BRIDGE PATTERN

Toni Linares

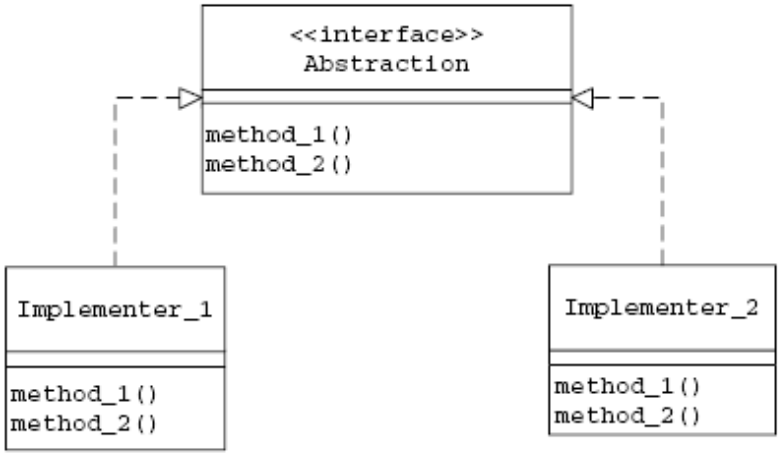
DESCRIPTION

- Bridge pattern promotes the separation of an abstraction’s interface from its implementation

WHAT'S ABSTRACTION?

- In general, the term *abstraction* refers to the process of identifying the set of attributes and behavior of an object that is specific to a particular usage.
- This specific view of an object can be designed as a separate object omitting irrelevant attributes and behavior. The resulting object itself can be referred to as an *abstraction*.
- A given abstraction may have one or more implementations for its methods (behavior).
- In terms of implementation, an abstraction can be designed as an interface with one or more concrete implementers

Abstraction as an interface with a set of Concrete Implementers



LIMITATIONS OF THIS APPROACH

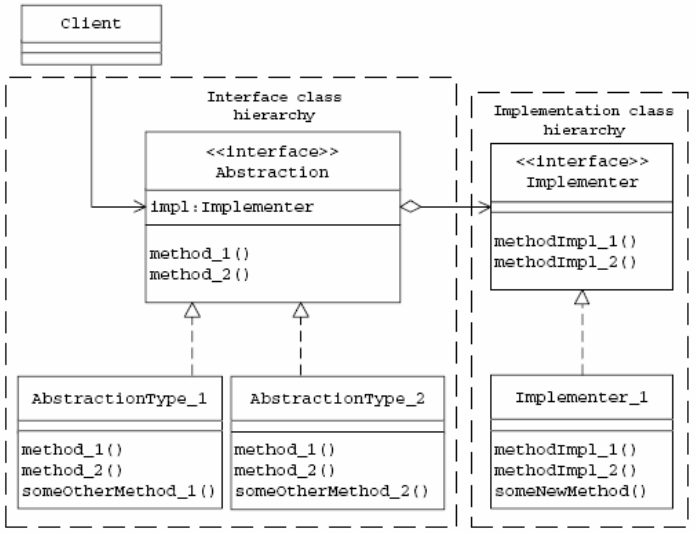
- When there is a need to subclass the hierarchy for some other reason, it could lead to an exponential number of subclasses and soon we will have an exploding class hierarchy.

- 2. Both the abstraction interface and its implementation are closely tied together and hence they cannot be independently varied without affecting each other.

USIGN BRIDGE PATTERN

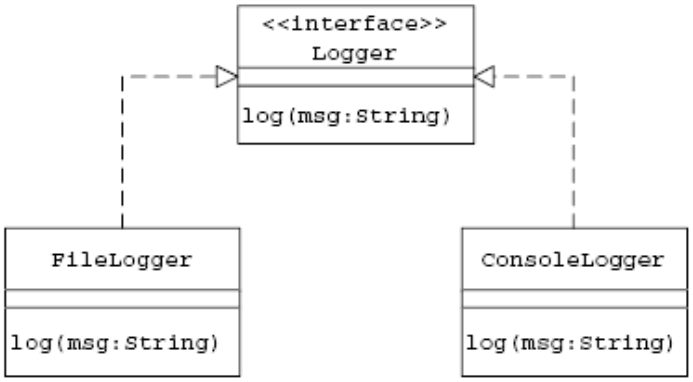
- Using the Bridge pattern, a more efficient and manageable design of an abstraction can be achieved.

- The design of an abstraction using the Bridge pattern separates its interfaces from implementations. Applying the Bridge pattern, both the interfaces and the implementations of an abstraction can be put into separate class hierarchies.



EXAMPLE : MESSAGE LOGGIN FUNCTIONALITY

- Because many different objects within an application may need the ability to log messages, the logging feature may be put into a separate class.
- The resulting class is an abstraction of the message logging functionality.
- A message can be logged to different types of destinations such as a file, console and others.
- Depending on the destination type, a different implementation of the logger abstraction is needed.



DESIGN PROBLEMS

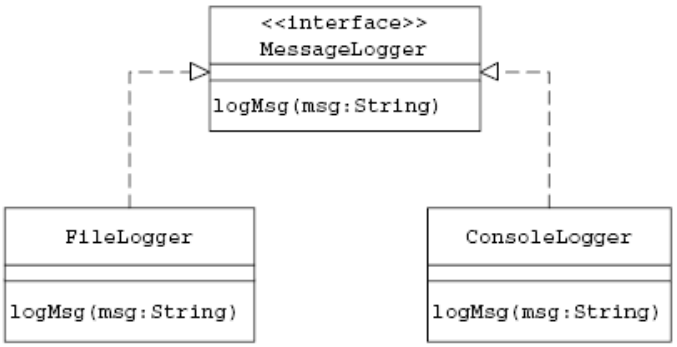
- After this design is implemented, let us suppose that an application object needs to log messages in a different format (e.g., in an encrypted form).
- The existing messaging logging functionality design is not sufficient without either:
  - Modifying different implementers
  - Extending the entire class hierarchy
- Having to modify the existing code in order to extend the functionality is not advisable and **violates the basic object-oriented open-closed principle**.
- Subclassing the class hierarchy for every different type of message format is also not recommended as it could result in an exponential number of subclasses and soon there will be an exploding class hierarchy.

USING BRIDGE PATTERN

- The Bridge pattern can be used in this case to provide the ability to add new message formats and new types of implementations to the logger abstraction.
- The Bridge pattern separates the interface and implementations into two separate class hierarchies so that they both can be modified without affecting each other.

ABSTRACTION IMPLEMENTATION DESIGN

- Implementers of the logger abstraction need to provide the actual implementation required to log messages to different destination types
  (FileLogger and ConsoleLogger) .
- These abstraction implementers can be designed as two concrete implementers of a common **MessageLogger** (Java)interface.



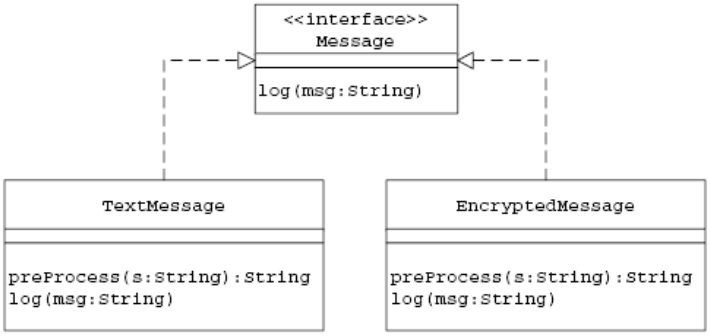
JAVA CODE

```
public interface MessageLogger {
  public void logMsg(String msg);
}
```

```
public class FileLogger implements MessageLogger {
  public void logMsg(String msg) {
    FileUtil futil = new FileUtil();
    futil.writeToFile('log.txt",msg, true, true);
  }
}
public class ConsoleLogger implements MessageLogger {
  public void logMsg(String msg) {
    System.out.println(msg);
  }
}
```

ABSTRACTION INTERFACE DESIGN

- The interface for the logger abstraction can be designed in the form of a set of classes representing different types of messages that a client object would like to log.
- These classes can be designed as implementers of a common Message (Java)interface.
- The Message interface declares a method log(String msg), which can be used by different client objects to log messages.
- **TextMessage** and **EncryptedMessage** abstraction interface classes can be designed as concrete implementers of a common **Message (Java)interface**.



JAVA CODE

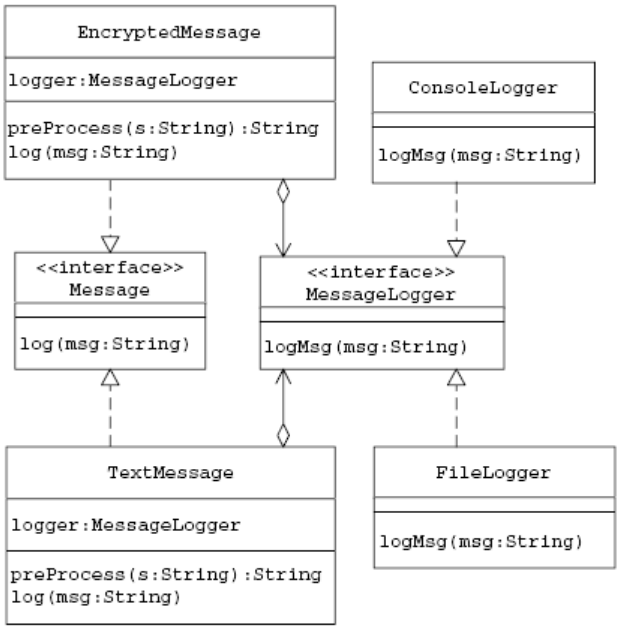
```
public interface Message {
  public void log(String msg);
}
```

```
public class TextMessage implements Message {
  private MessageLogger logger;
  public TextMessage(MessageLogger l) {
    logger = l;
  }
  public void log(String msg) {
    String str = preProcess(msg);
    logger.logMsg(str);
  }
  private String preProcess(String msg) {
    return msg;
  };
}
public class EncryptedMessage implements Message {
  private MessageLogger logger;
  public EncryptedMessage(MessageLogger l) {
    logger = l;
  }
  public void log(String msg) {
    String str = preProcess(msg);
    logger.logMsg(str);
  }
  private String preProcess(String msg) {
    msg = msg.substring(msg.length() - 1) +
          msg.substring(0, msg.length() - 1);
    return msg;
  };
}
```

JAVA CODE

```
public class Client {
  public static void main(String[] args) {
    //Create an appropriate implementer object
    MessageLogger logger = new FileLogger();
    //Choose required interface object and
    //configure it with the implementer object
    Message msg = new EncryptedMessage(logger);
    msg.log('Test Message");
  }
}
```

FINAL DESIGN



PROCESS FLOW EXAMPLE

