Intro to
Software
“Design Patterns“

Design Patterns

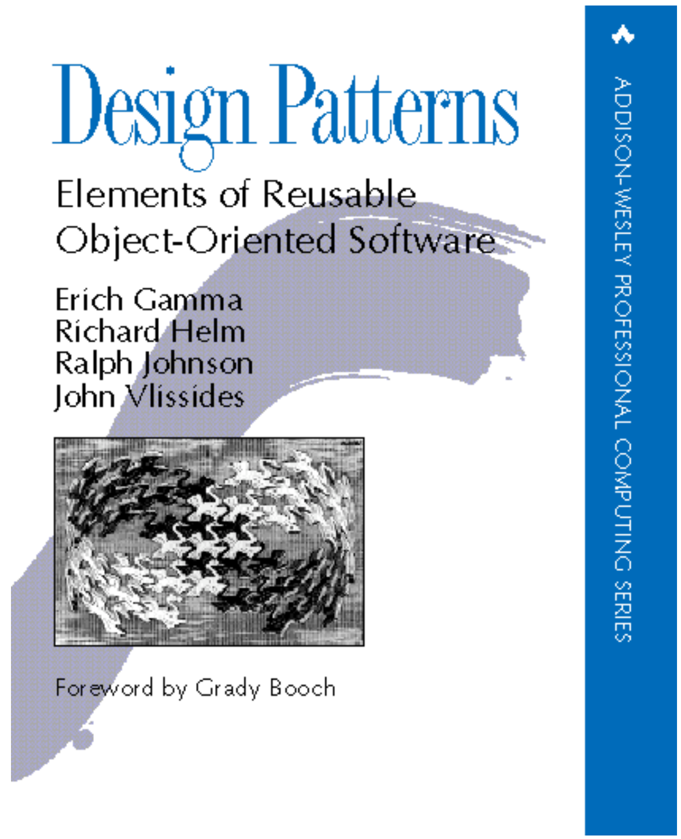
- Focus of a great deal of current attention in software development research and practice

What started the idea

Two books on *architecture* (not software)
by Christopher Alexander, et al.

and then came ...

(aka "Gang of four")



Uses earlier Booch Notation, rather than UML

Definition of a Pattern

- a solution to a recurrent problem
- not a "concrete" solution, but an abstract version of it
- four essential elements:
  - Name of the Pattern
  - The Problem
  - The Solution
  - Consequences, tradeoffs

Uses of Patterns

- Conversational “handle” on which to hang ideas and concepts
- To direct the developer to a known solution for a kind of problem
- To help focus a design
- As a vehicle for refining solution techniques

Patterns help to:

- solve specific design problems
- reduce the need for redesign
- provide reusable solutions
- act as templates
- pass on knowledge from experts to novices

slide lifted from: www.sis.port.ac.uk/%7Echandler/OOLectures/patterns/patterns.htm#Gang of Four

Patterns are not ...

- Classes
- Libraries
- Packages
- Macros
- Higher-order functions
- Template classes

However, some of these could conceivably capture some design patterns.

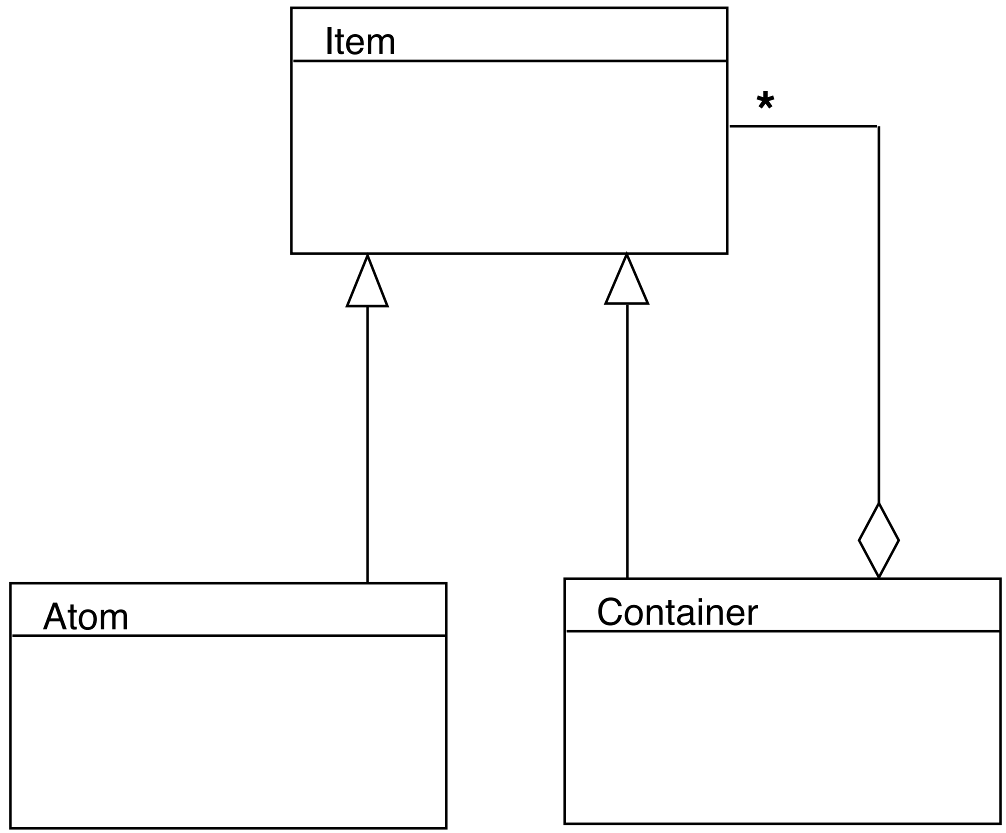
Examples of Patterns

- "Design Patterns" book lists 23 patterns, in several categories. These are sometimes annotated GoF ("Gang of Four").
- GRASPatterns in Larman book.
- Others have contributed many additional patterns.

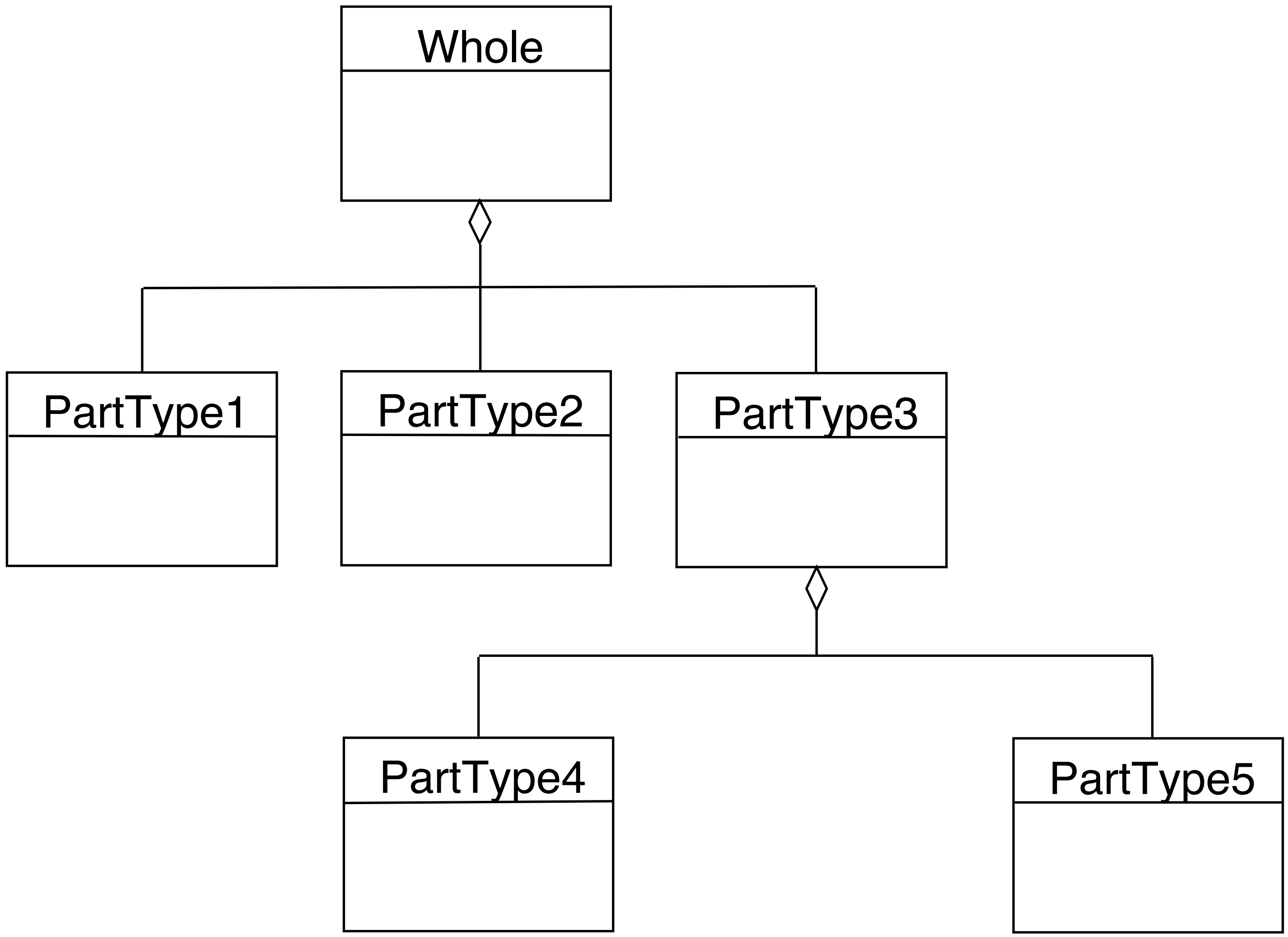
Example: Composite Pattern

- Name: Composite
- The Problem: Construct a class of objects wherein
  - Objects can be indivisible ***or*** have multiple components.
  - A collection of components can be treated as a ***single*** object by a client.
- The Solution: The one shown in the diagram
- Consequences, tradeoffs

UML for Composite

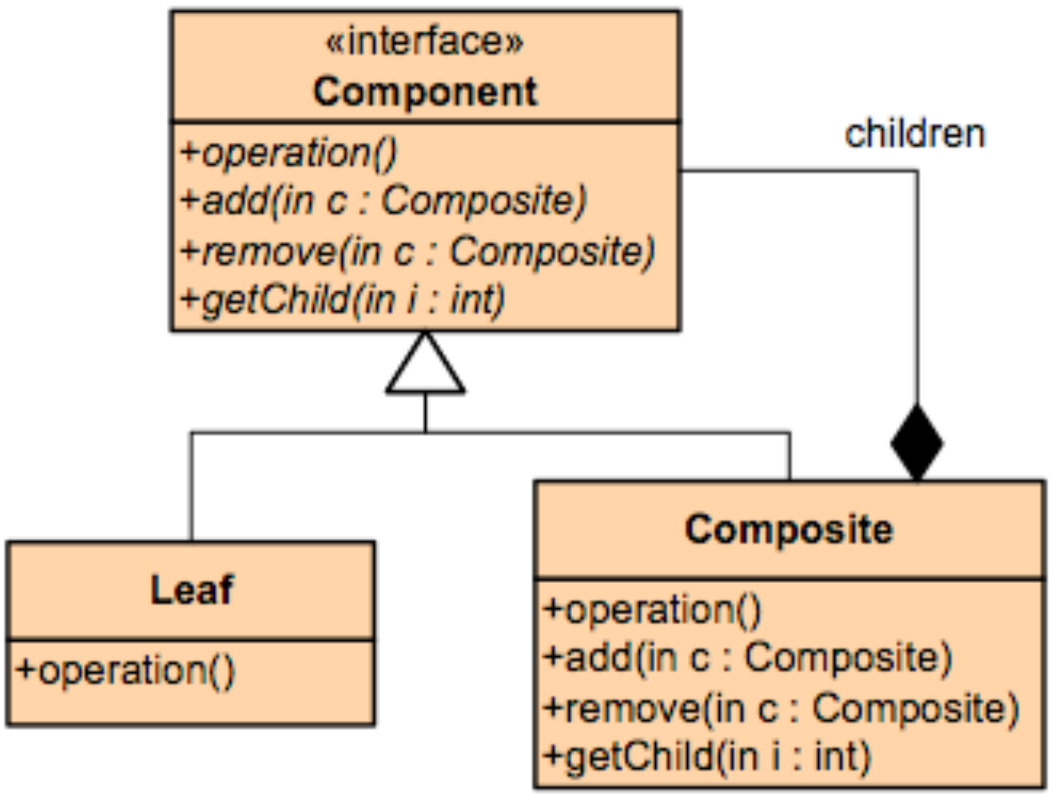


Non-Recursive Composite Pattern UML



A more detailed UML version

source: http://www.mcdonaldland.info/2007/11/28/40/



Composite

Type: Structural

What it is:
Compose objects into tree structures to represent part-whole hierarchies. Lets clients treat individual objects and compositions of objects uniformly.

Examples of Composite Patterns

- Drawing program shape: rectangle, oval, **group**: set of shapes
- File systems, files, directories, links
- S expressions

Tradeoffs in Composite Pattern

- Whether ***recursive*** structure is needed, or will "flat" structure suffice
- Whether ***ordering*** of components is significant
- Whether components refer to ***parents***
- Whether components can be **shared**
- Who should ***delete*** the *children*
- Whether children are represented as a ***list*** or simply enumerated
- Data structure issues (whether to use struct, array,linked list, etc.)

Patterns to be Discussed (alphabetized)

- Adapter
- Cache
- Command
- Composite
- Decorator
- Delegation
- Façade
- Interface
- Iterator
- Memento
- Model-View-Controller
- Observer
- Proxy
- Singleton
- State
- Stream
- Visitor

Iterator Pattern

- (GoF, p 257) aka Cursor Pattern
- provides a way to enumerate the elements in a container **without exposing the internal structure of the implementation.**
- There can be multiple Iterators on a given container.
- Examples:
  - Java: **Enumeration** and **Iterator** interfaces
  - C++: Standard Library, there are for many iterator template classes.

C++ Iterator Concepts

- Trivial Iterator
- Input Iterator
- Output Iterator
- Forward Iterator
- Bidirectional Iterator
- Random Access Iterator

See www.sgi.com/tech/stl/Iterators.html

C++ Iterator Types

- istream_iterator
- ostream_iterator

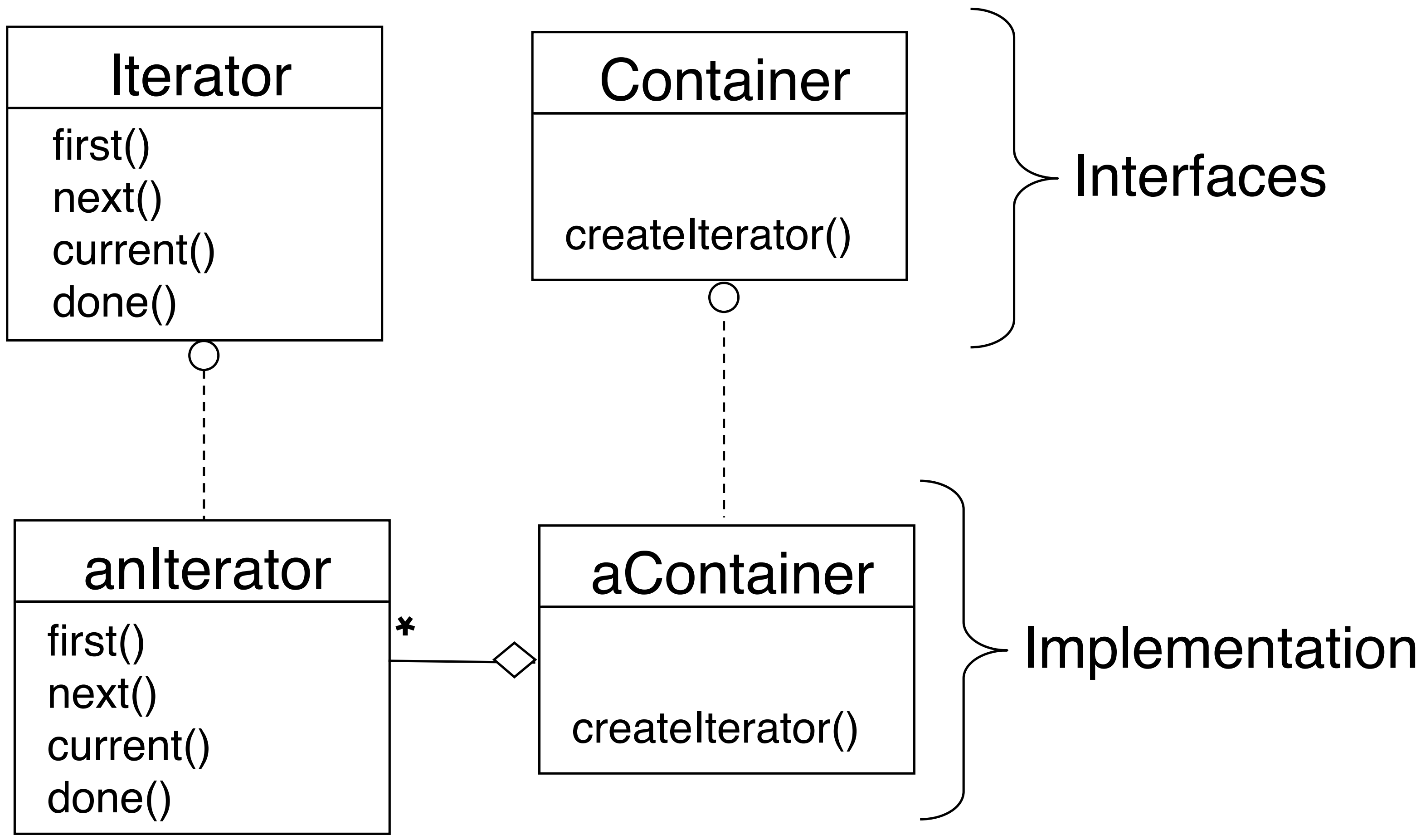
- reverse_iterator
- reverse_bidirectional_iterator
- insert_iterator
- front_insert_iterator
- back_insert_iterator

- input_iterator
- output_iterator
- forward_iterator
- bidirectional_iterator
- random_access_iterator

What Every Iterator Needs

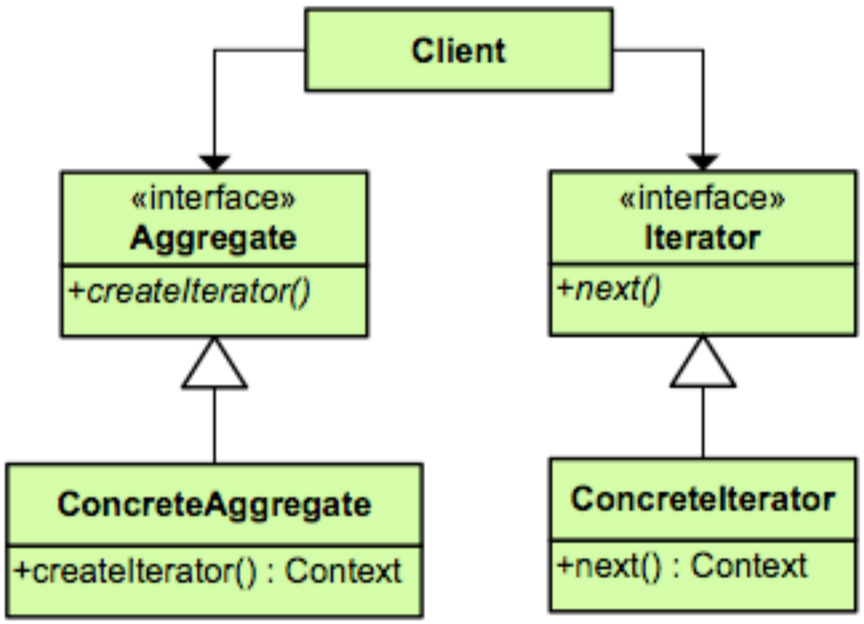
- A way to establish a specific element, such as the “first” element
- A way to access the **current** element
- A way to move on to the “next” element
- A way to indicate that there are no more elements

Iterator Pattern UML, with Interfaces



Another UML version

source: http://www.mcdonaldland.info/2007/11/28/40/



Iterator

Type: Behavioral

What it is:
Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

Iterator Patterns in Java: Enumeration Interface

- Informal Methods:
  - `creator`
  - `first()`
  - `next()`
  - `!done()`
  - `currentItem()`

- Java realization:
  - **Enumeration e =** ***container*****.elements();**
  - **(implied in initialization)**
  - **nextElement()**
  - **hasMoreElements()**
  - **none- use result of nextElement()**

Iterator Patterns in Java: **Iterator** Interface

- Informal Methods:
  - `creator`
  - `first()`
  - `next()`
  - `!done()`
  - `currentItem()`
- Java realization:
  - `Iterator e =` *`linkedList`*`.iterator();`
  - `(none)`
  - `next()`
  - `hasNext()`
  - `(none- save result of next ())`

Iterator Pattern in C++ STL

- General Methods:
  - first()
  - next()
  - done()
  - current()
- C++:
  Container<Type>::Iterator myIterator;
  - myIterator = container.begin();
  - myIterator++;
  - myIterator == container.end();
  - *myIterator
- **Syntax is that of a pointer**

Exercise

- Consider a composite that structures its elements as directed, ordered, **tree**.
- What kinds of iterators would you propose?
- What are the methods on each iterator?

Visitor Pattern

- (GoF, p 331)
- This is one of the more subtle patterns.
- Similar to the Iterator pattern, except that rather than passing objects outside during the enumeration, a Visitor object is passed **into** the Container.
- The Visitor works on the objects while *inside* the Container.
- **Purpose**: Can add **new operations** on a structure without the structure having to know details of the operation.

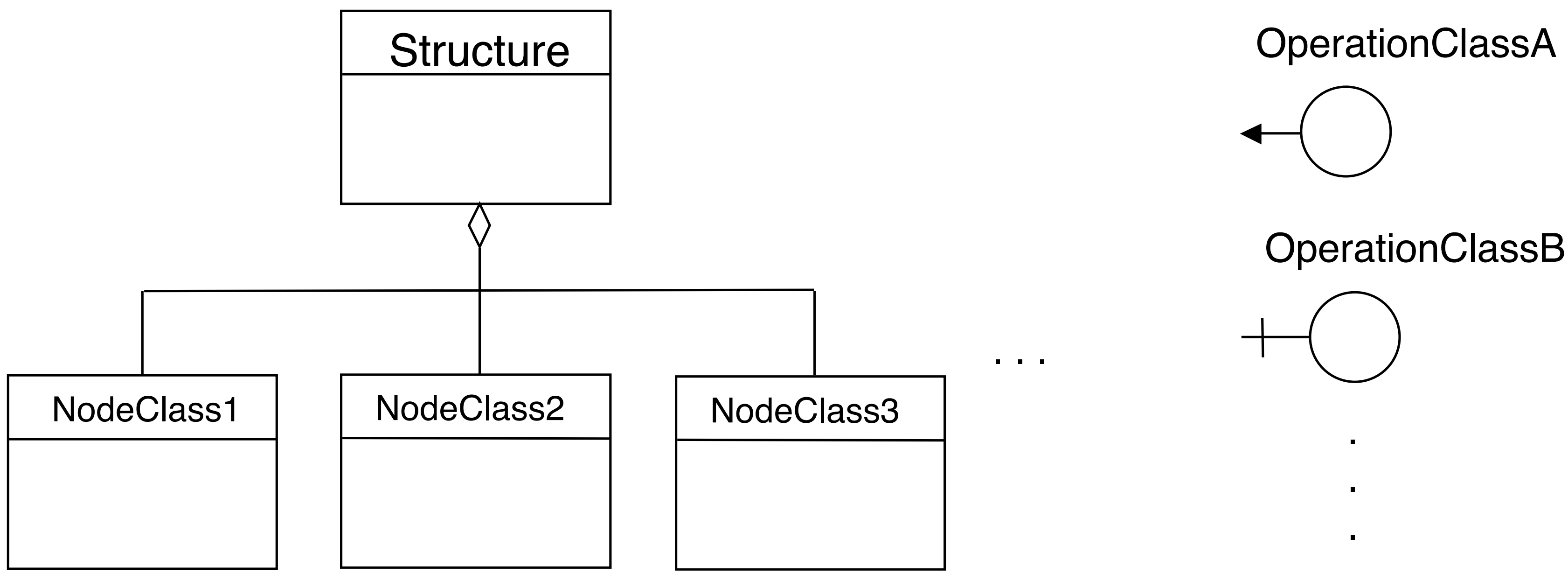
Similar to **map** in Functional Programming

- map(F, L) maps a function over a list.
- There is no explicit extraction of the list elements outside of the container.
- The function does not need to know how the list is structured; it just operates on single elements.

Visitor Details

- Suppose a structure of a given class contains sub-structures of various classes (call them "nodes").
- We plan to call a method on such structures, which will need to "visit" some or all of the nodes.
- We do *not* want to include in the definition of our structure the code for every type of operation that might be done on a node.
- We don't necessarily want to limit in advance the kinds of operations that are done with nodes.

Visitor Pattern Context



Setting Up for Visitor

- Each operation class is expected to have a method ***visit*** for each node class:
  - class OperationA

```
{
void visit(NodeClass1 n) …
void visit(NodeClass2 n) …
void visit(NodeClass3 n) …
}
```

  - class OperationB

```
{
void visit(NodeClass1 n) …
void visit(NodeClass2 n) …
void visit(NodeClass3 n) …
}
```

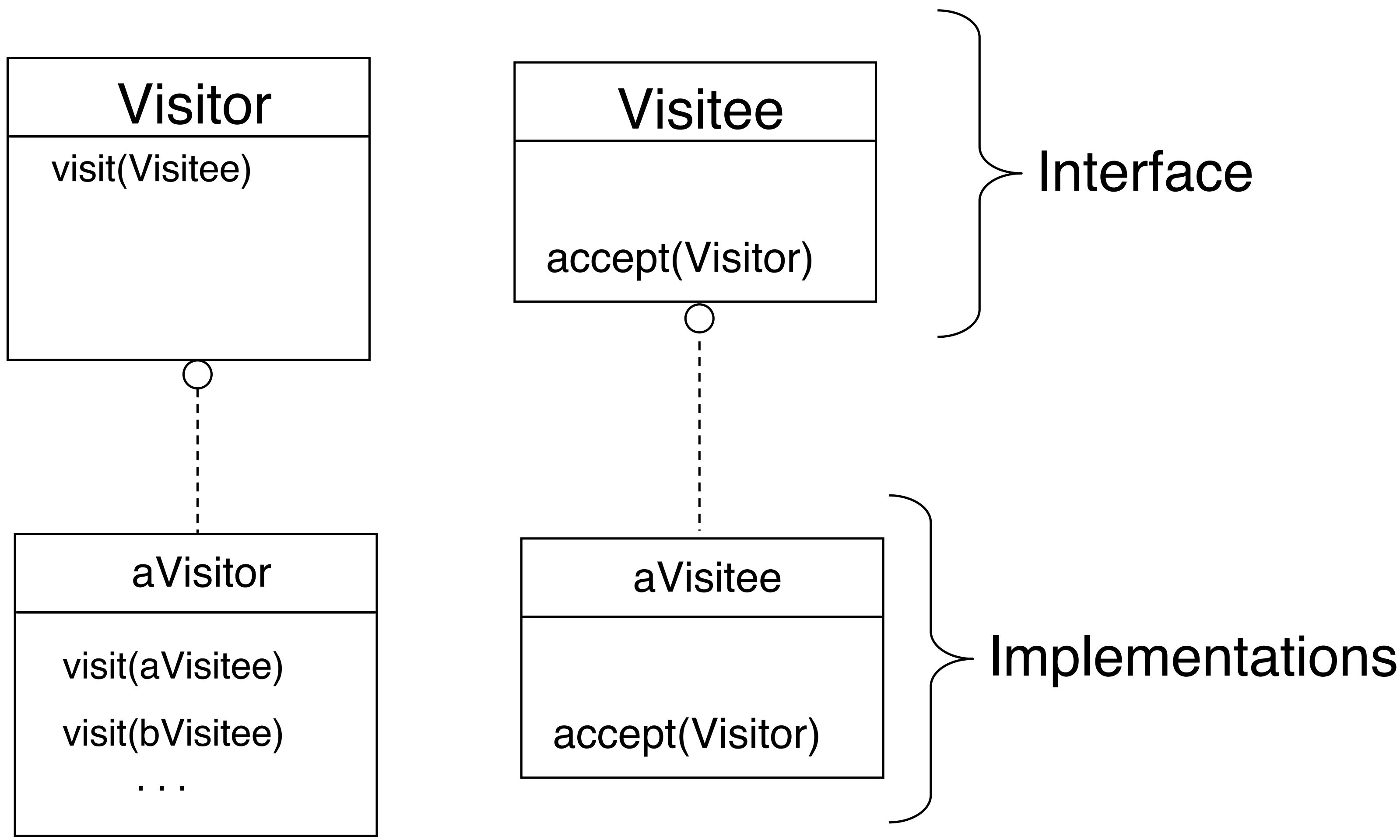
Setting Up for Visitor

- We characterize the preceding requirement by defining an interface, **Visitor**, that each operation class implements.
  - interface Visitor
    ```
    {
    void visit(NodeClass1 n);
    void visit(NodeClass2 n);
    void visit(NodeClass3 n);
    }
    ```
  - class OperationA implements Operation
    ```
    {
    void visit(NodeClass1 n) …
    void visit(NodeClass2 n) …
    void visit(NodeClass3 n) …
    }
    ```
  - class OperationB implements Operation
    etc.

Setting Up for Visitor

- Each node class provides a method ***accept*** with Visitor as argument which executes the corresponding aspect of the operation with the visitor.
  - ```
    class NodeClass1
     {
     void accept(Visitor v)
               v.visit(this);
     }
    ```
  - ```
    class NodeClass2
     {
     void accept(Visitor v)
               v.visit(this);
     }
    ```
- Generally, the meaning of visit will be distinct for each node class.
- This could be captured by having each node class implement an interface **Visitee**.

Visitor Pattern UML



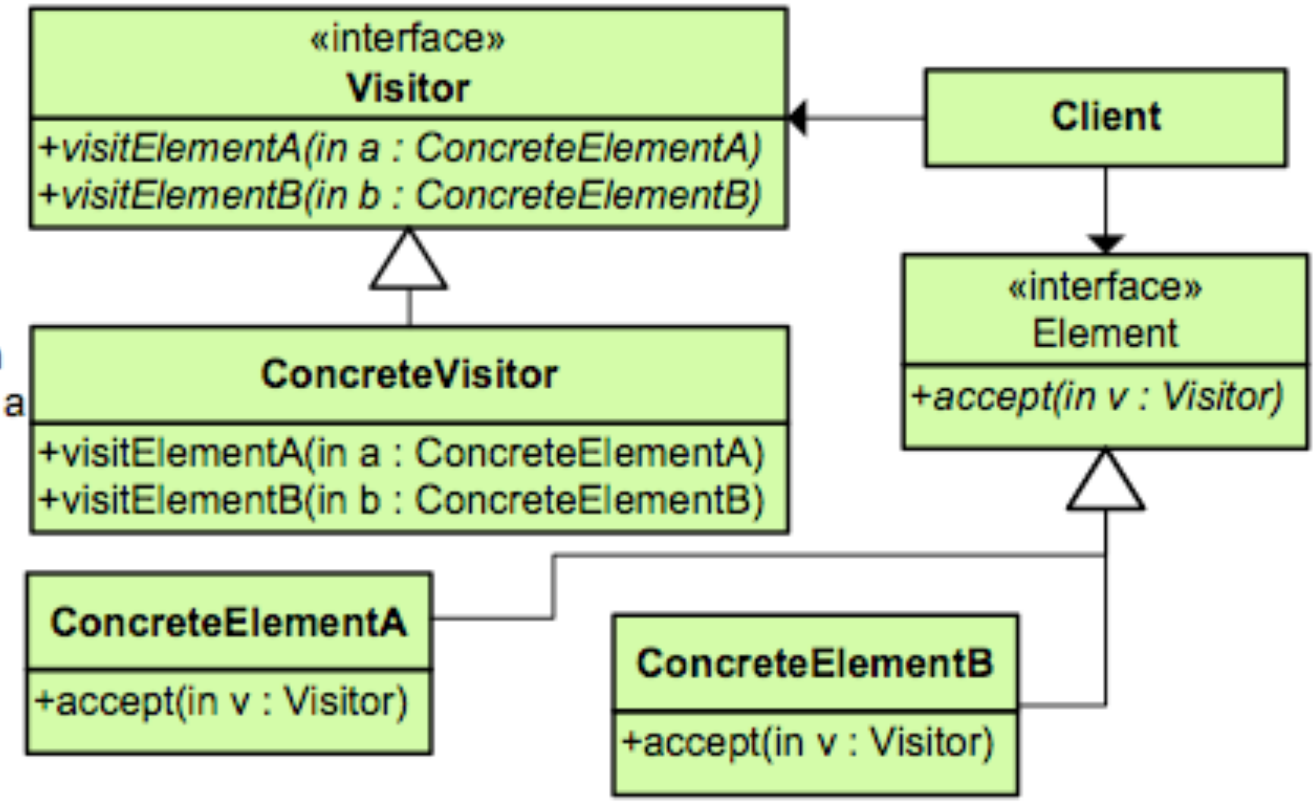
Another UML Expression

source: http://www.mcdonaldland.info/2007/11/28/40/

Visitor

Type: Behavioral

What it is:
Represent an operation to be performed on the elements of an object structure. Lets you define a new operation without changing the classes of the elements on which it operates.



Notes on Visitor

- The structure could be an instance of the Composite pattern. Thus visit() might be defined recursively.

- Visitor implements a form of "**double dispatch**": in OOP, there occasionally a need to dispatch not just on the object type by also on the type of argument to the object, and it is preferable to have a compile-time, rather than run-time, implementation. In Visitor, both the choice of Visitor method and the choice of Visitor can be determined at compile time. [Some languages, not Java or C++, support general multiple dispatch syntax.]

Visitor in C++

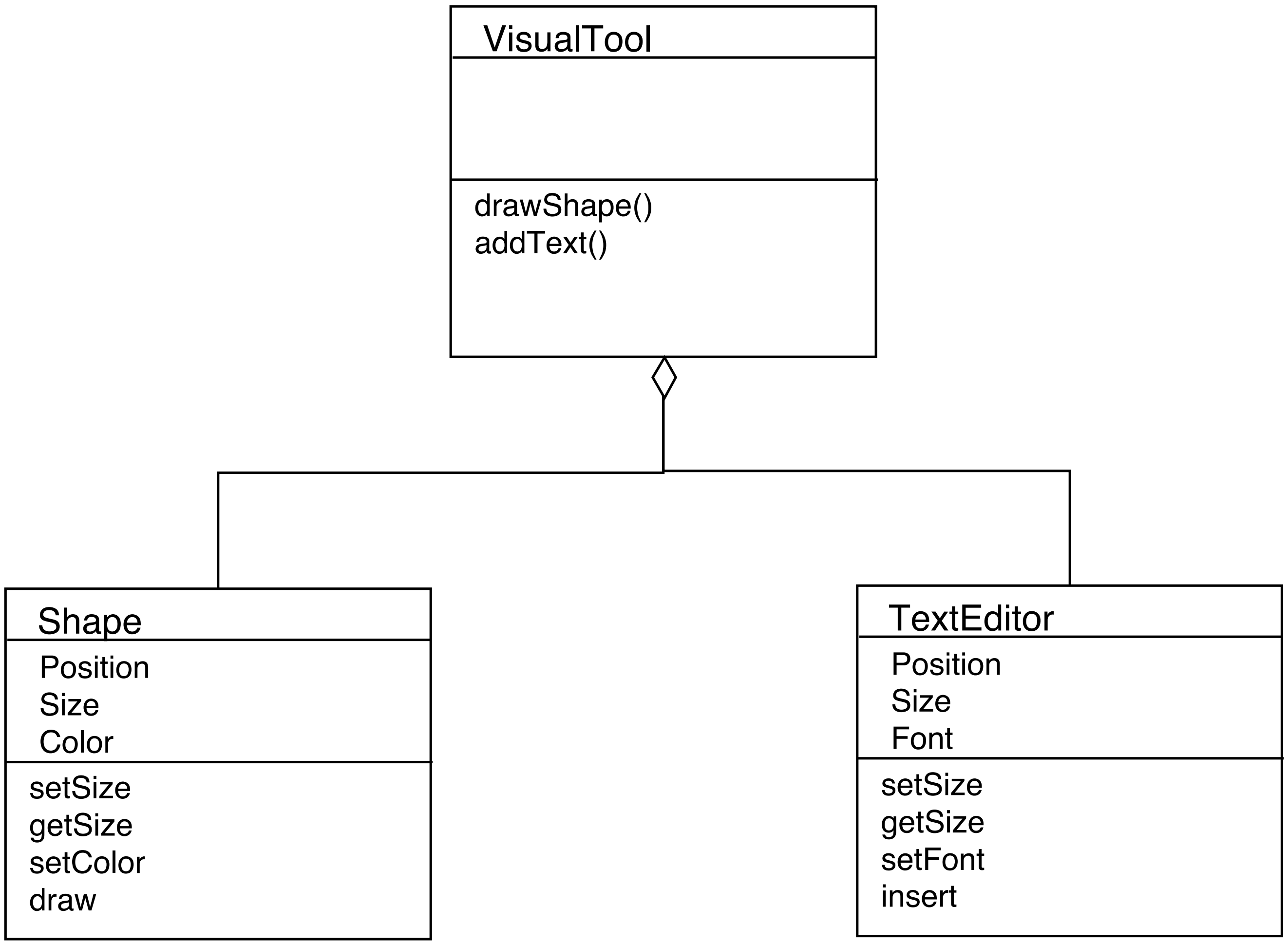
- ```
  class Visitee<VisiteeType>
  {
  accept(Visitor<VisitorType> v);
  ….
  }
  ```

- ```
  class Visitor<VisitorType>
  {
  visit(Visitee<VisiteeType> e);
  }
  ```

Façade Pattern

- (GoF, p 185)
- An entire sub-system or set of classes, etc. is given a **single simple interface** in order to
  - shield the user against the internal complexity of how the classes are used together
  - bundle together less-coupled components

Façade Pattern example



Façade Pattern

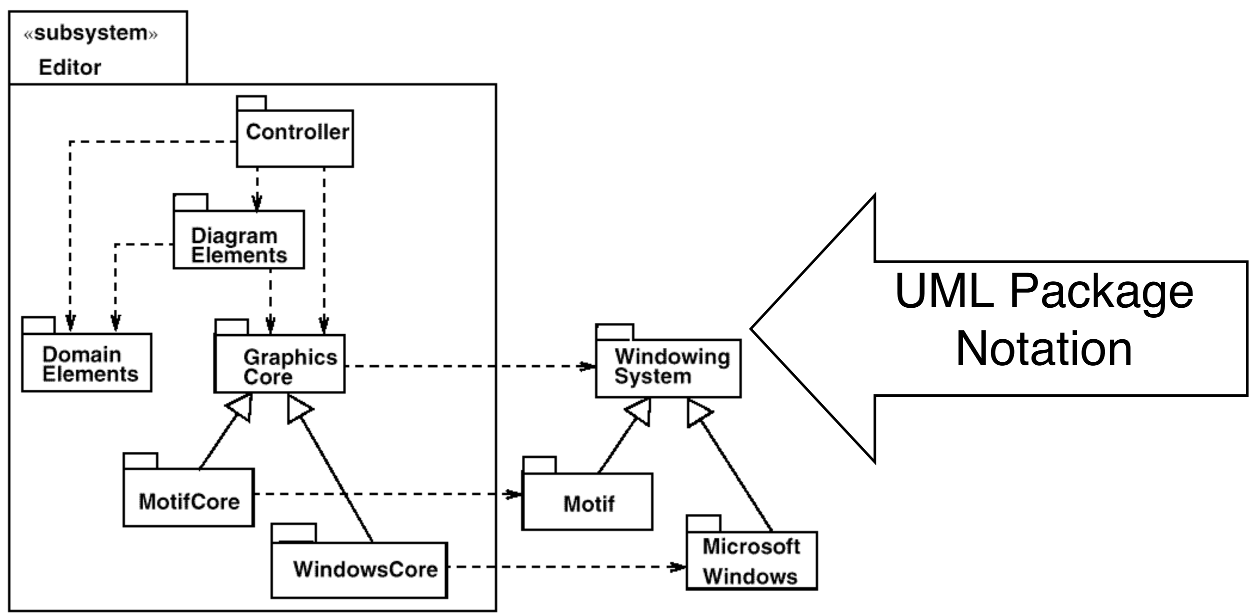
- In building using a façade pattern, it is important that the individual components not *depend* on the façade itself.
- This would introduce cross-coupling, which is undesirable.
- When the façade is “removed”, the components should “fall apart” as their original, uncoupled or loosely-coupled, entities.

Thought Question

- Does the Façade pattern violate the SRP (Martin's Single-Responsibility Principle)

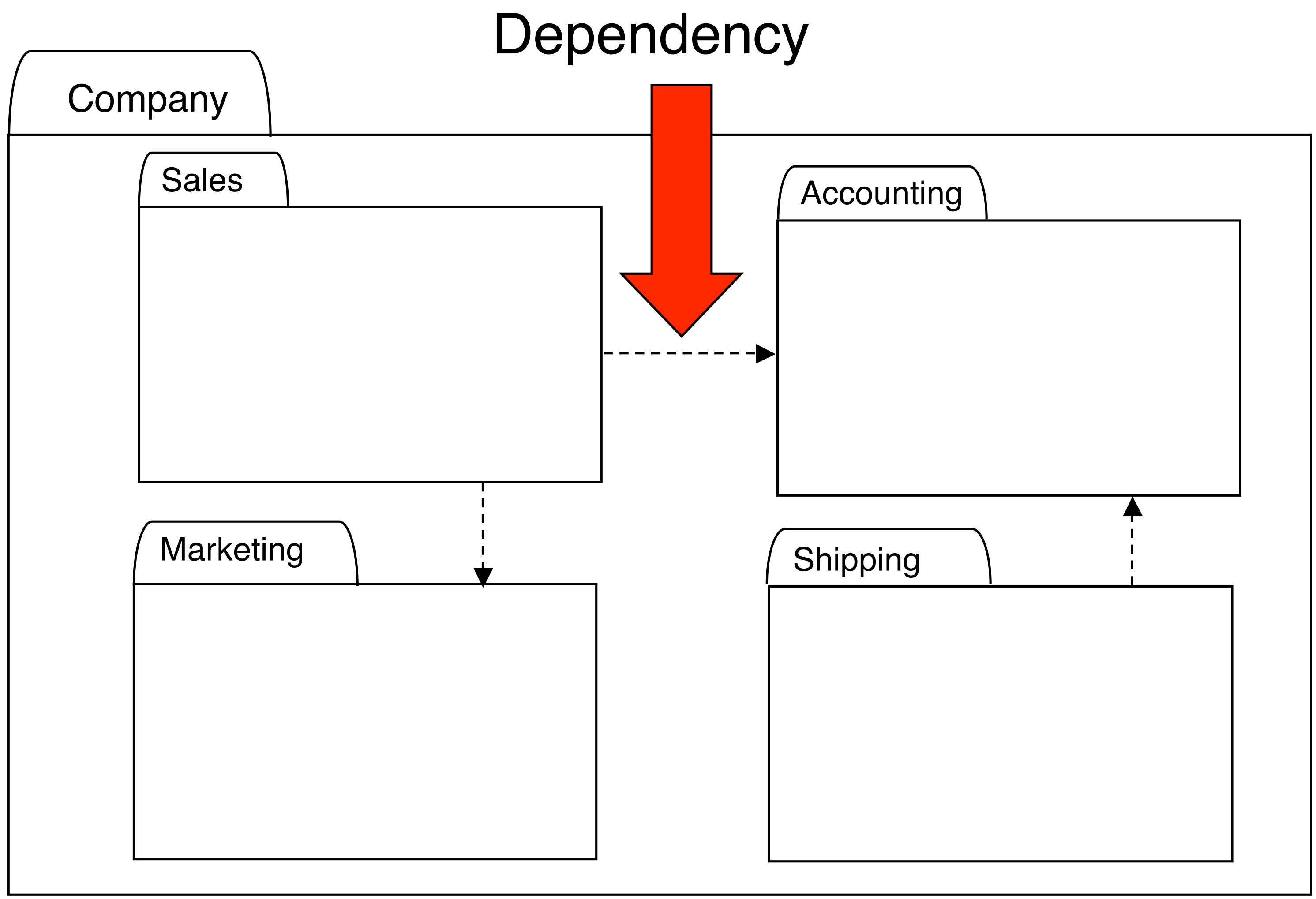
Rather than Make the Façade a Single Class, A **Package** can also be used

- A package is a group of related classes.
- Packages can have sub-packages.
- Visibility can be controlled



Brief Digression on Packages

Packages/Dependency notation in UML



Coupling / Cohesion Terminology

- Two packages (or classes, for that matter) between which there is a high-degree of inter-dependence are said to be **strongly-coupled**.
- Strong coupling is considered **undesirable**; **loosely-coupled** is better for packages and classes.

Coupling / Cohesion Terminology

- A set of methods for a class, or classes in a package, are said to be **cohesive** (or "coherent") if they provide aspects of a uniform model for dealing with objects.
- Cohesiveness is desirable; it is the mark of a careful design.

Coupling / Cohesion Summary

- Coupling is “bad“.
- Cohesion (coherence) is “good“.

- See also: GRASP (Larman):
  - Low-coupling pattern
  - High-cohesion pattern

About Packages

- **Innermost** packages contain classes
- **Cyclic dependencies** among packages should be avoided: break them, or combine into a single package.
- Packages can help delineate work-breakdown among teams, and can thus serve as a **management device**
- Consider **Java *package* construct**.
- Equivalent in C++ is **namespaces**

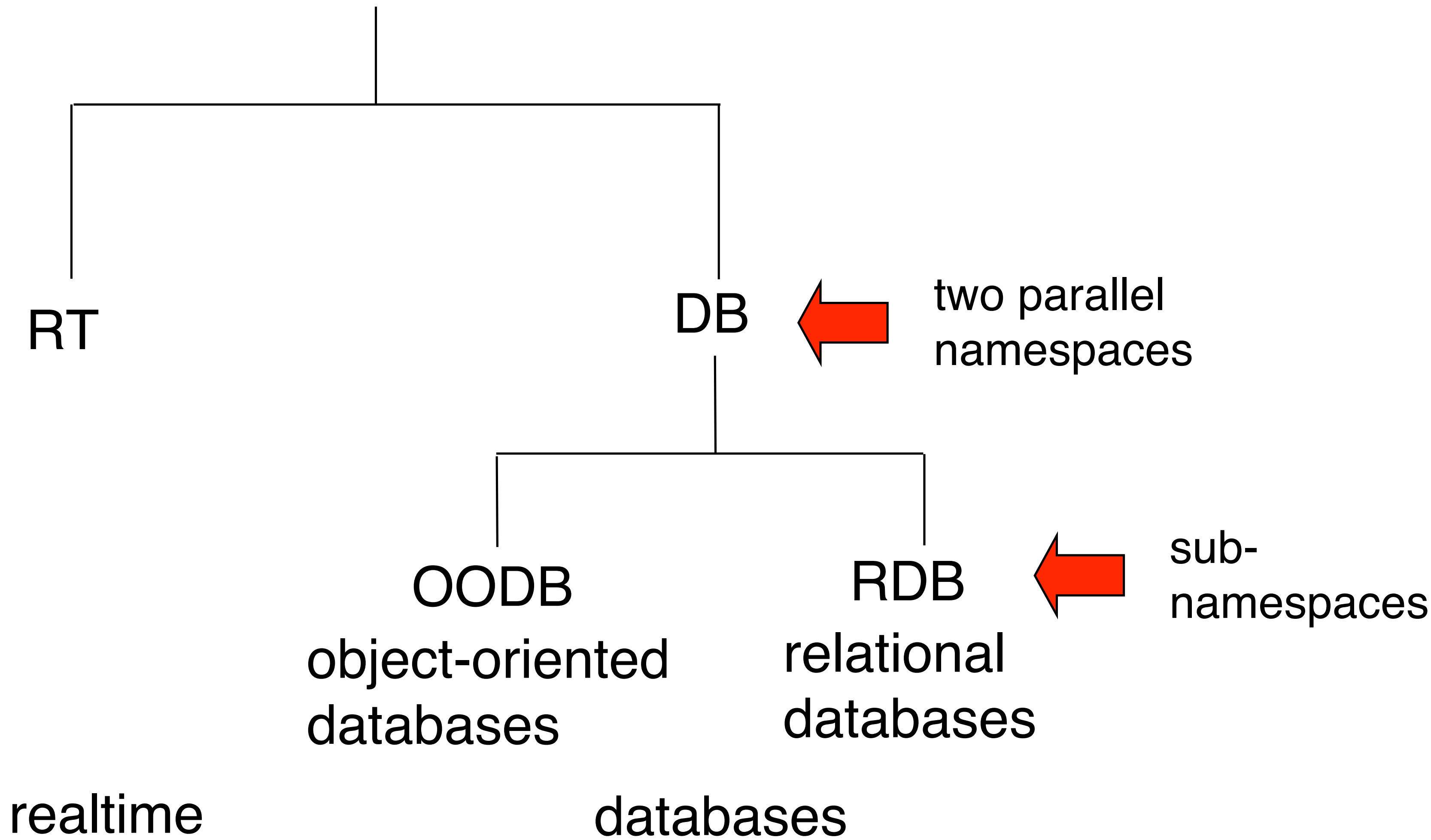
A Portion of a Java Package Hierarchy

```
java.net
java.rmi
     java.rmi.activation
     java.rmi.dgc
     java.rmi.registry
java.rmi.server
java.security
     java.security.acl
     java.security.cert
     java.security.interfaces
     java.security.spec
```

Namespaces in C++

- Without Namespaces, there is one big flat name space.
- The danger is conflicting names in different modules and name space "pollution" by modules with lots of names.
- Namespace construct allows structuring into an arbitrary number of hierarchical levels.

C++ Namespace Example



C++ Namespace Example

```
namespace RT
  {
  class A
    {
    };
  }
```

qualified names

RT::A

DB::OODB::A

```
namespace DB
  {
  namespace OODB
    {
    class A
      {
      };
    }
```

non-conflicting

DB::RDB::A

```
  namespace RDB
    {
    class A
      {
      };
    }
  }
```

Short-cut rules can lead to confusion if used.

Returning to Patterns

Decorator Pattern

- (GoF, p 175)
- aka Wrapper (one version)

- Enclose an object of one class in another class that "decorates" the original objects (e.g. scrollbars or a border around a window).

- Examples: streams of various types
  (OutputStream, **File**OutputStream, **Print**Stream, ...)

Decorator Class Structure

- At least three possibilities:
  - The decorating class **inherits** from the decorated class (“direct” decoration).
  - The decorating class **aggregates** or **composes** a member of the decorated class (“decoration by **delegation**”).
  - A third class aggregates or composes both the decorated class and the decorations (sometimes called “mix-ins”, or “traits”, which are not the same).

Adapter Pattern

- (GoF, p 139)
- aka Binding, Wrapper (second version)

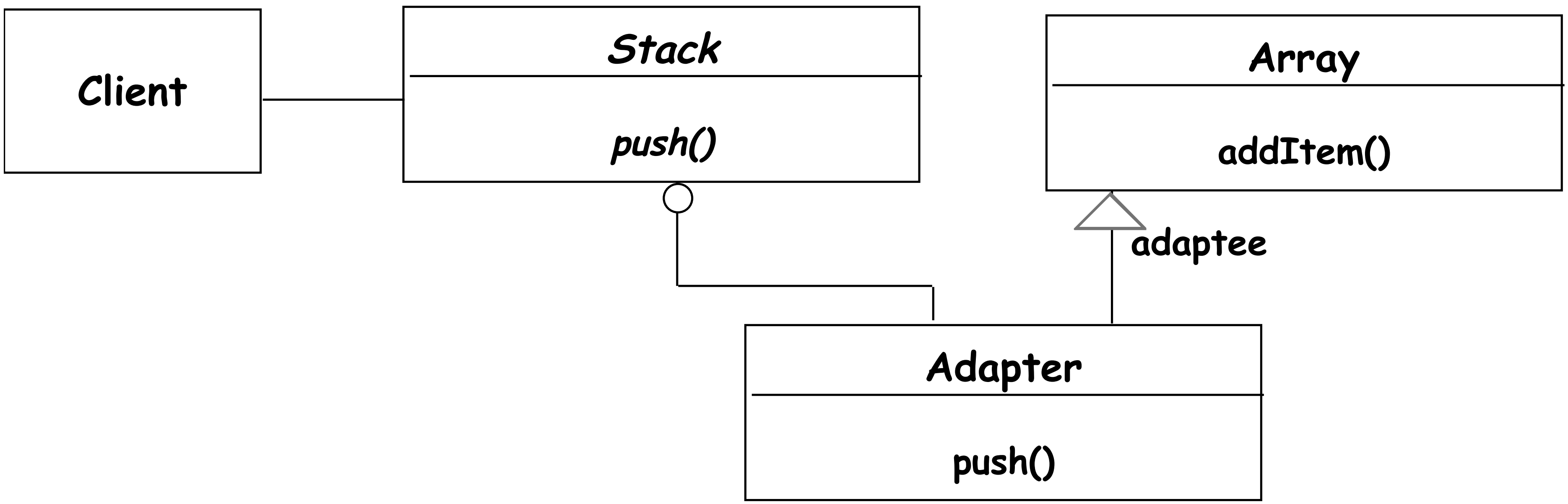
- Adapts one or more existing APIs to fit another API specification (one that clients expect).

- (API = Application Programming Interface)

Adapter Example

- An interface defines a **Stack**.
- A dynamic array implementation defines an **Array**.
- An adapter is an **ArrayStack**, i.e. a class satisfying the stack interface, implemented using an array.
- The roles of the Stack and Array classes are not symmetric.

Adapter Pattern based on Association



In C++, multiple inheritance would be used.

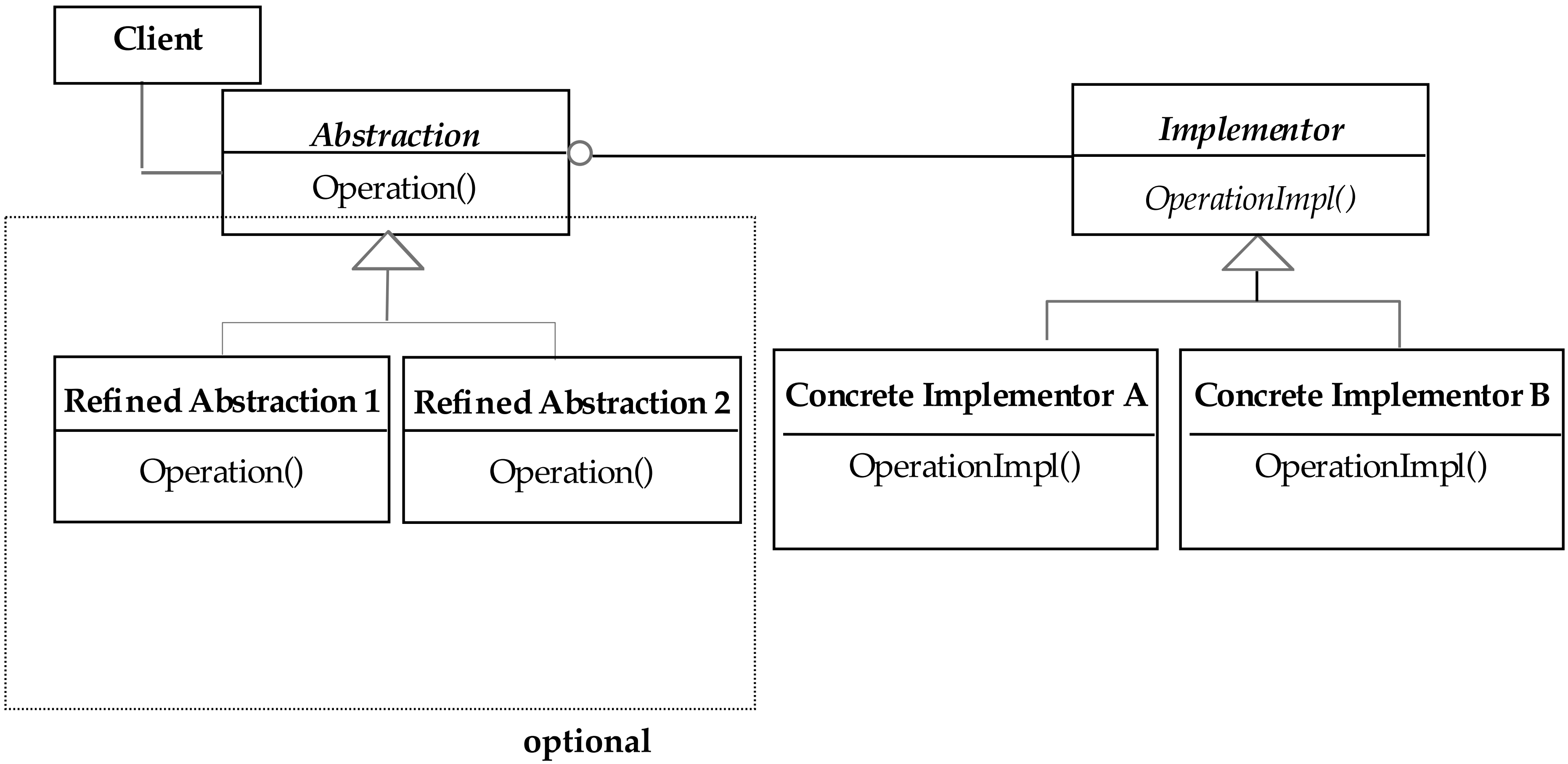
Bridge Pattern

- (GoF, p 151)
- aka "Handle/Body" or "Interface" pattern

- Abstract an API by providing an interface class, with the **intention by design** of providing ***different implementations*** for the interface.

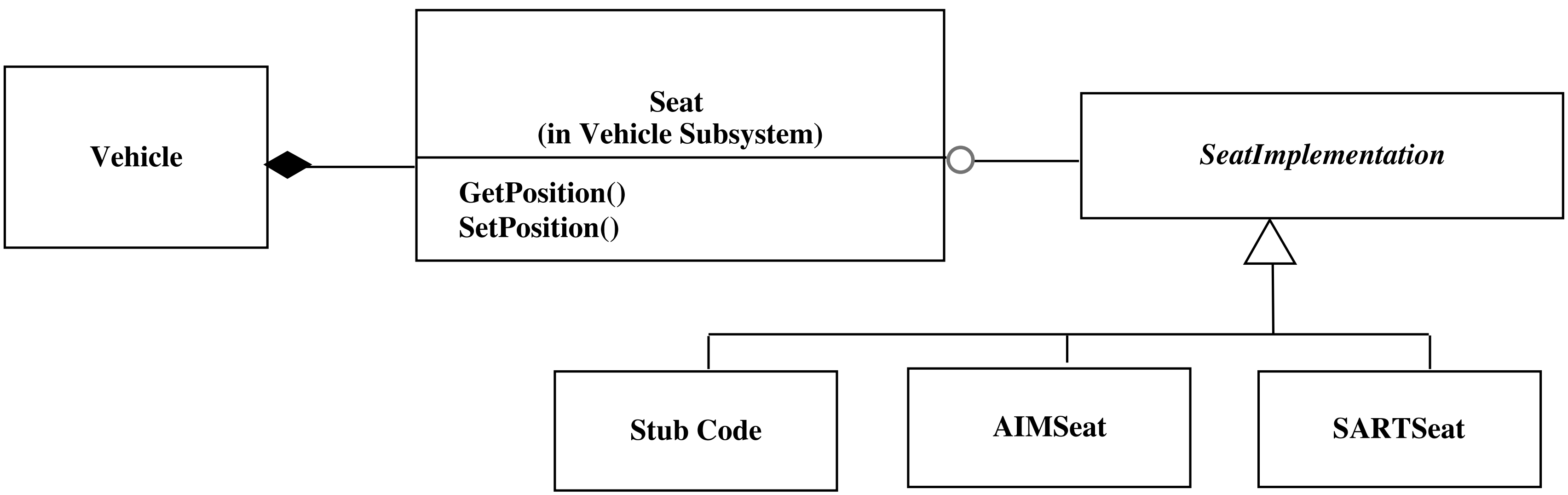
- Example: Java awt (abstract window toolkit) vs. M/S Windows or MacOS Windows (called "peer" classes)

Bridge Pattern UML



Using a Bridge

- Example: Interface to a component that is incomplete, not yet known or unavailable during testing
- JAMES Project (CMU): if seat (for vehicle) data is required to be read, but the seat is not yet implemented, not yet known or only available by a simulation, provide a bridge:



JAMES Bridge Example in Java

```
public interface SeatImplementation {
  public int GetPosition();
  public void SetPosition(int newPosition);
}
public class AimSeat implements SeatImplementation {
  public int GetPosition() {
    // actual call to the AIM simulation system
  }
  ...
}
public class SARTSeat implements SeatImplementation {
  public int GetPosition() {
    // actual call to the SART seat simulator
 }
  ...
}
```

Adapter vs. Bridge

- Adapter and Bridge are similar:
  - **Adapter**: Adapts **existing** classes to an expected interface. The interface and classes exist; the new thing is the adapter, a go-between class.
  - **Bridge**: Creates an abstract Interface **to be** implemented by multiple classes; keeps the implementations separate from the interface. The new thing is the interface.

Adapter vs. Bridge

- The **adapter** pattern is geared towards making unrelated components work together
  - Applied to systems after they're designed (**reengineering**, interface engineering).
- A **bridge**, on the other hand, is used in a design to let abstractions and implementations vary independently.
  - **Ab initio engineering** of an "extensible system"
  - New "beasts" can be added to the "object zoo", even if these are not known at analysis or system design time.

C++ Bridge Pattern

- ```
  class Stack<Item>
  {
  virtual void push(Item item) = 0;
  virtual Item pop() = 0;
  }

  class ArrayStack<Item> : public Stack<Item>
  {
  Item array[ ];
  }
  ```

Interface definition

An implementation of the interface
  ```

C++ Bridge Pattern (cont'd)

Another implementation
of the interface

- ```
  class ListStack<Item> : public Stack<Item>
  {
  list<item> List;
  }
  ```
  ```

Singleton Pattern

- (GoF, p 127)
- A class with exactly one instance (or a class, used only for its static members).
- Implements a **global access point**.
- Still use a class
  - Globals are still not a good idea
  - May wish to use multiple instances in the future.