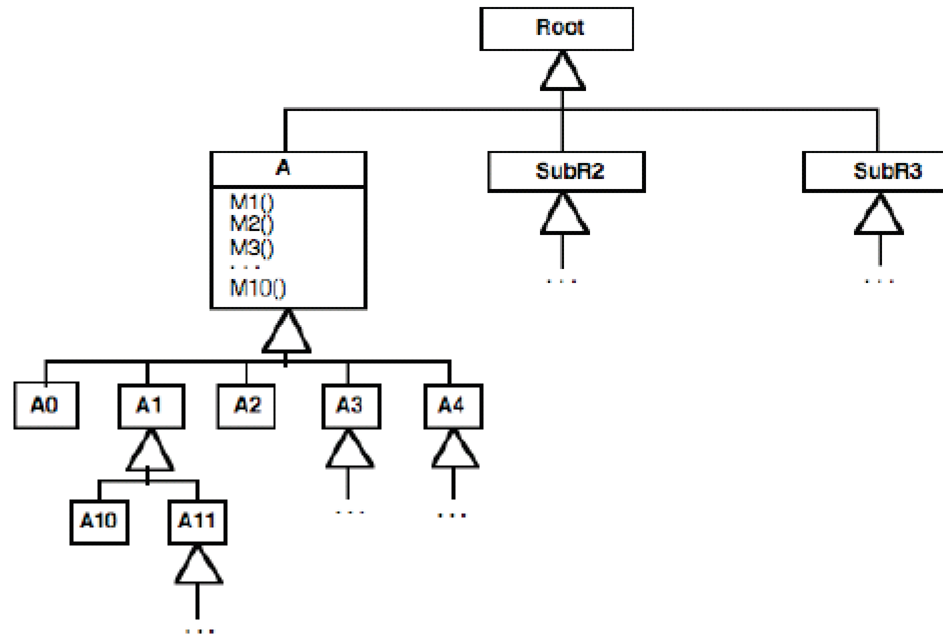
The Decorator Construction Principle

UNIVERSITÄT SALZBURG

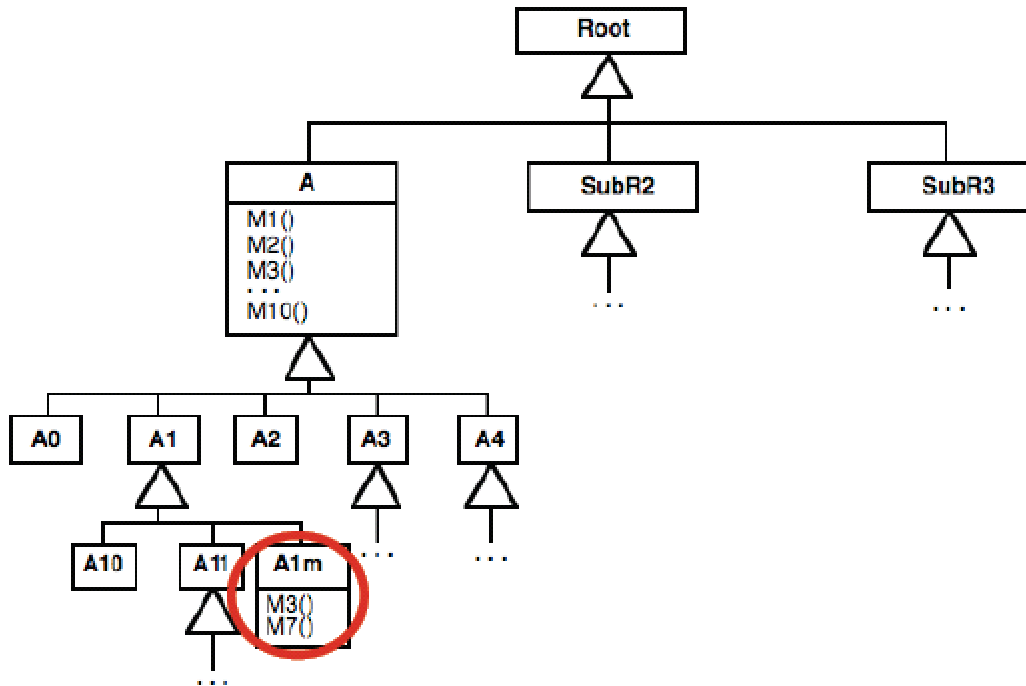
Motivation: Changes of a Class With Many Subclasses (I)



- Changes of M3() and M7() of class A necessary
- Change source text of A, if available?
- Change by inheritance?

UNIVERSITÄT SALZBURG

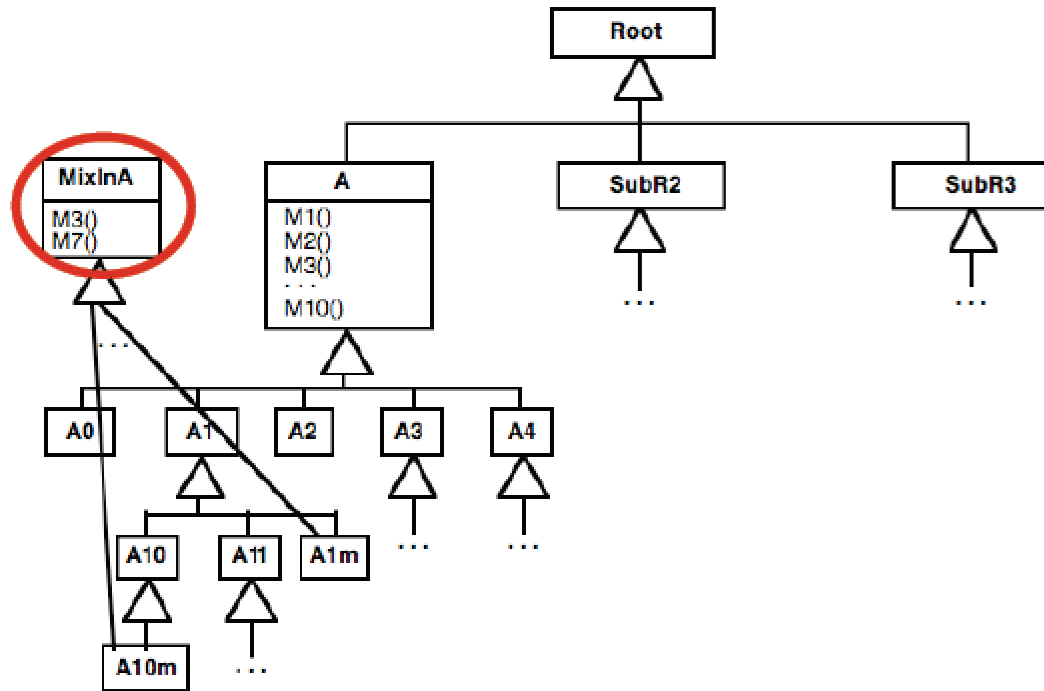
Motivation: Changes of a Class With Many Subclasses (II)



- For one class (e.g., A1m) the adaptation is meaningful
- For all subclasses of A this is too complicated

UNIVERSITÄT SALZBURG

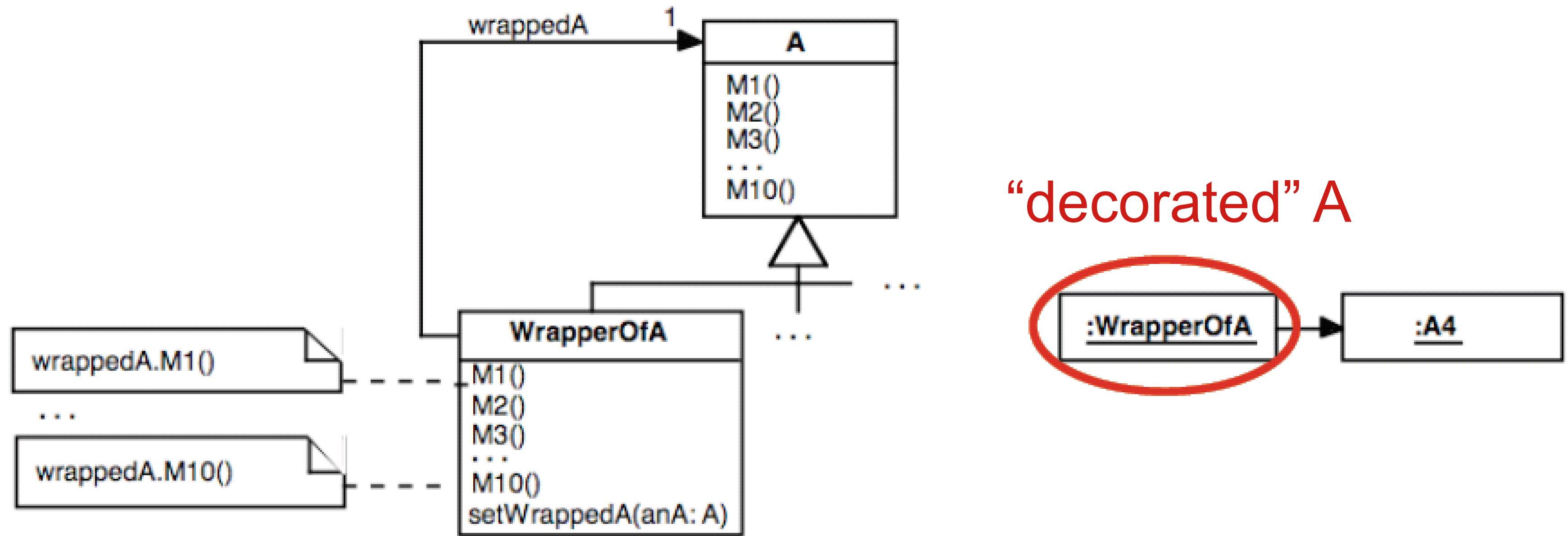
Motivation: Changes of a Class With Many Subclasses (III)



- In programming languages that support multiple inheritance, so-called **mixin** classes can be defined.
- Nevertheless a subclass must be formed for each class whose behavior is to be adapted.

UNIVERSITÄT SALZBURG

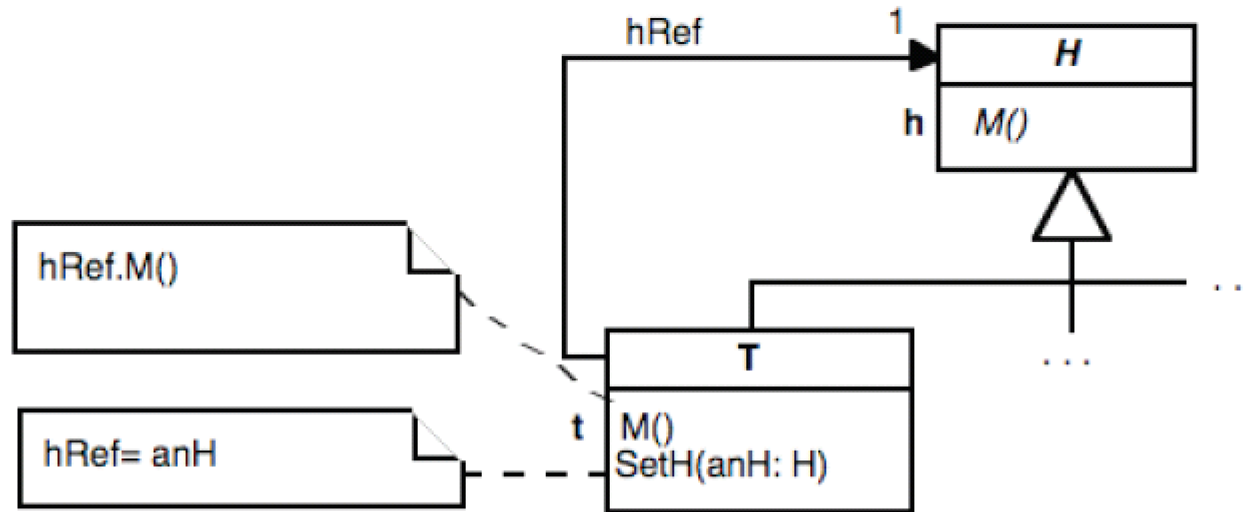
Adapting a class by composition rather than by inheritance



- All A's methods are overwritten in the class WrapperOfA, as the method call is delegated in each case to an object referenced by the instance variable *wrappedA* - with exception of those which are changed.
- Since WrapperOfA is a subclass of A, any time an object of static type A is demanded, an instance of the class WrapperOfA can be used. Since the instance variable *wrappedA* has the static type A, it can refer to each object of a subclass of A.

UNIVERSITÄT SALZBURG

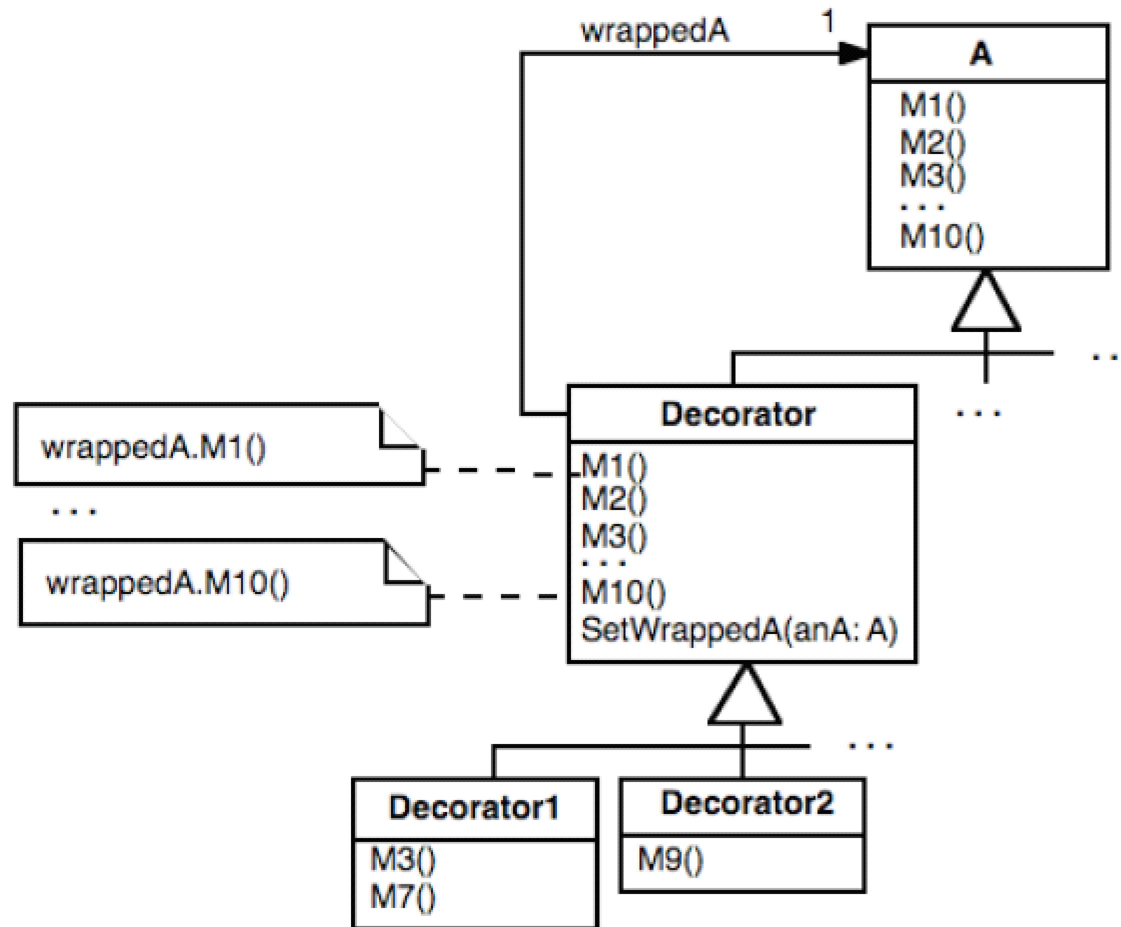
Decorator: Adaptation by composition with as many objects as desired



- The names of the template and hook methods are the same
- Setting of the decorated object with SetH()
- An instance of the decorator (filter) T as well as an instance of a subclass of H can be used by clients like an H-object. However, the behavior of the H-object is modified accordingly.
- H + decorator(s) can be used as a single object (see Composite).

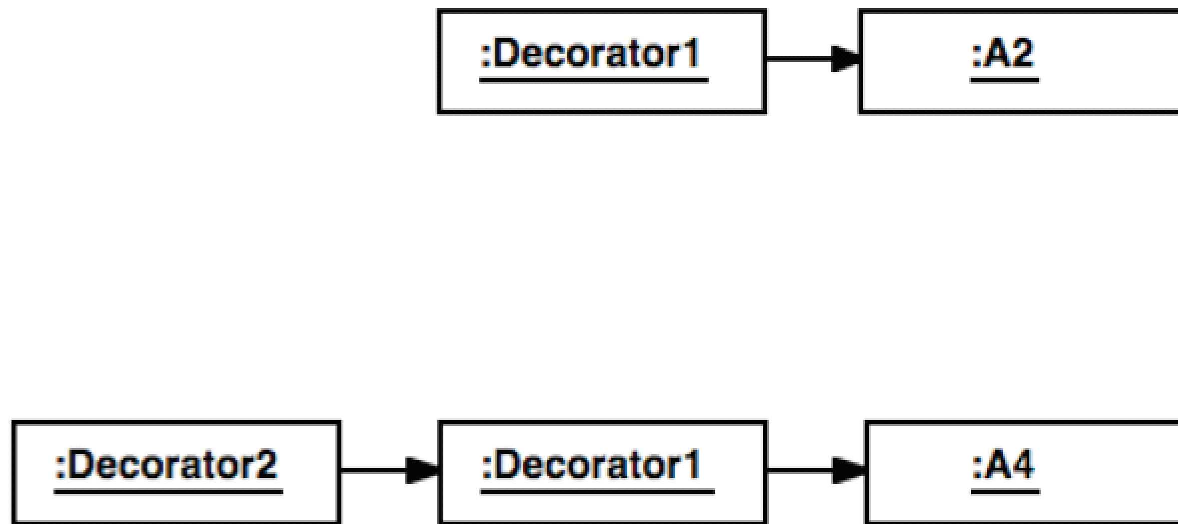
UNIVERSITÄT SALZBURG

Design suggestion when using several Decorator classes



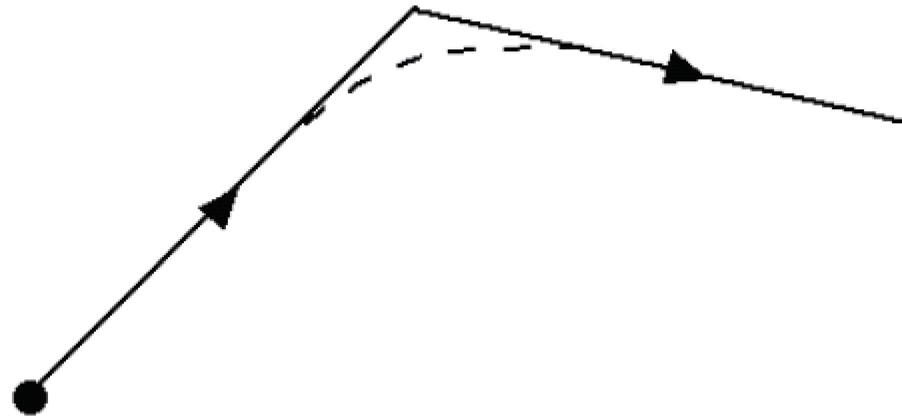
UNIVERSITÄT SALZBURG

Exemples of Compositions

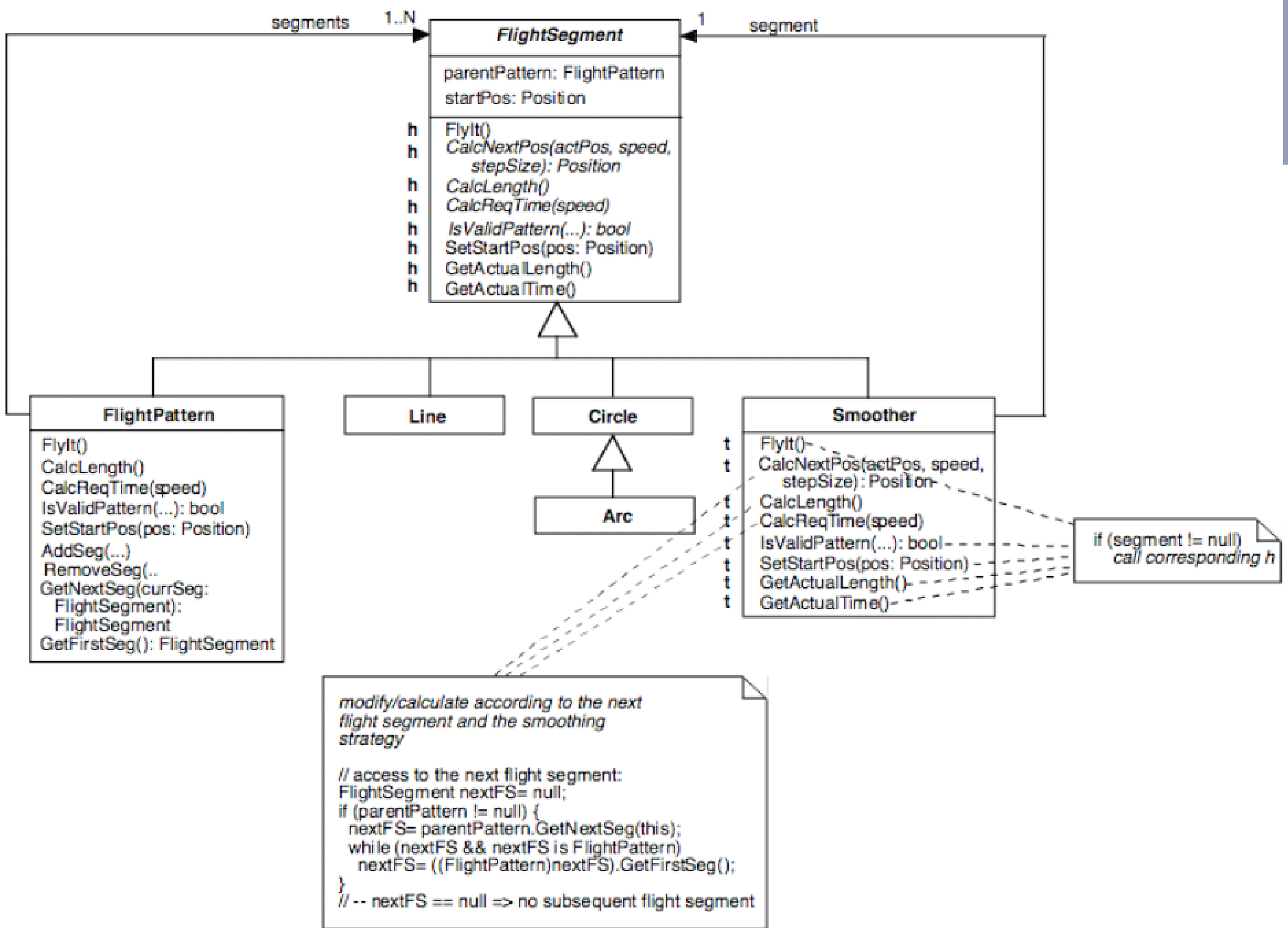


UNIVERSITÄT SALZBURG

Example: Smoothing Flight Patterns



UNIVERSITÄT SALZBURG


segments
1..N
FlightSegment
1
segment
parentPattern: FlightPattern
startPos: Position
h FlyIt()
h CalcNextPos(actPos, speed, stepSize): Position
h CalcLength()
h CalcReqTime(speed)
h IsValidPattern(...): bool
h SetStartPos(pos: Position)
h GetActualLength()
h GetActualTime()
FlightPattern
FlyIt()
CalcLength()
CalcReqTime(speed)
IsValidPattern(...): bool
SetStartPos(pos: Position)
AddSeg(...)
RemoveSeg(..
GetNextSeg(currSeg: FlightSegment): FlightSegment
GetFirstSeg(): FlightSegment
Line
Circle
Arc
Smoother
t FlyIt()
t CalcNextPos(actPos, speed, stepSize): Position
t CalcLength()
t CalcReqTime(speed)
t IsValidPattern(...): bool
t SetStartPos(pos: Position)
t GetActualLength()
t GetActualTime()
if (segment != null)
call corresponding h
modify/calculate according to the next flight segment and the smoothing strategy
// access to the next flight segment:
FlightSegment nextFS= null;
if (parentPattern != null) {
nextFS= parentPattern.GetNextSeg(this);
while (nextFS && nextFS is FlightPattern)
nextFS= ((FlightPattern)nextFS).GetFirstSeg();
}
// -- nextFS == null => no subsequent flight segment

Use of the Decorator Class Smoother

```
FlightPattern triangle= new FlightPattern();
triangle.SetStartPos(...);
triangle.AddSeg(new Smoother(new Line(...)));
triangle.AddSeg(new Smoother(new Line(...)));
triangle.AddSeg(new Line(...));
```

UNIVERSITÄT SALZBURG

Basic conditions for the application of the Decorator construction principle (I)

- The signature of H, which is the root class of the subtrees, is not to be extended by the subclasses of H. The reason for this is that additional methods in the subclasses cannot be considered into the Decorator.
- In order to guarantee fulfillment of this demand, it is necessary to transfer the common aspects of all subclasses of H into the root class. This requirement is not satisfied by many class libraries. In such cases, the application of the Decorator construction principle is not possible to full extent (see Decorator Smoother).

UNIVERSITÄT SALZBURG

Basic conditions for the application of the Decorator construction principle (II)

In our example, some specific methods for the mentioned objects must be explicitly called – for example, SetDirection() in Circle - since they cannot be passed on by a Smoother instance:

```
Circle circle= new Circle (...);
circle.SetDirection (cRight);
Smoother smoother= new Smoother (circle);
```

If the mentioned demand would be fulfilled, a Smoother instance could be treated like each specific FlightSegment object:

```
Smoother smoother= new Smoother (new Circle (...));
smoother.SetDirection (cRight);
```

A possibility of eliminating the flight-segment-specific methods is to let all characteristics be indicated only over the constructor of the respective class:

```
Smoother smoother= new Smoother (new Circle (…, cRight));
```

UNIVERSITÄT SALZBURG

Application of the Decorator principle to the design of more “lightweight” root classes

- The Decorator construction principle can be used to make classes close of the root of the class tree more lightweight. Functionality that is not needed in all classes is implemented in Decorator classes. Only the objects which need this special functionality receive it by composition with the appropriate Decorator instance.

- The Decorator construction principle can be fruitfully used both with the (first) design of a class hierarchy and with the extensions of class hierarchies.

Example: Clipping Mechanism in GUI libraries

- Clipping mechanism: cutting a GUI element to its fixed size
- Since the Clipping mechanism is not needed for all GUI elements, it is meaningful to plan the Clipping mechanism by a decorator class Clipper rather than in the root of the subtree (see the Decorator example in Gamma et al., 1995).

UNIVERSITÄT SALZBURG

Summary *Decorator*

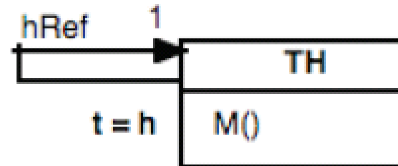
- \+ Simple adaptation by object composition
- \+ New decorator elements (Template classes, which are subclasses of the Hook class) can be defined, without having to change the subclasses of the Hook class.
- \+ „More lightweight “classes can be realized elegantly
- \- The Hook class should fulfill the mentioned basic condition (factoring in behavior from all subclasses)
- \- Additional indirection in method calls
- \- Complex interactions between involved objects

UNIVERSITÄT SALZBURG

The Chain of Responsibility Construction Principle

UNIVERSITÄT SALZBURG

Chain Of Responsibility (COR)



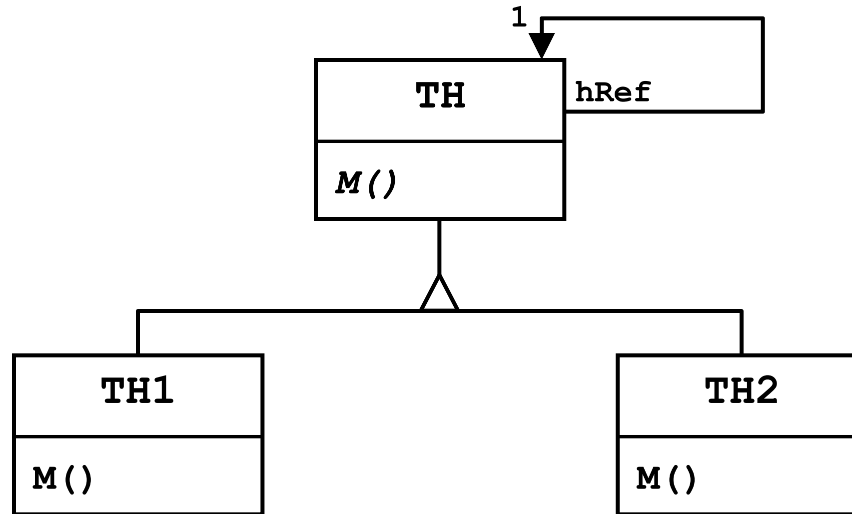
```
public void M(){

...                                    // try to satisfy the request

if (requestSatisfied == true)
   return;
else
   nextTH.M();
}
```

UNIVERSITÄT SALZBURG

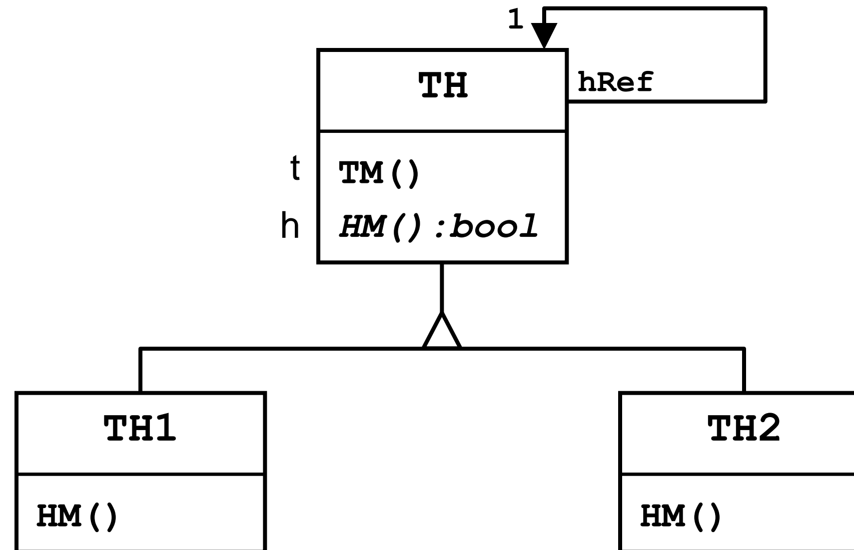
COR by Gamma et al.



- Different implementation of request servicing (the hook part) are provided by subclassing
- The subclasses must also care for the template part!

UNIVERSITÄT SALZBURG

COR With a Separate Hook

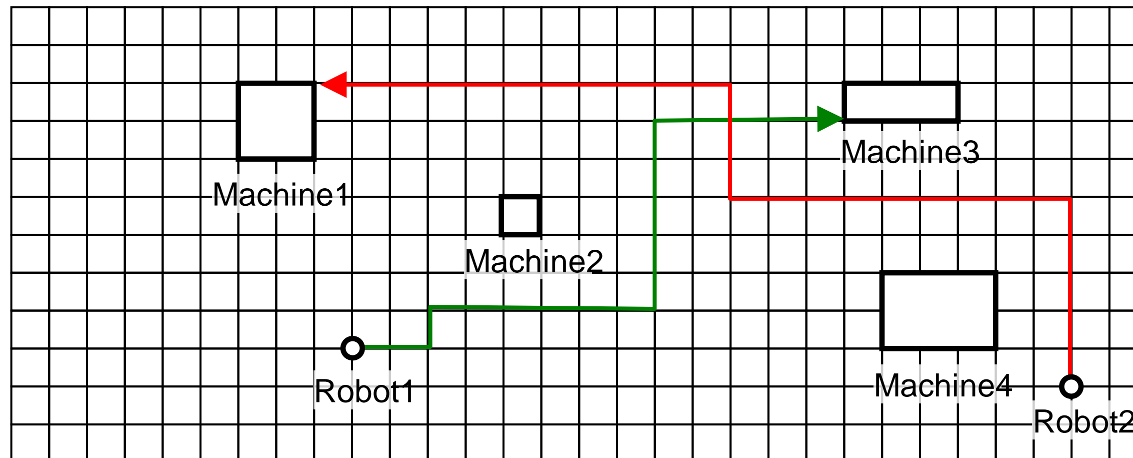



```
public final void TM(){
requestSatisfied = HM();
if (requestSatisfied == true)
   return;
else
   nextTH.TM();
}
```

UNIVERSITÄT SALZBURG

Application: Factory Floor Automation

- Grid layout
- Processing machines
- Mobile robots navigate between machines



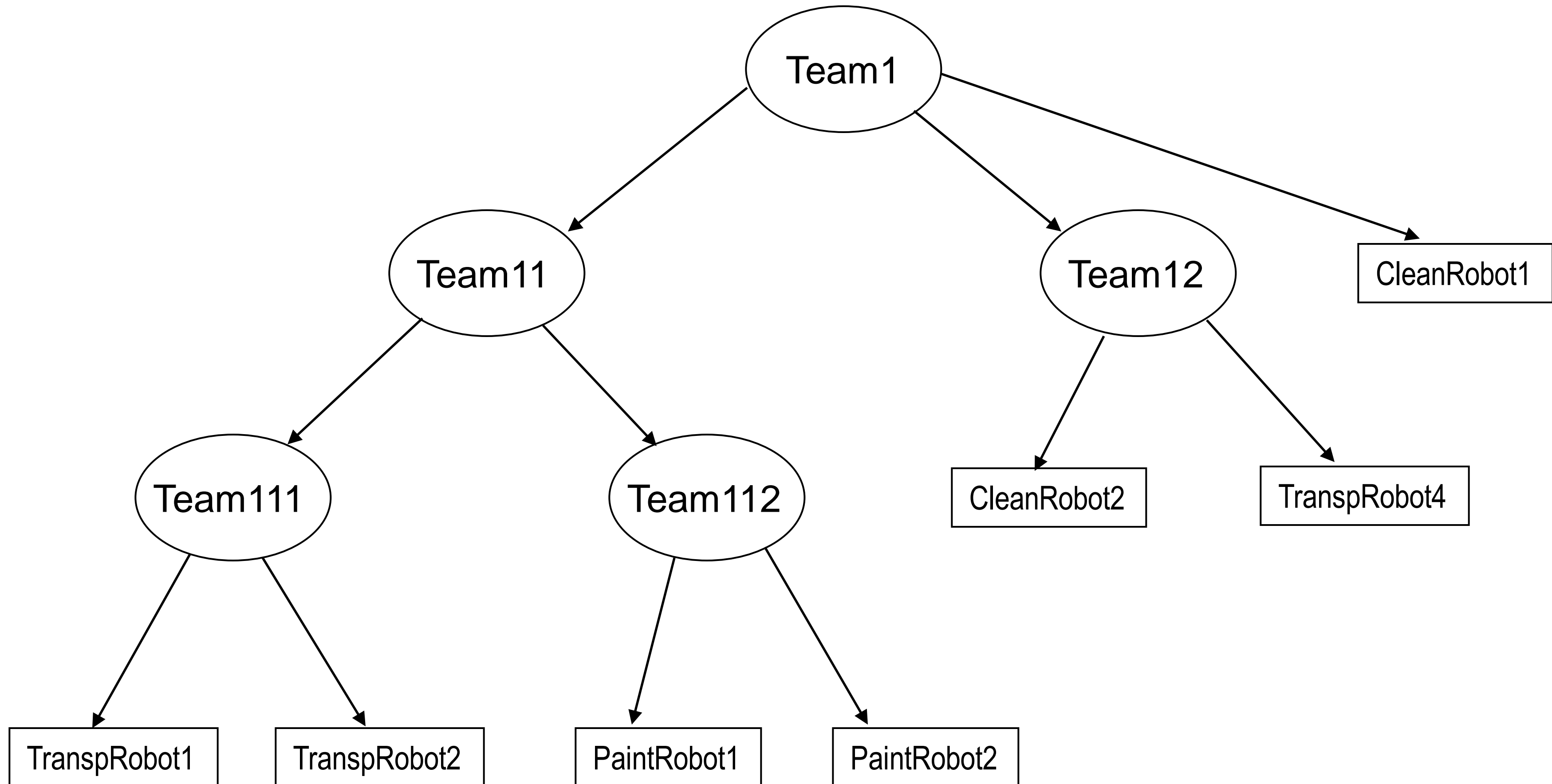
UNIVERSITÄT SALZBURG

Examples of robots

- Transportation robots
- Cleaning robots
- Painting robots
- Teams of robots!

UNIVERSITÄT SALZBURG

Example: COR and Composite



UNIVERSITÄT SALZBURG

Summary of the Characteristics of OO Construction Principles

UNIVERSITÄT SALZBURG

Characteristics of Template and Hook Methods

Features of T() and H()		Construction Principles				
		Hook Method	Hook Object	Composite	Decorator	COR
	Placing	T() and H() in the same class	T() and H() in separate classes			T() = H()
	Naming	T() and H() have different names		T() and H() have the same name		
	Inheritance	n.a.		H() inherits from T()		T() = H()

UNIVERSITÄT SALZBURG

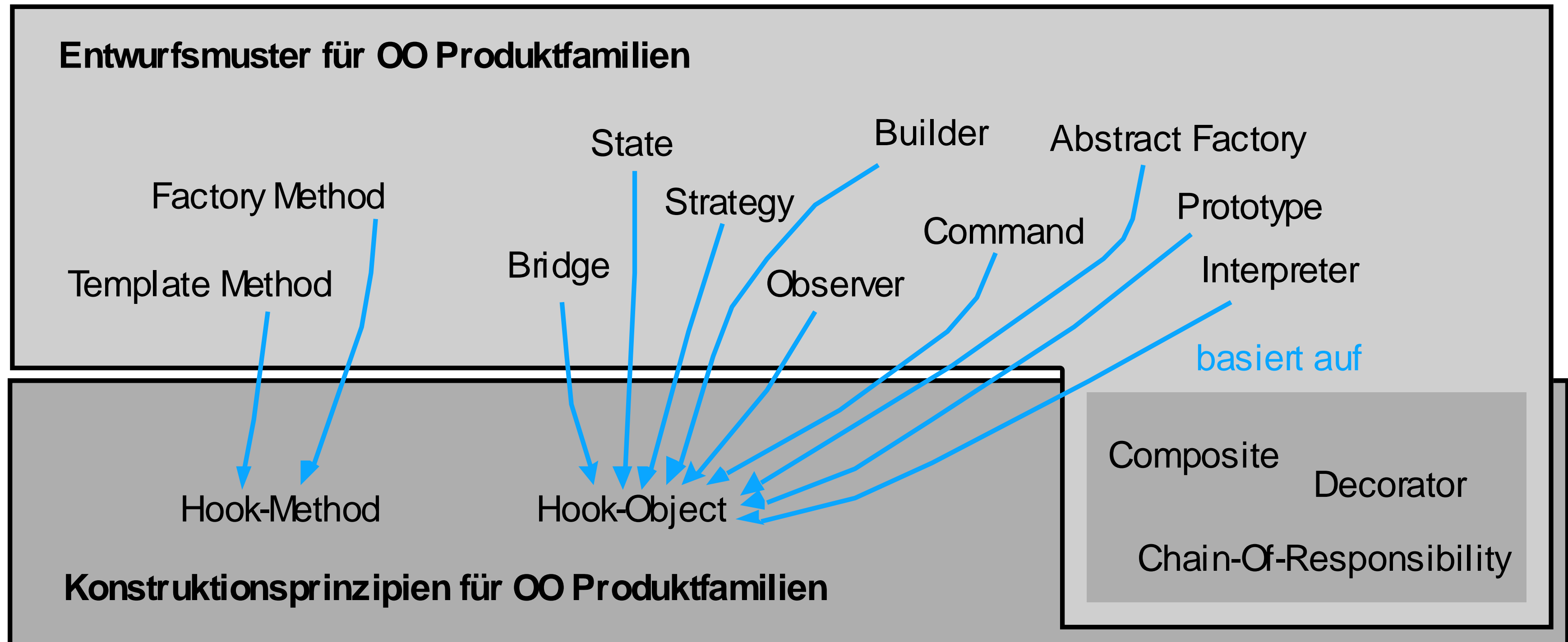
Adaptability

<table>
<tr><th></th><th colspan="5">Construction Principles</th></tr>
<tr><th></th><th>Hook Method</th><th>Hook Object</th><th>Composite</th><th>Decorator</th><th>COR</th></tr>
<tr><td>Number of involved objects</td><td>1</td><td>1(T) + 1(H)
or
1(T) + N(H)</td><td colspan="3">N objects which are used in the same way as a single object</td></tr>
<tr><td>Adaptability</td><td>By inheritance and instantiation of the corresponding class</td><td colspan="4">By composition
(at runtime, if necessary)</td></tr>
</table>

Construction Principles and Design Patterns

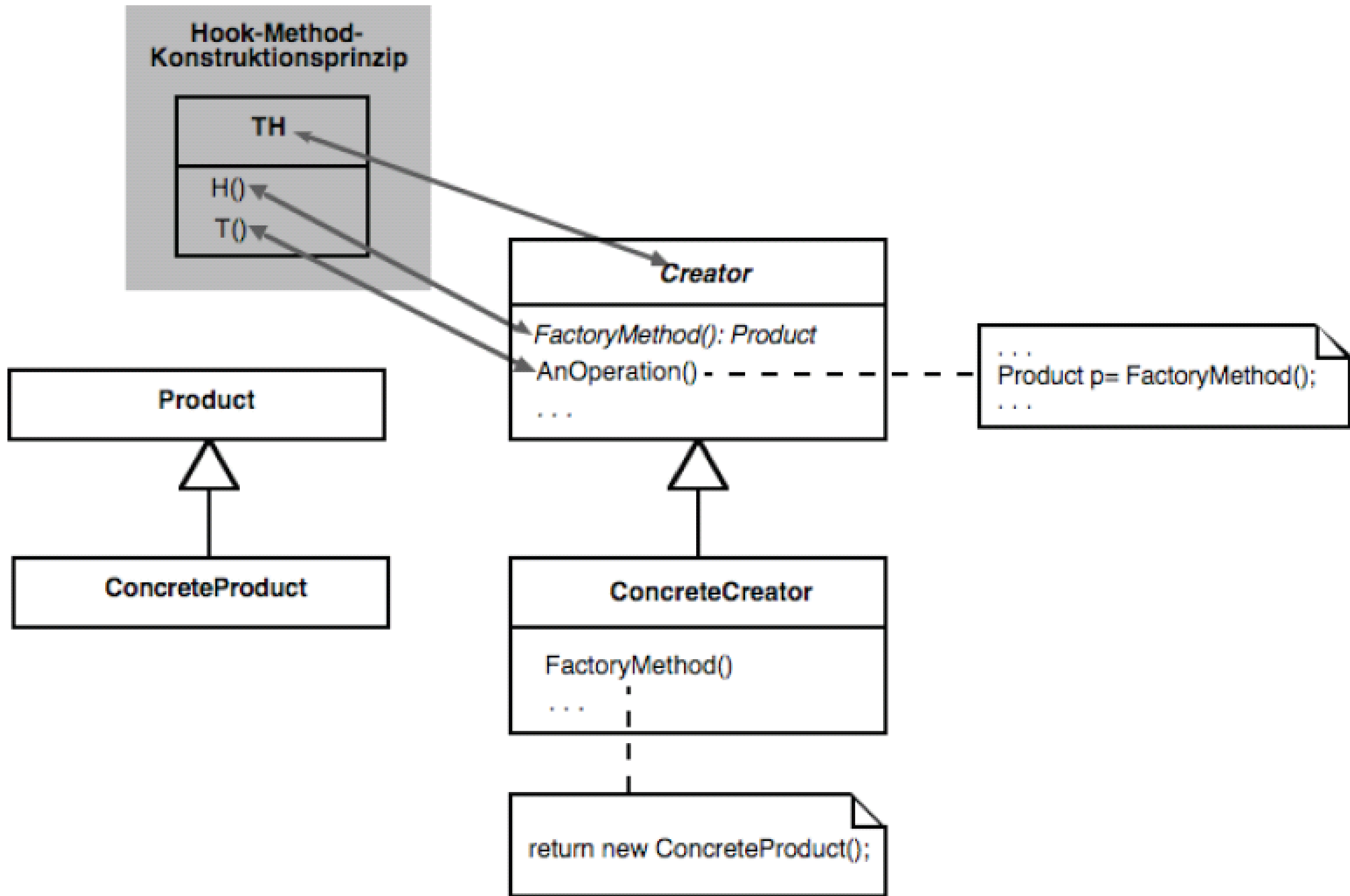
UNIVERSITÄT SALZBURG

14 out of the 23 Design Patterns from Gamma et al. Refer to OO Product Families



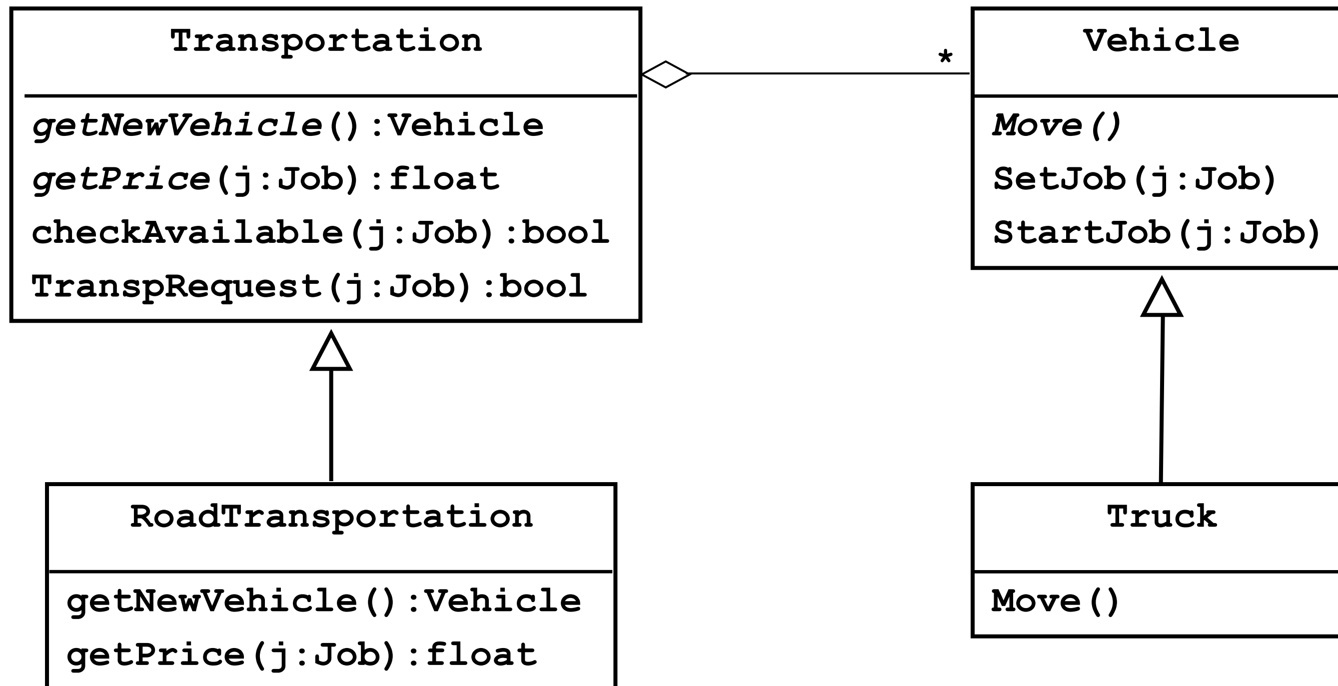
UNIVERSITÄT SALZBURG

Template and Hook Methods in the Factory Method Design Pattern



UNIVERSITÄT SALZBURG

Factory Method Example



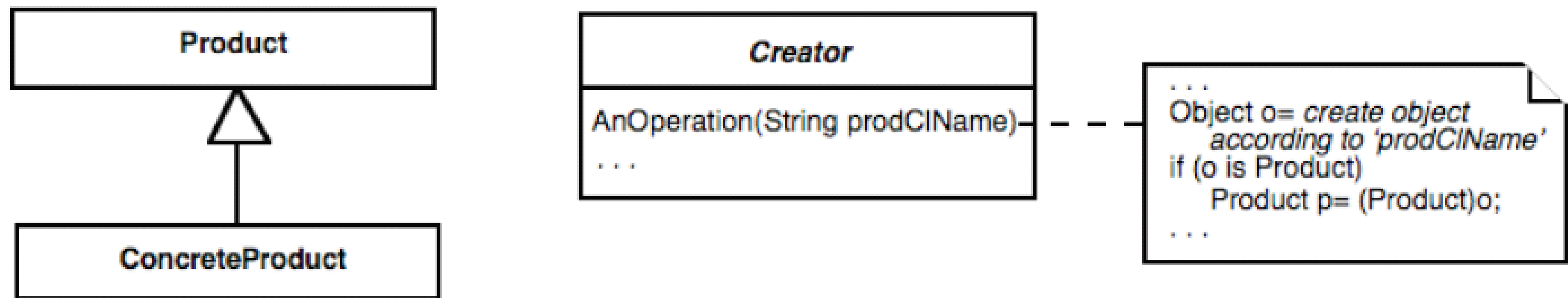
UNIVERSITÄT SALZBURG

Semantics of the Hook method/class is the basis for the naming in Design Patterns

- The name and the functionality of the Hook method and/or the Hook class express which aspect is kept flexible in a design pattern.
- In the **Factory Method** the object production is kept flexible.
- The same applies to the design patterns Abstract Factory, State, Strategy, Builder, Observer, Command, Prototype and Interpreter.
- This kind of the naming is meaningful and therefore it is recommended in the development of new design patterns. We postulate the following rule: Hook semantics determines the name of the design pattern. This enables a systematical designation of DPs.

UNIVERSITÄT SALZBURG

Flexible Object Production Based on Meta-Information (e.g. in Java and C#)

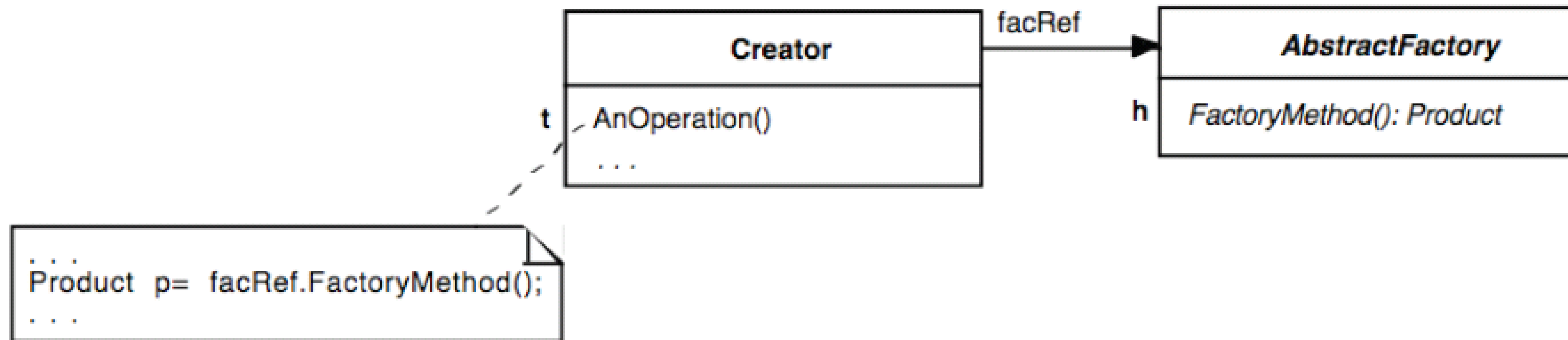


+ No subclassing necessary

- Static type checking is bypassed

UNIVERSITÄT SALZBURG

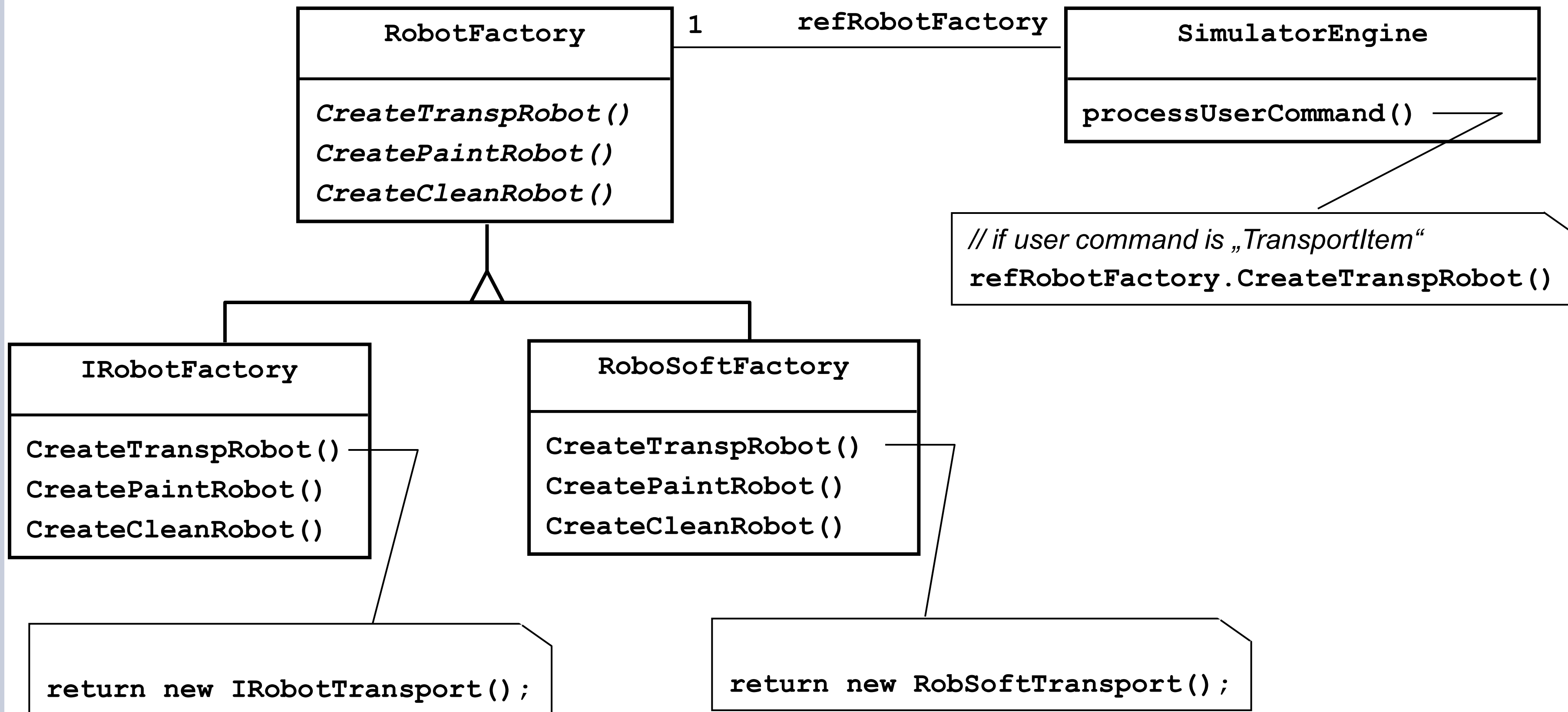
Factory Method (Hook Method) → Abstract Factory (Hook Object)



- The Hook method FactoryMethod () is simply shifted in a separate class or interface

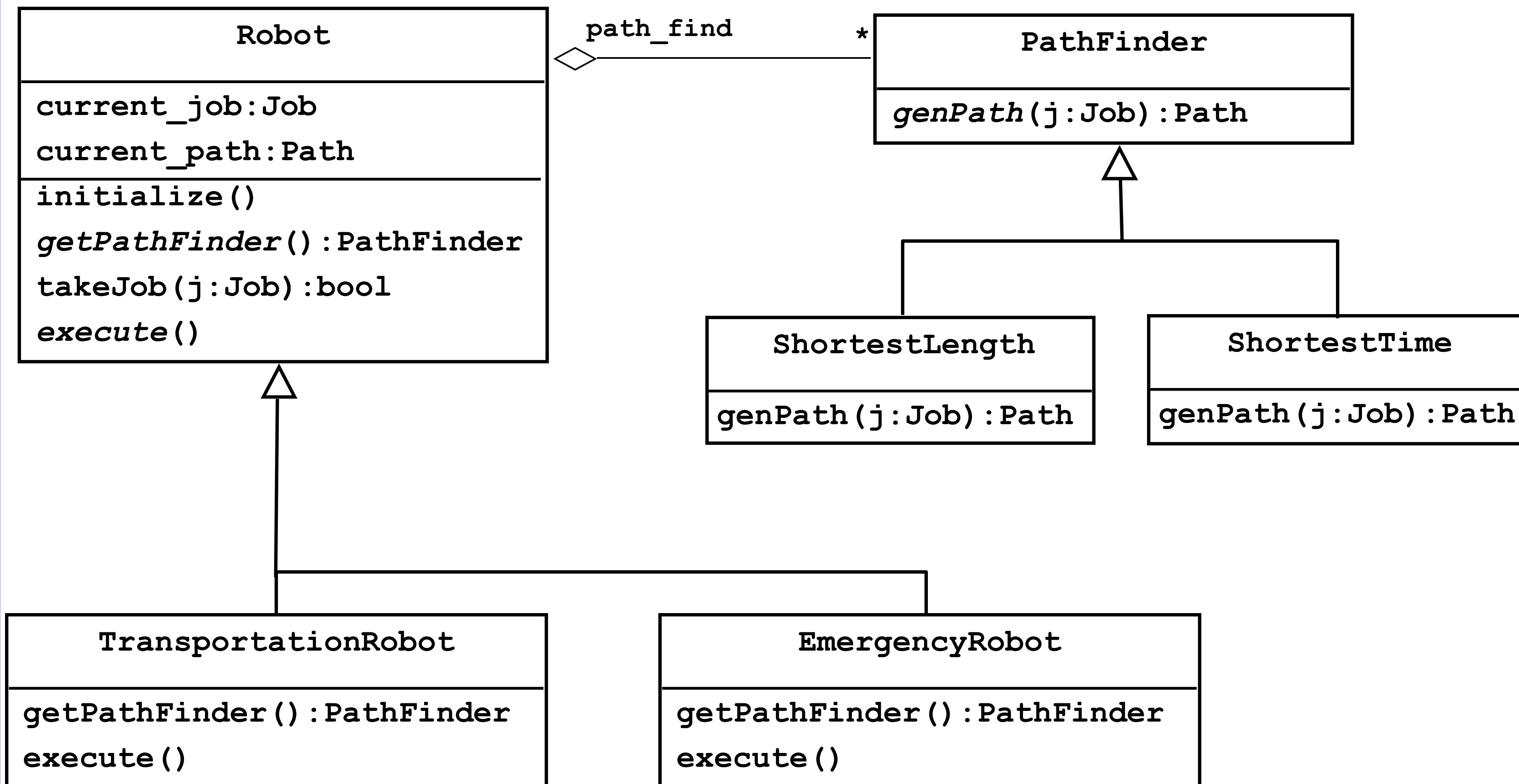
UNIVERSITÄT SALZBURG

Abstract Factory Example



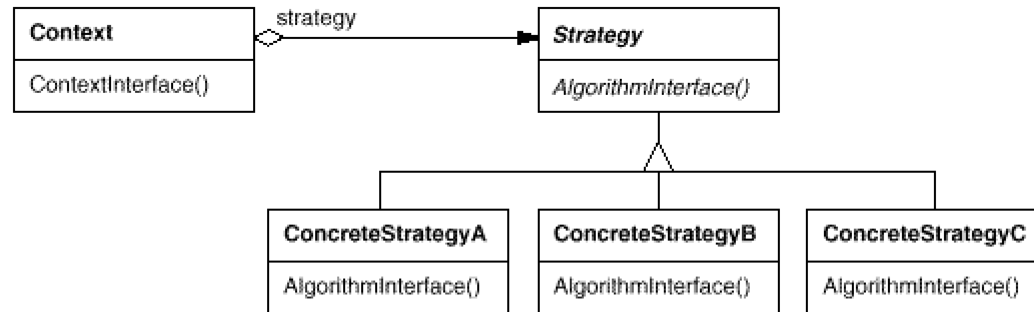
UNIVERSITÄT SALZBURG

The Strategy Design Pattern: Example



UNIVERSITÄT SALZBURG

The Strategy Design Pattern: Structure

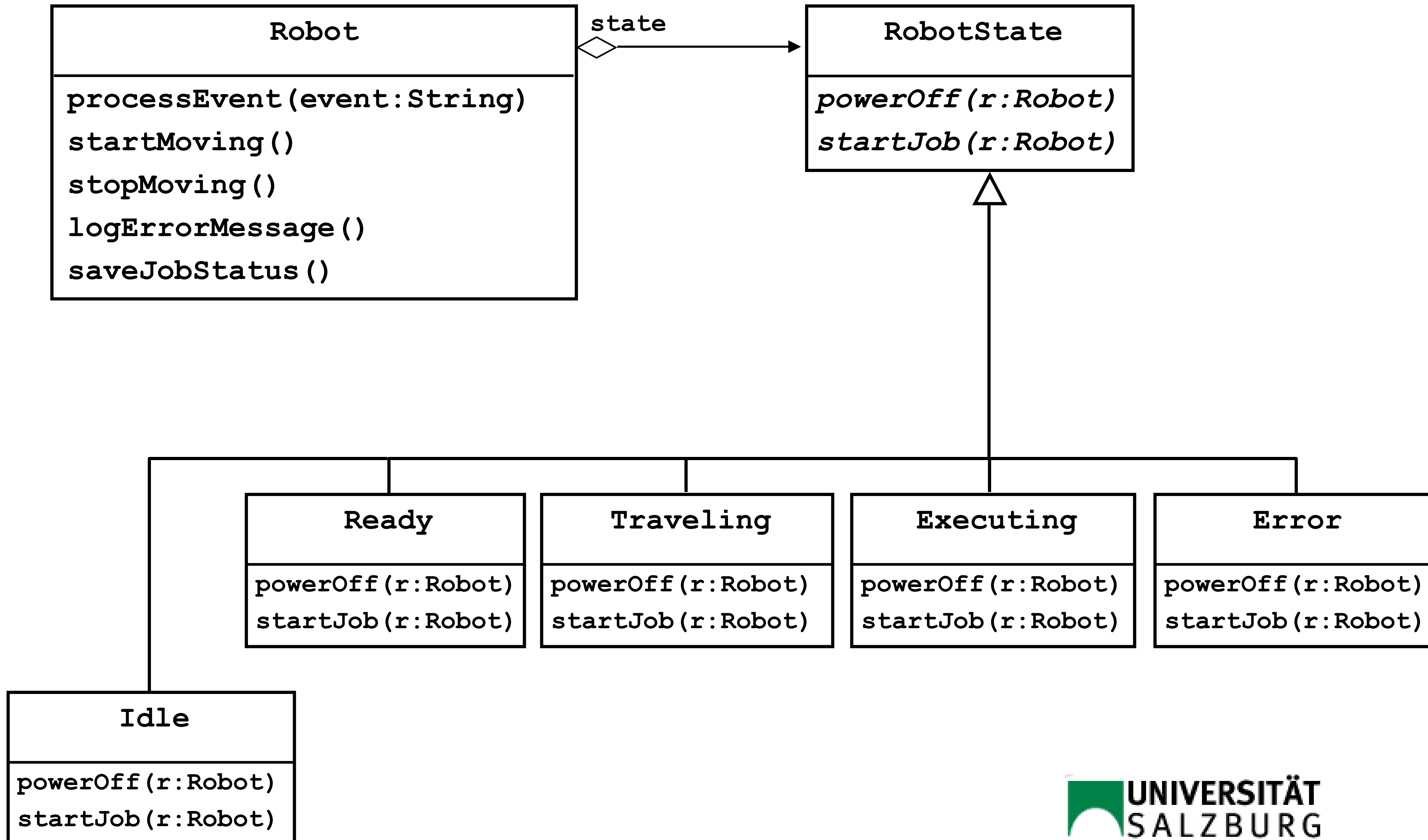


Used when

- A family of algorithms is needed
- A behavior is selected from a given set of behaviors by multiple conditional statements

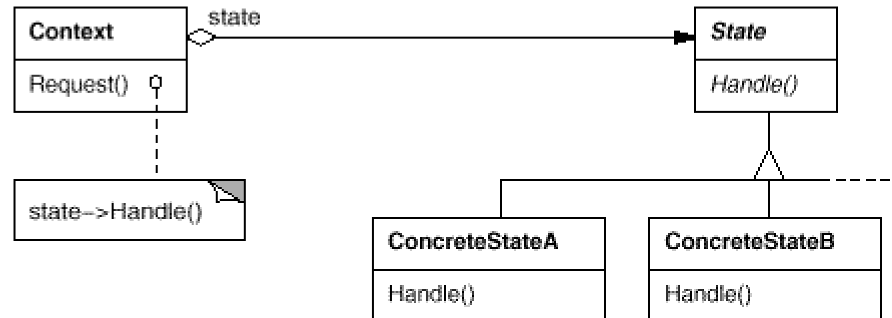
UNIVERSITÄT SALZBURG

The State Design Pattern: Example



UNIVERSITÄT SALZBURG

The State Design Pattern: Structure



Use:

- To implement state transition logic of the type *event[condition]/action* without large conditional statements
- When the same event may occur in different states with different conditions or actions