Cornell University
Computing and Information Science

CS 5150 Software Engineering
18. Reuse and Design Patterns

William Y. Arms

Software Reuse

It is often good to design a program to reuse existing components. This can lead to better software at lower cost.

Potential benefits of reuse

- Reduced development time and cost
- Improved reliability of mature components
- Shared maintenance cost

Potential disadvantages of reuse

- Difficulty in finding appropriate components
- Components may be a poor fit for application
- Quality control and security may be unknown

Evaluating Software

Software from well established developers is likely to be well written and tested, but still will have bugs and security weaknesses, especially when incorporated in unusual applications.

The software is likely to be much better than a new development team would write.

But sometimes it is sensible to write code for a narrowly defined purpose rather than use general purpose software.

Maintenance

When evaluating software, both commercial and open source, pay attention to maintenance. Is the software supported by an organization that will continue maintenance over the long term?

Reuse: Open Source Software

Open source software varies enormously in quality.

- Because of the processes for reporting and fixing problems, major systems such as Linux, Apache, Python, Lucene, etc. tend to be very robust and free from problems. They are often better than the commercial equivalents.
- More experimental systems, such as Hadoop, have solid cores, but their lesser used features have not been subject to the rigorous quality control of the the best software products.
- Other open source software is of poor quality and should not be incorporated in production systems.

Evaluating Applications Packages

Applications packages for business functions are provided by companies such as SAP and Oracle. They provide **enormous capabilities** and relieve an organization from such tasks as updating financial systems when laws change.

They are very expensive:

- License fees to the vendor.
- Modifications to existing systems and special code from the vendor.
- Disruption to the organization when installing them.
- Long term maintenance costs.
- The costs of changing to a different vendor are huge.

Cornell's decision (about 1990) to move to PeopleSoft (now part of Oracle) has cost the university several hundred millions of dollars.

If you are involved in such a decision insist on a **very thorough feasibility study**. Be prepared to take a least a year and spend several million dollars before making the decision.

Design for Change: Replacement of Components

The software design should anticipate possible changes in the system over its life-cycle.

New vendor or new technology

Components are replaced because its supplier goes out of business, ceases to provide adequate support, increases its price, etc., or because software from another source provides better functionality, support, pricing, etc.

This can apply to either open source or vendor-supplied components.

Design for Change: Replacement of Components

New implementation

The original implementation may be problematic, e.g., poor performance, inadequate back-up and recovery, difficult to trouble-shoot, or unable to support growth and new features added to the system.

Example. The portal nsdl.org was originally implemented using uPortal. This did not support important extensions that were requested and proved awkward to maintain. It was reimplemented using PHP/MySQL.

Design for Change: Replacement of Components

Additions to the requirements

When a system goes into production, it is usual to reveal both weaknesses and opportunities for extra functionality and enhancement to the user interface design.

For example, in a data-intensive system it is almost certain that there will be requests for extra reports and ways of analyzing the data.

Requests for enhancements are often the sign of a successful system. Clients recognize latent possibilities.

Design for Change: Replacement of Components

Changes in the application domain

Most application domains change continually, e.g., because of business opportunities, external changes (such as new laws), mergers and take-overs, new groups of users, new technology, etc., etc.,

It is rarely feasible to implement a completely new system when the application domain changes. Therefore existing systems must be modified. This may involve extensive restructuring, but it is important to reuse existing code as much as possible.

Design Patterns

Design patterns

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.

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.

Inheritance and Abstract Classes

Design patterns make extensive use of inheritance and abstract classes.

Classes can be defined in terms of other classes using inheritance. The generalization class is called the superclass and the specialization is called the subclass.

Abstract class

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

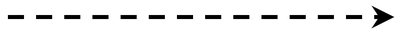
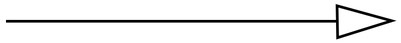
Delegation

Delegation

A class is said to delegate to another class if it implements an operation by resending a message to another class.

Delegation is an alternative to inheritance that can be used when reuse is anticipated.

Notation

ClassName	class name in italic indicates an abstract class
	dependency
	delegation
	inheritance

Adapter (Wrapper): Wrapping Around Legacy Code

Problem description:

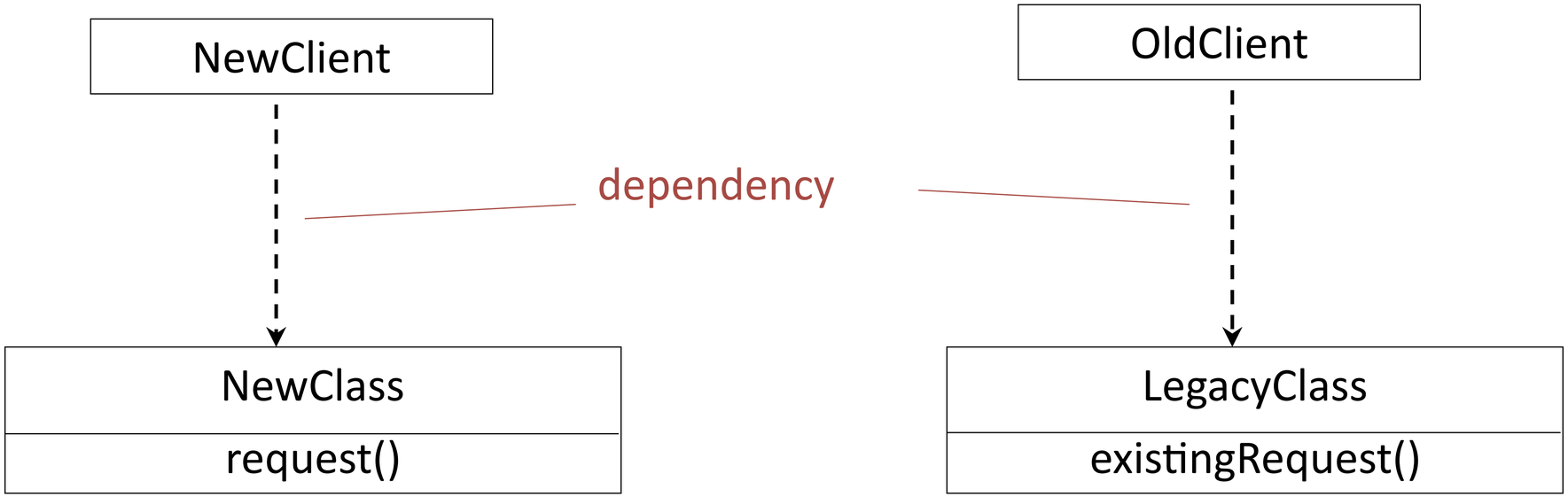
Convert the interface of a legacy class into a different interface expected by the client, so that the client and the legacy class can work together without changes.

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

Example:

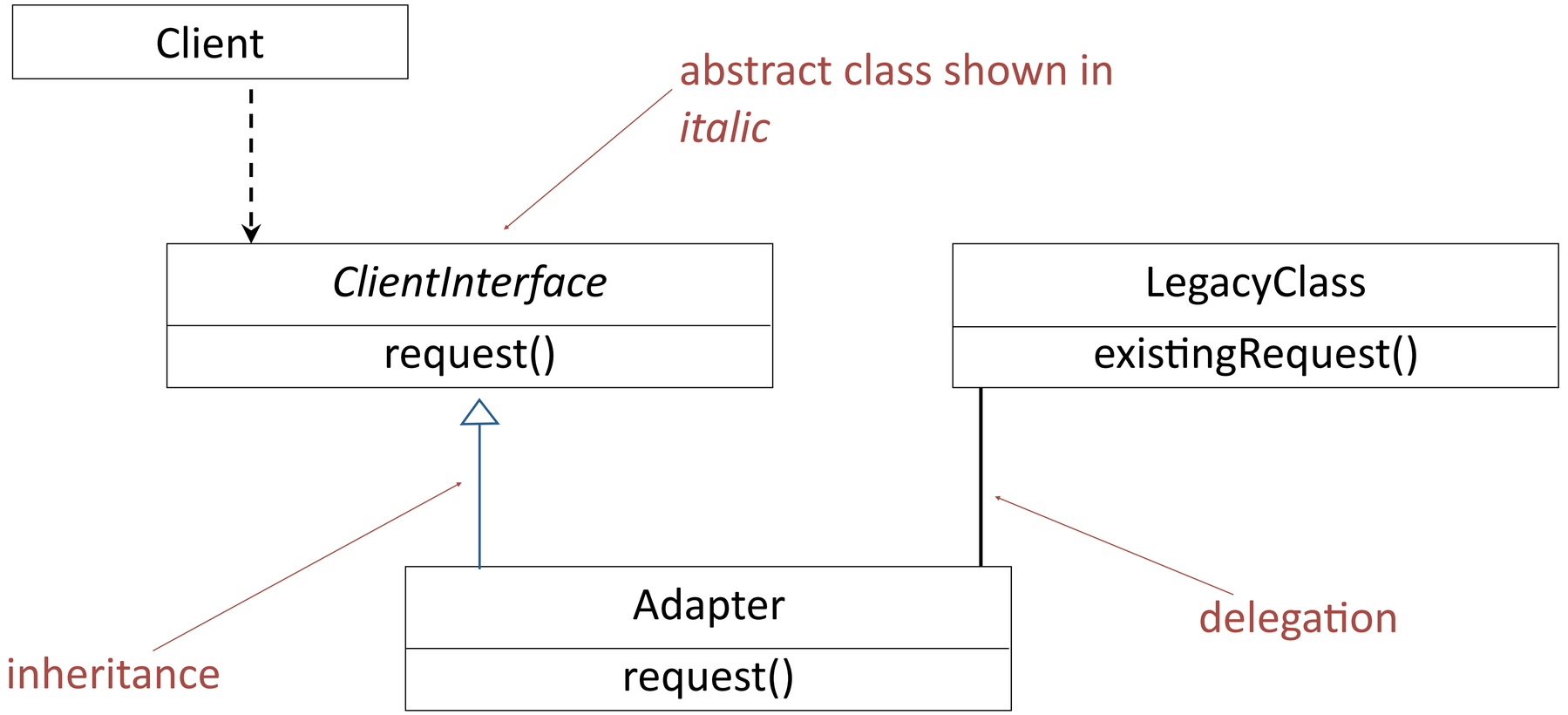
How do you use a web browser to access an information retrieval system that was designed for a different client?

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.

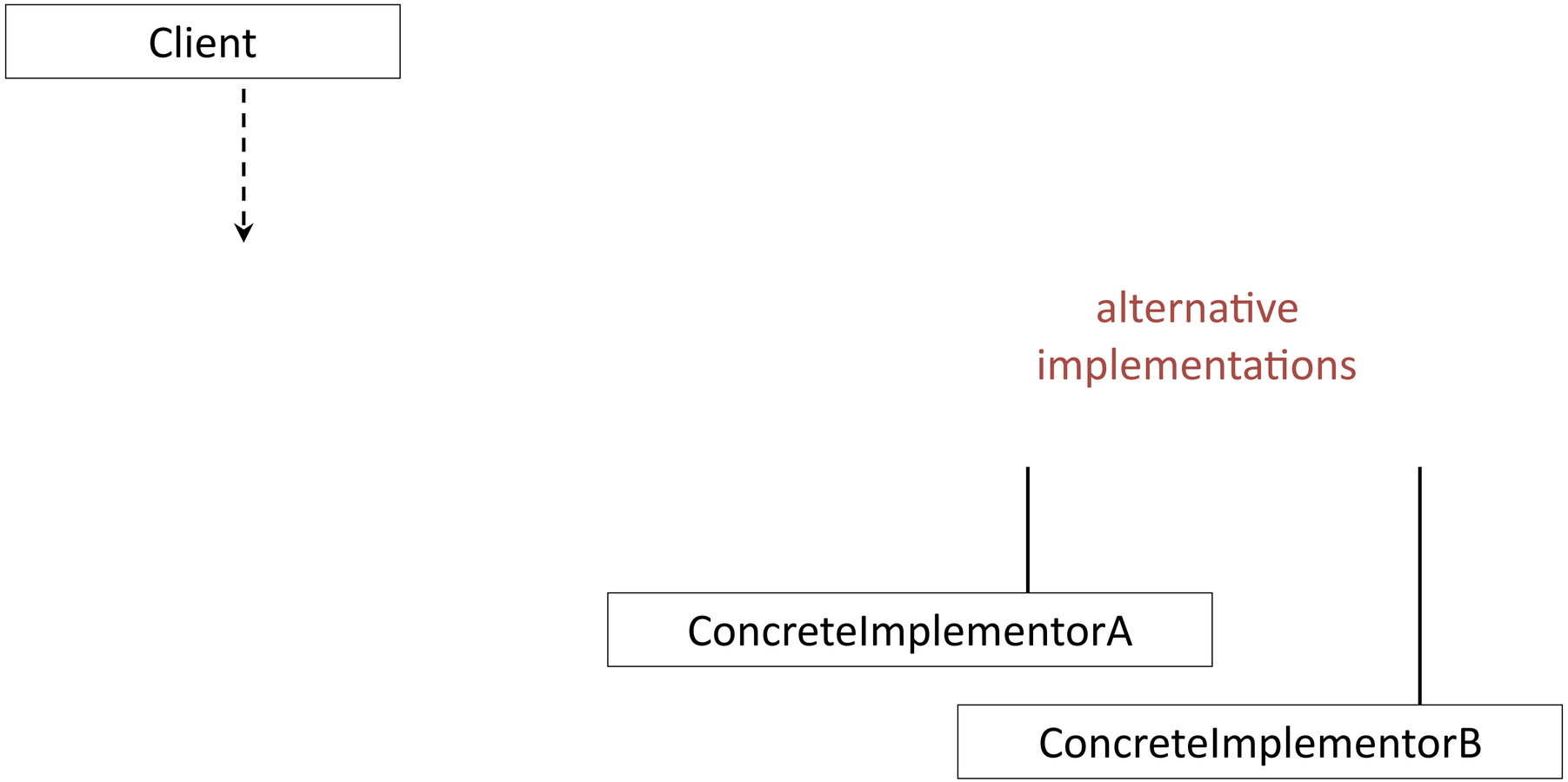
Bridge: Allowing for Alternate Implementations

Name: Bridge design pattern

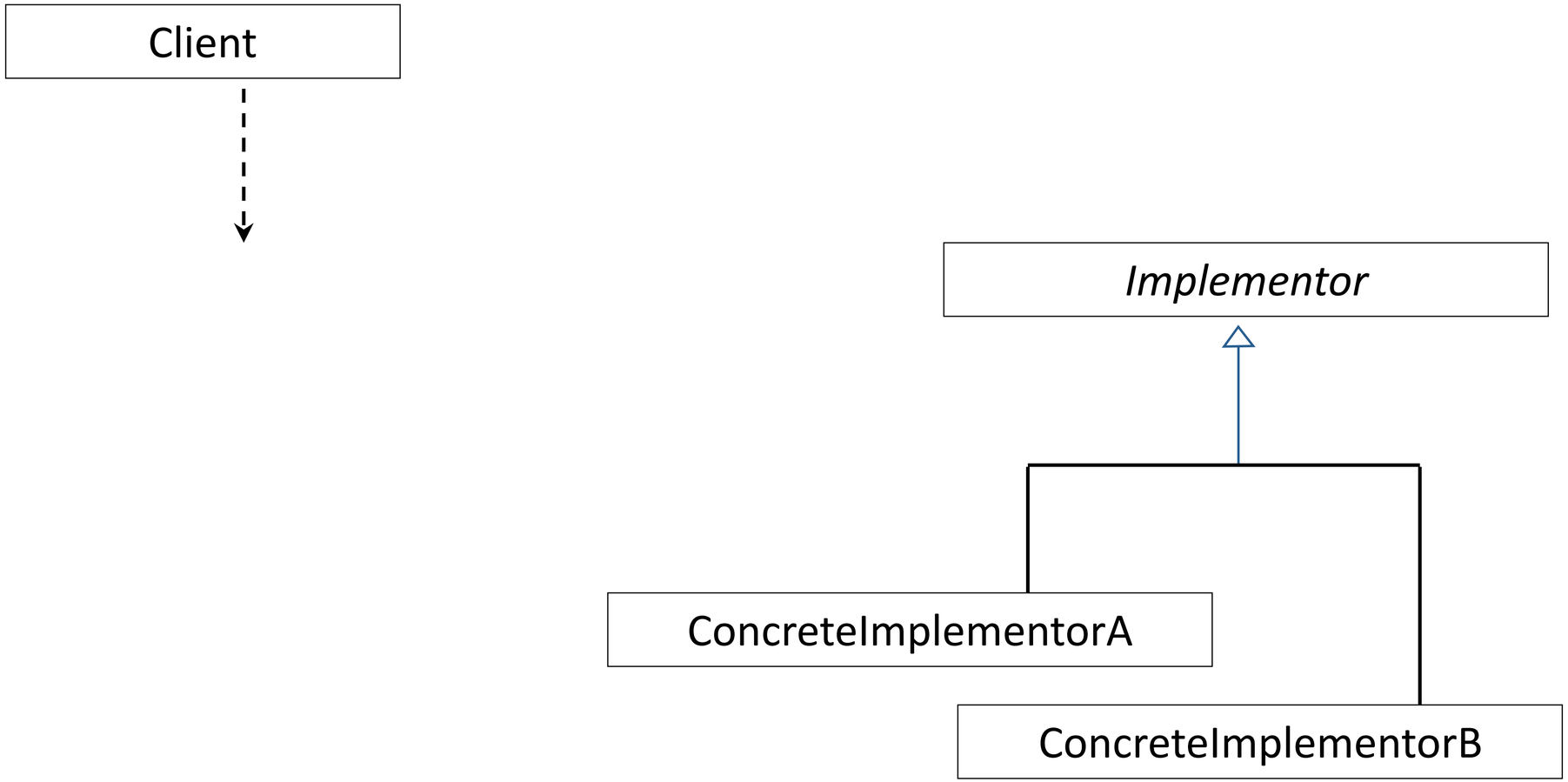
Problem description:

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).

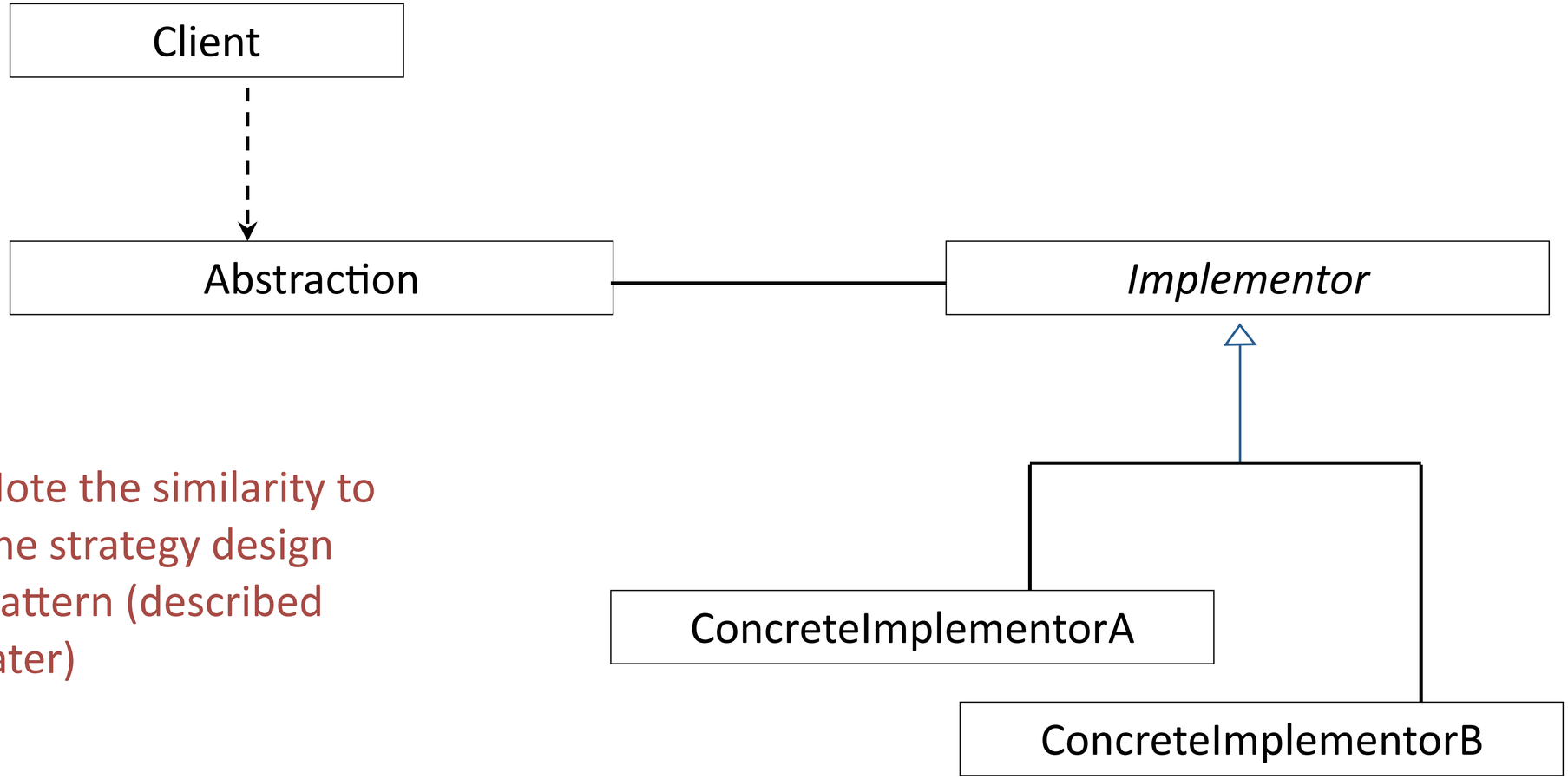
Bridge: Class Diagram



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: Consequences

Consequences:

- Client is shielded from abstract and concrete implementations
- Interfaces and implementations can be tested separately

Strategy: Encapsulating Algorithms

Name: Strategy design pattern

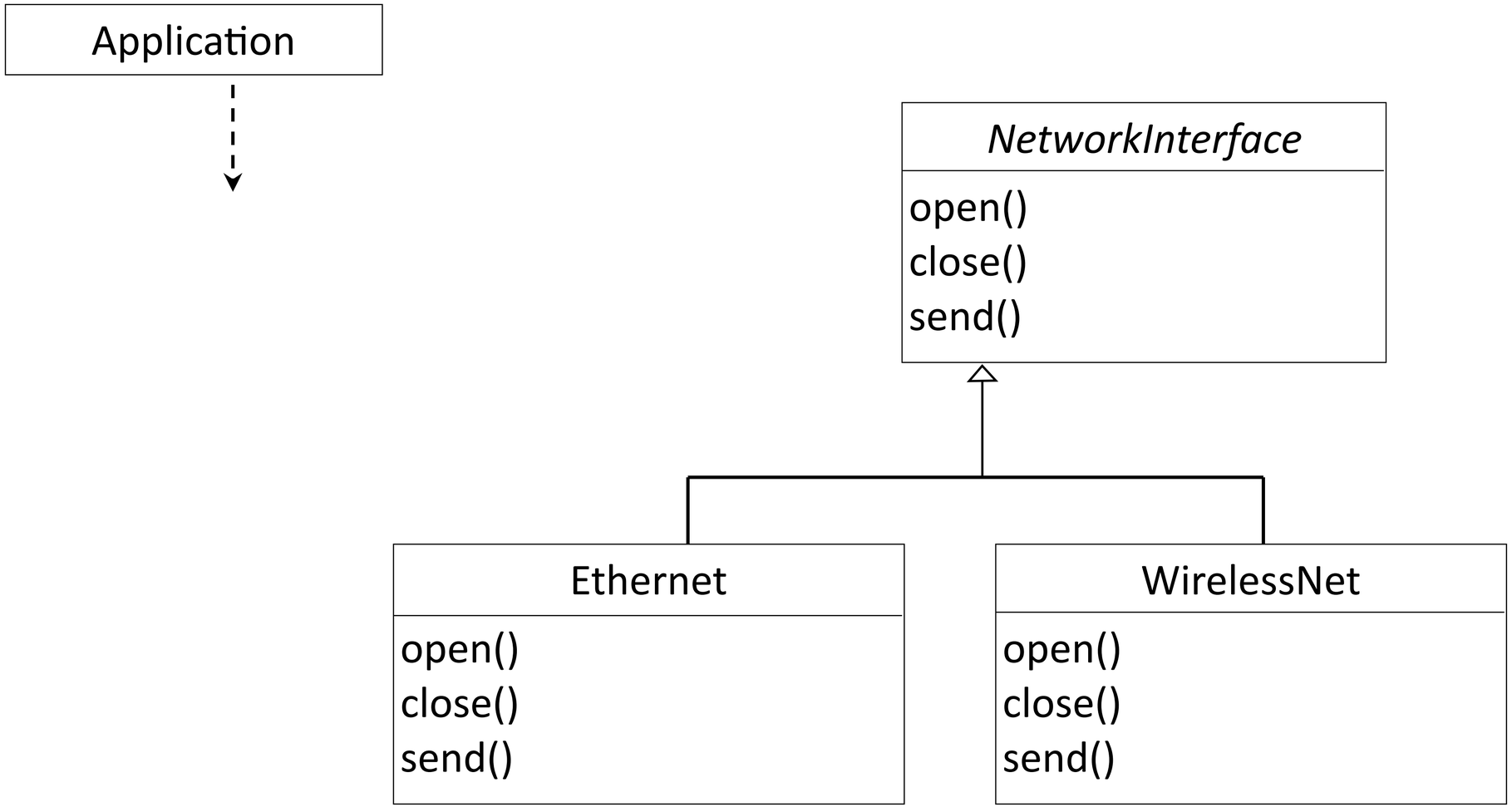
Example:

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.

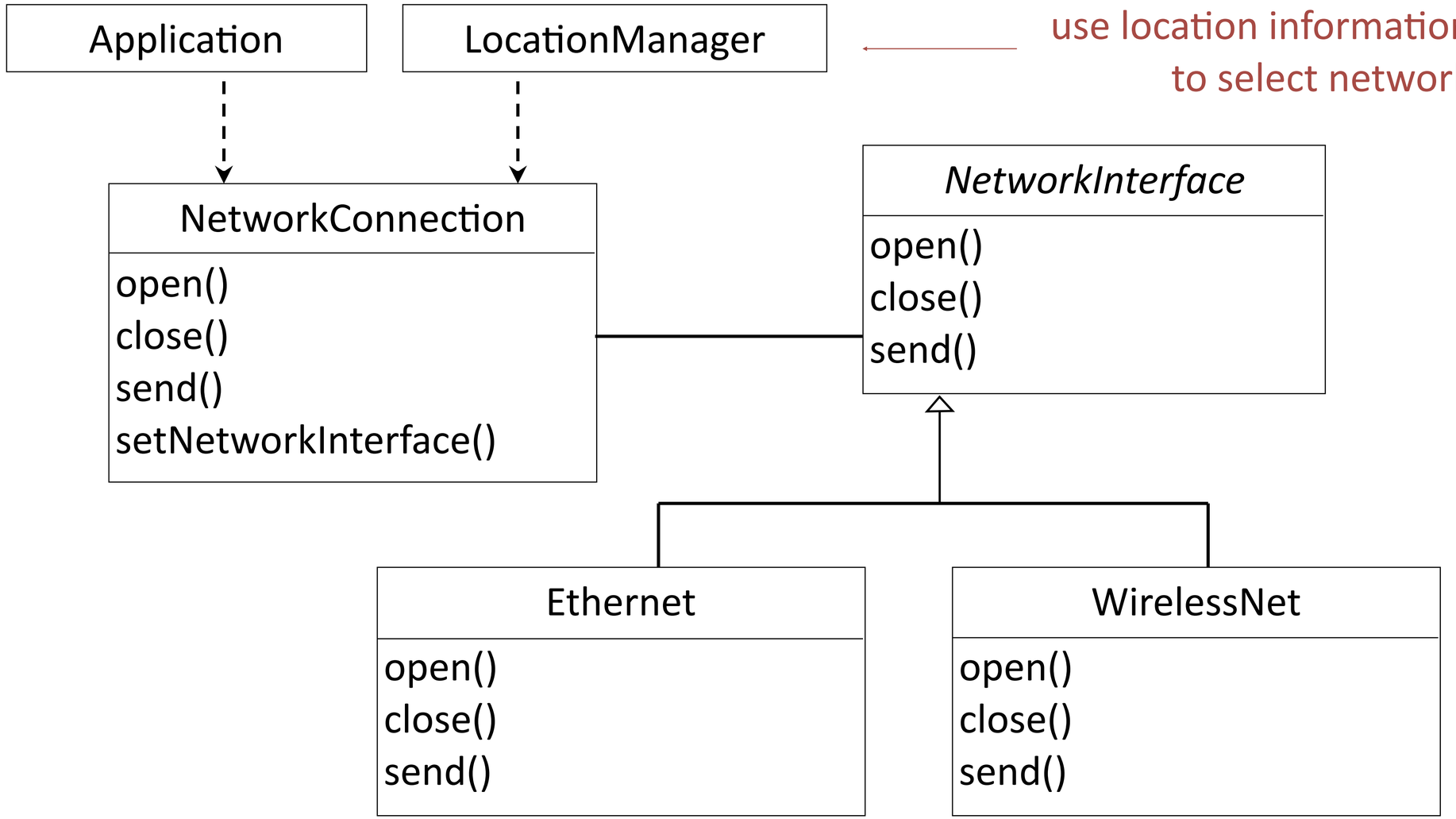
Problem description:

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

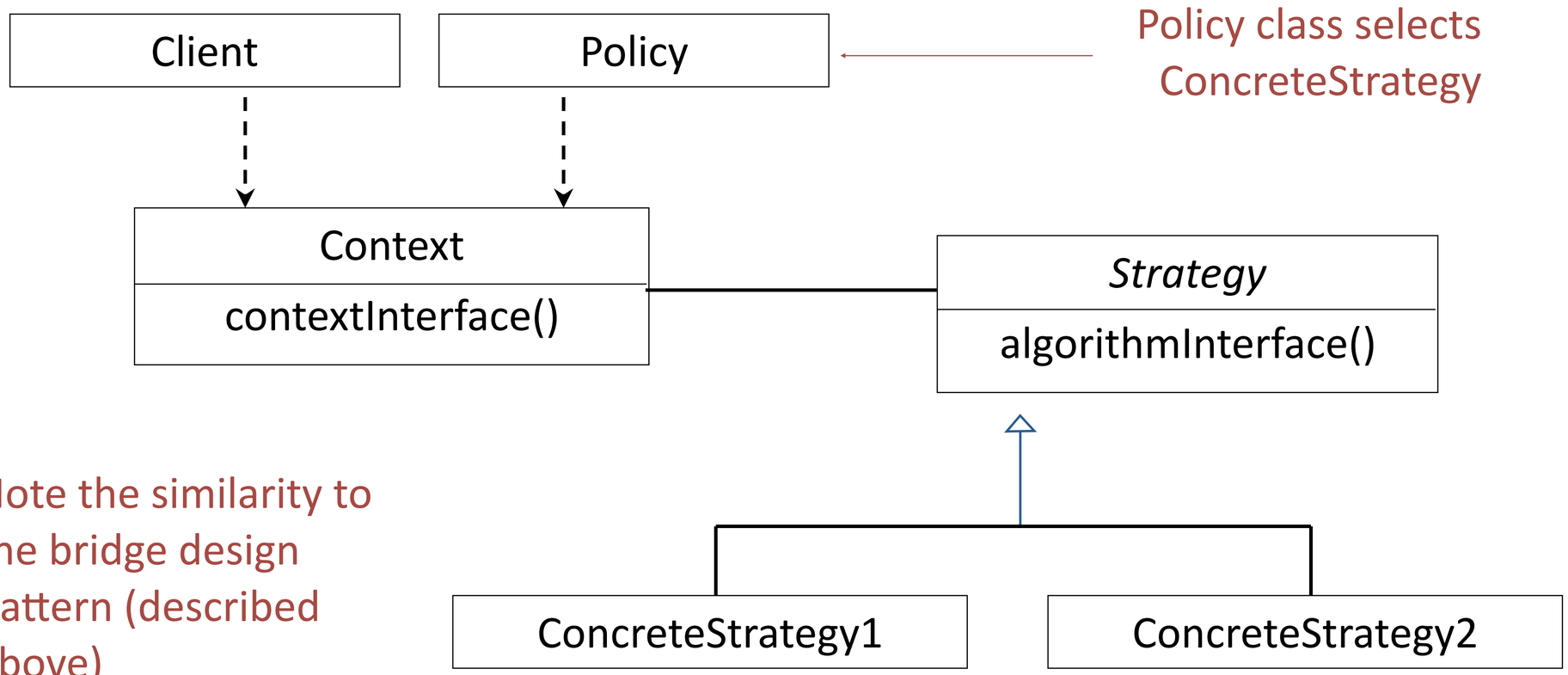
Solution:

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.

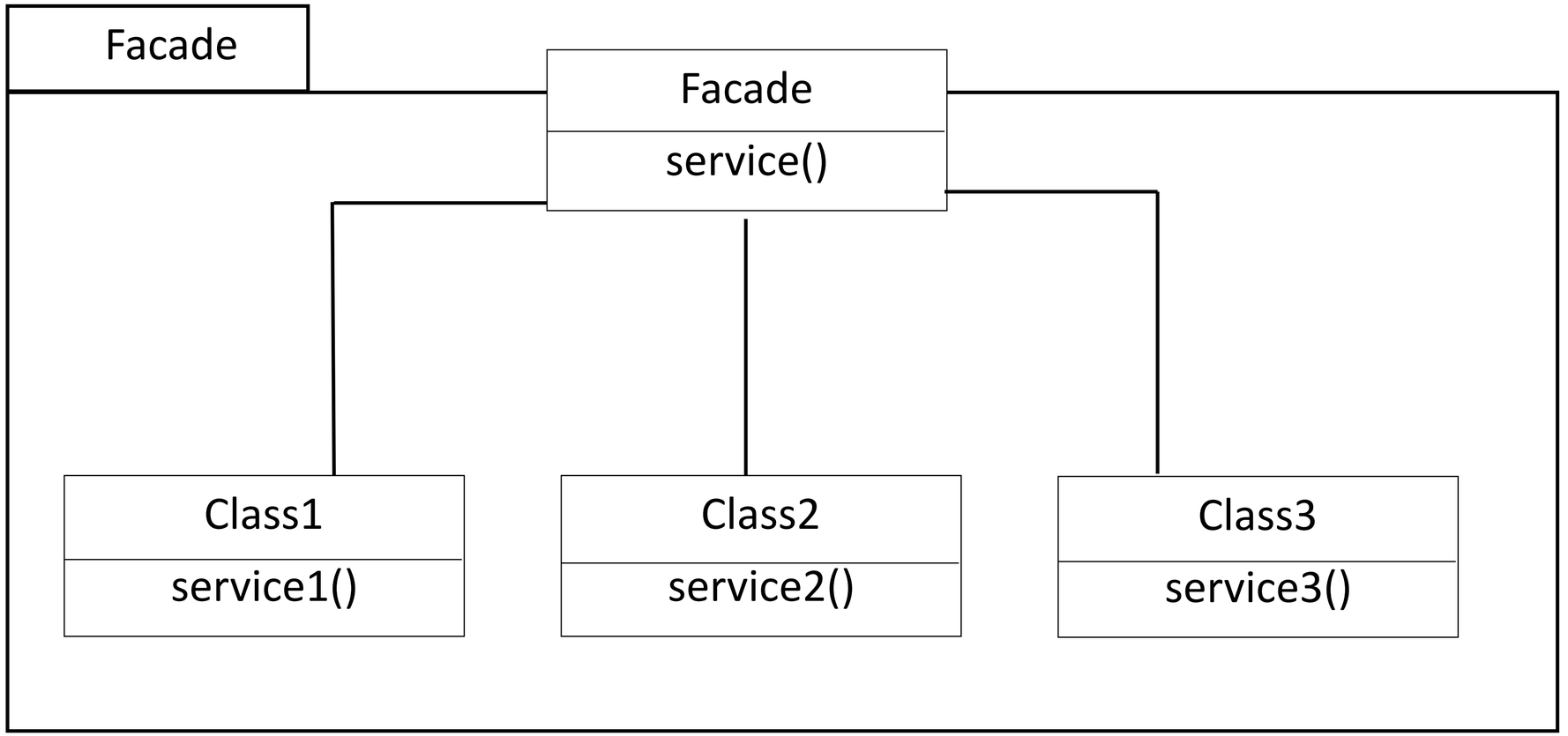
Example:

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.

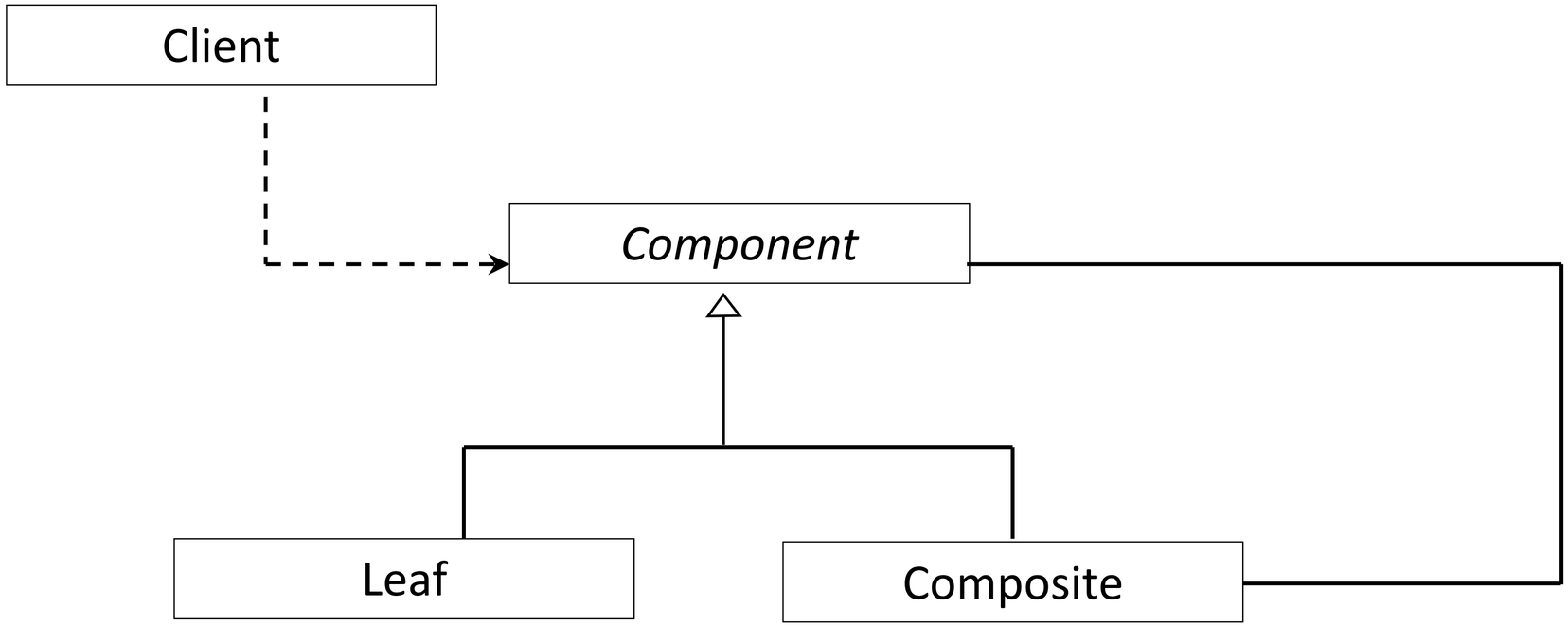
Composite: Representing Recursive Hierarchies

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.

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

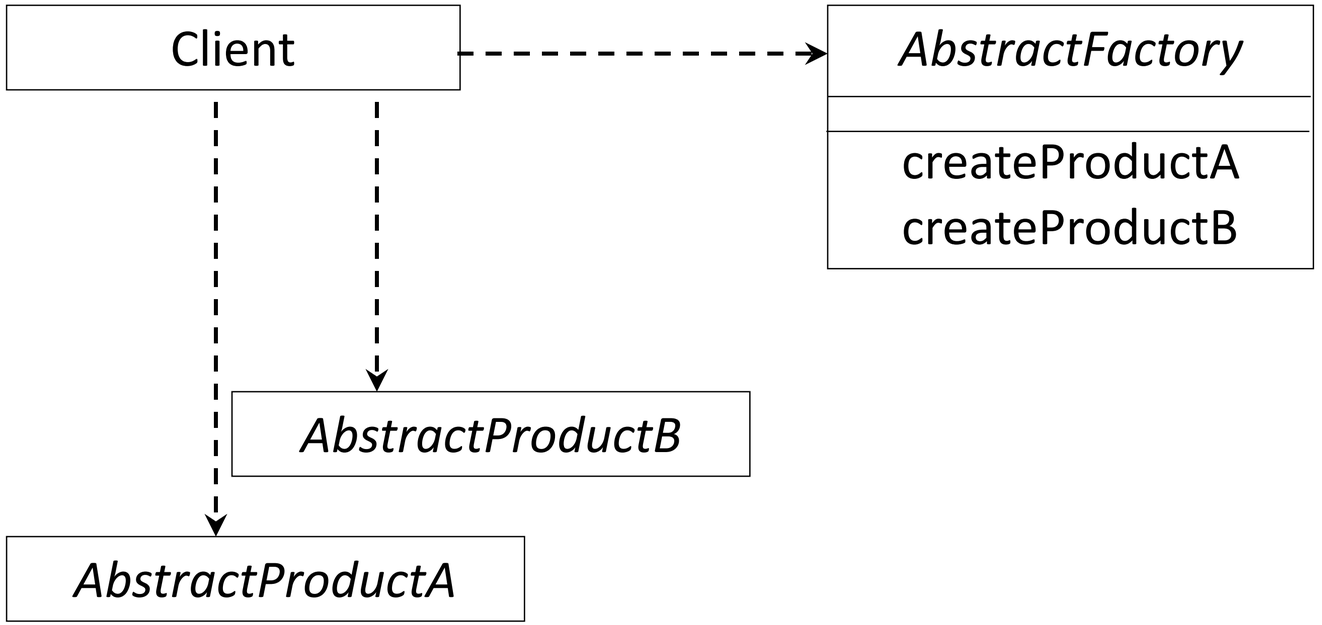
Abstract Factory: Encapsulating Platforms

Solution:

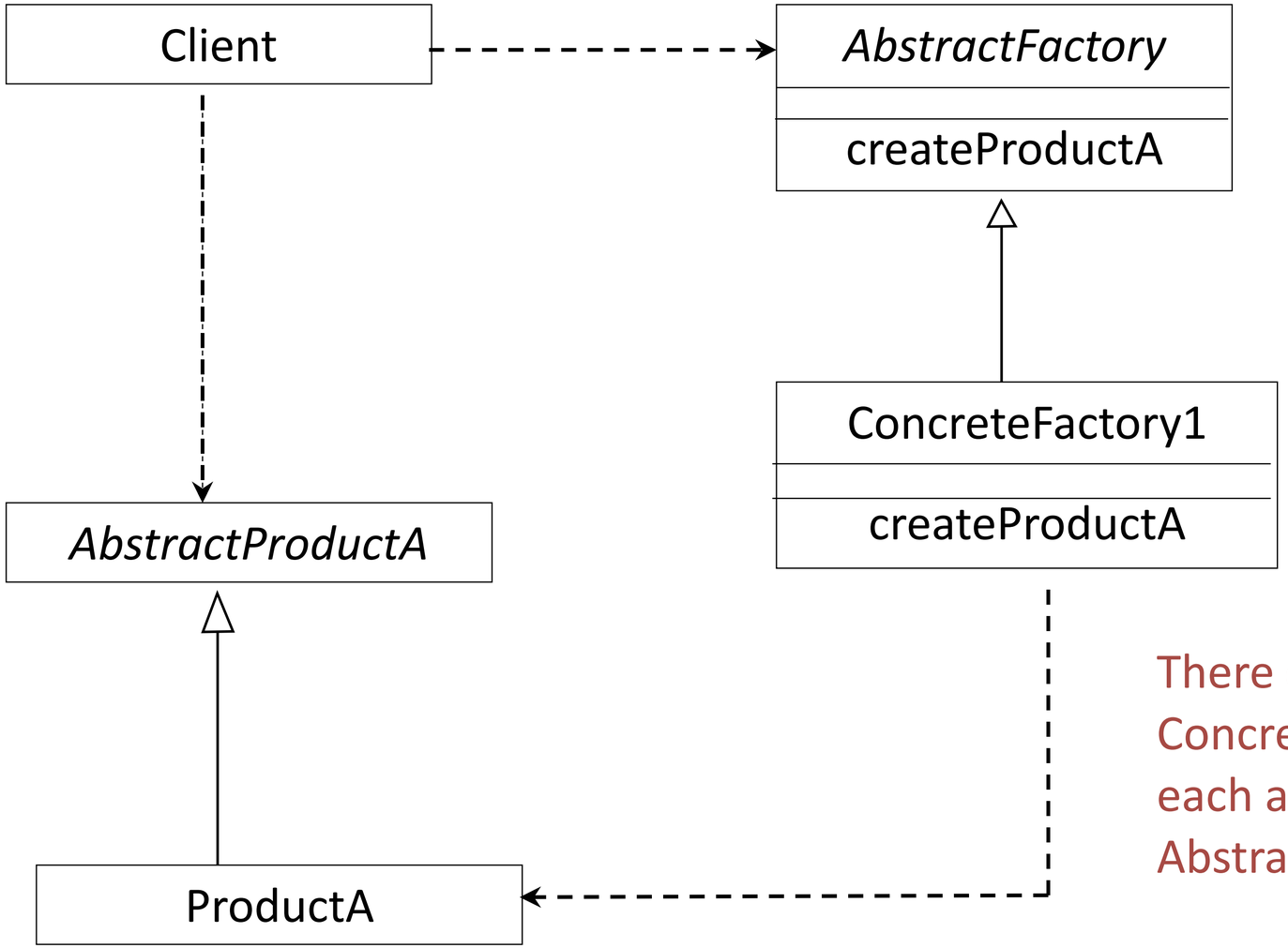
A platform (e.g., the application for a specific windowing system) is represented as a set of AbstractProducts, each representing a concept (e.g., button). An AbstractFactory class declares the operations for creating each individual product.

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

Abstract Factory: Class Diagram

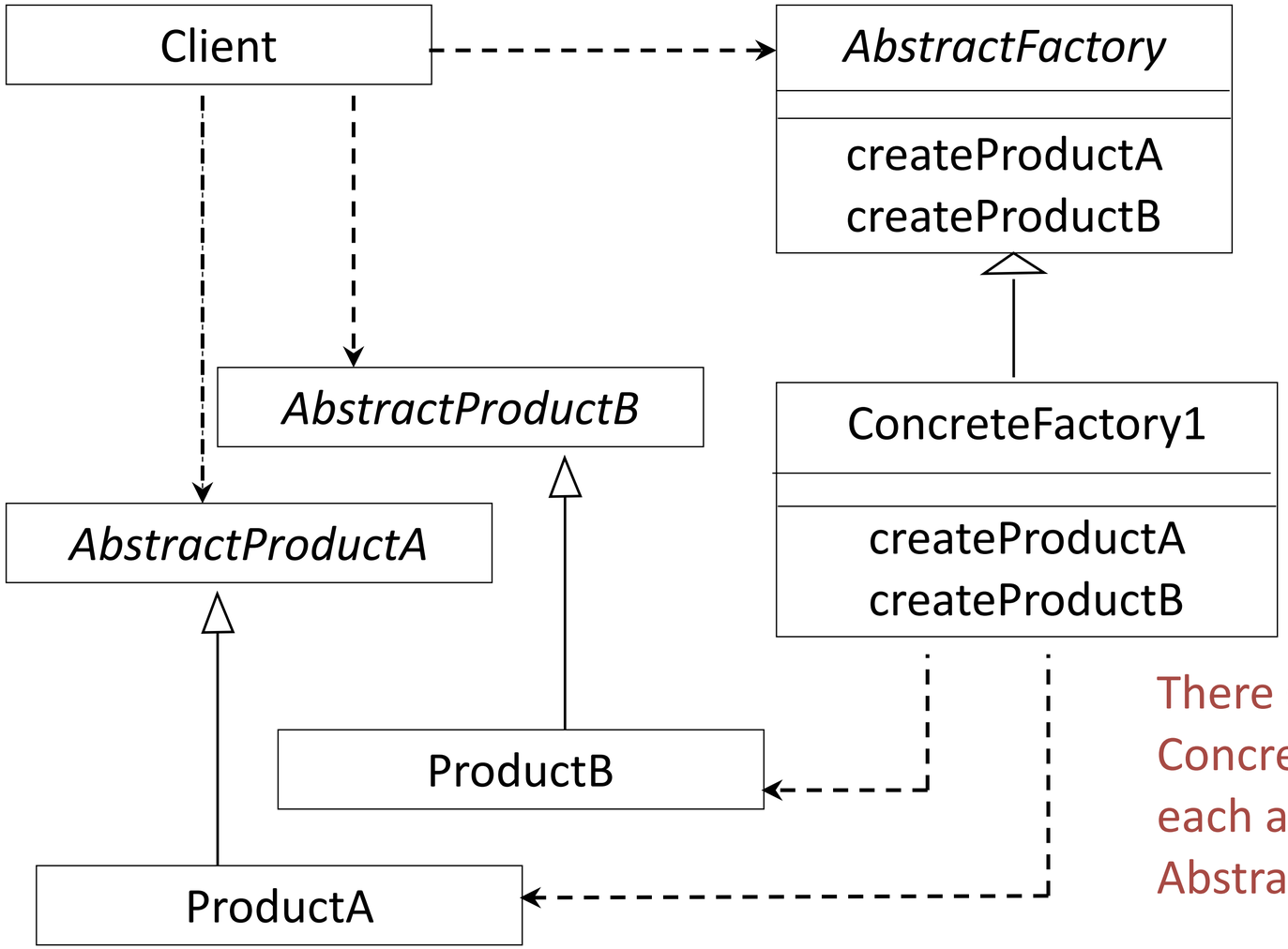


Abstract Factory: Class Diagram



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

Discussion

Discussion

See the interesting discussion in Wikipedia (December 16, 2016):

"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

Careful 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.

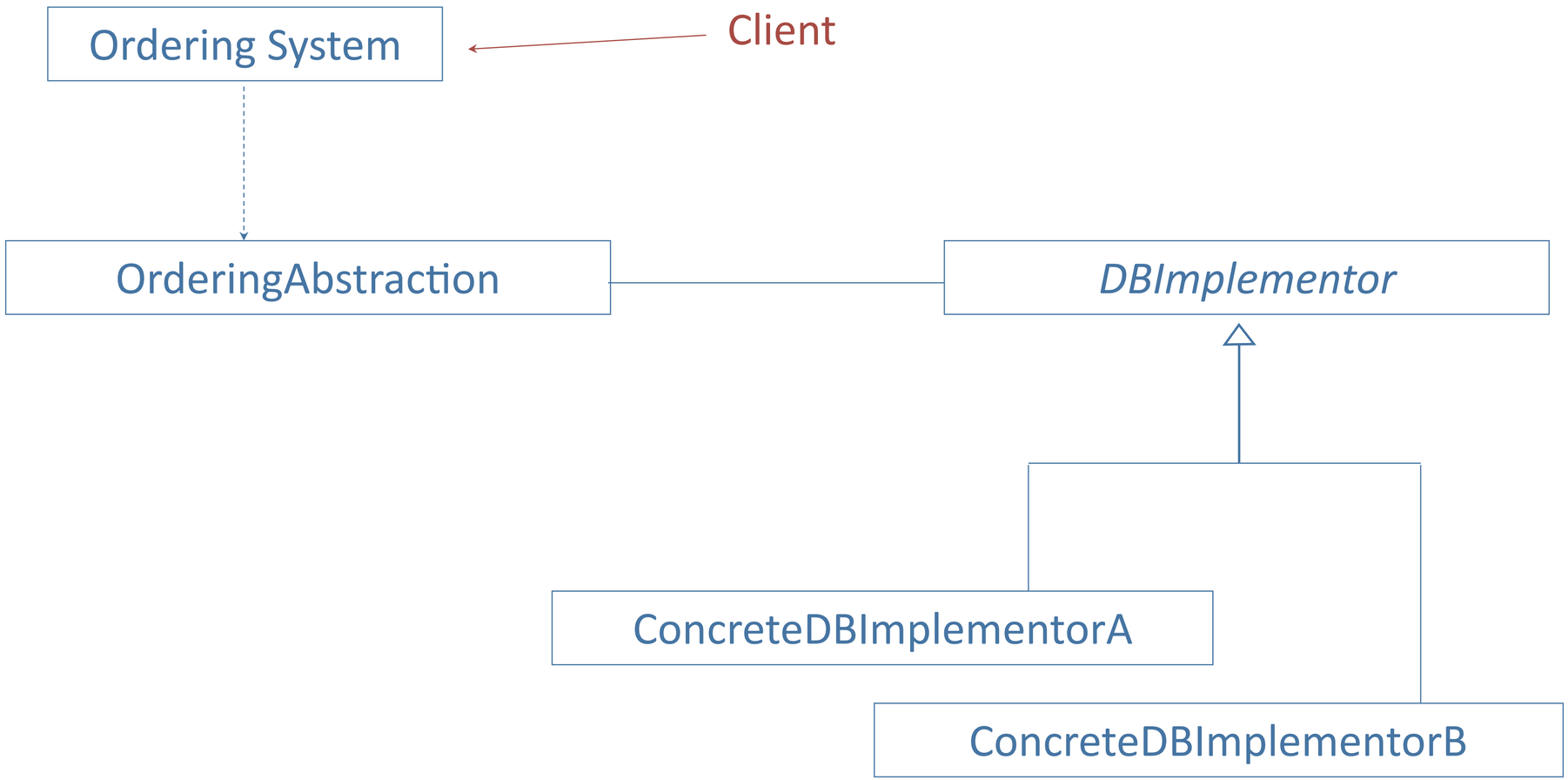
Bridge design pattern

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

Legacy Systems

Many data intensive systems, e.g., those used by banks, universities, etc. are legacy systems. They may have been developed forty years ago as batch processing, master file update systems and been continually modified.

- Recent modifications might include customer interfaces for the web, smartphones, etc.
- The systems will have migrated from computer to computer, across operating systems, to different database systems, etc.
- The organizations may have changed through mergers, etc.

Maintaining a coherent system architecture for such legacy systems is an enormous challenge, yet the complexity of building new systems is so great that it is rarely attempted.

Legacy Systems

The Worst Case

A large, complex system that was developed several decades ago:

- Widely used either within a big organization or by an unknown number of customers.
- All the developers have retired or left.
- No list of requirements. It is uncertain what functionality the system provides and who uses which functions.
- System and program documentation incomplete and not kept up to date.
- Written in out-of-date versions of programming languages using system software that is also out of date.
- Numerous patches over the years that have ignored the original system architecture and program design.
- Extensive code duplication and redundancy.
- The source code libraries and production binaries may be incompatible.

Legacy Requirements

Planning

In conjunction with the client develop a plan for rebuilding the system.

Requirements as seen by the customers and users

- Who are the users?
- What do they actually use the system for?
- Does the system have undocumented features that are important or bugs that users rely on?
- How many people use the fringe parts of the system? Where are they flexible?

Requirements as implied by the system design

- If there is any system documentation, what does it say about the requirements?
- Does the source code include any hints about the requirements?
- Is there code to support obsolete hardware or services? If so, does anybody still use them?

Legacy Code

Source code management

- Use a source code management system to establish a starting version of the source code and binaries that are built from this source code.
- Create a test environment so that the rebuilt system can be compared with the current system. Begin to collect test cases.
- Check the licenses for all vendor software.

Legacy Code

Rebuilding the software

An incremental software development process is often appropriate, with each increment released when completed.

The following tasks may be tackled in any appropriate order, based on the condition of the code. Usually the strategy will be to work on different parts of the system in a series of phases.

- Understand the original systems architecture and program design.
- Establish a component architecture, with defined interfaces, even if much of the code violates the architecture and needs adapters.
- Move to current versions of programming languages and systems software.
- If there are any subsystems that do not have source code, carry out a development cycle to create new code that implements the requirements.
- If there is duplicate code, replace with a single version.
- Remove redundant code and obsolete requirements.

Clean up as you go along.