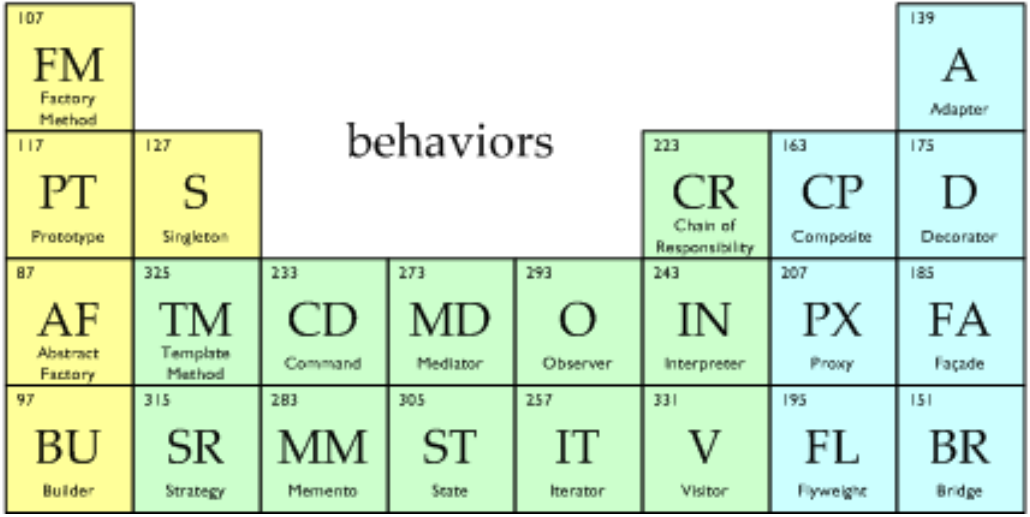
origins

structures




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

Q1. Based on the information, and the code above, which methods are overridden, and which methods are inherited? Draw a class diagram that shows the relationship between HashMap and MySet.

Q2. Given the following driver for MySet, present the output from the driver.

Q3. Suppose that you want to publish your MySet so that other programmers can use your class. Discuss potential problems in using your MySet class. What causes such problems?

Q4. Assume that you have decided to update your MySet so that it now utilizes ArrayList instead of HashMap. In MySetDriver, what statements will cause errors when you replace your set object with a MySetList object?

Q5. Discuss with your team members whether the use of inheritance for this situation was a good decision or not. Suppose, however, that you still want to reuse the code of HashMap for your set class. What would be an alternative solution?

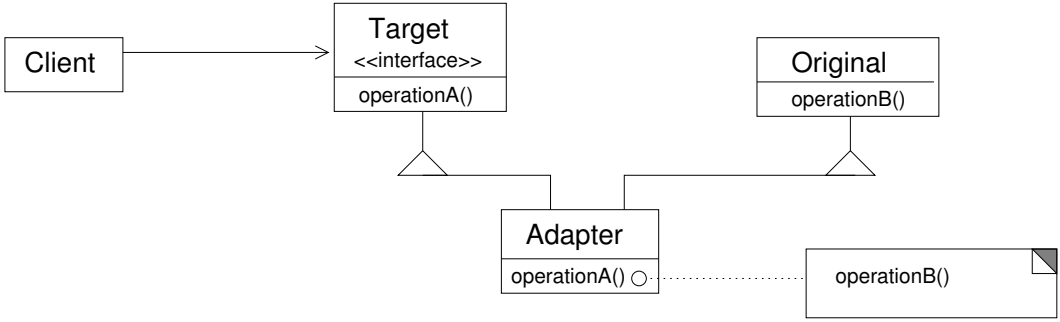
Q6. Assume that your client program requires a set object that has the following operations: `boolean add(char)`, `boolean remove(char)`, `void printSet()`. Suppose again that you want to reuse HashMap and bridge the gap between your program and Java HashMap. Refer to the client below and the adaptor design pattern. What should be implemented to achieve this solution? Present your solution in a class diagram. Specify necessary classes and methods in your diagram.

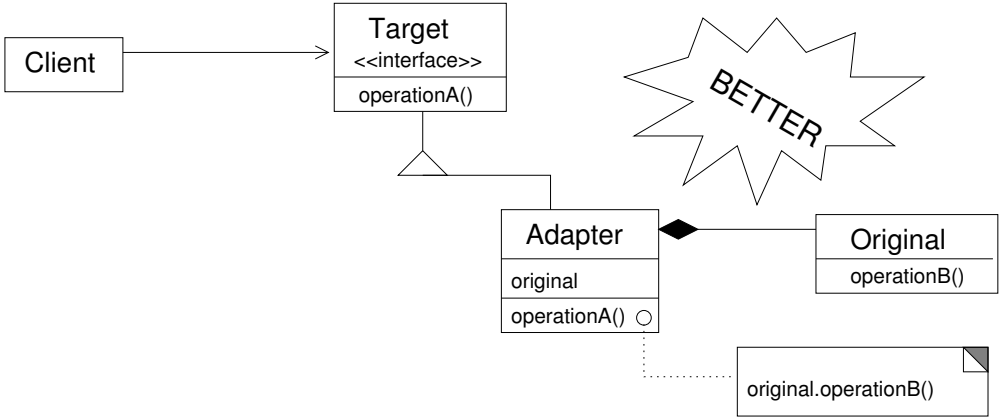
Q7. Refer to MyHashMap below, and answer the questions:

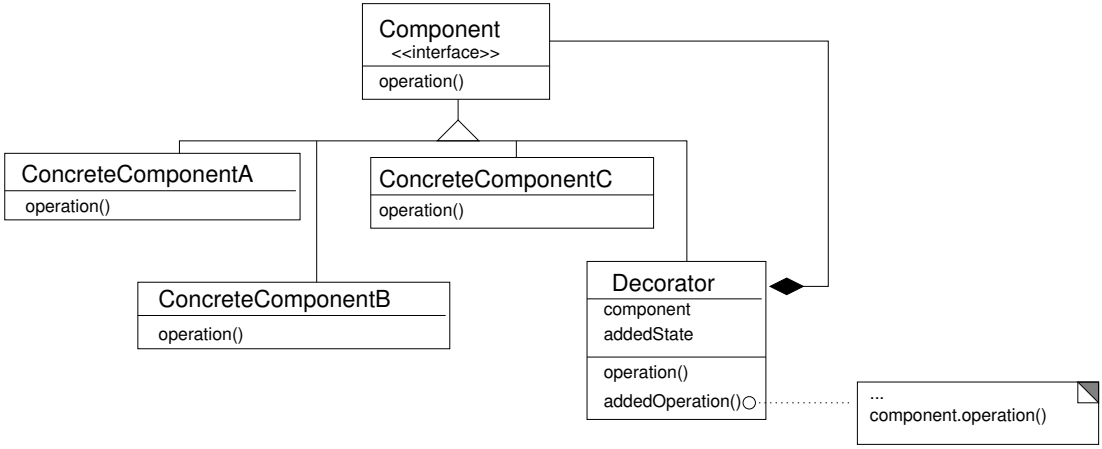
(a) Refer to your Adapter class in Q6. Can you replace the HashMap object with an object of MyHashMap? Why, or why not?

(b) Will the above SetInterfaceClient still be running for your program?

(c) Add the MyHashMap class to your diagram in Q6. In your diagram, indicate where the Liskov Substitution Principle has been applied.


Client
Target
<<interface>>
operationA()
Original
operationB()
Adapter
operationA()
operationB()


Client
Target
<<interface>>
operationA()
BETTER
Adapter
original
operationA()
Original
operationB()
original.operationB()


Component
<<interface>>
operation()
ConcreteComponentA
operation()
ConcreteComponentC
operation()
ConcreteComponentB
operation()
Decorator
component
addedState
operation()
addedOperation()
...
component.operation()

| *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`.<br>• `Adapter` works with `LegacyClass` and all of its subclasses.<br>• A new `Adapter` needs to be written for each specialization (e.g., subclass) of `ClientInterface`. |

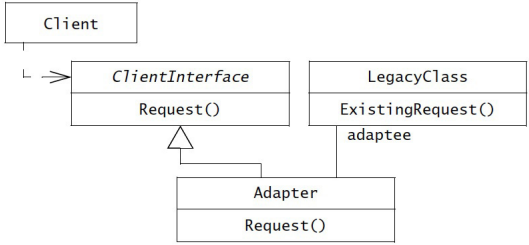
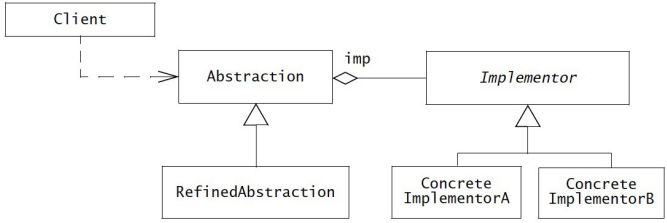



**Figure 8-5** An example of design pattern, Adapter (adapted from [Gamma et al., 1994]).



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. |

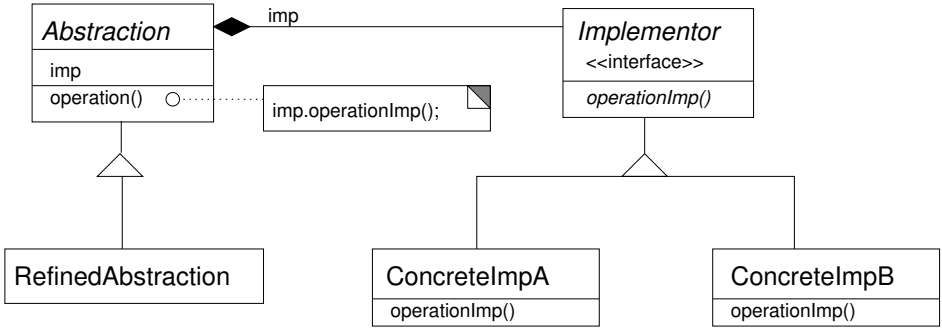



| | |
|---|---|
| **Consequences** | • `Client` is shielded from abstract and concrete implementations.<br>• Interfaces and implementations can be refined independently. |
| **Example** | • Testing different implementations of the same interface (Section 8.4.1). |
| **Related concept** | • The `Adapter`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

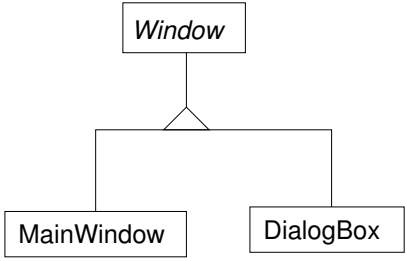
**Figure A-3** The Bridge design pattern (adapted from [Gamma et al., 1994]).

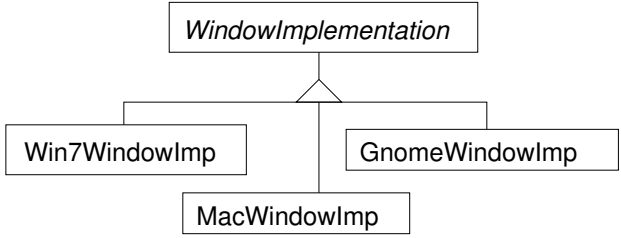


Bruegge and Detoit, pg 712




Based on Gamma et al, pg 152




Based on Gamma et al, pg 151