Software Architectures

From Design Patterns to Enterprise Architecture

Design Patterns III

Last week on “Software Architectures”

- Decorator Pattern
- Command Pattern
- Iterator Pattern
- Object Identity vs. Object Equality

Patterns

Visitor (331)

Visitor
Motivation

- Iterating over tree-like structures can be hard to implement.
- Sometimes the traversal strategy depends on the processing of the elements, which rules out Iterator
- You could implement the operations in each class, but this would lead to code duplication and scattering. Cohesion would suffer.

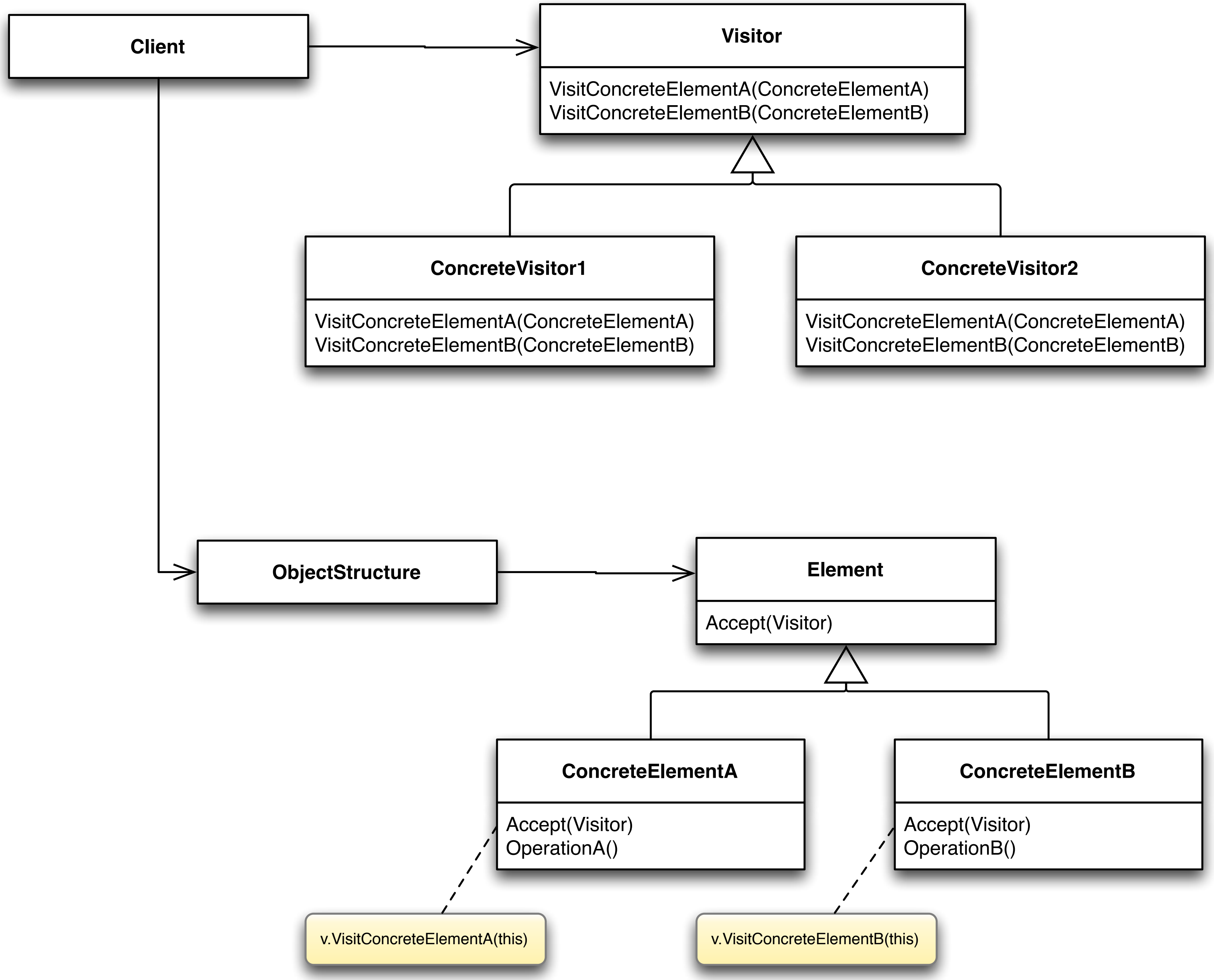
Visitor

Intent

- Represent an operation to be performed on the elements of an object structure.
- Visitor lets you define a new operation without extending the classes of the elements which it operates on.

Visitor

Structure


Client
Visitor
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor1
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor2
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ObjectStructure
Element
Accept(Visitor)
ConcreteElementA
Accept(Visitor)
OperationA()
ConcreteElementB
Accept(Visitor)
OperationB()
v.VisitConcreteElementA(this)
v.VisitConcreteElementB(this)

Visitor
Consequences

- It makes adding new operations easy
- It helps to structure operations better and help cohesion
- You may also accumulate state
- But, it is hard to add new ConcreteElement classes
- It also cannot work across different hierarchies
- It may lead you to break encapsulation on ConcreteElement classes

Patterns

Strategy (315)

Strategy

Motivation

- …many related classes differ only in their behavior rather than implementing different related abstractions
  Strategies allow to configure a class with one of many behaviors.
- …you need different variants of an algorithm
  Strategies can be used when variants of algorithms are implemented as a class hierarchy.
- …a class defines many behaviors that appear as multiple conditional statements in its operations
  Move related conditional branches into a strategy.

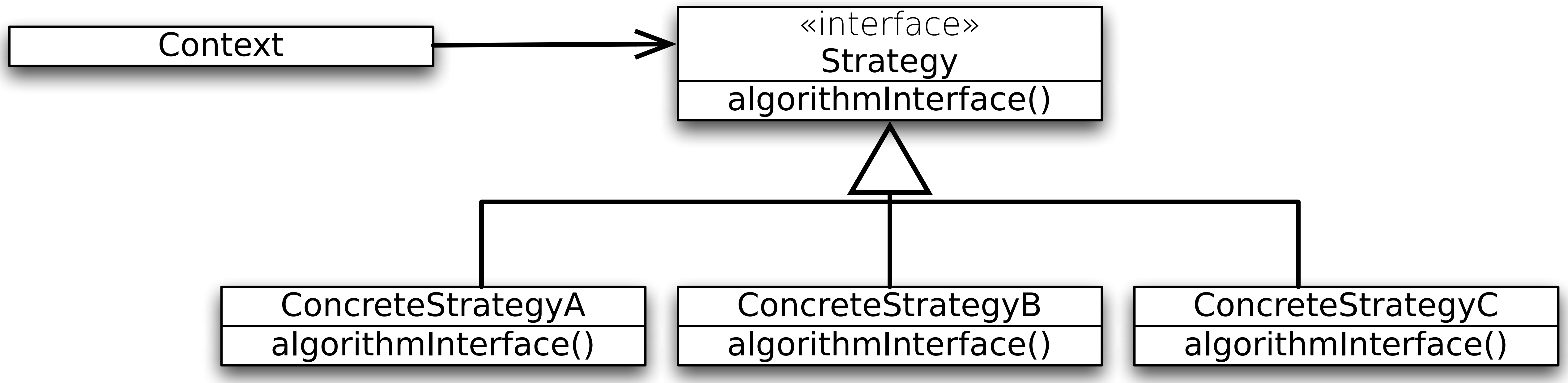
Strategy

Intent

- Define a family of algorithms, encapsulate each one, and make them interchangeable.
- Strategy lets the algorithm vary independently from clients that use it.

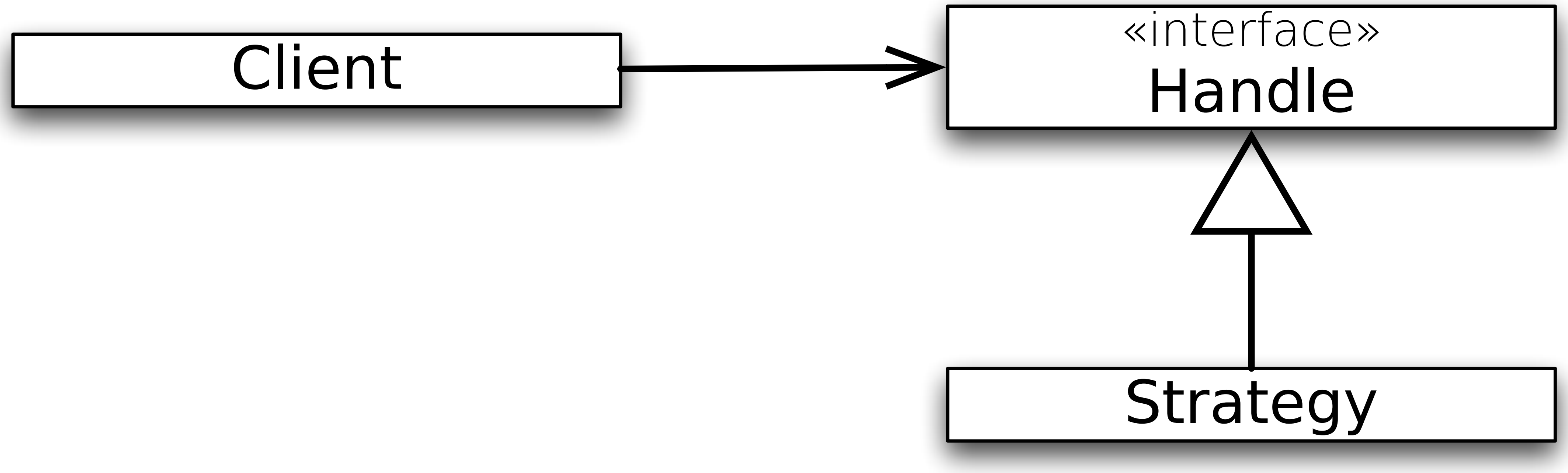
Strategy

Structure



Strategy

Structure



Strategy
Effects

- Sub-classing Context mixes algorithm‘s implementation with that of Context
  Context harder to understand, maintain, extend.
- When using sub-classing we can't vary the algorithm dynamically
- Sub-classing results in many related classes
  Only differ in the algorithm or behavior they employ.
- Encapsulating the algorithm in Strategy…
  - lets you vary the algorithm independently of its context
  - makes it easier to switch, understand, and extend the algorithm

Strategy

Downsides

- Clients must be aware of different strategies and how they differ, in order to select the appropriate one
- Clients might be exposed to implementation issues
- Use Strategy only when the behavior variation is relevant to clients

Group Exercise

Recognizing Patterns in Architecture

Patterns

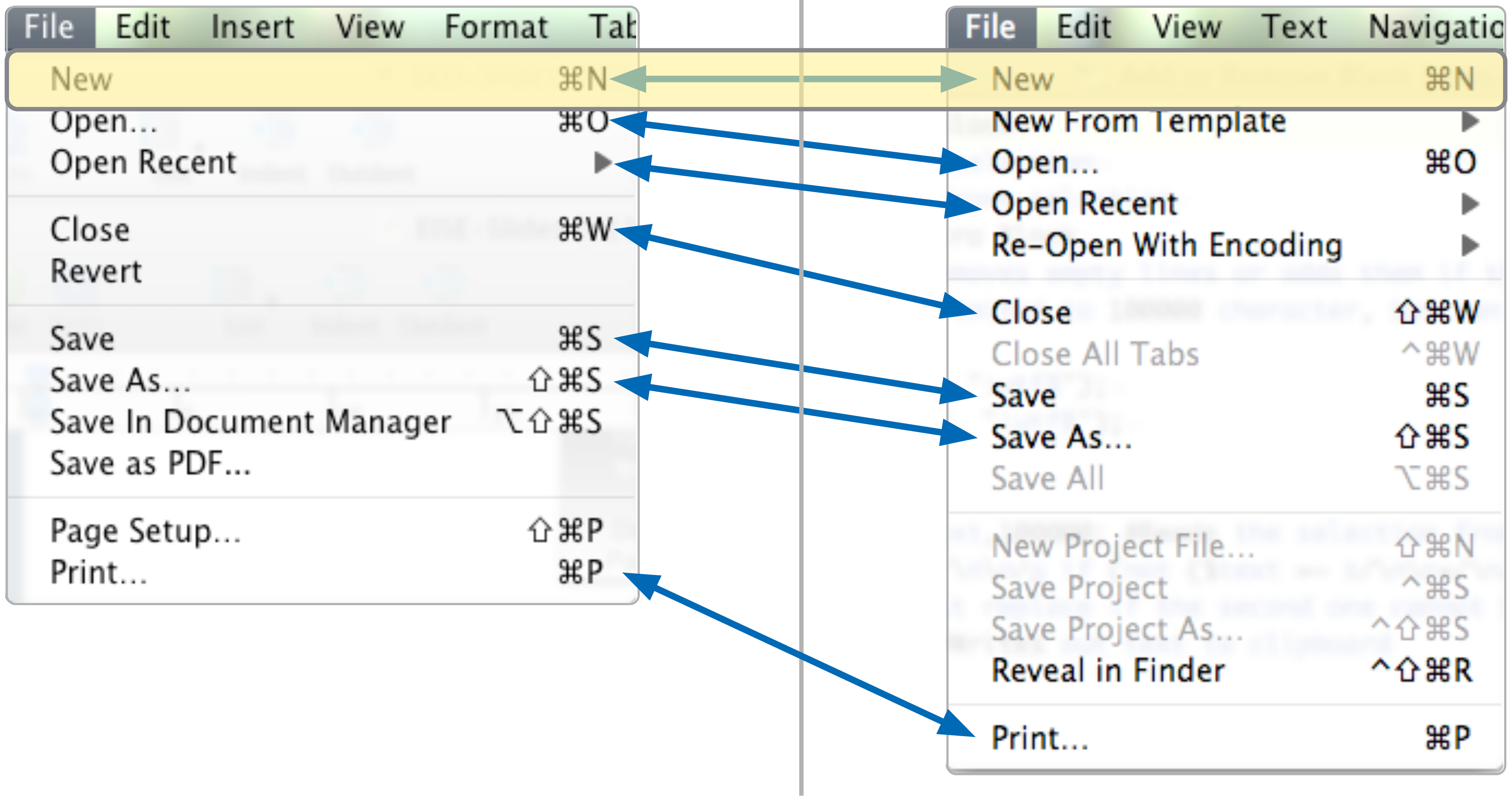
Factory Method (107)

Factory Method

Motivation

Nisus Writer Pro

TextMate

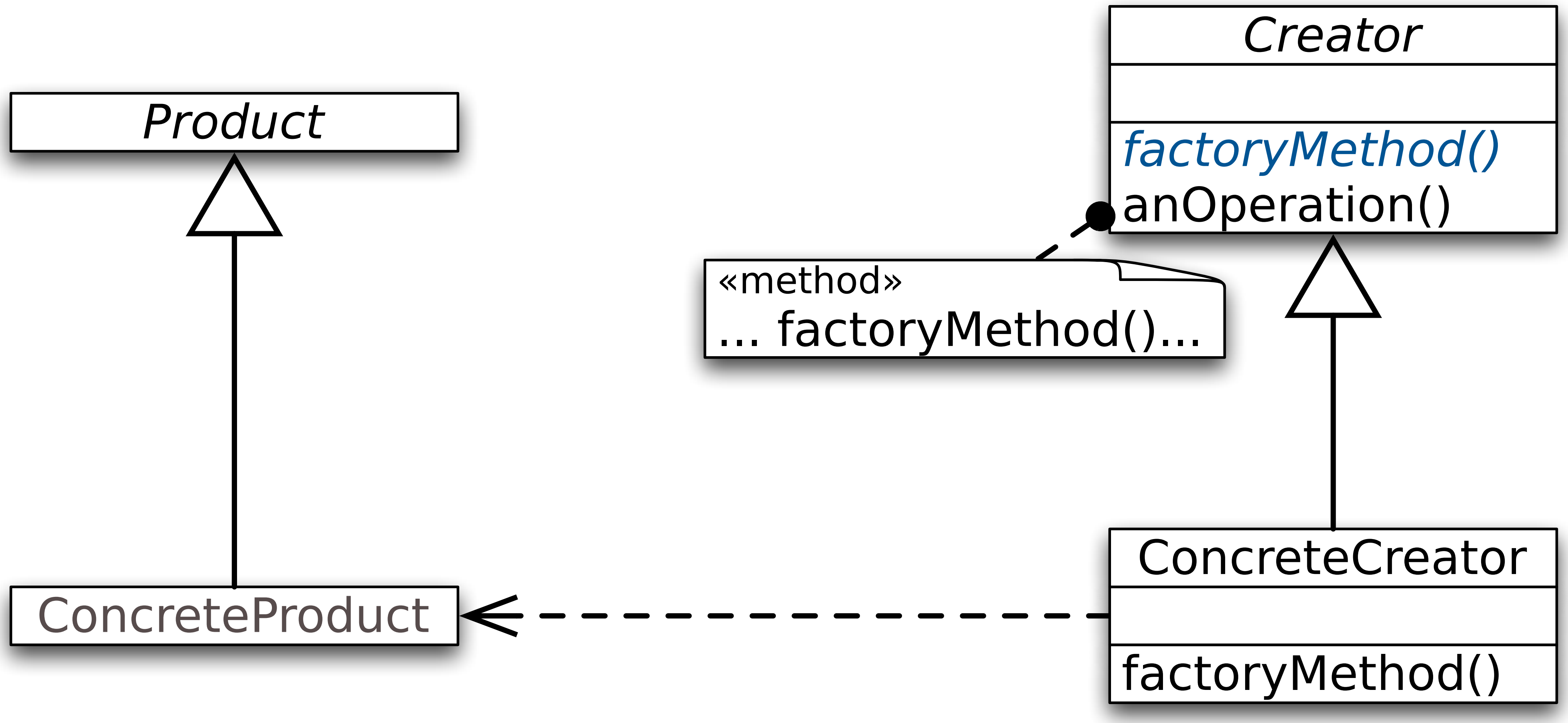


Factory Method

Intent

- Define an interface for creating an object, but let subclasses decide which class to instantiate.
- Factory Method lets a class defer instantiation to subclasses.

Factory Method
Structure



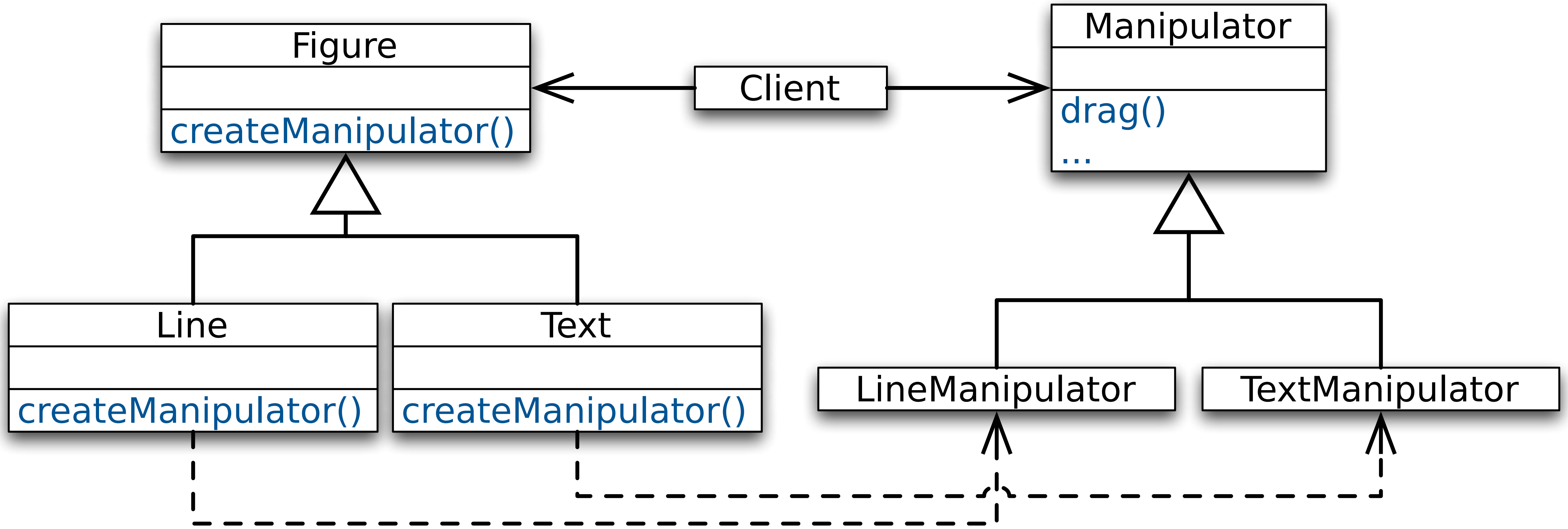
Factory Method

Consequences

- The framework’s code only deals with the Product interface; therefore it can work with any user-defined ConcreteProduct class
- Provides a hook for subclasses
  The hook can be used for providing an extended version of an object

Factory Method
Example

- May be used to connect different class hierarchies



Patterns

Singleton (127)

Singleton

Motivation

- In some cases a mechanism is required to enforce singularity of objects; i.e. it is necessary to enforce that there exists at most one instance of a class at runtime. For example, ...
  - in a system there should be only one printer spooler,
  - there should be only one class to handle interactions with the database.
- Two patterns for enforcing Singularity:
  - Singleton
  - Monostate, which we will not cover here...

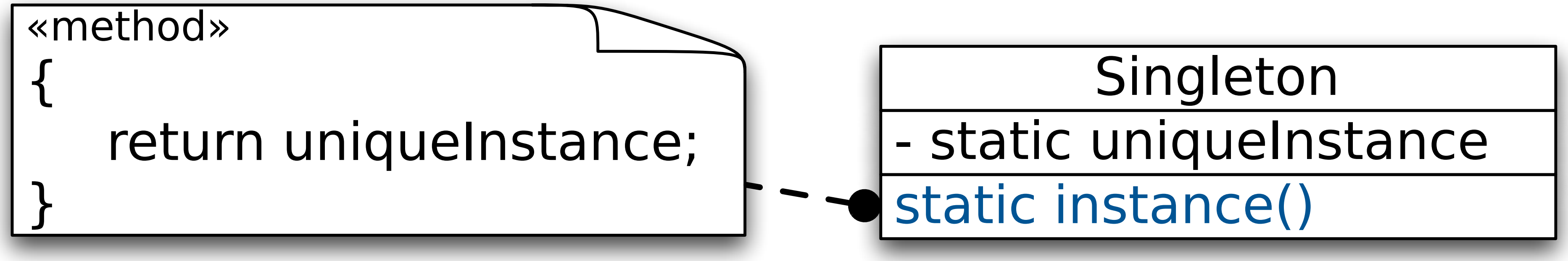
Singleton

Intent

- Ensure a class only has one instance, and provide a global point of access to it.

Singleton

Structure



«method»

```
{
    return uniqueInstance;
}
```

Singleton

- static uniqueInstance

static instance()

Singleton

Implementation

```
public class Singleton {

  private static Singleton theInstance = null;

  private Singleton();

  public static Singleton instance() {
    if (theInstance == null)
      theInstance = new Singleton();
    return theInstance;
  }
}
```

The constructor “should” be private or protected.

The implementation is not thread safe.

Singleton
Benefits

- Cross platform
  Using appropriate middleware, Singleton can be extended to work across many JVMs.
- Applicable to any class
- Can be created through derivation
  Given a class, you can create a subclass that is a Singleton.
- Lazy creation
  (Controlled access to sole instance.)
  If the singleton is never used, it is never created.

Singleton
Issues

- Beware: Often it is best to just create one instance of an object (using the constructor) at program initialization time and to use this object.
- Destruction is undefined
- Not inherited
  A class derived from a singleton is not a singleton.
- Nontransparent
  Users of a Singleton know that they are using a Singleton.

Wrap up

- Visitor Pattern
- Strategy Pattern
- Factory Method Pattern
- Singleton Pattern
- Exercise: Recognizing Patterns in Architecture