Software Engineering 2 – Design Patterns. Part 2

Hochschule für Technik Stuttgart

Marcus Deininger

SS 2021

Summary

- Design Patterns
  - Supported Principles
  - Categories
- Selected Patterns Part 2
- Exercise Part 2

Design Patterns – Supported Principles

Low Coupling / High Cohesion

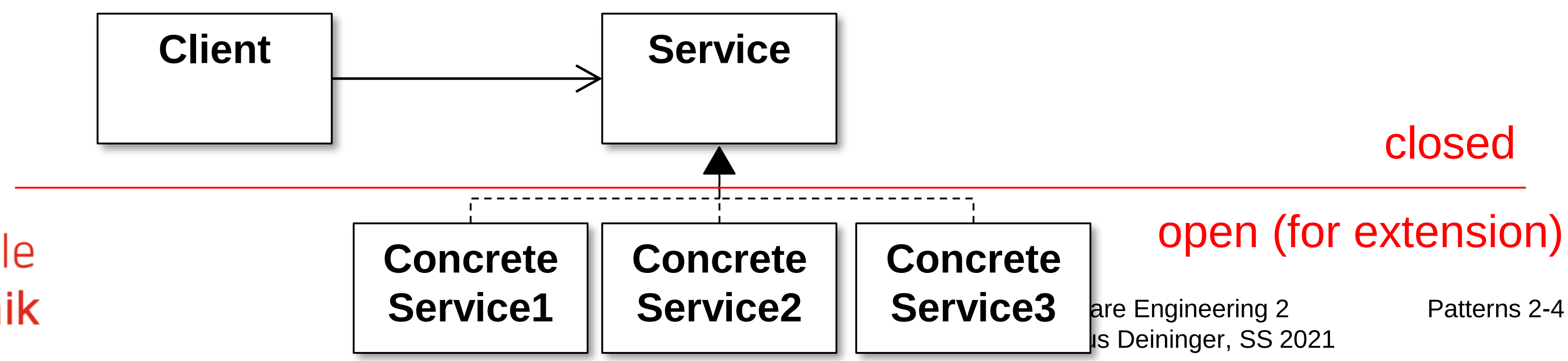
- Cohesion
  - the relationship of the elements within the module.
  - should be as high as possible
  - → all elements contribute to one task
- Coupling
  - the relationship between different modules.
  - should be as loose as possible
  - → the module does not depend on other modules
  - → can be changed / exchanged without ripple effects

Hochschule für Technik Stuttgart

Design Patterns – Supported Principles

Open/Closed-Principle (Bertrand Meyer)

- "Software entities like classes, modules and functions should be open for extension but closed for modifications."
- closed: the source code of the module inviolate; no one is allowed to make changes to the code → the module can be used without risk
- open: the module is open for extension → according to new requirements the module can be extended to behave in new and different ways



Design Patterns – Supported Principles

Dependency Inversion Principle (Robert C. Martin)

- Clients should only use/depend on interfaces / abstractions
- Details should be implemented by implementing classes / subclasses.

```java
public class Client {

    public void use(C c) {
        c.info();
    }
}
```

```java
public interface C {

    public void info();

}
```

The Client depends on the abstract interface.

```java
public class C1 implements C {

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

The implementing class provides the details.

Hochschule für Technik Stuttgart

Design Patterns – Supported Heuristics

- Inheritance should only used for specialization
- Change behavior through exchanging / overriding methods – don't use attributes for control.
- Minimize dependencies between classes.
- Do not use reflection.

Design Patterns – When to apply

- Observation: The main work takes place later in the maintenance!
- Idea: Foresee future changes already during development:
  - facilitate expected changes
  - prevent unintentional changes
- Patterns help to avoid dependencies by
  - low coupling (classes are independent)
  - high cohesion (classes focus on an objective)

  → easy to modify at designated locations.

Design Patterns – Remarks

- Design patterns support the development of structured and reusable applications
- Design patterns help to understand foreign programs and support reuse
- ***but***
  - Design Patterns alone do not guarantee project success
  - Design Patterns should not be used compulsively.

Design Patterns (Gamma et al.)

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

Additional Useful Patterns

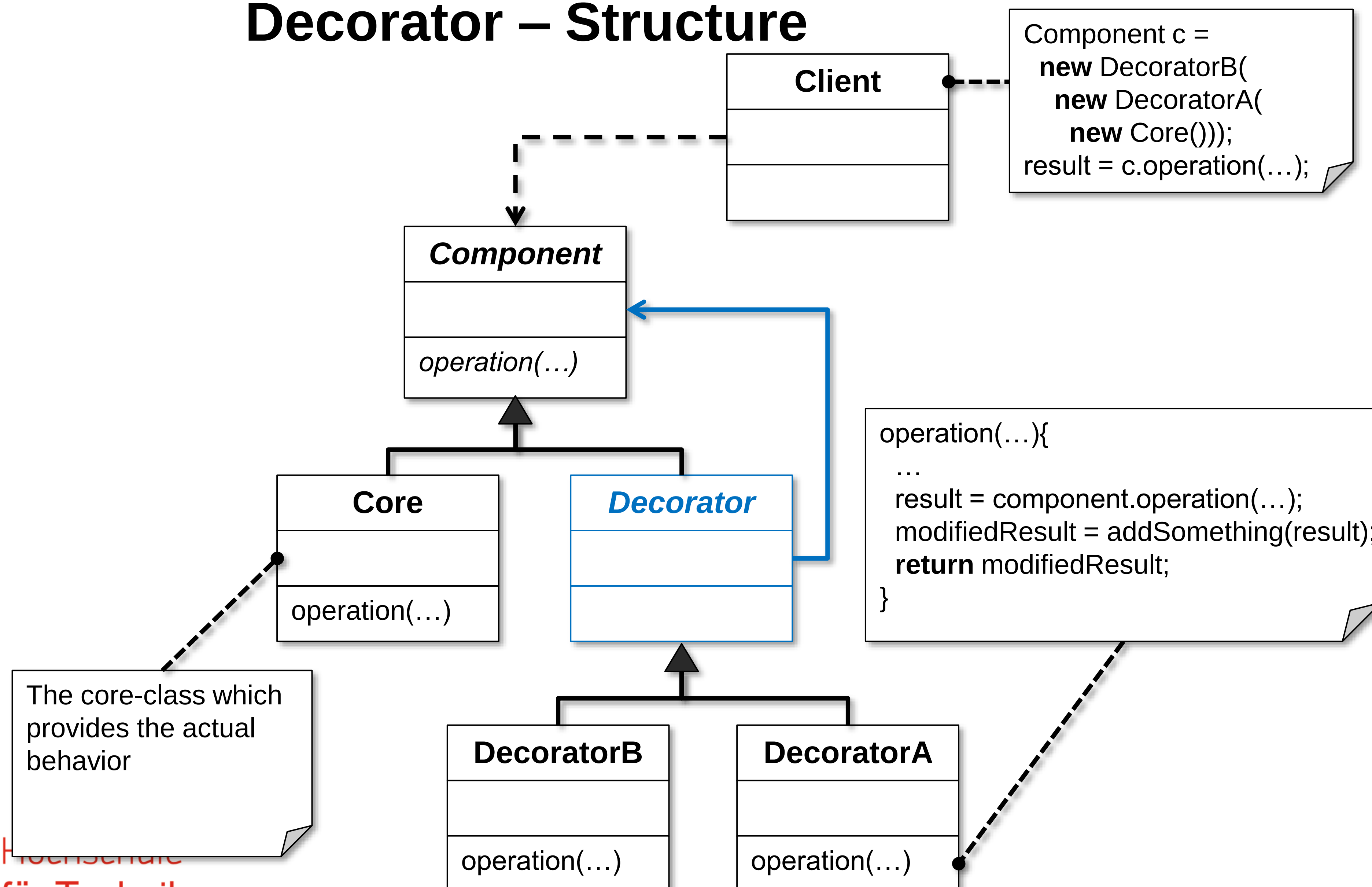
- Money: Modelling a currency without rounding issues (internally as int) and distribution ability.
- Registry: Global accessor for services or objects.
- Data Transfer Object: In a distributed environment transfer data is grouped in one object for better performance.
- Persistency Map: Stores once loaded records for faster re-access.
- Lazy Load: Objects do not contain all data – remaining data will be loaded on demand.

Selected Go4 Patterns (Part 2)

Structural Pattern: Decorator

- Aspects / Behavior should be added dynamically to an object
- the object to be extended is wrapped into decorators
  - each decorator may contain another decorator
  - or (finally) the object himself
- all calls are delegated through the decorator
- each decorator has the chance to add his “flavor” to the call (or response)

Decorator – Structure



Behavioral Pattern: Observer

- Definition of a 1:n-relationship between a state object (*Subject*) and several displaying objects
- If the subject changes its state the depending objects should be updated automatically
- the subject may defer the update until a sequence of operations is done

→ Separation of representation and business logic

Class Observable

Methods

Modifier and Type	Method and Description
`void`	`addObserver(Observer o)` Adds an observer to the set of observers for this object, provided that it is not the same as some observer already in the set.
`protected void`	`clearChanged()` Indicates that this object has no longer changed, or that it has already notified all of its observers of its most recent change, so that the `hasChanged` method will now return `false`.
`int`	`countObservers()` Returns the number of observers of this `Observable` object.
`void`	`deleteObserver(Observer o)` Deletes an observer from the set of observers of this object.
`void`	`deleteObservers()` Clears the observer list so that this object no longer has any observers.
`boolean`	`hasChanged()` Tests if this object has changed.
`void`	`notifyObservers()` If this object has changed, as indicated by the `hasChanged` method, then notify all of its observers and then call the `clearChanged` method to indicate that this object has no longer changed.
`void`	`notifyObservers(Object arg)` If this object has changed, as indicated by the `hasChanged` method, then notify all of its observers and then call the `clearChanged` method to indicate that this object has no longer changed.
`protected void`	`setChanged()` Marks this `Observable` object as having been changed; the `hasChanged` method will now return `true`.

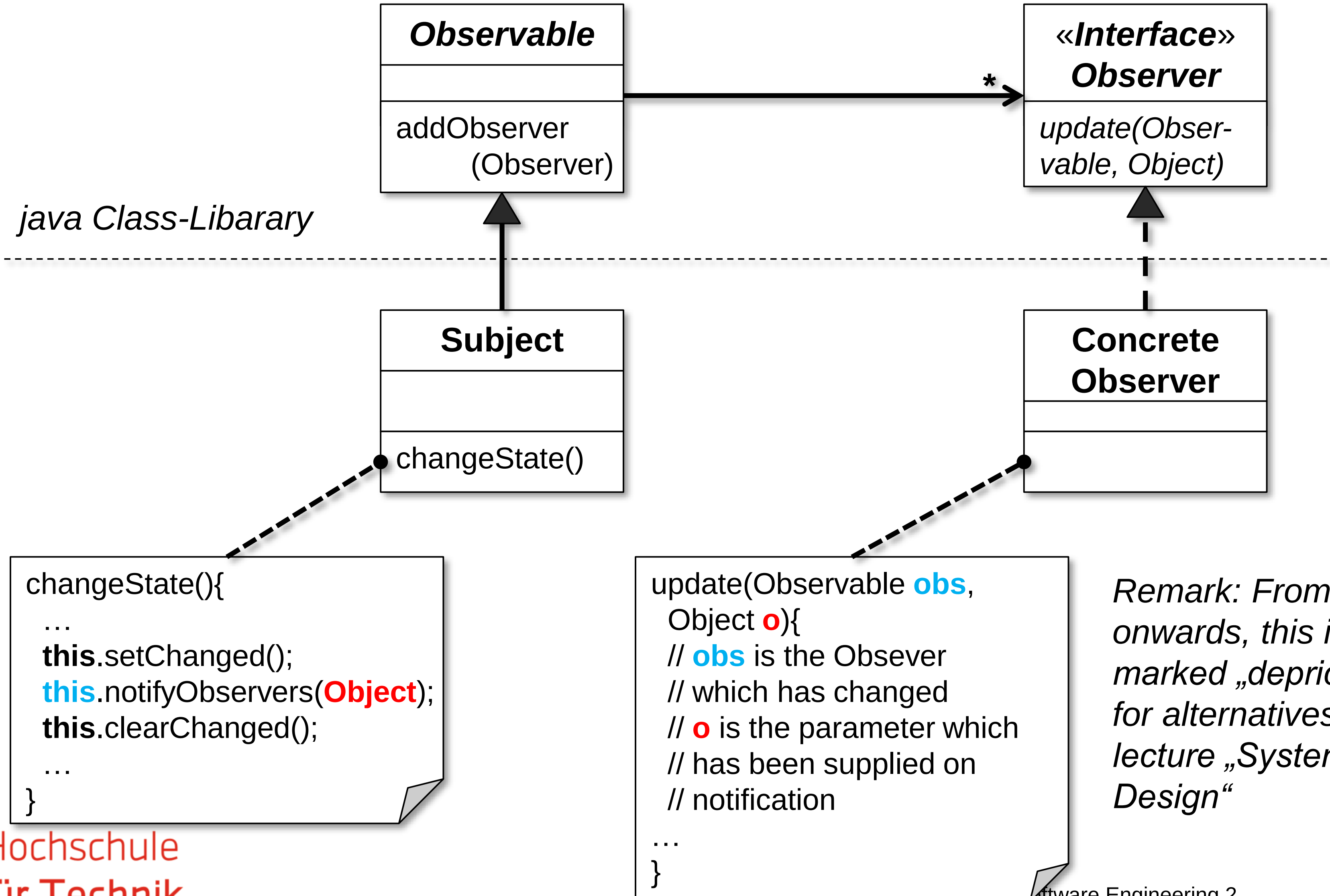
Interface Observer

Methods

Modifier and Type	Method and Description
`void`	`update(Observable o, Object arg)` This method is called whenever the observed object is changed.

From Java 9 onwards this will be deprecated- suggested replacement Property Change Listener

Observer – Structure



Remark: From Java 9 onwards, this is marked „depricated“; for alternatives see lecture „System Design“

Structural Pattern: Composite

Situation

- Modeling a Part-Whole-Relationship
- Parts (primitives) and composite objects (container) should provide the same interface

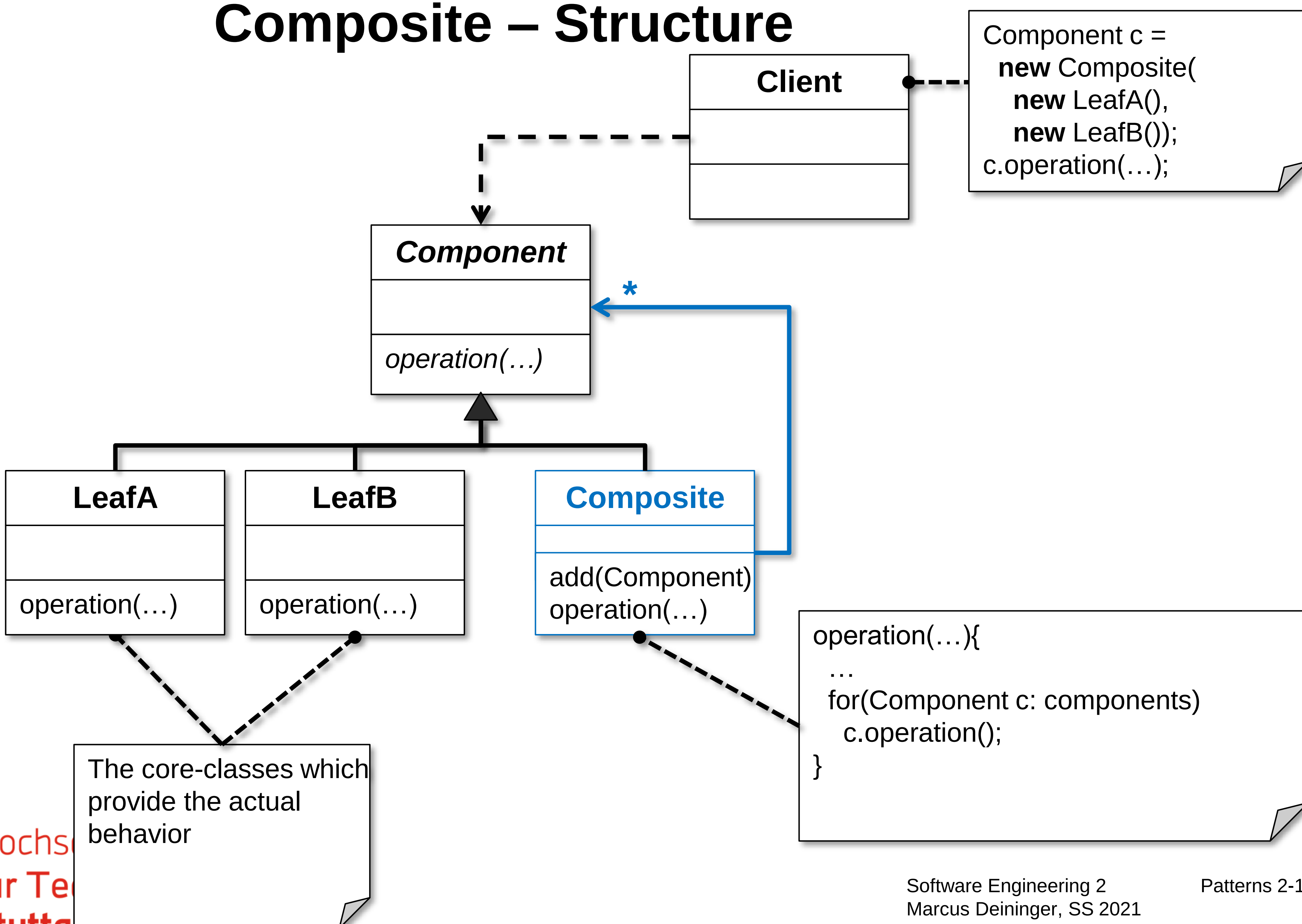
Solution

- Define a common superclass for primitives and containers

Examples

- all kinds of tree-structures: Parse-Trees, grouped graphical objects, GUI-Elements

Composite – Structure



Hochschule für Technik Stuttgart

Behavioral Pattern: Template Method

Situation

- Implementation of an algorithm with variants
- new variants should be added easily
- minimal code duplication

Solution

- the common parts of the algorithm → template method
- variant parts implemented by new methods → called by template method
- all variants have to follow the same schema

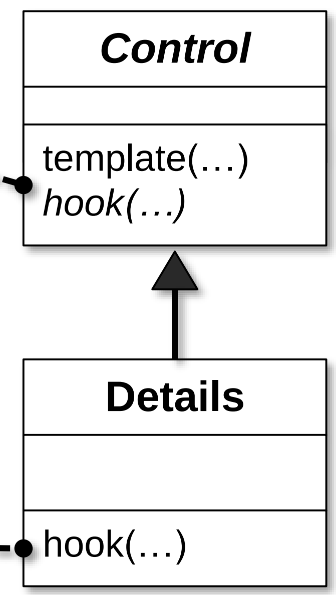
Template Method – Structure

The **Template-Method** provides the overall execution structure/ control:

```
template(…){
  …
  this.hook(…);
  …
}
```

The hook is an abstract method which calls for some (primitive) detail – which is not decidable on implementation

Only the hook has to be implemented to make the subclass running



Template Method – Remarks

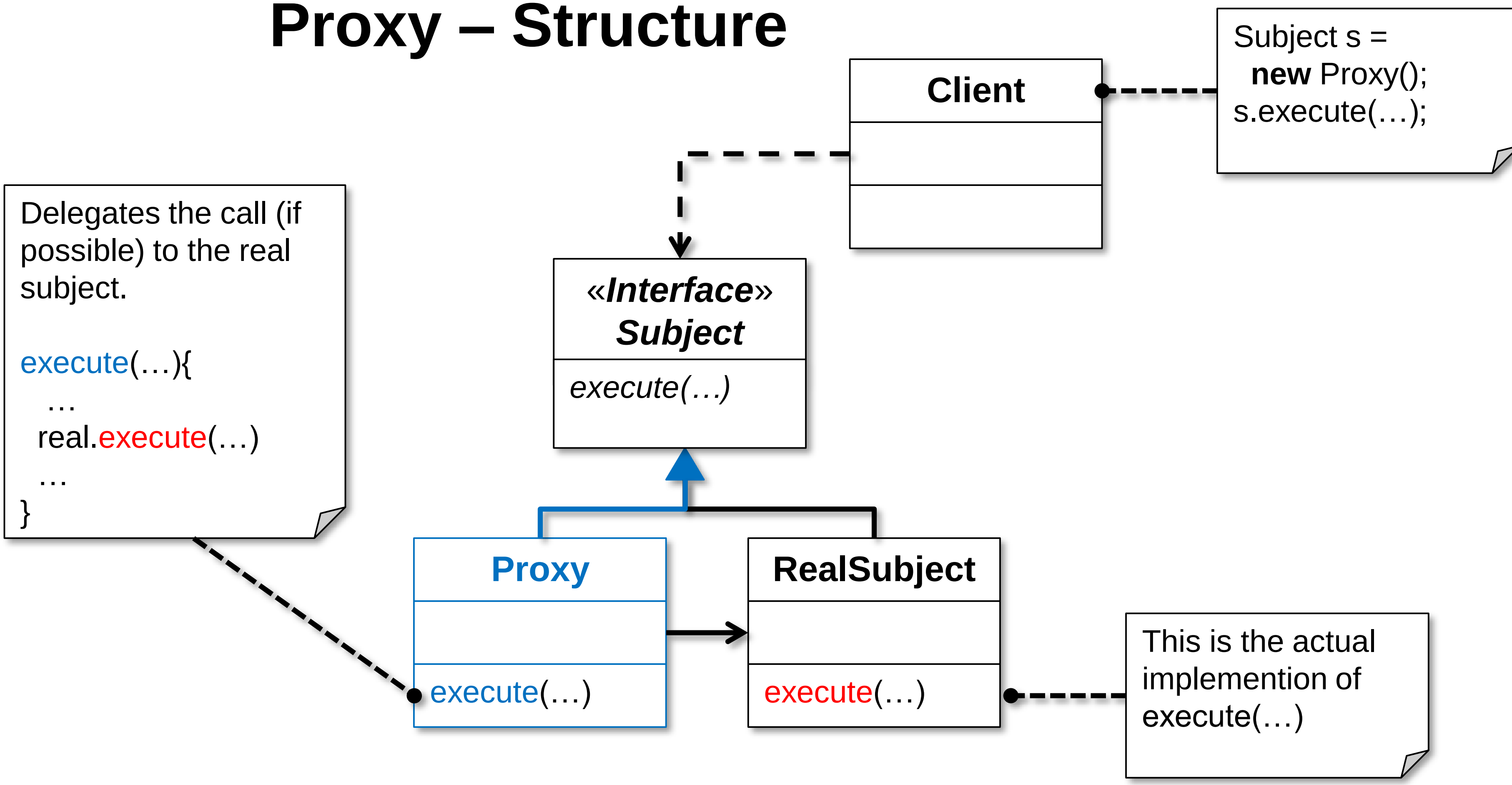
A main pattern for framework design

- inversion of control: „Don‘t call us, we‘ll call you“ → superclass calls methods of subclass
- Primitive operations need not to be abstract but may implement default behavior
- The number of primitive operations should be as minimal as possible

Structural Pattern: Proxy

- Instead of the real object a “proxy” should be used
- Calls from the proxy should be delegated to the real object
- Typically
  - the proxy defers the instantiation of the real object until it is really needed.
  - filters the calls/responses.

Proxy – Structure



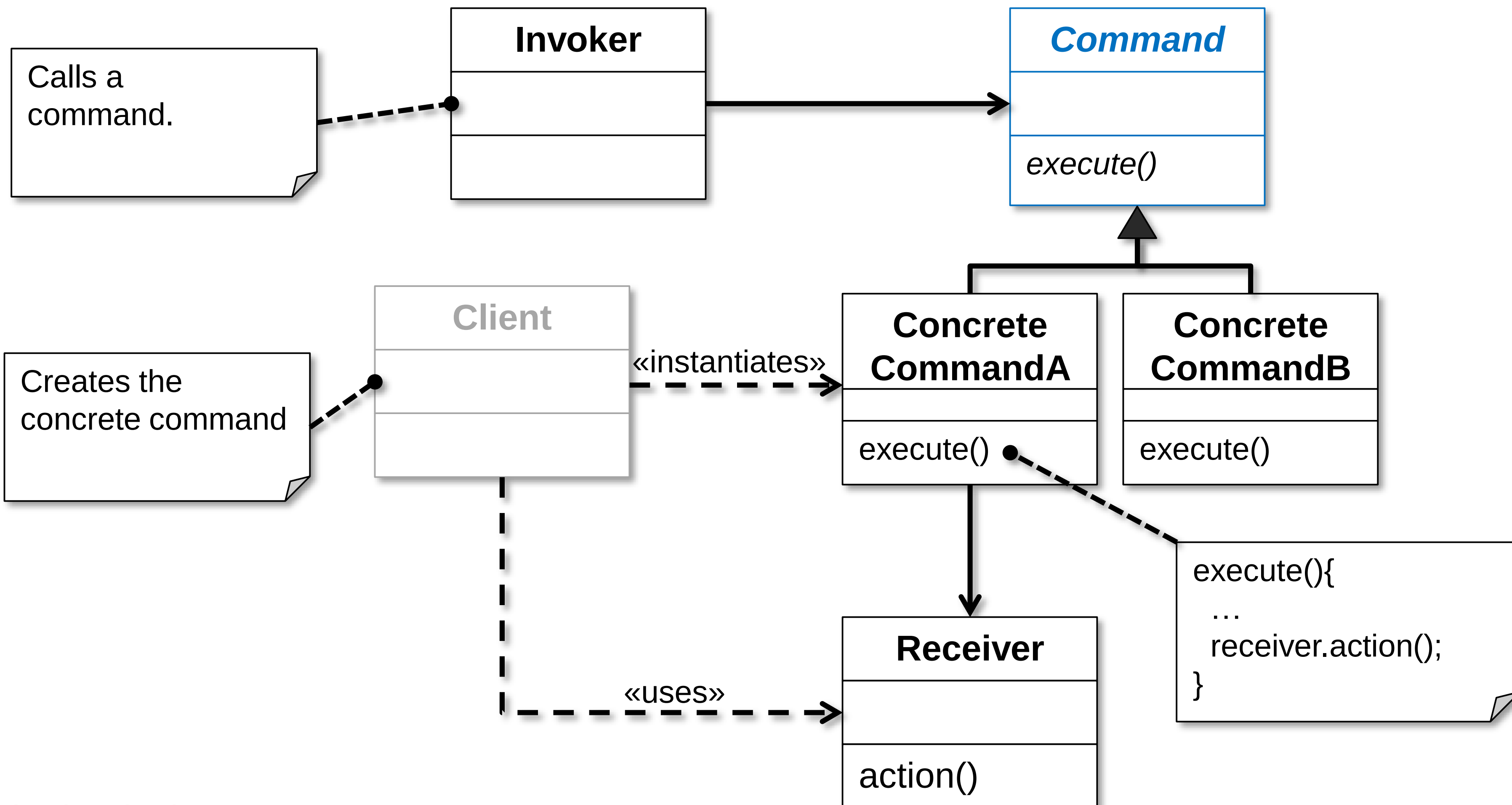
Hochschule für Technik Stuttgart

Behavioral Pattern: Command

- Encapsulation of methods
- Decoupling of Invoker and Receiver
- The Invoker is independent of the actual called method
- Methods can be added / exchanged during runtime
- Method calls can be logged / delayed / undone

→ Methods are treated as objects

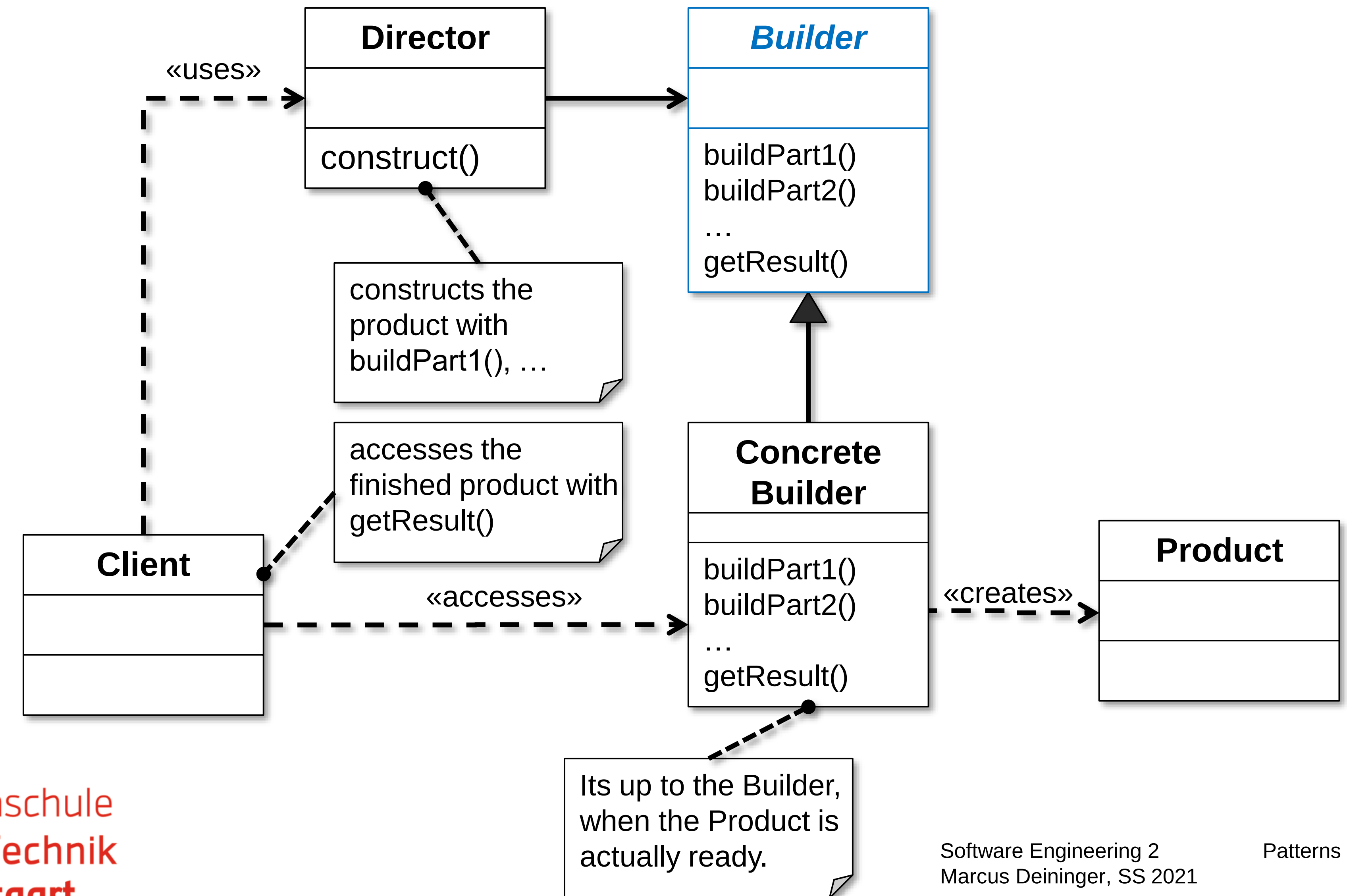
Command – Structure



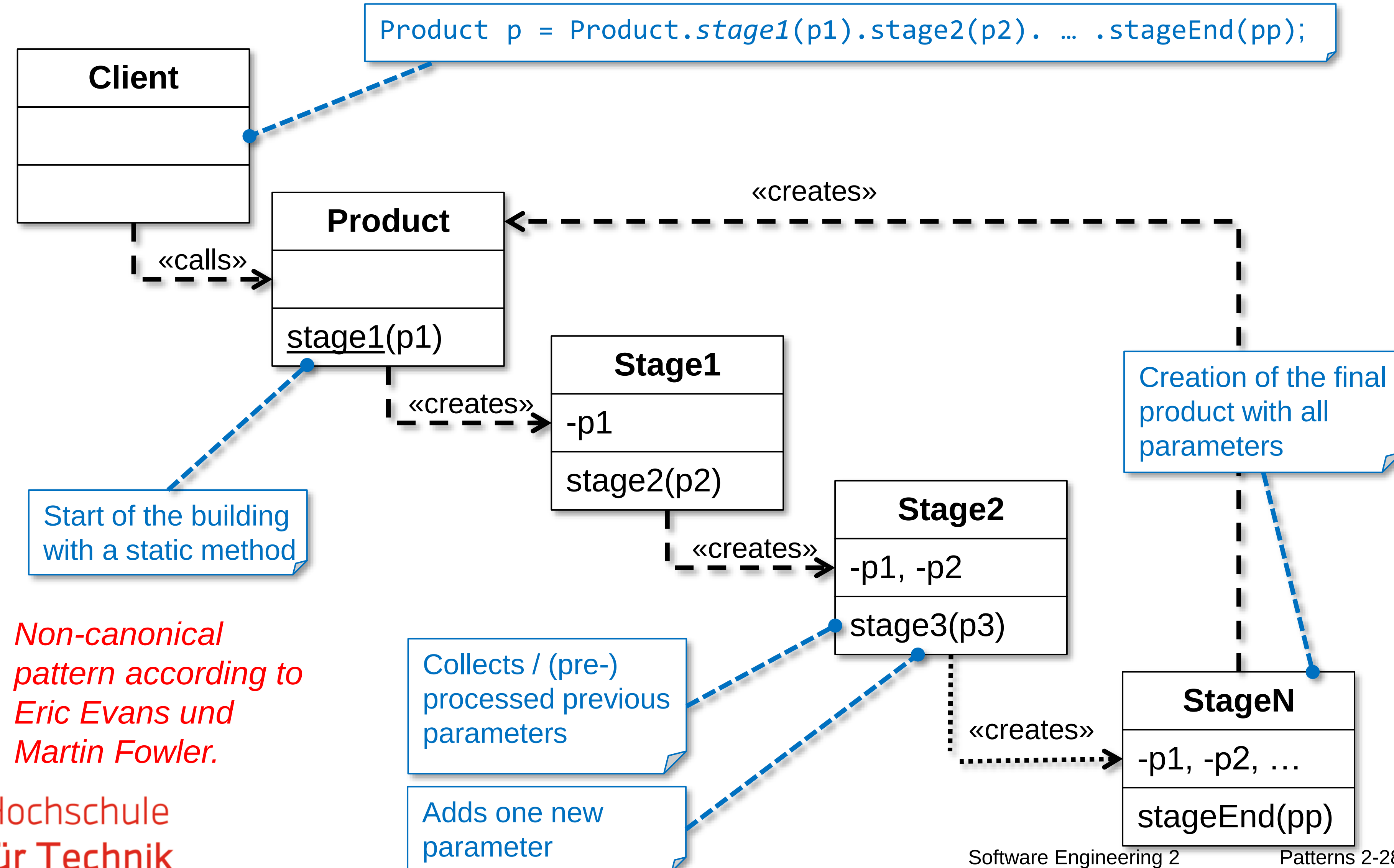
Creational Pattern: Builder

- Encapsulates the construction of a product and allows it to be constructed in steps (by a Director).
- The constructed Product is explicitly requested by the client.

Builder – Structure



Builder – With a Fluent API



Non-canonical pattern according to Eric Evans und Martin Fowler.

Singleton – Usage

Situation

- for one class only one instance should be created
- this instance should be globally accessed
- e.g. printer spooler, file system, window manager

Solution 1

- Access instance through global variable (doesn’t prevent multiple instantiation)

Solution 2

- Define class variables / methods only (no inheritance possible)

Singleton – Implementation

```
class Singleton {
        static private Singleton instance = null;
        static public Singleton getInstance() {
                if (instance == null)
                        instance = new Singleton();
                return instance;
        }
        /* no public constructor allowed
           for usage in subclasses declared as
           “private” */

        private Singleton() { ... }

        ... // instance variables and methods here
}
```

Attention: not thread-safe!

Singleton – Thread-safe-Implementation

```
class Singleton {
    private static Singleton instance = new Singleton();

    public static Singleton getInstance() {
        return instance;
    }
    ...
}
```

Initialization at class load time.

```
class Singleton {
    private static Singleton instance = null;

    public static Singleton getInstance() {
        if (instance == null)
            synchronized (Singleton.class) {
                if (instance == null)
                    instance = new Singleton();
            }
        return instance;
    }
    ...
}
```

Lazy initialization.

Until entering the synchronized block someone else may have it initialized.

Hochschule
für Technik
Stuttgart

References

- Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley, 2002.
- Freeman, E. et al.: Heads First Design Patterns. O'Reilly, 2004.
- Gamma, E., R. Helm, R. Johnson, J. Vlissides; Design Patterns, Addison-Wesley, 1996.
- Patterns Home Page http://hillside.net/patterns/
- An Object-Oriented Pattern Digest - Software Design Patterns Catalog, All Patterns Books and Resources http://patterndigest.com/