Selected problems of OOP

Ing. Lukáš Hruška

1

Hidden pointers

2

Exceptions

3

Loops, the object oriented way

4

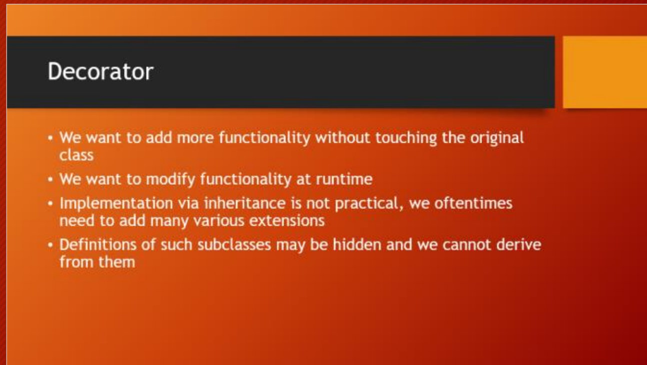
Design patterns

(part 1)

Overview

Overview

1. Design patterns (part 2)
2. Streams
3. Working with files
4. Reflections
5. Q&A



Decorator

- We want to add more functionality without touching the original class
- We want to modify functionality at runtime
- Implementation via inheritance is not practical, we oftentimes need to add many various extensions
- Definitions of such subclasses may be hidden and we cannot derive from them

Overview – next week

- Design patterns
- Streams
- Working with files
- Q&A
- More stuff

Pointers are awesome

No, really, they are

OOP is completely based on pointers

Object -> fields - structure in memory; methods - functions to manipulate the data

When a function is called, parameters are copied

If we post an object as a parameter, the reference is copied, not the object itself = we are accessing the same memory

When something doesn’t work as intended

- C/C++ - checking is just a waste of execution time, we should trust the programmer not to make mistake
  - SIGSEGV :)
  - Buffer overflow attacks – trick to push virus code onto the execution stack
- Java / C# … Exceptions – it is better if the program crashes than to let something malicious to happen
- Java
  - Checked exceptions – subclasses of Exception, must be caught or declared as thrown (even by main method), usually recoverable
  - Unchecked exceptions – subclasses of Error / RuntimeException, does not have to be caught or declared, usually irrecoverable

When something doesn’t work as intended 2

- Exception is an object that signifies that normal execution of the program has been interrupted in some way
  - Something forbidden happened in the system
  - Some programmer-specified conditions were violated
- try-catch-finally
  - try – identify the error-prone section of the code (i.e. reading from disk)
  - catch – executed <u>only</u> if the error has occurred; recovery from exception
  - finally – guaranteed to be executed regardless of the exception (release shared memory, close files, et c.)

When something doesn’t work as intended 3

- throw new Exception();
- If the method does not want to handle the exception (or can’t), it may leave it for someone else to handle down the call stack
  - In this case, the execution of method is interrupted
  - Methods that throw checked (user defined) exceptions (but do not handle them) have to be marked with throws keyword
- The exception may be caught in catch block and thrown anew

for (;;)

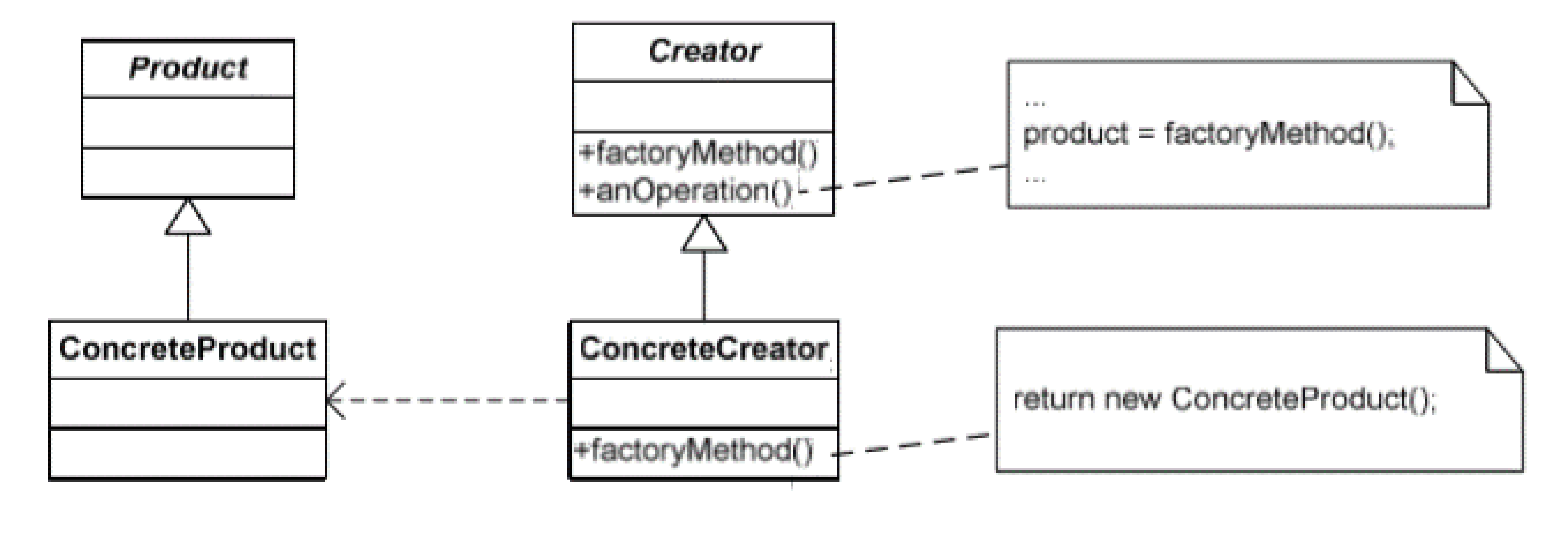
- `for(int i = 0; I < list.length(); i++)`
  - Prehistoric
  - Does not work on every type of collection, such as linked list
- Using iterators – the class has to implement Iterable interface
  - (C# IEnumerable)
- The consumer of iterator does not have to be aware of the actual collection’s type
- forEach, while
- Lazy vs eager loading
- (C# -> yield return – allows streaming collections one entry at a time)

Design patterns

- Avoid trial and error by using already established solutions
- Framework vs design pattern
- Creational
  - Factory (method), Abstract Factory, Singleton, Builder, Prototype
- Structural
  - Adapter, Composite, Decorator, Façade, Proxy
- Behavioural
  - Chain of responsibility, Iterator, Strategy, Visitor

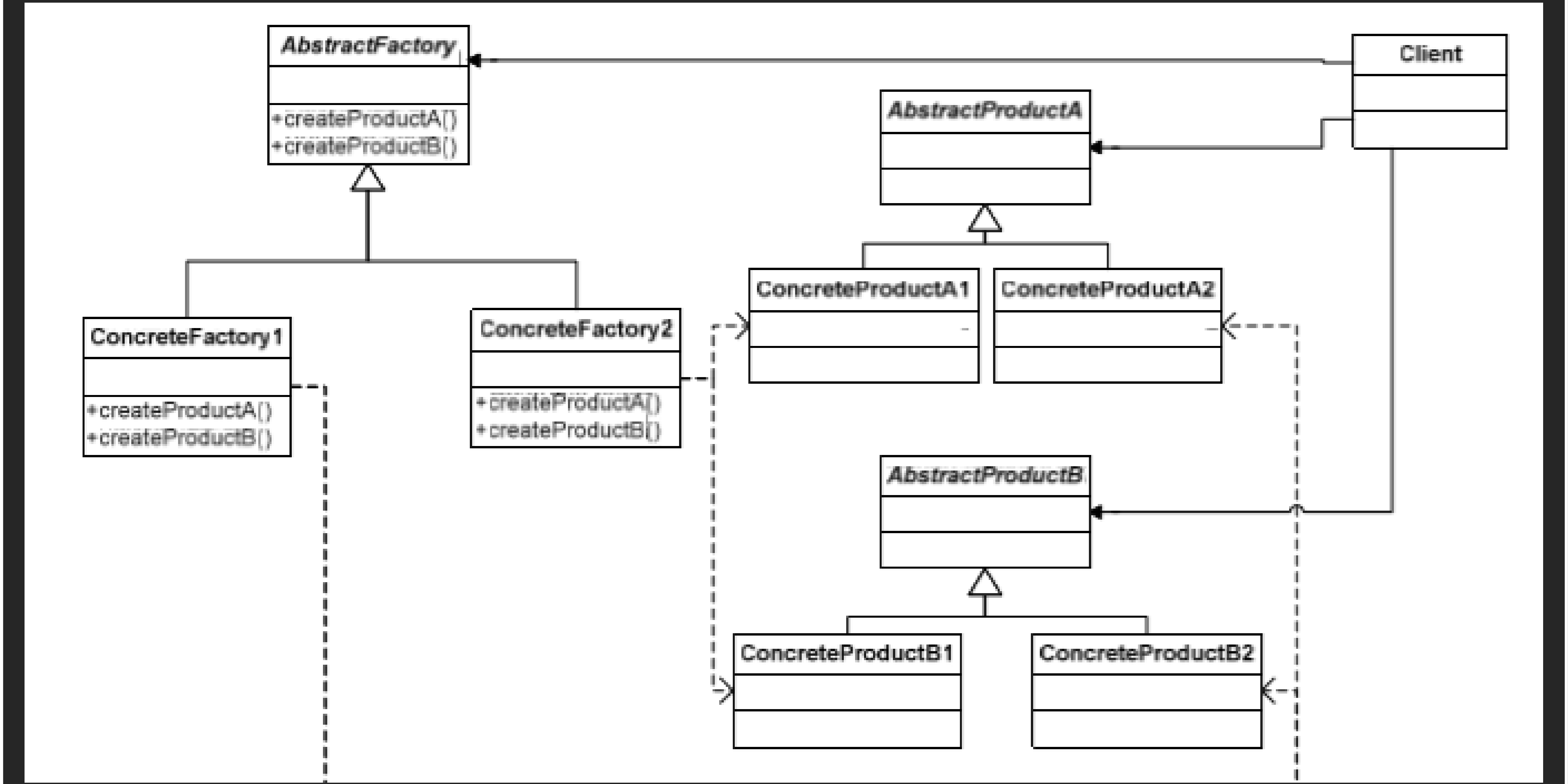
Factory (method)

- When you don’t know at design time what class object you need
- When all potential classes are in the same subclass hierarchy
- To centralize class selection code
- To encapsulate object creation


Product
Creator
+factoryMethod()
+anOperation()
...
product = factoryMethod();
...
ConcreteProduct
ConcreteCreator
+factoryMethod()
return new ConcreteProduct();

Abstract Factory

- Like factory, but everything is encapsulated
  - The method that orders the object
  - Factories, that build the object
  - Final objects
  - Final objects contain objects that use the strategy pattern
- Allows to create families of related objects without specifying a concrete class
- Use when there are many objects, that can be added dynamically during runtime
- (We will return to it later)


AbstractFactory
+createProductA()
+createProductB()
Client
AbstractProductA
ConcreteProductA1
ConcreteProductA2
ConcreteFactory1
+createProductA()
+createProductB()
ConcreteFactory2
+createProductA()
+createProductB()
AbstractProductB
ConcreteProductB1
ConcreteProductB2

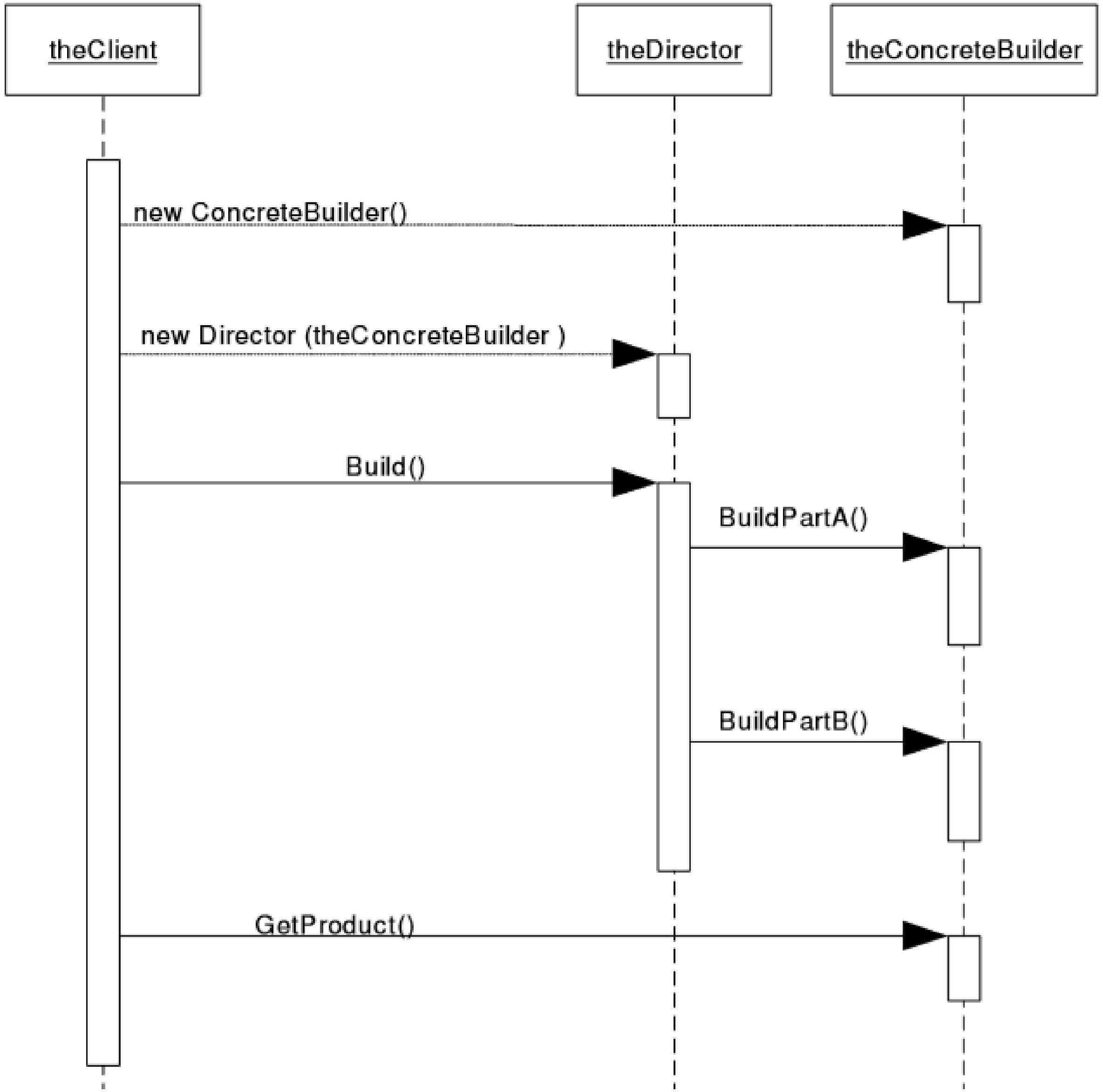
Singleton

- At most one instance may exist at any given time
- Sometimes we want only one instance of a class
- We want to provide easy access to this object
- We want to have guaranteed there won’t be any more instances

- Unlike static class, singleton may be passed as a parameter

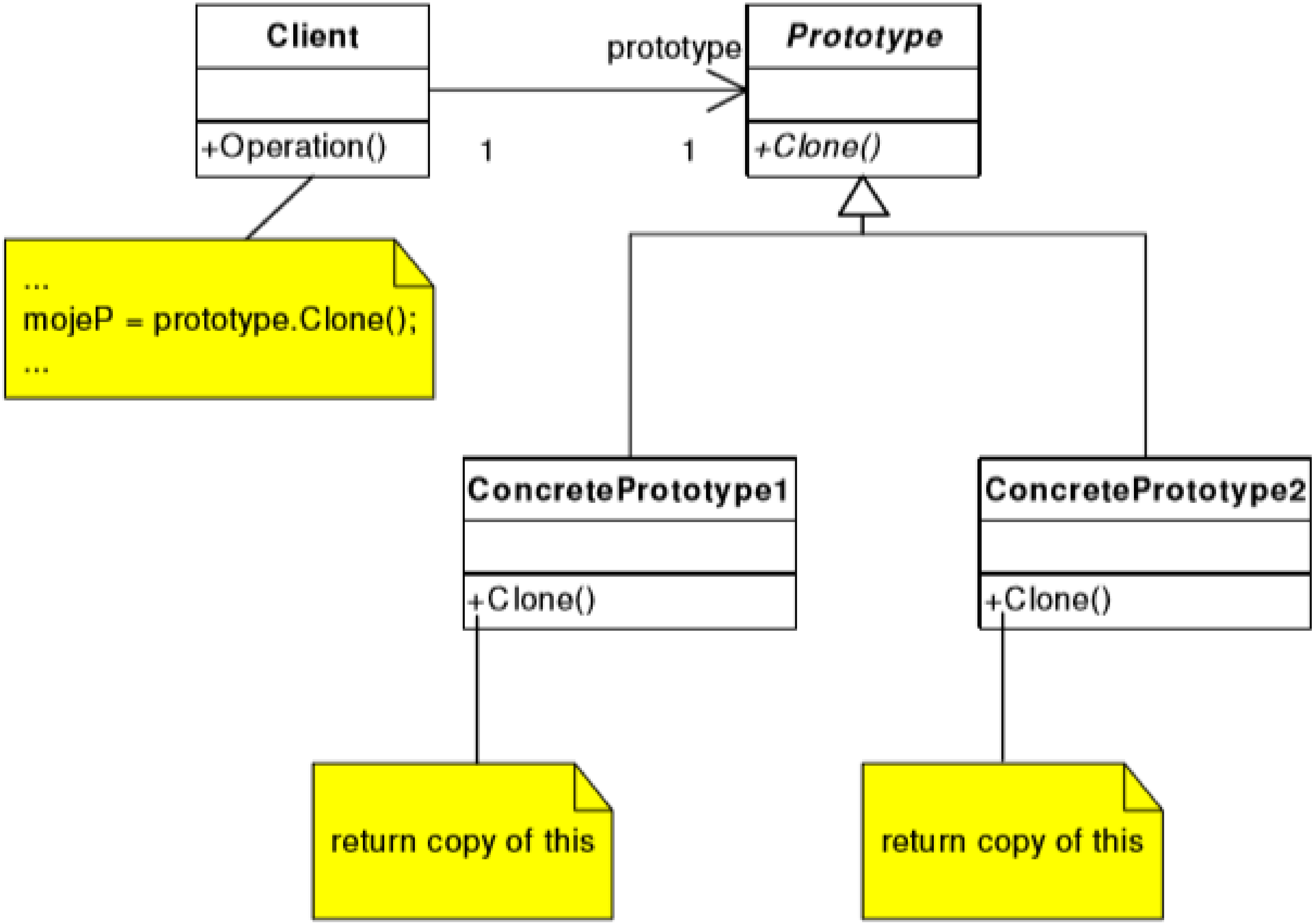
Builder

- Used to create objects made of a buch of other objects
  - Build an object made up from other objects
  - Creation of parts independent of the main object
  - Hide the creation of parts
  - The builder knows the specifics and nobody else does


theClient
theDirector
theConcreteBuilder
new ConcreteBuilder()
new Director (theConcreteBuilder)
Build()
BuildPartA()
BuildPartB()
GetProduct()

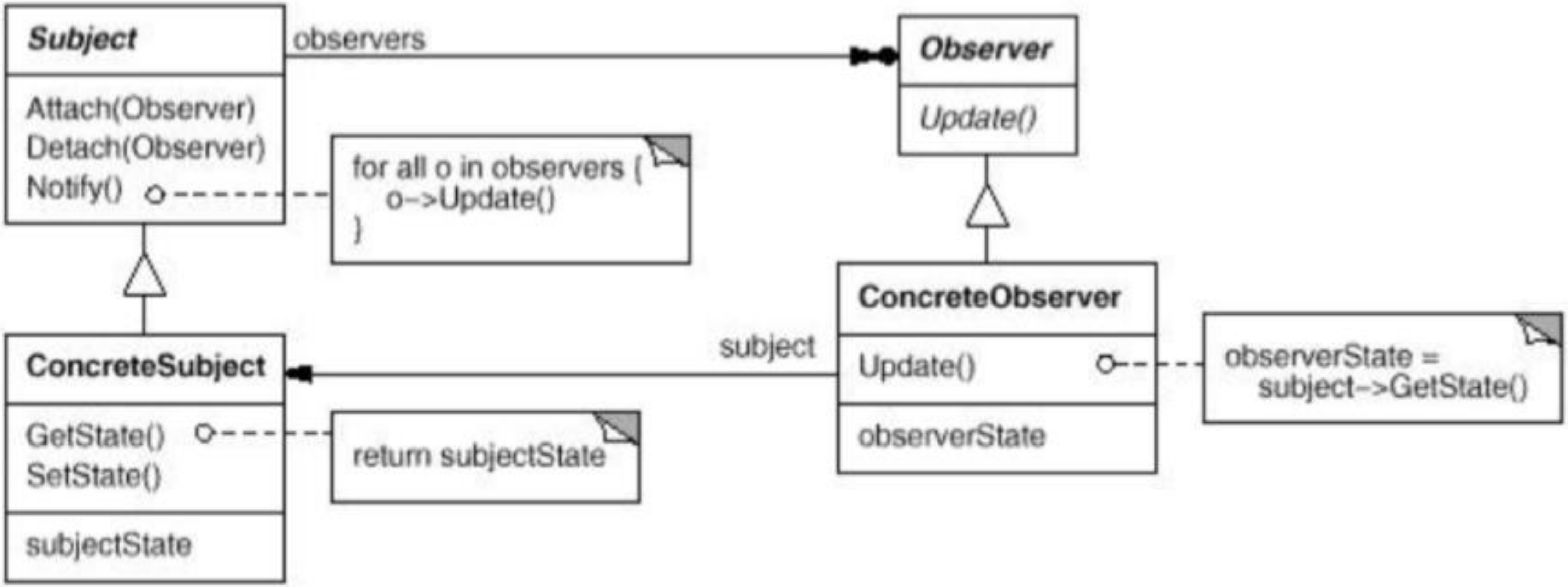
Prototype

- Structure of products has shared interface, which supports cloning of objects
- Client contains a list of prototype objects, from which are new objects being cloned
- Client uses this shared interface regardless of specific object type
- Watch out for shallow / deep cloning


Client
+Operation()
prototype
1
1
Prototype
+Clone()
...
mojeP = prototype.Clone();
...
ConcretePrototype1
+Clone()
ConcretePrototype2
+Clone()
return copy of this
return copy of this

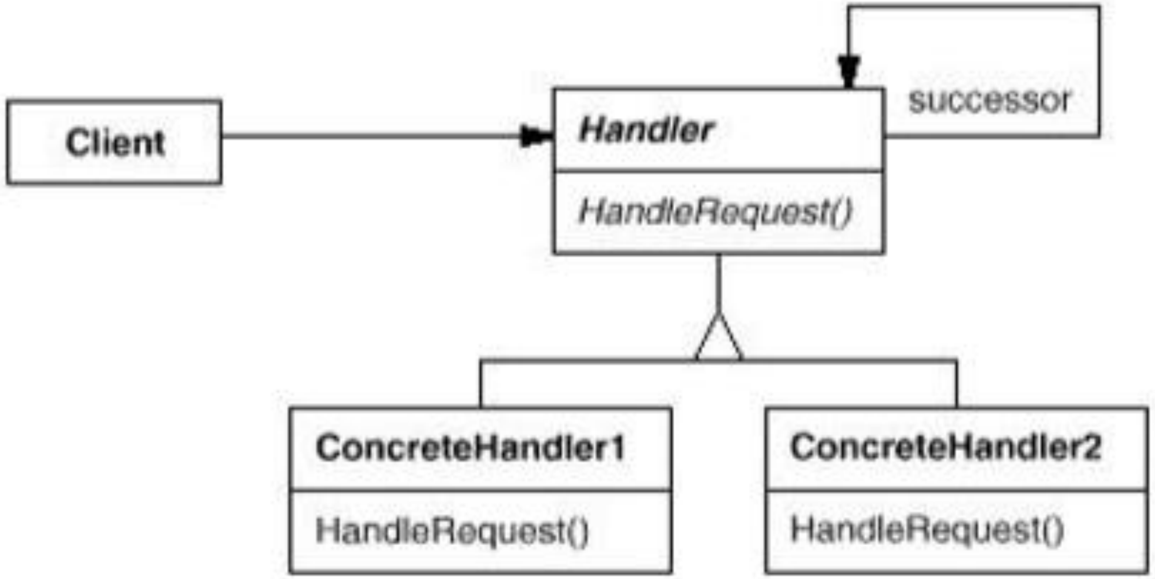
Observer

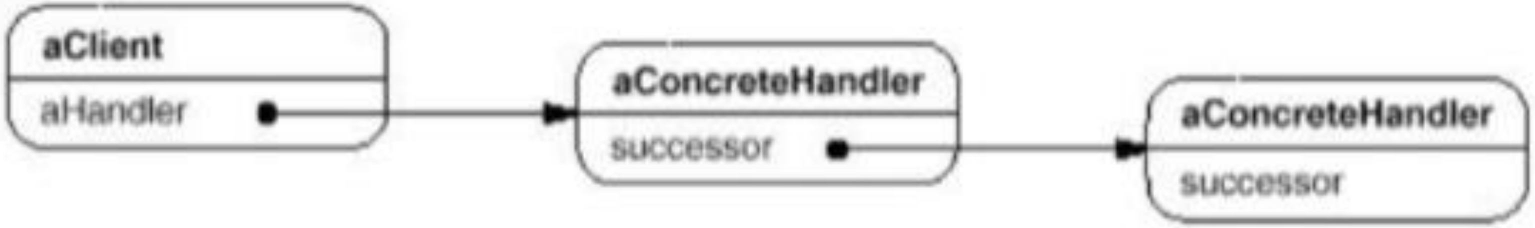
- Publisher-subscriber relation
- When change in one object should cause a change in another one
- When an object should inform other objects without knowing them


Subject
observers
Observer
Attach(Observer)
Detach(Observer)
Notify()
Update()
for all o in observers {
o->Update()
}
ConcreteObserver
subject
ConcreteSubject
Update()
observerState =
subject->GetState()
GetState()
SetState()
return subjectState
observerState
subjectState

Chain of responsibility

- Create a chain of objects, each able to perform different task
- They have to share a common interface
- If an object is able to handle the command, it does it and returns a value
- Otherwise it calls it successor for help

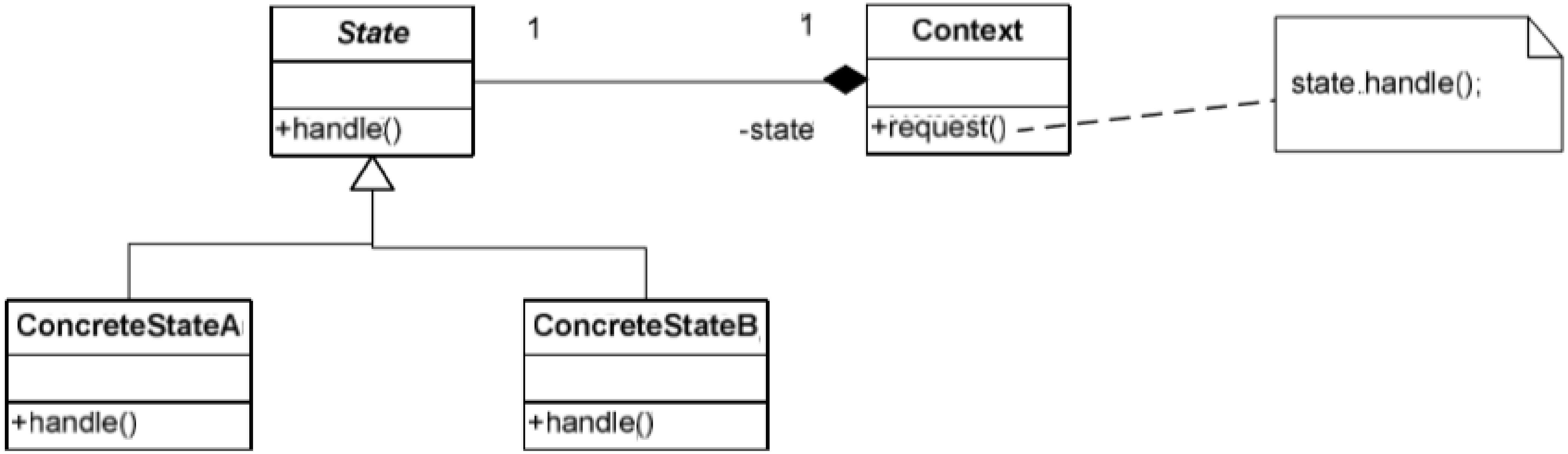

Client
Handler
HandleRequest()
successor
ConcreteHandler1
HandleRequest()
ConcreteHandler2
HandleRequest()


aClient
aHandler
aConcreteHandler
successor
aConcreteHandler
successor

Strategy

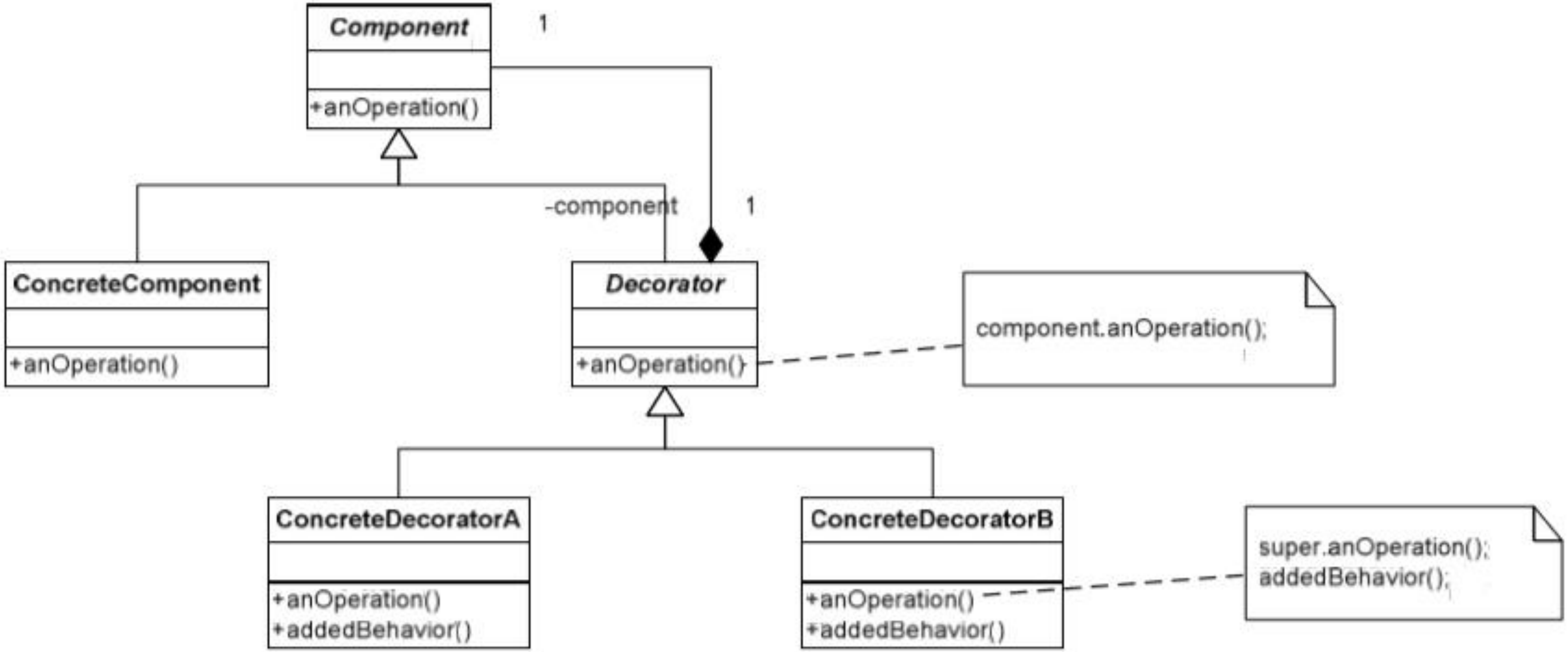
- Defines set of similar algorithms (functions) that are encapsulated and interchangeable

- Similar classes differ only in their behaviour
- We need different variations of the same algorithm
- Class defines many functions with conditional execution
- It is better to define such functionality in separate classes


State
+handle()
1
1
-state
Context
+request()
state.handle();
ConcreteStateA
+handle()
ConcreteStateB
+handle()

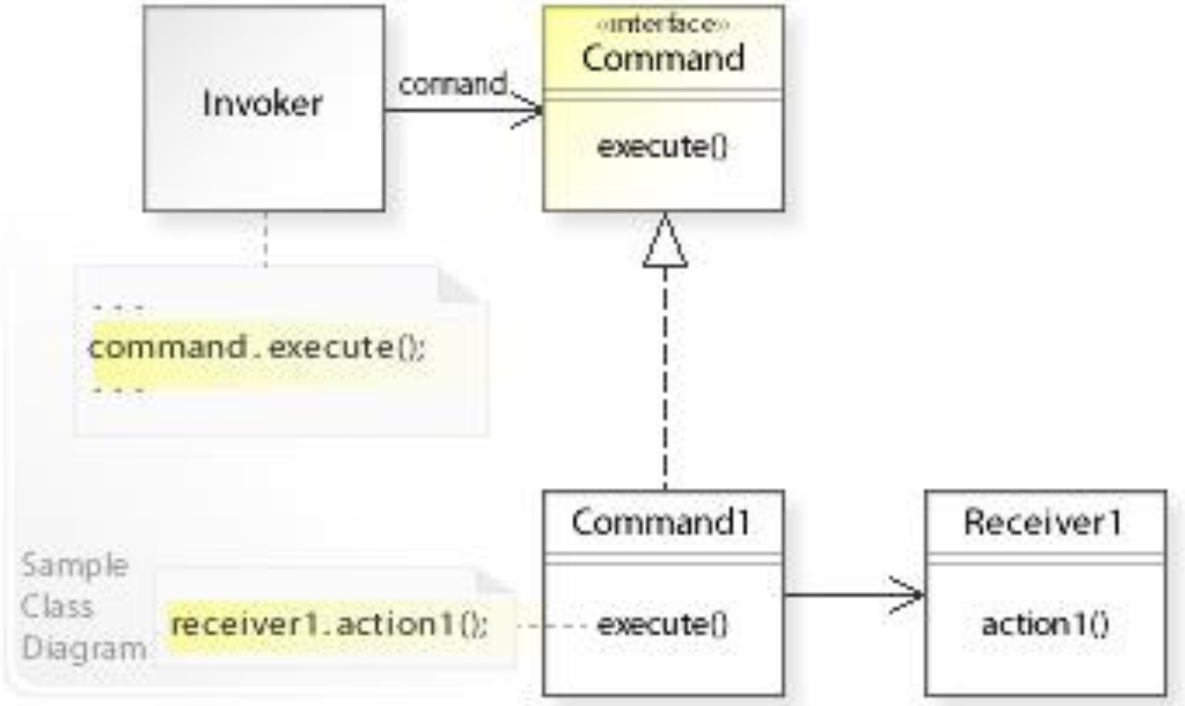
Decorator

- We want to add more functionality without touching the original class
- We want to modify functionality at runtime
- Implementation via inheritance is not practical, we oftentimes need to add many various extensions
- Definitions of such subclasses may be hidden and we cannot derive from them


Component
+anOperation()
1
-component
1
ConcreteComponent
+anOperation()
Decorator
+anOperation()
component.anOperation();
ConcreteDecoratorA
+anOperation()
+addedBehavior()
ConcreteDecoratorB
+anOperation()
+addedBehavior()
super.anOperation();
addedBehavior();

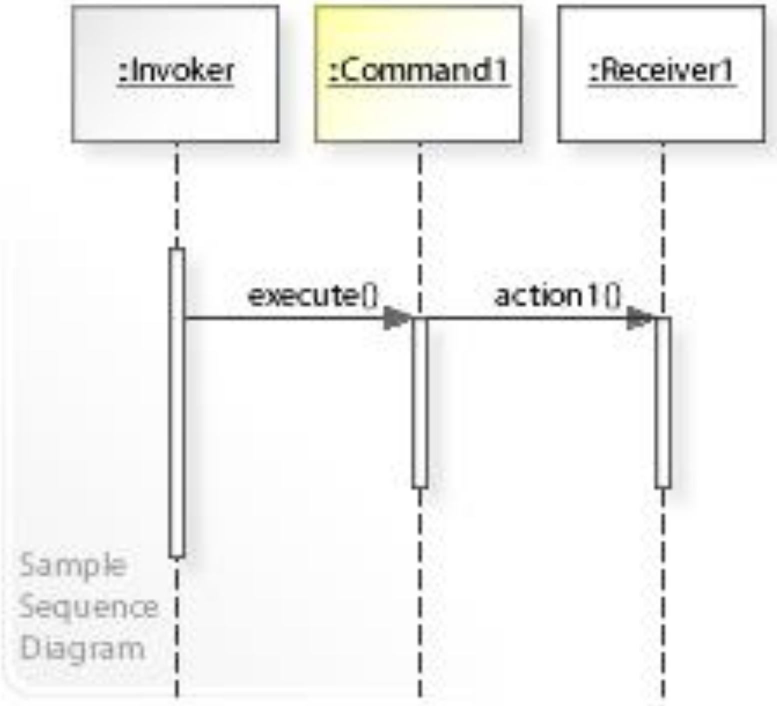
Command / Action<T>

- Allows the code to be prepared for execution later
- Allows to store a list of actions to be executed if a certain condition is met
- The invoker does not know anything about the command itself, only the interface
- Command stores the receiver when created
- Action<T> does not store the receiver, it gets passed as a execute method parameter



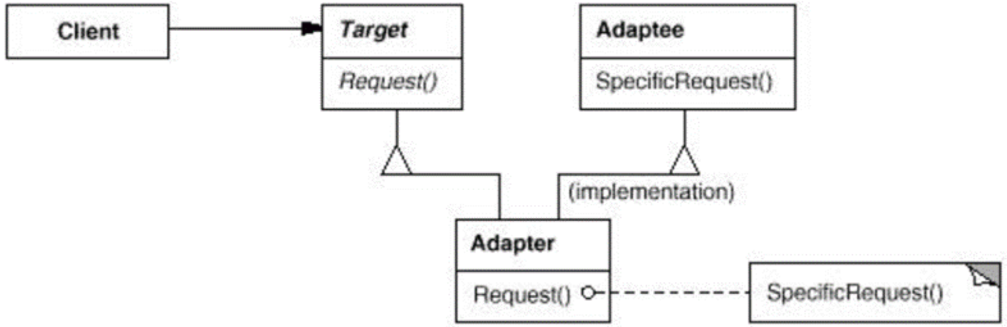
```
- - -
command.execute();
- - -
```

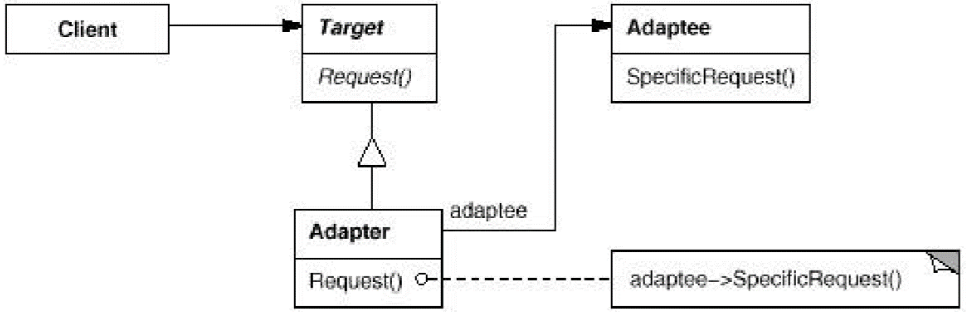
```
receiver1.action1();
```



Adapter

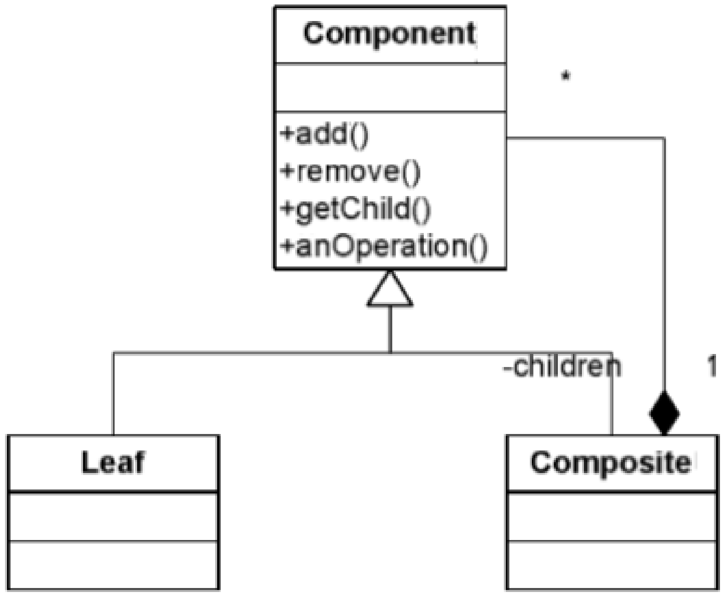
- Conversion of incompatible interfaces
- Cannot change the interface of the source code for some reason (compatibility, no source code available…)
- Two-way transparency – possible, but usually messy
- How much adaptation is needed?
  - Conversion of simple interfaces where we only need to rename operations
  - Implementation of completely different set of operation

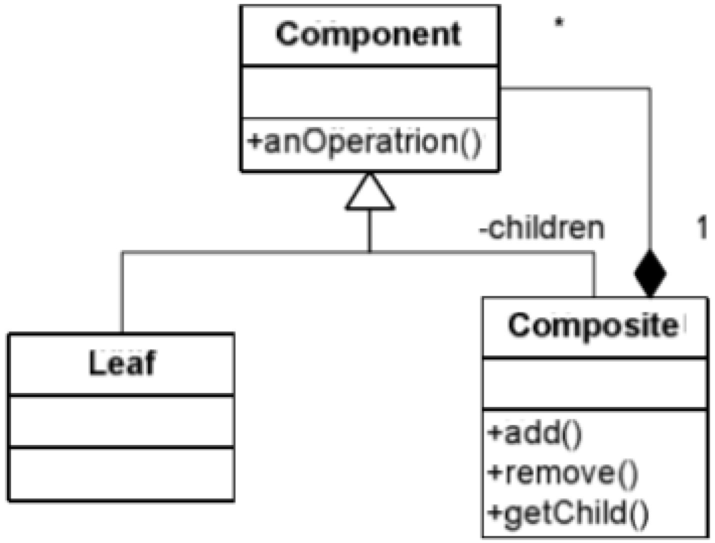

Client
Target
Request()
Adaptee
SpecificRequest()
(implementation)
Adapter
Request()
SpecificRequest()


Client
Target
Request()
Adaptee
SpecificRequest()
adaptee
Adapter
Request()
adaptee->SpecificRequest()

Composite

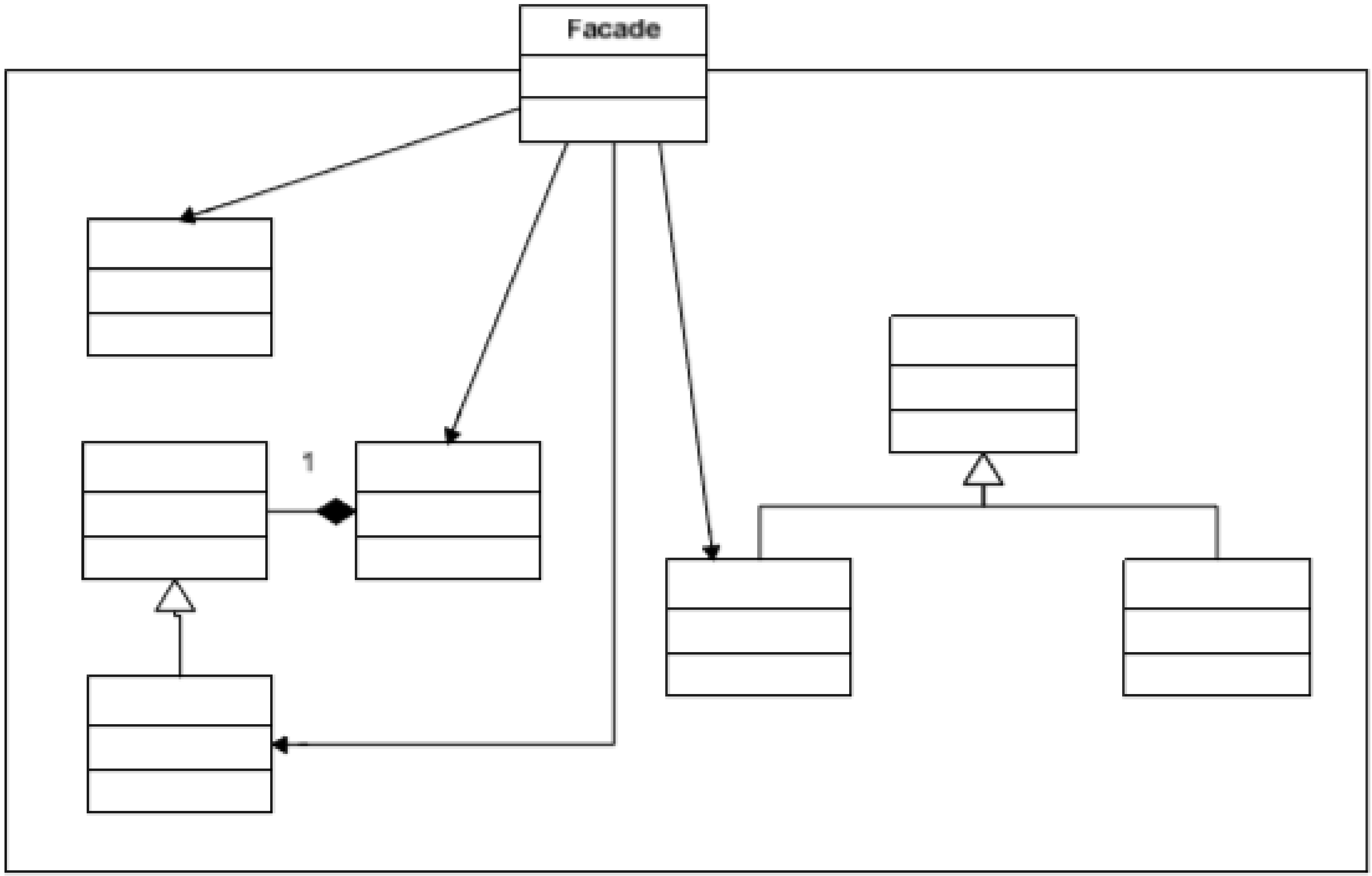
- Allows to treat separate objects and compositions of objects uniformly
- It is easy to add new types of components
- We can implement simple clients that do not need to distinct between composed objects and components
- Issues
  - Sometimes it is good to implement a reference to the parent object – it allows to apply Chain of Responsibility design pattern
  - Either transparent (inside component, which allows component and composed objects to use the same interface) or safe, which does not allow this


Component
+add()
+remove()
+getChild()
+anOperation()
*
-children
1
Leaf
Composite


Component
+anOperatrion()
*
-children
1
Leaf
Composite
+add()
+remove()
+getChild()

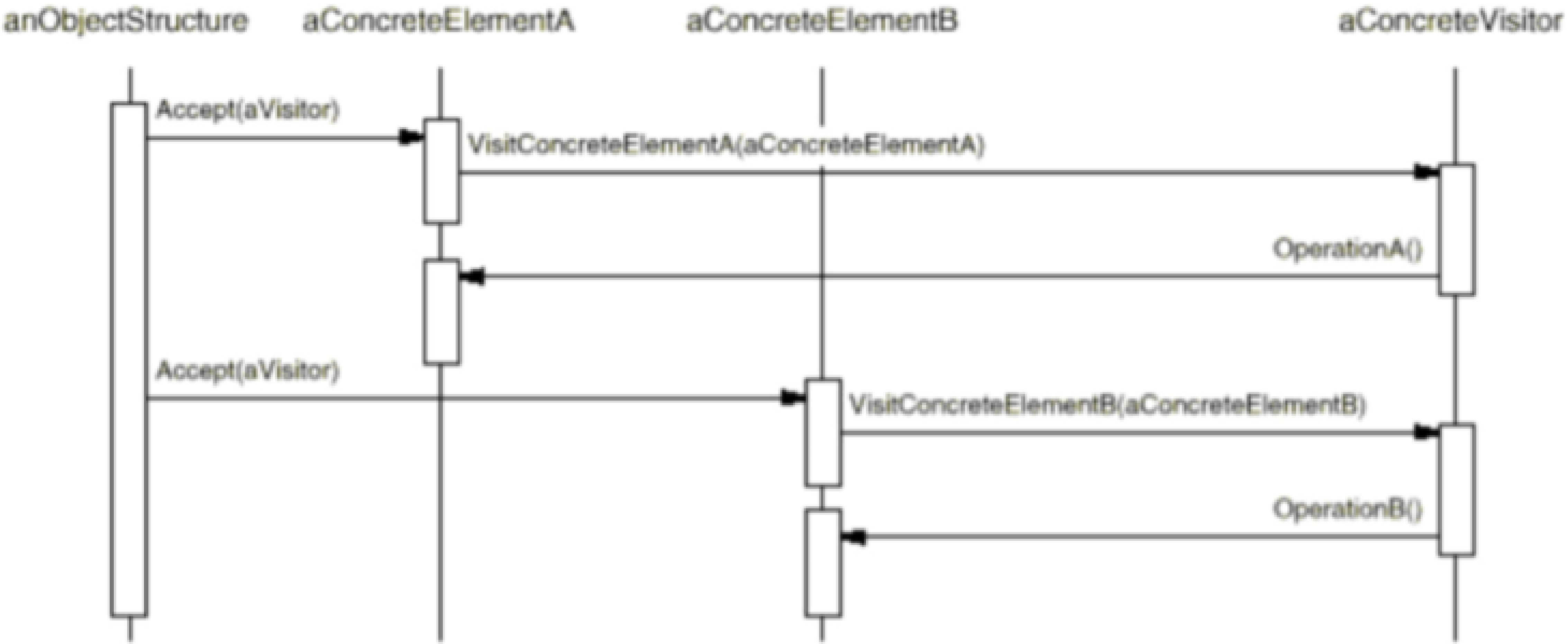
Facade

- Used to simplify complex interfaces (or hide them)
- Weakens binding among subsystems – more flexibility in modification
- Does not add functionality, only reduces existing one


Facade
1

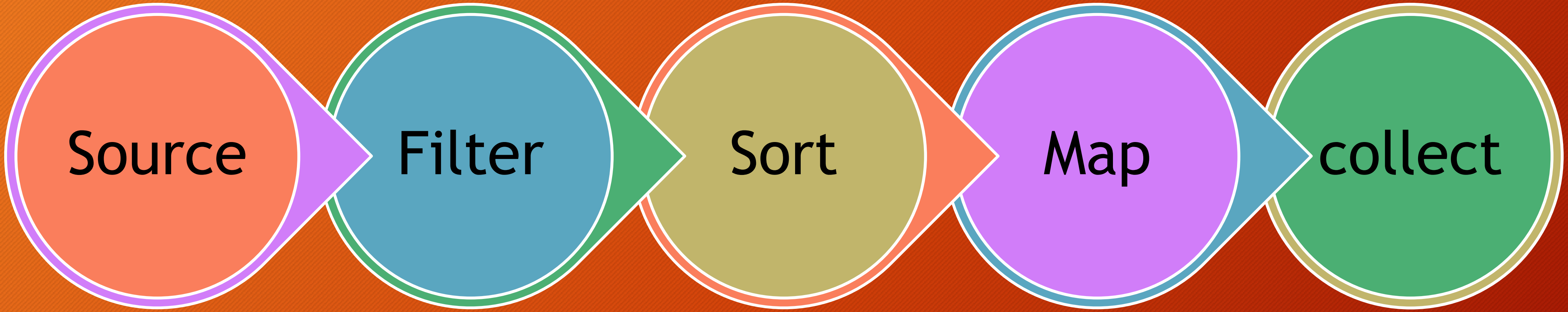
Visitor

- Allows to add methods to classes of different types without much altering to those classes
- It is possible to make completely new methods based on the class used
- Adding new concreteElement requires a lot of code changes
- ! It is necessary sometimes to violate object encapsulation of concreteElement


anObjectStructure
aConcreteElementA
aConcreteElementB
aConcreteVisitor
Accept(aVisitor)
VisitConcreteElementA(aConcreteElementA)
OperationA()
Accept(aVisitor)
VisitConcreteElementB(aConcreteElementB)
OperationB()

Method chain (Fluent interface)

- Allows invoking multiple method calls
- Each method returns the object
- No need to store intermediate results in local variables



Stream API

Stream API

- Starts with collection (List, set, array…)
- Filter – reduce the collection based on a predicate (lambda)
- It is possible to chain multiple filters
- Map – transforms the data (x -> x * x)
- Collect/Reduce – final operation which returns non-stream data (either a collection, object or a value, e.g. count)
- ParallelStream – multi-threading; good for large collections

Autocloseable (try with resources)

- Inspired by C# Disposable pattern
- Implements Autocloseable interface
- Automatically releases resources once finished / crashed
- May have catch and finally blocks, but doesn’t have to

```
try(x = new X()) {
   //do something with x
}
```

Data streams

- Sequence of data elements
- Possibly infinite length
- Processing stream
  - One item at a time
  - Multiple items (e.g. moving average)
- Can represent I/O
- stdin, stdout, stderr
- If stored, navigation functions, such as go to start / end can be used

Files

- The program needs to request a file handle from the operating system – Read / Write / Read & Write…
- Multiple handles may access the same file in read-only state
- Write is exclusive
- Utilize (file)streams
- Usually wrapped in more complex classes

Back to streams – memory stream

- ByteArrayInputStream / ByteArrayInputStream in Java (C# MemoryStream)
- Stream stored in memory
- Logic that uses file streams can be applied here (read, write, save to…)
- Can be later saved to the disk

Serialization

- Unlike structures in C, objects cannot be simply saved
  - We need to preserve references somehow
- Object serialization is just the tool for this
  - transient keyword – prevents serialization of the field
  - Has to “implement” Serializable interface
- Object is serialized and written into a stream, such as ObjectOutputStream, which is then stored or transferred
  - Type information is stored as well
- Deserialization – reverse process, we may override the process to inject initialization code

Reflections

- Reflection is an API which is used to examine or modify the behaviour of methods, classes, interfaces at runtime.
- Reflection gives us information about the class to which an object belongs and also the methods of that class which can be executed by using the object.
- `invoke(object, args)`
- Args – parameters (null, if no parameters are expected)

Reflections

- Advantages of Using Reflection:
  - Extensibility Features - an application may make use of external, user-defined classes by creating instances of extensibility objects using their fully-qualified names.
  - Debugging and testing tools: Debuggers use the property of reflection to examine private members on classes.
- Drawbacks:
  - Performance Overhead: Reflective operations have slower performance than their non-reflective counterparts, and should be avoided in sections of code which are called frequently in performance-sensitive applications.
  - Exposure of Internals: Reflective code breaks abstractions and therefore may change behavior with upgrades of the platform.
- Challenge: try to create Actor factory using reflections instead of if chain

Q & (maybe)A