Software Engineering 2
– Design Patterns. Part 1

Hochschule
für Technik
Stuttgart

Marcus Deininger

SS 2021

Summary

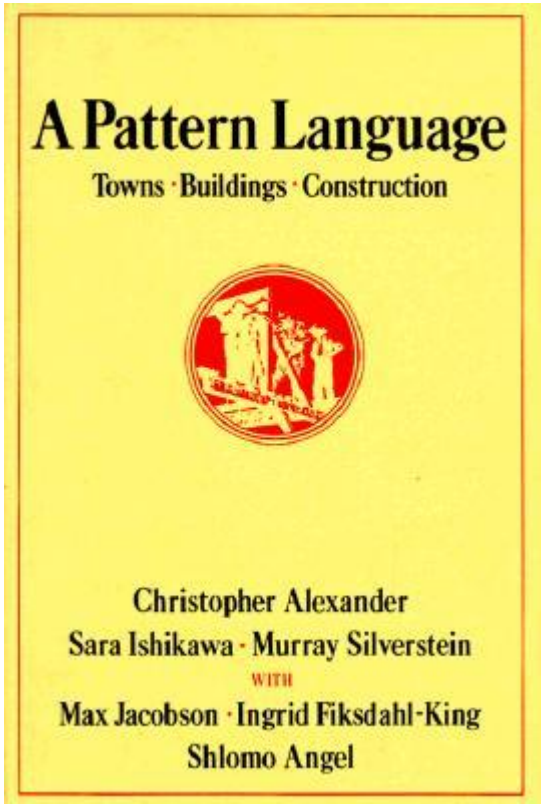
- Design Patterns
  - Foundations
- UML for Patterns
- Selected Patterns Part 1
- Exercise Part 1

Design Patterns

Original Inspiration

- Christopher Alexander et al.: “A Pattern Language: Towns, Buildings, Construction”, a 1977 book on architecture.
- Contains 253 patterns which solve typical architectural problems – together they form a “pattern language”

1985/89, Higashino Eishin Campus,
photograph by Robert Baum
https://www.flickr.com/photos/27392505@N00/



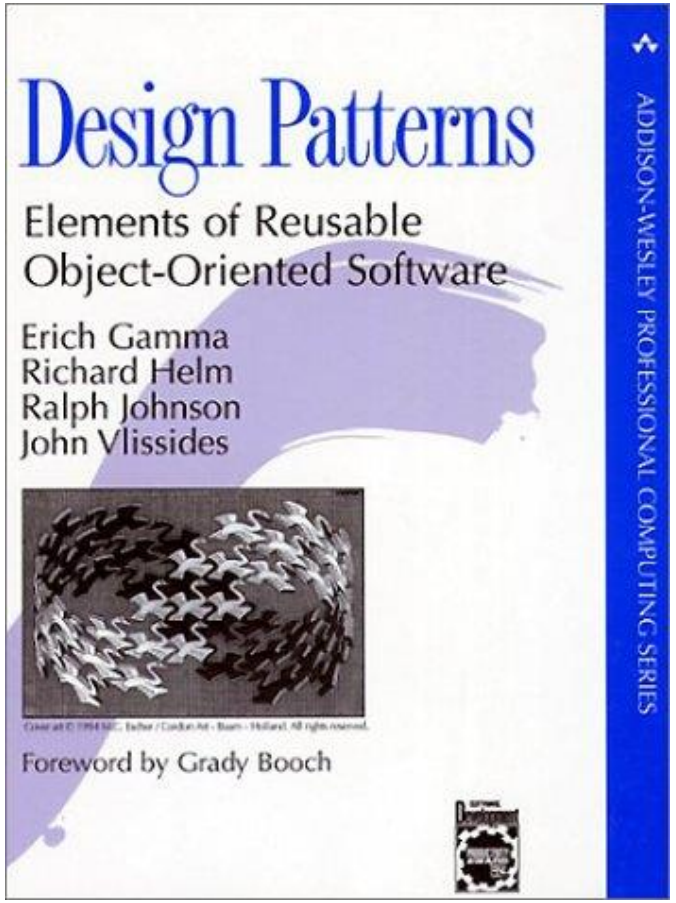
Design Patterns in Software-Development

Erich Gamma at ETH Zürich

- Development of an Editor-Toolkit (ET)
- Extension to the framework ET++ (with A. Weinand)
- Description of design solutions for selected problems (ph.d. thesis); called “Design Patterns” according to C. Alexander et al.

E. Gamma, J. Vlissides, R. Helm und R. Johnson

- *Design Patterns, Addison Wesley, 1996.*
- gives a classification for design patterns
- defines 23 design patterns
- “Design Pattern” is a technical term since then



A Better Book

Freeman, E., E. Freeman: Design Patterns. O'Reilly, 2004

- gives a selected, more readable introduction
- based on Java

Design Patterns in Software-Development

Level of abstraction

- *general* programming issues and *basic* solutions – independent of the programming language

→ applicable to all programming languages

Usage

- Usage in *Class Design*
- Definition of patterns of behavior and interaction between several classes
- Patterns for a special issues like EJB, .NET, database connect

Design Patterns – Definition

"A description of an object-oriented design technique which **names, abstracts** and **identifies aspects of a design structure** that are useful for creating an object-oriented design.

The design pattern identifies **classes** and **instances**, their **roles**, **collaborations** and **responsibilities**. Each design pattern focuses on a particular object-oriented design problem or issue. It describes when it applies, whether it can be applied in the presence of other design constraints, and the consequences and trade-offs of its use."

E. Gamma et al (1995): Design Patterns, Addison-Wesley, ISBN 0-201-63361-2

Design Patterns – Characteristics

Design Patterns

- represent a well-known and documented design experience which can be reused
- introduce an abstraction on the level of micro-architecture (that's on top of singular classes)
- may be combined freely
- back up the development and documentation of complex and heterogeneous software designs
- facilitate changeability and reusability
- are a vocabulary of design thus facilitating design
- may be used for reengineering too

Design Patterns – Foundations

Object-Oriented Programming

- Inheritance / Interfaces
  - General / abstract definition of behavior
- Polymorphism
  - Instances may be assigned to any compatible variable (i.e. of a superclass or implemented interface)
- Dynamic Binding
  - The actual executing methods are defined during runtime.

Design Patterns – Foundations. Example

```
public interface C {

     public void info();

}
```

All implementers of C guarantee to have **info()**.

```
public class C1 implements C {

     public void info() {
          System.out.println("C1");
     }
}
```

```
public class C2 implements C {

     public void info() {
          System.out.println("C2");
     }
}
```

Implementers of **C** have to provide **info()**.

```
public class Main {

     public static void main(String[] args) {
          C c;
          if(Math.random() <= 0.5)
               c = new C1();
          else
               c = new C2();
          c.info();
     }

}
```

Polymorphism: **c** may be assigned to **C1** or **C2**-objects.

During runtime an object is created randomly (either of C1 or C2)

Dynamic binding: Thus, only at runtime it is possible to decide, which implementation of **info()** is to be used.

Hochschule für Technik Stuttgart

Design Patterns – Foundations. Example

```
public class C1 {

      public void info(){
            this.print ();
      }

      public void print() {
            System.out.println("C1");
      }
}
```

All subclasses of C1 inherit **`info()`**.

Call of the current implementation of **`print()`**.

```
public class C2 extends C1 {

      public void print() {
            System.out.println("C2");
      }
}
```

Overrides the implementation of **`print()`**.

```
public class Main {

      public static void main(String[] args)
            C1 c;
            if(Math.random() <= 0.5)
                  c = new C1();
            else
                  c = new C2();
            c.info();
      }
}
```

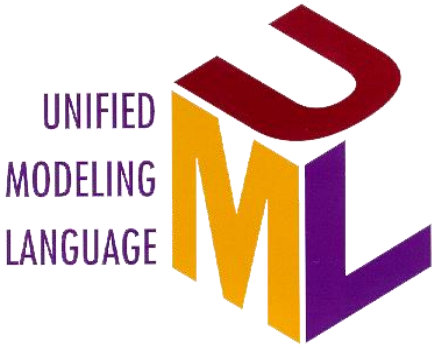
During runtime an object is created randomly (either of C1 or C2)

Polymorphism: **`c`** may be assigned to **`C1`** or **`C2`**-objects.

Dynamic binding: Again, only at runtime it is possible to decide, which implementation of **`print()`** is to be used.

Hochschule für Technik Stuttgart

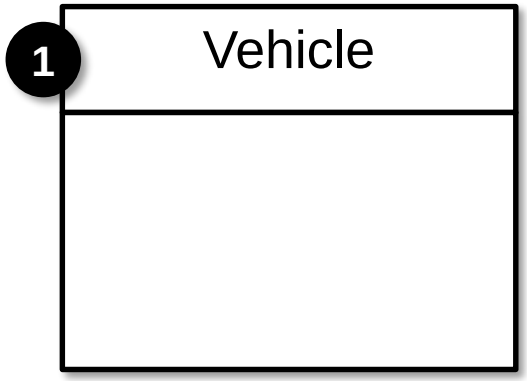
Short Retake: Unified Modelling Language (UML)



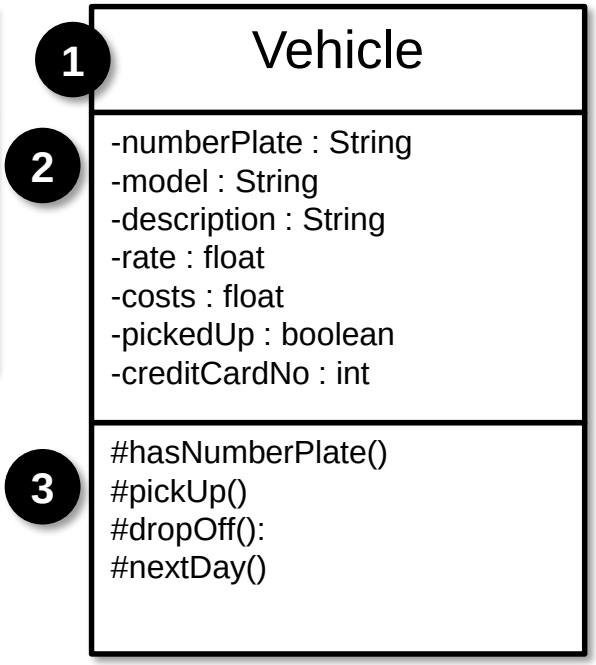
- Graphic notation for design and description of software systems
- Originally defined by Booch, Rumbaugh, Jacobson, 1995
- Development through the Object Management Group (OMG, www.uml.org)
- UML offers numerous Chart types to software systems graphically describe
- The most important are the package- and class diagrams: they show the classes and their relationships – sometimes additionally activity or sequence diagrams

Class Diagrams

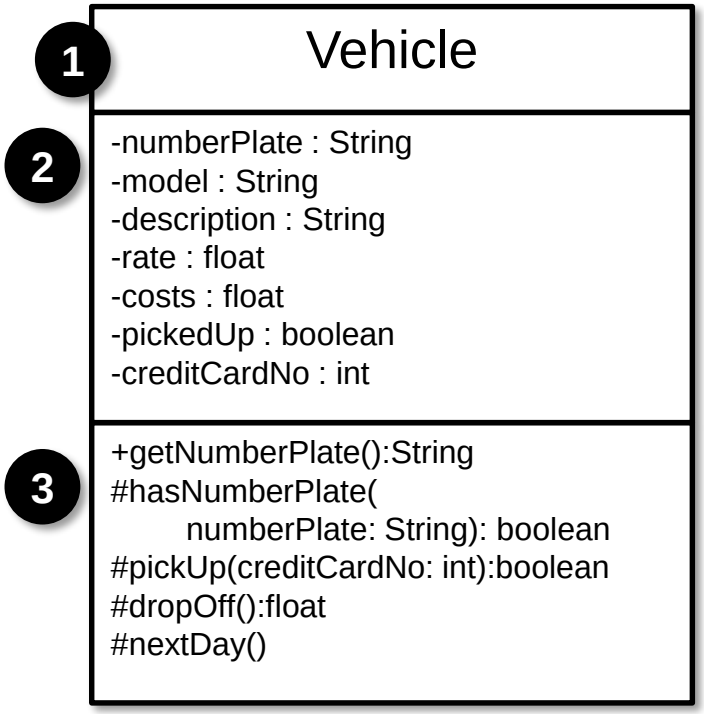
1. The name of the class
2. The fields of the class; -**private**, ~**default**, #**protected**, +**public**; static elements are underlined.
3. The methods of the class. Parameters, constructors, getters, setters may be left out.



Simple form

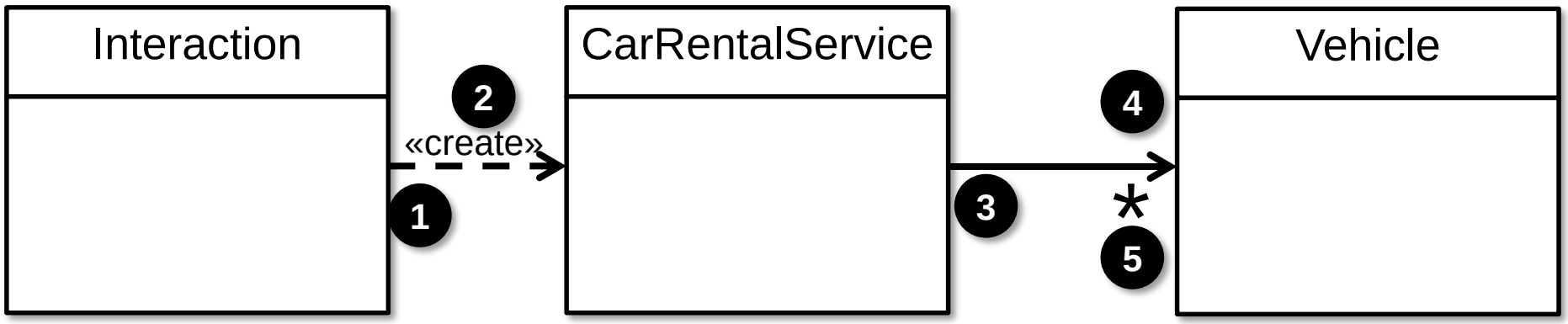


Simplified form with reduced method signatures



Complete form

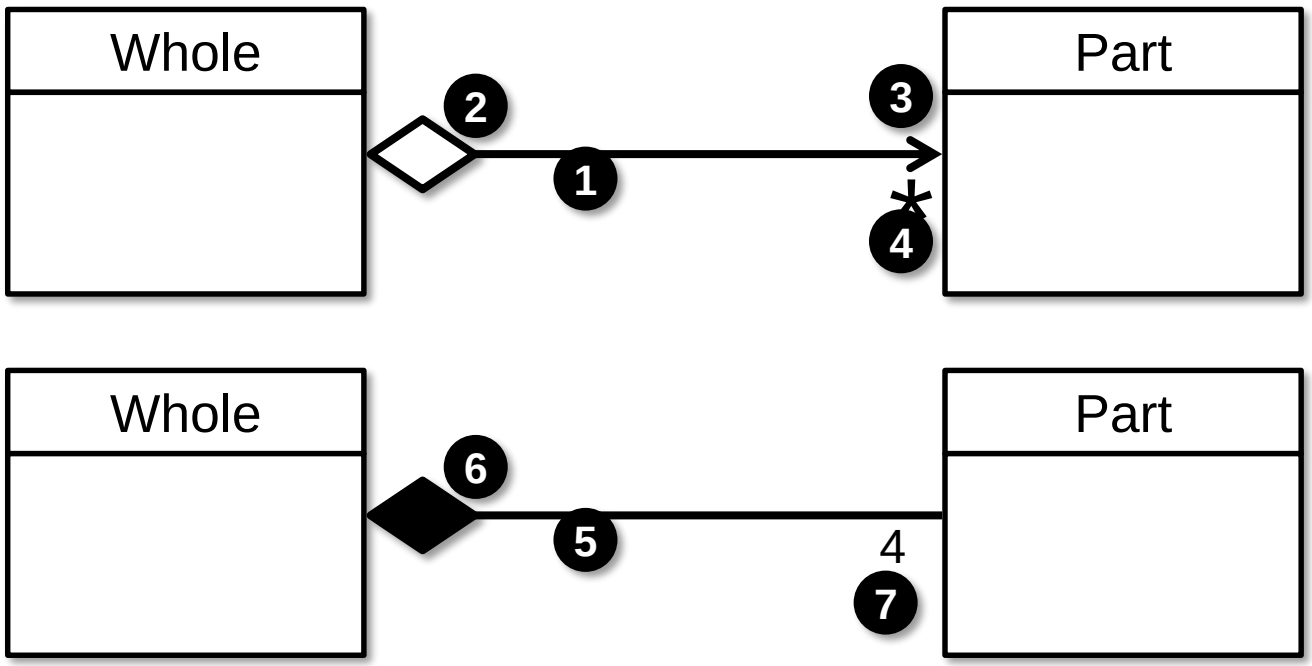
Relationships between Classes



1. A **general dependency**; Interaction the class CarRentalService
2. A **stereotype** explaining the dependency
3. A **membership-dependency**; CarRentalService has an instance variable (“field”) of Vehicle-Objects; additionally a role name could be added – if missing the role name equals the class name, i.e. `vehicle(s)`.
4. **Unidirectional dependency**; i.e. CarRentalService has a Vehicle-Field, Vehicle has no reference back.
5. **Multiplicity**; default (when missing) 1; * many; any other number is possible.

Attention: the resulting code-relationships are not between the classes but actually between the (future) objects derived from these classes!

The associations between the classes: "Is-part-of" relationship

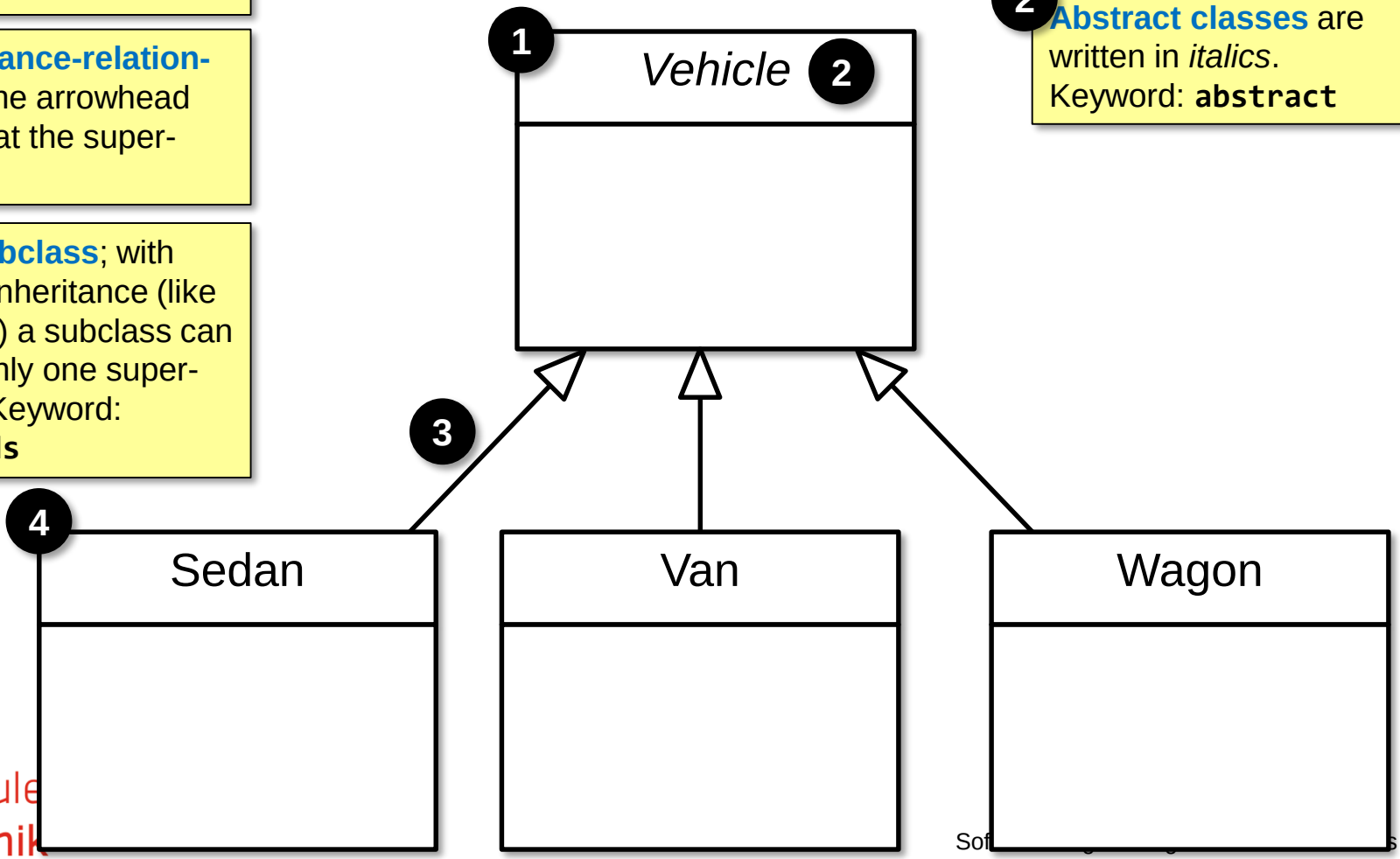


1. An **association**,
2. An **aggregation**: one object is part of another.
3. **Unilateral relationship** – a two-way relationship is not possible.
4. **multiplicity**
5. An **association**,
6. A **composition**: an object is **exclusively** part of another.
7. **multiplicity**: the whole consists of four parts.

Inheritance

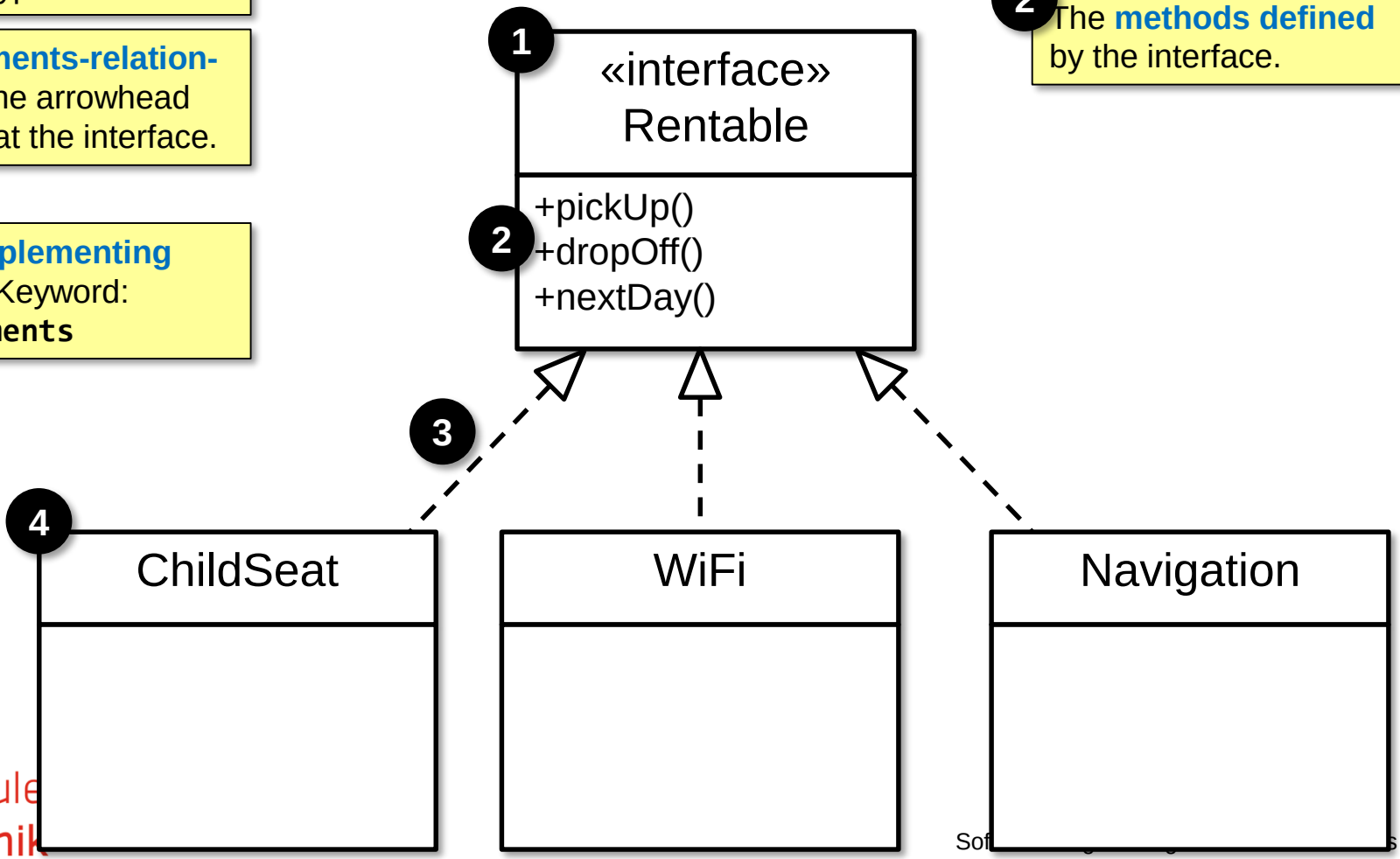
1. The “**superclass**”; this class inherits to its “subclasses”.
3. **Inheritance-relation-ship**; the arrowhead points at the super-class.
4. The **subclass**; with single inheritance (like in Java) a subclass can have only one super-class. Keyword: **`extends`**

2. **Abstract classes** are written in *italics*. Keyword: **`abstract`**



Interface

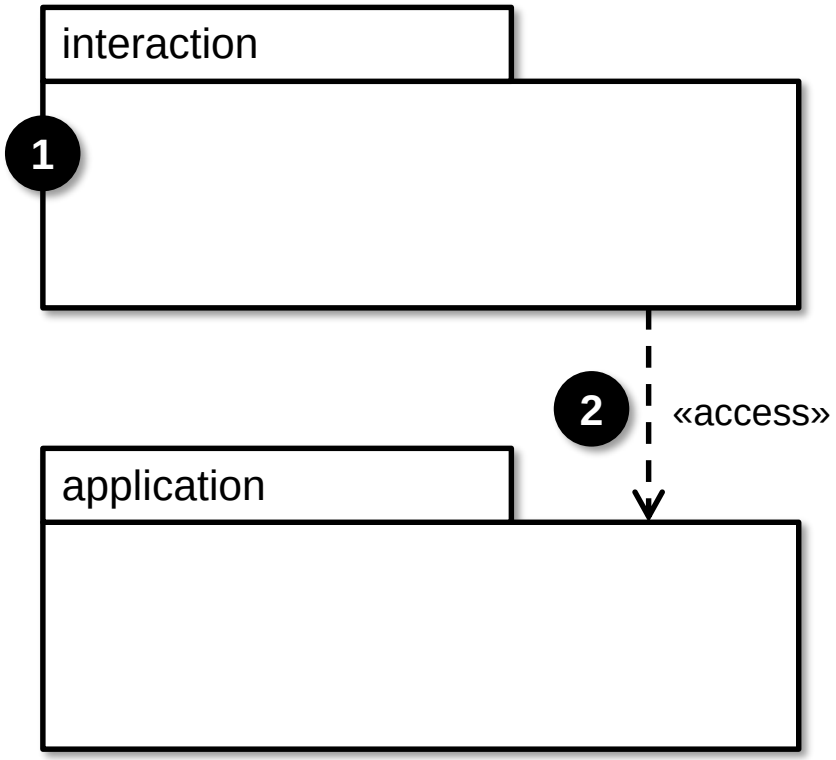
1. The **interface** name, denoted by an stereotype.
2. The **methods defined** by the interface.
3. **Implements-relation-ship**; the arrowhead points at the interface.
4. The **implementing class**. Keyword: **`implements`**



Package Diagrams – 1

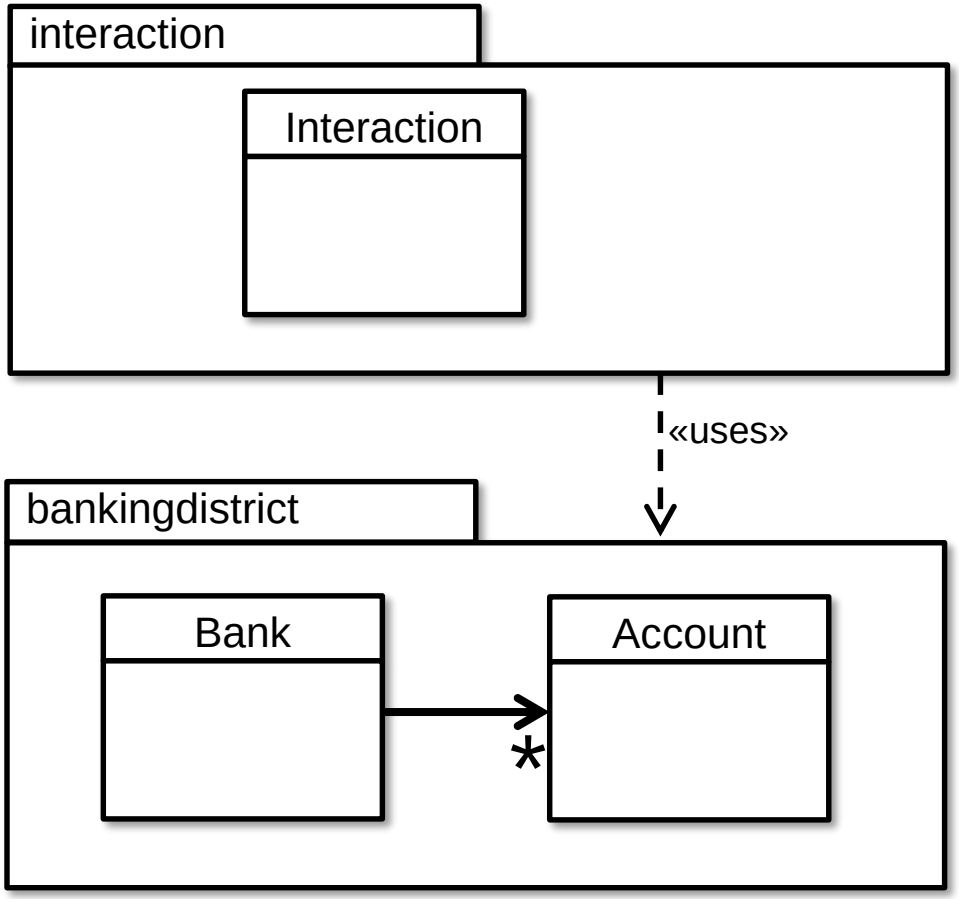
1. The **package**; it has a **name**. At the highest design level, the package is usually left blank. In later design stages sub-packages or classes will be inserted.

2. The package “interaction" **depends on (public) elements** from the package "application". The arrowhead indicates the direction of use. The **stereotype** «access» gives the type of usage.



Package Diagrams – 2

Packages may **contain** other packages or classes.



Usage of UML

- The ultimate goal is a clear representation
- Elements that are evident from the context or not relevant at this level of abstraction, can be omitted.

From UML to Java Code – 1

UML element ...	**... becomes in Java**
package	Java package
class	Java class (in the package)
field	instance or class variable (when underlined)
method	Java method
underlined, -, ~, #, +	**`static`**, **`private`**, *(No keyword "package", "default")*, **`protected`**, **`public`**
inheritance relationship	**`extends`**
interface relationship	**`implements`**

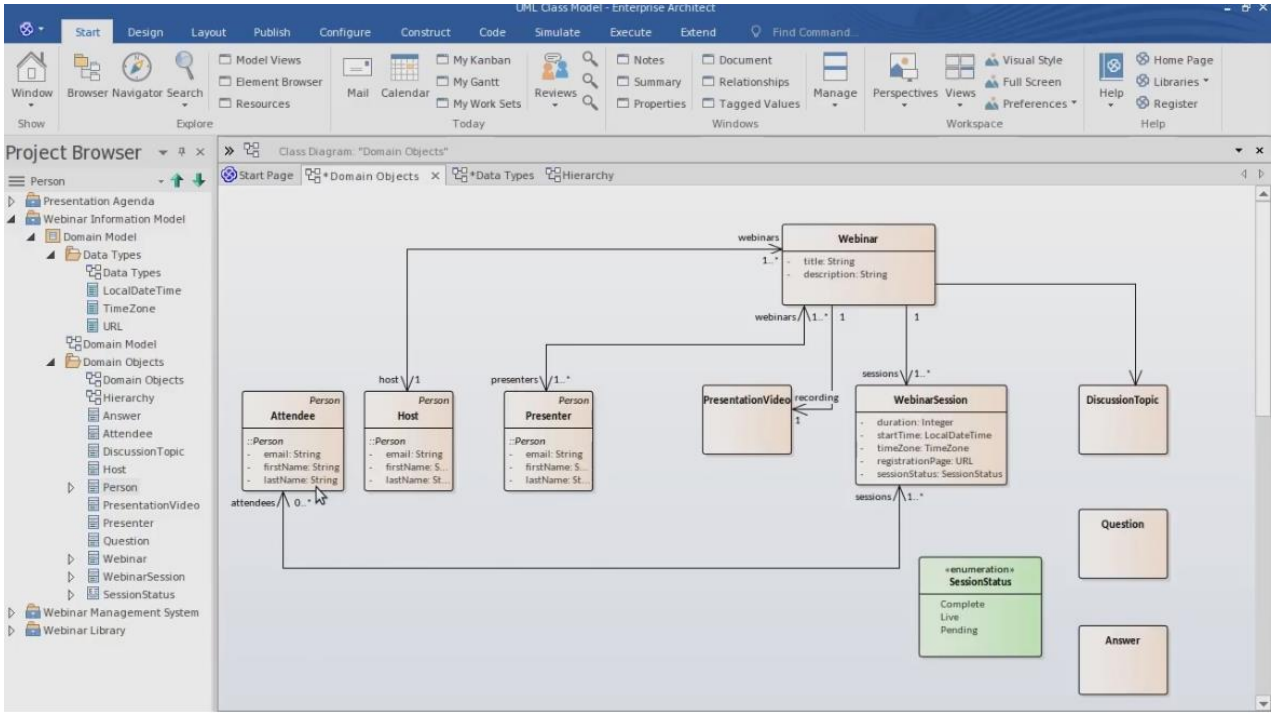
From UML to Java Code – 2

UML element ...	... becomes in Java
Membership dependency	
■ arrow end	"Host" class of the field, named after the role or the target class pointing at
■ arrow end	type of field (the target class of the arrow)
■ multiplicity 0 *or* 1	a field (with the name of the role or the name of the variable class, if the role is absent) – attention: this variable does not appear as an attribute of the class!
■ multiplicity *number n*	array of size n; possibly several variables
■ multiplicity *	array or collection
General dependency	usage of the class according to the stereotype
"Is-part-of"- relationship	implemented like membership dependency

From UML to Java Code – Code Generation

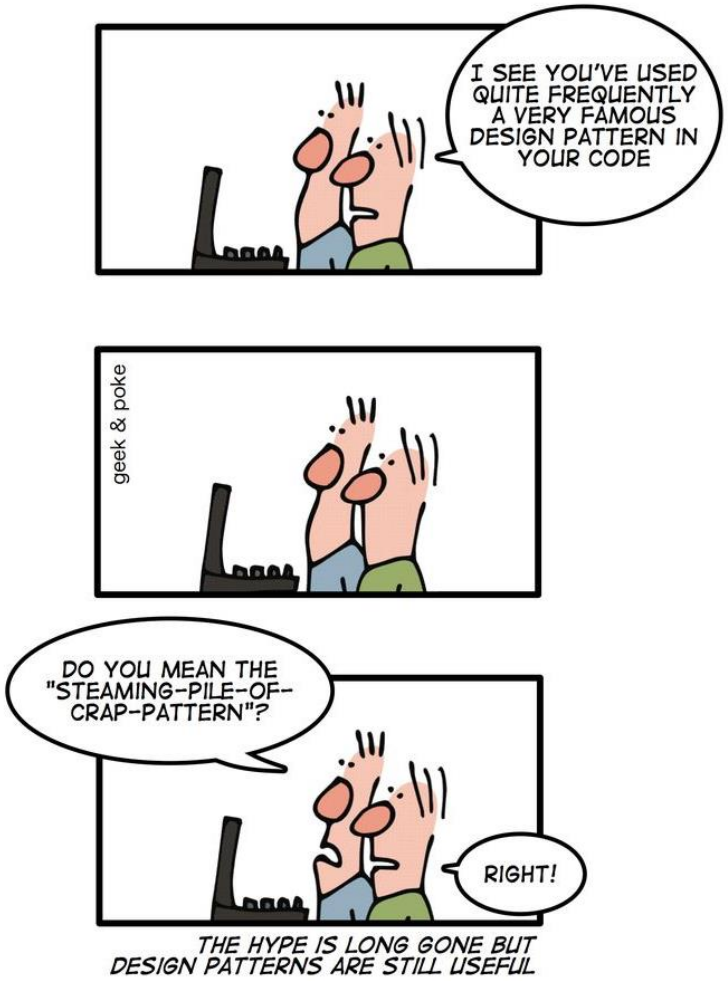
- "Paint programs" allow to draw UML diagrams
- from these drawings Java program-stubs are generated

→ graphical design, from the code is derived



Example: Enterprise Architect

Back to Design Patterns



http://geek-and-poke.com

Design Patterns (Gamma et al.)

Purpose / Scope	**Creational** Creating objects	**Structural** Building complex objects	**Behavioral** Accomplishing complex tasks
Class uses inheritance	Factory-Method	Adapter	Interpreter Template Method
Object uses associations	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Creational Patterns

Encapsulates / organizes the creation of objects

- Factory Method: Subclasses decide which concrete classes to create.
- Abstract Factory: Allows a client to create families of objects without specifying their concrete classes.
- Builder: Encapsulates the construction of a product and allows it to be constructed in steps.
- Singleton: Ensures one and only object is created.

Structural Patterns

Define a relationship between entities.

- Adapter: Wraps an object and provides a different interface to it.
- Bridge: Separates abstraction and implementation in different, corresponding hierarchies.
- Composite: A clients treat collections of objects and individual objects uniformly.
- Decorator: Wraps an object to provide new behavior.
- Façade: Simplifies the interface of a set of classes.
- Flyweight: Replaces many instances which differ only in their state (if at all) by one.
- Proxy: Wraps an object to control access to it.

Selected Behavioral Patterns

Define communication between entities.

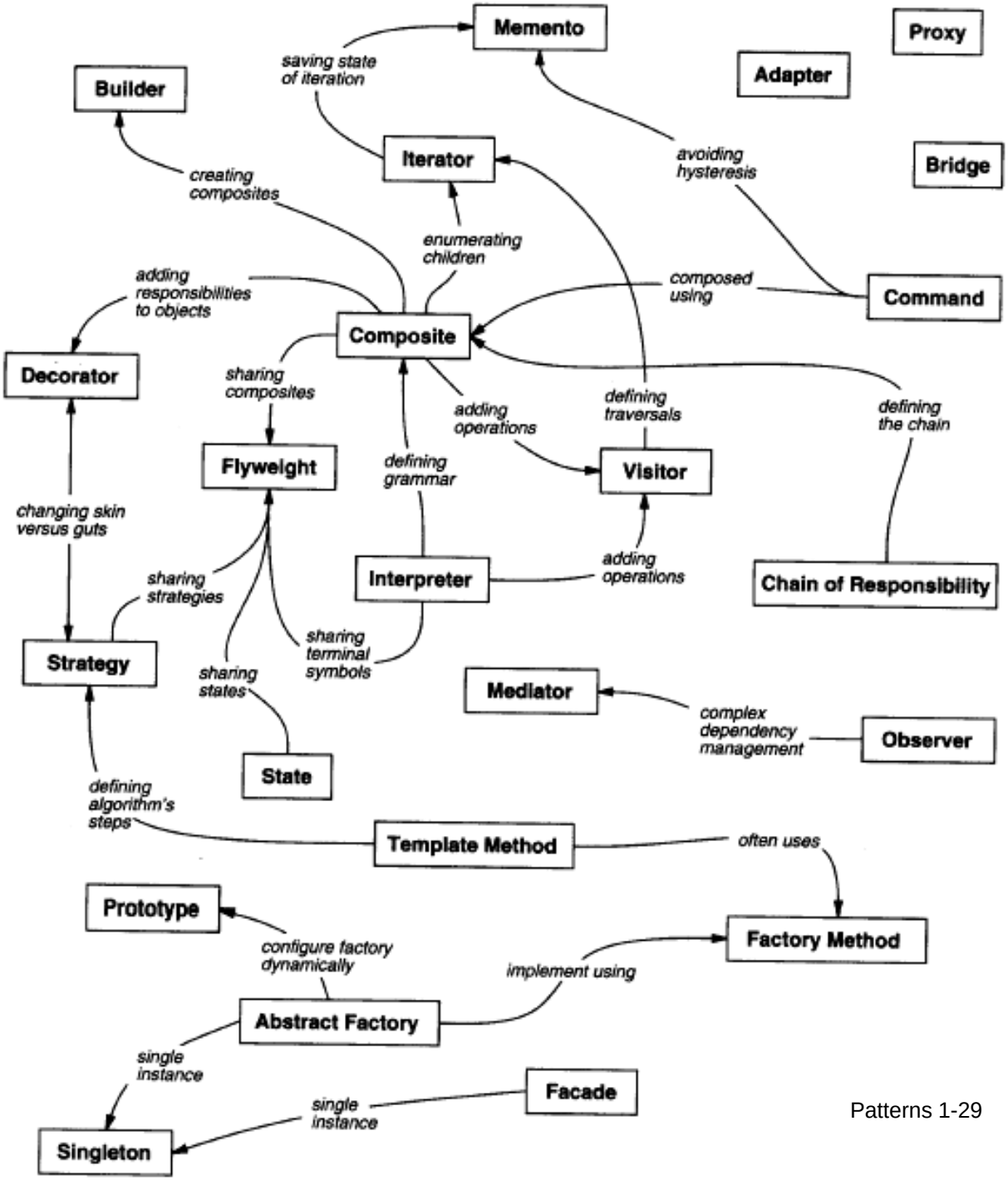
- Template Method: Subclasses decide how to implement steps in an algorithm.
- Chain of Responsibility: More than one object gets a chance to handle a request.
- Command: Encapsulates a request as an object.
- Iterator*: Provides a way to traverse a collection of objects without exposing its implementation.
- Observer*: Allows objects to be notified when state changes.
- State: Encapsulates state-based behaviors and uses delegation to switch between behaviors.
- Strategy: Encapsulates interchangeable behaviors and uses delegation to decide which one to use.

** Part of Java.*

Design Patterns – Relationships

The Design Patterns of Gamma et al. are highly related:

- Design Patterns may use each other
- Design Patterns use common components



Hochschule für Technik Stuttgart

Selected Go4 Patterns (Part 1)

Structural Pattern: Adapter

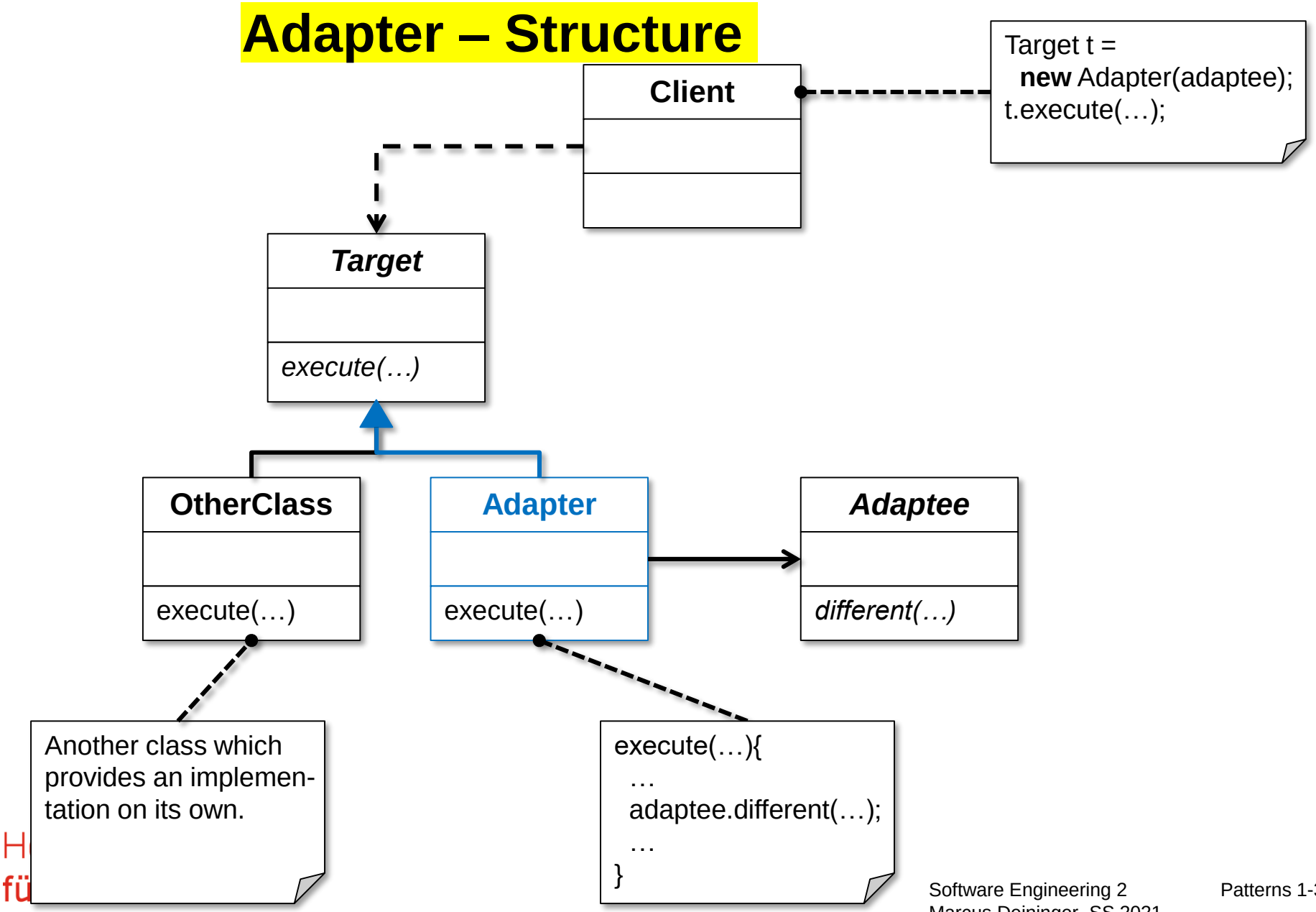
Situation

- One class ought to use another class – but the interfaces don't fit
- This mainly happens when combining or reusing different classes or class libraries

Possible Approaches

- Change the source code (if available)
- Make a subclass of the class to be used and redefine interfaces

Adapter – Structure

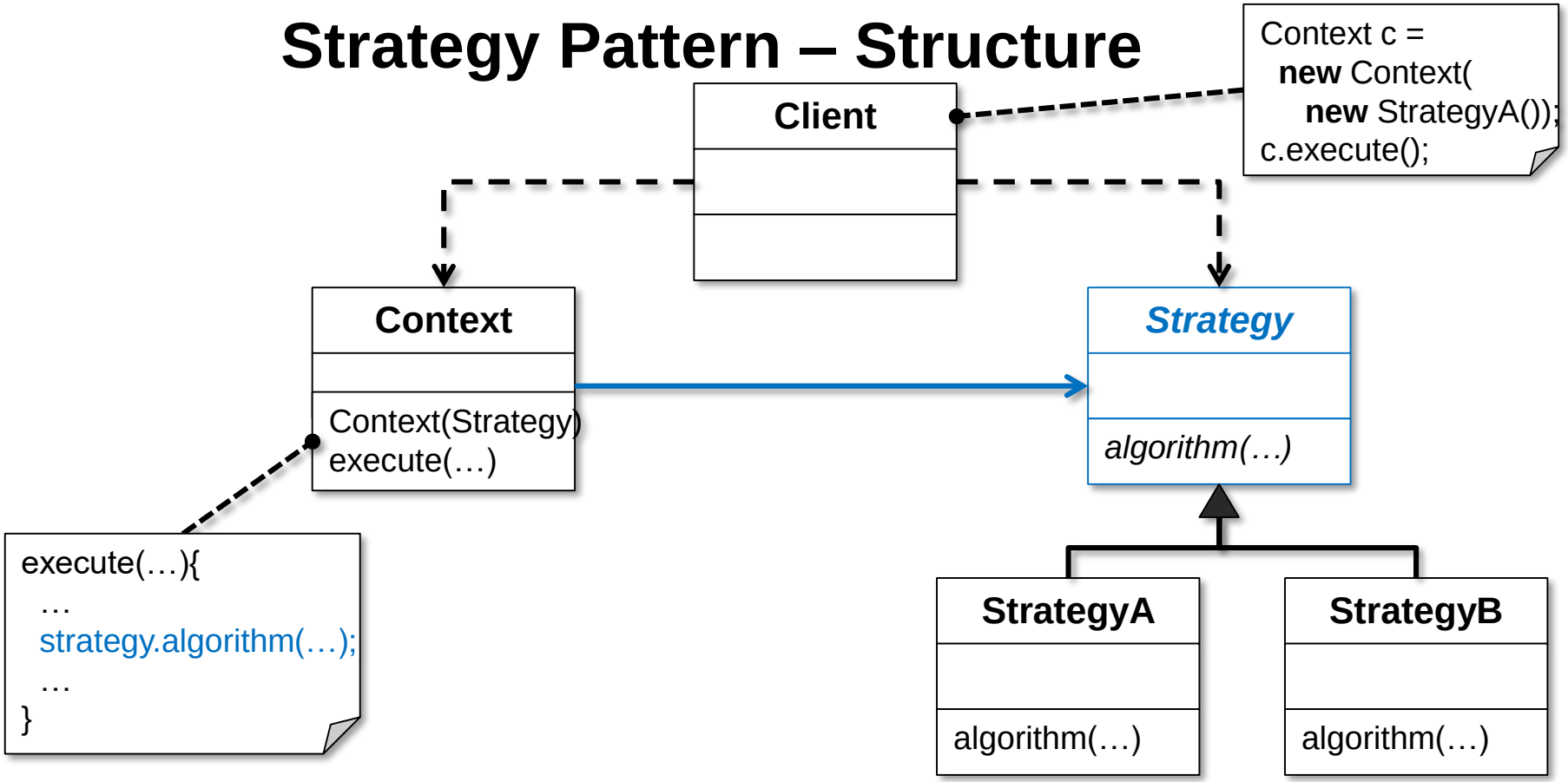


Behavioral Pattern: Strategy

The behavior of a class is separated into a strategy class

- classes may be configured with different behavior
- Different variants of an algorithm may be injected
- The algorithm should be hidden in a separate class
- The behavior of a class may change at runtime (probably triggered from outside the class)

Strategy Pattern – Structure



- The context is implemented without a specific algorithm.
- Upon creation the concrete strategy is supplied.
- Typically the strategy is stateless → low coupling

Creational Pattern: Factory-Method

- Abstract classes define an abstract creation method
- This method must be redefined by concrete subclasses later
- These subclasses are responsible for the final, consistent creation
- Factory-Methods are often used for instantiating the strategy (see Strategy-Pattern)

Factory-Method – Structure