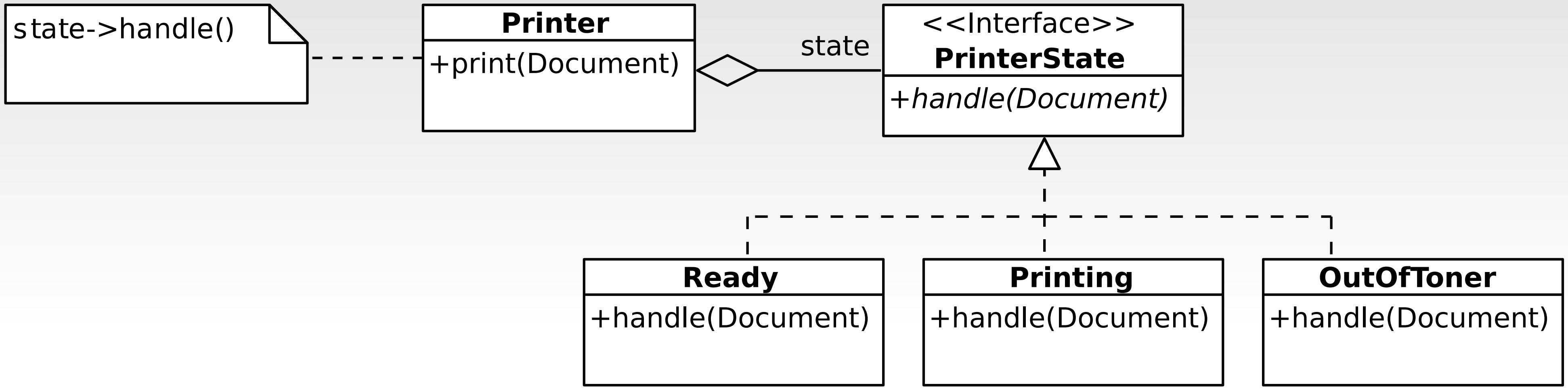
PA103 - Object-oriented Methods for Design of Information Systems

Design Patterns (cont.)

© Radek Ošlejšek
Fakulta informatiky MU
oslejsek@fi.muni.cz

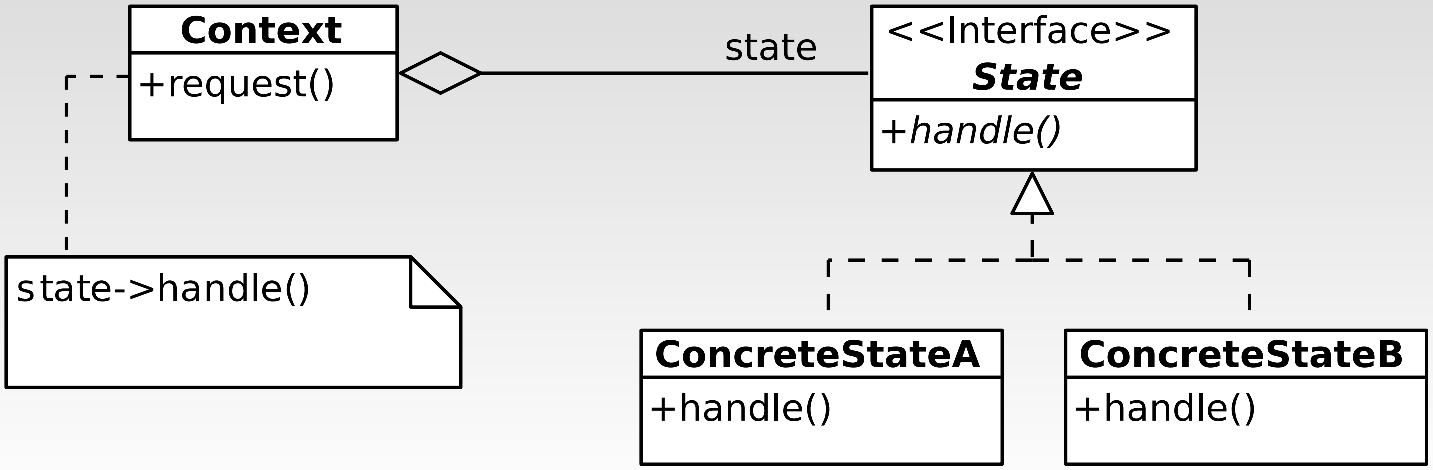
State – Motivation



Visually similar to Strategy and also with similar purpose.

Allow an object to alter its behavior when its internal state changes.

State Pattern



Context delegates state-specific requests to the current ConcreteState object.

A context may pass itself as an argument to the State object handling the request. This lets the State object access the context if necessary.

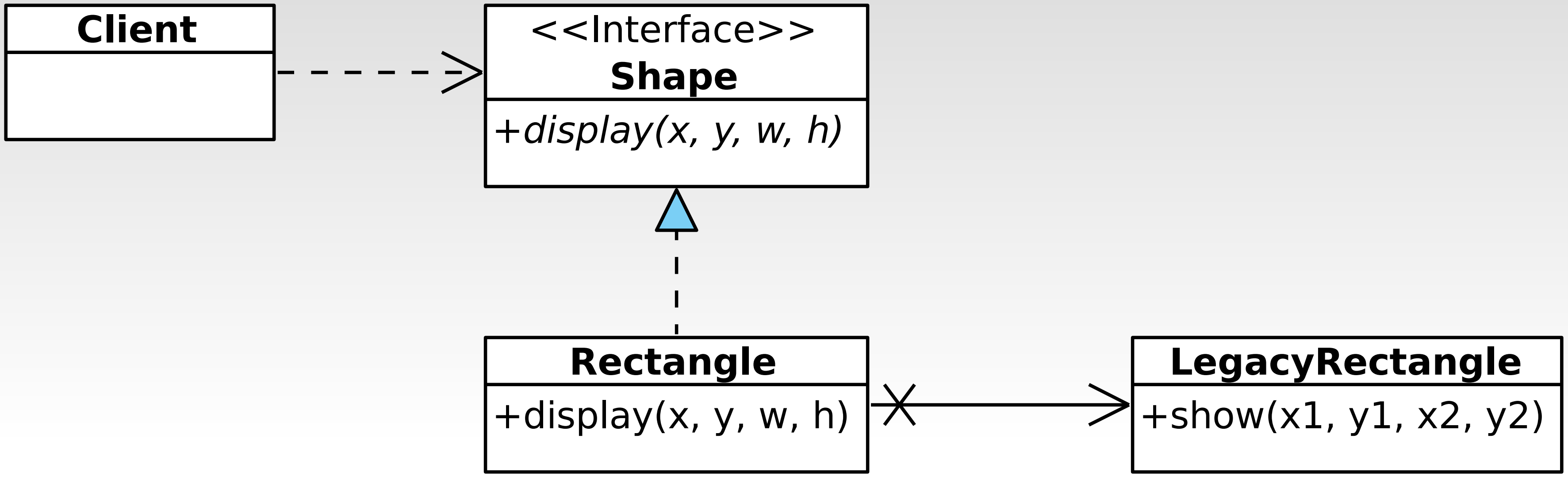
Context is the primary interface for clients. Clients can configure a context with State objects. Once a context is configured, its clients don't have to deal with the State objects directly.

Either Context or the ConcreteState subclasses can decide which state succeeds another and under what circumstances.

State Properties

- **Applicability:** Use State when
  - an object's behavior depends on its state, and it must change its behavior at run-time depending on that state.
  - operations have large, multipart conditional statements that depend on the object's state.
- **Consequences:**
  - It localizes state-specific behavior and partitions behavior for different states.
  - It makes state transitions explicit.
  - State objects can be shared (see Flyweight pattern later).

Adapter – Motivation



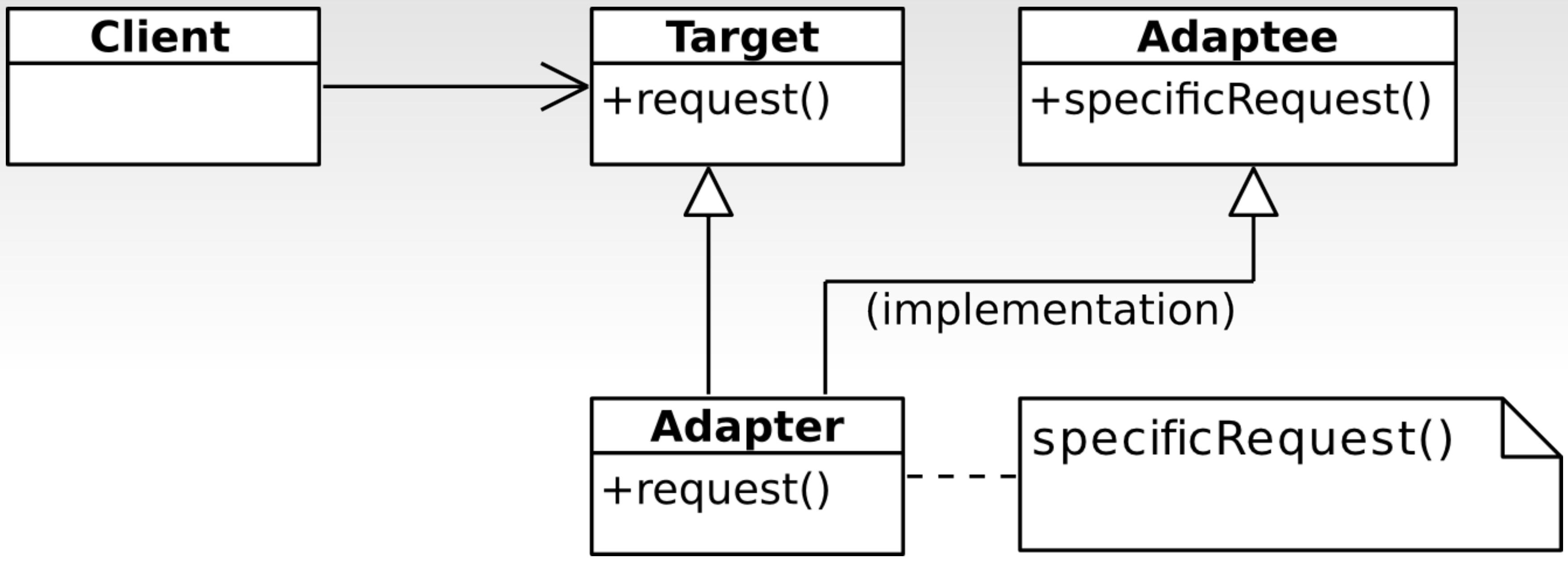
Also known as wrapper.

Converts interface to compatible form.

Similar principle to spanner bit sockets or power plug adapters.

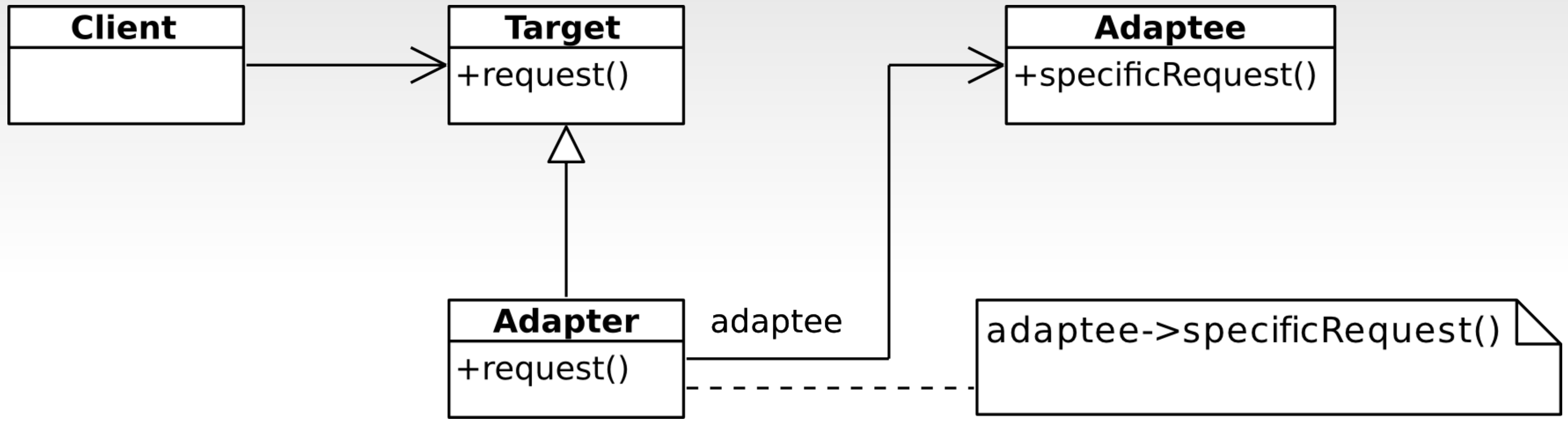
See e.g. java.owt.WindowsAdapter

Class Adapter



request() methods adapted by multiple inheritance.

Object Adapter



Request() methods adapted by re-sending to associated adaptee.

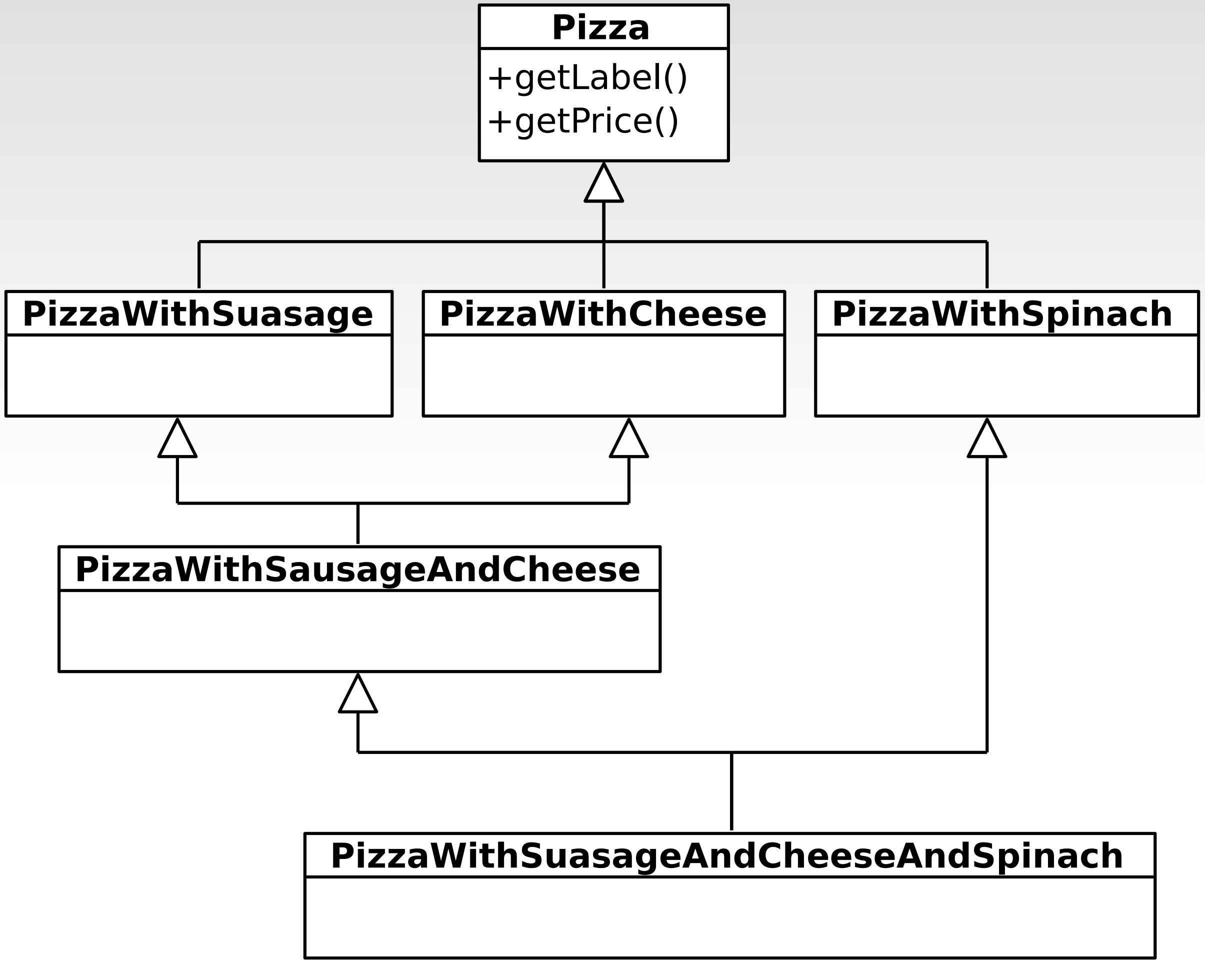
Adapter Properties

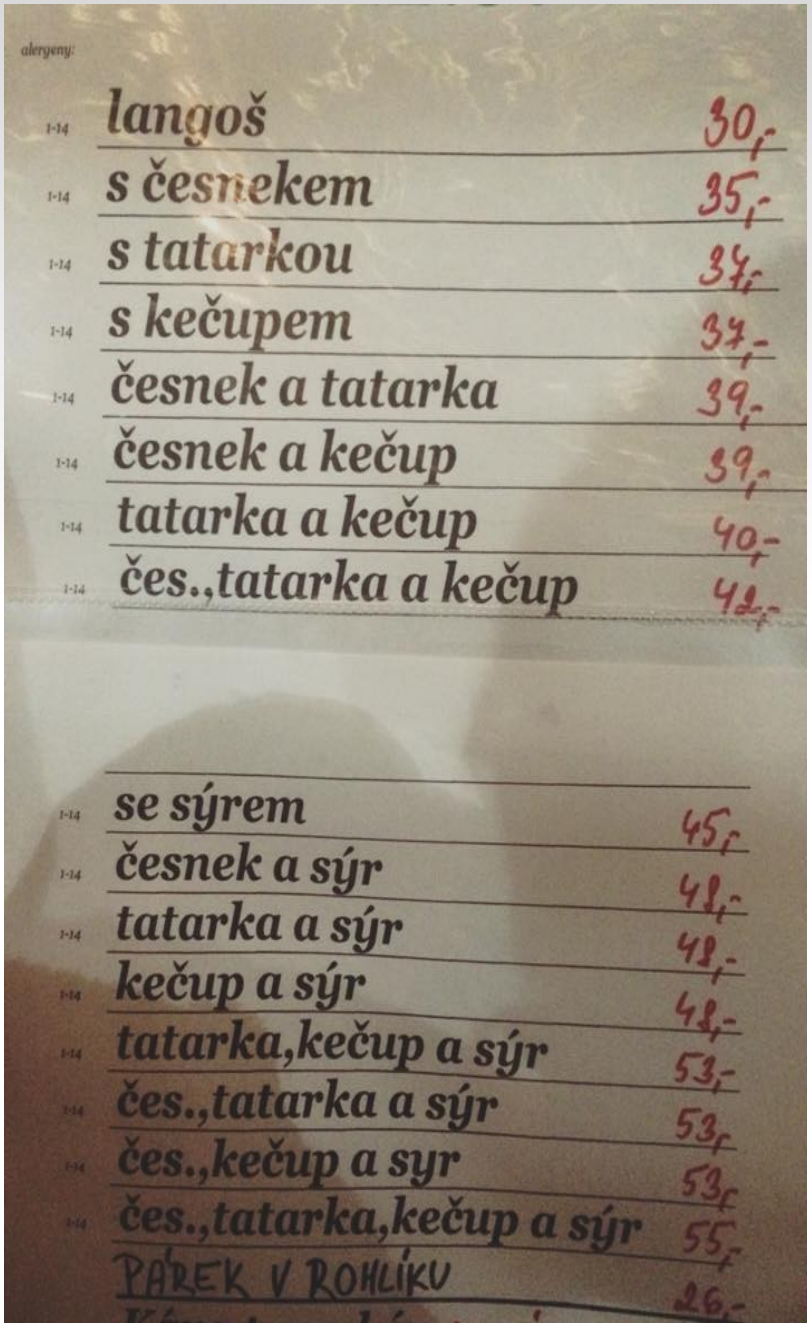
- **Applicability:** Use the Adapter pattern when
    - you want to use an existing class, and its interface does not match the one you need.
    - you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces.
    - **(object adapter only)** you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class.

Adapter Properties (cont.)

- **Consequences (class adapter):**
  - adapts Adaptee to Target by committing to a concrete Adapter class. As a consequence, a class adapter won't work when we want to adapt a class and all its subclasses.
  - lets Adapter override some of Adaptee's behavior, since Adapter is a subclass of Adaptee.
  - introduces only one object, and no additional pointer indirection is needed to get to the adaptee.
- **Consequences (object adapter):**
  - lets a single Adapter work with many Adaptees – that is, the Adaptee itself and all of its subclasses (if any).
  - makes it harder to override Adaptee behavior. It will require subclassing Adaptee and making Adapter refer to the subclass rather than the Adaptee itself.

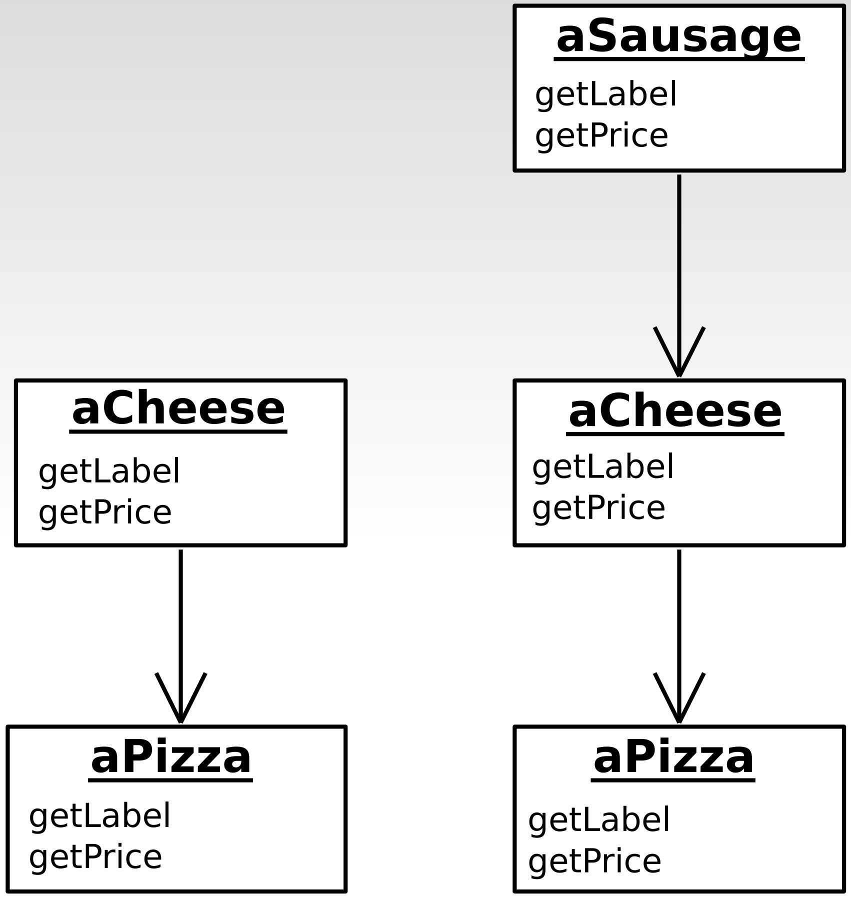
Decorator Pattern – Another Example

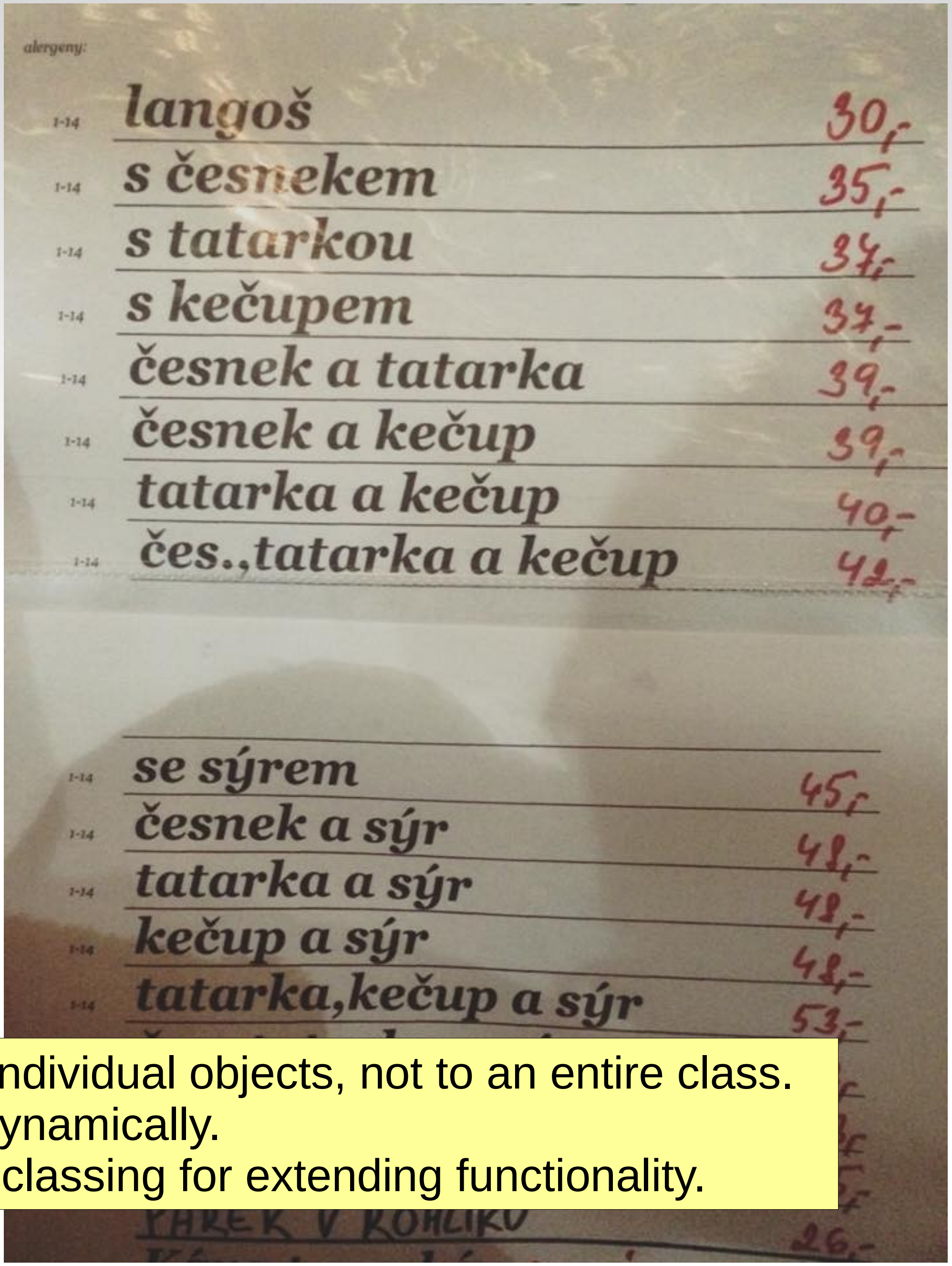




alergeny:		
1-14	langoš	30,-
1-14	s česnekem	35,-
1-14	s tatarkou	37,-
1-14	s kečupem	37,-
1-14	česnek a tatarka	39,-
1-14	česnek a kečup	39,-
1-14	tatarka a kečup	40,-
1-14	čes.,tatarka a kečup	42,-
1-14	se sýrem	45,-
1-14	česnek a sýr	48,-
1-14	tatarka a sýr	48,-
1-14	kečup a sýr	48,-
1-14	tatarka,kečup a sýr	53,-
1-14	čes.,tatarka a sýr	53,-
1-14	čes.,kečup a syr	53,-
1-14	čes.,tatarka,kečup a sýr	55,-
	PÁREK V ROHLÍKU	26,-

Decorator Pattern – Another Example



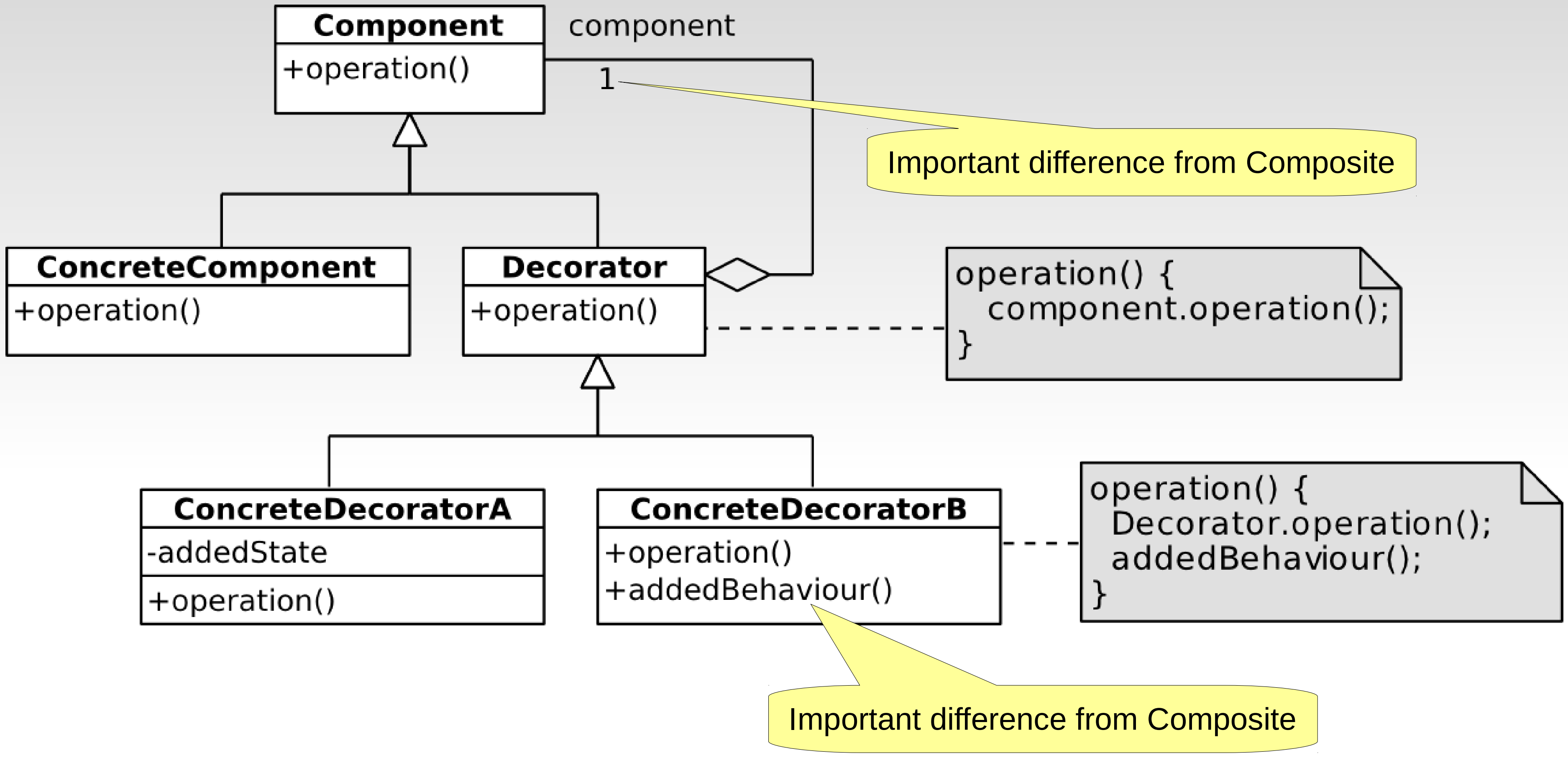


alergeny:

1-14	langoš	30,-
1-14	s česnekem	35,-
1-14	s tatarkou	37,-
1-14	s kečupem	37,-
1-14	česnek a tatarka	39,-
1-14	česnek a kečup	39,-
1-14	tatarka a kečup	40,-
1-14	čes.,tatarka a kečup	42,-
1-14	se sýrem	45,-
1-14	česnek a sýr	48,-
1-14	tatarka a sýr	48,-
1-14	kečup a sýr	48,-
1-14	tatarka,kečup a sýr	53,-
	PÁREK V ROHLÍKU	26,-

Sometimes we want to add responsibilities to individual objects, not to an entire class.
Attach additional responsibilities to an object dynamically.
Decorators provide a flexible alternative to subclassing for extending functionality.

Decorator Pattern



```
operation() {
  component.operation();
}
```

```
operation() {
  Decorator.operation();
  addedBehaviour();
}
```

Decorator subclasses are free to add operations for specific functionality.

The important aspect of this pattern is that it lets decorators appear anywhere a Component can.

Decorator Properties

- **Applicability:** Use Decorator
  - to add responsibilities to individual objects dynamically and transparently, that is without affecting other objects.
  - for responsibilities that can be withdrawn.
  - when extension by subclassing is impractical. Sometimes a large number of independent extensions are possible and would produce an explosion of subclasses to support every combination.
- **Consequences:**
  - More flexibility than static inheritance.
  - Avoids feature-laden classes high up in the hierarchy. Instead of trying to support all foreseeable features in a complex, customizable class, you can define a simple class and add functionality incrementally with Decorator objects.
  - Liability: A decorator and its component aren't identical.
  - Liability: Lots of little objects.

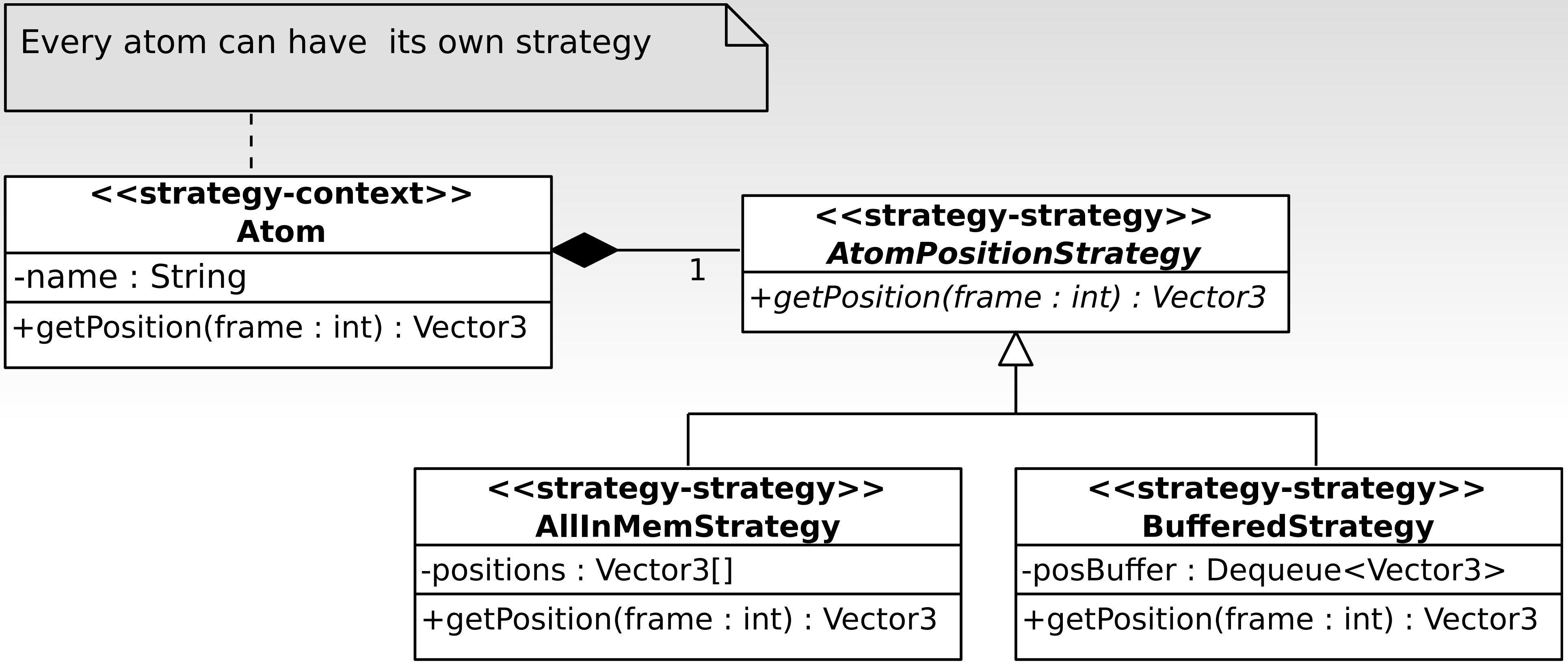
Decorator – Related Patterns and Applications

- **Adapter:** A decorator is different from an adapter in that a decorator only changes an object's responsibilities, not its interface; an adapter will give an object a completely new interface.
- **Composite:** A decorator can be viewed as a degenerate composite with only one component. However, a decorator adds additional responsibilities—it isn't intended for object aggregation.
- **Strategy:** A decorator lets you change the skin of an object; a strategy lets you change the guts. These are two alternative ways of changing an object.

- **Question:** Which standard well-known part of Java API is designed as Decorator?

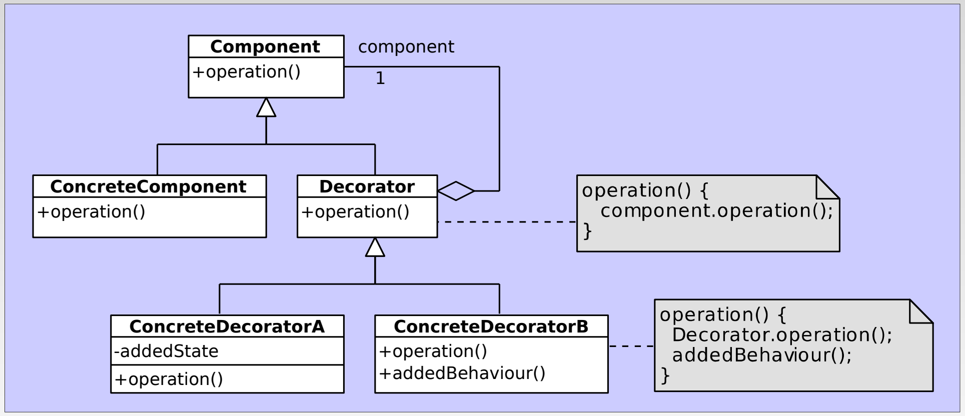
java.io.BufferedReader, ...

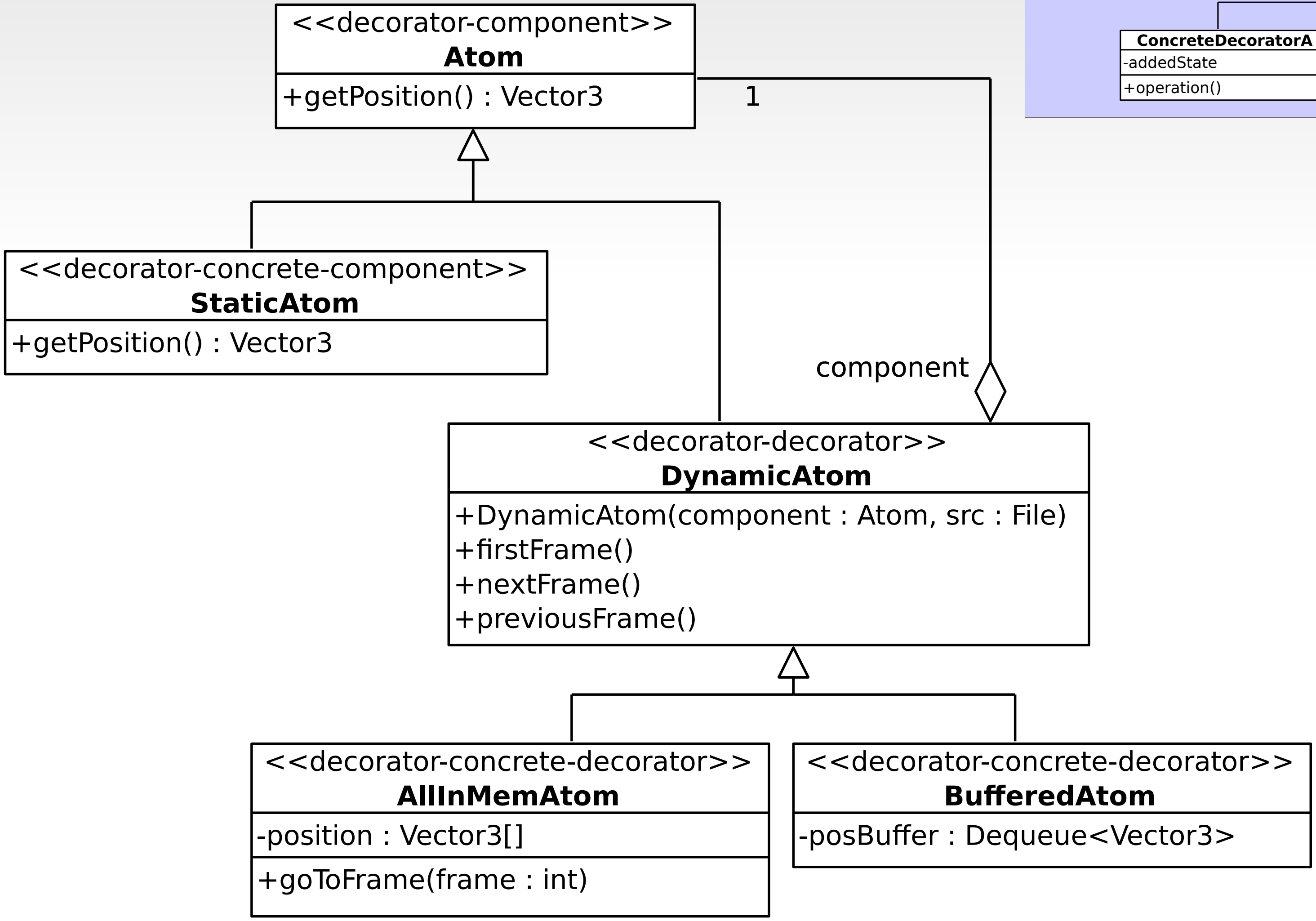
Decorator – Case Study


Every atom can have its own strategy
<<strategy-context>>
Atom
-name : String
+getPosition(frame : int) : Vector3
1
<<strategy-strategy>>
AtomPositionStrategy
+getPosition(frame : int) : Vector3
<<strategy-strategy>>
AllInMemStrategy
-positions : Vector3[]
+getPosition(frame : int) : Vector3
<<strategy-strategy>>
BufferedStrategy
-posBuffer : Dequeue<Vector3>
+getPosition(frame : int) : Vector3

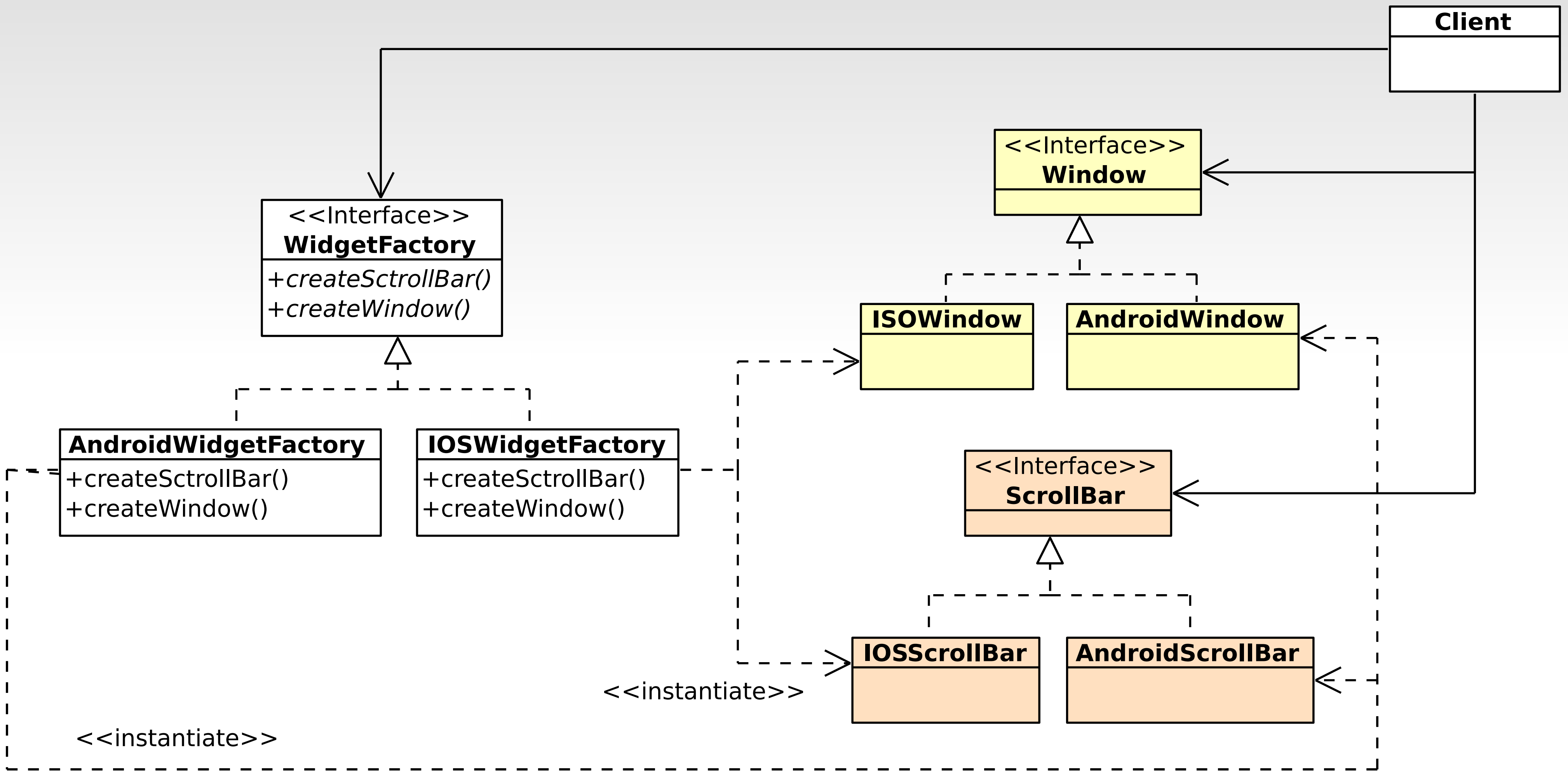
This is our previous model of "shaking" atoms decomposed by Strategy pattern.
How it could come off when we apply Decorator pattern instead?

Decorator – Case Study (cont.)

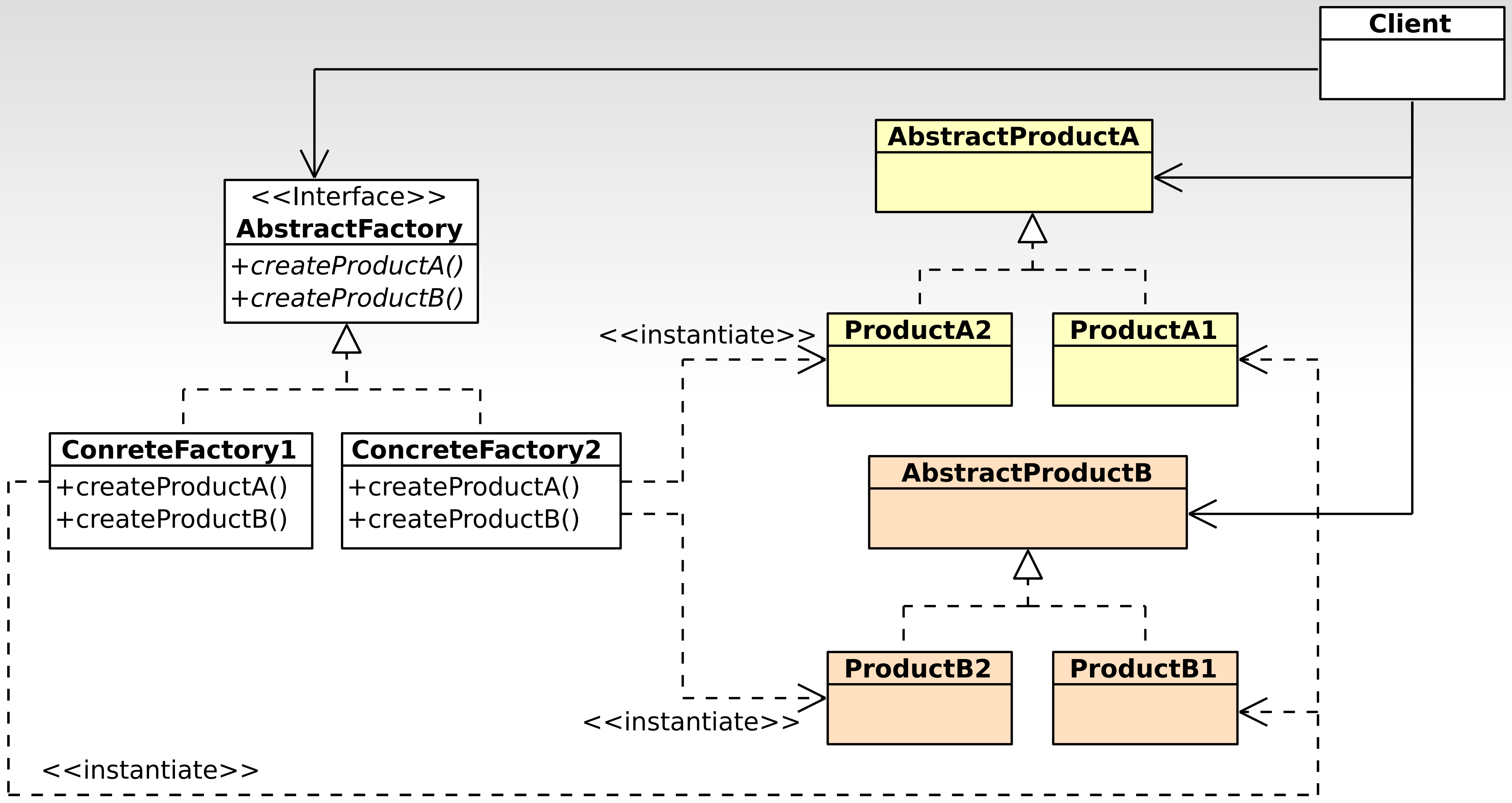




Abstract Factory – Motivation



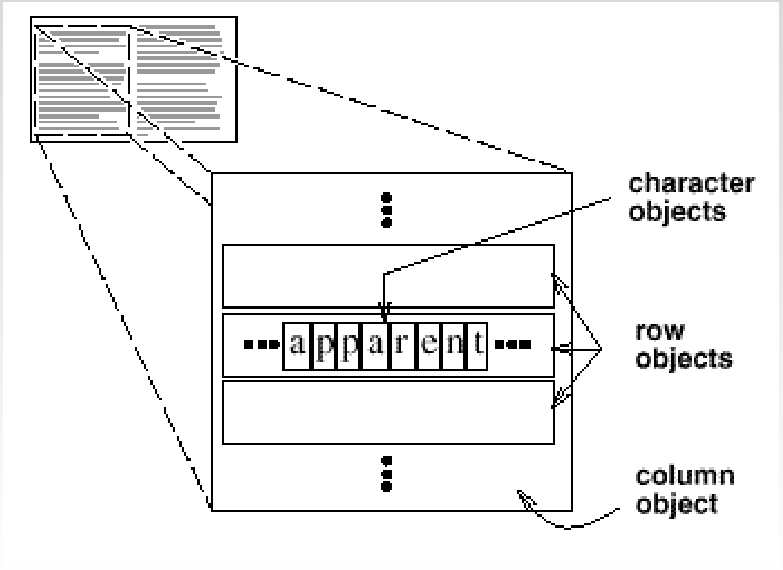
Abstract Factory Pattern

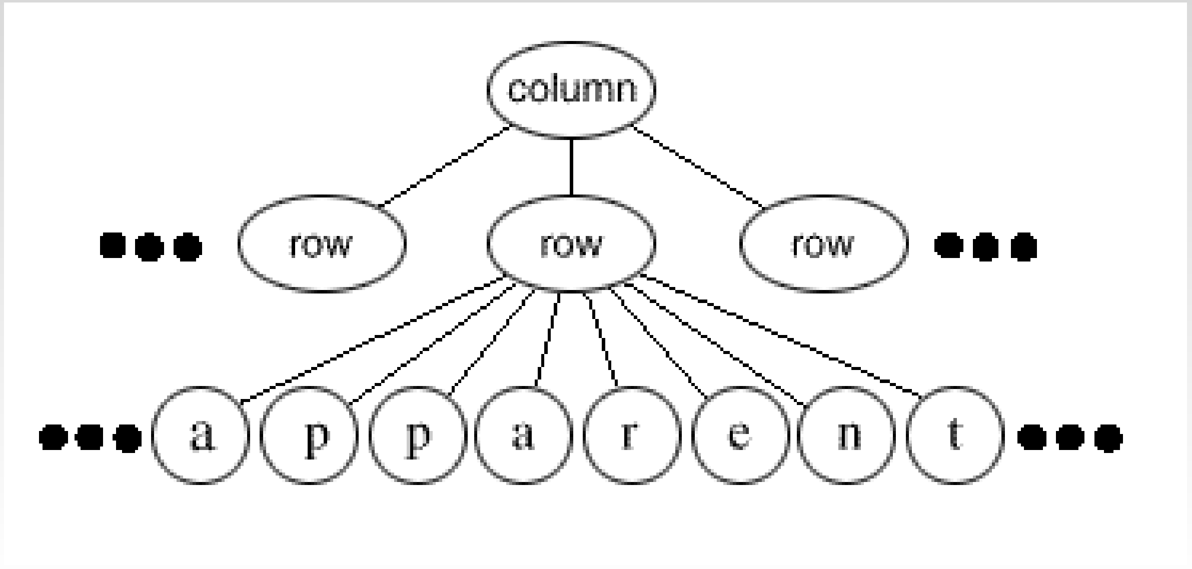


Abstract Factory Properties

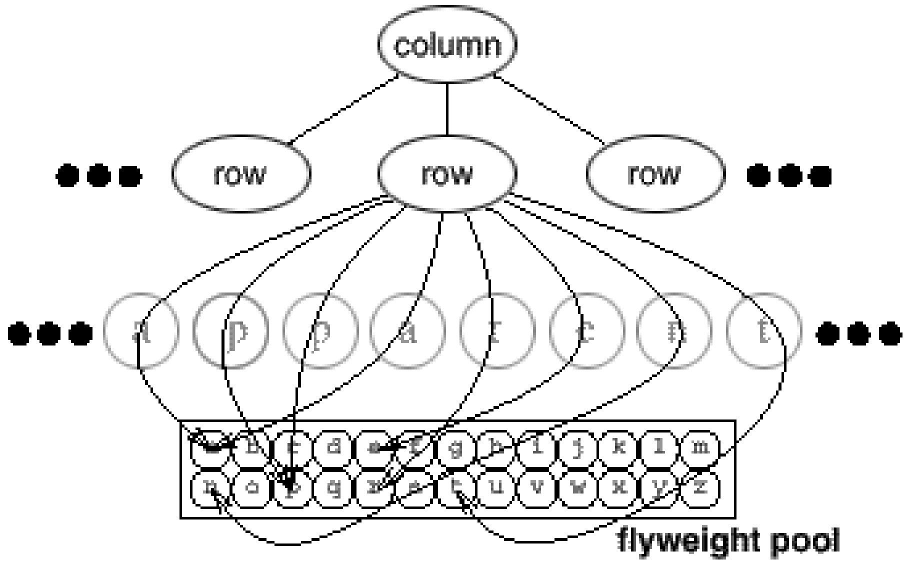
- **Applicability:** Use the Abstract Factory pattern when
  - A system should be independent of how its products are created, composed, and represented.
  - A system should be configured with one of multiple families of products.
  - A family of related product objects is designed to be used together, and you need to enforce this constraint.
  - You want to provide a class library of products, and you want to reveal just their interfaces, not their implementations.
- **Consequences:**
  - It isolates clients from implementation classes.
  - It makes exchanging product families easy.
  - It promotes consistency among products.
  - Supporting new kinds of products is difficult.

Flyweight Pattern – Motivation

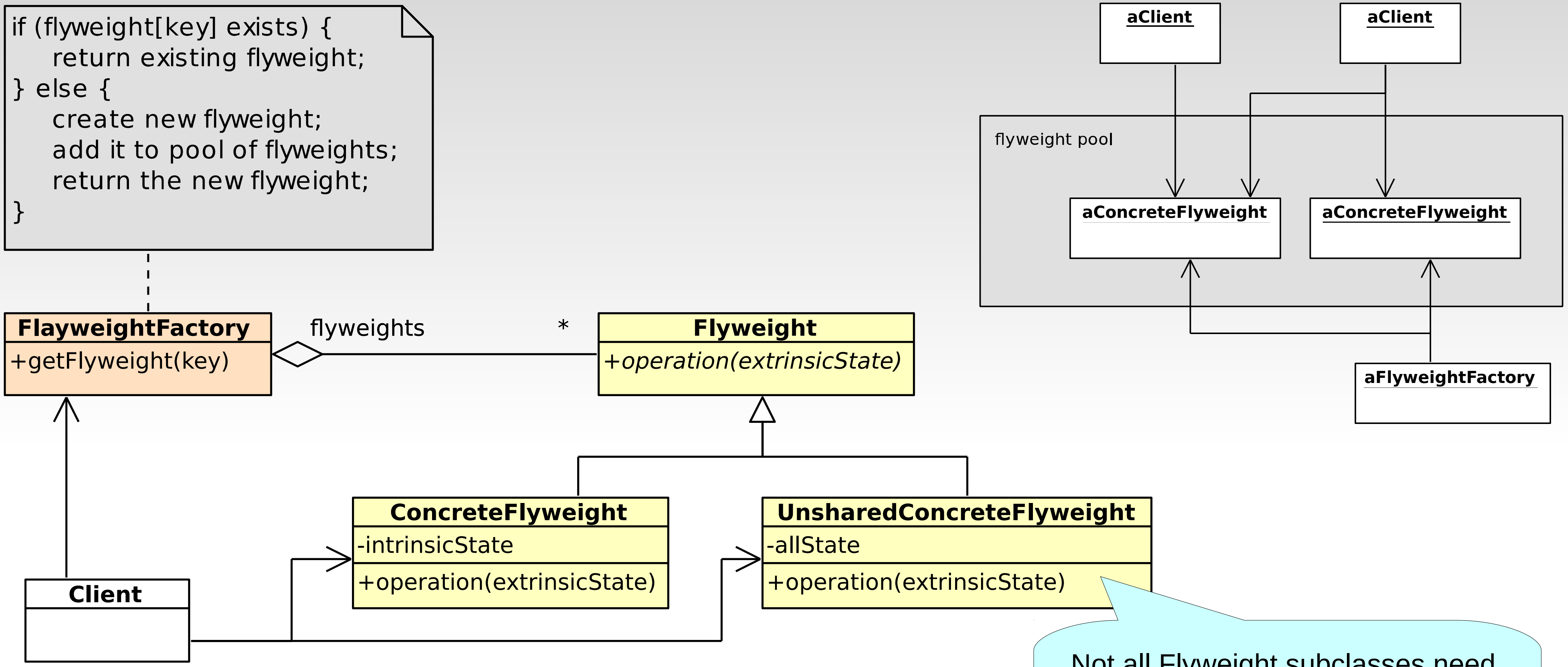




+ Flexibility at the finest levels, e.g. character set assign to each character.
- Even moderate-sized documents may require hundreds of thousands of character objects.



Flyweight Pattern



```
if (flyweight[key] exists) {
    return existing flyweight;
} else {
    create new flyweight;
    add it to pool of flyweights;
    return the new flyweight;
}
```

Not all Flyweight subclasses need to be shared, e.g. row/column. The Flyweight interface enables sharing, it doesn't enforce it.

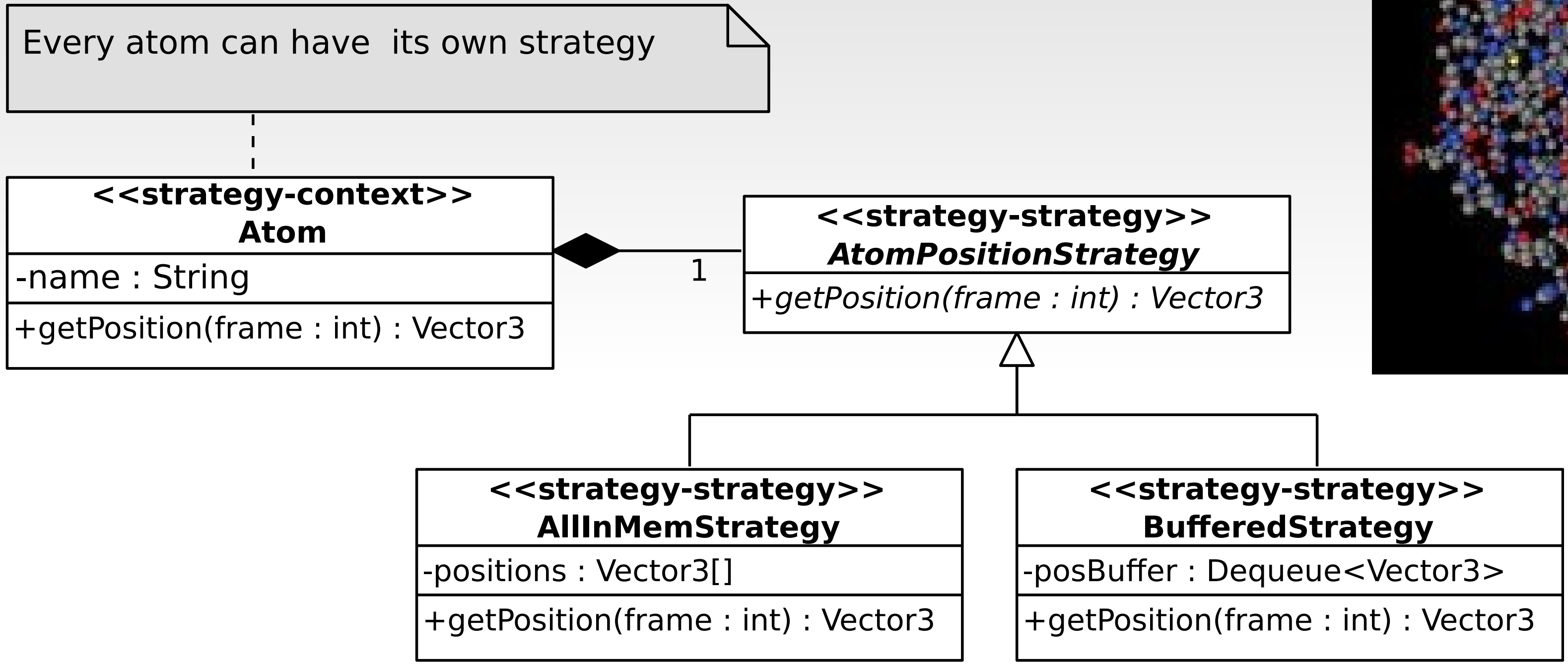
A flyweight is a shared object that can be used in multiple contexts simultaneously.

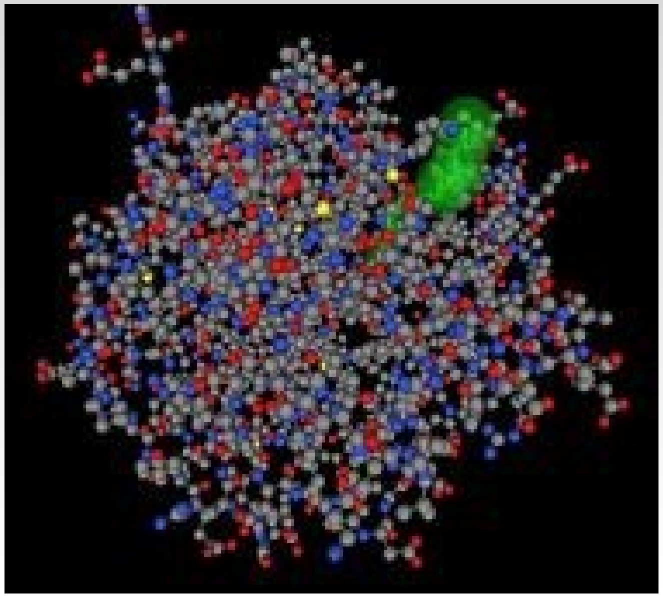
Intrinsic state is stored in the flyweight; it consists of information that's independent of the flyweight's context, thereby making it sharable.
Extrinsic state depends on and varies with the flyweight's context and therefore can't be shared.
Client objects are responsible for passing extrinsic state to the flyweight when it needs it.

Flyweight Properties

- **Applicability:** Apply the Flyweight pattern when **all** of the following are true:
  - An application uses a large number of objects.
  - Storage costs are high because of the sheer quantity of objects.
  - Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.
  - The application doesn't depend on object identity. Since flyweight objects may be shared, identity tests will return true for conceptually distinct objects.
- **Consequences:**
  - Flyweights may introduce run-time costs associated with transferring, finding, and/or computing extrinsic state, especially if it was formerly stored as intrinsic state. However, such costs are offset by space savings.
  - Storage savings are a function of several factors:
    - the reduction in the total number of instances that comes from sharing
    - the amount of intrinsic state per object
    - whether extrinsic state is computed or stored.
  - The more flyweights are shared, the greater the storage savings.
  - The Flyweight pattern is often combined with the *Composite* pattern to represent a hierarchical structure as a graph with shared leaf nodes.

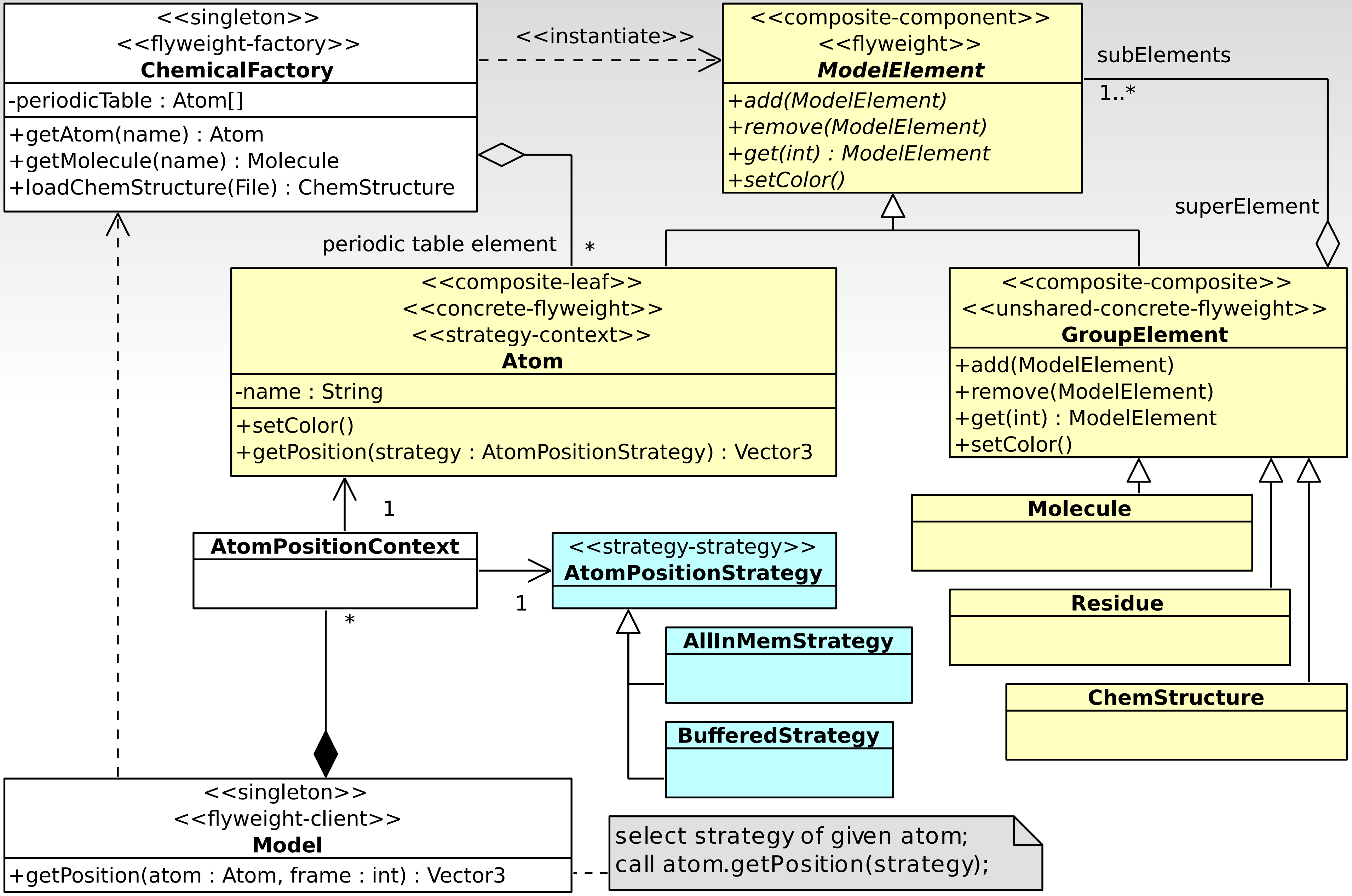
Flyweight – Case Study


Every atom can have its own strategy
<<strategy-context>>
Atom
-name : String
+getPosition(frame : int) : Vector3
1
<<strategy-strategy>>
AtomPositionStrategy
+getPosition(frame : int) : Vector3
<<strategy-strategy>>
AllInMemStrategy
-positions : Vector3[]
+getPosition(frame : int) : Vector3
<<strategy-strategy>>
BufferedStrategy
-posBuffer : Dequeue<Vector3>
+getPosition(frame : int) : Vector3

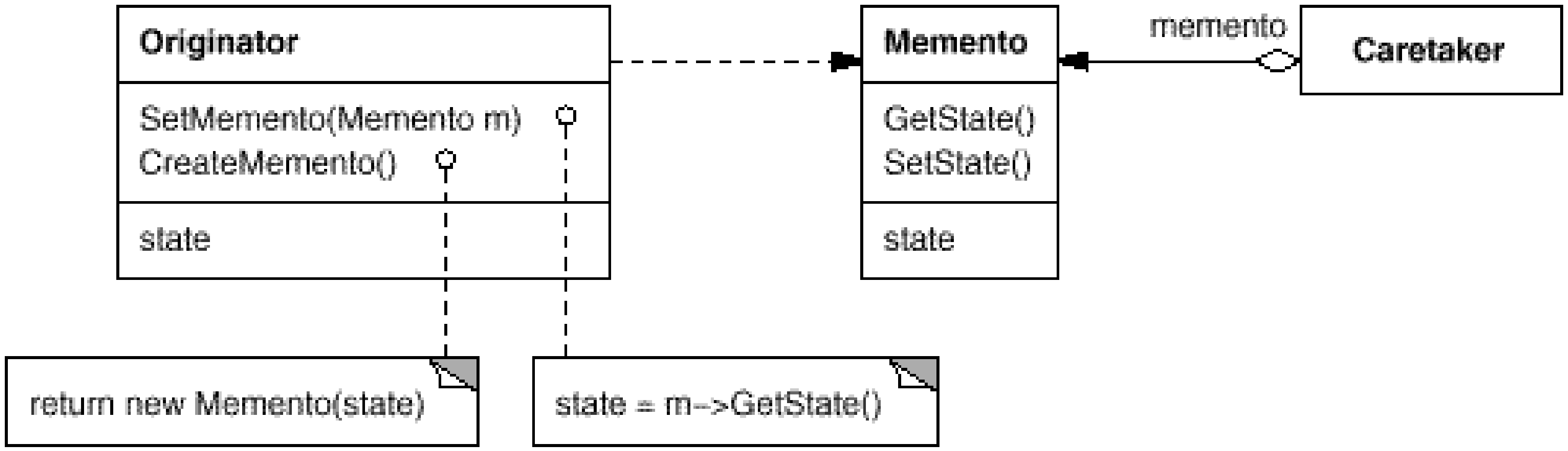


Problem:
Typical molecule consist of thousands of atoms of a few types, e.g. Oxygen and Hydrogen. Instantiating individual object for every single atom waste the memory because some pieces of information can be shared, e.g. atom name, charge, temperature factor, etc. On the other hand, some data are specific for each atom, e.g. the 3D position (calculated by means of the position strategy).

Flyweight – Case Study

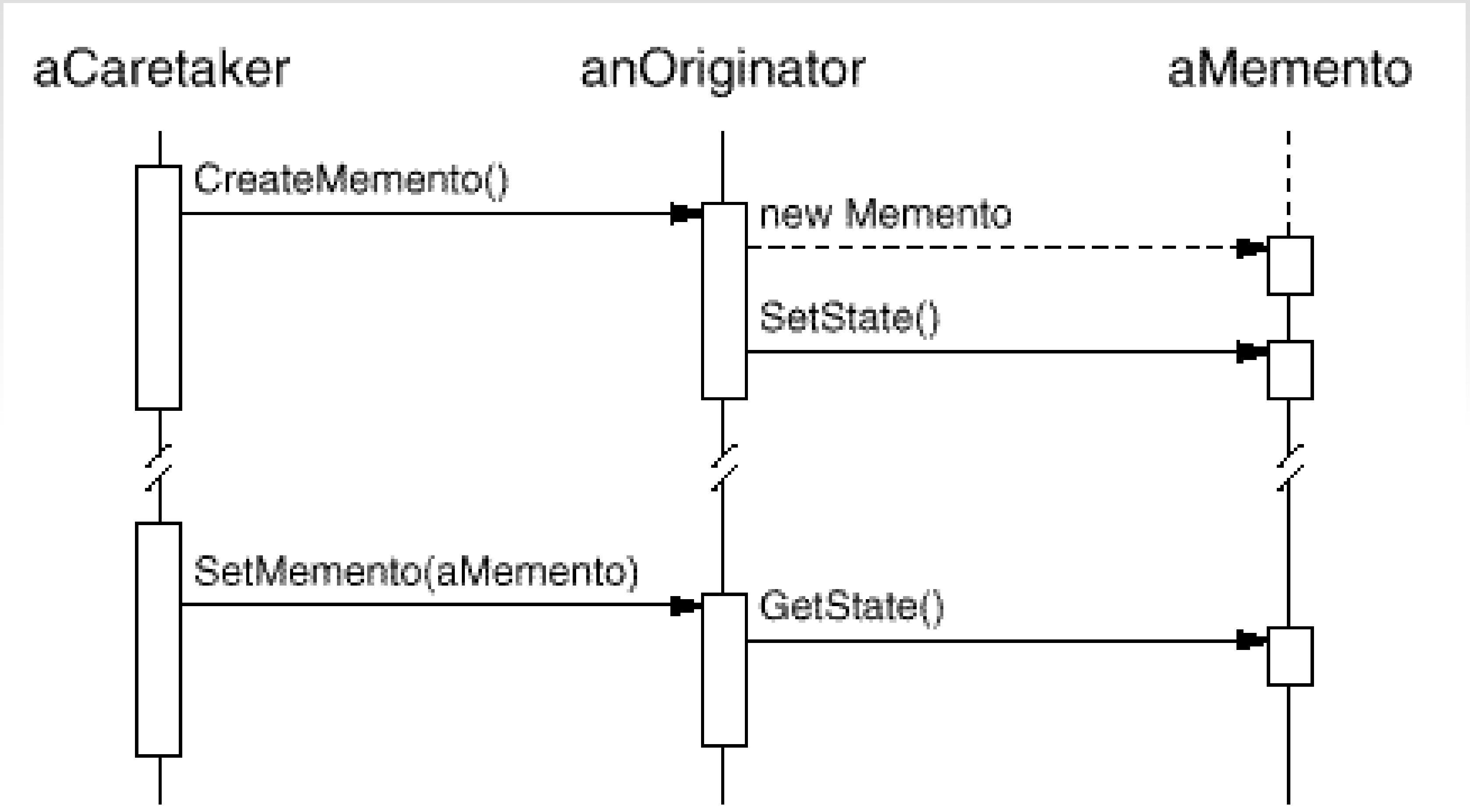


Memento Pattern



capture and externalize an object's internal state so that the object can be restored to this state later

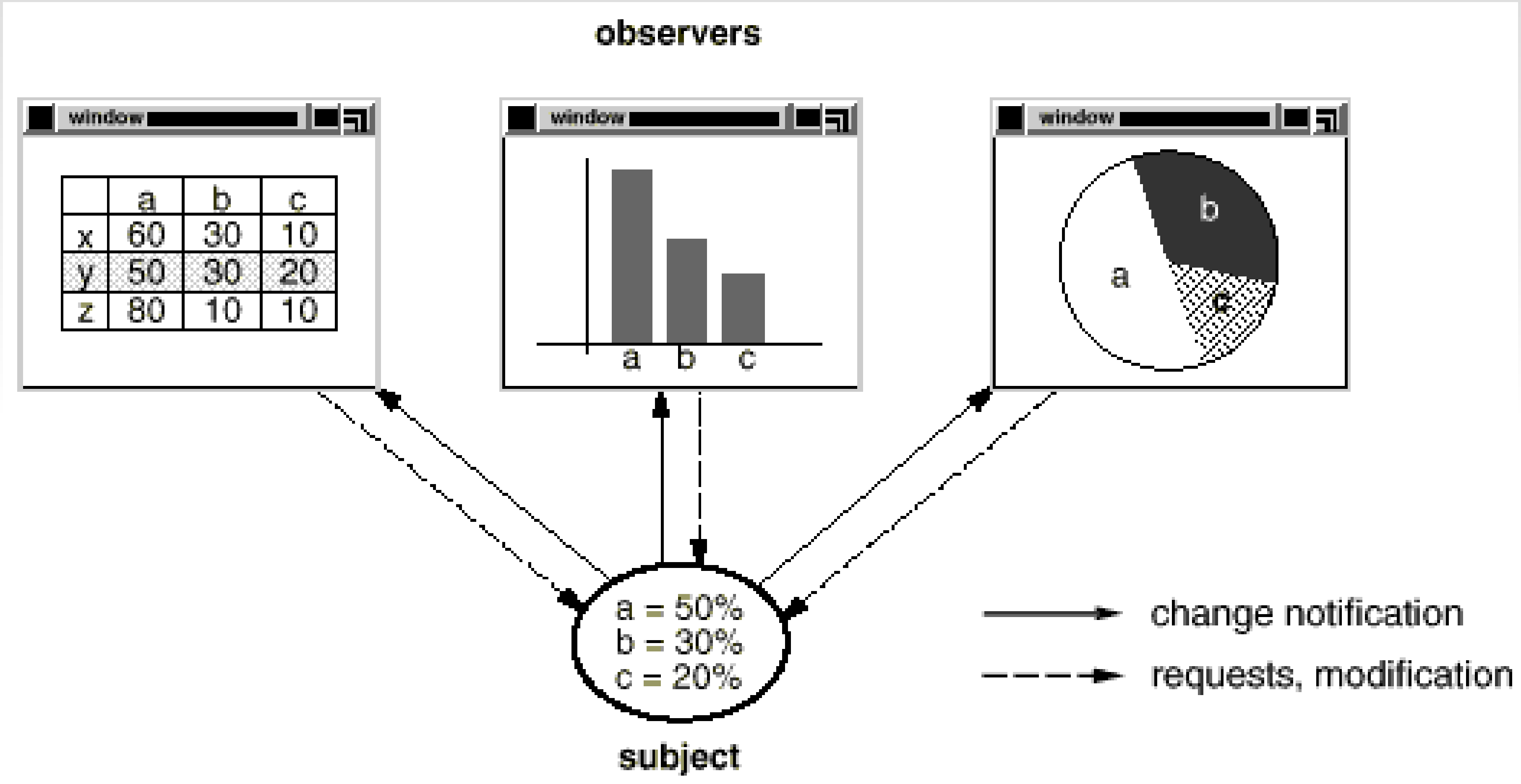
Memento - cooperation



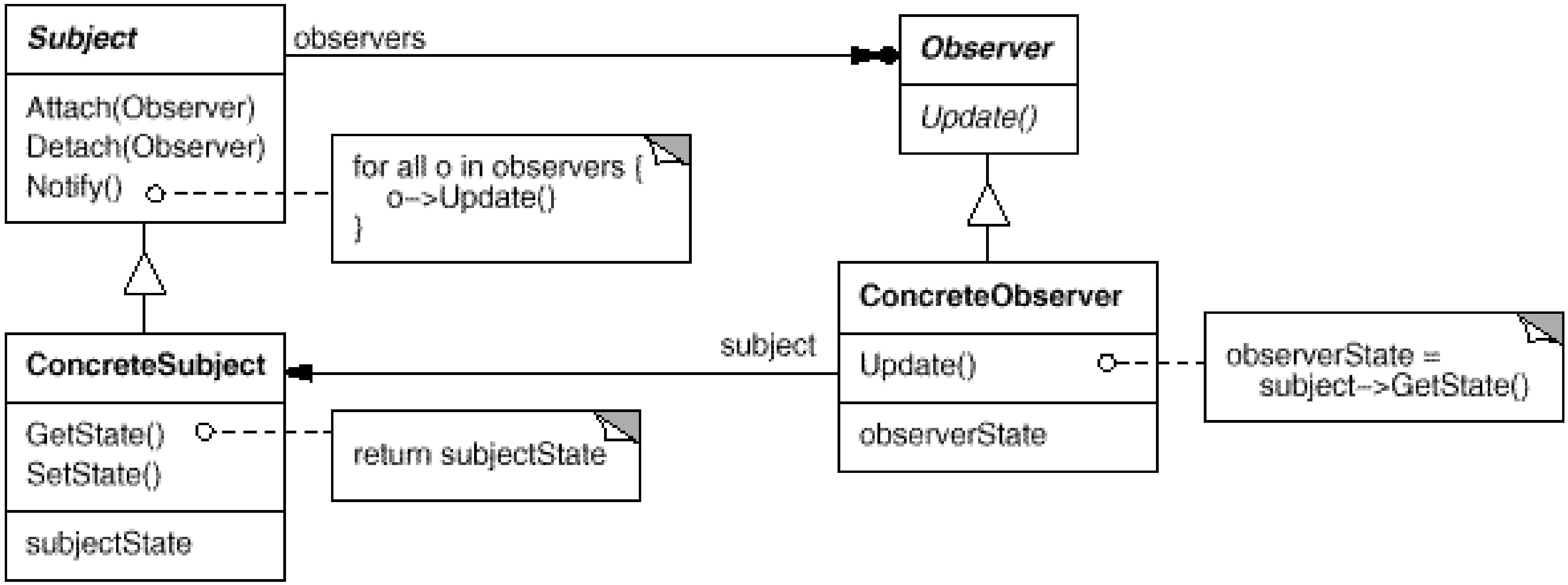
Memento - properties

- **Applicability:** Use the Memento pattern when
  - a snapshot of (some portion of) an object's state must be saved so that it can be restored to that state later, **and**
  - a direct interface to obtaining the state would expose implementation details and break the object's encapsulation.
- **Consequences:**
  - Preserving encapsulation boundaries.
  - It simplifies Originator. In other encapsulation-preserving designs, Originator keeps the versions of internal state that clients have requested. That puts all the storage management burden on Originator.
  - Using mementos might be expensive. Mementos might incur considerable overhead if Originator must copy large amounts of information to store in the memento or if clients create and return mementos to the originator often enough.
  - Defining narrow and wide interfaces. It may be difficult in some languages to ensure that only the originator can access the memento's state.
  - Hidden costs in caring for mementos. A caretaker is responsible for deleting the mementos it cares for. However, the caretaker has no idea how much state is in the memento.

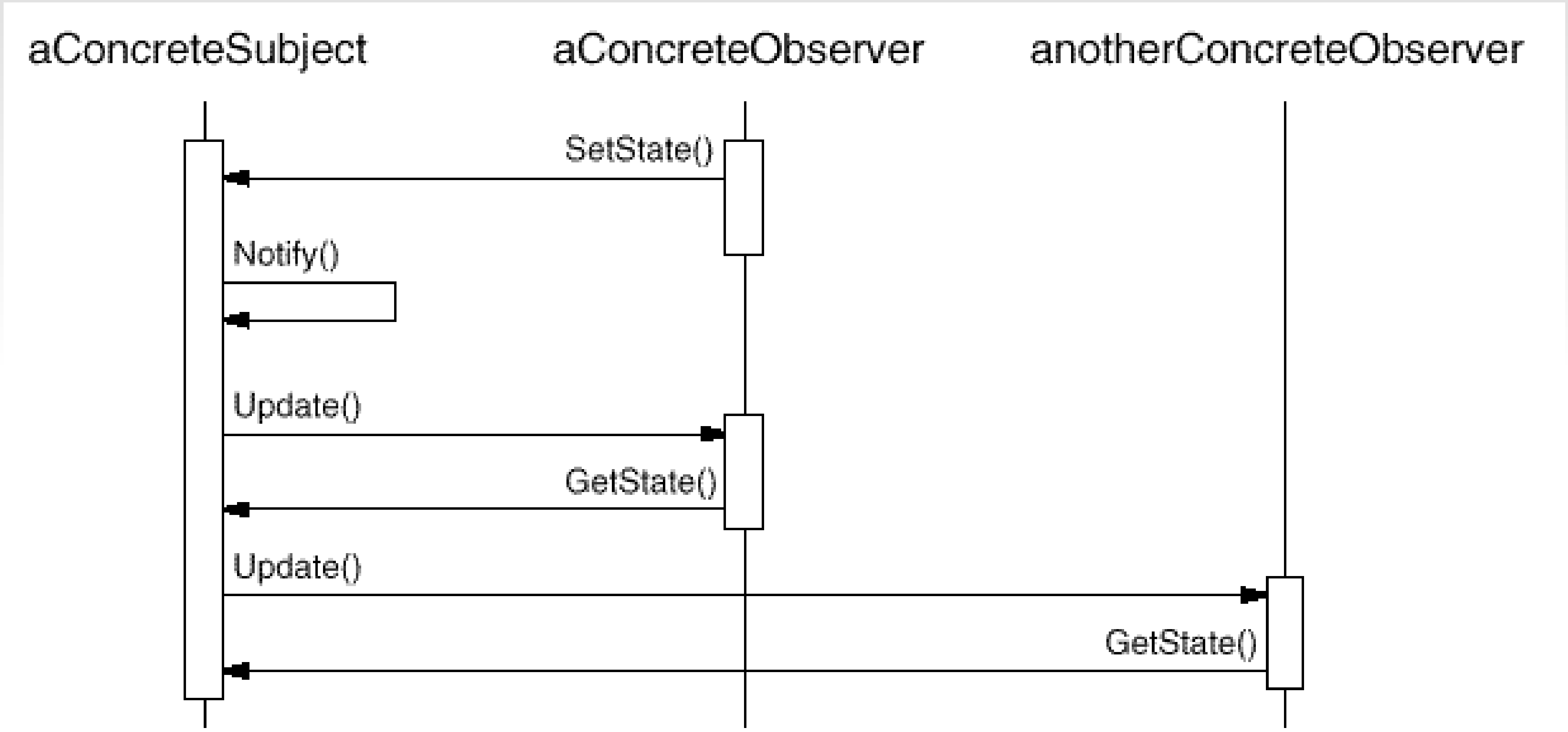
Observer – Motivation



Observer Pattern



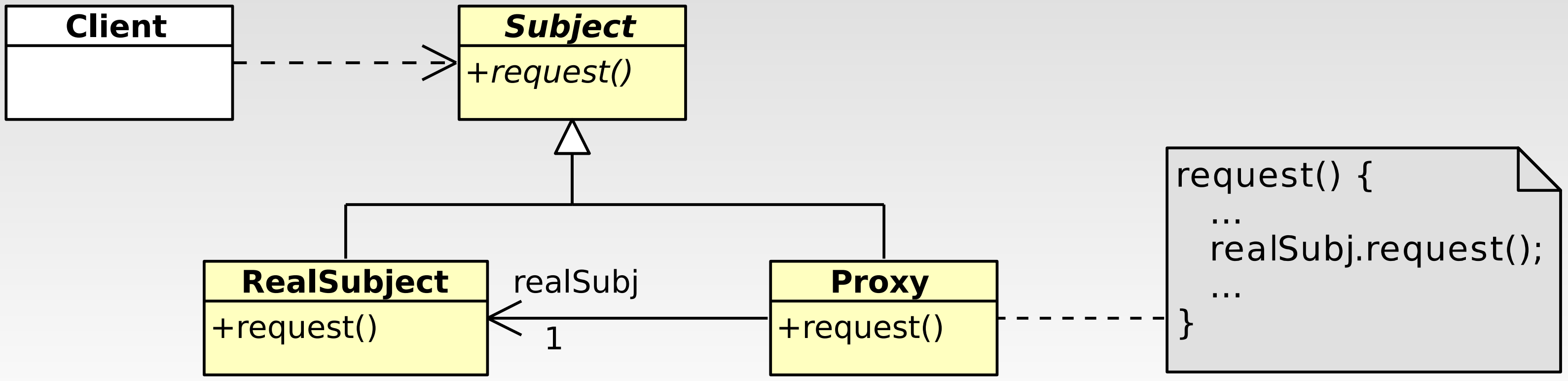
Observer – Collaborations

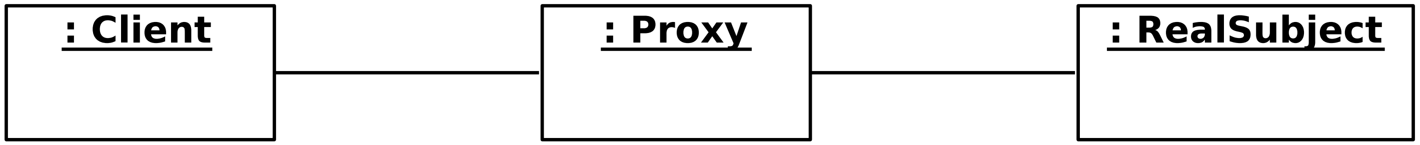


Observer Properties

- **Applicability:** Use the Observer pattern in any of the following situations:
  - When an abstraction has two aspects, one dependent on the other. Encapsulating these aspects in separate objects lets you vary and reuse them independently.
  - When a change to one object requires changing others, and you don't know how many objects need to be changed.
  - When an object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.
- **Consequences:**
  - Abstract coupling between Subject and Observer. All a subject knows is that it has a list of observers, each conforming to the simple interface of the abstract Observer class.
  - Support for broadcast communication.
  - Unexpected updates. Because observers have no knowledge of each other's presence, they can be blind to the ultimate cost of changing the subject. A seemingly innocuous operation on the subject may cause a cascade of updates to observers and their dependent objects.

Proxy Pattern



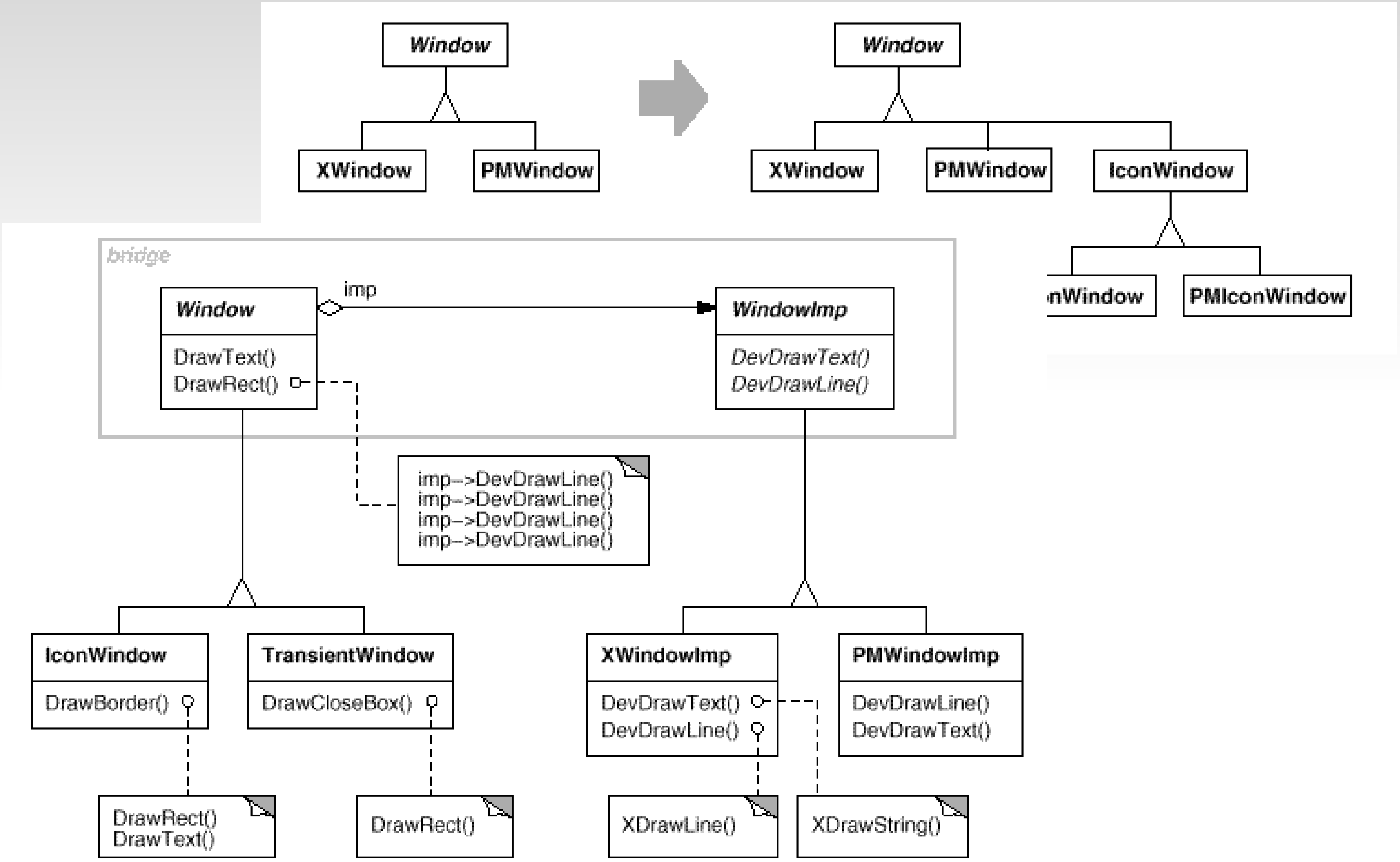


Example: mobile applications

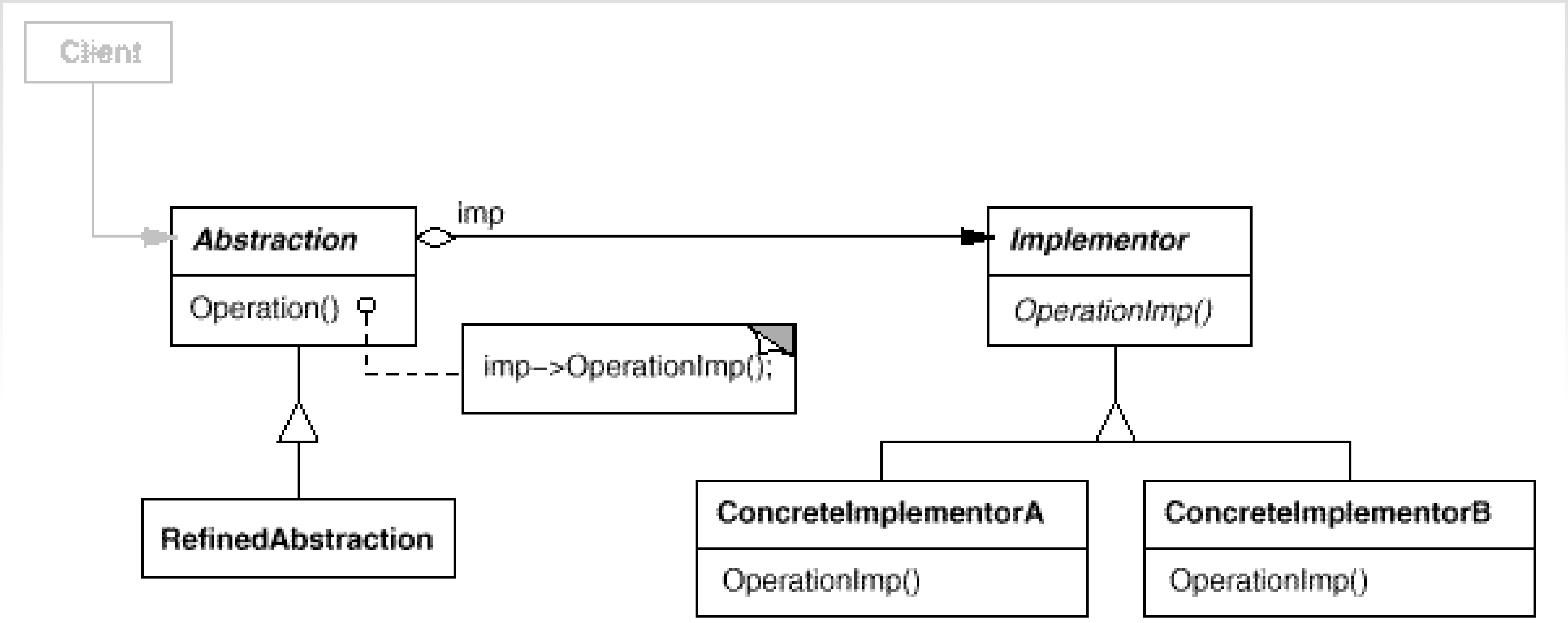
Proxy Properties

- **Applicability:** Proxy is applicable whenever there is a need for a more versatile or sophisticated reference to an object than a simple pointer. Here are several common situations in which the Proxy pattern is applicable:
  - A remote proxy provides a local representative for an object in a different address space.
  - A virtual proxy creates expensive objects on demand.
  - A protection proxy controls access to the original object.
  - A smart reference is a replacement for a bare pointer that performs additional actions when an object is accessed.
- **Consequences:** The Proxy pattern introduces a level of indirection when accessing an object. The additional indirection has many uses, depending on the kind of proxy:
  - A remote proxy can hide the fact that an object resides in a different address space.
  - A virtual proxy can perform optimizations such as creating an object on demand.
  - Both protection proxies and smart references allow additional housekeeping tasks when an object is accessed.

Bridge – Motivation



Bridge Pattern

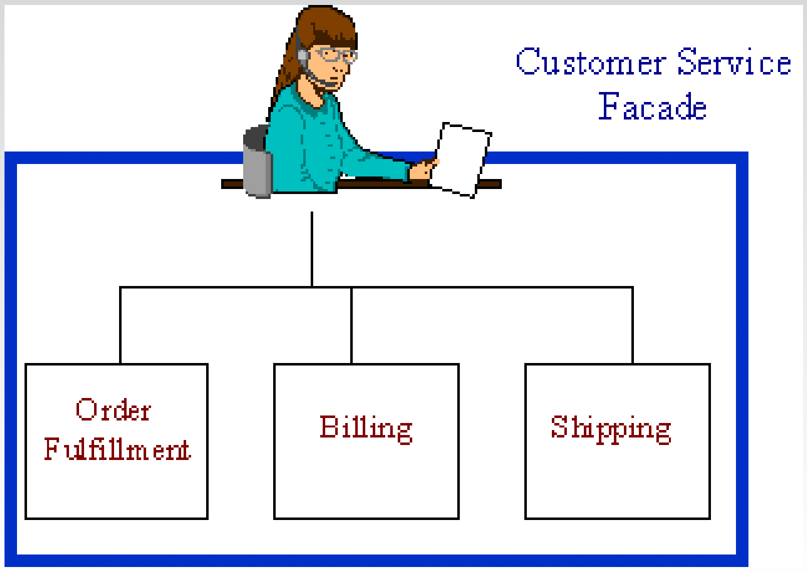


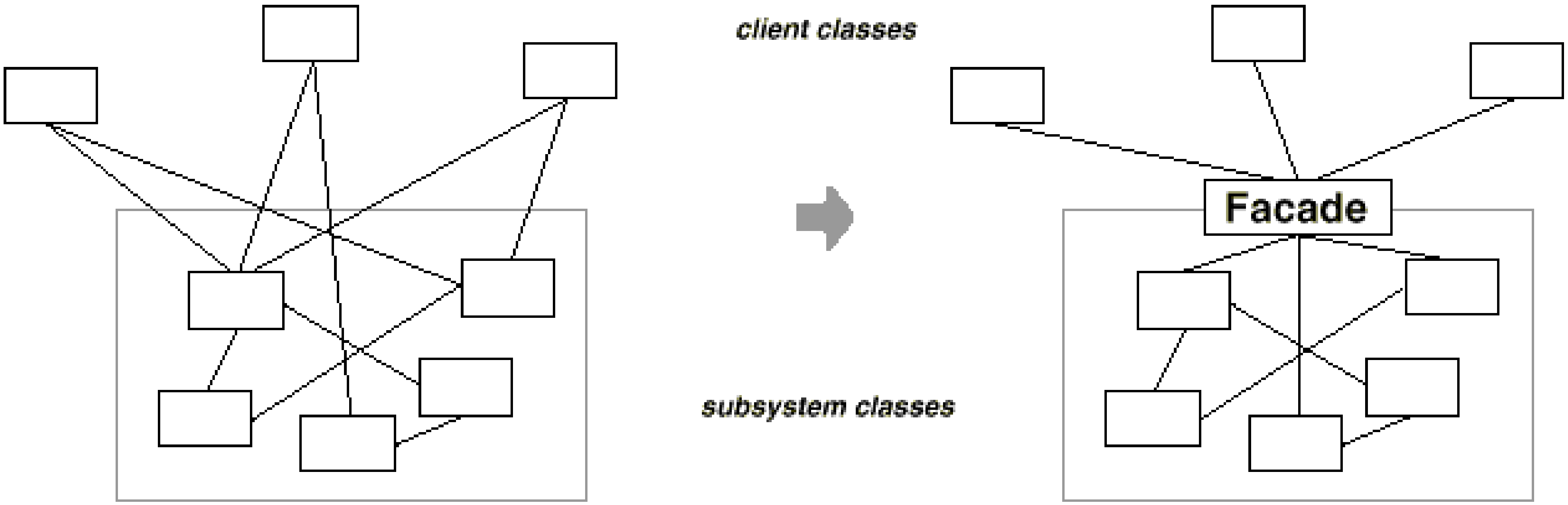
Bridge Properties

- **Applicability:** Use the Bridge pattern when:
    - you want to avoid a permanent binding between an abstraction and its implementation. This might be the case, for example, when the implementation must be selected or switched at run-time.
    - both the abstractions and their implementations should be extensible by subclassing. In this case, the Bridge pattern lets you combine the different abstractions and implementations and extend them independently.
    - changes in the implementation of an abstraction should have no impact on clients; that is, their code should not have to be recompiled.
    - you have a proliferation of classes. Such a class hierarchy indicates the need for splitting an object into two parts.
- **Consequences:**
    - Decoupling interface and implementation.
    - Improved extensibility.
    - Hiding implementation details from clients.

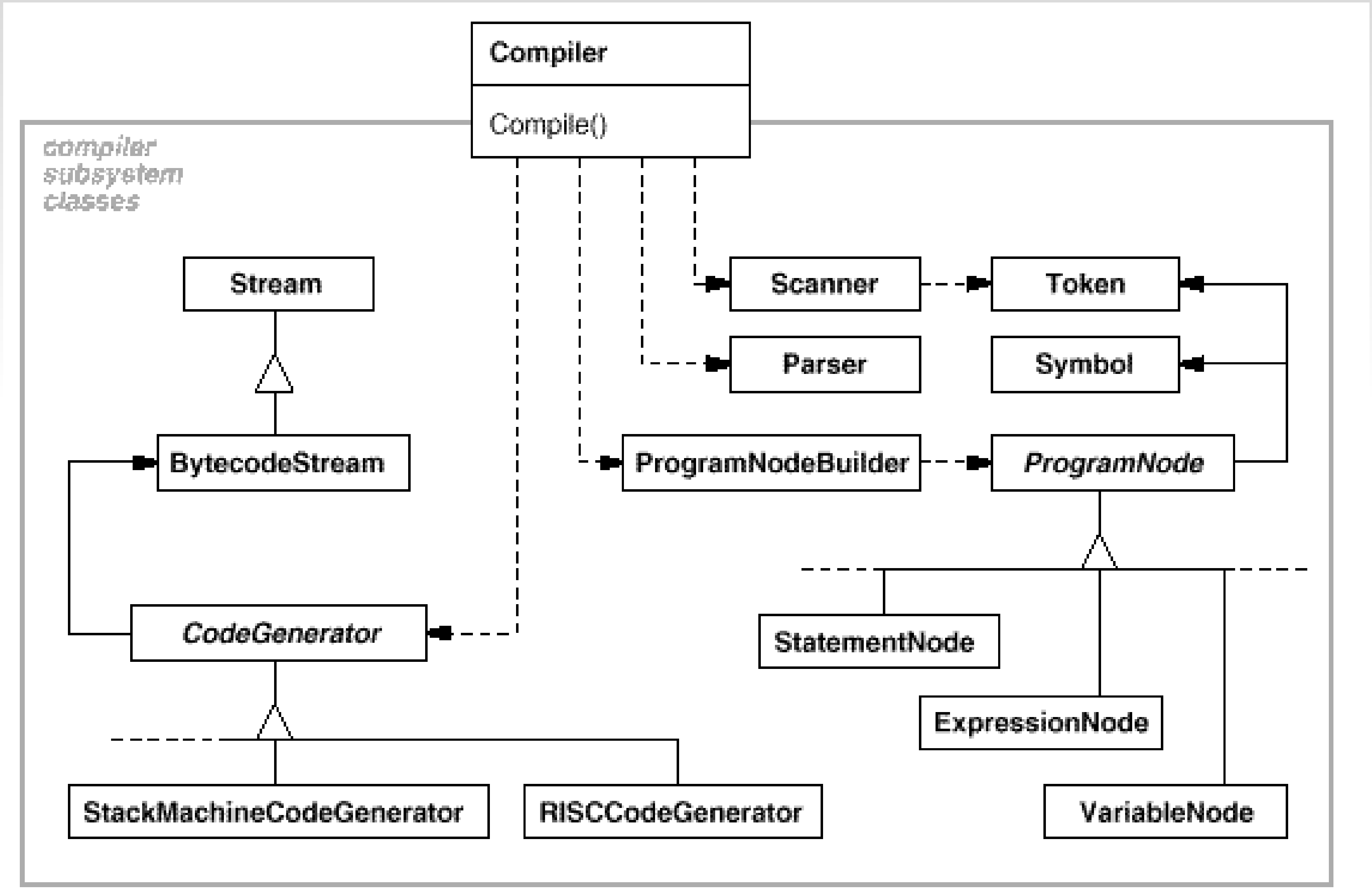
- **<u>Question:</u>** Which part of java.io is designed as Bridge?

Facade – Motivation

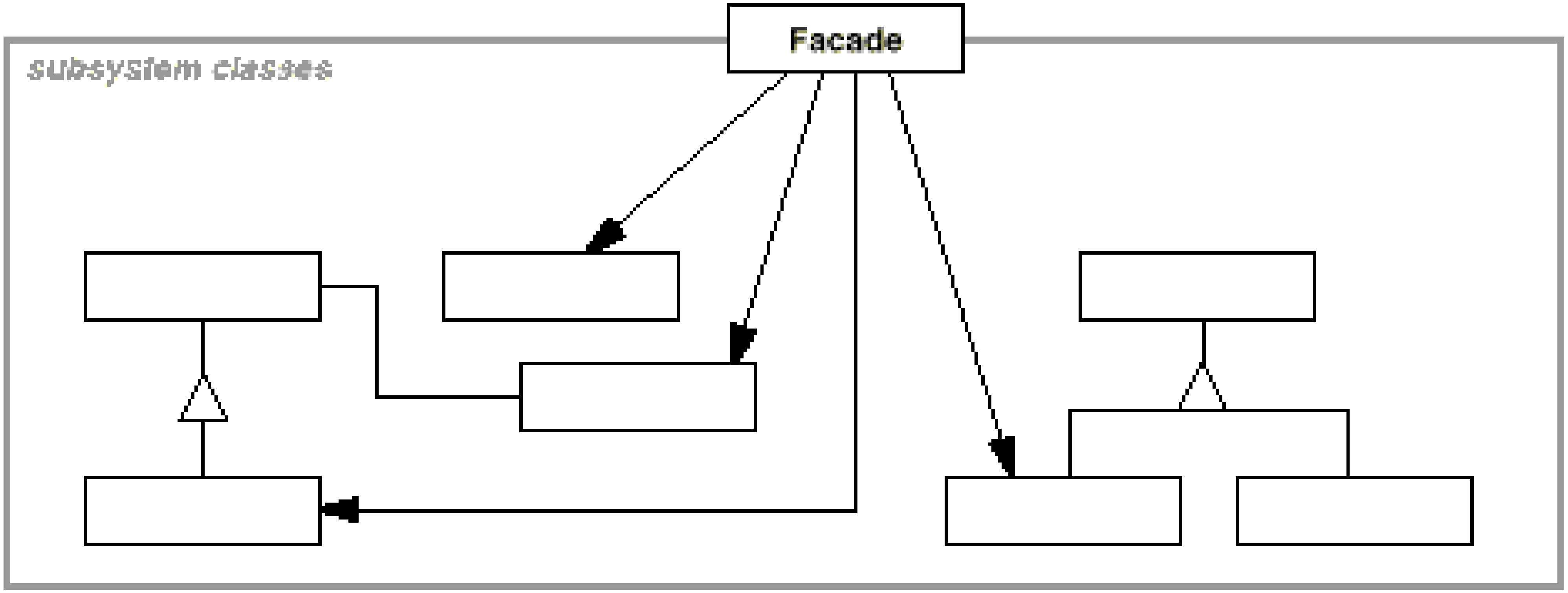




Facade – Motivation (cont.)



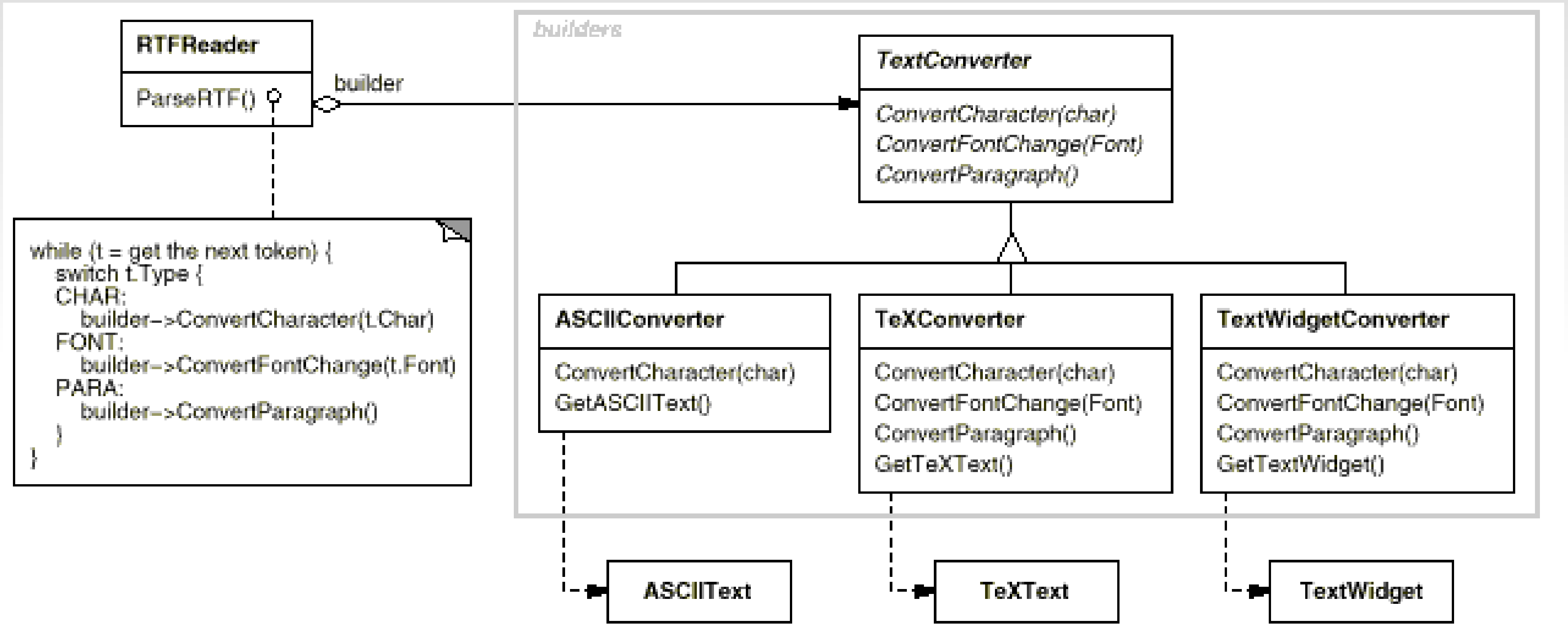
Facade Pattern



Facade Properties

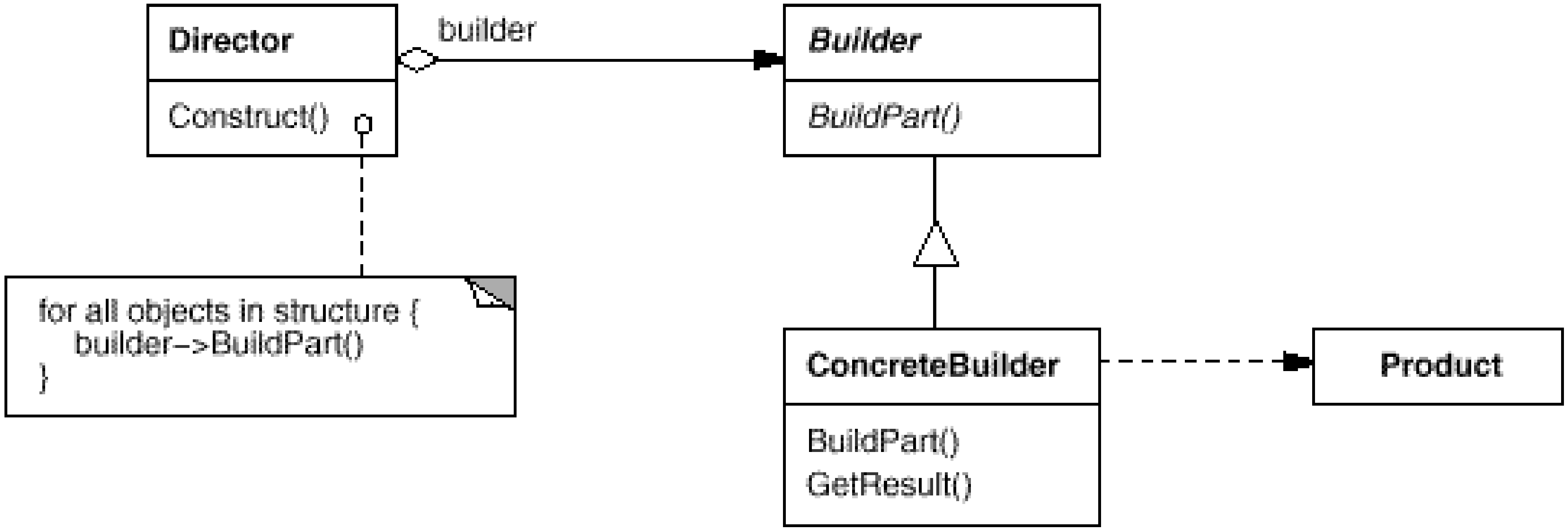
- **Applicability:** Use the Facade pattern when:
  - you want to provide a simple interface to a complex subsystem.
  - there are many dependencies between clients and the implementation classes of an abstraction. Introduce a facade to decouple the subsystem from clients and other subsystems, thereby promoting subsystem independence and portability.
  - you want to layer your subsystems. Use a facade to define an entry point to each subsystem level.
- **Consequences:**
  - It shields clients from subsystem components, thereby reducing the number of objects that clients deal with and making the subsystem easier to use.
  - It promotes weak coupling between the subsystem and its clients.
  - It doesn't prevent applications from using subsystem classes if they need to. Thus you can choose between ease of use and generality.

Builder -- Motivation

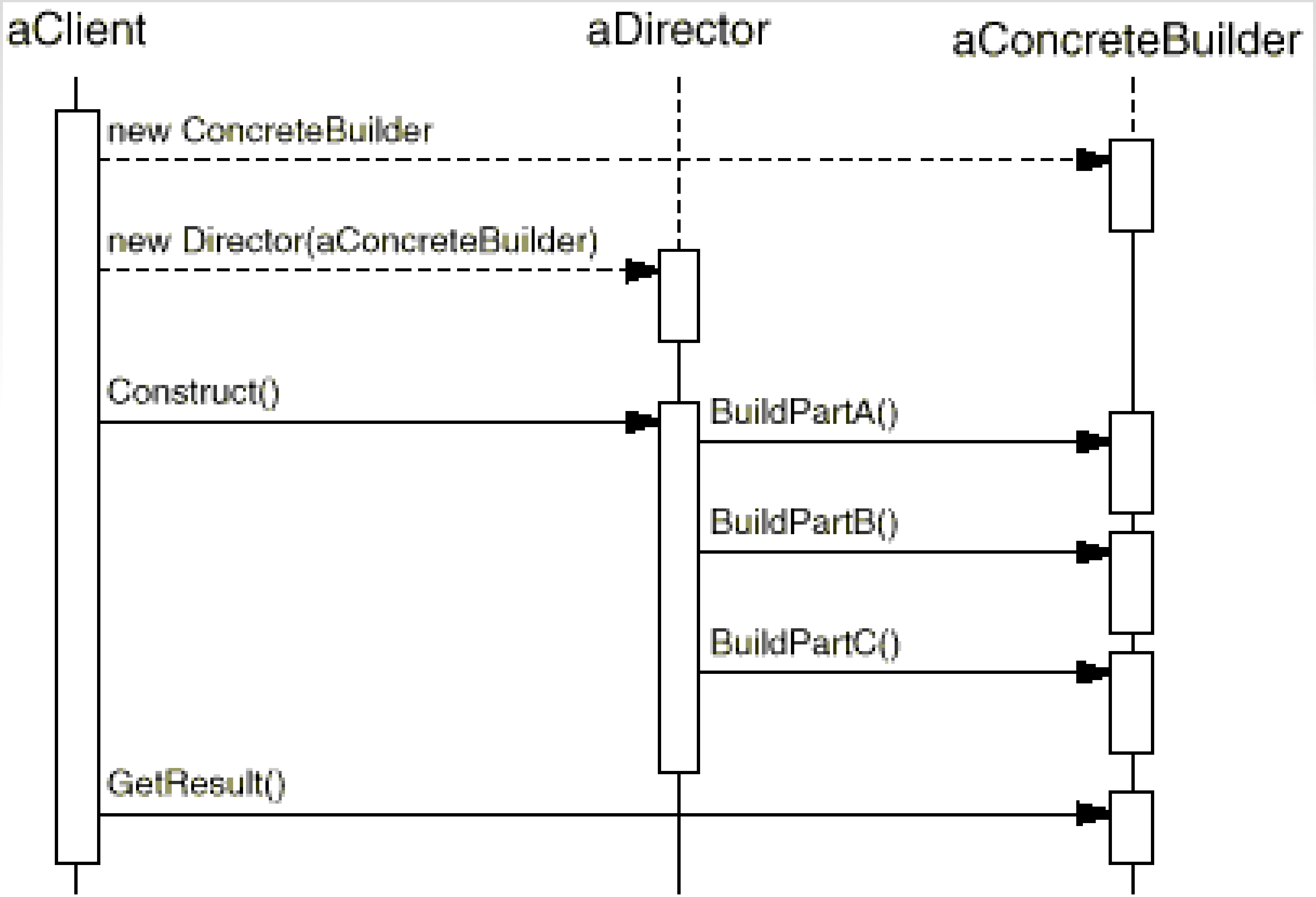


```
while (t = get the next token) {
    switch t.Type {
    CHAR:
        builder->ConvertCharacter(t.Char)
    FONT:
        builder->ConvertFontChange(t.Font)
    PARA:
        builder->ConvertParagraph()
    }
}
```

Builder Pattern



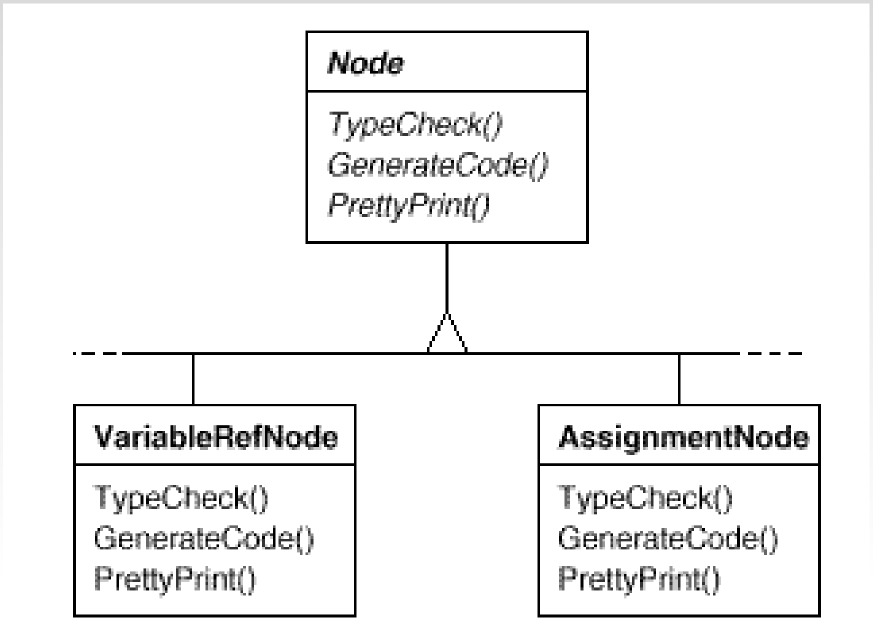
Builder – Collaborations

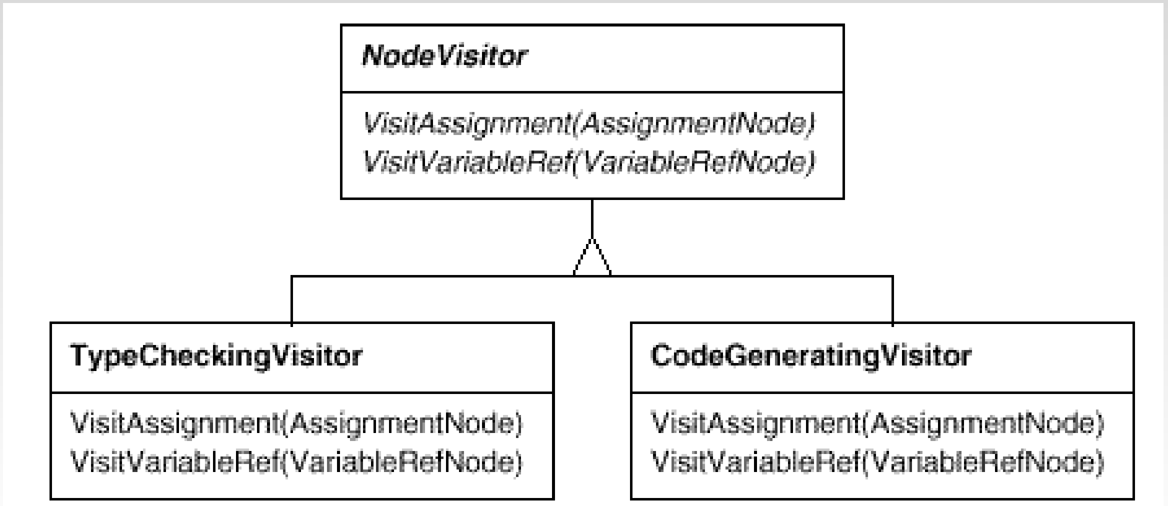


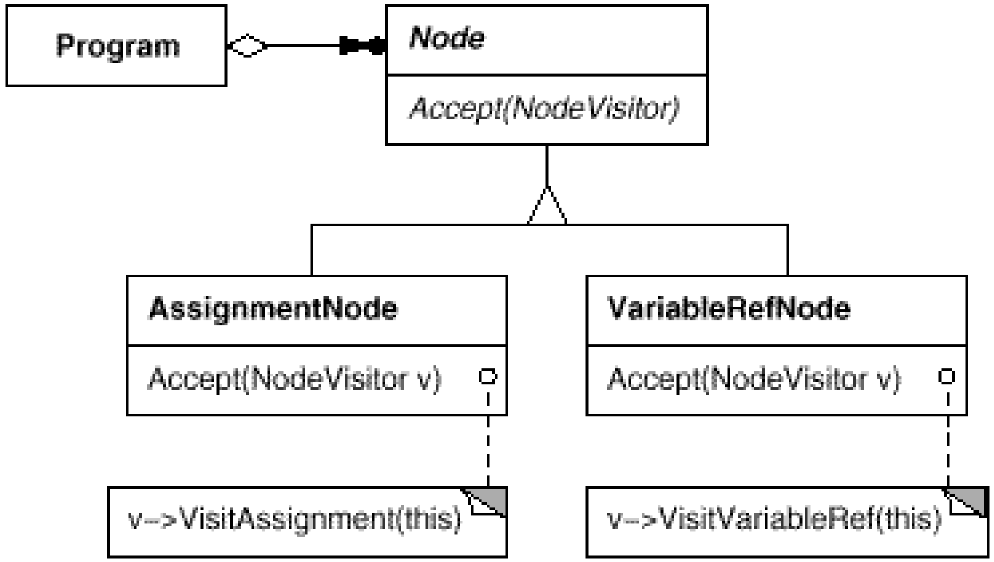
Builder Properties

- **Applicability:** Use the Builder pattern when
  - the algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.
  - the construction process must allow different representations for the object that's constructed.
- **Consequences:**
  - It lets you vary a product's internal representation.
  - It isolates code for construction and representation.
    - Each ConcreteBuilder contains all the code to create and assemble a particular kind of product. The code is written once; then different Directors can reuse it to build Product variants from the same set of parts. In the earlier RTF example, we could define a reader for a format other than RTF, say, an SGMLReader, and use the same TextConverters to generate ASCIIText, TeXText, and TextWidget renditions of SGML documents.
  - It gives you finer control over the construction process.

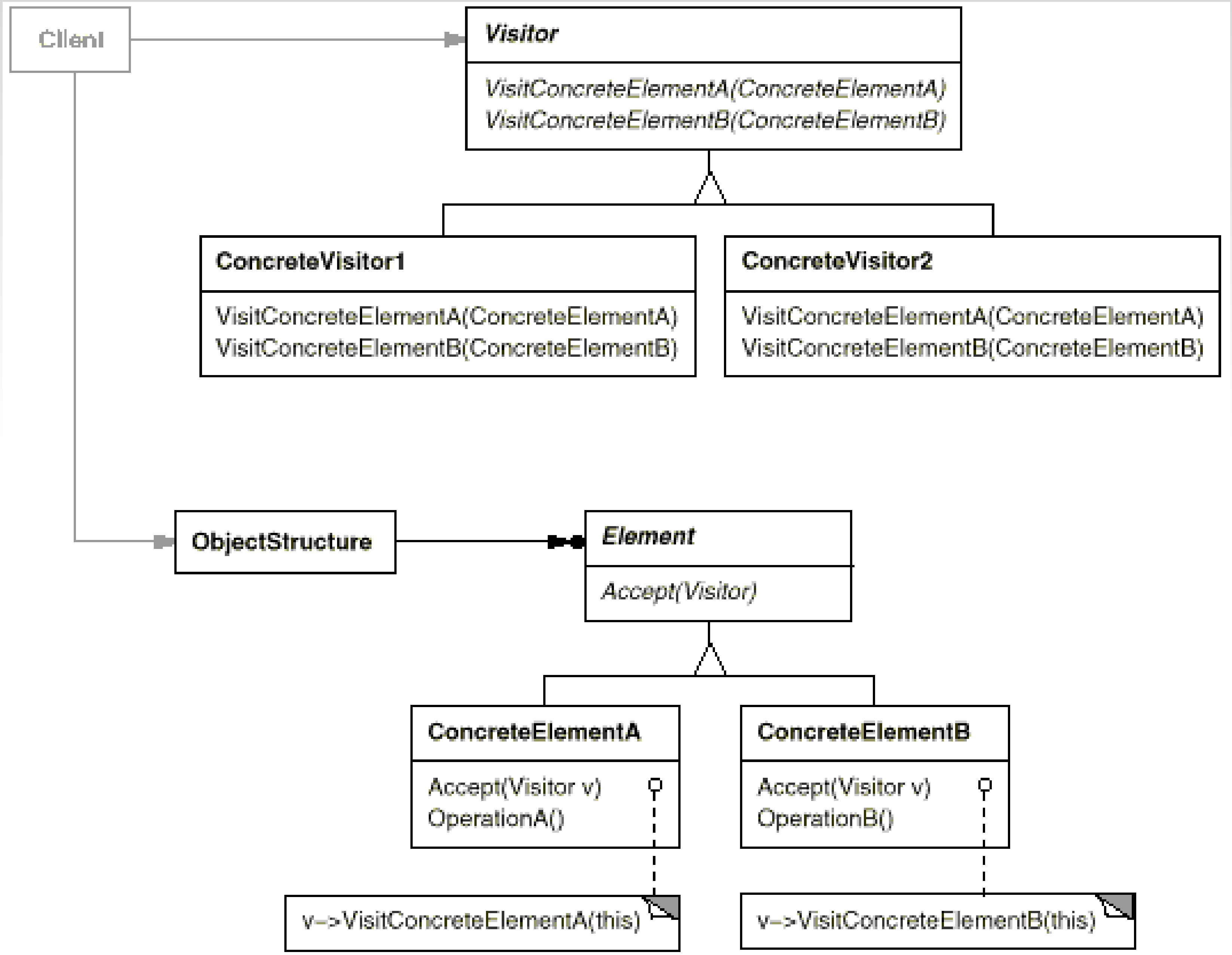
Visitor – Motivation



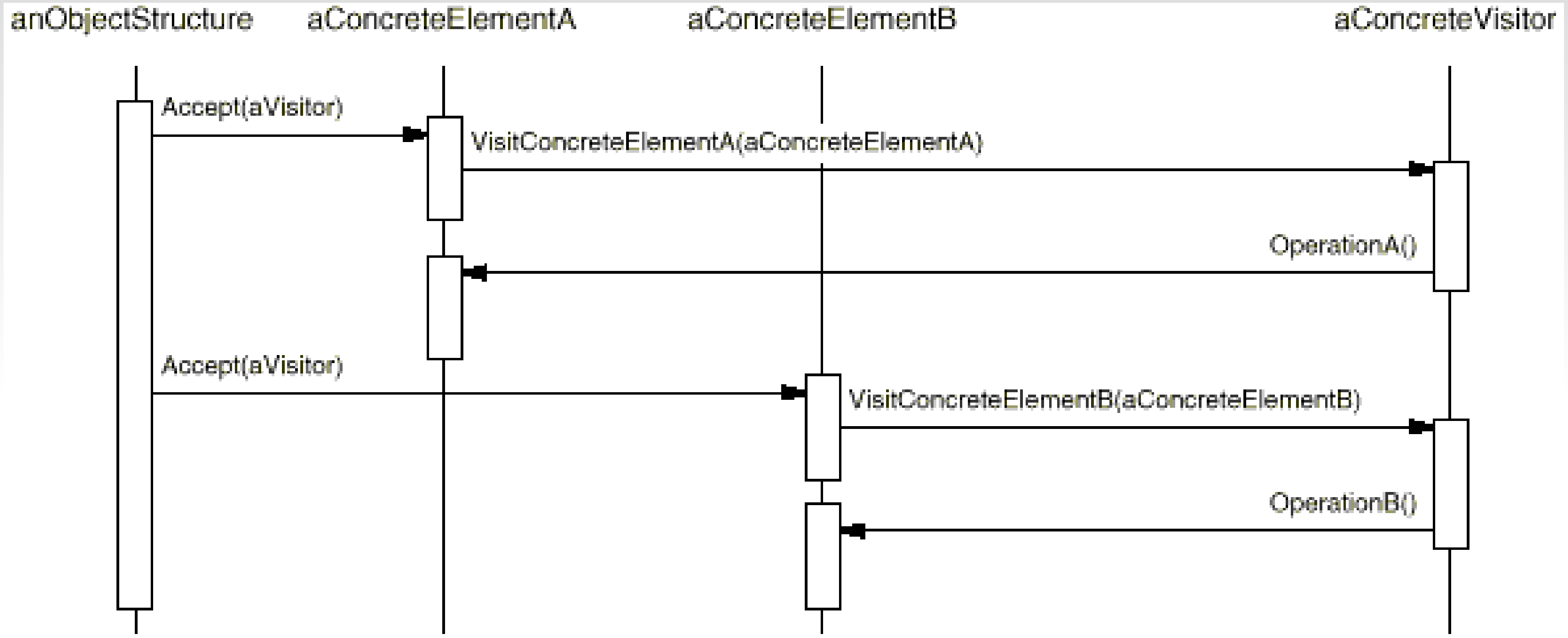

NodeVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
TypeCheckingVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)
CodeGeneratingVisitor
VisitAssignment(AssignmentNode)
VisitVariableRef(VariableRefNode)



Visitor Pattern


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

Visitor - Collaboration



Visitor Properties

- **Applicability:** Use the Visitor pattern when
  - an object structure contains many classes of objects with differing interfaces, and you want to perform operations on these objects that depend on their concrete classes.
  - many distinct and unrelated operations need to be performed on objects in an object structure, and you want to avoid "polluting" their classes with these operations.
  - the classes defining the object structure rarely change, but you often want to define new operations over the structure.
- **Consequences:**
  - Visitor makes adding new operations easy.
  - A visitor gathers related operations and separates unrelated ones.
  - Adding new ConcreteElement classes is hard.
  - Visiting across class hierarchies. Visitor can visit objects that don't have a common parent class. You can add any type of object to a Visitor interface.
  - Accumulating state.
  - Breaking encapsulation.

Questions?