Patterns in Software Engineering

L01 Introduction

Stephan Krusche

24 October 2022
Technical University of Munich

"Lecture hall is inadequate" pattern

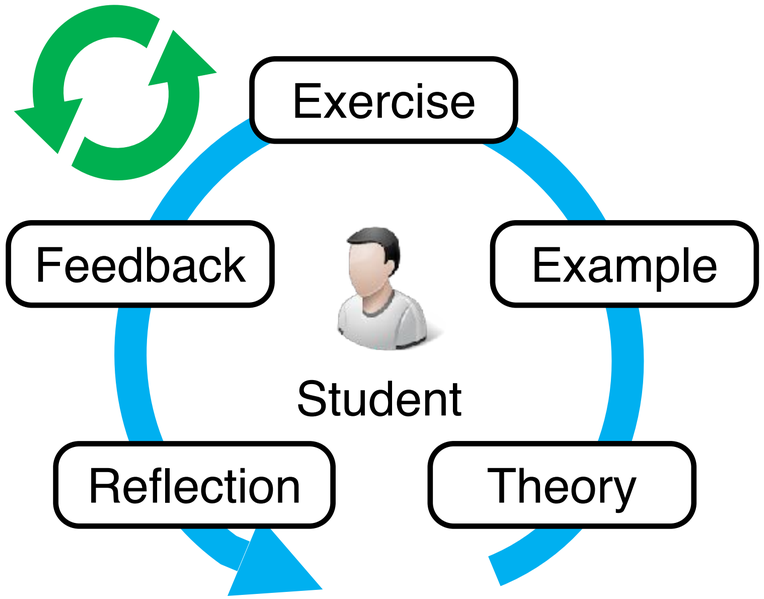
- **Context:** Lectures are overcrowded
  - ~830 registered students
  - All of you are interested in patterns in SE
- **Specific problem:**
  - We need to talk to you and you need to talk us
  - It is difficult to talk to you individually
- **Solution:**
  - We make the lecture interactive and use digital media
  - Blended, active, interactive learning

Blended Learning

- A formal education program in which a student learns at least in part through **delivery of content** and instruction via digital and online media with some element of **student control** over time, place, path, or pace
- Delivery of content: our job
- Student control: **in-class exercises**
- PSE is based on **interactive learning** (= blended learning + active learning)

Interactive learning in PSE

- **Goal:** directly involve students in the learning process using digital tools
  - Engage students in two aspects – **doing** things and **thinking** about those things
- PSE does not distinguish between lectures and exercises
  - One time slot **Monday 10:10 - 13:50** with 4 sections (40 min) per week
- In each section
  1) We teach you one (or more) concepts (e.g. patterns)
  2) You apply these concepts in **in-class exercises**
  3) You think about the concepts you are applying
- Artemis supports **interactive learning**
- Other names for this learning approach: experiential learning, just-in-time learning



Exercise philosophy

"Tell me and I will forget.

Show me and I will remember.

Involve me and I will understand.

Step back and I will act."

— Chinese Proverb

Outline

- **Course organization**
- Patterns definition
- Basic concepts 1
- Basic concepts 2

The lecture team

Prof. Dr. Stephan Krusche
Lecturer

Jan Philip Bernius
Exercise instructor

Snezhina Milusheva
Exercise instructor

+ 8 tutors (next slide)

8 tutors


Denitsa Asova


Yulia Nikirova

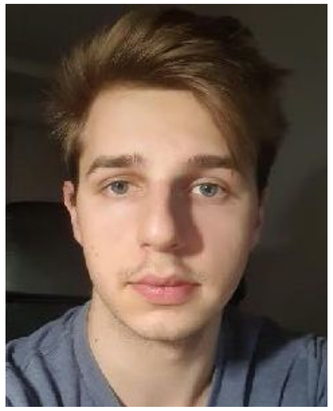

Abdelhadi Razouki


Simon Karan


Lucas Welscher

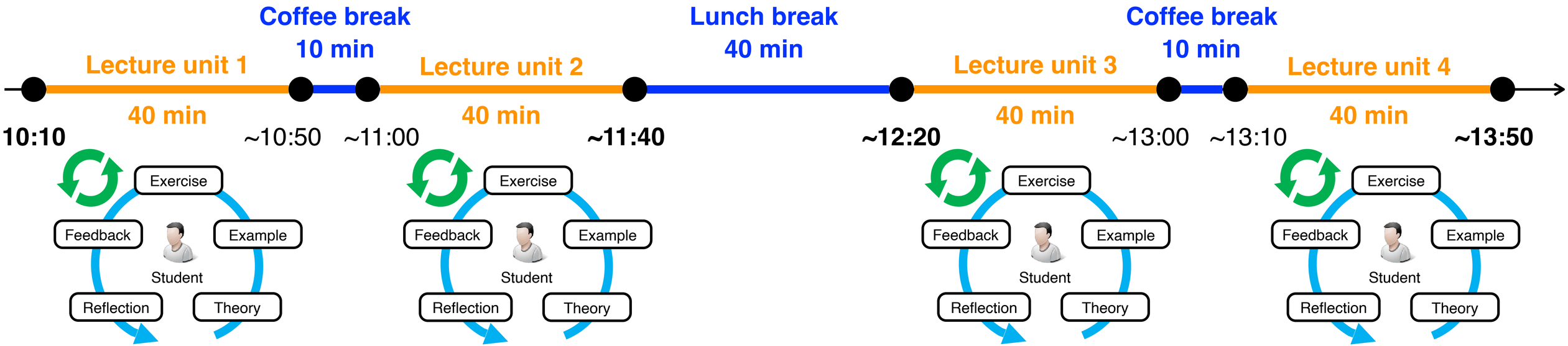

Katjana Kosic


Tobias Lippert


Michal Kawka

Lecture time and livestream

- **Monday: 10:10 - 13:50**



- Livestream: https://live.rbg.tum.de
- Lecture recordings will be available after a few days (due to editing)
- Tutors will be available in the lecture hall and online, assist you with in-class exercises and answer questions

Assumptions and requirements for this course

- **Assumption**
  - You want to learn how to apply patterns in software engineering
  - You want to learn more about modeling and object oriented software development
- **Requirements**
  - You should have passed **Praktikum: Grundlagen der Programmierung** (IN0002)
  - You should have passed **Introduction to Software Engineering** (IN0006)
  - You should have a lot of experience in object oriented programming with Java in an IDE
- **Beneficial**
  - You have practical experience with a large software system
  - You have already participated in a large software project
  - You have experienced major problems in developing a software system

Learning outcomes of the course

- Understand the principles of patterns in software development
- Understand the structure of pattern-based software systems
- Apply patterns in a variety of problem situations and in concrete applications
- Practice the application of different design and architectural patterns
- Explain the differences between patterns
- Choose the right pattern for a concrete problem statement
- Model the use of a pattern in UML
- Understand and apply test patterns
- Apply refactored solutions for antipattern and remove code smells

Course schedule (preliminary)

#	Date	Subject
	17.10.22	**No lecture, repetition week (self-study)**
1	24.10.22	Introduction
	31.10.22	**No lecture, repetition week (self-study)**
2	07.11.22	Design Patterns I
3	14.11.22	Design Patterns II
4	21.11.22	Architectural Patterns I
5	28.11.22	Architectural Patterns II
6	05.12.22	Antipatterns I
7	12.12.22	Antipatterns II
	19.12.22	**No lecture**
8	09.01.23	Testing Patterns I
9	16.01.23	Testing Patterns II
10	23.01.23	Microservice Patterns I
11	30.01.23	Microservice Patterns II
12	08.02.21	Course Review

Copyright

- Lecture slides, exercise resources, the livestream and recordings are **copyright protected**
- You are **not** allowed to distribute them to other people
- You are **not** allowed to publish them on the internet

All material of this course is protected by copyright and has been copied by and solely for the educational purposes of the university under license. You may not sell, alter or further reproduce or distribute any part of this material to any other person. Where provided to you in electronic format, you may only print it for your own private study and research.

Failure to comply with the terms of this warning may expose you to legal action for copyright infringement and / or disciplinary action by the university.

Tools in PSE

- ✓ Livestream using live.rbg.tum.de
- ✓ Artemis: https://artemis.in.tum.de
  - Download PDF slides before the lecture starts
  - Participate in exercises: modeling, quiz, text **A**, programming ⌨
  - Review your individual exercise results including feedback
  - Post questions and answers related to specific lectures and exercises
- Slack: https://patterns22.slack.com
  - Read announcements, communicate with instructors, tutors and other students
  - Post general questions about the organization
  - Post general questions about lectures and exercises
- Apollon: https://apollon.ase.in.tum.de

Require **JDK17** + **IntelliJ**

Registration for the Slack channel PSE 22/23

- Signup with your **@tum.de** email address: https://patterns22.slack.com.slack.com/signup
- Overview of workspace structure:
  - **`#announcements`** Announcements by the instructors
  - **`#exercises`** Questions and comments regarding the exercises
  - **`#lecture`** Questions and comments regarding the lecture
  - **`#organization`** Questions and comments regarding the organization
  - **`#tools`** Questions regarding the used tools

- Use these channels for their dedicated purposes
  - **Purpose:** enable communication, ask questions, share answers with everyone
  - Do **not** offend or bully each other! Do **not** distribute hate speech or fake news!

Code of conduct

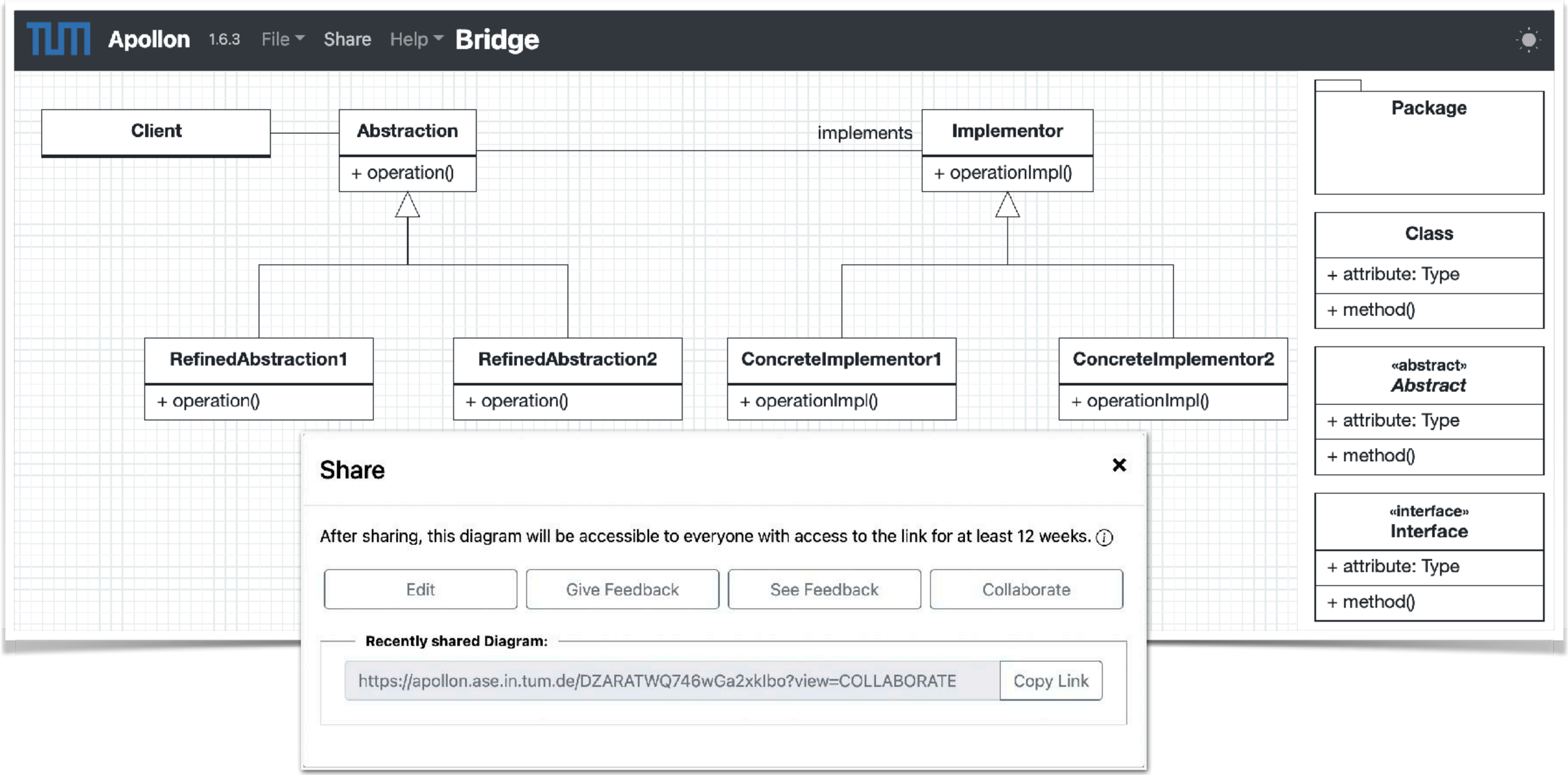
- Respect others and their opinions
- Use appropriate language
- Stay on topic
- Use threads to reply to a specific question
- Do **not** abuse the platform and follow the laws
- Do **not** impersonate someone else
- Do **not** post answers / solutions to exercises
- Do **not** spam and wait patiently for a response

🔗 TUM Digital Learning Commitment

Violations may result in a permanent exclusion from Slack, or may even lead to an exclusion from Artemis!

You can find the full version of the code of conduct as PDF on https://artemis.in.tum.de/courses/209/lectures/446

Apollon


Apollon 1.6.3 File Share Help Bridge
Client
Abstraction
+ operation()
implements
Implementor
+ operationImpl()
RefinedAbstraction1
+ operation()
RefinedAbstraction2
+ operation()
ConcreteImplementor1
+ operationImpl()
ConcreteImplementor2
+ operationImpl()
Package
Class
+ attribute: Type
+ method()
«abstract»
Abstract
+ attribute: Type
+ method()
«interface»
Interface
+ attribute: Type
+ method()
Share
After sharing, this diagram will be accessible to everyone with access to the link for at least 12 weeks.
Edit
Give Feedback
See Feedback
Collaborate
Recently shared Diagram:
https://apollon.ase.in.tum.de/DZARATWQ746wGa2xklbo?view=COLLABORATE
Copy Link

Final exam: onsite + computer based

- **Date:** not yet scheduled
- **Time:** 75 min
- Focuses on **problem solving** and the **application** of software engineering knowledge
  - Exam questions will be similar to in-class exercises
  - **Note:** learning by heart won't help you
- You can get up to 100 points in the exam
  - Bring your own notebook
  - Open Internet exam (no communication allowed)
- **Note: no** repeat exam

Possible grade key*

Exam points	➡	Grade
[0, 40)	➡	5.0
[40, 45)	➡	4.7
[45, 50)	➡	4.3
[50, 55)	➡	4.0
[55, 60)	➡	3.7
[60, 65)	➡	3.3
[65, 70)	➡	3.0
[70, 75)	➡	2.7
[75, 80)	➡	2.3
[80, 85)	➡	2
[85, 90)	➡	1.7
[90, 95)	➡	1.3
[95, 100]	➡	1.0

$(a, b] = \{ x \mid a < x \leq b \}$ $[a, b) = \{ x \mid a \leq x < b \}$

Rounding: to the nearest first decimal place (half up)

* subject to change

Bonus system

- Participate in **in-class** exercises to receive points 🏆
- Mapping of your **exercise score** in Artemis to **additional exam points**
  - Prerequisite: you have passed the exam! (**hard requirement**)
- **Additional requirement:** you must create at least 2 screencast presentations of your exercise solutions
  - Explain the source code next to the UML model (e.g. IDE and Apollon next to each other)
  - Submit the recorded presentation to a tutor
  - Check your presentation score on Artemis (within 7 days)
  - More details and a tutorial how to create screencasts will follow soon

Exercise score	—>	Additional exam points
[0%, 5%)	->	0.0
[5%, 10%)	->	0.5
[10%, 15%)	->	1.0
...	...	...
[90%, 95%)	->	9.0
[95%, 100%)	->	9.5
100 %	->	10.0

$(a, b] = \{ x \mid a < x \leq b \}$ $[a, b) = \{ x \mid a \leq x < b \}$

Bonus examples

- You cannot get a better final grade than 1.0
- Your bonus only applies if you pass the exam with at least 4.0

Exam points	➡	Exam grade	Exercise score	➡	Additional exam points	Final points*	➡	Final grade
49.5	➡	4.3	100 %	➡	10	49.5	➡	4.3
66.0	➡	3.0	47.8 %	➡	4.5	70.5	➡	2.7
75.0	➡	2.3	49.9 %	➡	4.5	79.5	➡	2.3
79.5	➡	2.3	5.0 %	➡	0.5	80.0	➡	2.0
100.0	➡	1.0	100 %	➡	10.0	100.0	➡	1.0

Your bonus only applies if you pass the final exam with at least 4.0

You cannot get a better final grade than 1.0

* **Final points** = **exam points** + **additional exam points** (only if exam points >= 50)

Exercise rules

- No obligation to participate, but **highly recommended**
- You can only submit your exercise solutions on Artemis
- Each exercise has a **due date** until it should be completed
  - If you miss this deadline, your solution will **not** be assessed
  - You can still participate after the due date (called "ungraded results" and "practice" mode)
- In-class exercises must be completed and submitted **independently**
  - Every student has to solve and submit the exercises individually (group work is not allowed)
  - There are no exceptions and there are no warnings, the first plagiarism case will be punished

Typically until the next lecture

Cheating ("Unterschleif") leads to exclusion from bonus system and may even lead to the grade "U void/fraud - 5,0 nicht ausreichend" in TUMonline which has severe consequences

- Please take into account: offering such a bonus is a lot of work for us
  - If you do not agree to these rules, please do not participate in the exercises!

Consequences of plagiarism

- Allgemeine Prüfungs- und Studienordnung (APSO)

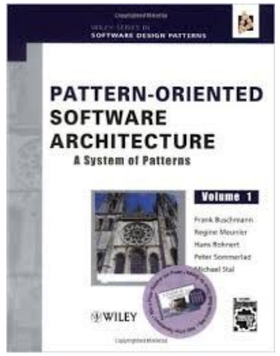
  General examination and study regulations

  https://portal.mytum.de/archiv/kompendium_rechtsangelegenheiten/apso/Lesb-F-APSO-vom-18-03-2011-mit-5-AeS-vom-21-12-2020.pdf/download

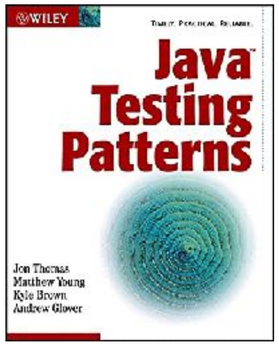
- Only available in German, but translated to English here for your convenience
  - **§ 22:** "If the student attempts to influence the result of an examination by cheating or using unauthorized aids, the examination in question will be graded unsatisfactory", i.e. **5,0 (U)**
  - **§ 24:** "Otherwise, failed module examinations may be repeated as often as desired, subject to the deadlines specified in § 10. This does not apply in the case of failure due to cheating or a breach of regulations according to § 22. In this case, the failed examination may only be repeated once."

Main literature

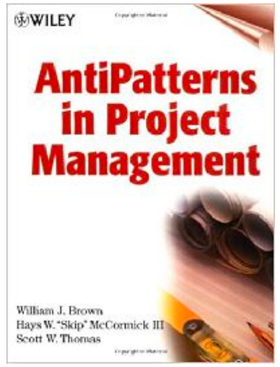
Design Patterns. Elements of Reusable Object-Oriented Software (Prentice Hall), 1994



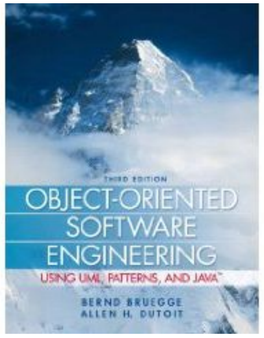
Pattern-Oriented Software Architecture (Wiley), 1996



Java Testing Patterns (Wiley) 2004



Anti-Patterns and Patterns in Software Configuration Management (Wiley), 1999



Object-Oriented Software Engineering: Using UML, Patterns and Java (Pearson Education), 2009

Outline

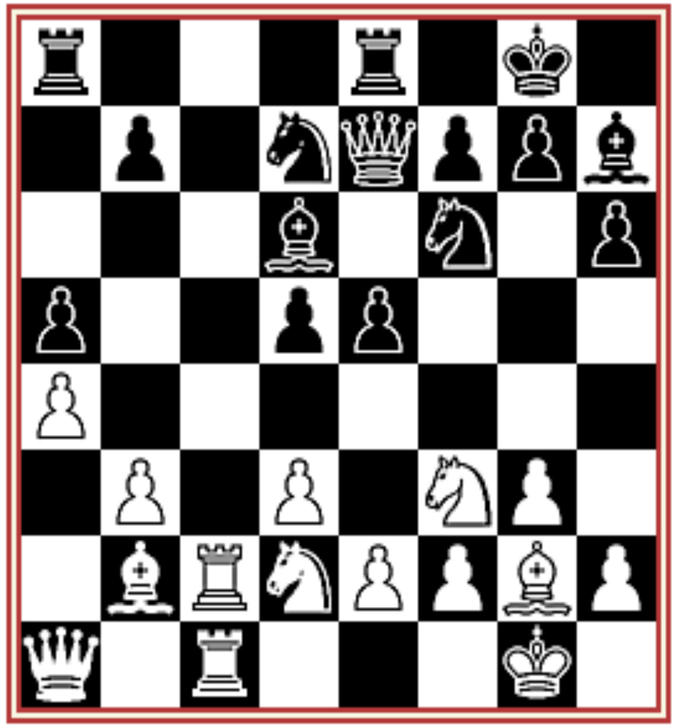
- Course organization
- **Patterns definition**
- Basic concepts 1
- Basic concepts 2

What is this?

- 1.Nf3 d5 2.c4 c6 3.b3 Bf5 4.g3 Nf6 5.Bg2 Nbd7 6.Bb2 e6 7.O-O Bd6 8.d3 O-O 9.Nbd2 e5 10.cxd5 cxd5 11.Rc1 Qe7 12.Rc2 a5 13.a4 h6 14.Qa1 Rfe8 15.Rfc1

It is called a
domain-specific language (DSL)

Another View



- Snapshot of the game Reti-Lasker, New York 1924
- Comment from a chess master: "Lasker has the center, but Reti has fianchettoed, so he has the advantage…."

Fianchetto: one basic building block of chess knowledge

There is a **pattern** behind this

It is called **fianchetto**: usually taught in books about chess openings

- Comment in a newspaper chess column:
- "We can see that Reti has allowed Lasker to occupy the center but Reti has positioned both Bishops to hit back at this, and has even backed up his Bb2 with a Queen on a1!"

Patterns are building blocks of software engineering knowledge

- Objective of this course: **appreciate** patterns
- Patterns are a **fundamental** concept
- Helpful in managing software engineering knowledge
- They are useful "knowledge pieces"
- Build software systems in the context of **frequent change** by **reducing complexity** and **isolating change**

Desired outcome of the course

- At the end of the semester you understand how to
  - produce a high quality software system based on patterns that reduce the complexity and isolate the effect of change
- You have also acquired **technical** knowledge
  - Patterns in software lifecycle activities
  - Design patterns, architectural patterns, testing patterns
- And you have acquired **managerial** knowledge
  - Patterns in management activities
  - Antipatterns, communication patterns…

Acquire **technical** knowledge

- Understand patterns
- Learn a catalog of patterns
- Understand reuse and different reuse scenarios
- Master system complexity through patterns
- Be able to do pattern-based software development

Acquire **managerial** knowledge

- Pattern-based project management
- Patterns in cross-functional activities
- Knowledge management
- Proactive change

Software development is more than just writing code

- **Problem solving**
  - Understand a problem
  - Propose a solution and plan
  - Engineer a system based on the proposed solution using a good design
- **Dealing with complexity**
  - Creating abstractions with models
  - Notations for abstractions
- **Knowledge management**: elicitation, analysis, design, validation of the system and the solution process
- **Rationale management**: making the design and development decisions explicit to all stakeholders involved

How can we describe software engineering knowledge?

- Software engineering knowledge is **not** only a set of **algorithms**
- It also contains a **catalog of patterns** describing **generic solutions** for **recurring problems** which cannot be coded in a programming language
  - Description usually in natural language
  - Presentation in form of a schema consisting of sections with text and pictures (drawings, UML diagrams, etc)

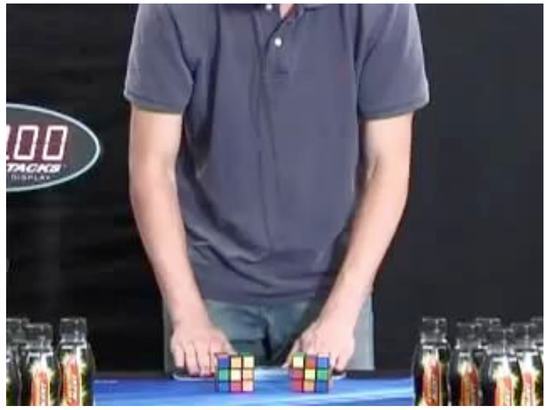
Algorithm vs. pattern

- **Algorithm**
  - A method for solving a problem using a finite sequence of well-defined instructions for solving a problem
  - Starting from an initial state, the algorithm proceeds through a series of successive states
  - Terminates in a final state
- **Pattern**
  - „A pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem in such a way that you can use this solution a million times over, without ever doing it the same way twice"
    --- Christopher Alexander, A Pattern language

Algorithm examples: solving the Rubik cube

- **Speed:** Rubik's cube in 6 seconds
- **Concurrency:** 2 cubes at the same time
- **Young user:** 5 year old kid
- **Handicapped user:** blindfolded user
- Rubik cube by a **robot**

http://www.youtube.com/watch?v=jI_zjWssn2g



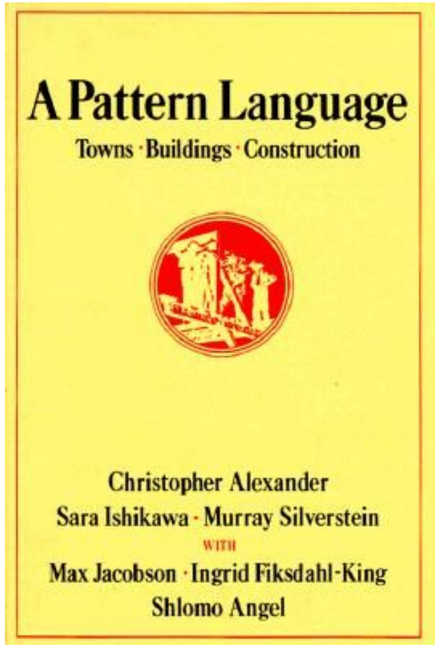
http://www.youtube.com/watch?v=RW3akfdEGI8

http://www.youtube.com/watch?v=uNcf7KD3QUg

https://www.youtube.com/watch?v=LtPnVGU8q2k

http://www.youtube.com/watch?v=bNAnUygqOYc

Where did it start?



- Christoph Alexander
- A Pattern Language: Towns, Buildings, Construction
- Oxford University Press, 1971

Christopher Alexander
* 1936 Vienna, Austria
- More than 200 building projects
- Creator of the "Pattern language"
- Professor emeritus at UCB

Pattern definition

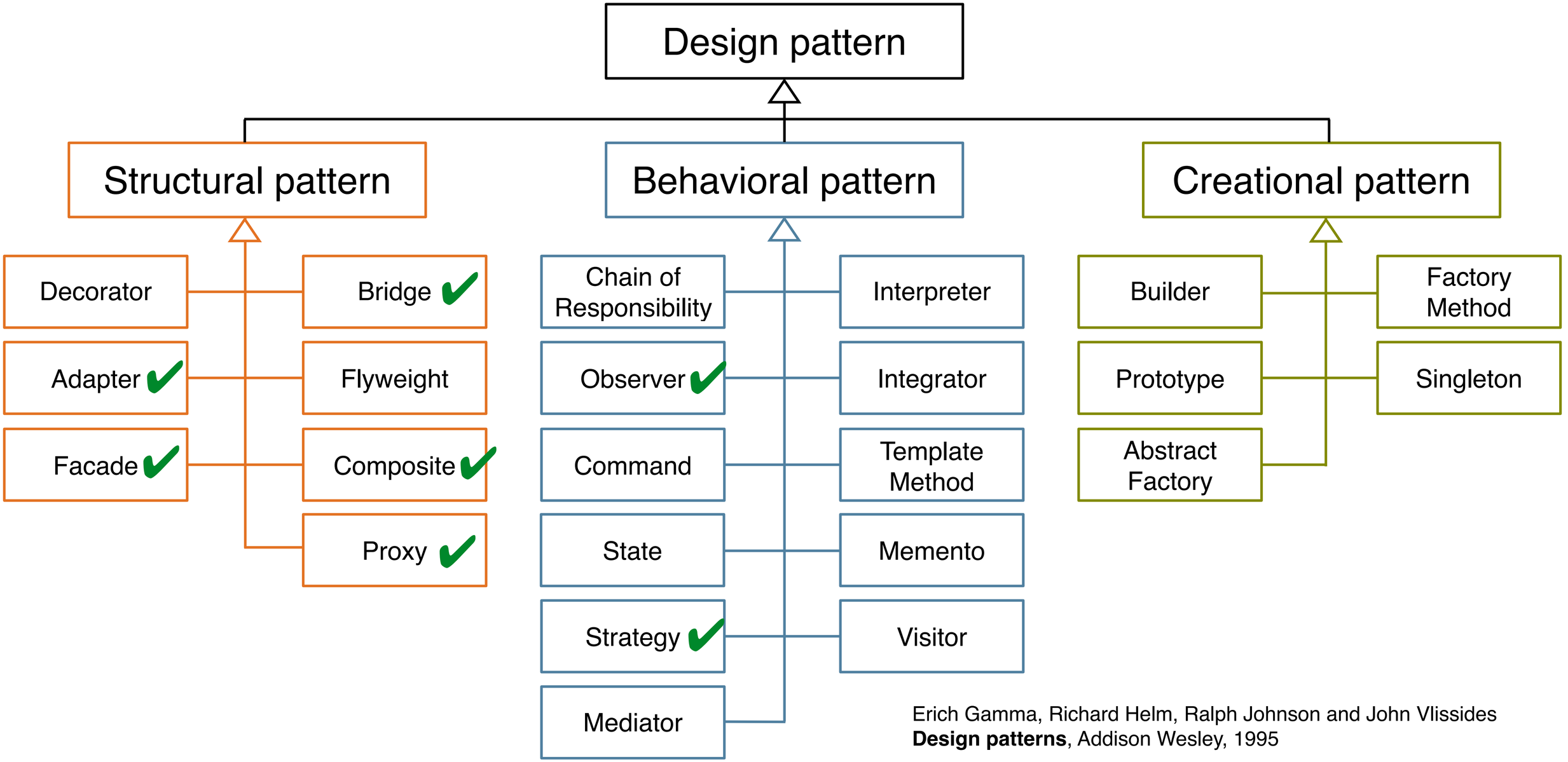
- Original definition (Christopher Alexander): a pattern is a three-part rule, which expresses a relation between a certain **context**, a **problem**, and a **solution**
- Patterns originated in architecture
  - "Buildings have been built for thousands of years by users who where not architects"
  - "Users know more about what they need from buildings and towns than an architect"
  - "Good buildings are based on a set of design principles that can be described with a pattern language”

Although Christopher Alexander’s pattern language is about architecture and urban planning, his ideas are applicable to many other disciplines, in particular software engineering

However: end users need architects…

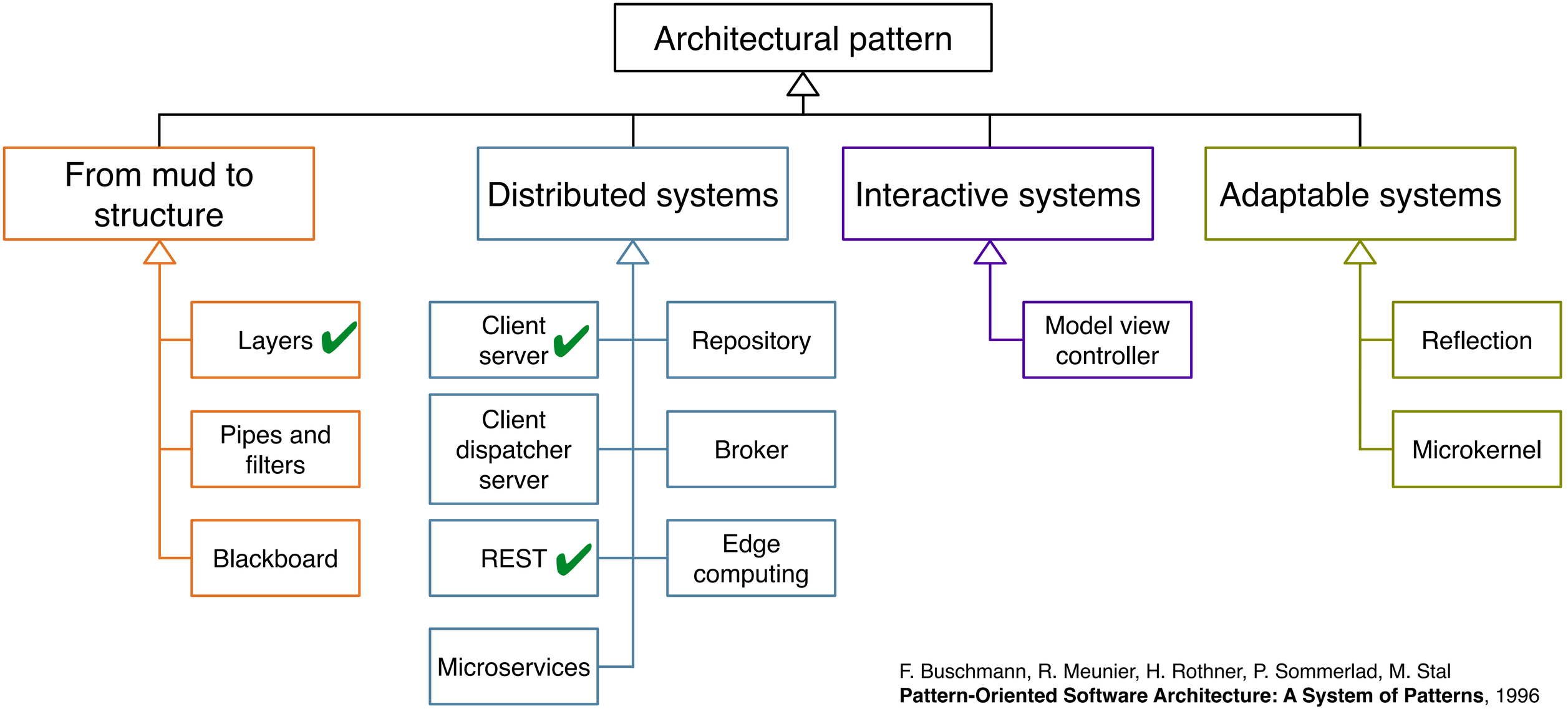
… desperately

Design patterns overview



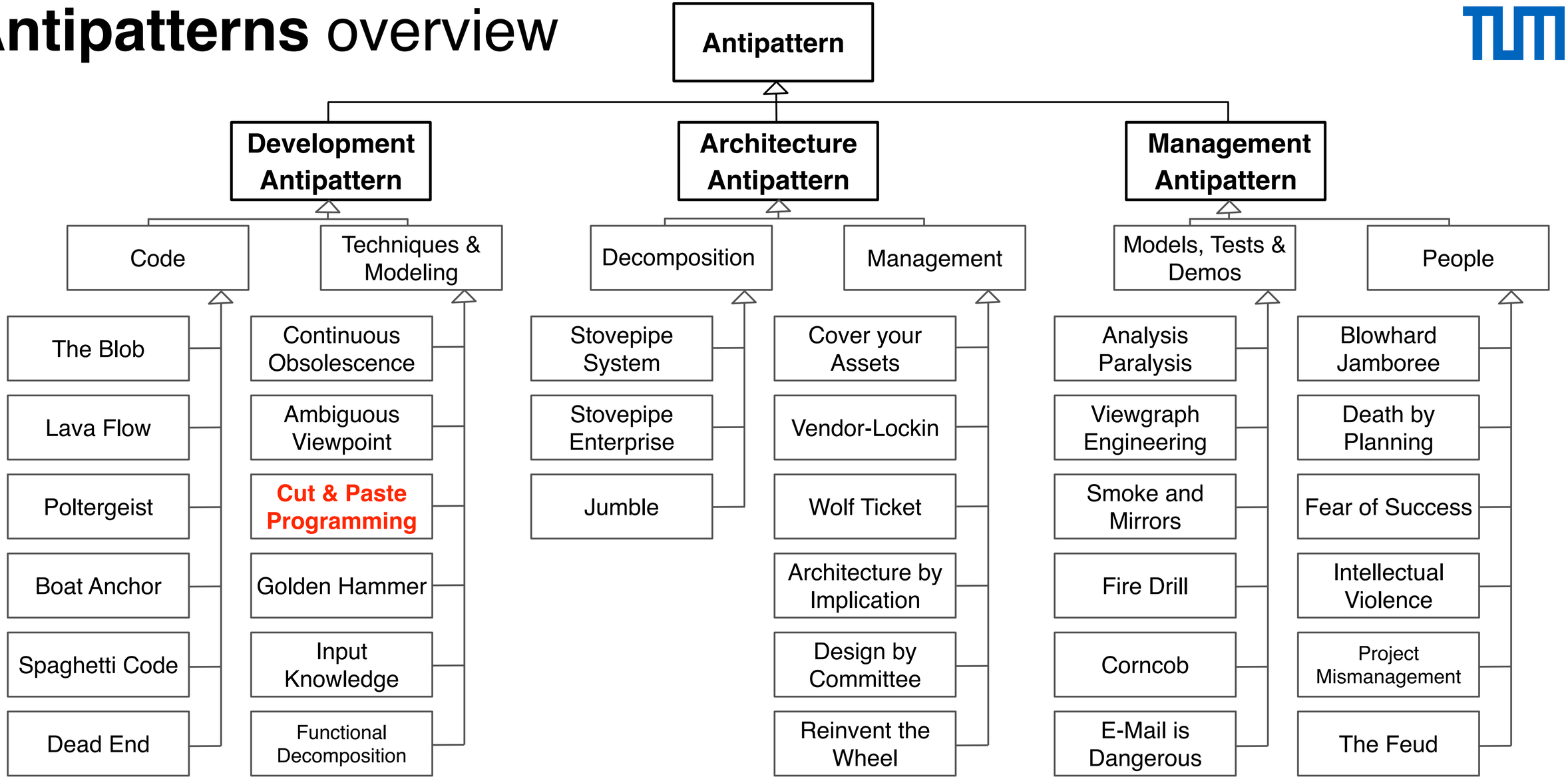
Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides
Design patterns, Addison Wesley, 1995

Architectural pattern overview



F. Buschmann, R. Meunier, H. Rothner, P. Sommerlad, M. Stal
Pattern-Oriented Software Architecture: A System of Patterns, 1996

Antipatterns overview

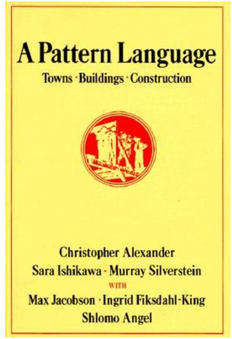


William J. Brown: **Anti-Patterns and Patterns in Software Configuration Management**, Wiley 1999

Schemata for describing patterns

1. Alexander's Schema ("Alexandrian Form")

- **A Pattern Language – Towns Buildings Construction**, Christopher Alexander, Sara Ishikawa, Murray Silverstein, Vol. 2, Oxford University Press, New York, 1977

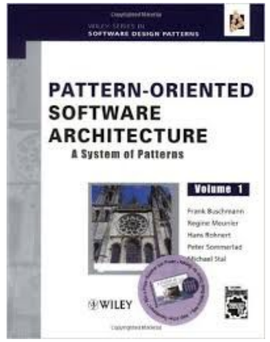


2. Gang of Four Schema

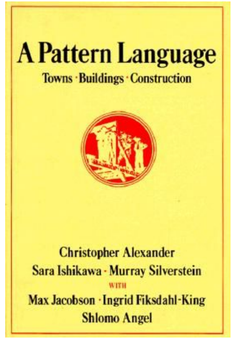
- **Design Patterns: Elements of Reusable Object-Oriented Software** by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison Wesley, October 1994

3. Gang of Five Schema

- **Pattern-Oriented Software Architecture: A System of Patterns**, Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal, Wiley and Sons Ltd., 1996



1. Alexander’s Schema ("Alexandrian Form")



- **Name of the Pattern**
- **Example:** picture of an example for the pattern
- **Context:** the range of situations in which the pattern can be used
- **Problem**: short description and elaborate description
- **Solution**: description and diagram
- **Conclusion**
- Sections don‘t have explicit headings, sections are separated by symbols
  - 3 diamonds between context and problem and after the solution
  - 3 diamonds between solution and conclusion
- Keyword **"therefore"** to separate the problem from the solution

2. Gang of Four Schema

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

- Pattern name
- Intent
- Also known as
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample code
- Known uses
- Related patterns

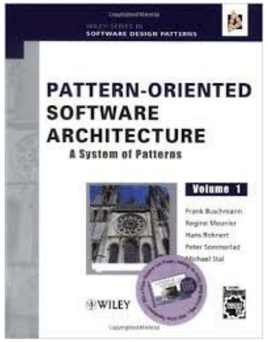
See: pages 6-7 [Gamma et al 95]

3. Gang of Five Schema

Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stal

- Name
- Also known as
- Example
- Context
- Problem
- Solution
- Structure
- Dynamics
- Consequences
- Implementation
- Sample code
- Known uses
- Related patterns

See: pages 20-21 [Buschmann et al 96]



Pattern language and pattern catalogs

- **Pattern language**
  - Collection of patterns and the rules to combine them into an architectural style
  - Form a vocabulary for understanding and communicating ideas
  - Describe software frameworks or families of related systems
- **Pattern catalog**
  - Collection of related patterns
  - Typically subdivides the patterns into at least a small number of broad *categories*
  - May include some amount of cross referencing between patterns

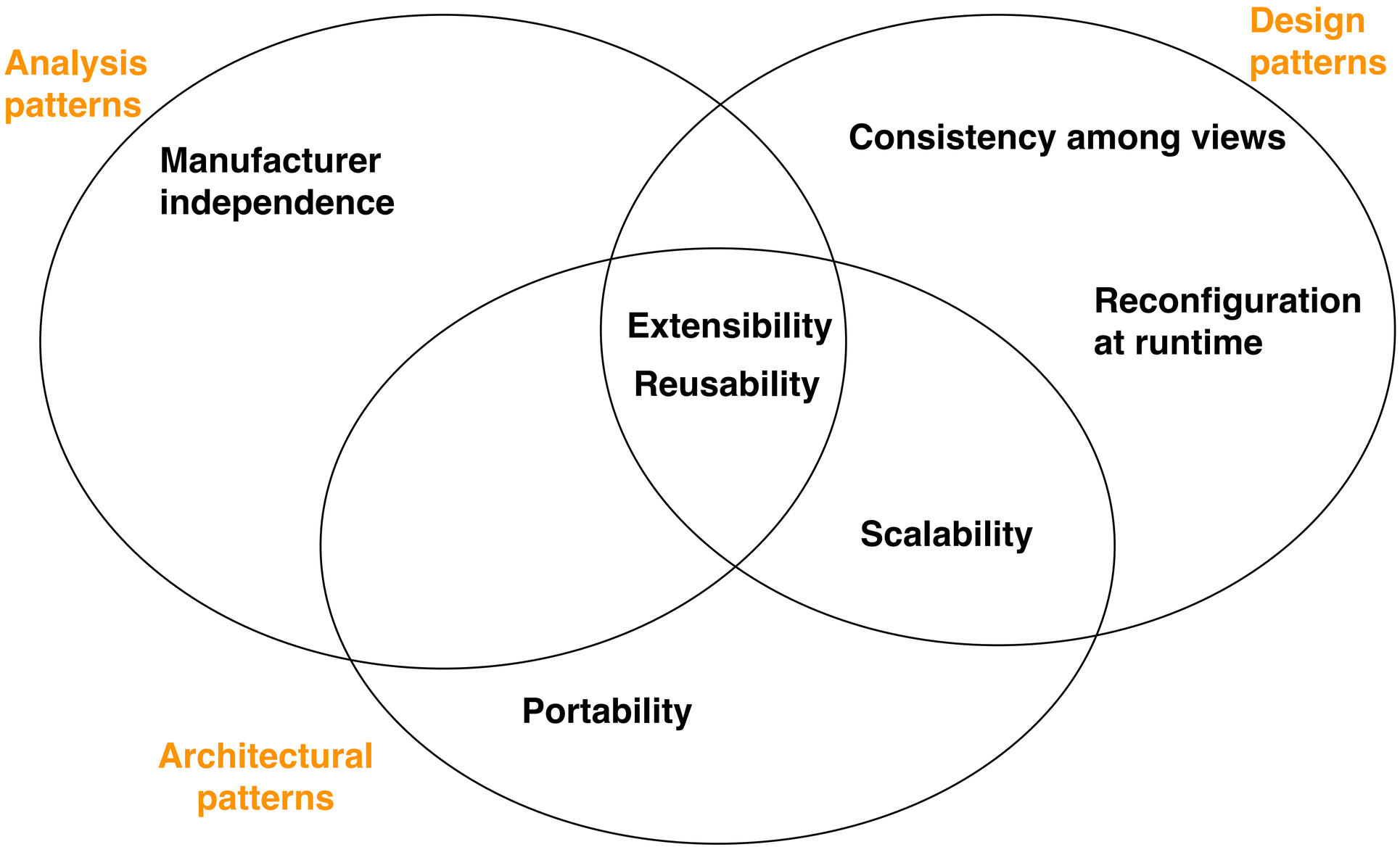
Categorization of patterns (**Gang of Five**)

- **Architectural pattern**
  - Fundamental structural organization for software systems
  - Set of predefined subsystems, specifies responsibilities, and includes guidelines for organizing the relationships between them
- **Design pattern**
  - Scheme for the software subsystems and the relationships between them
  - Common structures of communicating components to solve a general problem within a particular context
- **Idiom (coding pattern)**
  - Low-level pattern specific to a programming language
  - Implementation of particular aspects of components or the relationships between them using the features of the given language

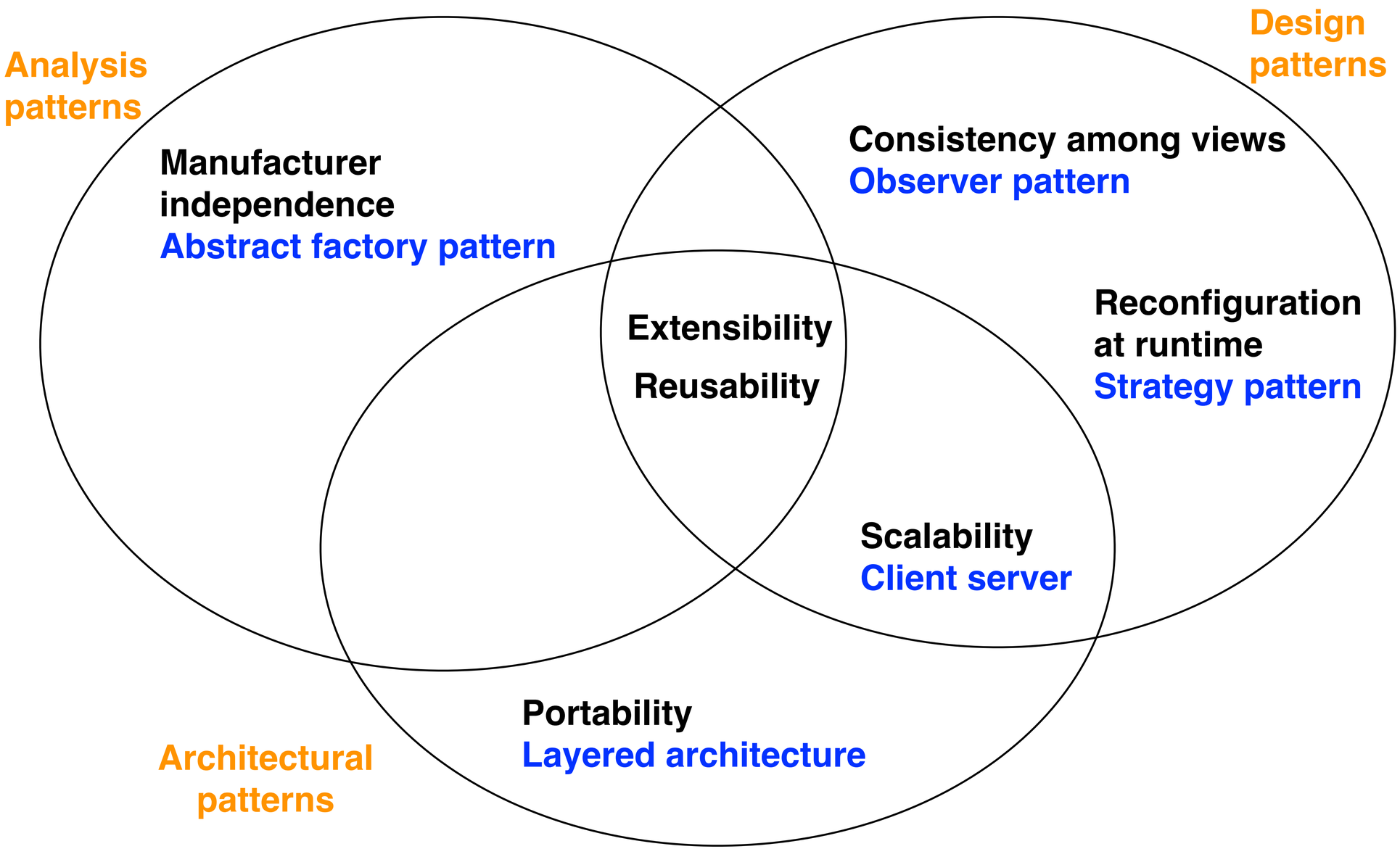
Categorization used in the PSE course

- Patterns for **development** activities
  - Analysis patterns
  - Architecture patterns
  - Design patterns
  - Testing patterns
- Patterns for **cross-functional** activities
  - Process patterns
  - Agile patterns
  - Build and release management patterns
- **Antipatterns** and **code smells**

Design patterns address nonfunctional requirements



Design patterns address nonfunctional requirements



A **L01E01 Strategy vs. bridge pattern** Not started yet.

Start exercise **Easy** **Due date in 7 days**

5 min

4 pts

- **Problem statement:** Explain the differences between the strategy and the bridge pattern in your own words

Outline

- Course organization
- Patterns definition
- **Basic concepts 1**
- Basic concepts 2

Important features of object-oriented programming

- Encapsulation
- Information hiding
- Inheritance
- Abstraction
- Polymorphism
- Delegation

Encapsulation

- Objects are a self-contained set of data and behavior

- An object can determine which part of its data and behavior is exposed to the outer world
  - Access modifiers: public, private, protected

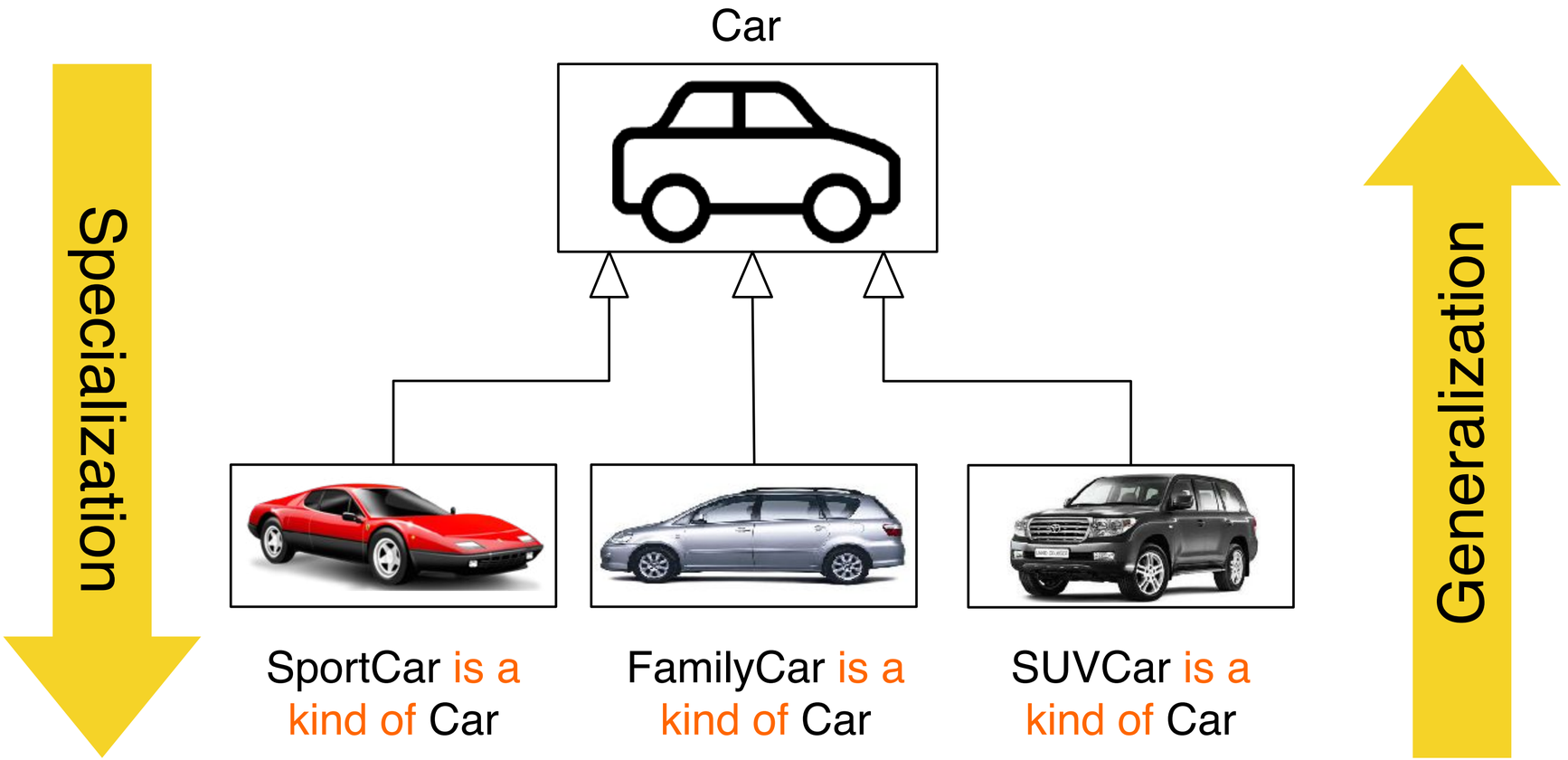
- What an object exposes to the outer world is called its public interface

Information hiding

- Principle of information hiding (David Parnas): a **caller** (class, subsystem) does not need to know anything about the internals of the **callee**

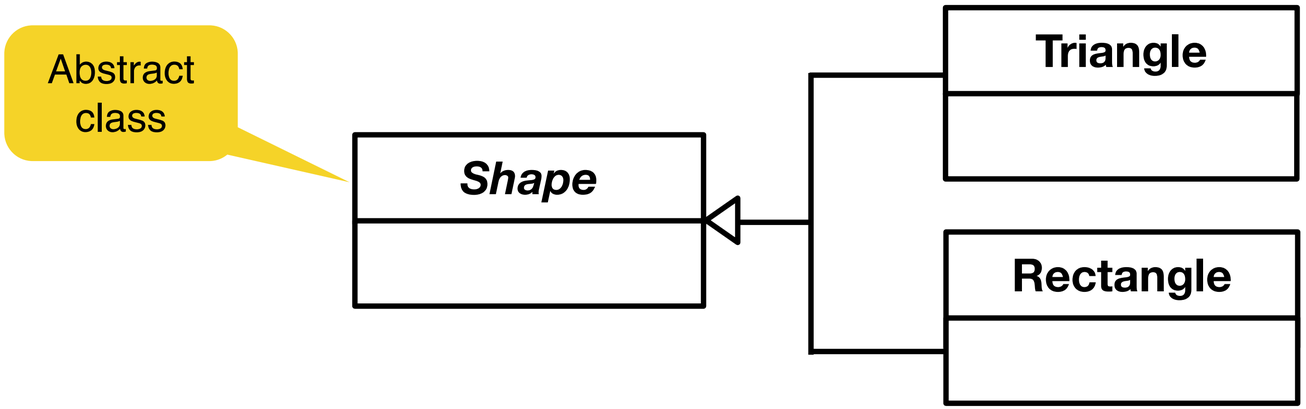
- Good object design heuristic: use visibility modifiers to minimize the visibility of attributes and operations
  - Make all attributes of a class private
  - Make all the operations of a class private, unless the operations are needed by a client of the class ← Object of another class
  - Only public methods can be used to modify a classes attributes

Inheritance



Abstraction

- Model the problem in terms of classes and the relationships between them
- Abstraction allows to define an *abstract* structure which holds common states, behaviors or attributes
- Abstract classes define what needs to be shared among sub classes, yet it is not possible to create an instance of an abstract class
- Example: geometric shapes



- Abstract classes in Java can be even more restrictive forcing sub classes to implement a method as specified inside the abstract super class

Coupling and cohesion

- **Coupling:** measures the dependencies between subsystems
- **Cohesion:** measures the dependencies among classes within a subsystem

- What makes **good** design?

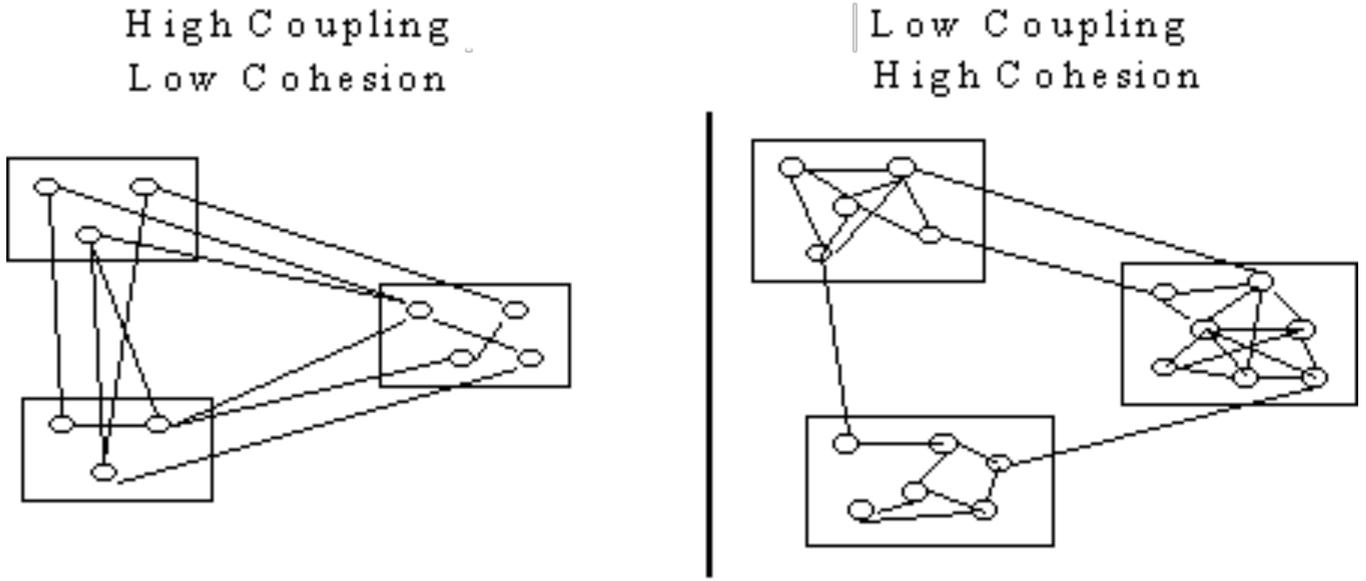
Coupling and cohesion

- **Coupling:** measures the dependencies between subsystems
- **Cohesion:** measures the dependencies among classes within a subsystem

- What makes **good** design?
  - **Low** coupling
    - The subsystems should be as independent as possible
    - A change in one subsystem should not affect any other subsystem
  - **High** cohesion
    - A subsystem should only contain classes which depend heavily on each other

Coupling and cohesion

- **Low** coupling
  - The subsystems should be as independent as possible
  - A change in one subsystem should not affect any other subsystem
- **High** cohesion
  - A subsystem should only contain classes which depend heavily on each other.



Law of demeter (LoD)

- To achieve **low coupling** and **high cohesion**, follow the **principle of least knowledge**, also called **law of demeter**
- A method M of an object O may only invoke the methods of the following kinds of objects
  - O itself
  - M's parameters
  - any objects created/instantiated within M
  - Objects that are O's instance variables
- For Java, this can be formulated as "Use only one dot"
  - **Use only one level of method calls**
    - `object0.doSomething();`
  - **Try not to use two or more levels of method calls**
    - `object0.getObjectP().getObjectQ().doSomething();`

train wreck

"Ask, don’t tell" - Principle

- **Bad example:**

```
SortedList pseStudents = tum.getStudentList(Lectures.PSE);
pseStudents.addElementWithKey(student.getKey(), student);
```

- **Problem:** assumes knowledge about the internals of the studentList representation in **`tum`**

- **Better (using the "ask, don’t tell" principle):**

```
tum.addStudentToLecture(student, Lectures.PSE);
```

- Advantages: less coupling, not dependent on implementation details
- Disadvantages: many wrappers needed, can lead to inefficiency

What is good design?

- Good design prevents **code duplication**
- Good design reduces **complexity**
- Good design is prepared for **change**
- Good design provides **low coupling**
- Good design provides **high cohesion**

L01E02 Model a car rental system in UML Not started yet.

Start exercise | Easy | Due date in 7 days

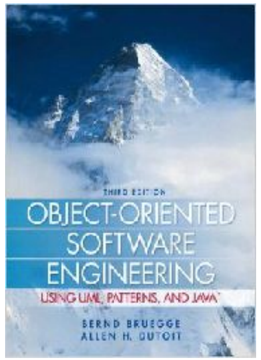
10 min

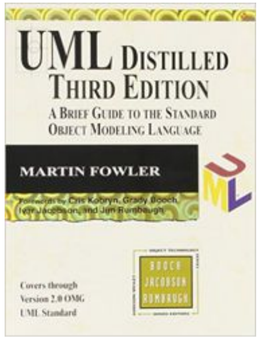
7 pts

- **Problem statement:** your customer, the owner of "AwesomeCar" company, has asked you to develop a car rental system. The system allows persons to rent cars. A car can be a sport car …
- Read the full problem statement on Artemis
- **Your task:** model this system using a UML class diagram
- **Hints**
  - Use Abbott's technique to identify nouns, adjectives and verbs
  - Add those as classes, attributes, methods and associations

Helpful resources for modeling

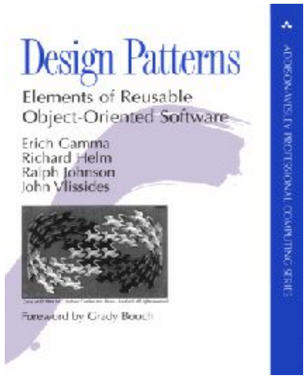
- UML- Bernd Bruegge and Allen H. Dutoit
  - Object-Oriented Software Engineering Using UML, Patterns, and Java
  - Chapter 2 Modeling with UML
  - Chapter 10 Mapping Models to Code
- UML Distilled: A Brief Guide to the Standard Object Modeling Language (3rd Edition) – Martin Fowler





Helpful resources for mapping models into languages

- Patterns in Java
  - Head First Design Patterns: Entertaining Introduction to patterns with Java (available in TUM digital library)
  - https://sourcemaking.com
- Patterns in C++
  - GoF book uses C++
  - Additional source Wikibook: http://en.wikibooks.org/wiki/C%2B%2B_Programming/Code/Design_Patterns



Important features of object-oriented programming

- ✓ Encapsulation
- ✓ Information hiding
- ✓ Inheritance
- ✓ Abstraction
- Polymorphism
- Delegation

Polymorphism

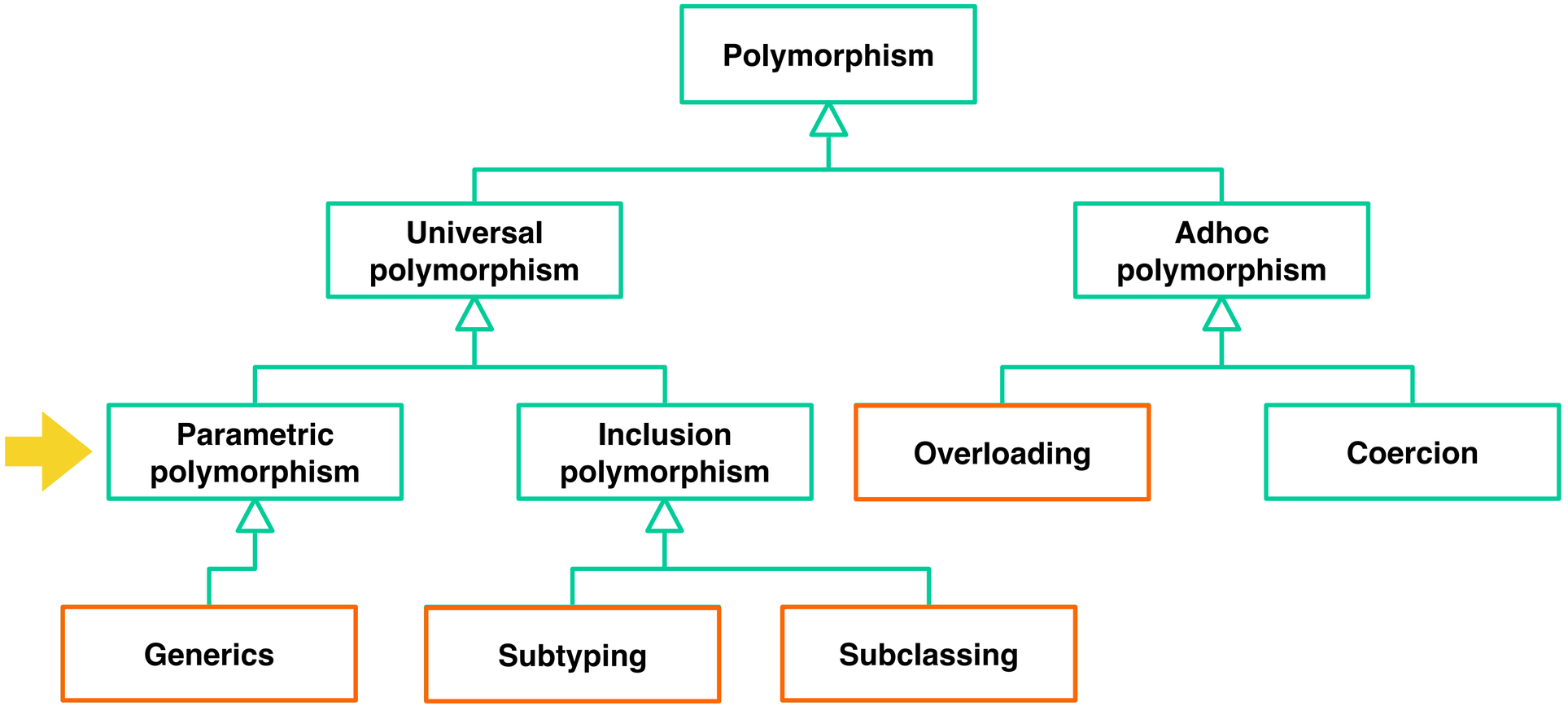
Polymorphism (general definition)

- The ability of an object to assume different forms or shapes
- Ancient greek for
  - πολύς (polys) = many, much
  - μορφή (morphe) = form, shape

Polymorphism (computer science)

- The ability of an abstraction to be realized in multiple ways
- The ability of an interface to be realized in multiple ways
- The dynamic treatment of objects based on their type

Polymorphism: a taxonomy



Adapted from: Cardelli and Wegner, On Understanding Types, Data Abstraction, and Polymorphism. 1985

Parametric polymorphism - generics

Consider the following **List** implementations:

```
class StringList {
    boolean add(String x) {…}
    String get(int index) {…}
}

class DateList {
    boolean add(Date x) {…}
    Date get(int index) {…}
}
```

For reasons of brevity, we omit the method implementations and visibility modifiers

Can we exploit the similarities of these classes?

Yes, by using parametric polymorphism (that is, by introducing a **generic** type)

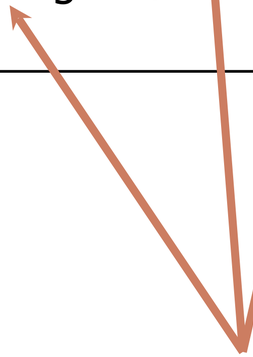
Parametric polymorphism - example

We define a generic type **`List`**!

```
class StringList {
    boolean add(String x) {…}
    String get(int index) {…}
}

class DateList {
    boolean add(Date x) {…}
    Date get(int index) {…}
}
```

```
class List<E> {
    boolean add(E x) {…}
    E get(int index) {…}
}
```



E is a type parameter

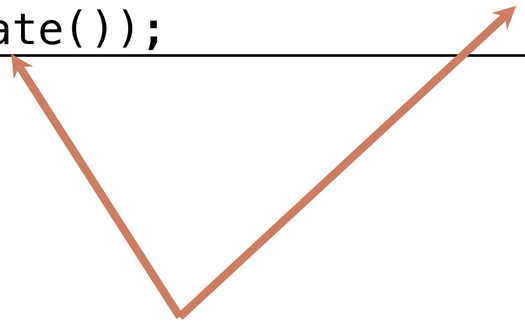
Parametric polymorphism - example

- This generic List type can then be instantiated for both **String** and **Date**

```
List<String> stringList = new ArrayList<String>();
stringList.add("Foo");

List<Date> dateList = new ArrayList<Date>();
dateList.add(new Date());
```

```
class List<E> {
    boolean add(E x) {…}
    E get(int index) {…}
}
```



Instantiating the
type parameter

Generic types and operations

- A type is called a generic type if it has a type parameter
  - For example **`ArrayList<E>`** is a generic type
- A type parameter is a placeholder for a specific type
  - The type parameter must be specified at the instantiation of a variable of the generic type
  - In the previous example, E is a type parameter
- Operations on generic types are called generic operations
  - In the previous example, **`add()`** and **`get()`** are generic operations

Instantiating a generic type

The generic type **`List<E>`** allows us to use lists with objects of arbitrary type

Example: instantiating a `List` of `Strings`

```
String s = "test";
List<String> a = new ArrayList<String>();
a.add(s);
String b = a.get(0);
```

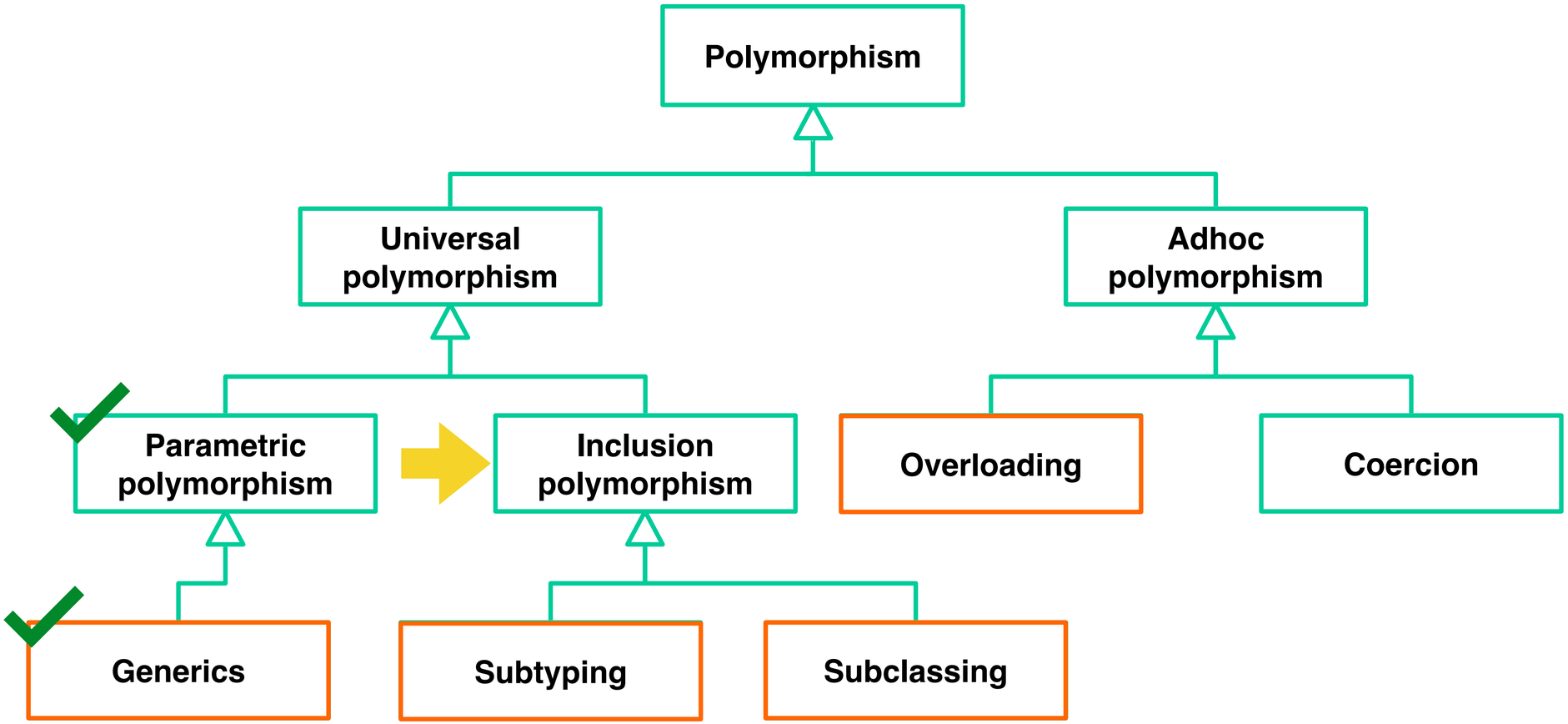
The compiler complains when using a different type:

```
a.add(new Date()); //COMPILE ERROR
```

Outline

- Course organization
- Patterns definition
- Basic concepts 1
- **Basic concepts 2**

Polymorphism: a taxonomy



Adapted from: Cardelli and Wegner, On Understanding Types, Data Abstraction, and Polymorphism. 1985

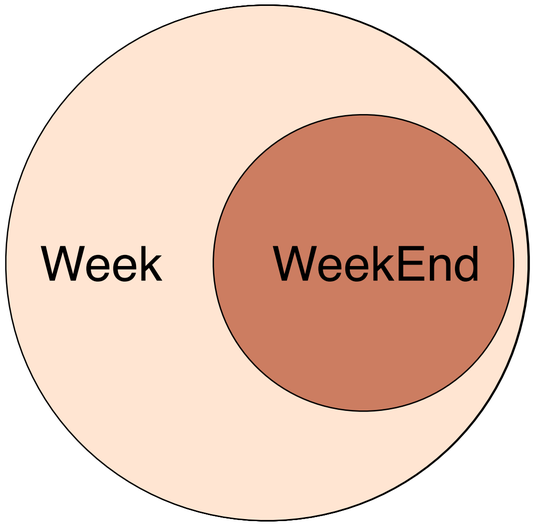
Inclusion polymorphism

1) Subtyping (based on the subset metaphor)
2) Subclassing (based on the inheritance metaphor)
   - Enables overriding of operations

Inclusion polymorphism: **subtyping**

Type A is a subtype of another type B, exactly when all A, considered as a set of values, is a subset of B.

[Adapted from Cardelli and Wegner 85]



Example:

Week = `{ Mon, Tue, Wed, Thu, Fri, Sat, Sun }`

WeekEnd = `{ Sat, Sun }`

WeekEnd is a subtype of Week:

$$\text{WeekEnd} \subseteq \text{Week}$$

Another **subtyping** example: mountain bike and bicycle

- Is every mountain bike a bicycle?
  - Yes! A mountain bike is a bicycle with an added suspension fork
- Mountain bike $\subseteq$ bicycle

Whenever a bicycle is used in a program, a mountain bike can be substituted without problems
(i.e. it behaves the way one expects)

This is called Liskov's substitution principle

Liskov's substitution principle (original version)

- Barbara Liskov, MIT
- Substitution principle first formulated in 1987 (OOPSLA '87)
- Turing Award 2008: for contributions to practical and theoretical foundations of programming language and system design, especially related to data abstraction, fault tolerance, and distributed computing

"If for each object O_S of type S there is an object O_T of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when O_S is substituted for O_T, then S is a subtype of T. "

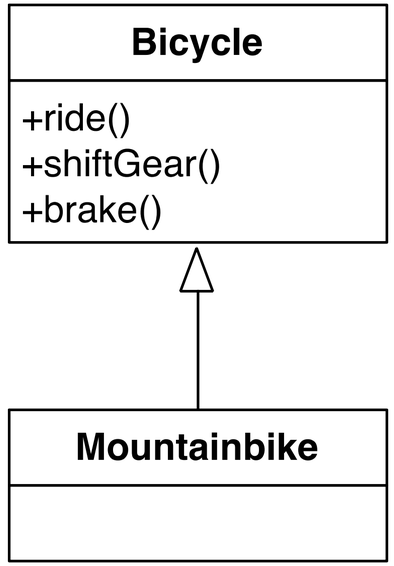
Liskov's substitution principle (simplified)

S is a subtype of T, if an object of type S can be substituted for an object of type T without changing the behavior of a program

- Example
  - We have a legacy system that uses a class T
  - While extending the system, we add a subclass of T called S
  - S has a few additional methods
  - If S can be used instead of T without changing the behavior of the system, S satisfies the Liskov Substitution Principle
  - E.g. any occurrence of bicycle (T) can be substituted by mountain bike (S)

Conformity with the **Liskov's substitution principle**

- We saw that mountain bikes can be substituted for bicycles with regards to the **Liskov's substitution principle**

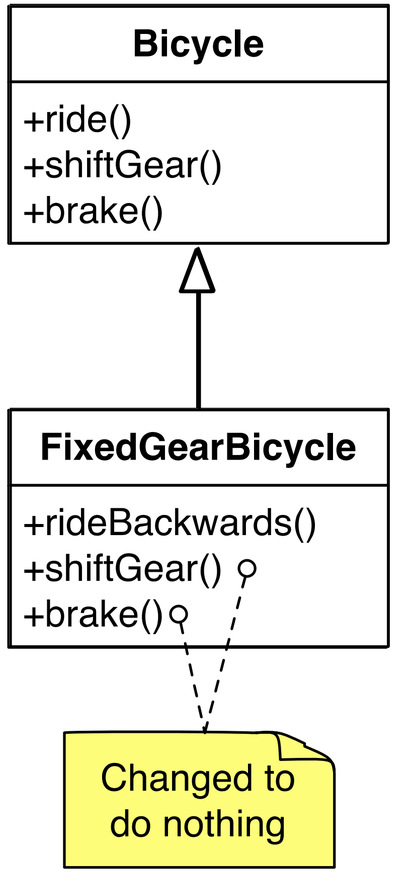


- Is every **`Mountainbike`** also a **`Bicycle`**?
- A **`Mountainbike`** basically is just a bike with added suspension
- In any context a **`Bicycle`** can be used in, a **`Mountainbike`** **can be substituted** without problems

The substitution principle is **fulfilled**!

Conformity with the **Liskov's substitution principle**

- What about fixies (fixed gear bicycles)?



- Is every **`FixedGearBicycle`** also a **`Bicycle`**?
- A **`FixedGearBicycle`** can be ridden backwards
- Any context which depends on **`brake()`** to stop the bike will fail with a **`FixedGearBicycle`**
- Any context which tries to shift gears will not work as intended
- In some contexts a **`Bicycle`** can be used in, a **`FixedGearBicycle`** **cannot** be substituted without problems

The substitution principle is **violated**!

Influence of **Liskov's substitution principle** on subtyping

- According to Liskov, a subtype must satisfy the substitution principle
- Therefore the following rules apply to subtypes
- **Operations**
  - All supertype operations must have corresponding subtype operations
  - Each subtype operation
    - Must have a **weaker precondition**: require less (or the same) than the superclass operation
    - Must have a **stronger postcondition**: guarantee more (or the same) than the superclass operation
- **Invariants** (called properties by Liskov)
  - Any invariant of the supertype (e.g. value constraints) must be guaranteed by the subtype as well

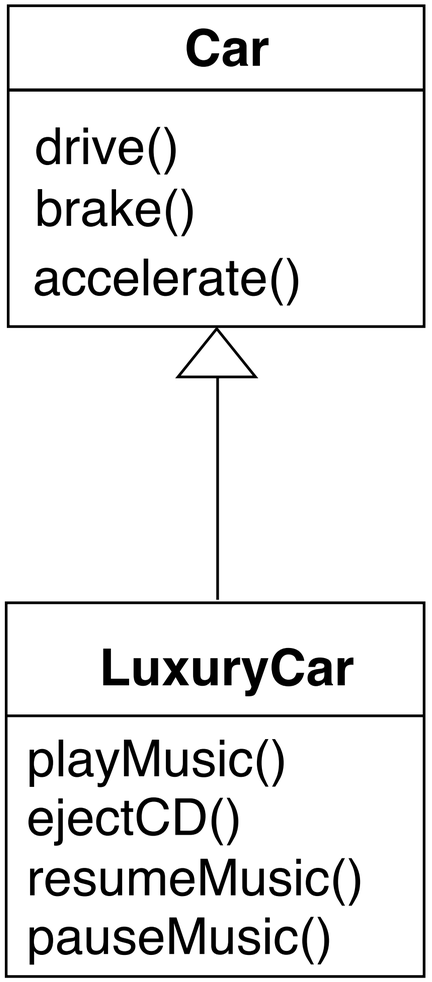
Subtyping in Java

- Interfaces and classes in Java do not guarantee any properties, therefore they are not subtypes according to Liskov's definition
- Therefore it is easy to construct subtypes in Java that do not follow the **Liskov substitution principle**
  - Example: the super type invariant requires elements to be sorted in ascending order while the subtype sorts the elements in descending order
- The responsibility for maintaining the **Liskov substitution principle** lies on the shoulders of the programmer

Inclusion polymorphism: subclassing

- Subclassing simply denotes the usage of an inheritance association
- Subclassing is mainly used
  - In **analysis**: formulate **taxonomies** (class hierarchies)
  - In **object design**: **reuse** implementations
- **Alternative:** delegation

Example for subclassing in a taxonomy



Superclass

```
public class Car {
    public void drive() { ... }
    public void brake() { ... }
    public void accelerate() { ... }
}
```

Subclass

```
public class LuxuryCar extends Car {
    public void playMusic() { ... }
    public void ejectCD() { ... }
    public void resumeMusic() { ... }
    public void pauseMusic() { ... }
}
```

Inclusion polymorphism: **overriding**

- **Overriding** is a special case in subclassing where a method implemented in the superclass is reimplemented in the subclass
  - This concept is at the heart of object oriented programming
  - It means, there are several methods, all with the same signature, but with different implementations
  - Which of these methods should be invoked?
    - The selection depends on the type of the object at runtime

- **Example for bad code**

```
class Bruce {
    public Emotion getEmotion() {
        if (isHulk) return Emotion.Anger;
        else return Emotion.Calmness;
    }
}
```

Inclusion polymorphism: **overriding**

- The selection of which method to execute at runtime allows to change the behavior of an object without extensive case distinctions

  - **Example for better code**

```
class Bruce extends Hero {
    public Emotion getEmotion() {
        return Emotion.Calmness;
    }
}

class Hulk extends Hero {
    public Emotion getEmotion() {
        return Emotion.Anger;
    }
}
```

Bad use of **overriding**

One can override the operations of a superclass with completely new meanings

Example:

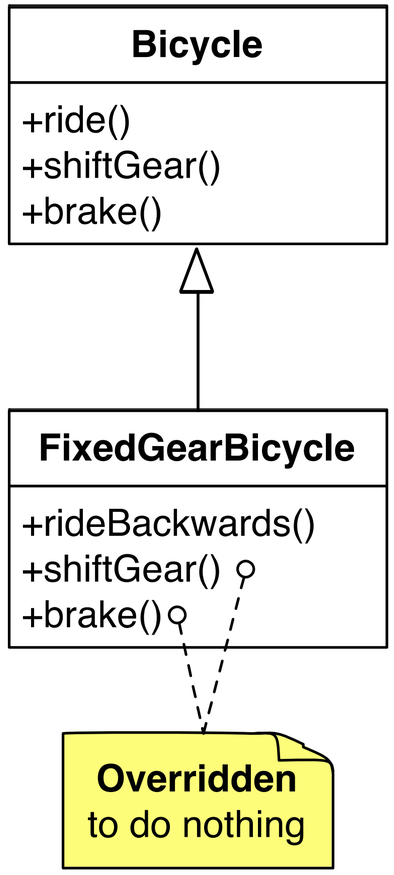
```
public class SuperClass {
    public int add (int a, int b) { return a + b; }
    public int subtract (int a, int b) { return a - b; }
}

public class SubClass extends SuperClass {
    public int add (int a, int b) { return a - b; }
    public int subtract (int a, int b) { return a + b; }
}
```

→ **We have redefined addition as subtraction and subtraction as addition!**

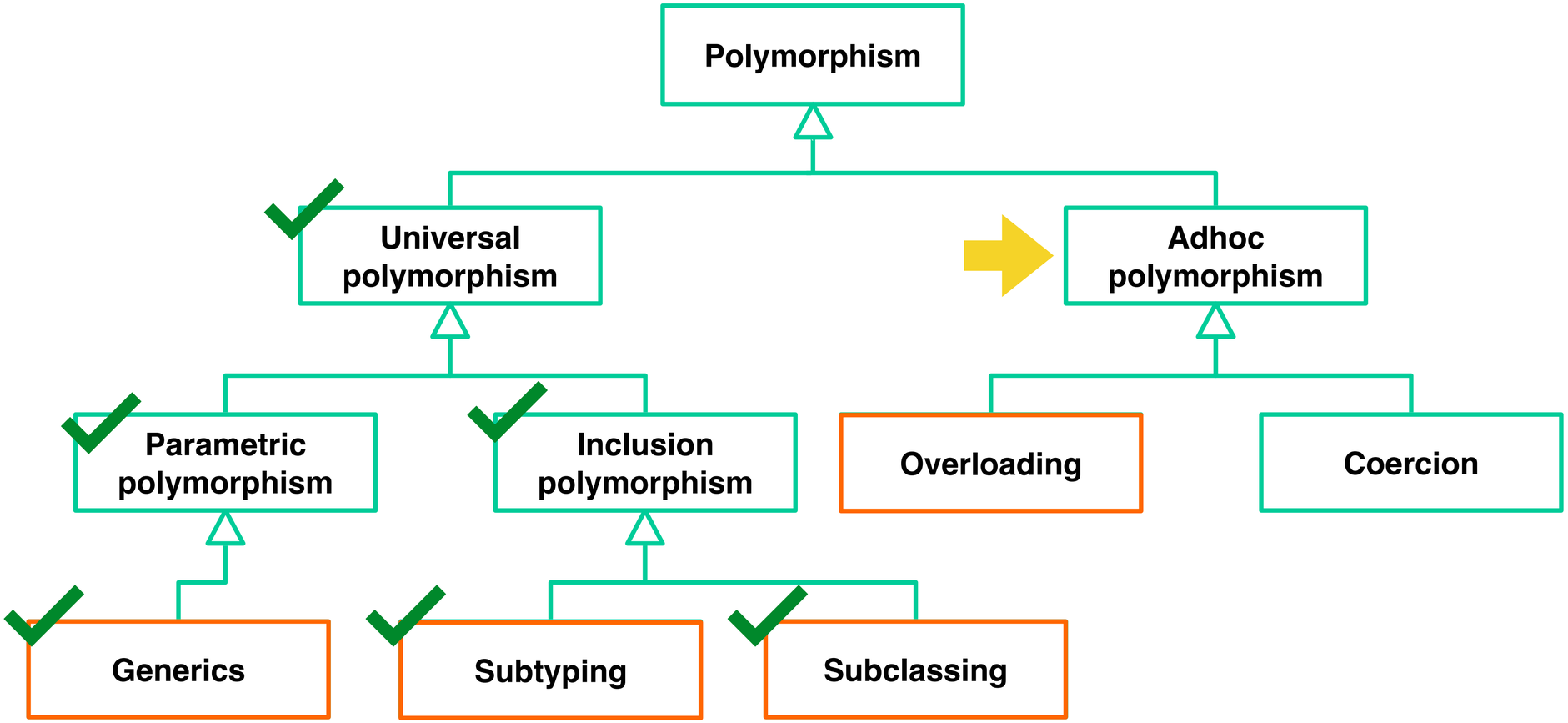
Recall the **FixedGearBicycle** example

- Also a **bad** use of overriding
- **Never override a method to do nothing at all (violates Liskov)!**



- Is a **`FixedGearBicycle`** a subclass of **`Bicycle`**?
  - Yes
- Does it follow the **Liskov substitution principle**?
  - **No**, because any client who expects to be able to **`shiftGear()`** to change speeds, or depends on **`brake()`** to stop will experience unexpected behavior, namely nothing

Polymorphism: a taxonomy



Adapted from: Cardelli and Wegner, On Understanding Types, Data Abstraction, and Polymorphism. 1985

Adhoc polymorphism: **overloading**

- There are four signatures for **`max()`** in the **`java.lang.Math`** class:
  - `public static int max(int a, int b)`
  - `public static long max(long a, long b)`
  - `public static float max(float a, float b)`
  - `public static double max(double a, double b)`
- As a programmer, you can just invoke ***`max()`***, e.g.:
  - `Math.max(1, 2);`
- The compiler then selects the appropriate overloaded method

Definition of **overloading**

- Overloading: the ability to let a feature name denote two or more operations [Meyer97]
- The signature is used to decide which of the possible operations is meant by a particular instance of that name
  - Signature: parameter types and return result type of a method
- The compiler makes the decision

Review of Java terminology

- **Overriding**: dynamic dispatch of instance method

```
class A {
    public int foo() { ... }
}

class B extends A {
    public int foo() { ... } // overrides A.foo()
}
```

- **Overloading**: methods with the **same name** but **different signatures** – binding is done at compile time

```
class A {
    public int foo(int i) { ... }   //int overloading, called for foo(2);
    public long foo(long i) { ... } //long overloading, called for foo(2L);
}
```

Binding

- **Binding:** establishes the mappings between names and data objects and their descriptions
- **Early binding** (static binding, at compile time)
  - The premature choice of operation variant, resulting in possibly wrong results and (in favorable cases) run-time system crashes
- **Late binding** (dynamic binding, at run time)
  - The guarantee that every execution of an operation will select the correct version of the operation, based on the type of the operation’s target
- "Typing addresses the existence of at least one operation; binding addresses the choice of the right one among these operations, if there is more than one candidate." [Meyer97]

Early binding – Java example

Class methods are selected at compile time

```
class B {
    public static String foo() { return "foo"; }
}

class A extends B {
    public static String foo() { return "bar"; }
}

public class C {
    public static void main(String[] args) {
        B o1 = new B();
        B o2 = new A();
        A o3 = new A();
        System.out.println(o1.foo());
        System.out.println(o2.foo());
        System.out.println(o3.foo());
    }
}
```

```
Results in:
"foo"
"foo"
"bar"
```

Late binding – Java example

Instance methods are selected at run time

```
class B {
    public String foo() { return "foo"; }
}

class A extends B {
    public String foo() { return "bar": }
}

public class C {
    public static void main(String[] args) {
        B o1 = new B();
        B o2 = new A();
        A o3 = new A();
        System.out.println(o1.foo());
        System.out.println(o2.foo());
        System.out.println(o3.foo());
    }
}
```

```
Results in:
"foo"
"bar"
"bar"
```

L01E03 Quiz

Start quiz

Live Easy

Not started yet.

Due date in 5 minutes

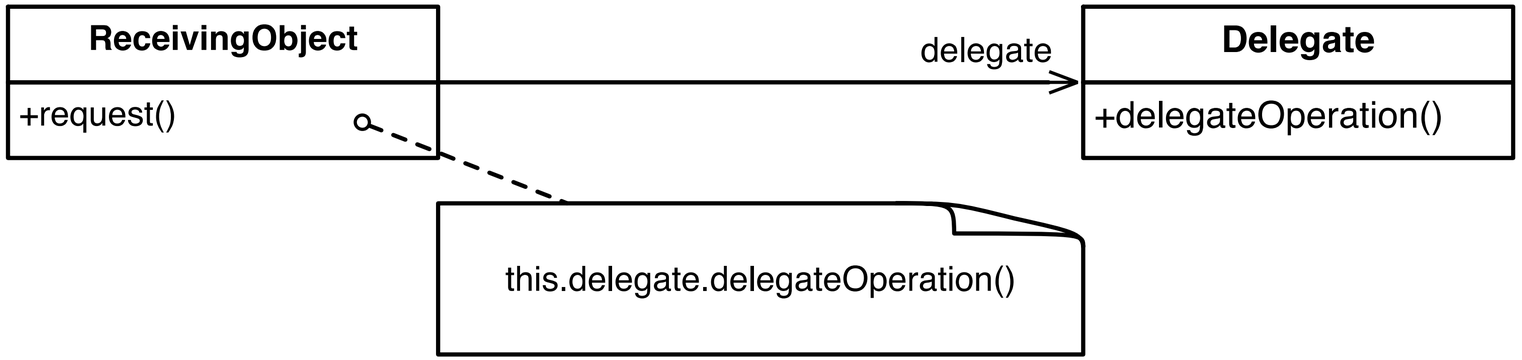
5 min

3 pts

- Open the live quiz in Artemis now
- Participate in the quiz
- Submit your answer within the due date

Delegation

- The **`ReceivingObject`** delegates a request to perform an operation to its **`Delegate`**

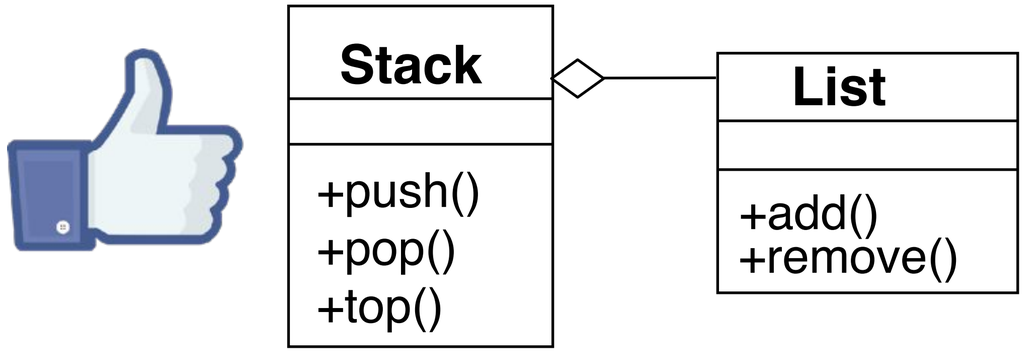


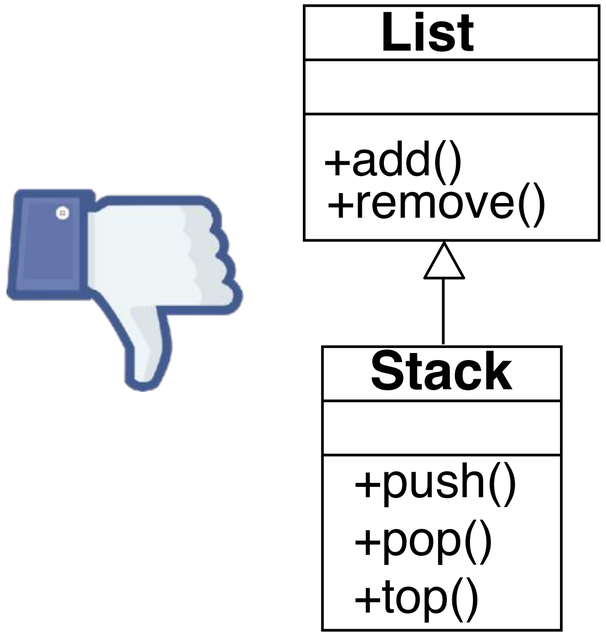
Delegation

- Involves passing a method call to another object, transforming the input if necessary
- Extends the behavior of an object
- A simple mechanism for **code reuse** in which an operation resends a message to another class to accomplish the desired behavior

Delegation vs. subclassing

- **Main goal: prevent code duplication**
- Delegation: a class catches an operation and sends it to another class
- Subclassing: a subclass extends a super class with new operations or overrides existing operations (inheritance)
- What is better?





Summary

- Patterns are a great way to describe **reusable knowledge** for **recurring design problems**
  - Antipatterns: useful for describing lessons learned
- We use **interactive learning** to teach you different kind of patterns
- Object oriented programming, UML and Java are foundations for this course
- The theoretical foundations of the concepts presented in this lecture come from compiler and programming language design
- For patterns, it is important to understand these underlying concepts
  - Like an architect needs to know building materials, you should know about compilers and programming languages and how they support patterns
  - **Polymorphism** and **delegation** are basic building blocks of design patterns and architectural patterns

Literature

- Bruegge & Dutoit: Object-oriented SE, 3rd edition 2009, Pearson Education
- Christopher Alexander et al: A Pattern Language: Towns, Buildings, Construction, Oxford University Press, 1977
- Christopher Alexander et al: The Timeless Way of Building, Oxford University Press, 1979
- Martin Fowler: Analysis Patterns Reusable Object Models, Addison Wesley, 1996
- Frank Buschmann et al: Pattern-Oriented Software Architecture, Vol 1: A System of Patterns, Wiley, 1996
- Eric Gamma et al: Design patterns, Addison Wesley, 1995
- Elisabeth Freeman et al: Head First Design Patterns, O'Reilly 2004
- Jon Thomas et al: Java Testing Patterns, Wiley 2004
- Scott Ambler: Process Patterns, Cambridge University Press 1998
- William J. Brown et al: Anti-Patterns and Patterns in Software Configuration Management, Wiley 1999
- Amr Elssamadisy: Agile Adoption Patterns, Addison-Wesley 2008
- Richard P. Gabriel: Patterns of Software: Tales from the Software Community, Oxford University Press 1998

Literature

- David Parnas: On the criteria to be used in decomposing systems into modules, Communications of the ACM archive, Volume 15, Issue 12 (December 1972), Pages 1053 - 1058
- Barbara Liskov: Data Abstraction and Hierarchy, Keynote address OOPSLA '87, ACM SIGPLAN Notices, Volume 23, Issue 5, May 1988.
- Luca Cardelli and Peter Wegner: On Understanding Types, Data Abstraction, and Polymorphism. Computing Surveys of the ACM, Vol.17, No.4, p.472-522, ACM, December 1985
- Joshua Bloch and Neal Gafter: Java(TM) Puzzlers: Traps, Pitfalls, and Corner Cases, Addison-Wesley Professional, 2005.
- Bertrand Meyer: Object-oriented Software Construction, Prentice Hall International, 1997.
- C. Strachey: Fundamental Concepts in Programming Languages. Higher Order Symbolic Computing, vol. 13, 2000, pp. 11-49.
- P. Wegner: Dimensions of Object-Based Language Design. Proceedings OOPSLA 87, pp. 168-182, ACM, New York, 1987
- William R. Cook: On Understanding Data Abstraction, Revisited. Proceedings OOPSLA 2009, ACM, New York, 2009
- Z. Splawski: Polymorphism - Prose of Java Programmers. Annales UMCS Informatica AI 4, pp. 291-303, 2006
- S. Devadas and D. Jackson: Lecture 14 of MIT "6.170 Laboratory in Software Engineering" taught in Fall 2005
- H. Lieberman: Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems. Proceedings OOPSLA 86, pp. 214-223, ACM, New York, 1986
- L. A. Stein: Delegation Is Inheritance. Proceedings OOPSLA 87, pp. 138-146, ACM, New York, 1987.

Online readings

- C. Alexander: A City is not a Tree, 1965: http://www.patternlanguage.com/leveltwo/archivesframe.htm?/leveltwo/../archives/alexander1.htm
- Brad Appleton, Patterns and Software: Essential Concepts and Terminology, 2000: http://www.bradapp.com/docs/patterns-intro.html
- Richard Gabriel, Patterns of Software: Tales from the community, Oxford Press, 1996, online version: http://www.dreamsongs.com/Files/PatternsOfSoftware.pdf (Great foreword by Christopher Alexander)