Name	Adapter Design Pattern
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.
Solution	An `Adapter` class implements the `ClientInterface` expected by the client. The `Adapter` delegates requests from the client to the `LegacyClass` and performs any necessary conversion.
Consequences	• `Client` and `LegacyClass` work together without modification of neither `Client` nor `LegacyClass`. • `Adapter` works with `LegacyClass` and all of its subclasses. • A new `Adapter` needs to be written for each specialization (e.g., subclass) of `ClientInterface`.

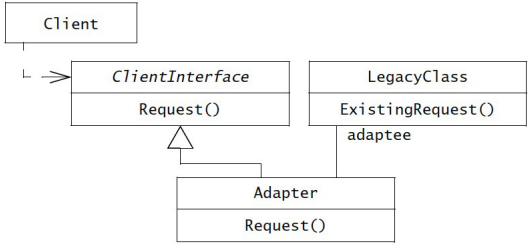
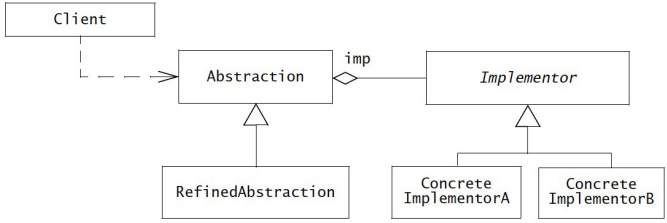


Figure 8-5 An example of design pattern, Adapter (adapted from [Gamma et al., 1994]).

Bruegge and Detoit, pg 319

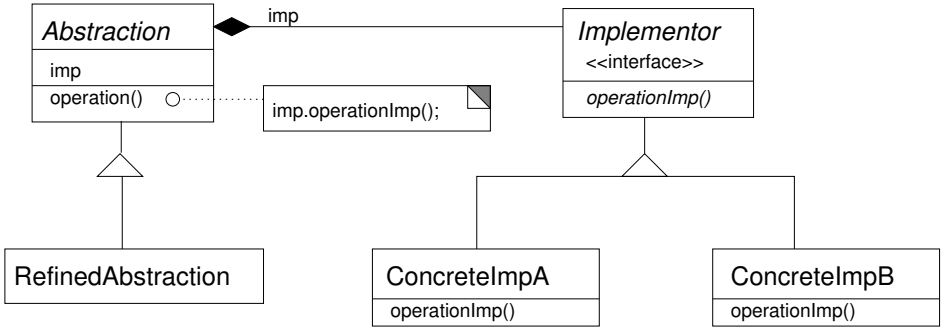
Name	Bridge Design Pattern
Problem description	Decouple an interface from an implementation so that implementations can be substituted, possibly at runtime.
Solution	The *Abstraction* class defines the interface visible to the client. The *Implementor* is an abstract class that defines the lower-level methods available to *Abstraction*. An *Abstraction* instance maintains a reference to its corresponding *Implementor* instance. *Abstraction* and *Implementor* can be refined independently.



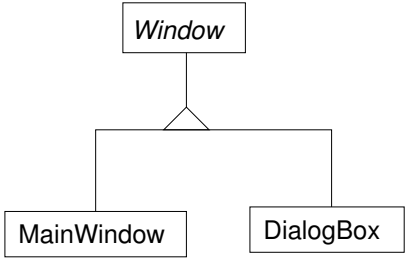
Consequences	• `Client` is shielded from abstract and concrete implementations. • Interfaces and implementations can be refined independently.
Example	• Testing different implementations of the same interface (Section 8.4.1).
Related concept	• The `Adapter` pattern (Section A.2) fills the gap between two interfaces.

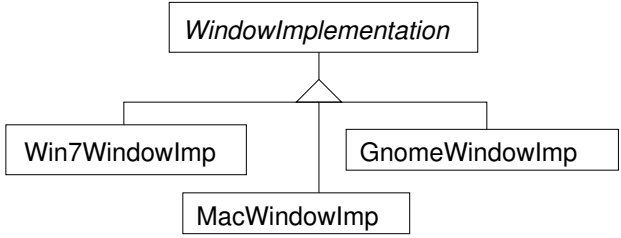
Figure A-3 The Bridge design pattern (adapted from [Gamma et al., 1994]).

Bruegge and Detoit, pg 712



Based on Gamma et al, pg 152



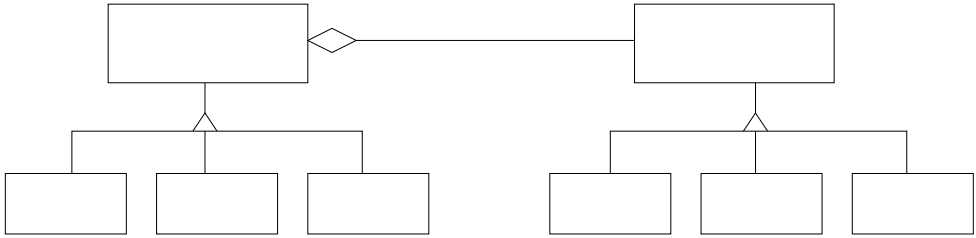


Based on Gamma et al, pg 151

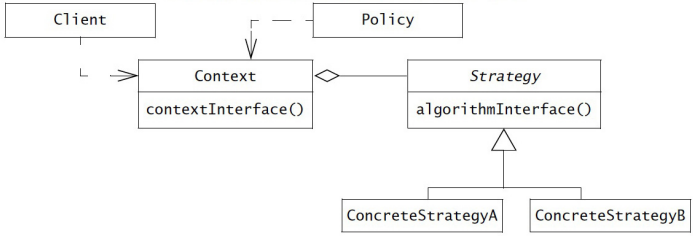
Q1. Referring to `BridgeDemo`, develop a class diagram that shows a solution to the given situation by using the bridge design pattern.

Q2. Determine whether the solution in Q1 follows the Liskov Substitution Principle or not.

Q3. Consider again `BridgeDemo` and its output. Referring to the definition of the bridge design pattern, discuss with your team members the role of each class. Explain how the bridge design pattern enhances the program's extensibility



Name	Strategy Design Pattern
Problem description	Decouple a policy-deciding class from a set of mechanisms so that different mechanisms can be changed transparently from a client.
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 `Strategy` class describes the interface that is common to all mechanisms that `Context` can use. The `Policy` class creates a `ConcreteStrategy` object and configures the `Context` to use it.



Consequences	• `ConcreteStrategies` can be substituted transparently from `Context`. • `Policy` decides which `Strategy` is best, given the current circumstances (e.g., speed vs. space trade-off). • New algorithms can be added without modifying `Context` or `Client`.
Example	• Network switching in mobile applications (Section 8.4.3): A mobile application needs to deal with a variety of network access protocols (phone, wireless, LAN), depending on the context of the user (location, communication costs, etc.). To decouple the policy from selecting a network from the network interface, we encapsulate the network access protocol implementations with a `Strategy` pattern.
Related concept	`Adaptor` pattern (Section A.2) and `Bridge` pattern (Section A.3).

Figure A-9 The Strategy design pattern (adapted from [Gamma et al., 1994]).

Assume that your program needs to use different types of networks depending on the location: LAN, SAN, and wireless networks are available for the program, and `NTInterface` has been developed to facilitate working with these networks.

Q4. Refer to the statements in `NTDriver` and consider the diagram provided. Complete the diagram by adding the missing classes according to the strategy design pattern. Determine the number of additional classes needed, and describe the main functionality of each class.

Q5. Assume that a client (such as your program) has the statements in `NTDriver`. Explain how these statements are used to establish an initial network connection. You may refer to the code in Q6 and Q8 for context.

Q6. Complete the code in class `NTConnection`

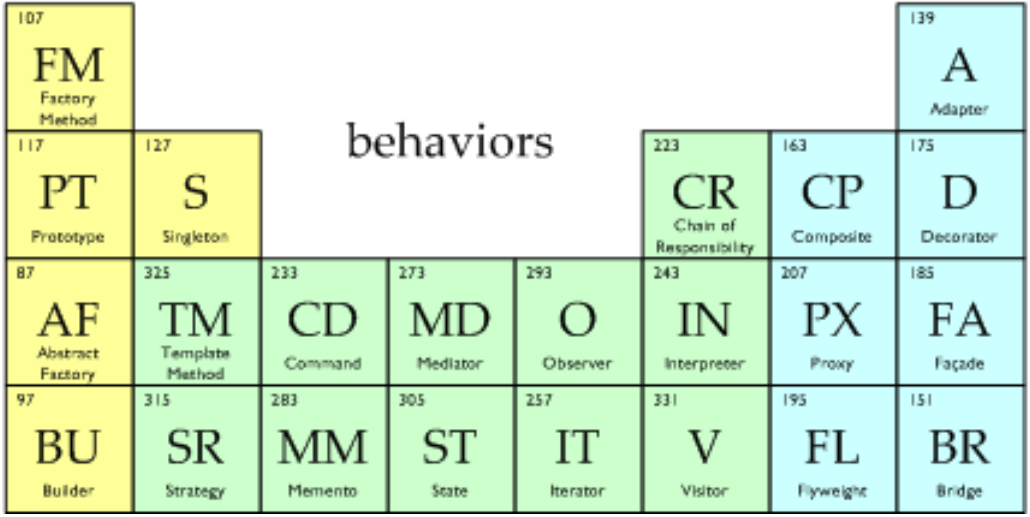
Q7. Explain the functionality of `setNTInterface()` in the code from Q6. Which class in the diagram other than the client program, may call this method?

Q8. Consider the provided pseudo-code for class `NTSelector`. Fill in the blanks, including the question in the comments.

Q9. Does the client program need to be aware of network changes during running? Summarize the benefits of using the strategy pattern.

origins

structures



http://www.vincehuston.org/dp/patterns