Technische Hochschule Rosenheim
Technical University of Applied Sciences

Object-oriented programming
Chapter 12 – Refactoring

Prof. Dr Kai Höfig

Contents

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Revision: UML
- Refactoring
- Code smells
- Design patterns

Inheritance (revision)

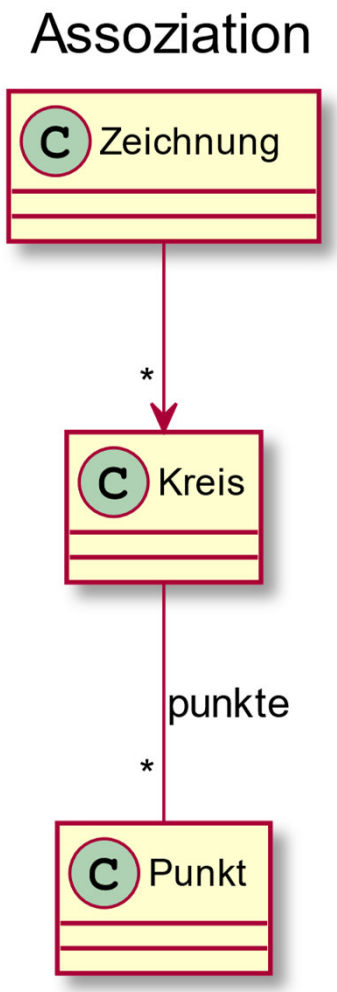
Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Inheritance in OOP represents an *is a* relationship
  - A human is a mammal
- Systematic specialisation from top to bottom
- The *subclass* thus inherits all the features of the *top-level base class*
- Subclasses inherit all accessible members of the base class:
  - Constructors and destructors
  - Attributes / class variables
  - Methods and operators
- What is not inherited?
  - Members that are declared as *private*

Association

Technische Hochschule Rosenheim
Technical University of Applied Sciences

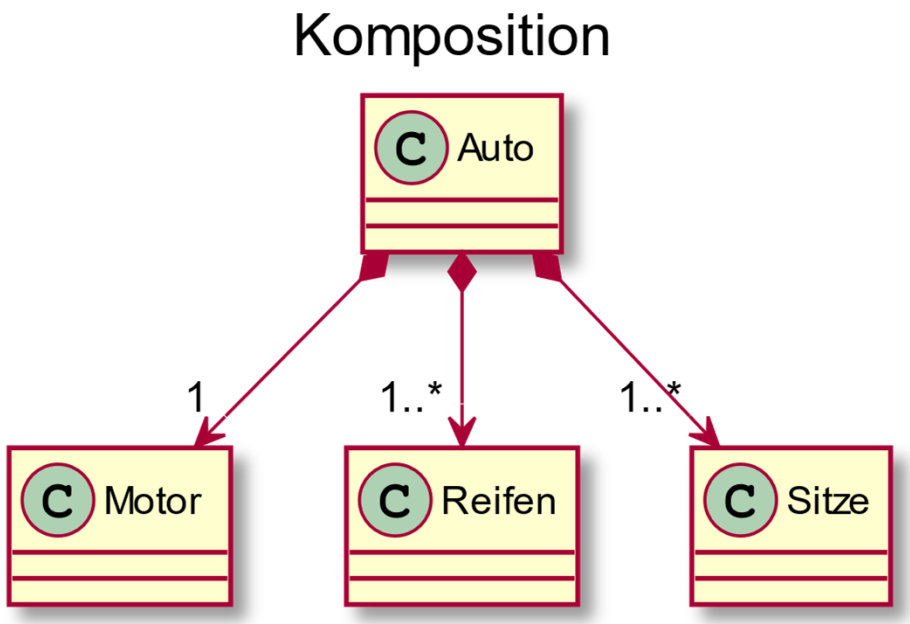
- Loose connection between objects (reference)
- Objects know each other (direction of reference)
- Cardinalities
- Name



Composition

Technische Hochschule Rosenheim
Technical University of Applied Sciences

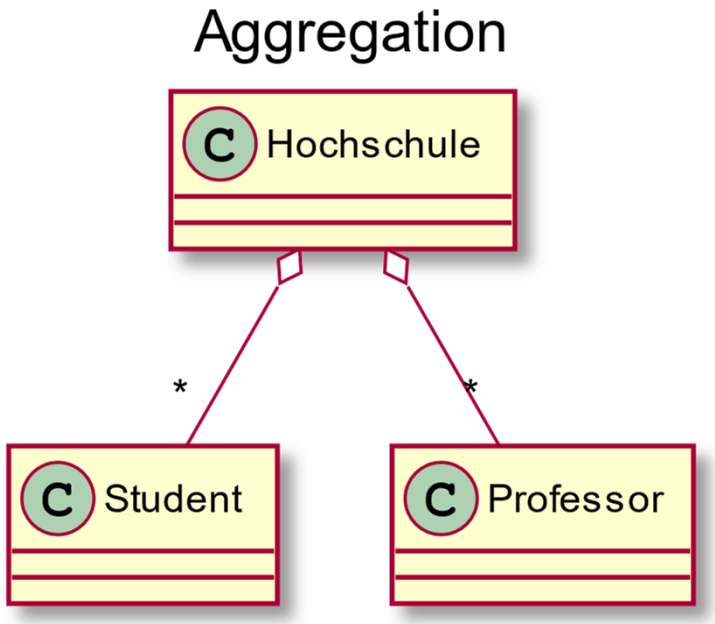
- Real and complex objects usually consist of small and simple objects
  - Car consists of tyres, engine, seats ...
  - PC consists of CPU, motherboard, RAM, ...
- In object-oriented paradigms, this relationship is called **composition**
- Forms a *consists of* or *has a* relationship



Aggregation

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Aggregation is a special form of composition
- Also forms a *has a* relationship
- However, "possession" class has no ownership rights
  - Referenced classes continue to exist and will not be destroyed if the class is destroyed
  - Referenced classes are also not automatically created if the referencing class is created



Refactoring

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- **Improving the design of existing code**

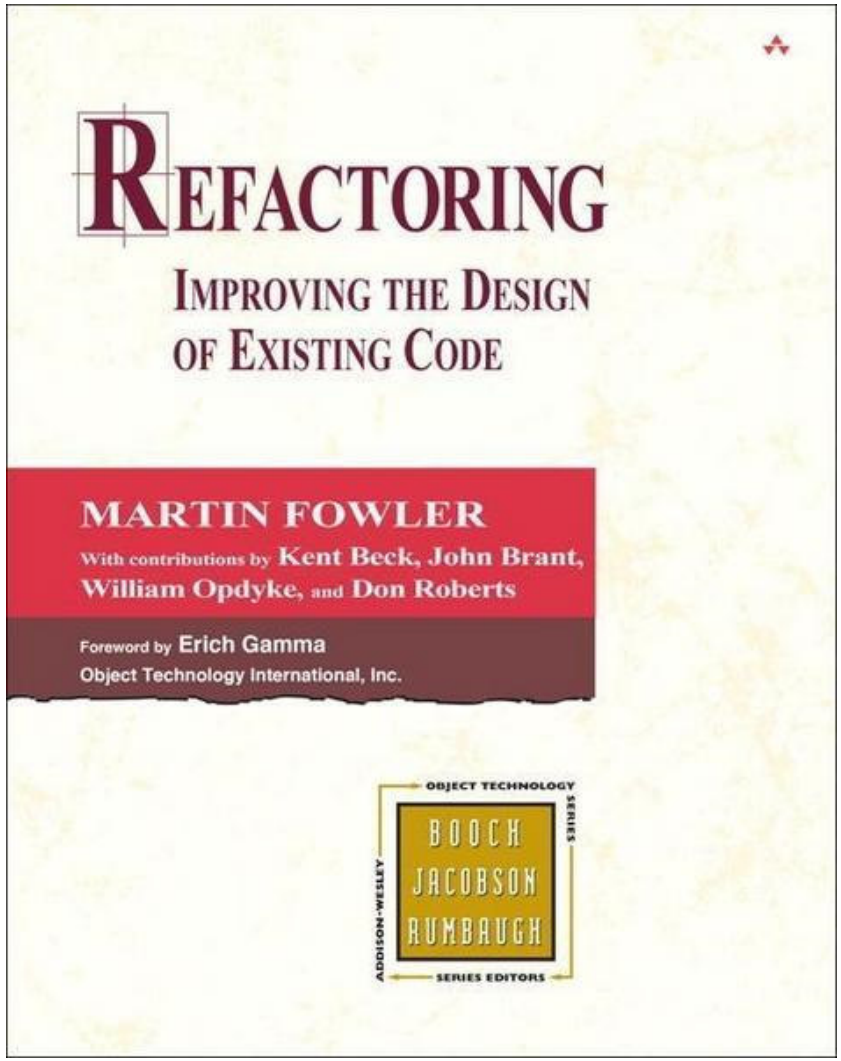
- “A change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour.” -- Martin Fowler,1999

- **A bit of history**

- In the 1980s, Ward Cunningham and Kent Beck already started explicitly with refactoring during their work with Smalltalk. They developed an Xtreme Programming (XP) software process, of which refactoring is an integral part. [-1999]
- Ralph Johnson also worked with refactoring during his work on frameworks [~1990]
- William Opdyke wrote the first scientific paper on this topic, about "Preservation of semantics when using refactoring". He is also the author of the first "refactoring" list [1992]
- Based on these ideas, John Brant, Don Roberts and Ralph Johnson developed the Refactoring Browser in Smalltalk [1996]

Book

Technische Hochschule Rosenheim
Technical University of Applied Sciences



Why refactoring?

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Increases the quality of the design
  “Building it this way is stupid, but we did it that way because it was faster”
  “Ugh, I’ll come back to this later and fix it up”
  “We should have done X, but ...”
- Improves the comprehensibility of the code
- Helps with searching for errors (troubleshooting/debugging)
- Improves the reusability

When to use refactoring?

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Copying and pasting is your enemy!
- We easily copy mistakes!
- Sometimes we copy code that we don't understand!

- So it’s better to: “refactor"
- before new functionality is added!
- to find errors!
- during the code review!
- if there are "code smells"

Code smells

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- A code smell is a hint that something has gone wrong somewhere in your code [. . .] Note that a code smell is a hint that something might be wrong, not a certainty [. . .] Calling something a code smell is not an attack; it's simply a sign that a closer look is warranted.

- -- http://xp.c2.com/CodeSmells.html

Code smell – "duplicated code or bad design"

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Ensure code is uniform if the same code structure appears in multiple places: » Same expression in methods of the same class » Same expression in sibling subclasses » Code is similar but not the same (are there possibilities here?) » Methods do the same thing but with another algorithm

Code smell – "same expression in subclasses"

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
public class Employee {
}
```

```
public class Salesman {
    String getName(){
    return "";
    }
}
```

```
public class Engineer {
    String getName(){
    return "";
    }
}
```

- Becomes…

```
public class Employee {
    String getName(){
    return "";
    }
}
```

```
public class Salesman {
}
```

```
public class Engineer {
}
```

Refactoring techniques – "pull up to super"

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- If code is common code across all subclasses, it belongs in the superclass.

```
class Manager extends Employee {
  public Manager(String name, String id, int grade) {
    this.name = name;
    this.id = id;
    this.grade = grade;
    }
  // ...
}
```

- Becomes...

```
class Manager extends Employee {
  int grade;
  public Manager(String name, String id, int grade) {
    super(name, id);
    this.grade = grade;
    }
  // ...
}
```

Code smell – "switch statements"

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Switch statements are often indications of poor object-oriented design. The `EuropeanBird` class can be derived from the `Bird` class, and the switch statement disappears.

```
class Bird {

  public static final int EUROPEAN = 1;
  public static final int AMERICAN = 2;
  public static final int NORWEGIAN_BLUE = 3;

  int type;

  public double getSpeed() {
    switch (type) {
      case EUROPEAN:
        return getBaseSpeed();
      case AMERICAN:
        return getBaseSpeed() - numberOfLoad();
      case NORWEGIAN_BLUE:
        return getBaseSpeed()*2;
        }
    throw new RuntimeException("Unreachable");
    }
…
}
```

```
public class EuropeanBird extends Bird {
  public double getSpeed() {
    return getBaseSpeed();
    }
}
```

Code smell – "comments as design"

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Comments should not be used to prescribe the use, but only to document what the method does.

```
/**
 * FOR INTERNAL USE ONLY (called by controller)!
 *
 * Updates this view based on the ViewDef passed.
 *
 * @param pViewDef ViewDef, based on which this view should be updated.
 * @return receiver modified
 */
public void updateWith( ViewDef pViewDef ) {
  if (pViewDef==null) return;
  setUpdatePending(false);
    View lView = replaceOrUpdateViewFor( pViewDef );
  lView.updatePresentationMode( pViewDef );
  if ( isReinitialise() )
    lView.toReinitialise();
}
```

Refactoring techniques – "extract method"

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- If code fragments are used multiple times or logically belong together, then move that code into a separate new method, and replace the old code with a call of the method.

```
void printOwing() {
    printBanner();
  // Print details.
  System.out.println("name: " + name);
    System.out.println("amount: " + getOutstanding());
}
```

- Becomes...

```
void printOwing() {
    printBanner();
    printDetails(getOutstanding());
}

void printDetails(double outstanding) {
    System.out.println("name: " + name);
    System.out.println("amount: “ + outstanding);
}
```

Refactoring techniques – "extract method"

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
public void actionPerformed(ActionEvent e) {
    ...
  // out
  for (int i = 0; i < classes.size(); i++) {
    System.out.println(classes.get(i));
    }
  // compile
  for (int i = 0; i < classes.size(); i++) {
    Compiler.compileClass((Class) classes.get(i));
    }
  // create database
  DatabaseBuilder lDatabase = new DatabaseBuilder(
      "mysql", "jdbc.mysql.MySqlDriver");
  try {
    lDatabase.executeScript(script, ";");
    } catch (Exception ex) {
    ex.printStackTrace();
    }
  // load classes
  Method lMethod = lClass.getDeclaredMethod(
      "newInstance", new Class[]{Properties.class});
        ...
}
```

Refactoring techniques – "remove dead code"

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
if(false) {
    doSomethingThatUsedToMatter();
}
```

- Becomes...

```
//nothing
```

Refactoring techniques – "replace constructor with factory function"

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- If more is done in the constructors than just initialising the parameters of an object, it is worth using a factory method. Otherwise, cascading constructor calls might occur, which are difficult to coordinate.

```
public class Employee {
    String name;

    Employee(String name) {
    this.name = name;
  }
..
}
```

becomes

```
public class Employee {
    String name;

  private Employee(String name) {
    this.name = name;
  }

  static Employee create(String name) {
        Employee e = new Employee(name);
    // do some heavy lifting.
    return e;
    }

}
```

Design patterns

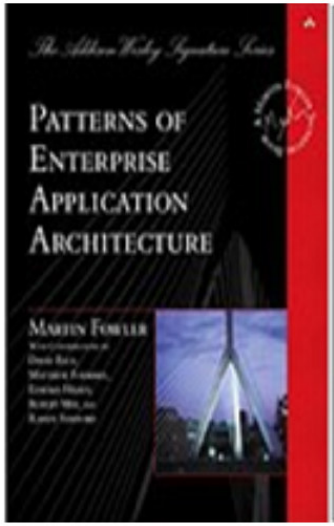
Technische Hochschule Rosenheim
Technical University of Applied Sciences

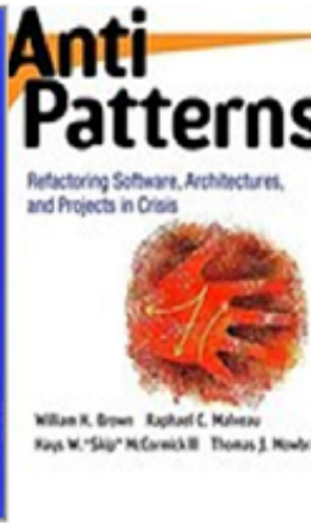
- Design patterns are proven solutions for recurring design problems in software development
- They describe the essential design decisions (class and object arrangements)
- The use of design patterns makes a design flexible, reusable, extendible, easier to use and stable for changes
- Many years of professional experience of many software developers are manifested in the design patterns
- At the same time, design patterns teach effective object-oriented modelling

Design patterns

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Do not reinvent the wheel!
- Use patterns and learn from others!





The "Gang of Four" (GoF): design patterns

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- 23 different design patterns (Java, C++ and Smalltalk)
- 3 categories (creational, structural, behavioural)

- Defined description structure
  - Name (including synonyms)
  - Task and context
  - Description of the solution
  - Structure (components, relationships)
  - Interactions and consequences
  - Implementation
  - Sample code

Design patterns in categories – creational patterns

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Abstraction
  - Makes a system independent of how its objects are instantiated, composed and represented

- Factory
- Abstract factory
- Builder
- Prototype
- Singleton

Design patterns in categories – structural patterns

Technische Hochschule Rosenheim
Technical University of Applied Sciences

Combination of classes/objects into larger structures, in order to realise new functionality

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Design patterns in categories – behavioural patterns

Technische Hochschule Rosenheim
Technical University of Applied Sciences

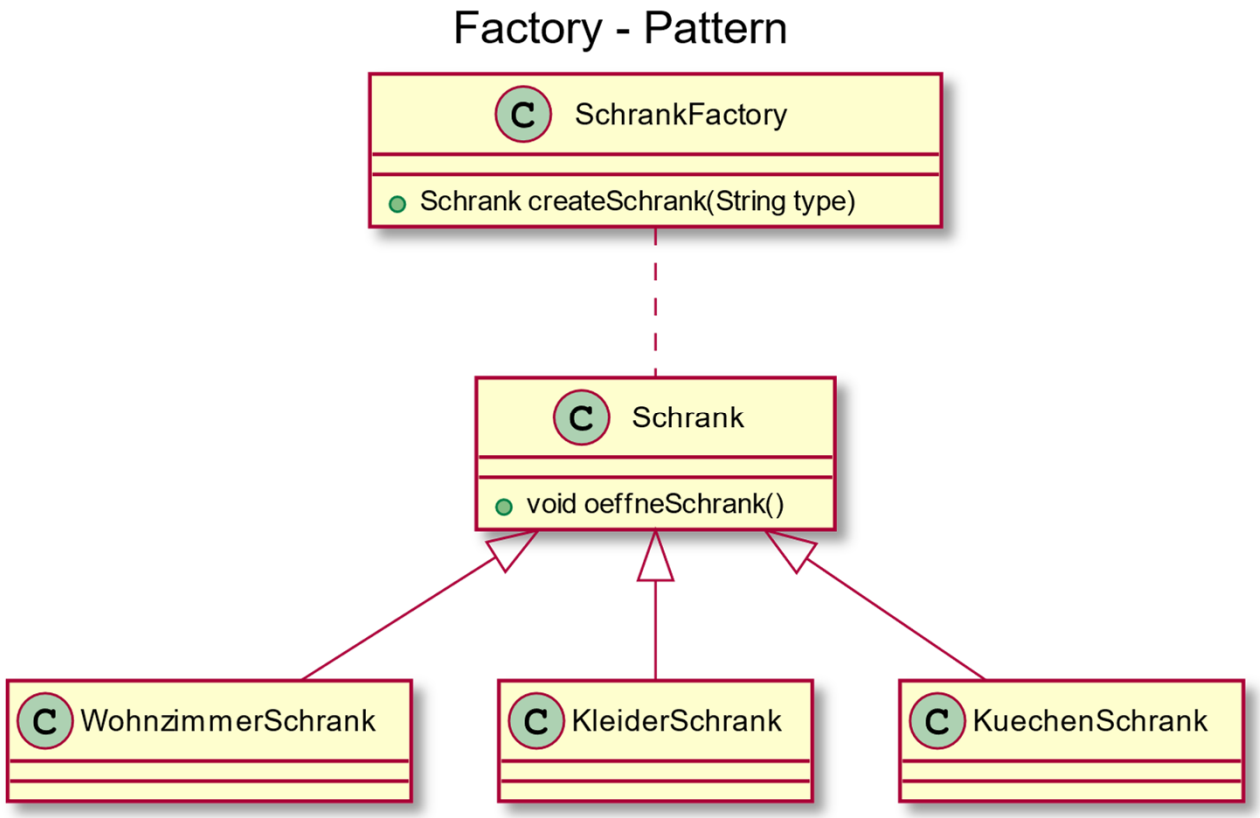
Combination, division of labour, responsibilities between classes or objects

- Interpreter
- Template method
- Chain of responsibility
- Command
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Visitor

Design patterns – *factory*

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- The factory method design pattern is used to decouple the client from the specific instantiation of a class. The object created can be exchanged elegantly. It is often used to **separate (central) object processing and (individual) object creation**.



Design patterns – *factory*

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- The (sometimes) complex creation of objects, which in turn can consist of other objects, is no longer carried out by the caller or the use of a cupboard, but similar to the factory method, is outsourced to a method. However, this pattern does not only create one type, but several subtypes.

This...

```
public static void main(String[] args) {
  Cupboard cupboard = CupboardFactory.
      createCupboard("ClothingCupboard");
  cupboard.openCupboard();
}
```

..becomes

```
public static void main(String[] args) {
  Cupboard cupboard = new ClothingCupboard();
  cupboard.openCupboard();
}
```

Design patterns – *factory*

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
public class CupboardFactory {
  public static Cupboard createCupboard(String cupboardType) {
    if (cupboardType.equals("ClothingCupboard")){
      return new ClothingCupboard();
        }
    if (cupboardType.equals("KitchenCupboard")){
      return new KitchenCupboard();
        }
    if (cupboardType.equals("LoungeCupboard")){
      return new LoungeCupboard();
        }
    return null;
    }
}
```

Design patterns: observer patterns

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- The observer pattern is one of the most widely used and well-known patterns
- In this pattern, there are two types of participants: a subject, which is observed, and one or more observers, which want to be notified of any changes to the subject
- The idea of the observer pattern is to give the subject to be observed the task of notifying the observers of a change, if any change occurs
- The observers no longer have to query the subject at regular intervals, but instead can rely on receiving a notification of a change

Observable and Observer

Technische Hochschule Rosenheim
Technical University of Applied Sciences

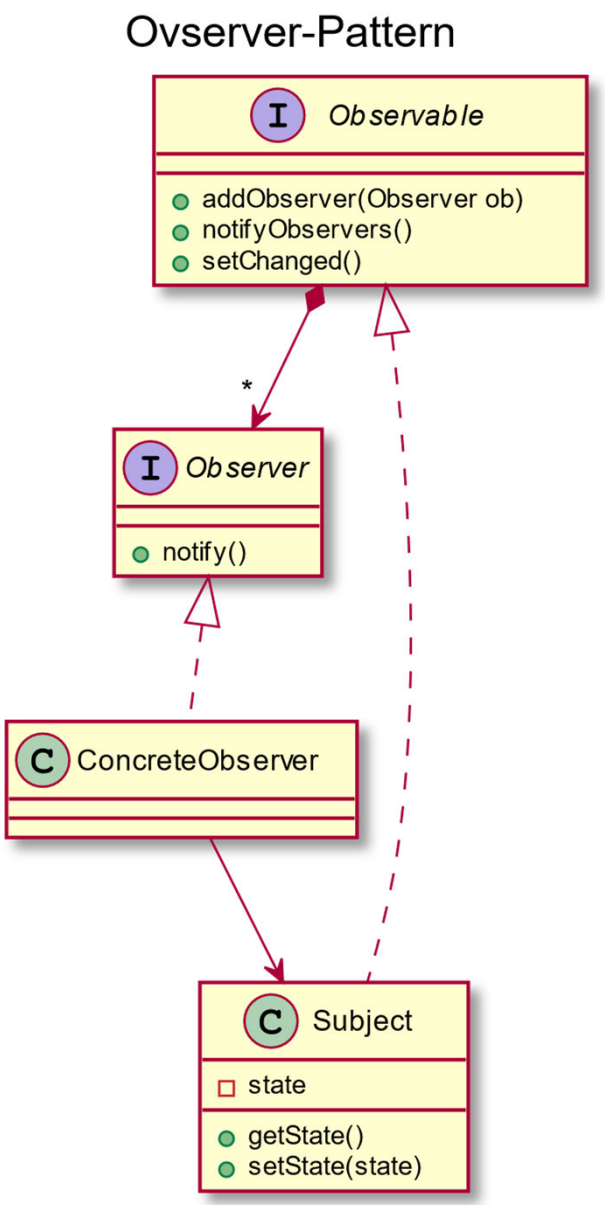
```
class Notifier extends Observable {

  public Notifier(){
    this.addObserver(new Listener_1());
    this.addObserver(new Listener_2());
        tell("blah, blah...");
    }


  public void tell(String info){
    if(countObservers()>0){
            setChanged();
            notifyObservers(info);
        }
    }
}
```

Design patterns

- **Observer patterns (II)**



Design patterns – *singleton*

Technische Hochschule Rosenheim
Technical University of Applied Sciences

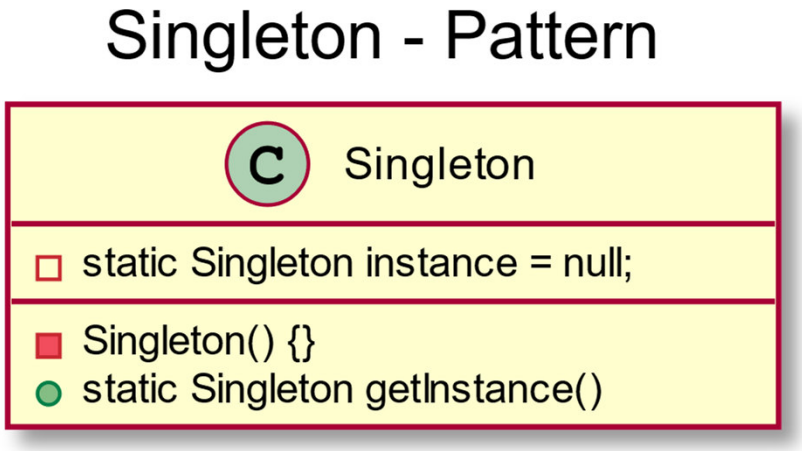
- The singleton design pattern is considered to be the simplest design pattern in terms of complexity, since it only consists of a single class.
- A class is implemented as a singleton if only one instance of it is allowed.
- Singleton guarantees that there can be at most one instance of a class, and provides a global access point to that instance.
- Examples:
  - Database connections
  - Dialogues
  - Logging objects
  - Objects that manage global data and settings

Design patterns – *singleton*

Technische Hochschule Rosenheim
Technical University of Applied Sciences

As already mentioned, the singleton design pattern consists of only one class. The main characteristics of this are:

- A static variable of the same type
- A private constructor
- A static method that returns the instance



Design patterns – *singleton*

Technische Hochschule Rosenheim
Technical University of Applied Sciences

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

  private Singleton() {
    /* Initialisation goes here! */
  }

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

}
```

Design patterns – *composite*

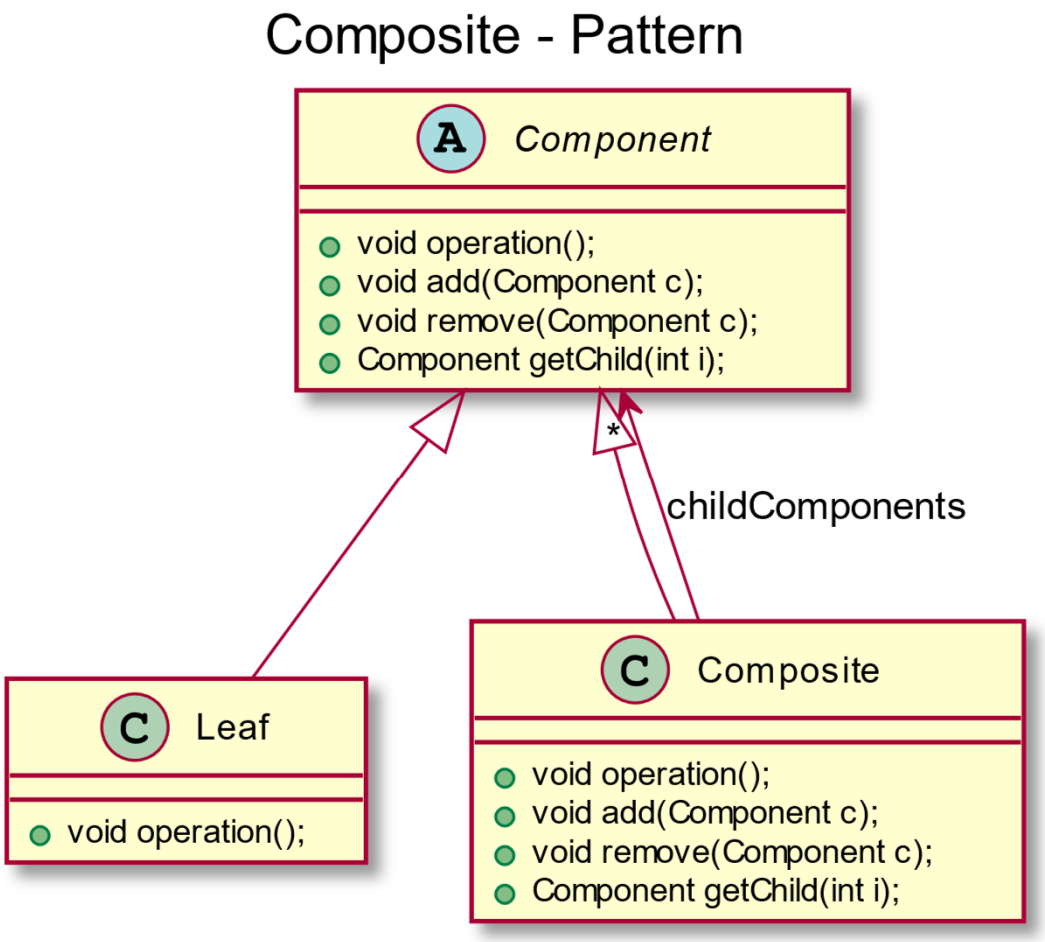
Technische Hochschule Rosenheim
Technical University of Applied Sciences

The composite design pattern enables a nested structure to be treated uniformly, regardless of whether it is an individual element or a container for additional elements

- A common interface is defined for the element containers (whole, aggregate, node - Composite) and for the individual elements (part - Leaf): Component.
- This Component interface defines the methods to be applied uniformly to Composite and Leaf objects. Composites often delegate calls (operate()) to their Components, which can be individual Leafs or also compound Composites.
- A client no longer has to distinguish between Composite and Leaf!

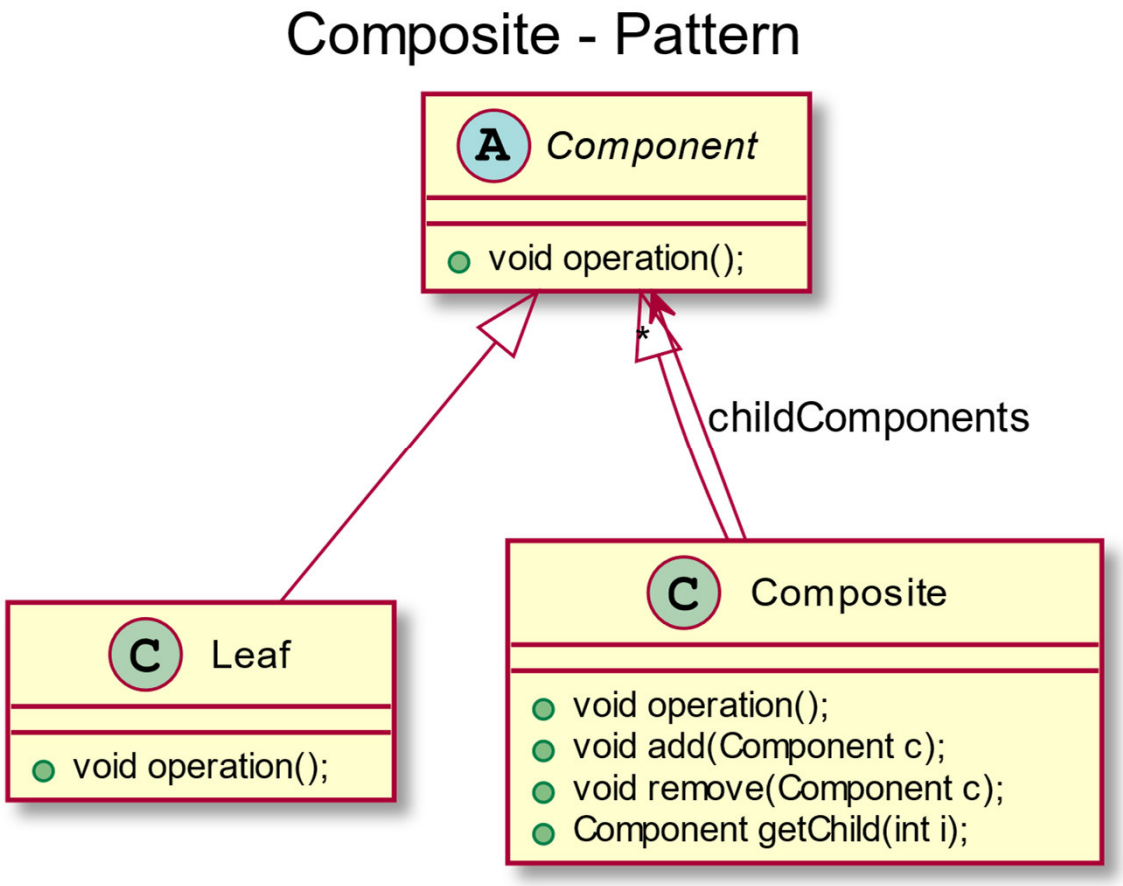
Design patterns – *composite*

Technische Hochschule Rosenheim
Technical University of Applied Sciences



Design patterns – *composite* (alternative)

Technische Hochschule Rosenheim
Technical University of Applied Sciences



Design patterns – *composite*

Technische Hochschule Rosenheim
Technical University of Applied Sciences

```
class Composite extends Component{

  //here: Components kept as a list
  private List<Component> childComponents = new ArrayList<Component>();

  //recursive call of kindComponents
  public void operation() {
        System.out.println("I am a Composite. My children are:");
    for (Component childComps : childComponents) {
            childComps.operation();
        }
    }

  //overwrite the default implementation
  public void add(Component comp) {
    childComponents.add(comp);
    }
  public void remove(Component comp) {
    childComponents.remove(comp);
    }
  public Component getChild(int index) {
    return childComponents.get(index);
    }
}
```

References

Technische Hochschule Rosenheim
Technical University of Applied Sciences

- Design Patterns @ Refactoring Guru
- TutorialPoint - Design Pattern
- [GeeksForGeeks - Design Pattern] (https://www.geeksforgeeks.org/design-patterns-set-1-introduction/)
- OODesign - GoF Pattern