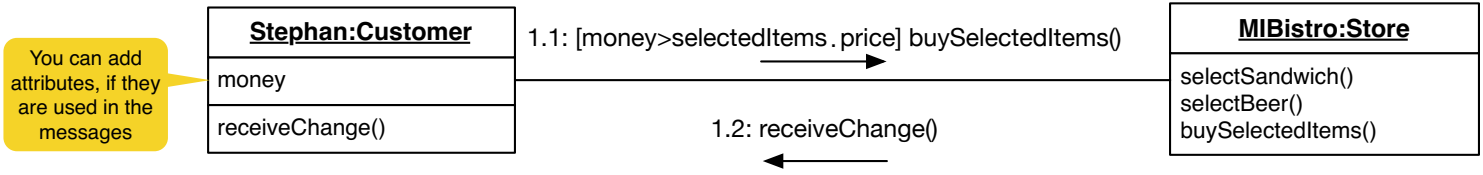
Inclusion (reuse)

UML communication diagrams: message examples

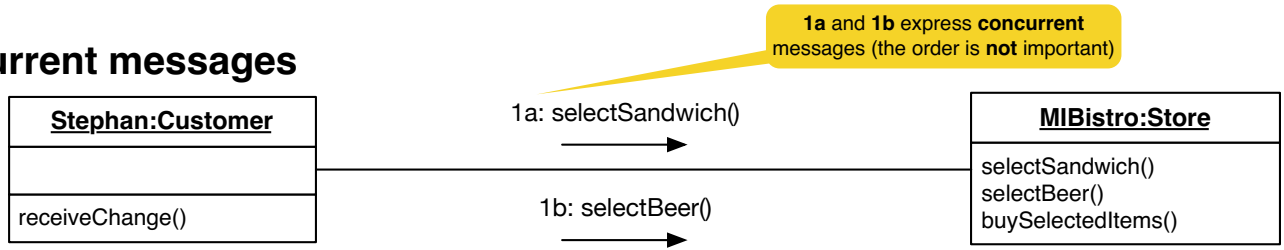
1. Sequential messages



2. Conditional messages



3. Concurrent messages



Course topics

1. Software engineering as a problem solving activity
2. Abstraction and modeling
3. Requirements analysis
4. **System design and architectural patterns**
5. Object design and design patterns
6. Testing
7. Software lifecycle modeling
8. Software configuration and release management
9. Software quality management
10. Project management

This overview shows the main influence

From analysis to system design

Nonfunctional requirements

1. Design goals
- Additional nonfunctional requirements
- Design trade-offs

Functional model

2. Subsystem decomposition
- Layers vs. partitions
- Architectural style
- Cohesion & coupling

Dynamic model

3. Concurrency
- Identification of parallelism

Object model

4. Hardware/ software mapping
- Identification of nodes
- Special purpose systems
- Buy vs. build
- Network connectivity

5. Persistent data management
- Storing persistent objects
- Filesystem vs. database

Functional model

8. Boundary conditions
- Initialization
- Termination
- Failure

Dynamic model

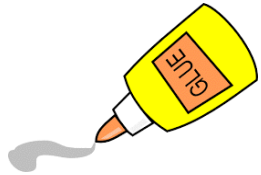
7. Software control
- Monolithic
- Event-driven
- Conc. processes

6. Global resource handling
- Access control
- ACL vs. capabilities
- Security

Coupling and cohesion of subsystems

- **Goal:** reduce system complexity while allowing change
- **Cohesion** measures dependency between classes **within** one subsystem

➡ **High cohesion**: the classes in the subsystem perform similar tasks and are related to each other via many associations



- **Low cohesion**: lots of miscellaneous and auxiliary classes, almost no associations

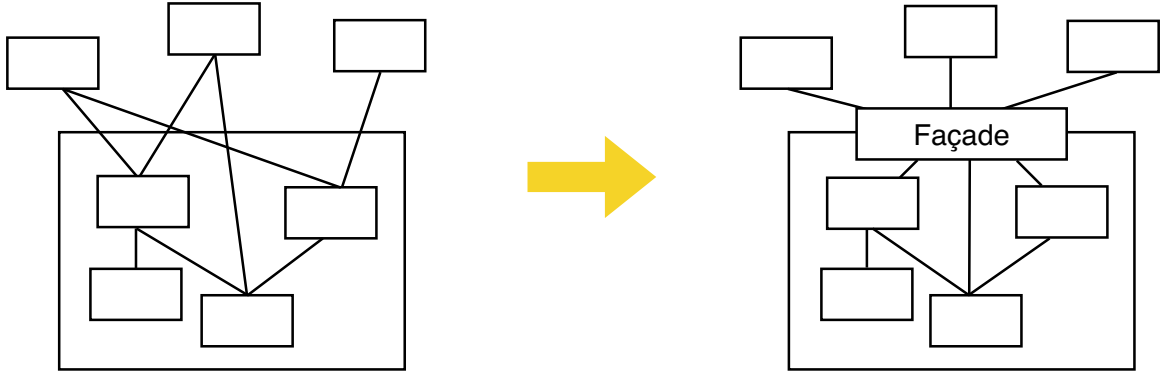
- **Coupling** measures dependency between subsystems
  - **High coupling**: changes to one subsystem will have a high impact on the other subsystem

➡ **Low coupling**: a change in one subsystem does not affect any other subsystem

Good system design

Façade design pattern: reduces coupling

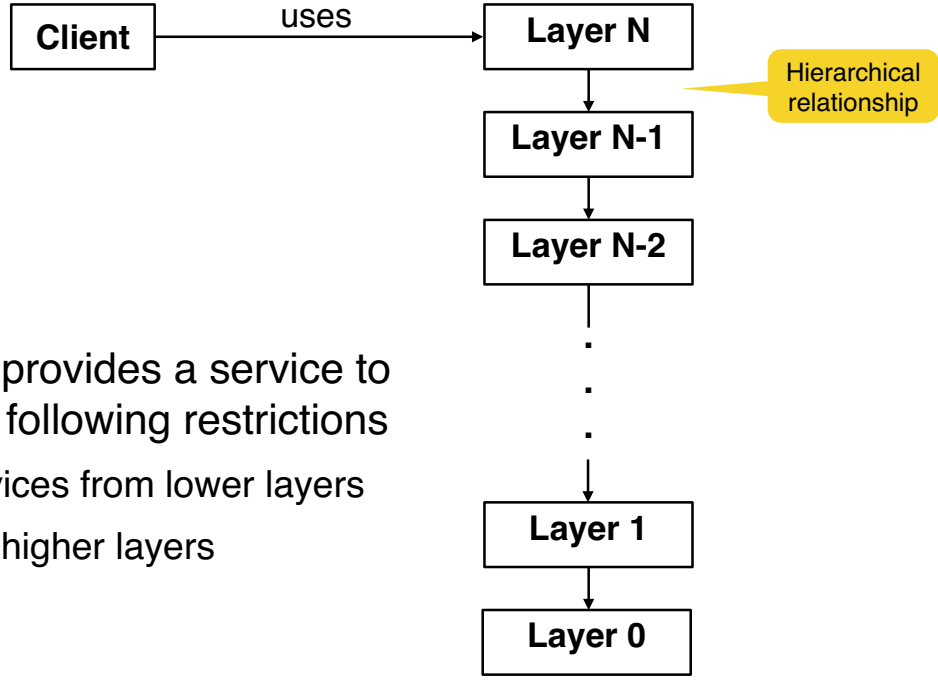
- Provides a **unified interface** for a subsystem
  - Consists of a set of public operations
  - Each public operation is delegated to one or more operations in the classes behind the façade
- Defines a higher-level interface that makes the subsystem easier to use (i.e. it abstracts out the gory details)
- Allows to hide design spaghetti from the caller



Architectural style vs. architecture

- **Subsystem decomposition:** identification of subsystems, services, and their relationships to each other
- **Architectural style:** a pattern for a subsystem decomposition
- **Software architecture:** instance of an architectural style

Layered architectural style



- A **layer** is a subsystem that provides a service to another subsystem with the following restrictions
  - A layer only depends on services from lower layers
  - A layer has no knowledge of higher layers

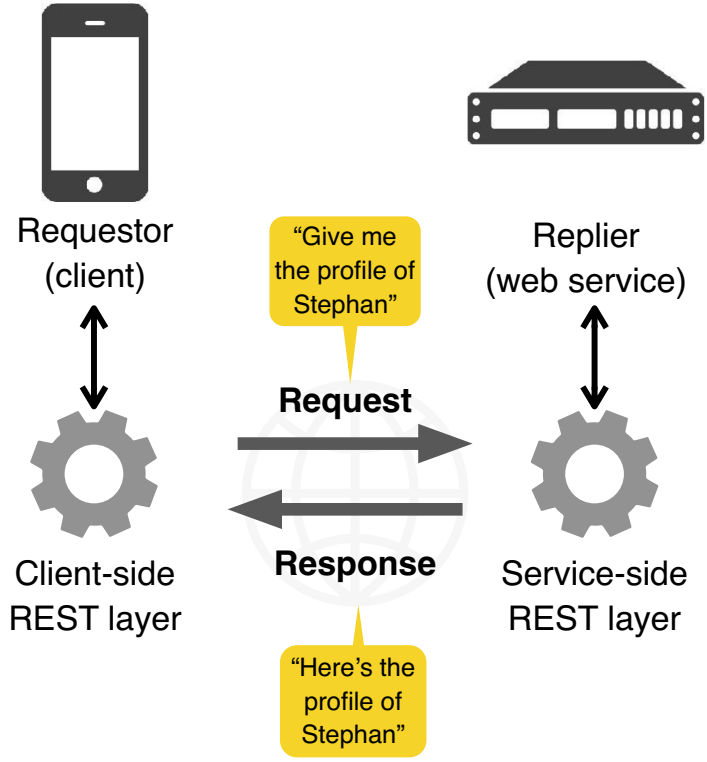
Client server architectural style vs P2P

- One or more servers provide services to clients
- Each client calls a service offered by the server
  - **Server** performs service and returns result to client
  - **Client** knows interface of the server
  - **Server** does not know the interface of the client
- Response is typically immediate (i.e. less than a few seconds)
- End users interact only with the client

REST architectural style

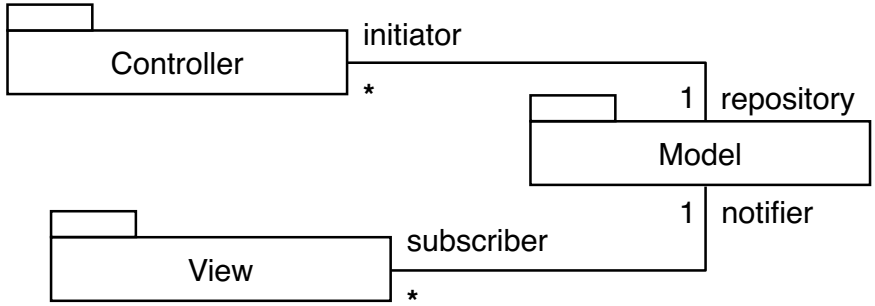
1. **Requestor** sends a message to **replier**
2. **Replier** receives and processes the **request**
3. **Replier** returns a message in **response**

- Stateless: the replier does not keep a history of old requests
- Request and replier use multiple layers to handle requests and responses



Model view controller (MVC) architectural style

- **Model:** process and store application domain data (entity objects)
- **View:** display information to the user (boundary objects)
- **Controller:** interact with the user and update the model (which notifies the view)
  - **Important:** view and controller together comprise the user interface
  - A change propagation mechanism ensures consistency between the user interface(s) and the model



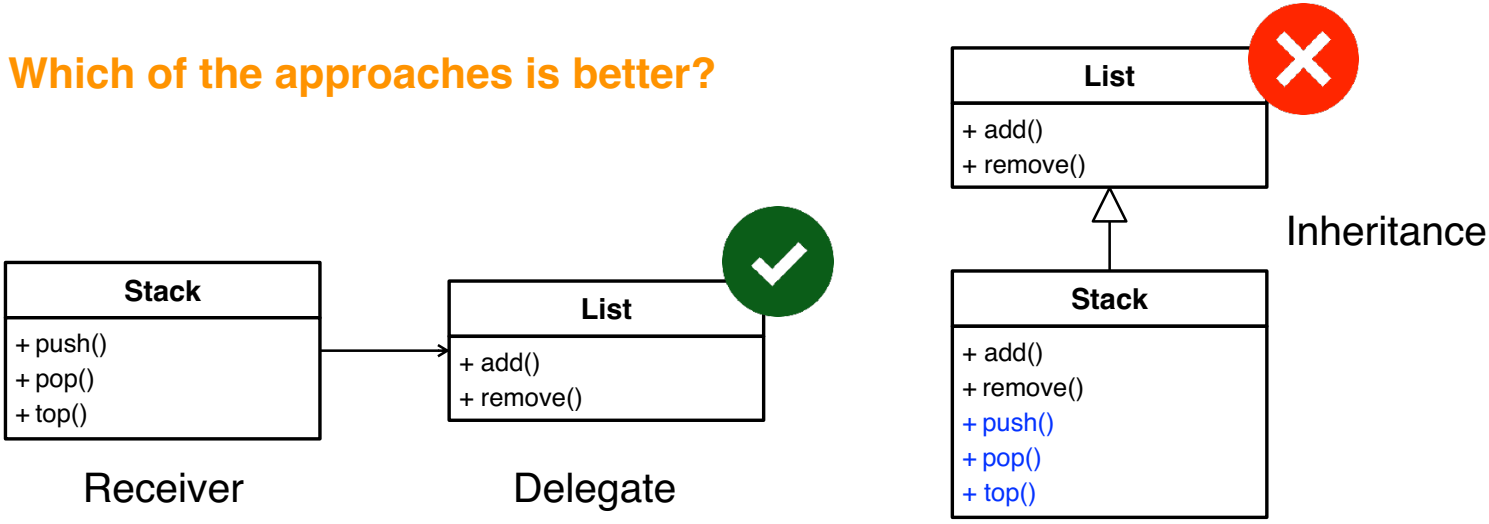
Course topics

1. Software engineering as a problem solving activity
2. Abstraction and modeling
3. Requirements analysis
4. System design and architectural patterns
5. **Object design and design patterns**
6. Testing
7. Software lifecycle modeling
8. Software configuration and release management
9. Software quality management
10. Project management

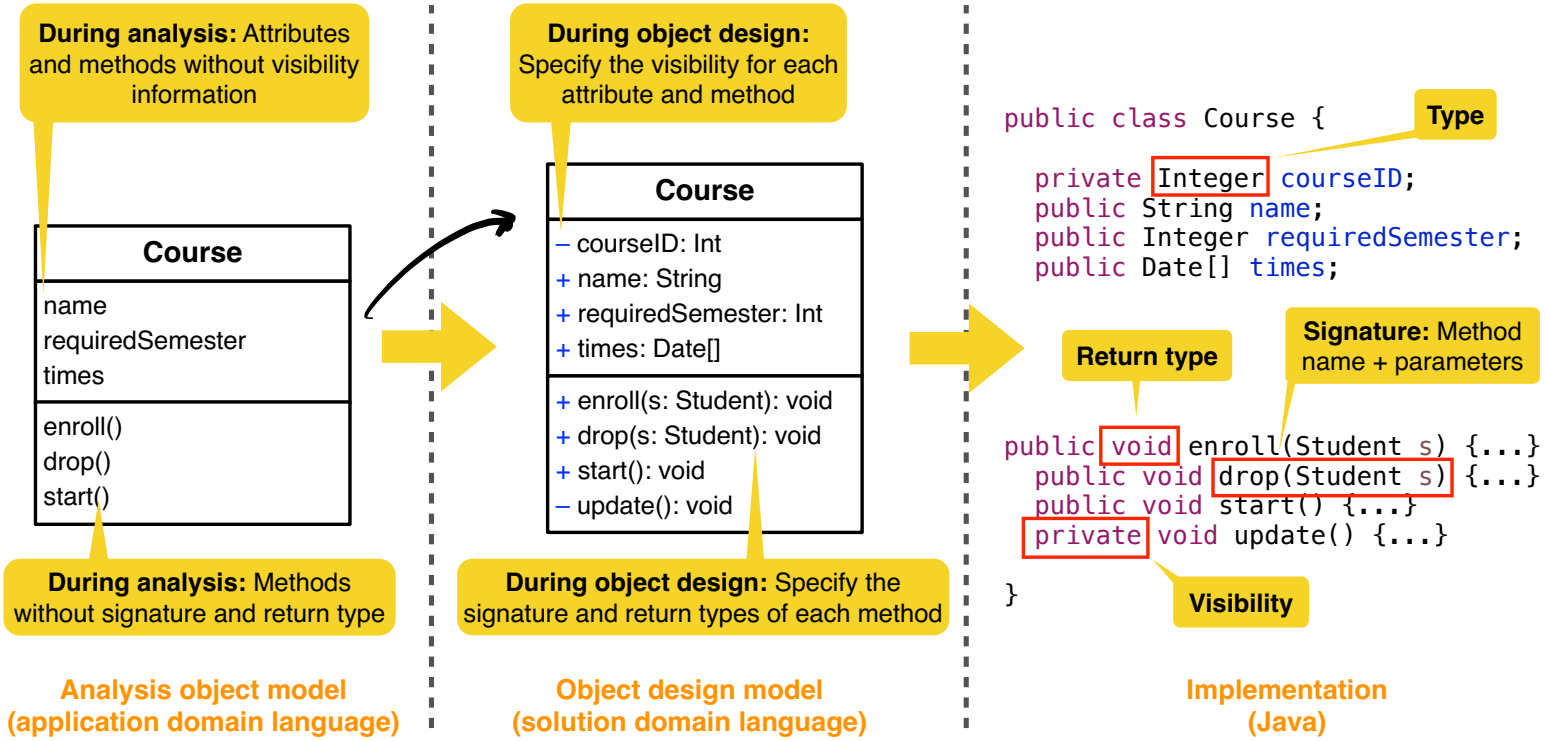
Reuse: implementation inheritance vs. delegation

- **Implementation inheritance:** extending a base class by a new operation or overriding an existing operation
- **Delegation:** catching an operation and sending it to another object

- **Which of the approaches is better?**

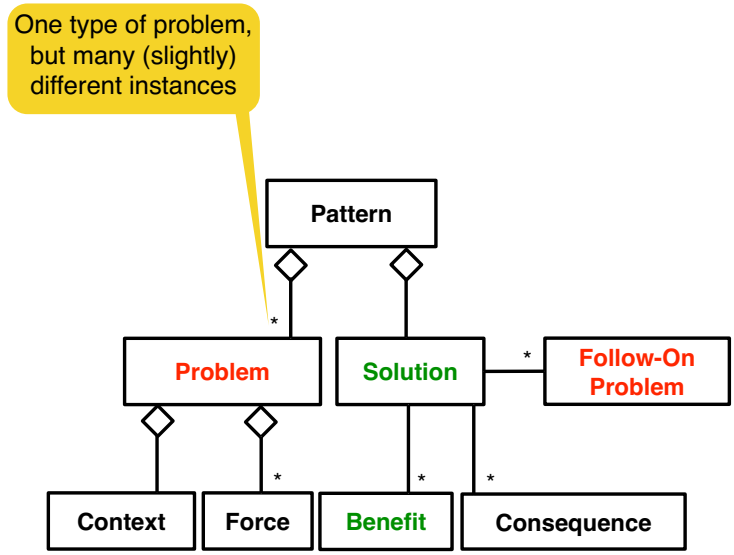


Distinction between **analysis** object model and object **design** model

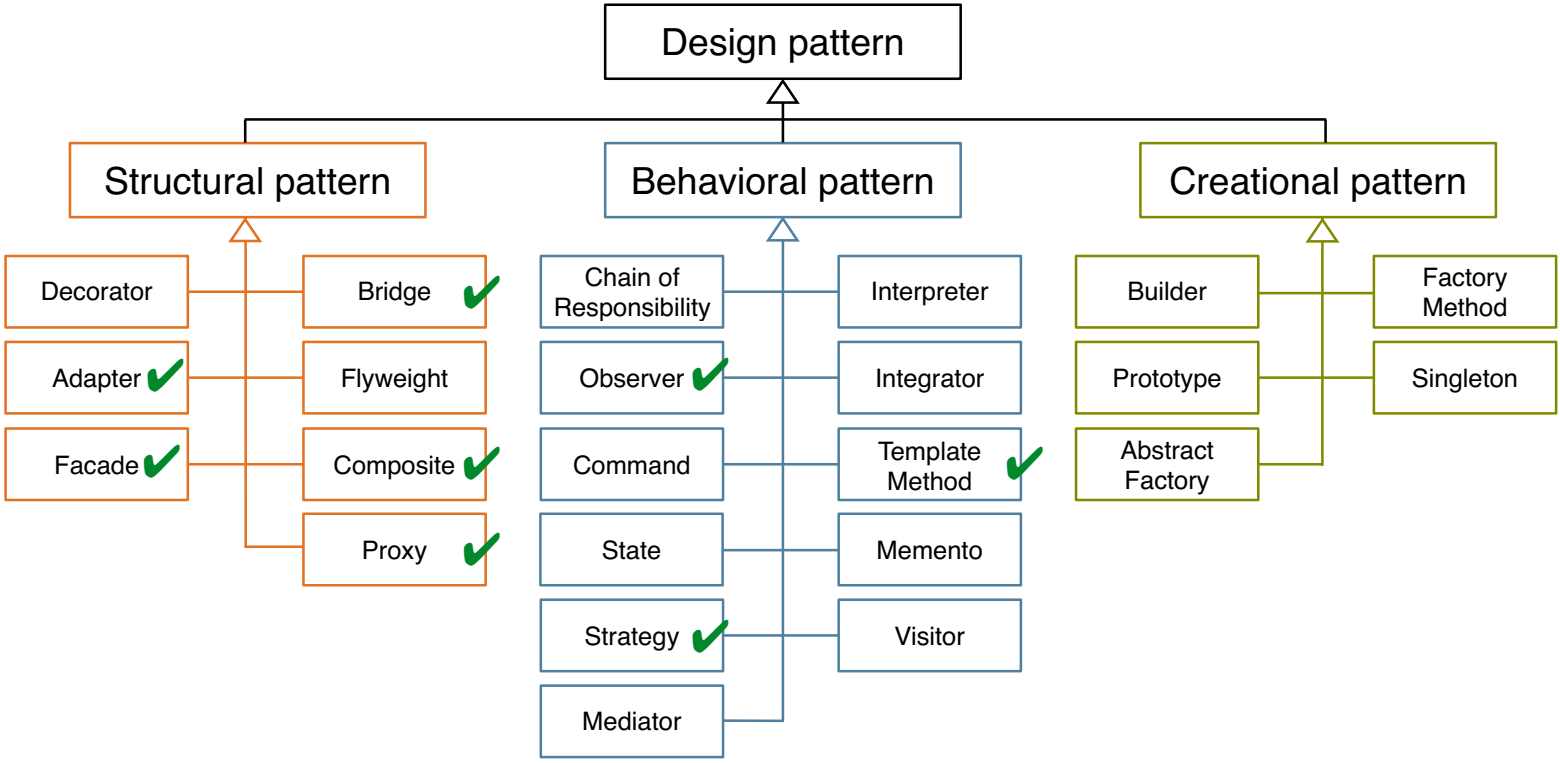


Modeling a pattern in UML

- The **Problem** explains the actual situation in form of context and forces
  - The **Context** sets the stage where the pattern takes place
  - **Forces** describe why the problem is difficult to solve
- The **Solution** resolves these forces with benefits and consequences
  - **Benefits** describe positive outcomes of the solution
  - **Consequences** explain effects, results, and other outcomes of the application of the pattern
- **Follow-On Problems** can occur when you apply the solution

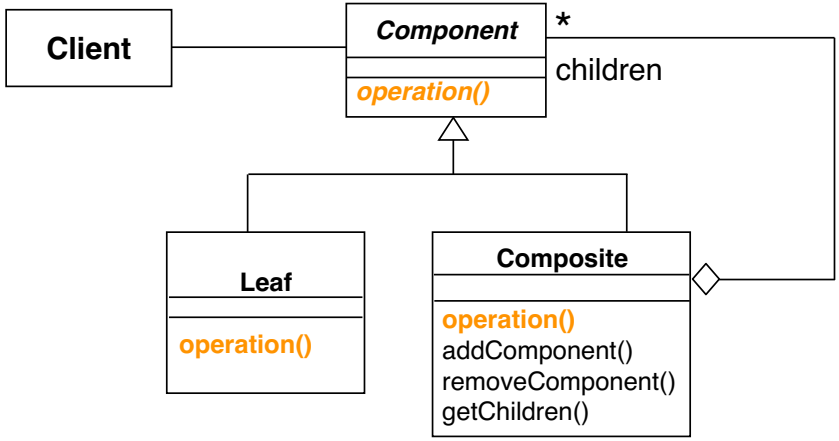


Design pattern taxonomy

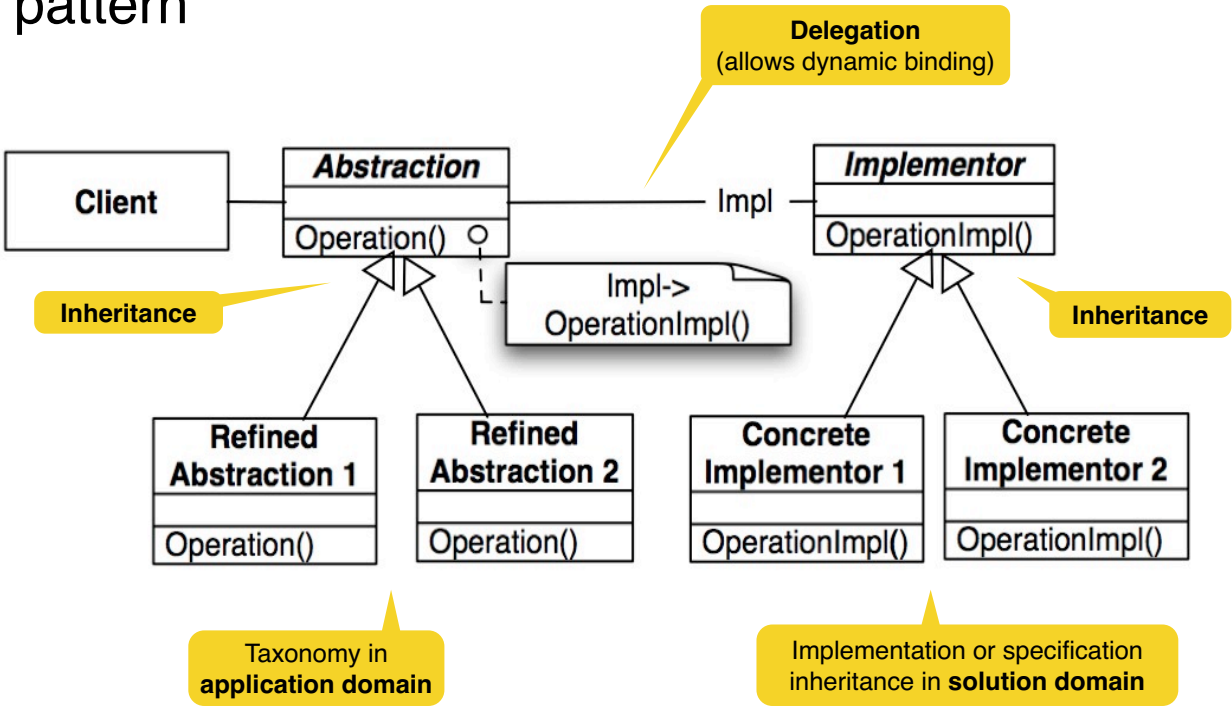


Composite pattern

- **Problem:** there are hierarchies with arbitrary depth and width (e.g. folders and files)
- **Solution:** the composite pattern lets a **`Client`** treat an individual class called **`Leaf`** and **`Compositions`** of **`Leaf`** classes uniformly

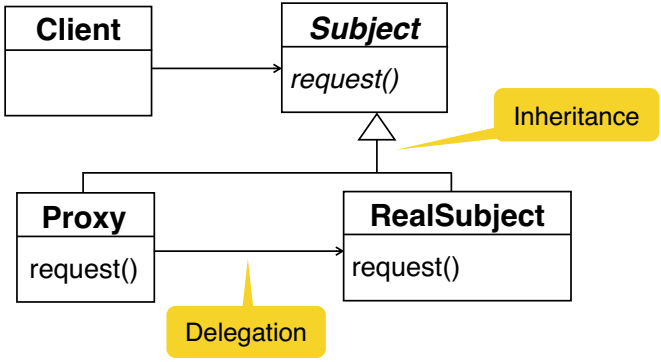


Bridge pattern

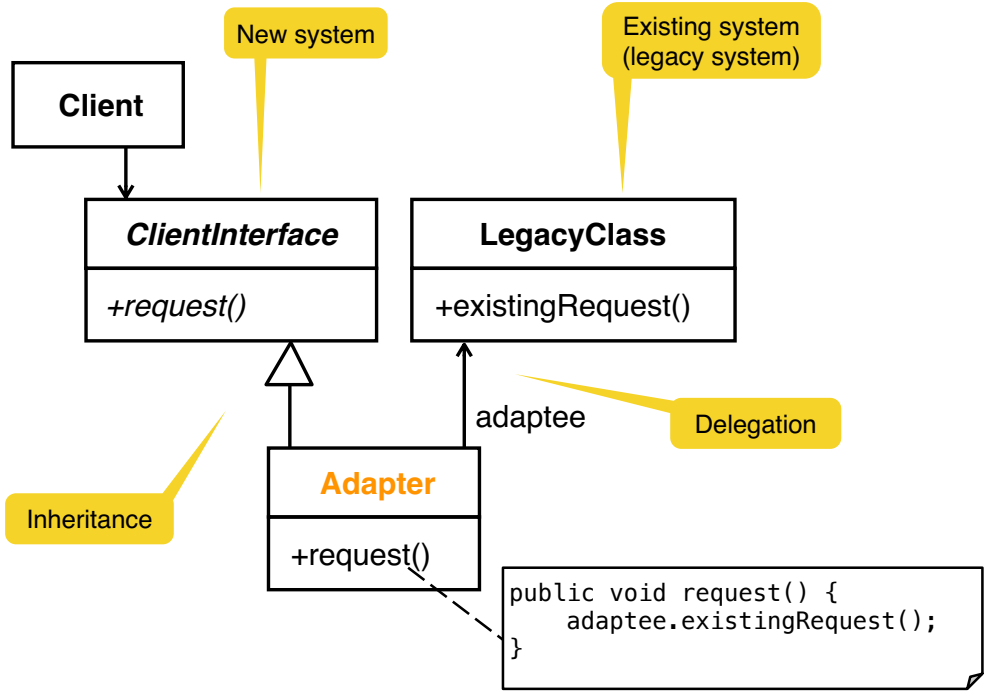


Proxy pattern

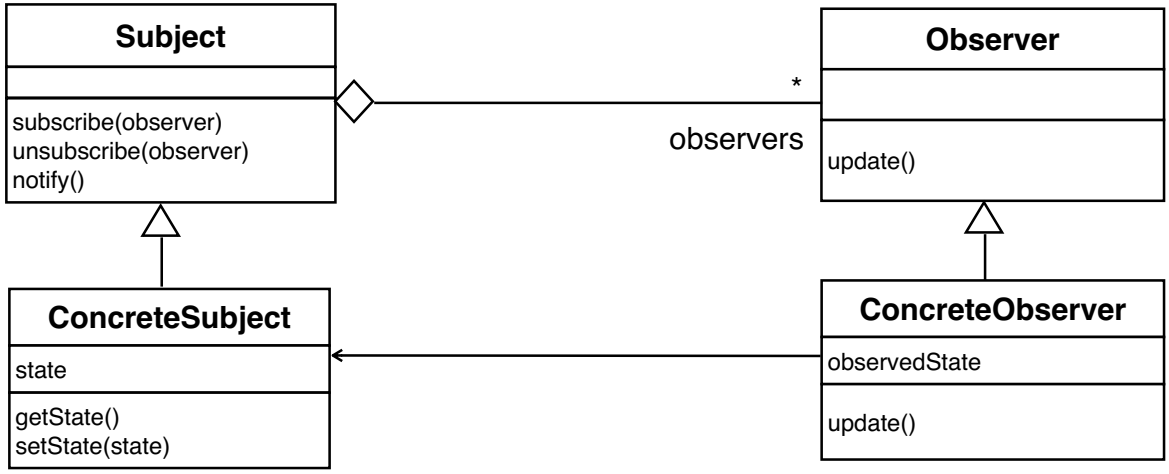
- **`Proxy`** and **`RealSubject`** are subclasses of the *abstract* class **`Subject`**
- The **`Client`** always calls **`request()`** in an instance of type **`Proxy`**
- The implementation of **`request()`** in **`Proxy`** then uses delegation to access **`request()`** in **`RealSubject`**



Adapter pattern

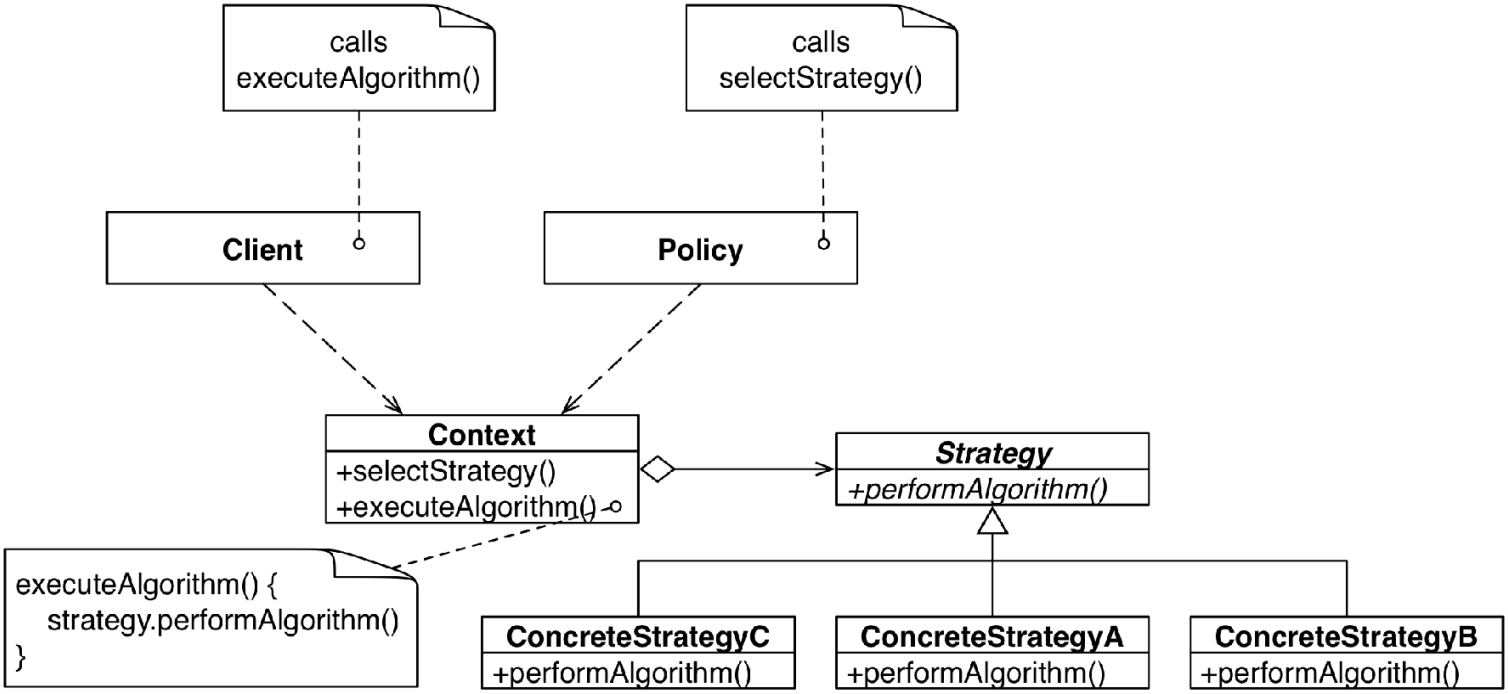


Observer pattern



- The **`Subject`** represents the entity object
  - The state is contained in the subclass **`ConcreteSubject`**
- **`Observers`** attach to the **`Subject`** by calling **`subscribe()`**
- Each **`ConcreteObserver`** has a different view of the **`state`** of the **`ConcreteSubject`**
  - The state can be obtained and set by the subclasses of type **`ConcreteObserver`**

Strategy pattern



L12E02 Observer Pattern Not started yet.

Start exercise **Medium** Due date: end of today

20 min

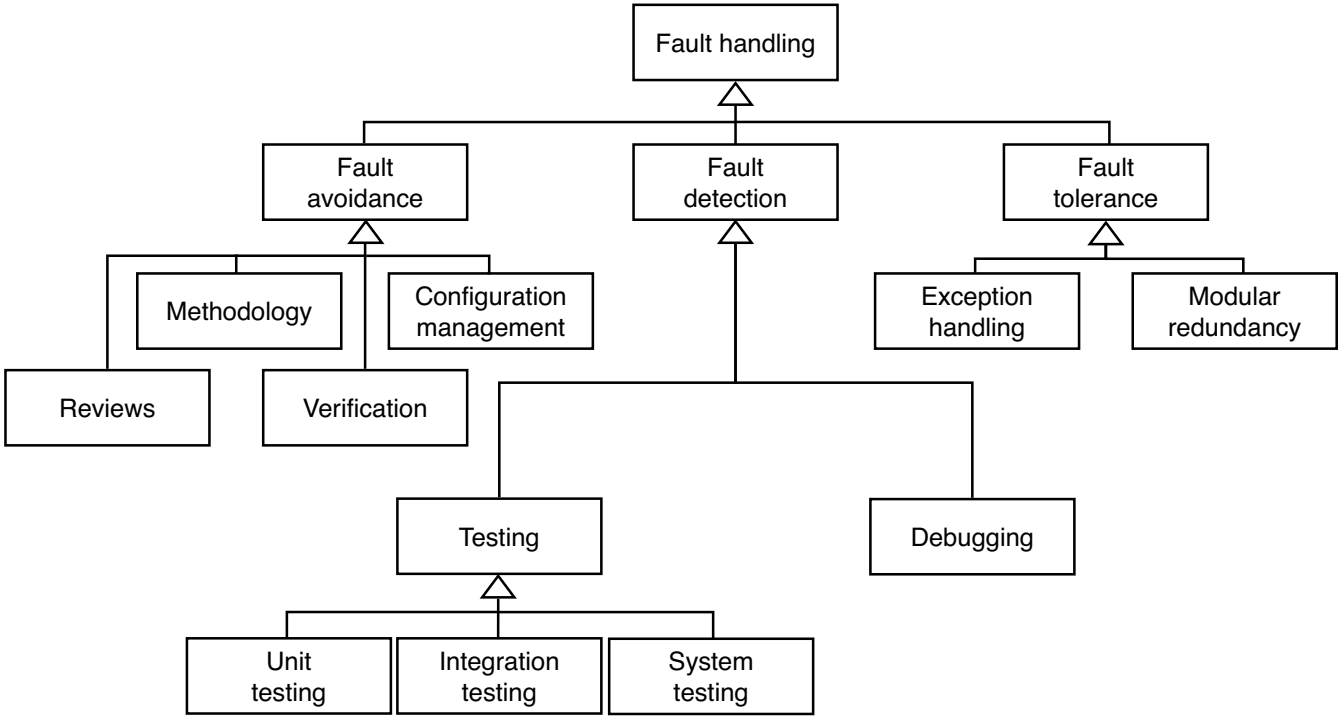
10 pts

- **Problem statement**
  - The city of Munich wants to contribute to a sustainable environment by offering innovative, safe, and user-friendly means of transport: the **PEVolve** system
  - **PEV:** E-Moped, E-Bikes (electronic bicycle/pedelec), and E-Kickscooter
  - Riders want to see in real time, which PEVs are available for rent
  - **Your task:** use the **notification + push** variant of the **observer pattern** to observe available and rented PEVs

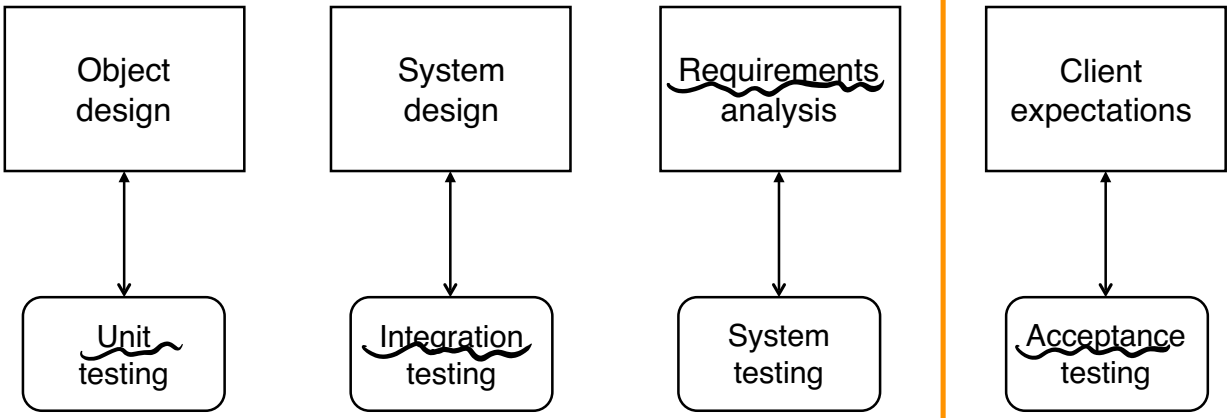
Course topics

1. Software engineering as a problem solving activity
2. Abstraction and modeling
3. Requirements analysis
4. System design and architectural patterns
5. Object design and design patterns
6. **Testing**
7. Software lifecycle modeling
8. Software configuration and release management
9. Software quality management
10. Project management

Taxonomy for fault handling techniques



Testing activities



Internal (e.g. testers in the team)

External (e.g. client)

JUnit

- An open source Java framework (test system) for writing and executing unit tests
- The unit test for the class **`Money`** should test the **`add()`** method
- Below is an example test for the addition of money

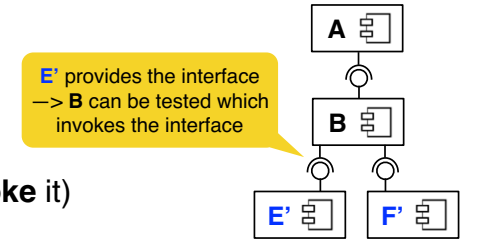
```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class MoneyTest {
    @Test
    void testSimpleAdd() {
        Money m12CHF = new Money(12, Currency.CHF);
        Money m14CHF = new Money(14, Currency.CHF);
        Money expected = new Money(26, Currency.CHF);
        Money observed = m12CHF.add(m14CHF);
        assertEquals(expected, observed);
    }
}
```

The test passes, if both parameters are equal, otherwise the test throws an exception of type **AssertionError**

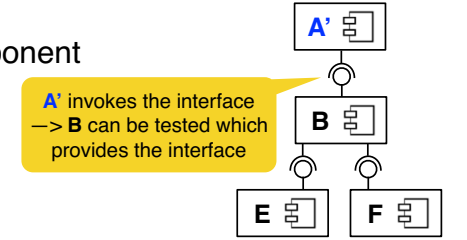
Drivers and stubs

- Both are doubles that replace the actual component (subsystem or class) during testing
- **Stub:**
  - **Provides** the same interface as the actual (replaced) component
  - Each operation is implemented very simply (e.g. always returns the same value)
  - Allows to test other components (which **require** the interface and **invoke** it)
  - Used in top-down integration
  - **Example: E’** and **F’**



Use of stubs (**E’** and **F’**) when top-down testing B

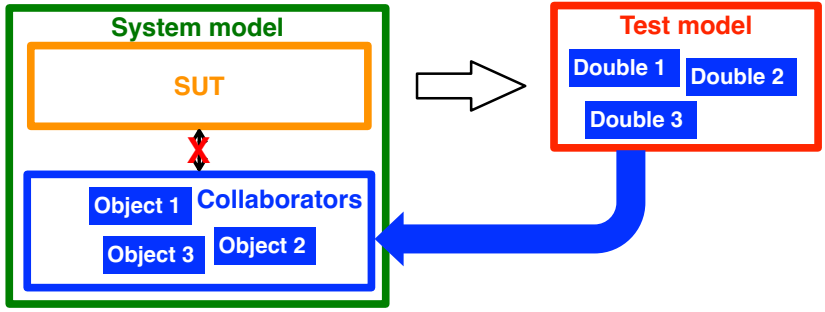
- **Driver:**
  - **Invokes** and **requires** the same interface as the actual (replaced) component
  - Each operation of the interface is invoked for testing purposes
  - Allows to test other components (which **provide** the interface)
  - Used in bottom-up integration
  - **Example: A’**



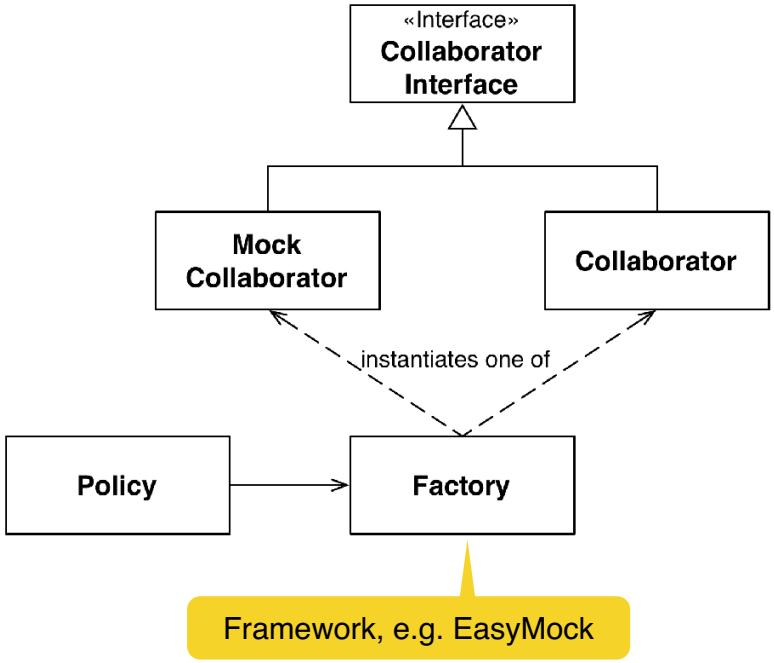
Use of a driver **A’** when bottom-up testing B

Object oriented test modeling

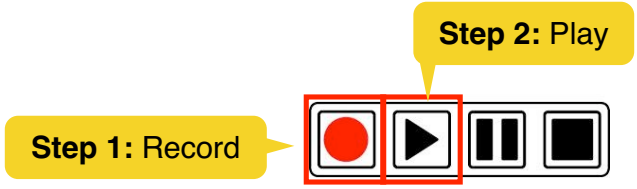
- Start with the **system model**
- The system contains the **SUT** (system under test)
- The **SUT** does not exist in isolation, it interacts with other participating objects in the system model that are not yet implemented: **collaborators**
- The **test model** is derived from the **SUT**
- To be able to interact with **collaborators**, we add objects to the **test model**
- These are called **test doubles** (substitutes for the **collaborators** during testing)



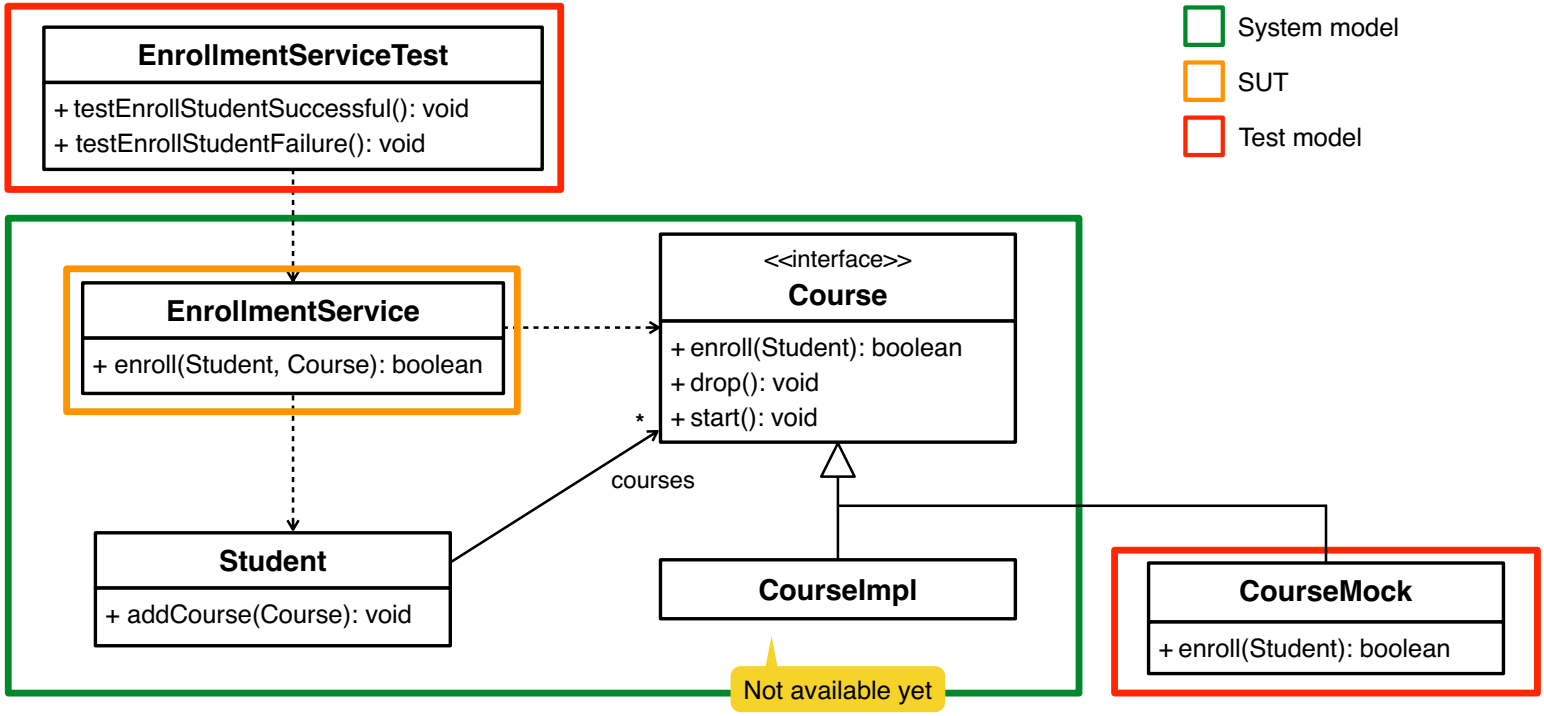
Mock object pattern



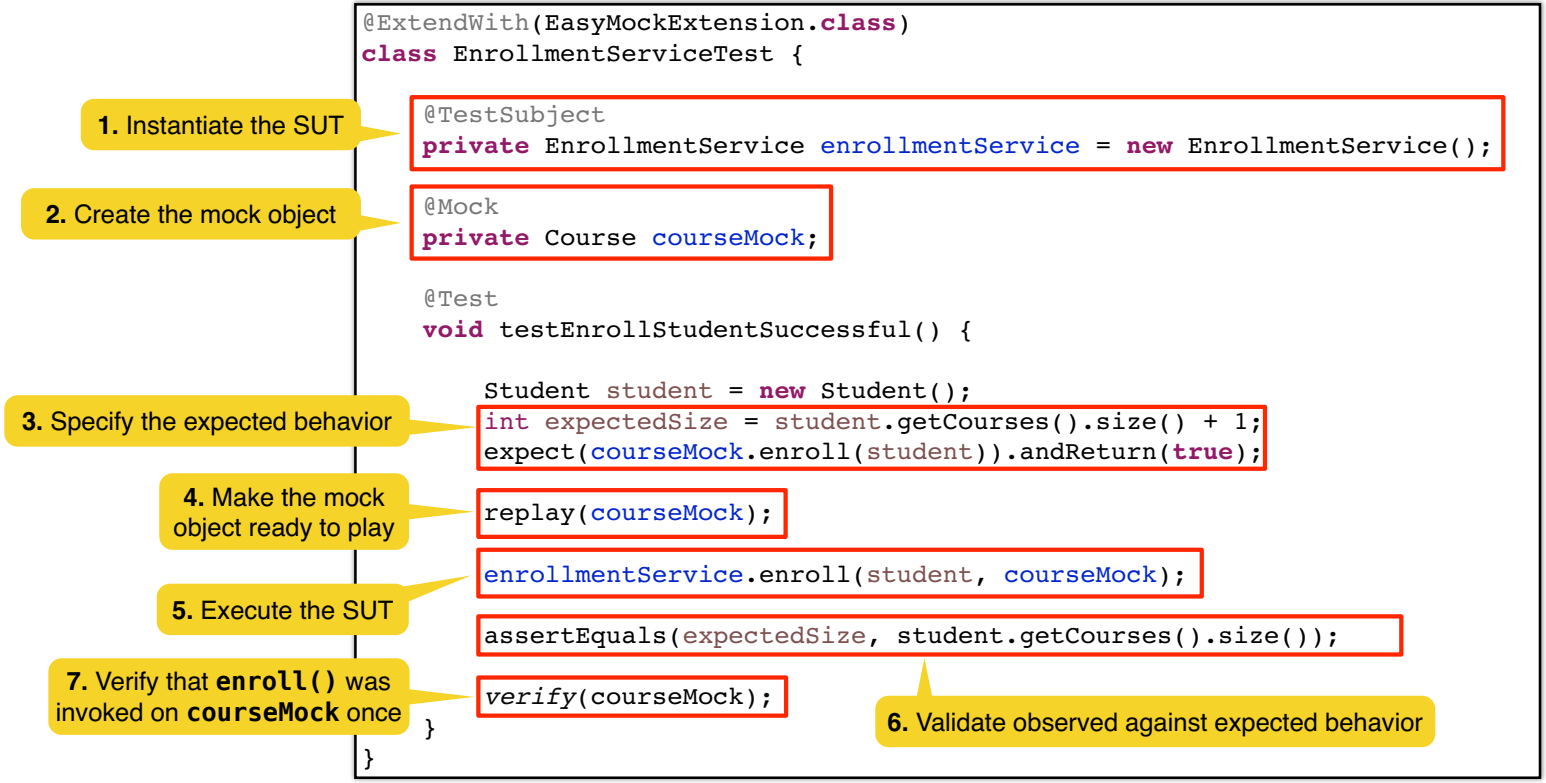
- A **mock object** replaces the behavior of a real object called the collaborator and returns hard-coded values
- A mock object can be created at startup time with the factory pattern (not covered in the lecture, look it up in Gamma's book)
- Mock objects can be used for testing the state of individual objects as well as the interaction between objects
- The use of mock objects is based on the **record play metaphor**



Example: university app with a mock object



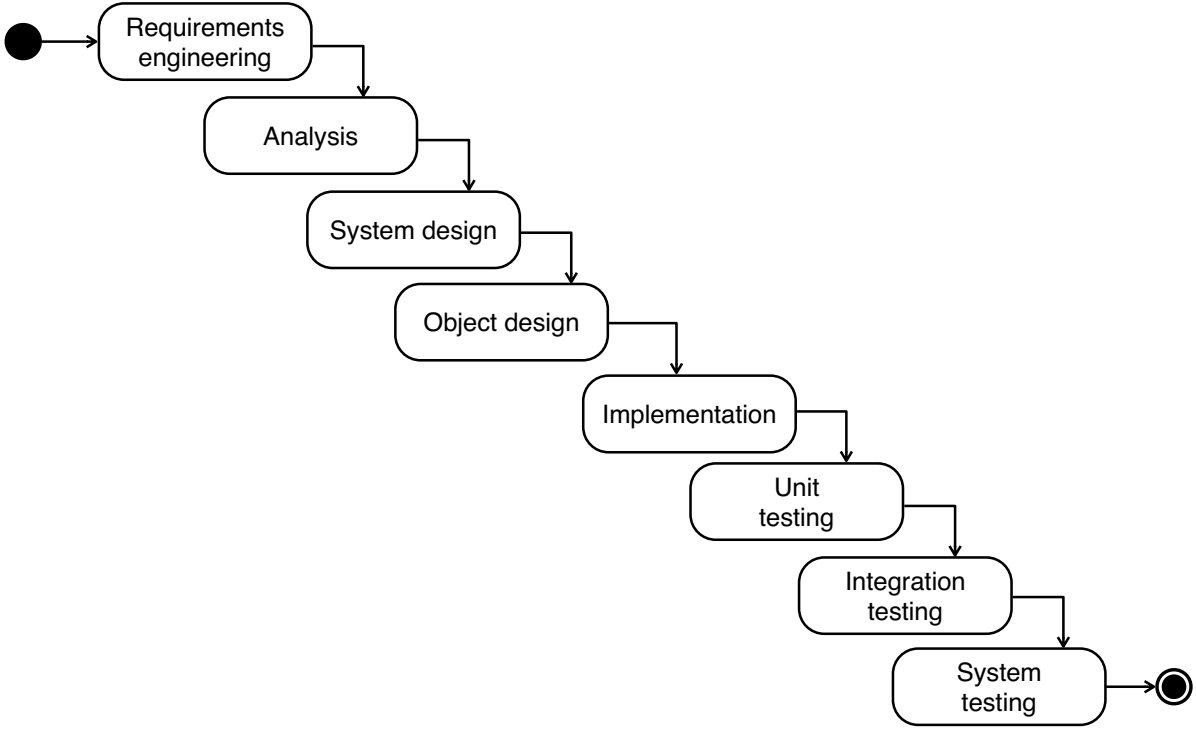
Unit test for enrolling students with EasyMock



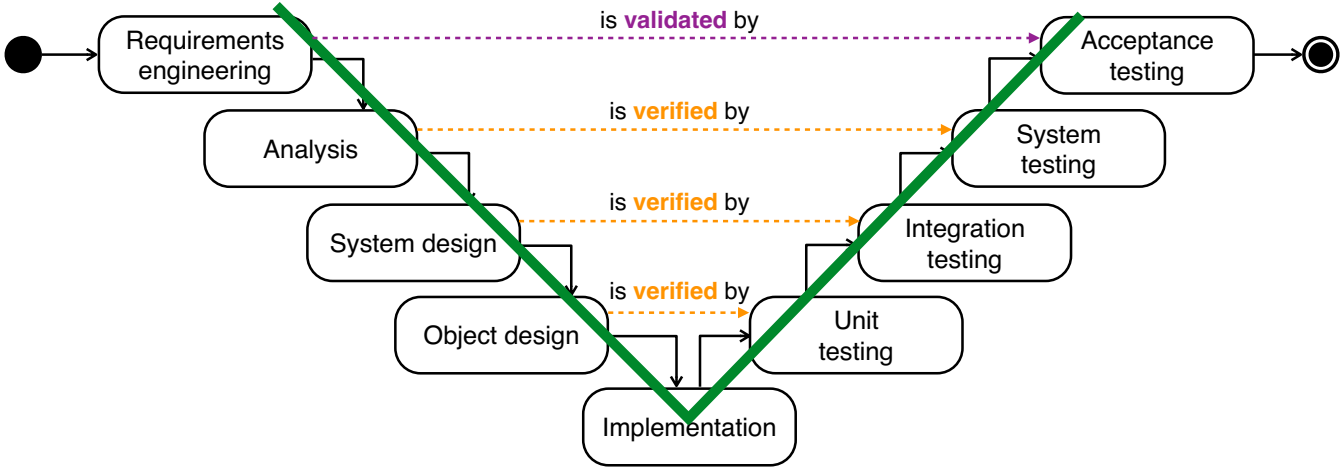
Course topics

1. Software engineering as a problem solving activity
2. Abstraction and modeling
3. Requirements analysis
4. System design and architectural patterns
5. Object design and design patterns
6. Testing
7. **Software lifecycle modeling**
8. Software configuration and release management
9. Software quality management
10. Project management

Waterfall model

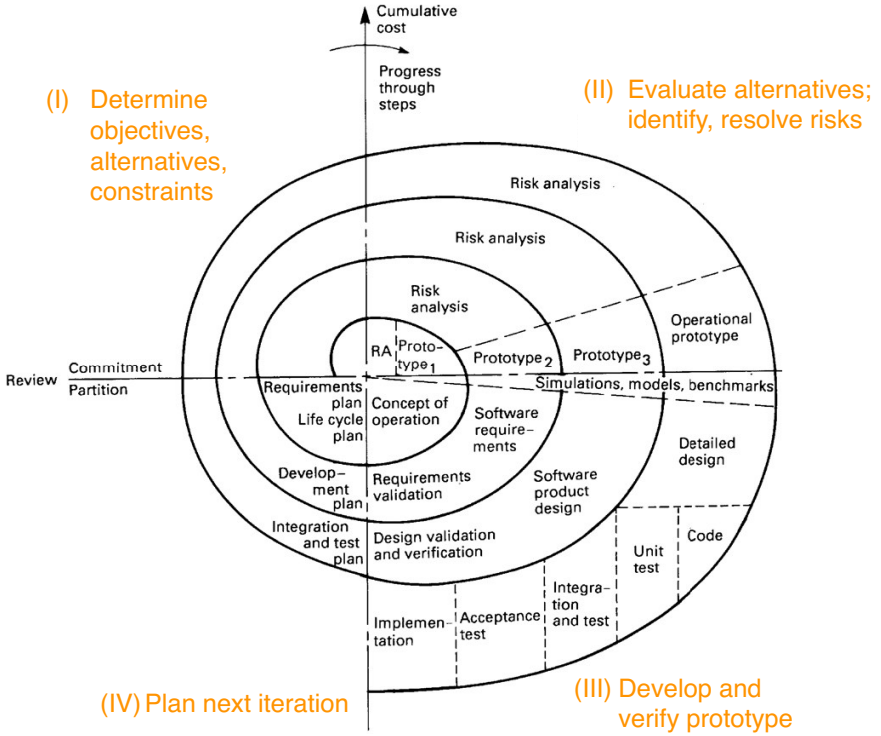


V-Model

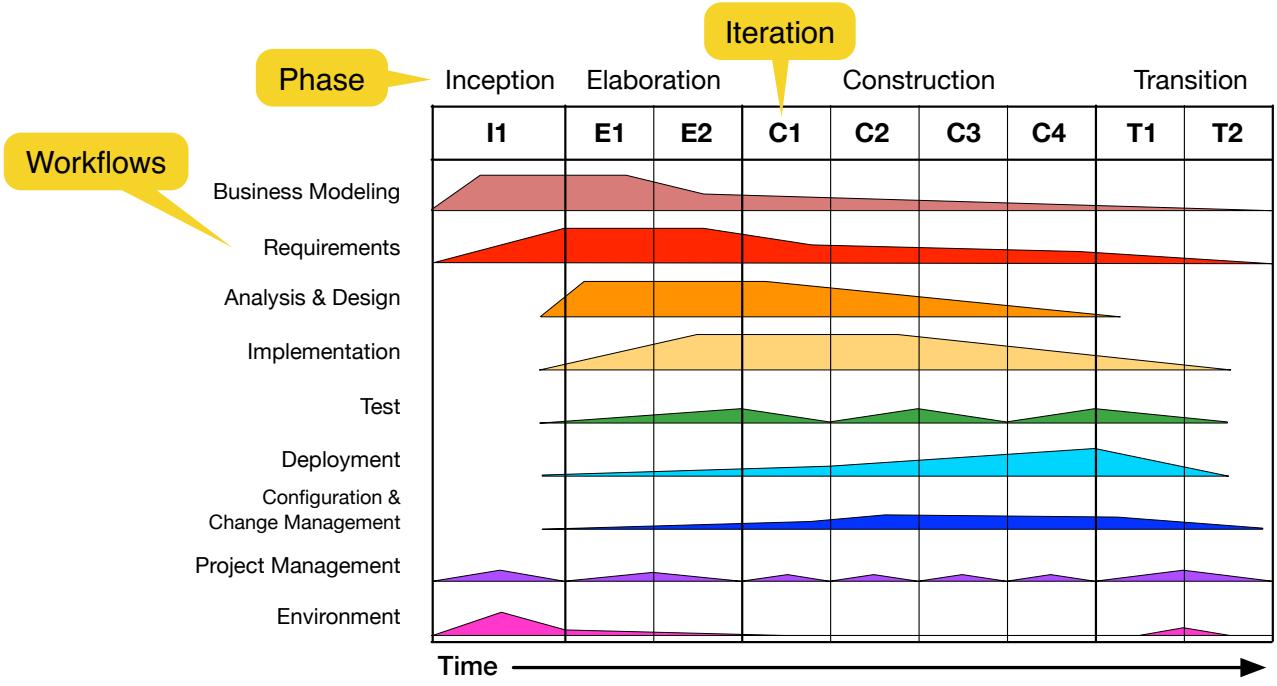


- **Validation:** Assurance that a product, a service or a system meet the needs of the customer and other identified stakeholders (often involves acceptance and suitability with **external** customers)
- **Verification:** Evaluation whether or not a product, a service, or a system comply with a regulation, requirement, specification, or imposed condition (often an **internal** process)

Spiral model



Unified process



Defined vs. empirical process

Defined process

Planned

Follows strict rules

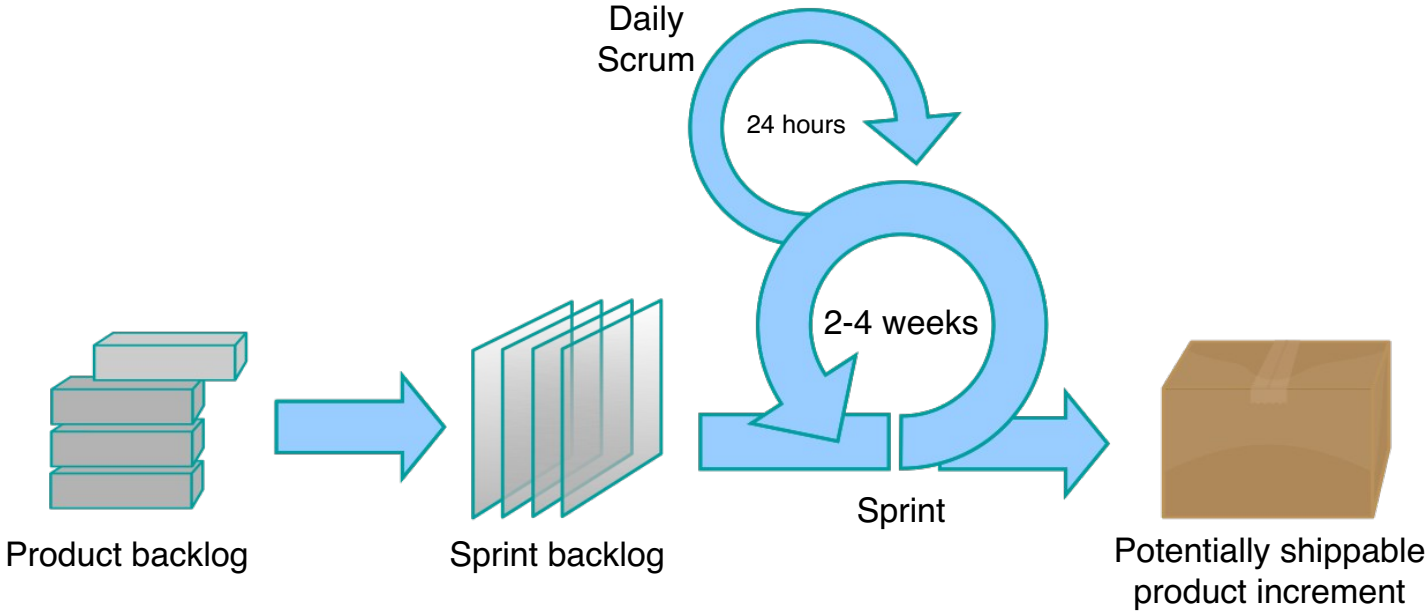
Avoids deviations

Empirical process

Not entirely planned

Inspect and adapt

Scrum


Daily
Scrum
24 hours
2-4 weeks
Sprint
Product backlog
Sprint backlog
Potentially shippable
product increment

Kanban

- An agile model for software development

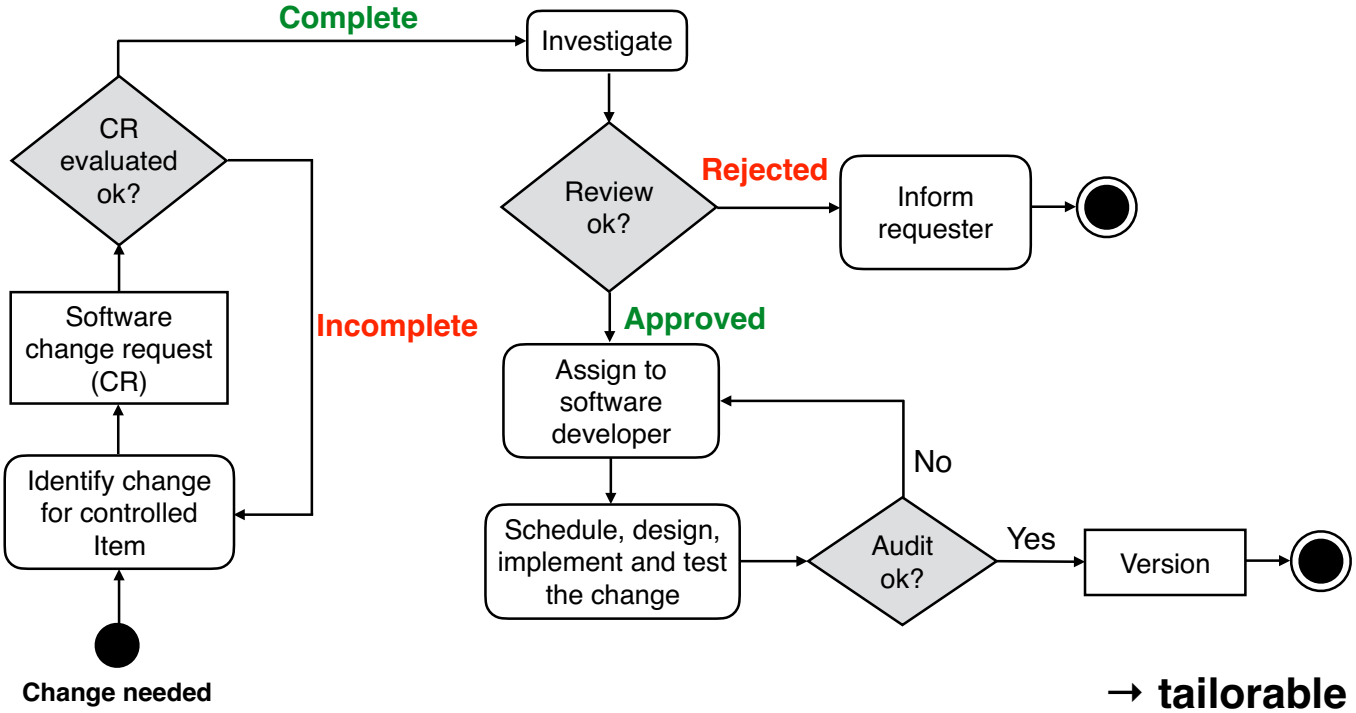
Four basic principles

1) Start with **existing** process
2) Agree to pursue **incremental**, **evolutionary** change
3) **Respect** the current process, roles, responsibilities and titles
4) **Leadership** at all levels

Course topics

1. Software engineering as a problem solving activity
2. Abstraction and modeling
3. Requirements analysis
4. System design and architectural patterns
5. Object design and design patterns
6. Testing
7. Software lifecycle modeling
8. **Software configuration and release management**
9. Software quality management
10. Project management

Example of a change control process (UML activity diagram)



→ tailorable process

Most important git commands (activities)

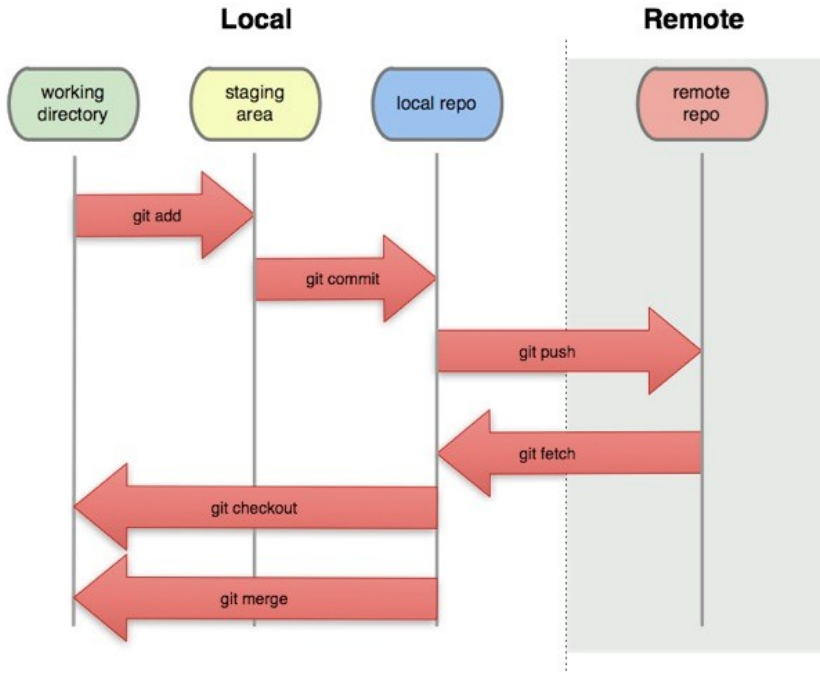
git add:
Add changed files to the staging area

git commit:
Commit selected changed files of the staging area to your local repository

git push:
Upload local commits to a remote repository

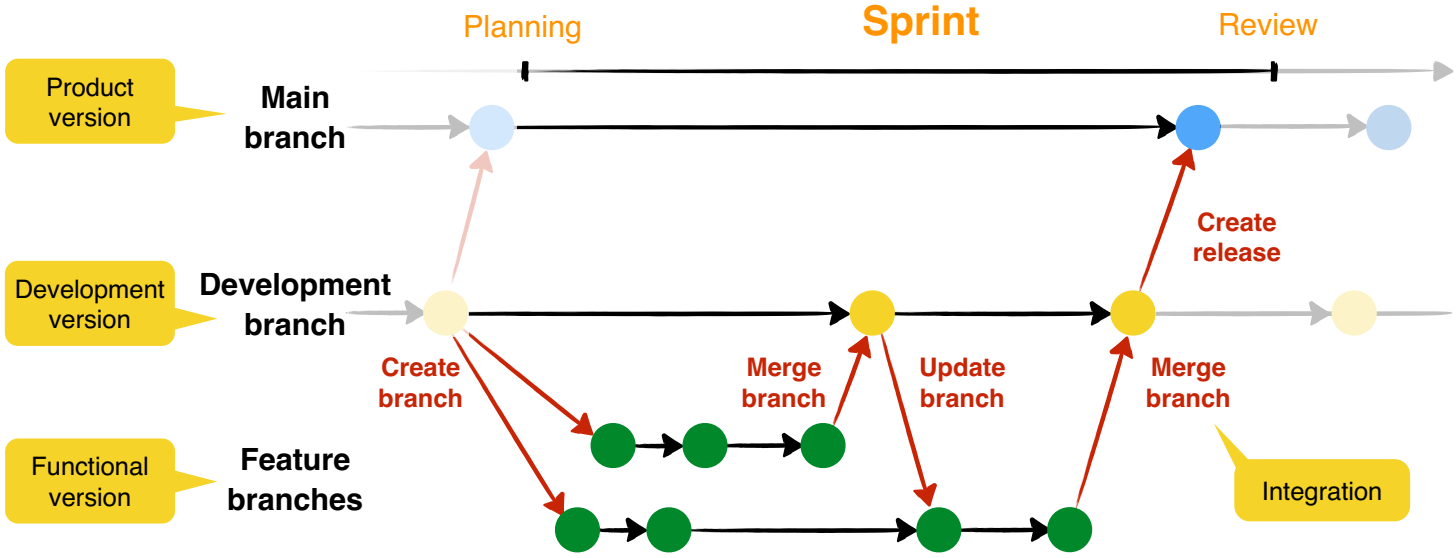
git pull (fetch & merge):
Download and merge remote commits into your working copy

git clone (fetch & initial checkout):
Clone a complete repository into a new working directory

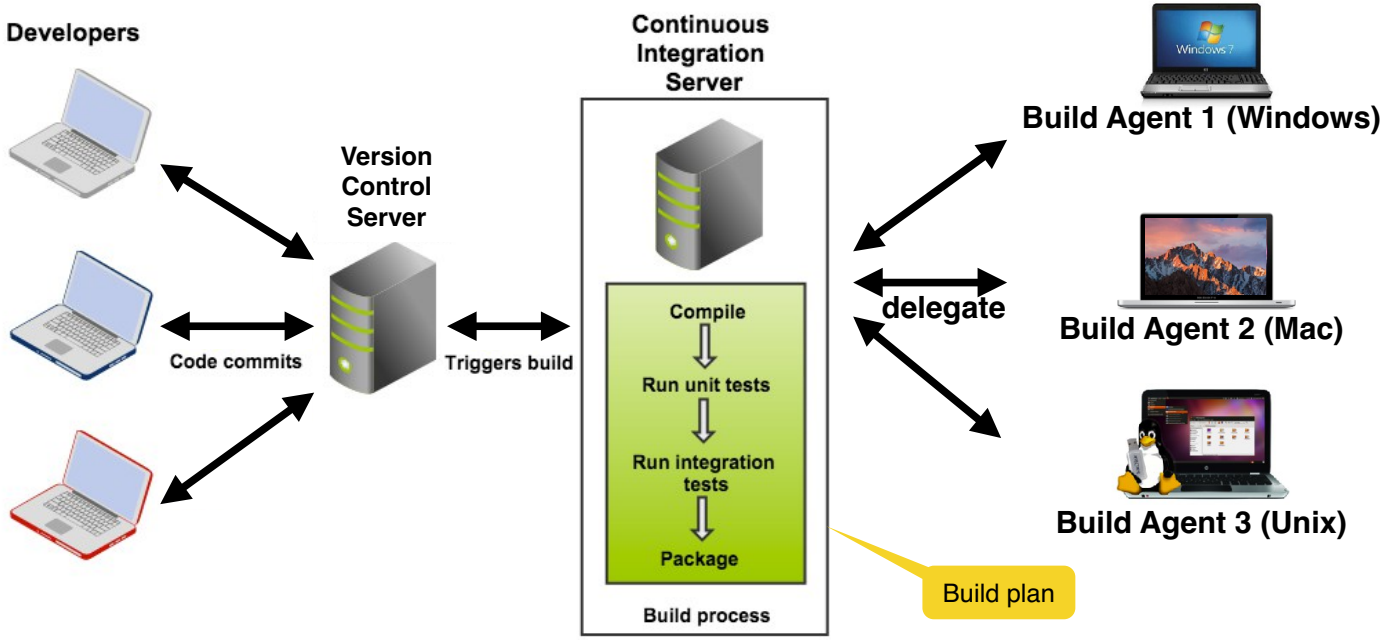


Example of a git branch management model (simplified)

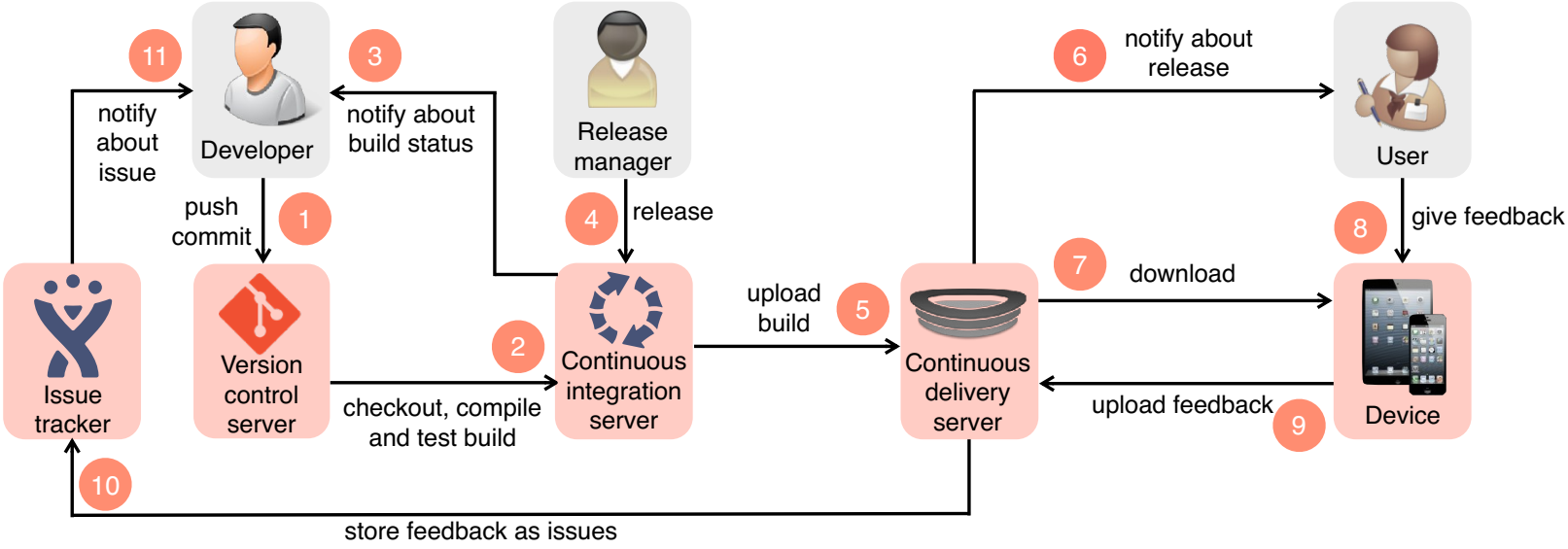
- **Main branch:** store external releases (e.g. Scrum product increment)
- **Development branch:** store internal promotion (release candidates)
- **Feature branches:** store incremental development and explorations



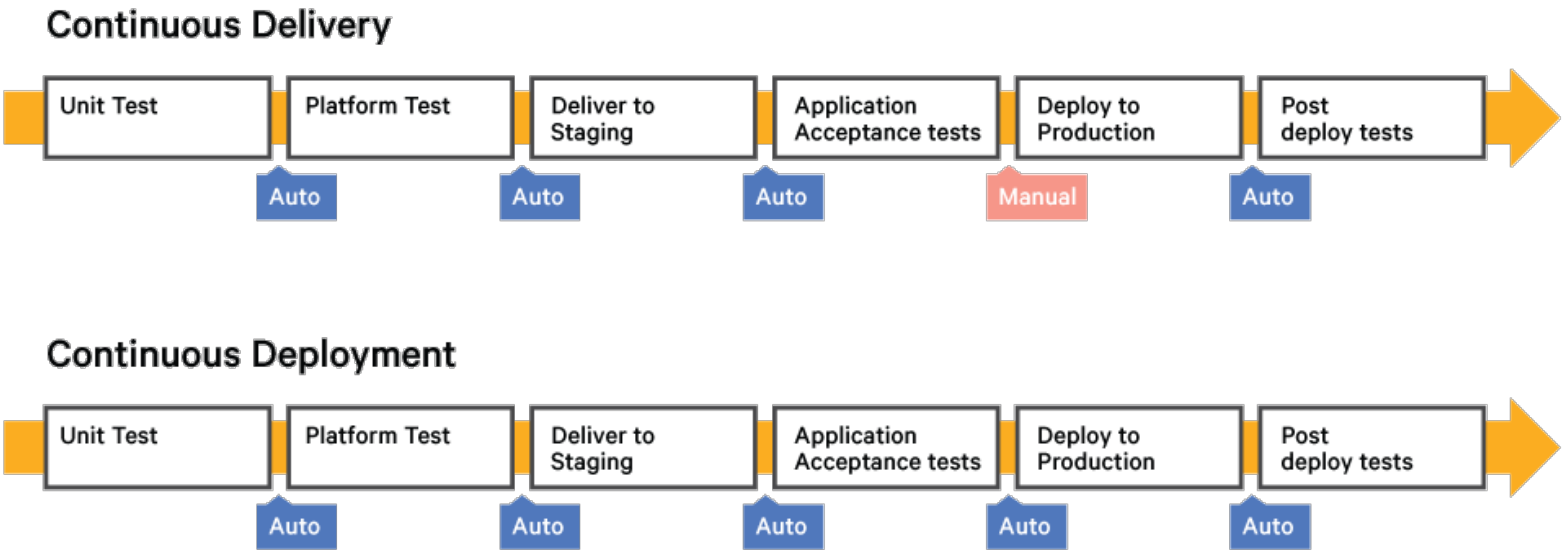
Continuous integration overview



Example of a build and release management workflow with feedback

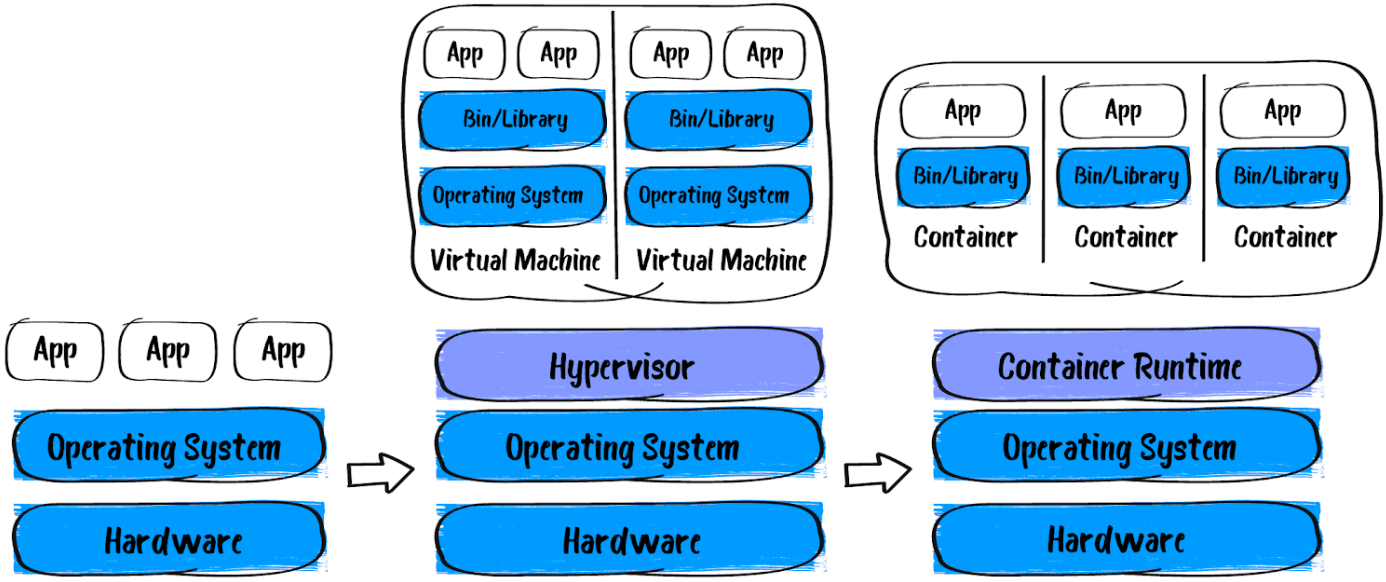


Continuous delivery vs. continuous deployment



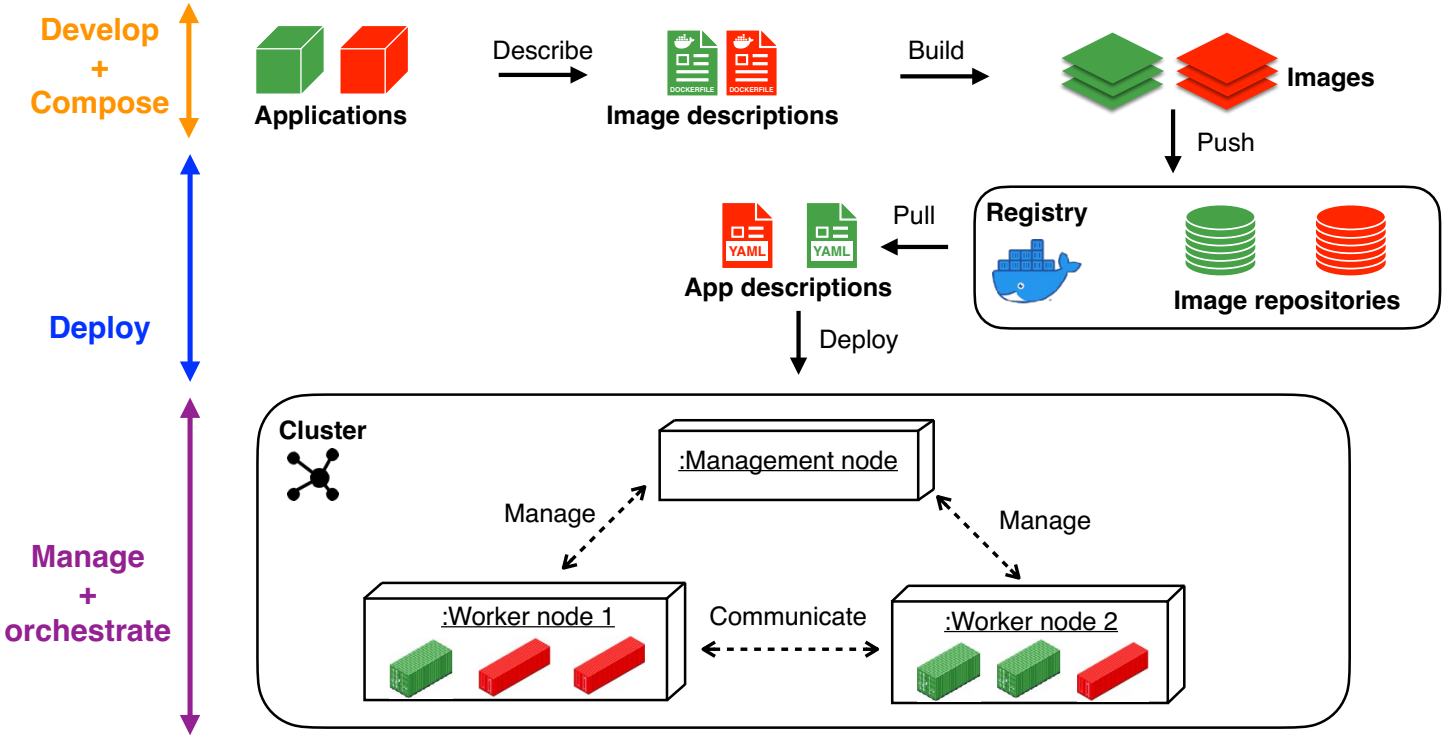
Source: https://puppet.com/blog/continuous-delivery-vs-continuous-deployment-what-s-diff

From traditional to container deployment



Source: https://www.intexsoft.com/blog/post/what-is-kubernetes.html

Example: Deployment and orchestration with containers



Course topics

1. Software engineering as a problem solving activity
2. Abstraction and modeling
3. Requirements analysis
4. System design and architectural patterns
5. Object design and design patterns
6. Testing
7. Software lifecycle modeling
8. Software configuration and release management
9. **Software quality management**
10. Project management

Software quality management (QM) activities

Process

Project

Product

Definition: **software quality**

“Conformance to explicitly stated functional and nonfunctional requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software"

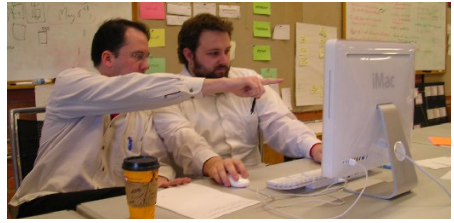
— Pressman

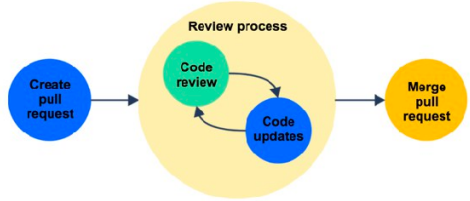
Code review

- **Objective:** improve the code quality (clean code)
- Examples: pair programming, pull requests
- **Advantages**
  - \+ Improved code quality
  - \+ Knowledge transfer
  - \+ Improved developer communication and culture
- **Disadvantages**
  - – Higher costs
  - – Slower development
  - – Repetitive tasks
- Static code analysis can automate repetitive aspects of code reviews

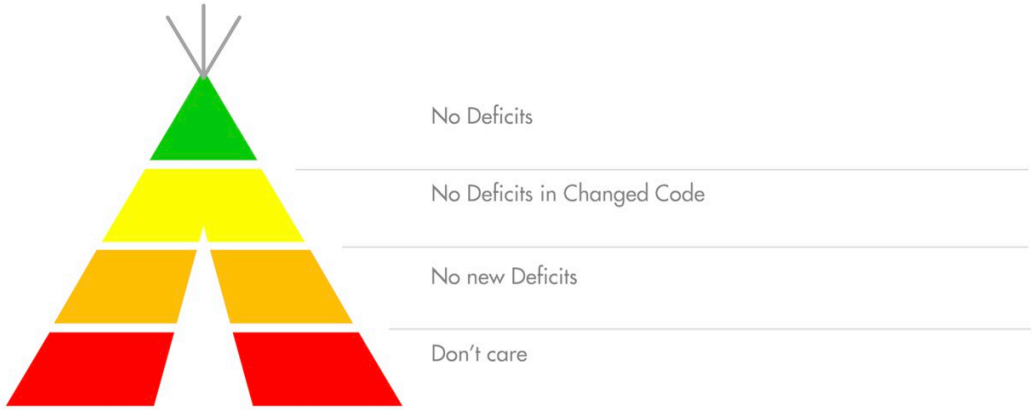
➡ However: manual code reviews are still needed and useful

➡ Best practice: only review code manually if all automatic checks have passed





Leave the code as you would like to find it

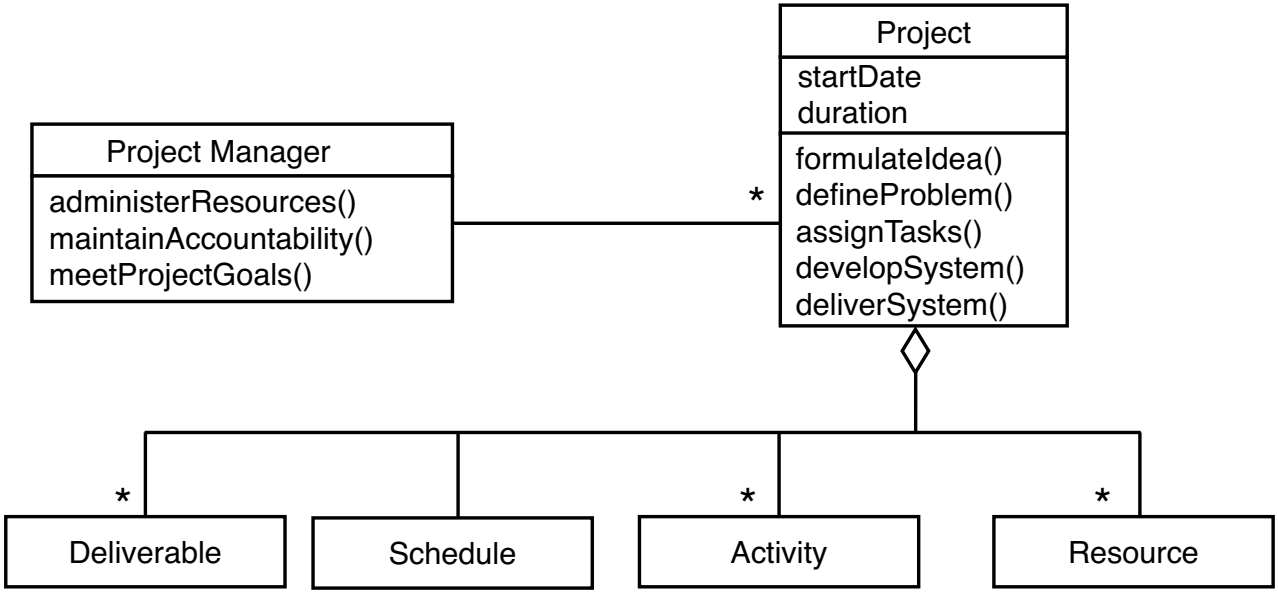


There is no silver bullet for code quality

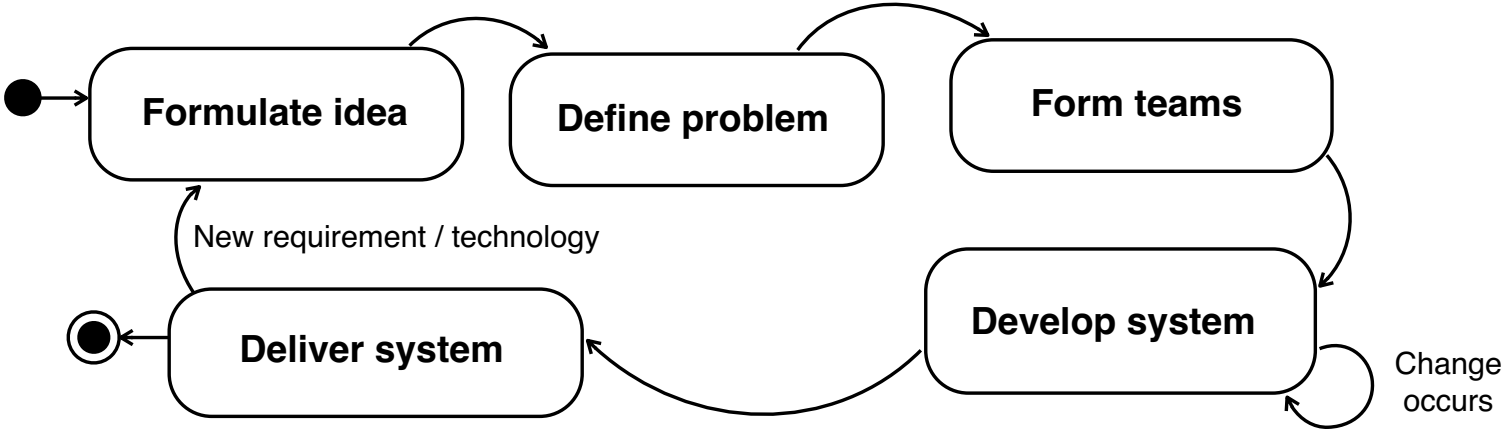
Course topics

1. Software engineering as a problem solving activity
2. Abstraction and modeling
3. Requirements analysis
4. System design and architectural patterns
5. Object design and design patterns
6. Testing
7. Software lifecycle modeling
8. Software configuration and release management
9. Software quality management
10. **Project management**

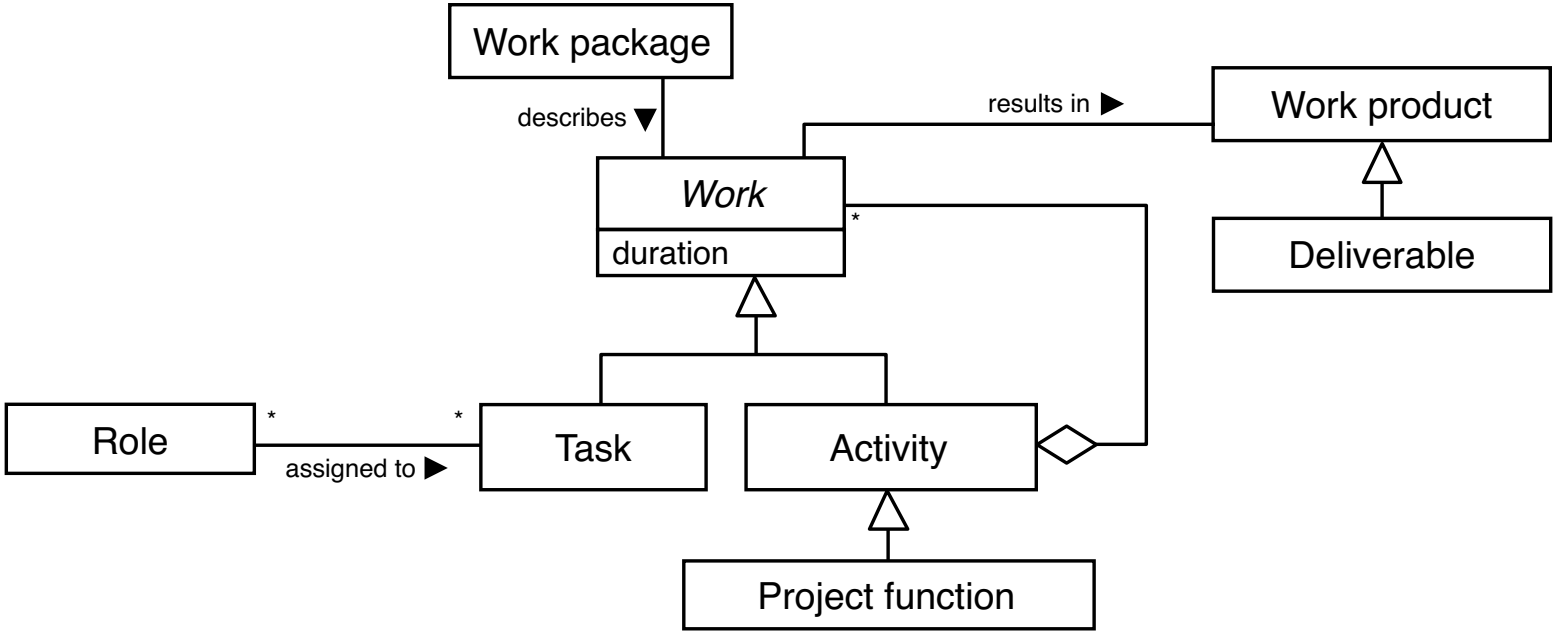
Modeling a project: **object model**



Modeling a project: **dynamic model**



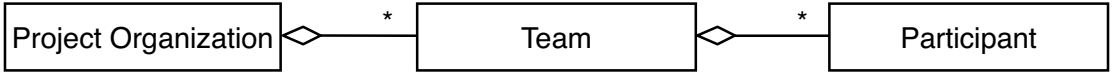
Work breakdown structure



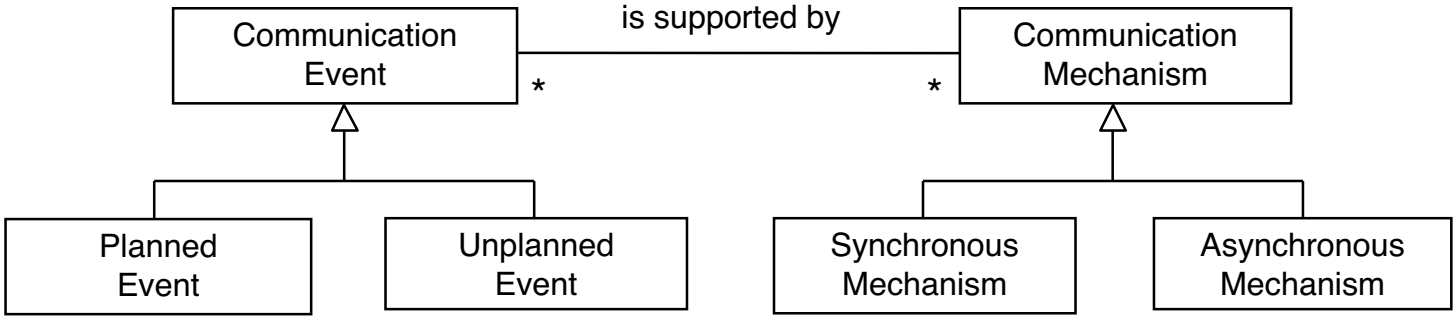
Work breakdown structure: the aggregation of the work to be performed in a project. Often called WBS (in traditional projects) or epics (in agile projects)

Project organization

- Defines the relationships among resources, in particular the participants, in a project
- A project organization should define
  - Who decides (decision structure)
  - Who reports their status to whom (reporting structure)
  - Who communicates with whom (communication structure)
- 3 types
  - Functional organization
  - Project-based organization
  - Matrix organization



Modeling communication



Good Luck!