State Diagram: is this example ok?
The Decorator Construction Principle
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?
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
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.
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.
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 just like any H-object. However, the behavior of the H-object is modified accordingly.
- H + decorator(s) can be used as a single object (see Composite).
Design suggestion when using several Decorator classes
Exemples of Compositions
Example: Smoothing Flight Patterns
Use of the Decorator Class Smoother

```java
FlightPattern triangle = new FlightPattern();
triangle.SetStartPos(...);
triangle.AddSeg(new Smoother(new Line(...)));
triangle.AddSeg(new Smoother(new Line(...)));
triangle.AddSeg(new Line(...));
```
Basic conditions for the application of the Decorator construction principle (I)

- The signature of H, which is the root class of the subtrees, should not 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).
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:

```java
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:

```java
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:

```java
Smoother smoother = new Smoother (new Circle (...), cRight));
```
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).
Summary *Decorator*

- Simple adaptation by object composition
- New decorator elements can be defined, without having to change existing 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
The Chain of Responsibility Construction Principle
public void M() {
    ...
    // try to satisfy the request
    if (requestSatisfied == true) {
        return;
    } else {
        nextTH.M();
    }
}
Different implementation of request servicing (the hook part) are provided by subclassing

The subclasses must also care for the template part!
public final void TM(){
    requestSatisfied = HM();
    if (requestSatisfied == true)
        return;
    else
        nextTH.TM();
}
Application: Factory Floor Automation

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

- Transportation robots
- Cleaning robots
- Painting robots
- Teams of robots!
Example: COR and Composite

Team1

Team11

Team111

Team112

Team12

CleanRobot1

CleanRobot2

TranspRobot4

TranspRobot1

TranspRobot2

PaintRobot1

PaintRobot2
Summary of the Characteristics of OO Construction Principles
### Characteristics of Template and Hook Methods

<table>
<thead>
<tr>
<th>Features of T() and H()</th>
<th>Construction Principles</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Hook Method</td>
</tr>
<tr>
<td>Placing</td>
<td>T() and H() in the same class</td>
</tr>
<tr>
<td>Naming</td>
<td>T() and H() have different names</td>
</tr>
<tr>
<td>Inheritance</td>
<td>n.a.</td>
</tr>
</tbody>
</table>
## Adaptability

### Construction Principles

<table>
<thead>
<tr>
<th>Number of involved objects</th>
<th>Hook Method</th>
<th>Hook Object</th>
<th>Composite</th>
<th>Decorator</th>
<th>COR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1(T) + 1(H) or 1(T) + N(H)</td>
<td>N objects which are used in the same way as a single object</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Adaptability</th>
</tr>
</thead>
<tbody>
<tr>
<td>By inheritance and instantiation of the corresponding subclass</td>
</tr>
</tbody>
</table>
Construction Principles and Design Patterns
14 out of the 23 Design Patterns from Gamma et al. Refer to OO Product Families

Entwurfsmuster für OO Produktfamilien

Konstruktionsprinzipien für OO Produktfamilien

- Factory Method
- Template Method
- State
- Strategy
- Builder
- Abstract Factory
- Command
- Observer
- Interpreter
- Prototype
- Hook-Method
- Hook-Object
- Composite
- Decorator
- Chain-Of-Responsibility
Template and Hook Methods in the Factory Method Design Pattern

```
Hook-Methods
Konstruktionsprinzip
```

```
TH

H()

T()
```

```
Creator

FactoryMethod(): Product

AnOperation()

...
```

```
Product

ConcreteProduct
```

```
ConcreteCreator

FactoryMethod()

...

return new ConcreteProduct();
```

```
Product p = FactoryMethod();
...
```
Factory Method Example

Transportation
- `getNewVehicle()`: Vehicle
- `getPrice(j: Job): float`
- `checkAvailable(j: Job): bool`
- `TranspRequest(j: Job): bool`

Vehicle
- `Move()`
- `SetJob(j: Job)`
- `StartJob(j: Job)`

RoadTransportation
- `getNewVehicle()`: Vehicle
- `getPrice(j: Job): float`

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

+ No subclassing necessary
- Static type checking is bypassed
The Hook method `FactoryMethod()` is simply shifted in a separate class or interface.
Abstract Factory Example

RobotFactory
- CreateTranspRobot()
- CreatePaintRobot()
- CreateCleanRobot()

IRobotFactory
- CreateTranspRobot()
- CreatePaintRobot()
- CreateCleanRobot()

RoboSoftFactory
- CreateTranspRobot()
- CreatePaintRobot()
- CreateCleanRobot()

return new IRobotTransport();

refRobotFactory

SimulatorEngine
- processUserCommand()

// if user command is "TransportItem"
refRobotFactory.CreateTranspRobot()

return new RoboSoftTransport();
The Strategy Design Pattern: Structure

Used when

- A family of algorithms is needed
- A behavior is selected from a given set of behaviors at runtime
The Strategy Design Pattern: Example

```java
Robot

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>target: Point</td>
</tr>
<tr>
<td>currLocation: Point</td>
</tr>
<tr>
<td>initialize()</td>
</tr>
<tr>
<td>takeJob(j:Job):bool</td>
</tr>
<tr>
<td>setOpMode(om:OPMODE)</td>
</tr>
<tr>
<td>execute()</td>
</tr>
</tbody>
</table>

LocationCalculator

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>getNextLocation(crLoc): Point</td>
</tr>
</tbody>
</table>

ShortestLength

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>genPath(j:Job):Path</td>
</tr>
</tbody>
</table>

ShortestTime

<table>
<thead>
<tr>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>genPath(j:Job):Path</td>
</tr>
</tbody>
</table>

if (om.equals(OPMODE.NORMAL))
    nextLocCalc = new ShortestLength();
else if (om.equals(OPMODE.EMERGENCY))
    nextLocCalc = new ShortestTime();

nextLocCalc.getNextLocation(currLocation);
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
The State Design Pattern: Example

Robot

- processEvent(event:String)
- startMoving()
- stopMoving()
- logErrorMessage()
- saveJobStatus()

RobotState

- powerOff(r:Robot)
- startJob(r:Robot)

State

- Ready
  - powerOff(r:Robot)
  - startJob(r:Robot)

- Traveling
  - powerOff(r:Robot)
  - startJob(r:Robot)

- Executing
  - powerOff(r:Robot)
  - startJob(r:Robot)

- Error
  - powerOff(r:Robot)
  - startJob(r:Robot)

Idle

- powerOff(r:Robot)
- startJob(r:Robot)
The Flyweight Design Pattern: Structure

Use:
- To avoid employing a large number of objects with similar state
- When objects with intrinsic state can be re-used by different clients
The Flyweight Design Pattern: Example(I)

Intersection of type A
Intersection of type B
The Flyweight Design Pattern: Example(II)

- New intersection types can be easily added!