Software Design Summary

Short introduction to Java (mainly differences to C++)

Some issues about classes

```
Obj d1;
Obj d2(d1) //💣 No default copy constructor in Java!
Obj d2 = d1 //no copy construction, reference copy
Obj d3 = d1.clone() //copying with clone method inherited from JavaBaseClass
if(d2 == d1) //True. Reference comparison only
if(d3 == d1) //False. Nachprüfen!!
```

- Classes have a default constructor.
- The default constructor is removed, if own constructors are implemented.

Type and implementation of a class

Type	• Specification of methods using *pre- and postconditions*. • Specification of allowed method invocation sequences using *state diagrams*. • Specification of state using *invariants*.
Implementation	• *Fields (aka attributes)* • *Methods* • *Constructors*

Static methods and attributes

Per class, there is always only one instance of static methods and attributes, i.e. all instances of that class share the same static methods or attributes.

	Instance methods and attributes	Class methods and attributes
Declaration	Without static	With static
Exists	In each object	Only once per class
Fields are released	From the garbage collector	At the end of the program
Calling methods and attrs	Object name	Class name

```
class Mouse{
    public Mouse(){noOfMice++; age=12}
    static int noOfMice = 0;//One variable for all instances of class
    static final int noOfCats = 1;//Constant
    static void classInfo(){…} //Same function for all instances
    public int age;
}
Mouse mickey;
System.out.println(Mouse.noOfMice) //1
System.out.println(mickey.age) //12
```

This-constructor

```
class Cat{
    private int age;
    private bool alive;
    public Cat(int age){this.age = age; alive = true;}
    public Cat(Cat c){this(c.age,c.alive)}//Constructor using this
}
```

Inheritance

- Keyword *extends*: `class A extends B {…}`
- Only single inheritance possible
- Viewpoint code inheritance: The inherit has more fields & methods

- Viewpoint interface inheritance: The inherit is a subtype, a specialization

Notions

- Type: In Java, a type is defined by a class.
- Subtype: A subtype is therefore defined by a subclass. A subtype defines a is-a relationship.

Method overriding (aka overwriting)

```
class Mouse{
    public Mouse(int age){this.age = age}
    public void sleep(){*/sleep for 4 hours/*}
}

class Dormouse{
    Public Dormouse(int age){super(age)}//Must provide an own constructor
since there is no default constructor in base class anymore.
    Public void sleep(){*/sleep for 8 hours/*}
}
```

- Do not mistake overriding with overloading!
- The overwritten method of the base class is hidden for the derived class.
- A base class method can be invoked by the keyword *super*
- If no constructor is defined for the derived class, Java tries to invoke the default constructor of the base class (if it exists)

Static type vs. dynamic type

```
class X {
int get(double x){return 0;}
}
class Y extends X {
int get(double x){return 1;}
int get(char y){return 2;}
}
public class Test {
    public static void main(String[] args){
          X xy = new Y();
          System.out.println(xy.get(3.4));//Output: 1
          System.out.println(xy.get('a'));//Error
    }
}
```

- The static type of xy is X. The dynamic type of xy is Y.
- At runtime, methods in X are overwritten by methods in Y which have the same function signature, but no methods are added to X! Since there is no method in X which could handle a char as parameter, xy.get('a') will produce an error.

Exceptions

Example

```
try{
    a=getAccount();
    a.withdraw(x);
    return true;
} catch (LimiOverflowException e){
    System.out.println("Account is over the limit")
}
```

```
void withdraw() throws LimitOverflowException{
    if(balance < 0){
          throw new LimitOverflowException();
    }
}

class LimitOverflowException extends Exception{
    //Implementation of exception
}
```

Checked / unchecked exceptions

- Unchecked exceptions include Exception, RuntimeException and their subclasses. They do not have to be explicitly declared with the throws keyword.
- Checked exceptions have to be declared in a method and must be handled.
- Exceptions are computationally expensive and shouldn't be used for program control.

Exceptions & Polymorphism

In a subclass of a class which throws exceptions,

- Not all exceptions need to be declared by the subclass
- Exceptions may be omitted
- No new exceptions may be declared

Why: According to the Liskov Substitution principle, it must be possible to put a subclass in place for the parent class. If a parent method in a try-block throws exception A, the exception must be caught in a subsequent catch block. If this catch-block handles only exception A, but the subclass implementation of a method in the try-block throws exceptions A and B, the catch-block can't handle exception B, which leads to an error.

Advanced Java Topics

- Covariant typing: Covariant return, means that when one overrides a method, the return type of the overridden method is allowed to be a subtype of the overridden method's return type.

@Override method	Overridden method
Subtype of return type	Return type

- Static initializer: Within a class definition, static blocks, i.e. static{…} are executed during initialization of the class. Usually, it's a good place to define static variables.
- Instance initializer: An *instance initializer* declared in a class is executed when an instance of the class is created
- Order of static initializers: Example (24 May 13, Slide 36)
- Keyword final: Final classes, final methods, final attributes
- <?>: Type wildcard
- Executor, Callable, Future (in relation to Command pattern)
- Dynamic proxy class. (10 May 2013, Slides 38 ff.) Allows

Keywords

- Final: Doesn't allow subclassing, overriding. Final variable can be initialized only once. However, this doesn't guarantee immutability, since members of the final field could still be changed (directly).
- Abstract: Doesn't allow instantiation of class
- Static: Only one instance per class

Visibility in packages

- Private: Access only from within the same class

- Internal (no declaration modifier): Access only from within the same package
- Protected: Access from within the package and from all subclasses (even when declared in other packages)
- Public: Access from everywhere possible.

Liskov substitution principle

If q(x) is a provable attribute of x, where x is of class T, q(y) must also hold , where y is of class S, which is a subclass of T.

Why inheritance is dangerous

Motivating example / problem

```
public class StringOutputStream extends FilterOutputStream {
    public StringOutputStream(OutputStream s){ … }
    void write (char ch) { … }
    void write (String s) { … }
}
public class CountedOutputStream extends StringOutputStream
{
    public CountedOutputStream (OutputStream s){ super(s); }
    private int c = 0;
    public int writtenChars(){ return c; }
    void write (char ch) { c=c+1; super.write(ch); }
}
```

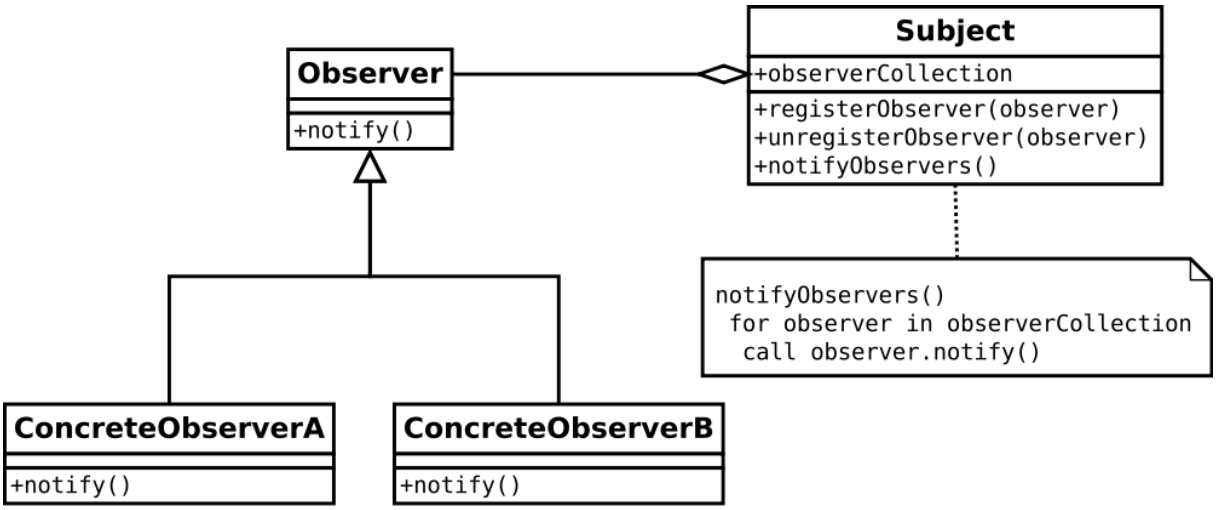
The characters are only counted if write(String s) calls write(char ch) for every character.

- **Knowledge about baseclass needed**: It's important to know which virtual methods are called from other virtual methods. There may be dependencies between virtual methods that must hold otherwise the method doesn't work properly.
- **Encapsulation is broken**: Need for knowing implementation details of base class breaks encapsulation.

Solutions

- A robust variant to inheritance is **forwarding** (using Decorator or Composite pattern). In contrast to inheritance base class code is not "replaced" but effectively used. ➔ No reuse, but use.
- **Template pattern**: Provide extension points, but don't touch code on which other methods depend.
- Policy: Non-private, non-static subclassable methods must either be
    - Abstract
    - Final
    - Have an empty implementation

Observer pattern



Concrete example

Subject (aka Observable)	Temperature
ConcreteObserverA	A button which raises the temp.
ConcreteObserverB	A button which lowers the temp.

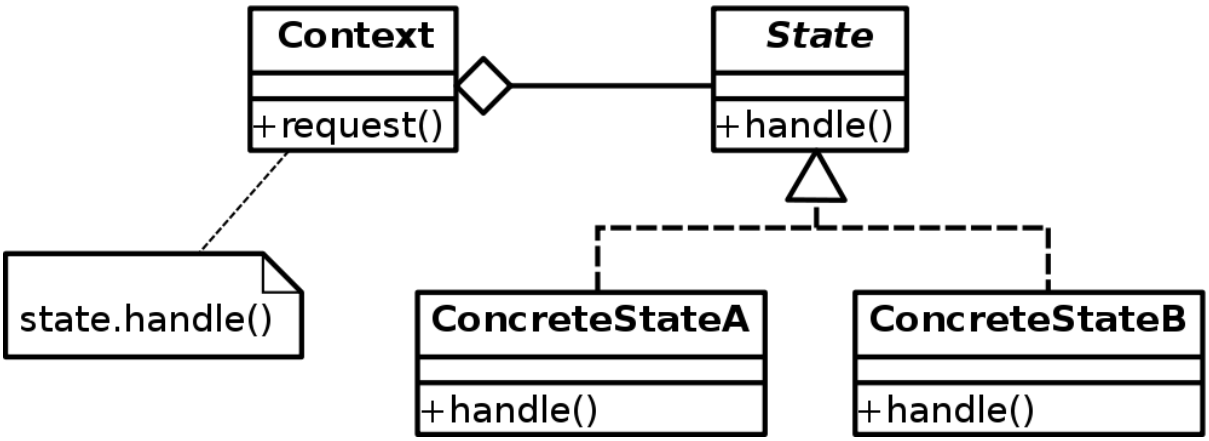
Issues

- Subject only knows Observer interface, no concrete observers
- Notification is broadcast, works for an unlimited number of observers
- Several subjects are possible (TemperatureSensor, LightSensor, etc.)
- If a subject belongs to multiple observers, it needs to be passed as an argument to the update function. Concrete observers might have different implementations for different observers (State pattern)
- Push / Pull model: In the push model, the subject sends all relevant data to the update function of the concrete observer. In the pull model, the concrete observers have a reference to the subject, thereby accessing its data.

Disadvantages

- A simple operation on a subject may cause a cascade of updates
  Solutions: Asynchronous update at idle time, explicitly after a set of change operations
- For N observables and M observers, the situation can become quite complicated, as every observable must be connected with every observer.
  Solution: Mediator. The mediator handles the communication with the observers. Observables only know the mediator. The mediator is a subclass of Observable and implements the Observer interface.
- Cyclic dependencies: If there are cyclic dependencies in the notify chain, a cycle break must be implemented. => Check if state has really changed.
- Causality: How to make sure that the order of state changes is preserved. Example: 12. April, Slide 23
  Solutions: Queuing of notifications, Queuing of state changes, enumerate state changes
- Memory management: If for example windows are stored as observers, they need to be properly deleted If not needed anymore. Even if windows are disposed, a reference of them is still saved in the observer and therefore the resources are not freed.

State pattern



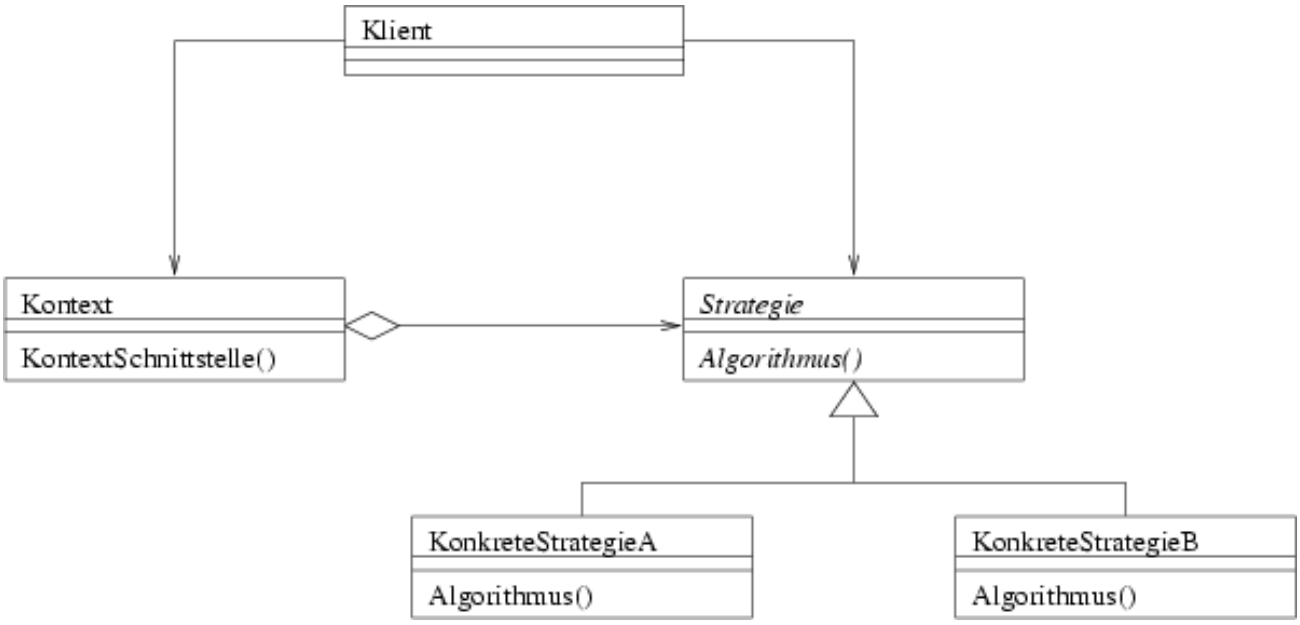
Concrete example

Context	JDraw
State	Abstract interface with a function onMouseMove
ConcreteStateA	Grid with spacing 20
ConcreteStateB	Grid with spacing 50

Advantages

- Avoid conditional statements: The state pattern helps to avoid long if else statements or switch statements, polluting the code.
- State transitions are atomic: Since a state transition involves the change of only one member variable, the system is always in a well-defined state.

Strategy pattern



Advantages

- Alternative to subclassing: Inheritance mixes context with behavior => bad
- Strategies eliminate conditional statements

Disadvantages

- Communication overhead between Strategy and Context: The strategy interface is shared by all strategies no matter whether or not they use the information. One strategy might need many parameters, while the other doesn't need any of them. (E.g. fully 3d schemes versus splitting schemes).
- Different strategies might not be entirely different => Code duplication. Mitigation: Partial subclassing, Template pattern (i.e. factorization of common code)

When to use a strategy pattern

- Context class uses different variants of algorithm
- Context can deal with new implementations of algorithm
- To avoid multiple conditional statements

When not to use a strategy pattern

- Only one algorithm which depends on parameters (Grid size 10/20/50)

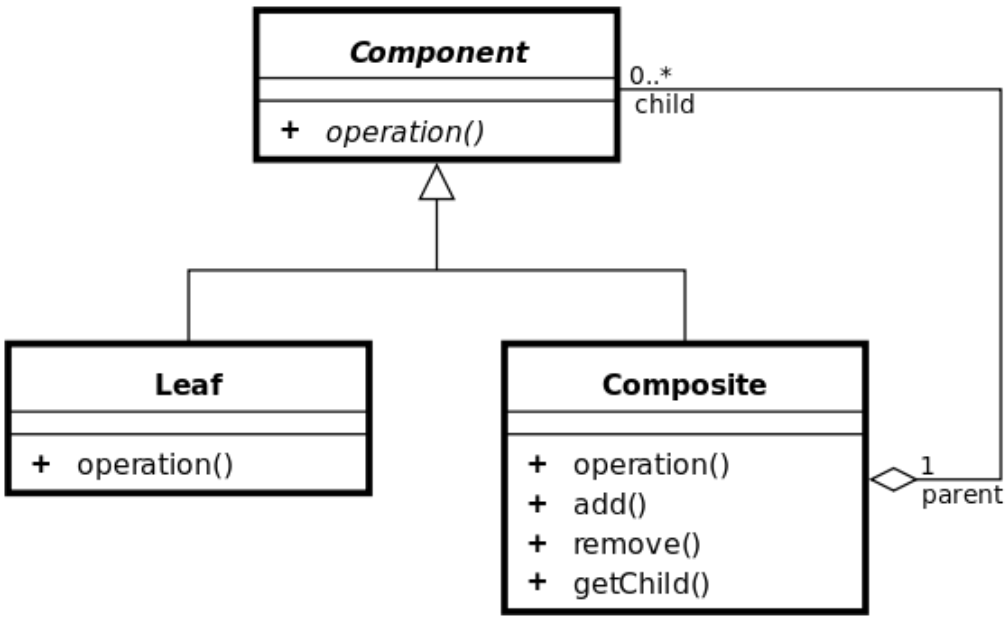
Differences between State and Strategy pattern

- States store a reference to the context object that contains them. Strategies do not.
- States are allowed to replace themselves (IE: to change the state of the context object to another state), while Strategies are not.
- Strategies are passed to the context object as parameters, while States are created by the context object itself.
- Strategies only handle a single, specific task, while States provide the underlying implementation for everything (or most everything) the context object does.

Null Object pattern

The null object is a dummy class which represents no strategy. Its methods are empty. The Null object could be used as a base class for other implementations.

Composite pattern



Composite may consist of leaves and other composites.

Composite pattern allows treating composite objects equally to leaves.

Specific problems: Sometimes it is difficult to find a general description of a method. E.g. resizing a line using a bounding box has the problem that there are two possibilities to place a line into the same bounding box.

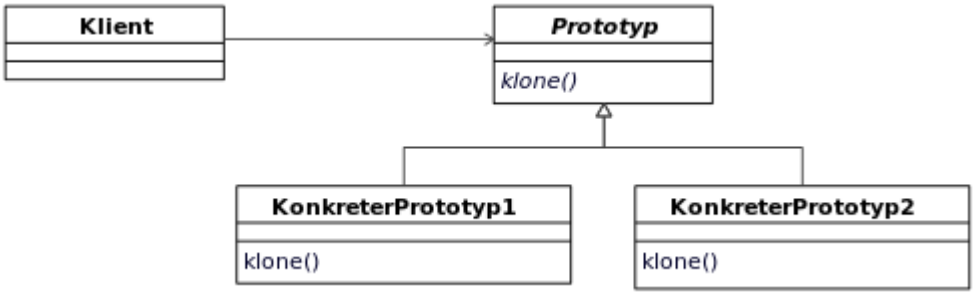
Composite has additional methods. If placed in Composite then Composites and leaves must be handled separately. If placed in Component, there must be a dummy implementation in Leaf.

Composition is only allowed if the structure is directed acyclic graph. Cycles are prohibited! Moreover, a leaf should not belong to multiple composites.

Model-View-Controller pattern

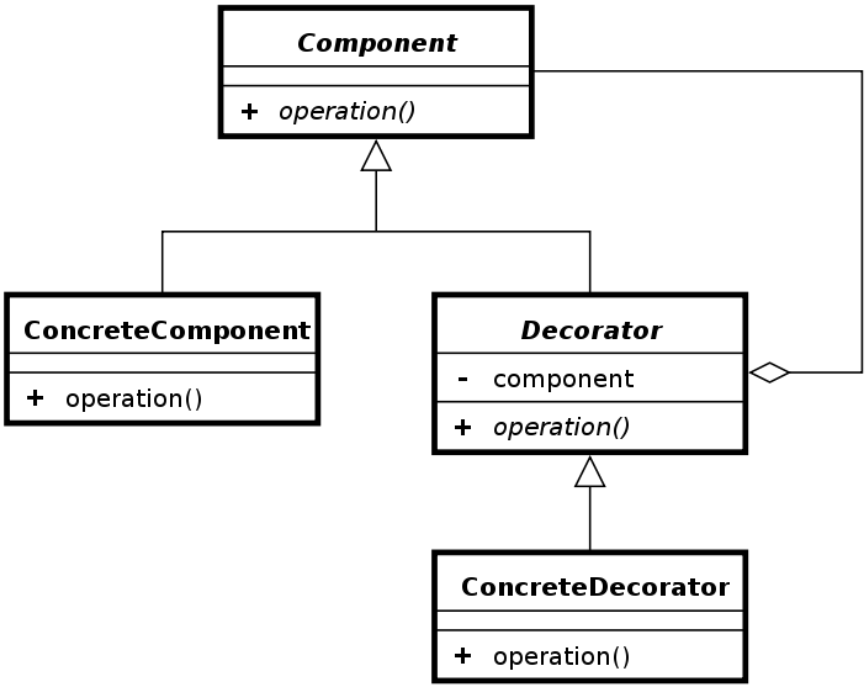
- Model: Representation of application data & business logic
- View: Presentation of the data (multiple views are possible) and the source of user interaction
- Controller: Controls and mediates input, forwards it to the model

Prototype pattern



- In Java: Realized with Object.clone(). Object.clone() creates a new instance without a constructor invocation. All attributes are copied (member-copy). Object.clone() throws an exception if the calling class (every class is a subclass of Object in Java) does not implement the Cloneable interface.
- A shallow copy is Object.clone with a cast: (MyClass) Object.clone()
- Immutable object must not be cloned, as they cannot be changed
- Final fields cannot be changed in the clone method
- Initialization of clones: When clones need to be initialized with special parameters, this cannot be done in the clone method in general, since this would preclude a common clone method in the interface.
- Main liability: Implementing the clone method
- **Alias references**: Alias references and cycles have to be handled manually when implementing a deep-copy (Slide 21, 2 May 2013)
- **Information hiding**: Return values that are of primitive type are protected from being changed. But any other returned object can be altered because the objects are returned by reference instead of a read-only copy. Therefore, mutable input parameters must be copied before they are returned.
- **Immutable objects**: Consistent, thread safe, safe in the presence of ill-behaving code
  Must be final, no setter methods, any return types must be primitive, immutable or deep copy

Decorator pattern



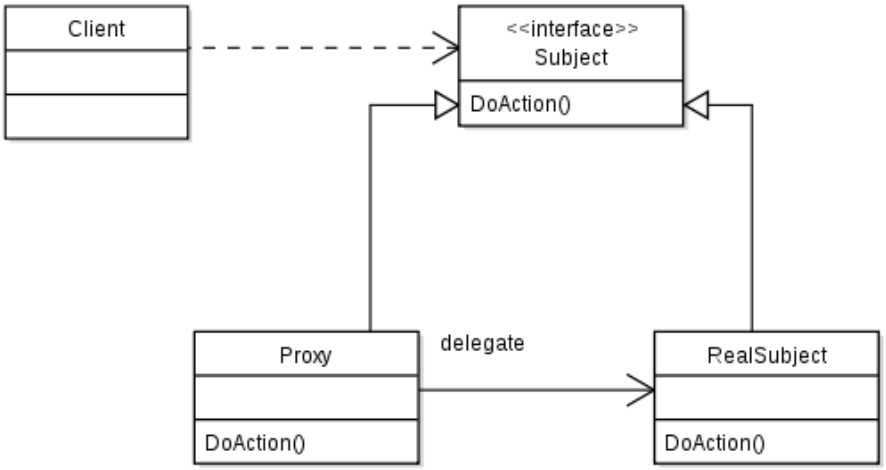
Advantages

- More flexibility than static inheritance: Decorators can be attached and detached at runtime. It's difficult to realize such flexibility with inheritance.
- Efficiency: Multiple decorators can be added in a simple way. With inheritance, every combination on would have to be hard-coded, which leads to a bloated code. For n features, decorator needs n+1 additional classes whereas inheritance needs 2^n-1 additional classes.

Disadvantages

- Decorator has not same type as the decorated component. Ex: Decorated group cannot be ungrouped as it is not a group. Solution: instanceof operator
- Lots of little very similar objects.
- Methods called in decorated class are not executed in decorator. Fix(Animate(Figure)) leads to an animated (moving) figure, but which is not moveable with the cursor. In an implementation with inheritance, the animator's use of move() would be blocked by the fixator's override of move()

Proxy pattern



Motivation

Provide a placeholder for computational costly objects, and construct these objects only when needed. Provide a "gate keeper" to control access to the original object. Copy-on-write: File is only copied, if it was written to.

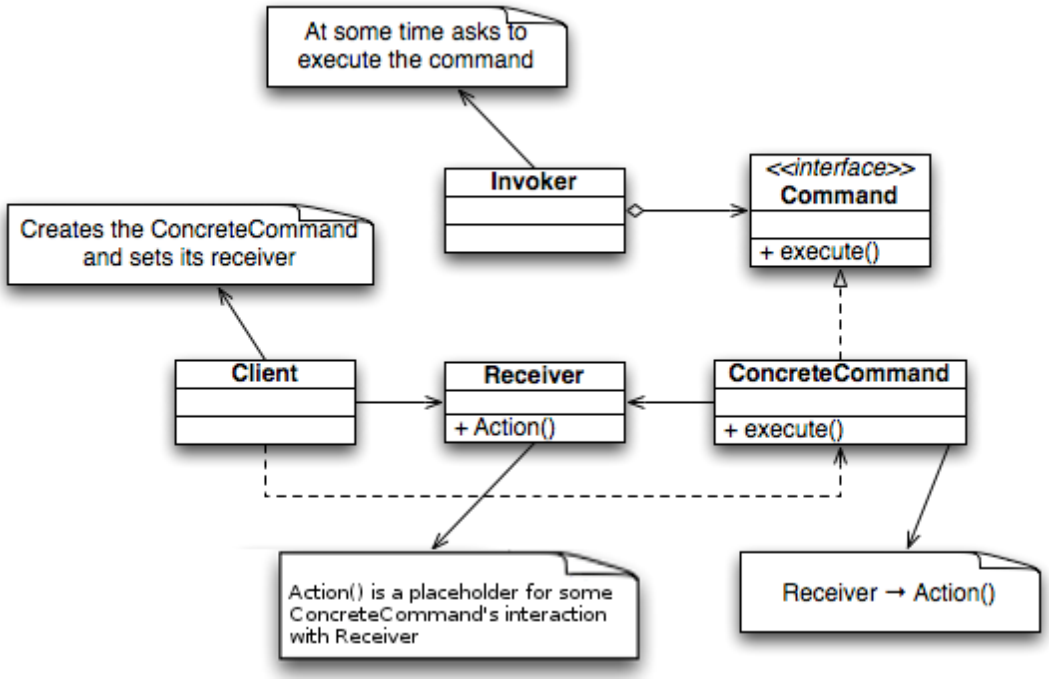
Applicability

- Remote proxy: Provides a local representative for an object in a different address space
- Virtual proxy: Creates expensive objects on demand
- Protection proxy: Controls access to the original object.
- Smart reference: Performs additional action when original object is accessed.
- Cache proxy: Caches already returned results

Proxy vs Decorator

- Proxy is not recursive
- Proxy control access, decorator adds new responsibilities

Command pattern



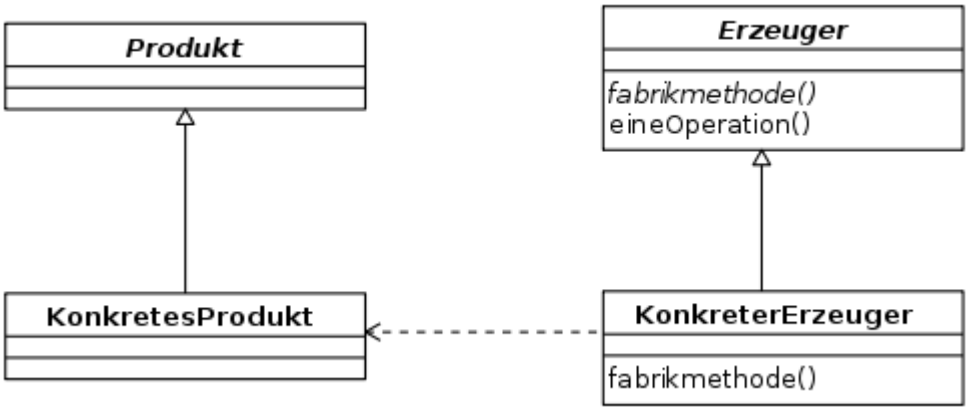
Motivation

- Undo / Redo, transactions
- Copy and save the changes rather than the states. Usually, less data has to be stored for changes than for states.
- Changes (undo, redo parameters) are stored in the ConcreteCommand object. The Command interface provides two methods: Undo and Redo
- To implement undo / redo, the invoker needs two stacks one for the undo commands, the other for the redo commands.

Advantages

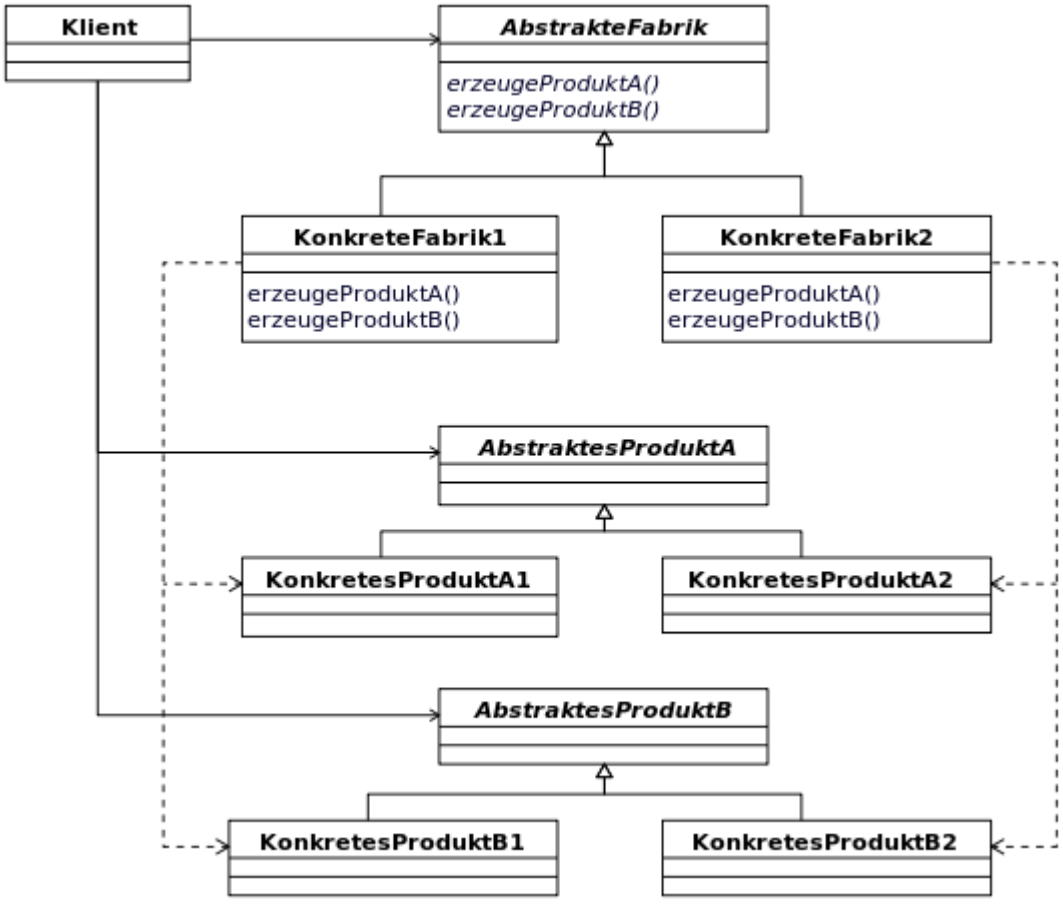
- Commands can be assembled into a composite command
- Easy to add new commands

Factory method



- Creational patterns abstract the object instantiation process, i.e. they hide how objects are created (with new)
- Make program code independent of concrete classes
- Exchanging the created class is demanding and error-prone
- Factory method: Delegates instantiation to subclass
  Abstract factory: Delegates instantiation to another object
- Typically used in the context of parallel class hierarchies
- Example: Abstract restaurant serves abstract dishes. Subclasses: Pizzeria (factory aka creator) serves Pizza (product), Sushi bar serves Sushi, KFC serves fried chicken etc.
  http://de.wikipedia.org/wiki/Fabrikmethode#Beispiele

Abstract factory pattern



Application examples: Pluggable look and feel (Motif, Mac, Windows), Test- and productive version factories.

Example: Concrete Factories: Windows and Mac. AbstractProductA: Scrollbar, AbstractProductB: Button

Where is the concrete factory set?

- Factory may be created by client (AbstractFactory a = new ConcreteFactory())
- In the abstract Factory class itself
- In a special class

```
Class CurrentFactory{
      Private CurrentFactory(){}
      Private static Factory current = null;
      Public static Factory getFactory(){return current;}
      Public static void setFactory(Factory f){
            If(f==null) throw new IllegalArgumentException();
            current = f;
      }
}
```

How is a concrete Factory registered?

- Externally: `CurrentFactory.setFactory(new Factory1());`
- Automatically by factory itself

```
Class Factory1 implements Factory{
Public A createA(){…}
Public B create(){…}
Static{ CurrentFactory.setFactory(new Factory1());}
Private Factory1(){};
}
```

Advantages

- Isolates concrete classes: Clients work with abstract classes, not with actual implementation classes
- Exchanging product families is very easy: Needs a change of one line, namely where the abstract factory is initialized.
- Inherently guarantees consistency among different factories

Disadvantage

- It's difficult to extend the abstract factory with more products

Dependency Injection

- GUI class has references to multiple concrete factory instances => All possible factories are loaded, even though only one is needed at runtime.
- Dependency injection in Java done with Java Beans

Singleton pattern

- Ensure that a class has only a single instance
- Problems with the static approach
    - We may require runtime information to prepare the static class
    - Order in which static initializers are called is not statically defined (24 May 13, Slide 36)
    - Static methods cannot implement an interface
    - Singleton idea

```
Public final class Registry{
    Private Registry(){
    private static Registry instance = new Registry();
    public static Registry getInstance(){return instance;}
}
```

- Problems:
    - Hidden coupling: Whether a Singleton is used or not is not clear from the interface, but only from the implementation. Why: the Singleton cannot be constructed from outside and therefore can't be a class member.
    - Violation of Single Responsibility principle. Creation of an object AND its use
    - Difficulties in testing: Polymorphic substitution by a mock object is difficult.
    - Concurrency problems: Need to make sure that Singleton class is thread-safe.
- Alternative implementation with enum

```
public enum SingletonDriver implements Driver {
    INSTANCE;
    public String toString(){return "Singleton";}
    public void playSong(File file){ ... }
}
```

- Enums are inherently thread-safe
- Enums guarantees unique instance. Access with SingletonDriver.INSTANCE
- Interfaces may be implemented, but no classes may be extended, since enums implicitly extend Java.lang.enum
- The constructor of an enum must be private or internal.

```
public enum Planet {
    MERCURY (3.303e+23, 2.4397e6),

    private final double mass;   // in kilograms
    private final double radius; // in meters
    Planet(double mass, double radius) {
        this.mass = mass;
        this.radius = radius;
    }
```