Cornell University
Computing and Information Science

CS 5150 Software Engineering

Design Patterns

William Y. Arms

Design Patterns

Sources:

E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994

The following discussion of design patterns is based on Gamma, et al., 1994, and Bruegge and Dutoit, 2004.

Wikipedia has good discussion of many design patterns, using UML and other notation, with code samples.

Design Pattern

Design patterns are template designs that can be used in a variety of systems. They are particularly appropriate in situations where classes are likely to be reused in a system that evolves over time.

- Name

  [Some of the names used by Gamma, et al. have become standard software terminology.]

- Problem description

  Describes when the pattern might be used, often in terms of *modifiability* and *extensibility*.

- Solution

  Expressed in terms of classes and interfaces.

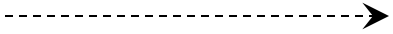
- Consequences

  Trade-offs and alternatives.

Notation

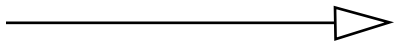
ClassName

class name in italic indicates an abstract class



dependency

delegation



inheritance

whole/part association

Abstract Classes

Abstract class

Abstract classes are superclasses which contain abstract methods and are defined such that concrete subclasses extend them by implementing the methods. Before a class derived from an abstract class can become concrete, i.e. a class that can be instantiated, it must implement particular methods for all the abstract methods of its parent classes.

The incomplete features of an abstract class are shared by a group of subclasses which add different variations of the missing pieces.

Wikipedia 4/2/08

Adapter (Wrapper): Wrapping Around Legacy Code

Problem description:

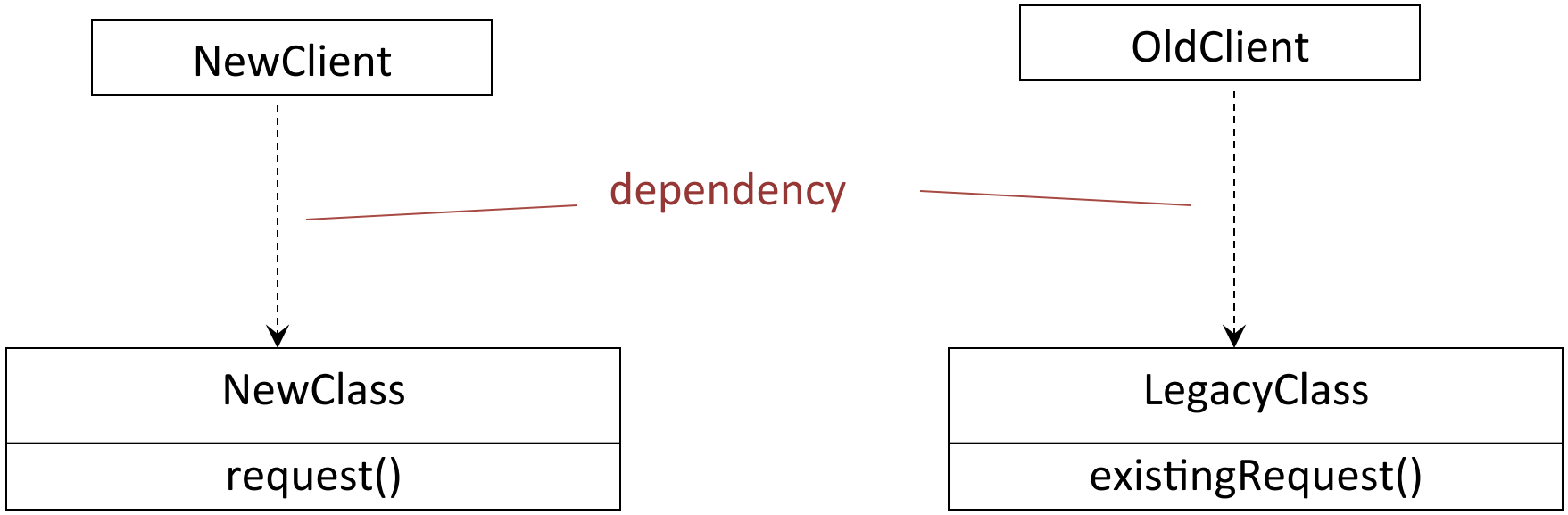
Convert the interface of a legacy class into a different interface expected by the client, so that the client and the legacy class can work together without changes.

This problem often occurs during a transitional period, when the long-term plan is to phase out the legacy system.

Example:

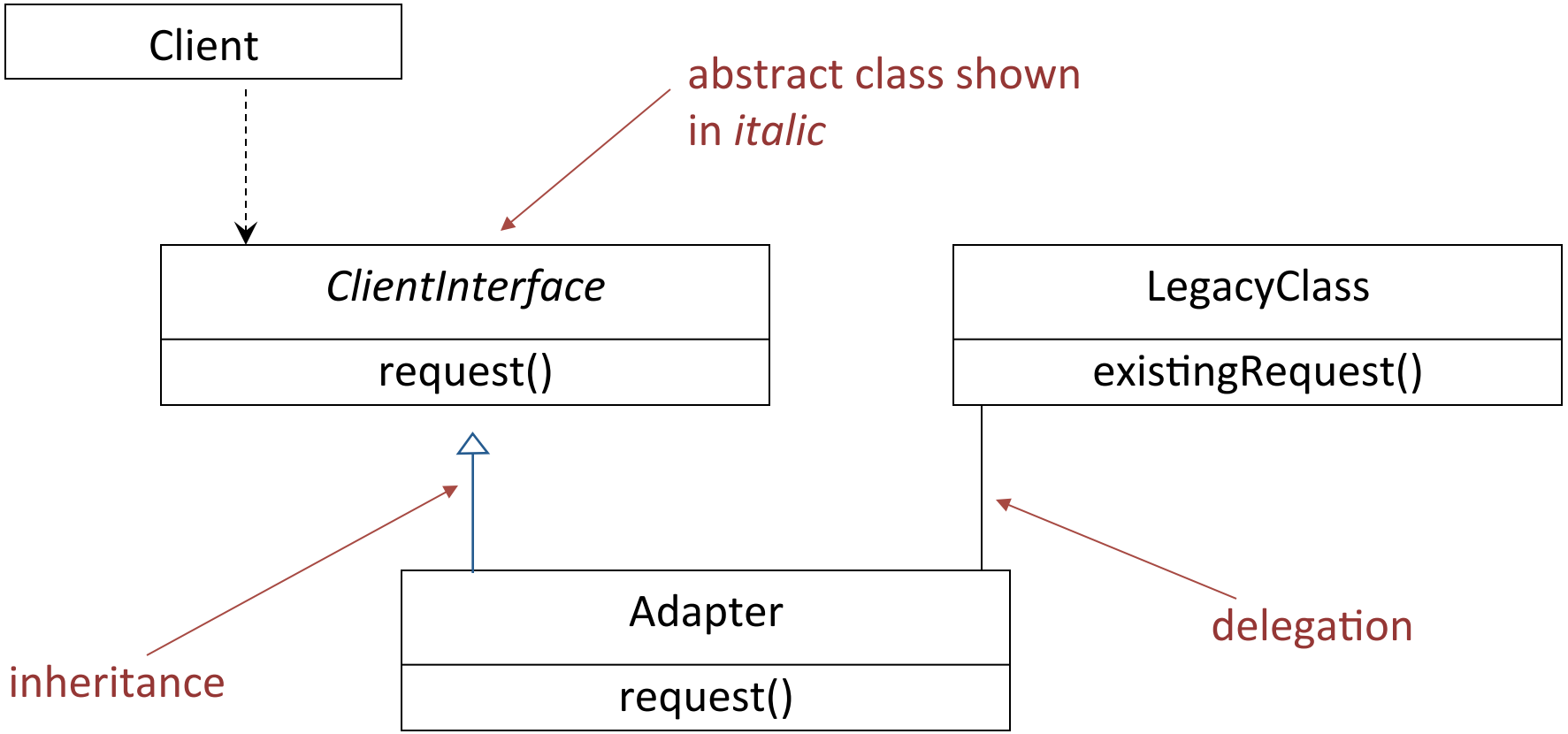
How do you use a web browser to access an information retrieval system that was designed for a different client?

Adapter Design Pattern: The Problem



During the transition, how can
NewClient be used with LegacyClass?

Adapter Design Pattern: Solution Class Diagram



Adapter Design Pattern: Consequences

The following **consequences** apply whenever the Adapter design pattern is used.

- Client and LegacyClass work together without modification of either.
- Adapter works with LegacyClass and all of its subclasses.
- A new Adapter needs to be written if Client is replaced by a subclass.

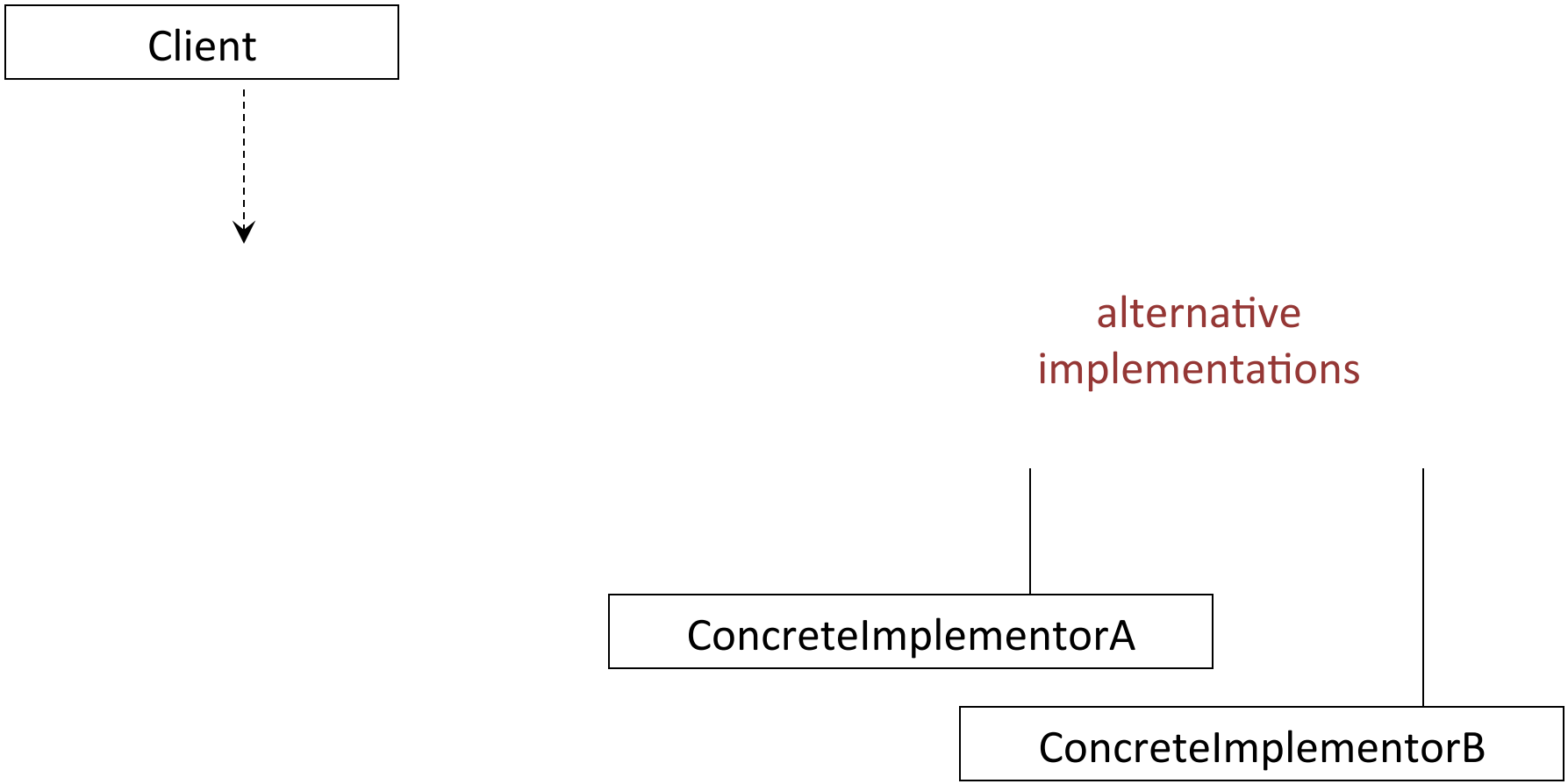
Bridge: Allowing for Alternate Implementations

Name: Bridge design pattern

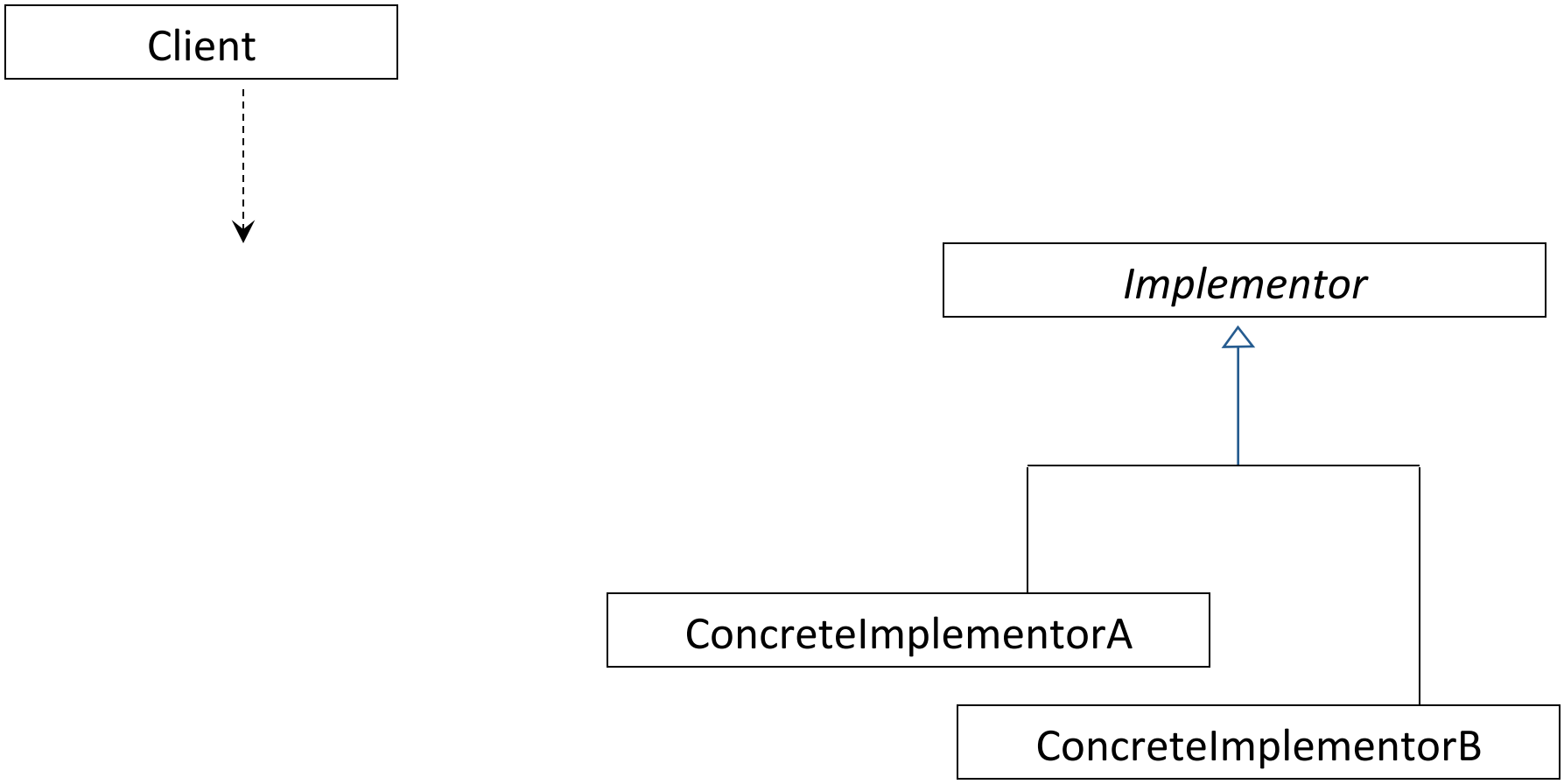
Problem description:

Decouple an interface from an implementation so that a different implementation can be substituted, possibly at runtime (e.g., testing different implementations of the same interface).

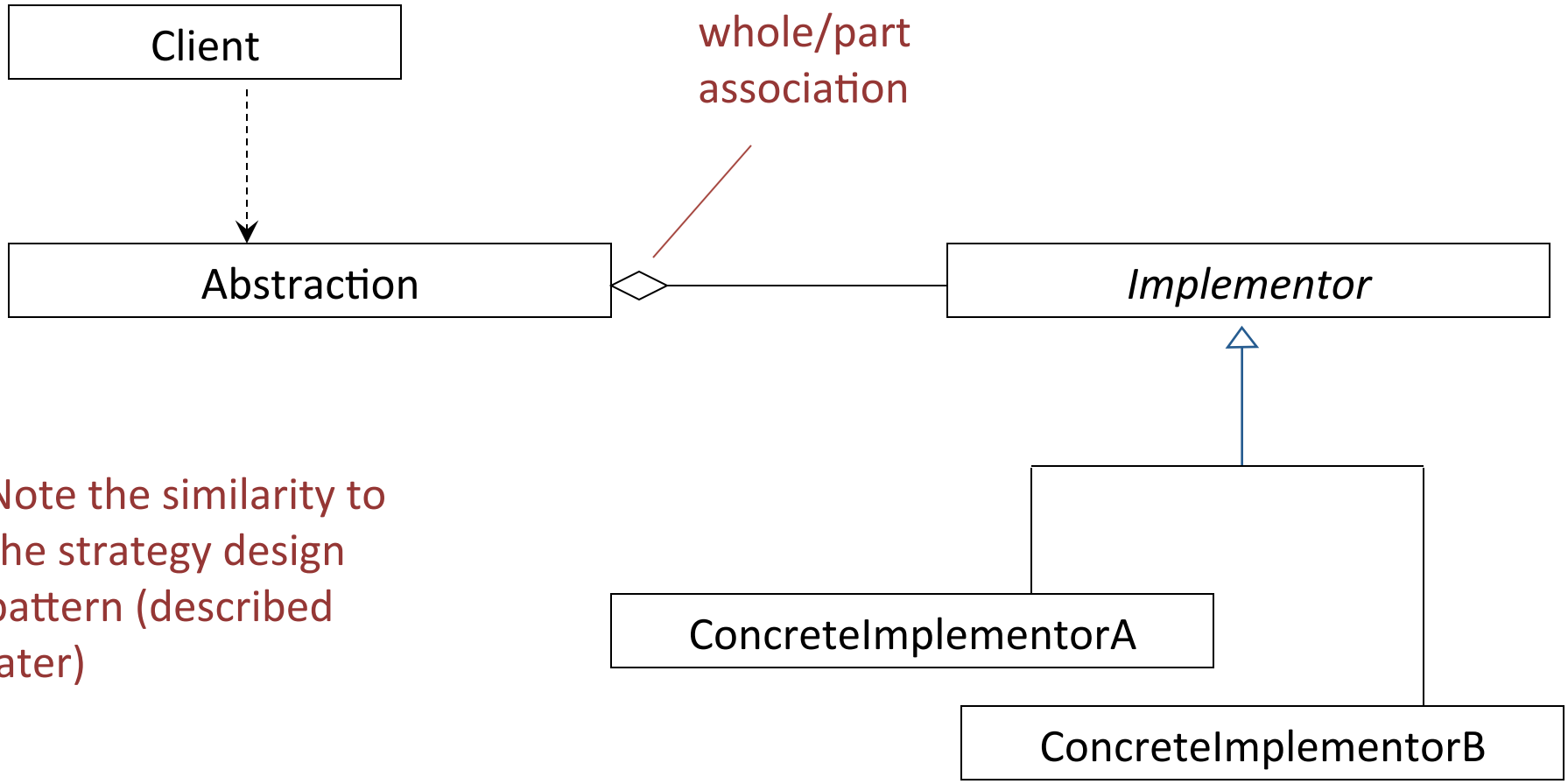
Bridge: Class Diagram



Bridge: Class Diagram



Bridge:
Class Diagram



Note the similarity to the strategy design pattern (described later)

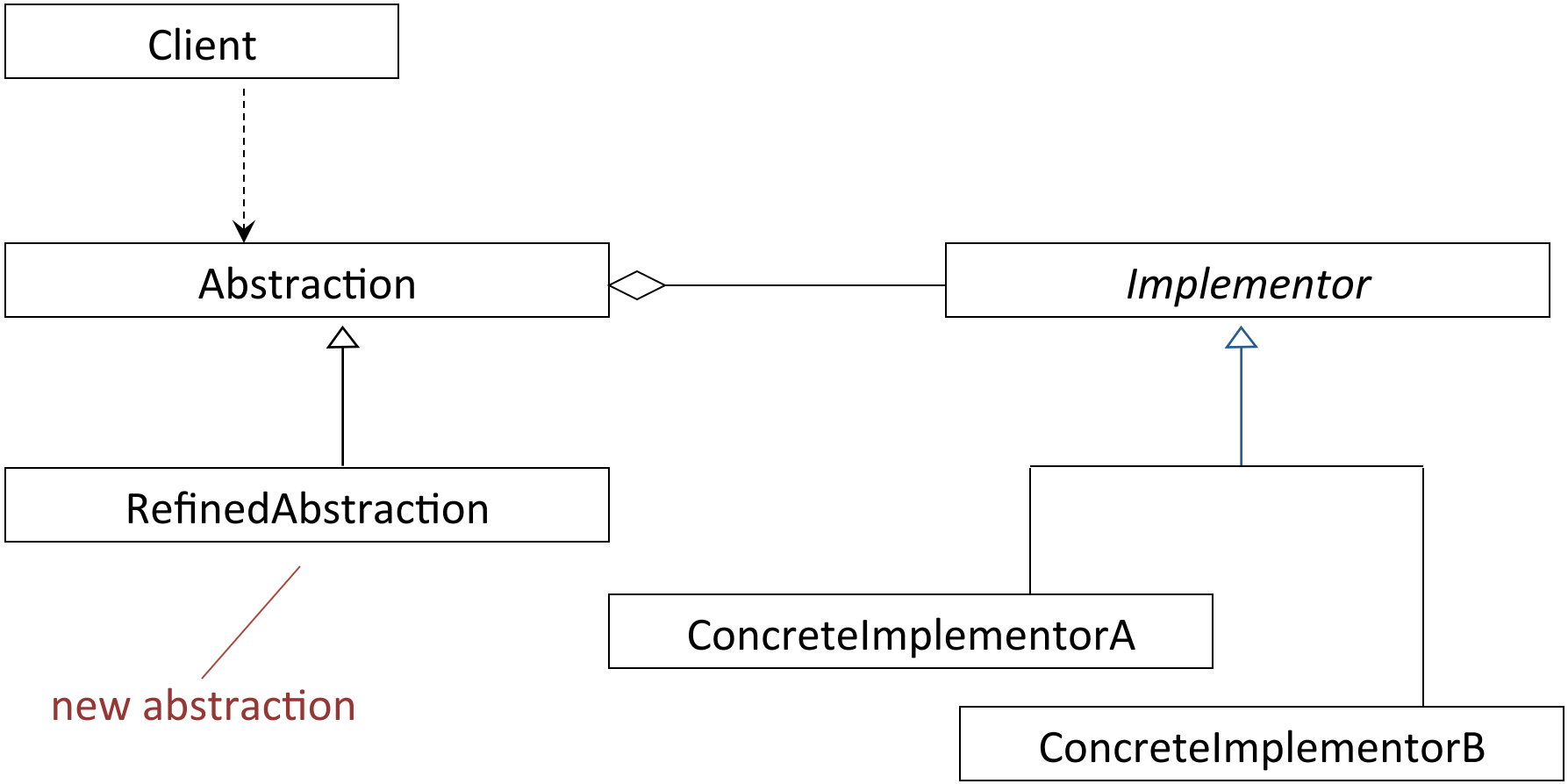
Bridge: Allowing for Alternate Implementations

Solution:

The Abstraction class defines the interface visible to the client. Implementor is an abstract class that defines the lower-level methods available to Abstraction. An Abstraction instance maintains a reference to its corresponding Implementor instance.

Abstraction and Implementor can be refined independently.

Bridge:
Class Diagram



Bridge: Consequences

Consequences:

Client is shielded from abstract and concrete implementations

Interfaces and implementations can be tested separately

Strategy: Encapsulating Algorithms

Name: Strategy design pattern

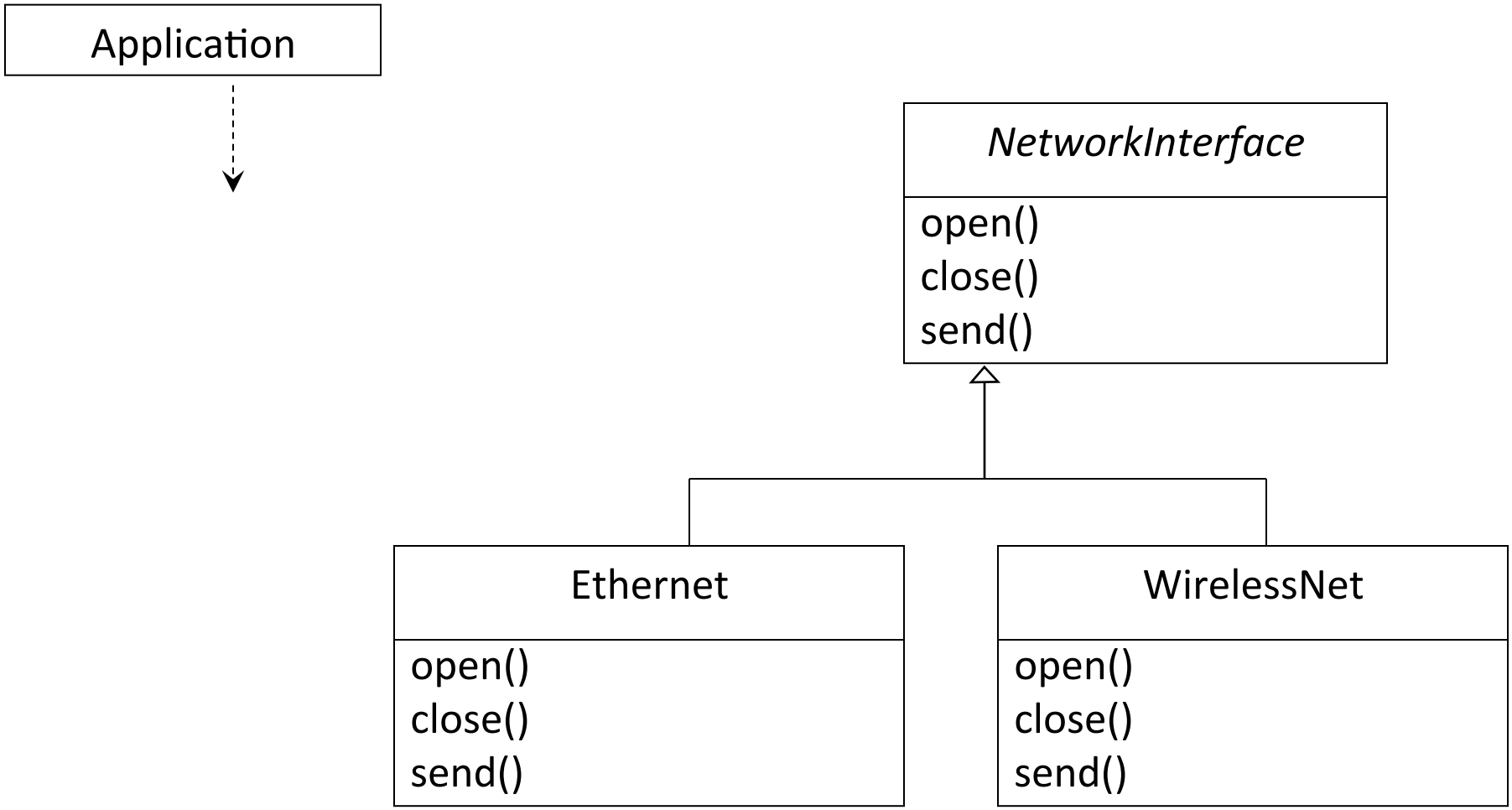
Example:

A mobile computer can be used with a wireless network, or connected to an Ethernet, with dynamic switching between networks based on location and network costs.

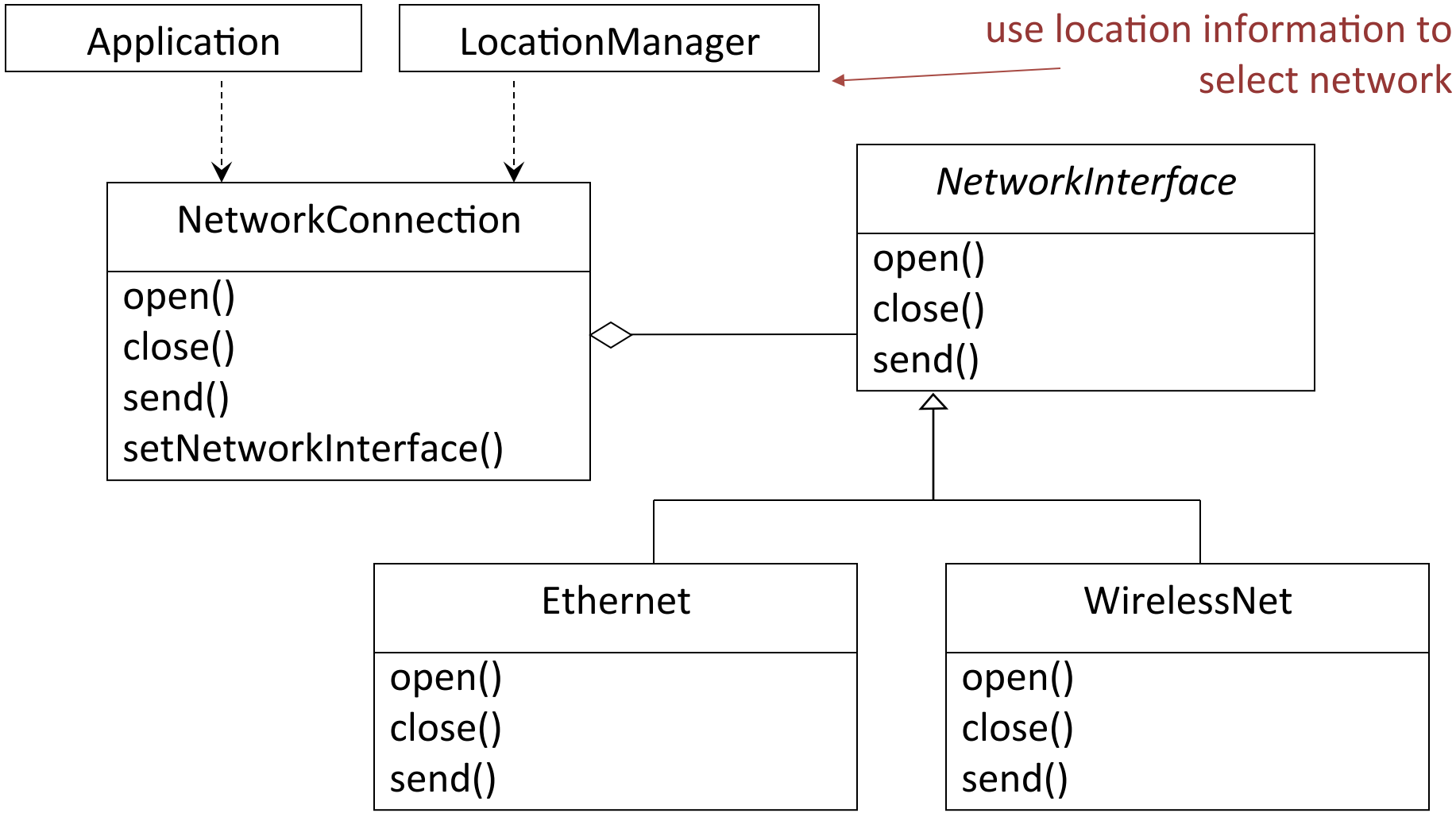
Problem description:

Decouple a policy-deciding class from a set of mechanisms, so that different mechanisms can be changed transparently.

Strategy Example: Class Diagram for Mobile Computer



Strategy Example: Class Diagram for Mobile Computer



Strategy: Encapsulating Algorithms

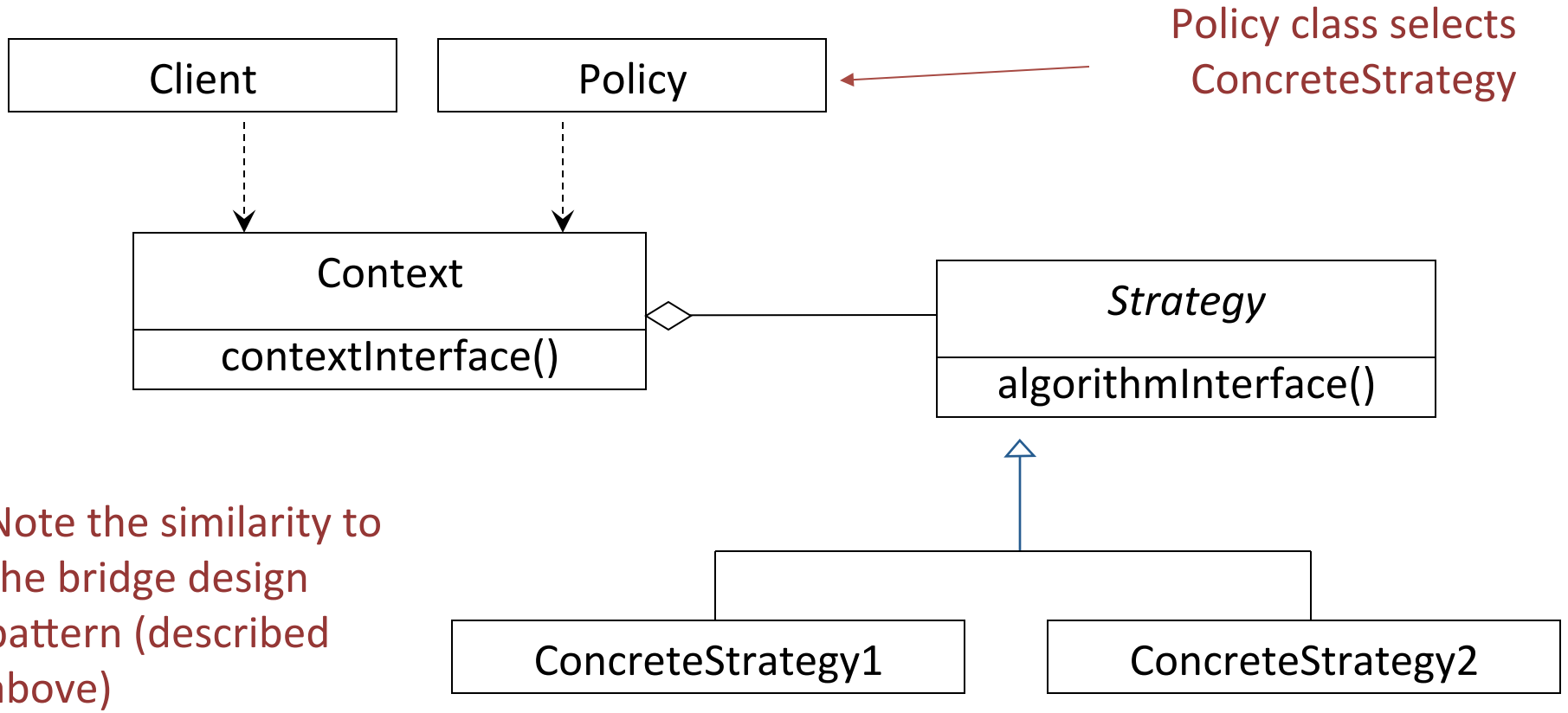
Solution:

A Client accesses services provided by a Context.

The Context services are realized using one of several mechanisms, as decided by a Policy object.

The abstract class Strategy describes the interface that is common to all mechanisms that Context can use. Policy class creates a ConcreteStrategy object and configures Context to use it.

Strategy: Class Diagram



Note the similarity to the bridge design pattern (described above)

Strategy: Consequences

Consequences:

ConcreteStrategies can be substituted transparently from Context.

Policy decides which Strategy is best, given the current circumstances.

New policy algorithms can be added without modifying Context or Client.

Facade: Encapsulating Subsystems

Name: Facade design pattern

Problem description:

Reduce coupling between a set of related classes and the rest of the system.

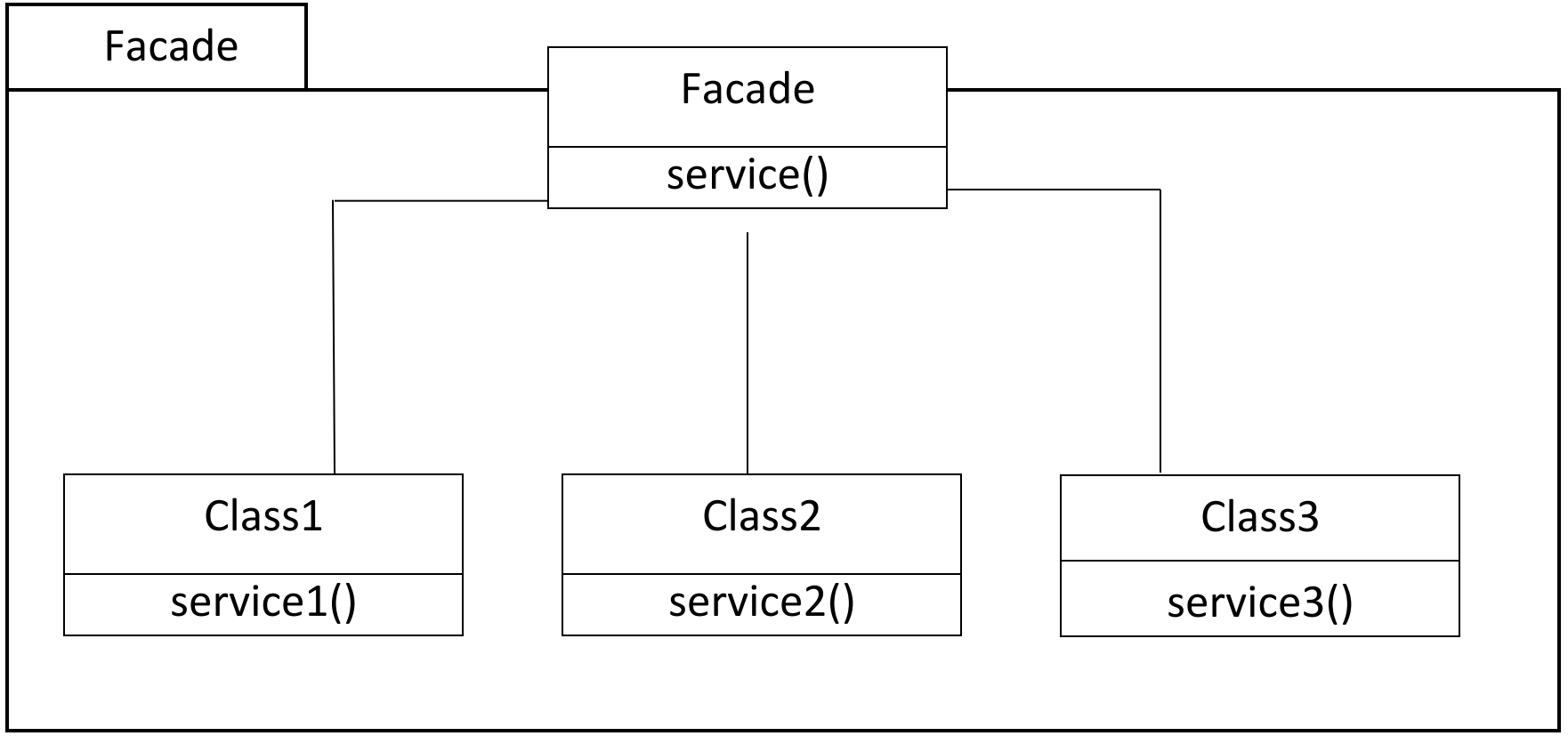
Example:

A Compiler is composed of several classes: LexicalAnalyzer, Parser, CodeGenerator, etc. A caller invokes only the Compiler (Facade) class, which invokes the contained classes.

Solution:

A single Facade class implements a high-level interface for a subsystem by invoking the methods of the lower-level classes.

Facade: Class Diagram



Facade: Consequences

Consequences:

- Shields a client from the low-level classes of a subsystem.
- Simplifies the use of a subsystem by providing higher-level methods.
- Enables lower-level classes to be restructured without changes to clients.

Note. The repeated use of Facade patterns yields a layered system.

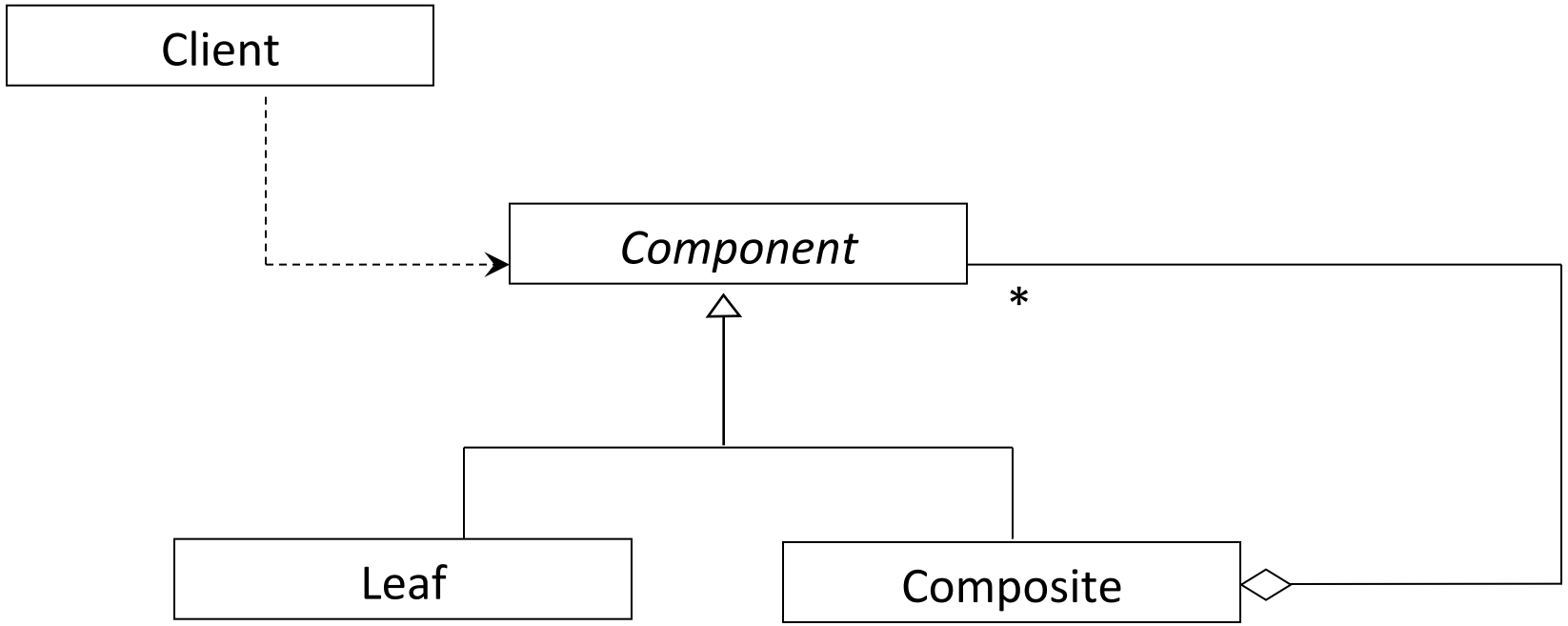
Composite: Representing Recursive Hierarchies

Name: Composite design pattern

Problem description:

Represent a hierarchy of variable width and depth, so that the leaves and composites can be treated uniformly through a common interface.

Composite: Class Diagram



Composite: Representing Recursive Hierarchies

Solution:

The Component interface specifies the services that are shared between Leaf and Composite. A Composite has an aggregation association with Components and implements each service by iterating over each contained Component. The Leaf services do the actual work.

Composite: Consequences

Consequences:

Client uses the same code for dealing with Leaves or Composites.

Leaf-specific behavior can be changed without changing the hierarchy.

New classes of Leaves can be added without changing the hierarchy.

Proxy: Encapsulating Expensive Objects

Name: Proxy design pattern

Problem description:

Improve performance or security of a system by delaying expensive computations, using memory only when needed, or checking access before loading an object into memory.

Solution:

The ProxyObject class acts on behalf of a RealObject class. Both implement the same interface. ProxyObject stores a subset of the attributes of RealObject. ProxyObject handles certain requests, whereas others are delegated to RealObject. After delegation, the RealObject is created and loaded into memory.

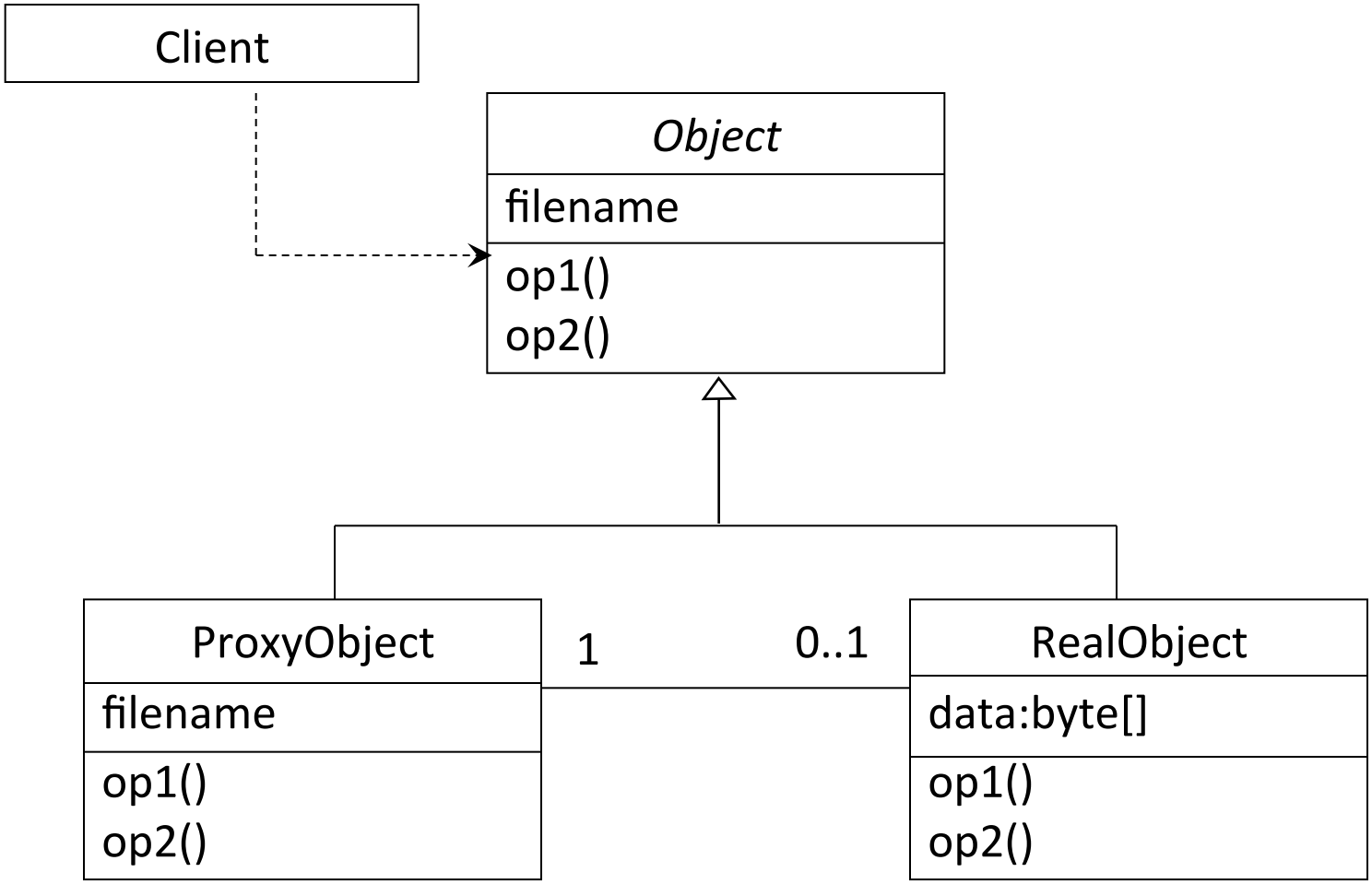
Proxy: Example

An abstract class *SortStrings* defines an interface for sorting lists of strings.

ProxySortStrings is a class that sorts lists of strings very quickly in memory and delegates larger lists.

RealSortStrings is a class that sorts very large lists of strings, but is expensive to create and execute on small lists.

Proxy: Class Diagram



Proxy: Consequences

Consequences:

Adds a level of indirection between Client and RealObject.

The Client is shielded from any optimization for creating RealObjects.

Abstract Factory: Encapsulating Platforms

Name: Abstract Factory design pattern

Problem description:

Shield the client from different platforms that provide different implementations of the same set of concepts

Example:

A user interface must have versions that implement the same set of concepts for several windowing systems, e.g., scroll bars, buttons, highlighting, etc.

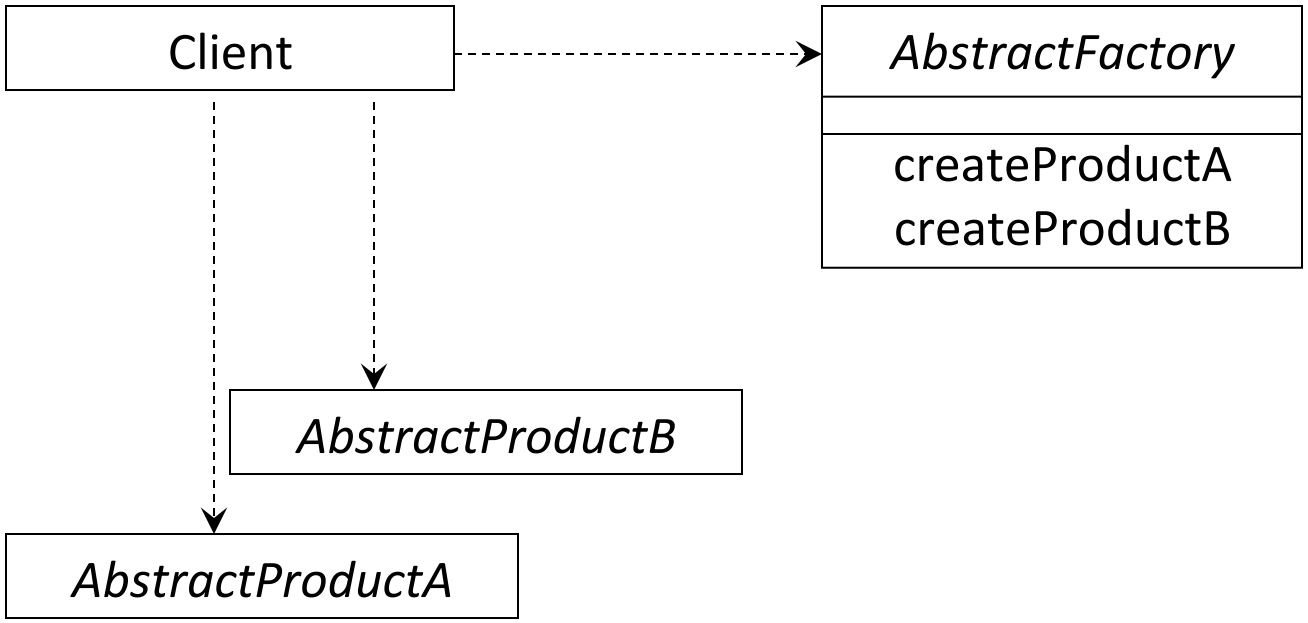
Abstract Factory: Encapsulating Platforms

Solution:

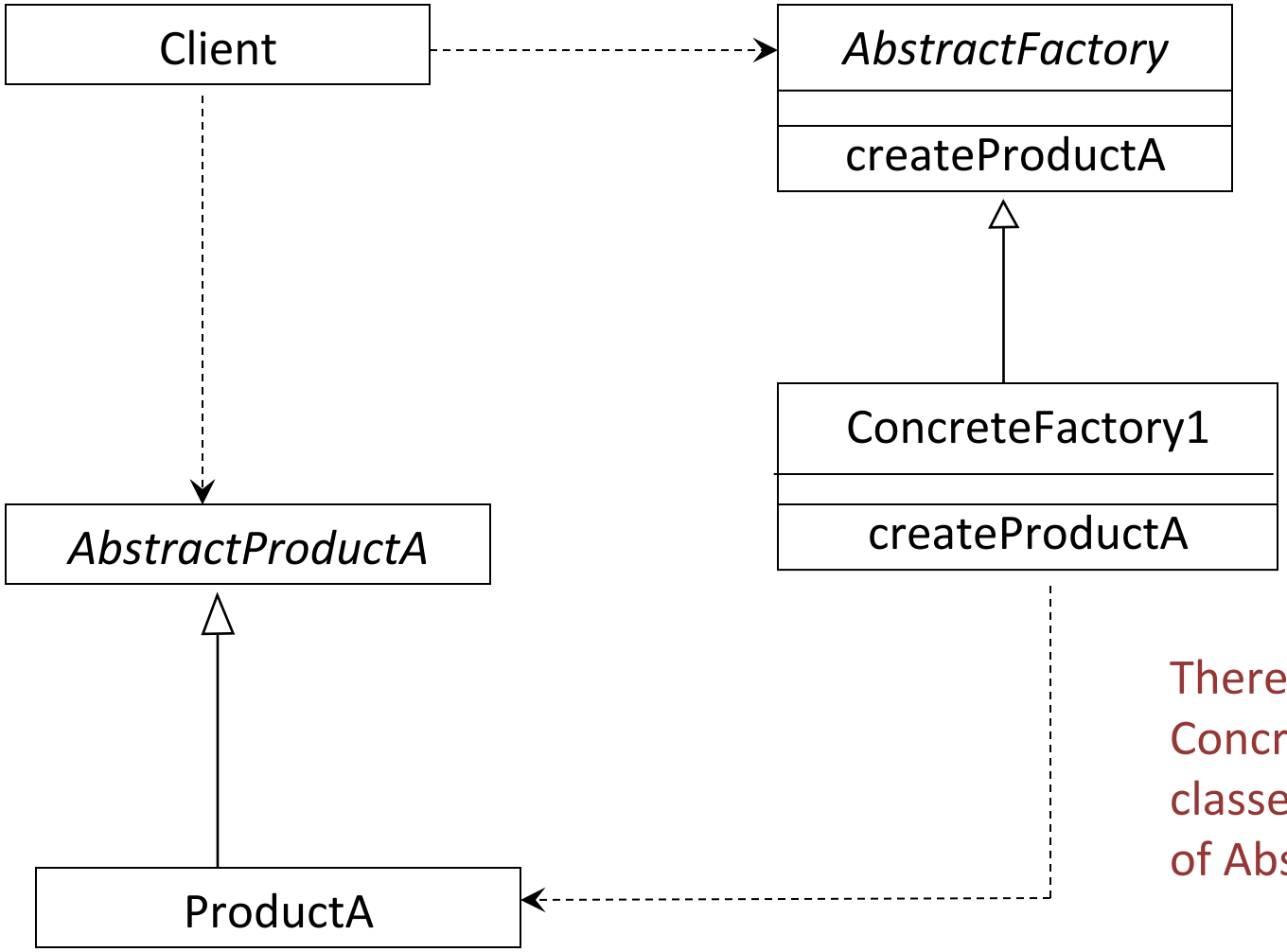
A platform (e.g., the application for a specific windowing system) is represented as a set of AbstractProducts, each representing a concept (e.g., button). An AbstractFactory class declares the operations for creating each individual product.

A specific platform is then realized by a ConcreteFactory and a set of ConcreteProducts.

Abstract Factory: Class Diagram

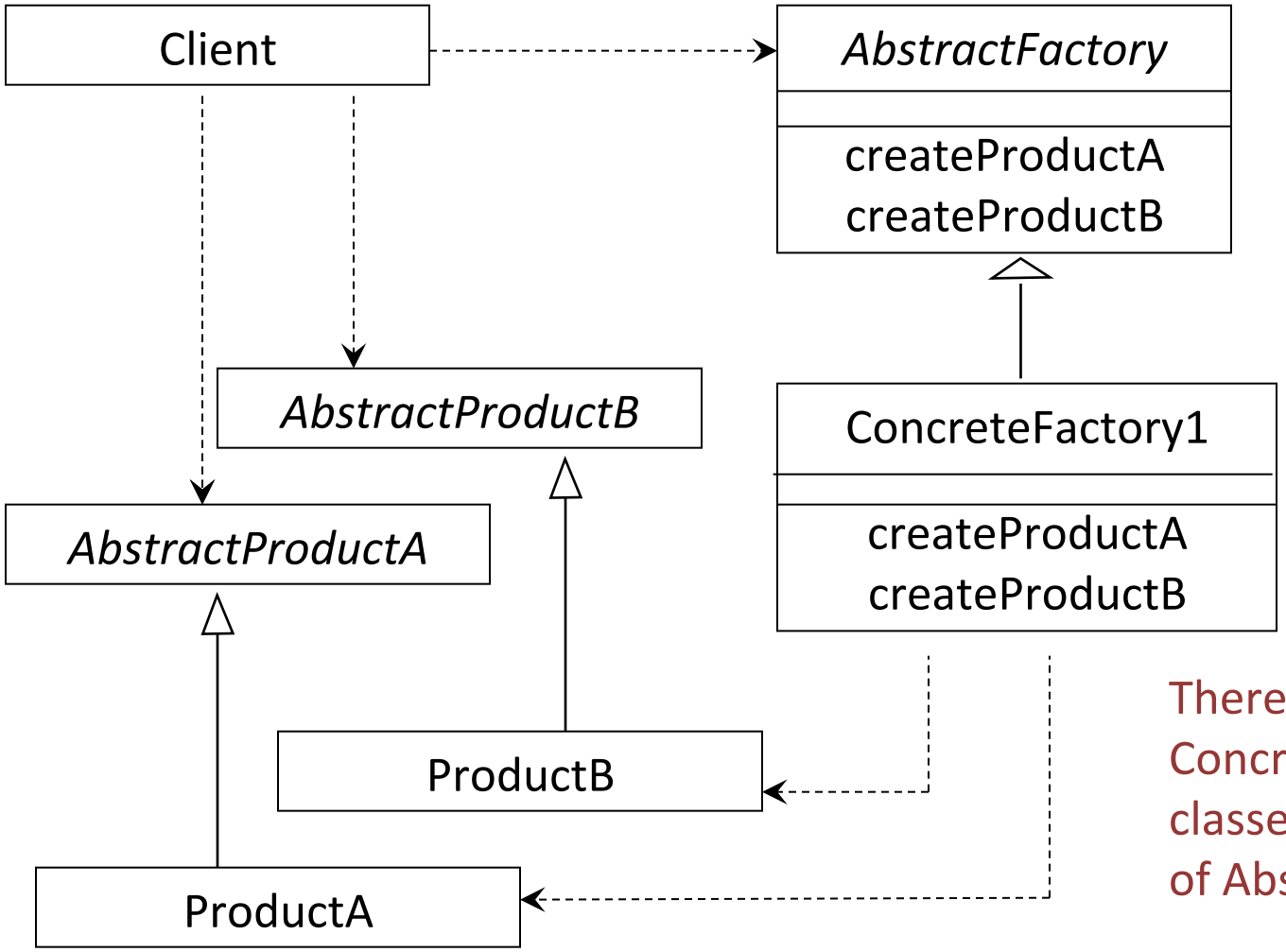


Abstract Factory: Class Diagram



There could be several ConcreteFactory classes, each a subclass of AbstractFactory

Abstract Factory: Class Diagram



There could be several ConcreteFactory classes, each a subclass of AbstractFactory

Abstract Factory: Consequences

Consequences:

Client is shielded from concrete products classes

Substituting families at runtime is possible

Adding new products is difficult since new realizations must be created for each factory

Abstract Factory: Discussion

Discussion

See the interesting discussion in Wikipedia (October 25, 2010):

"Use of this pattern makes it possible to interchange concrete classes without changing the code that uses them, even at runtime. However, employment of this pattern, as with similar design patterns, may result in unnecessary complexity and extra work in the initial writing of code."

An Old Exam Question

A company that makes sports equipment decides to create a system for selling sports equipment online. The company already has a **product database** with specification, marketing information, and prices of the equipment that it manufactures.

To sell equipment online the company will need to create: a **customer database**, and an **ordering system** for online customers.

The plan is to develop the system in two phases. During Phase 1, simple versions of the customer database and ordering system will be brought into production. In Phase 2, major enhancements will be made to these components.

An Old Exam Question

Carefully design during Phase 1 will help the subsequent development of new components in Phase 2.

(a) For the interface between the ordering system and the customer database:

i Select a design pattern that will allow a gradual transition from Phase 1 to Phase 2.

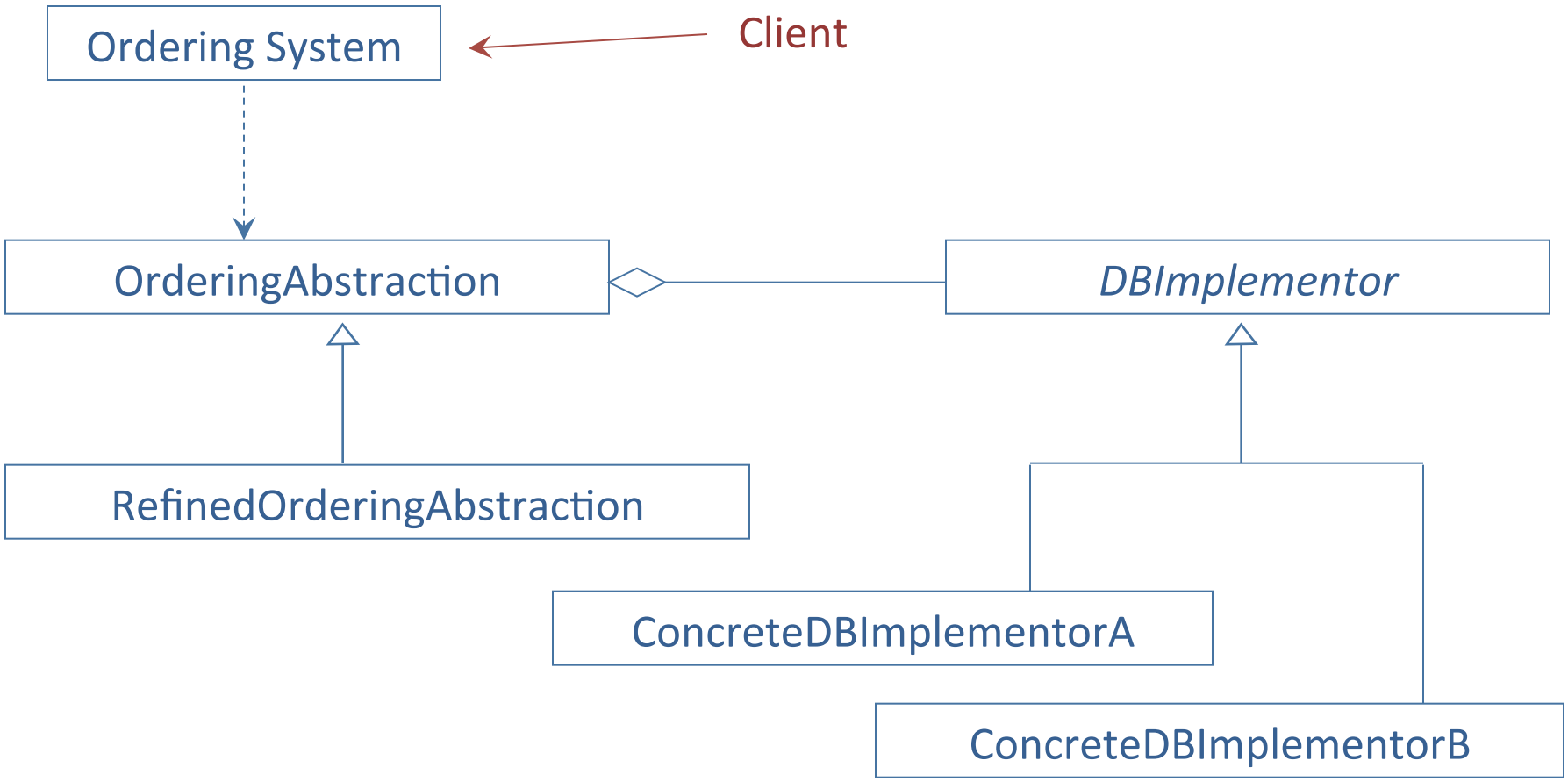
Bridge design pattern

ii Draw a UML class diagram that shows how this design pattern will be used in Phase 1.

If your diagram relies on abstract classes, inheritance, delegation or similar properties be sure that this is clear on your diagram.

[See next slide]

An Old Exam Question



An Old Exam Question

(c) How does this design pattern support:

i Enhancements to the ordering system in Phase 2?

By subclassing OrderingAbstraction

ii A possible replacement of the customer database in Phase 2?

By allowing several ConcreteDBImplementor classes