Factory Method

Intent

- Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

Also Known As

- Virtual Constructor

Motivation (Partha Kuchana)

- In general, all subclasses in a class hierarchy inherit the methods implemented by the parent class.
- A subclass may override the parent class implementation to offer a different type of functionality for the same method.
- When an application object is aware of the exact functionality it needs, it can directly instantiate the class from the class hierarchy that offers the required functionality.
- An application object needs to implement the class selection criteria to instantiate an appropriate class from the hierarchy to access its services.

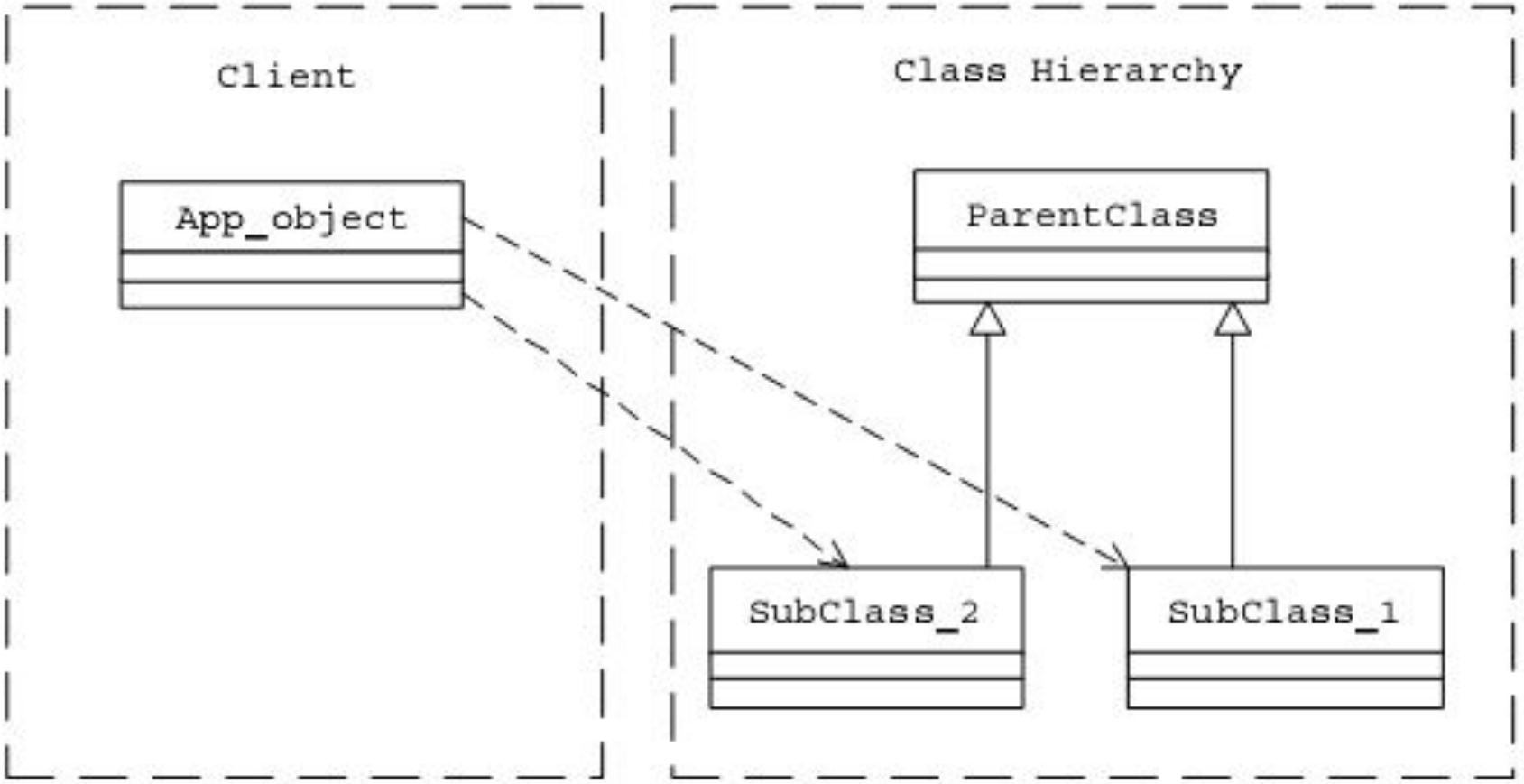
Design without Factory Method

- Because every application object that intends to use the services offered by the class hierarchy needs to implement the class selection criteria, it results in a high degree of coupling between an application object and the service provider class hierarchy.
- Whenever the class selection criteria change, every application object that uses the class hierarchy must undergo a corresponding change.
- Because class selection criteria needs to take all the factors that could affect the selection process into account, the implementation of an application object could contain inelegant conditional statements.

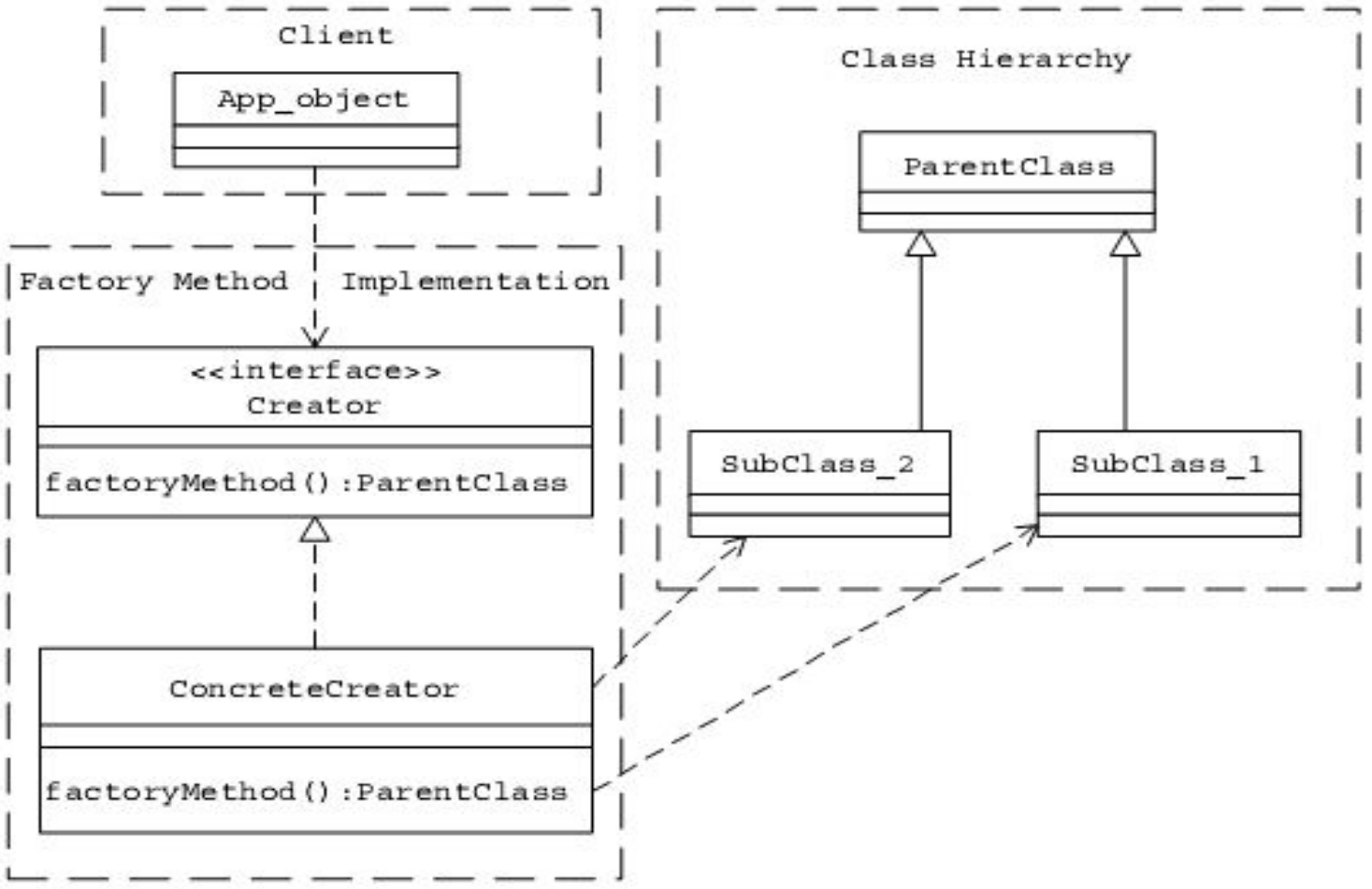
A Factory method

- Selects an appropriate class from a class hierarchy based on the application context and other influencing factors.
- Instantiates the selected class and returns it as an instance of the parent class type.

Client Object Directly Accessing a Service Provider Class Hierarchy



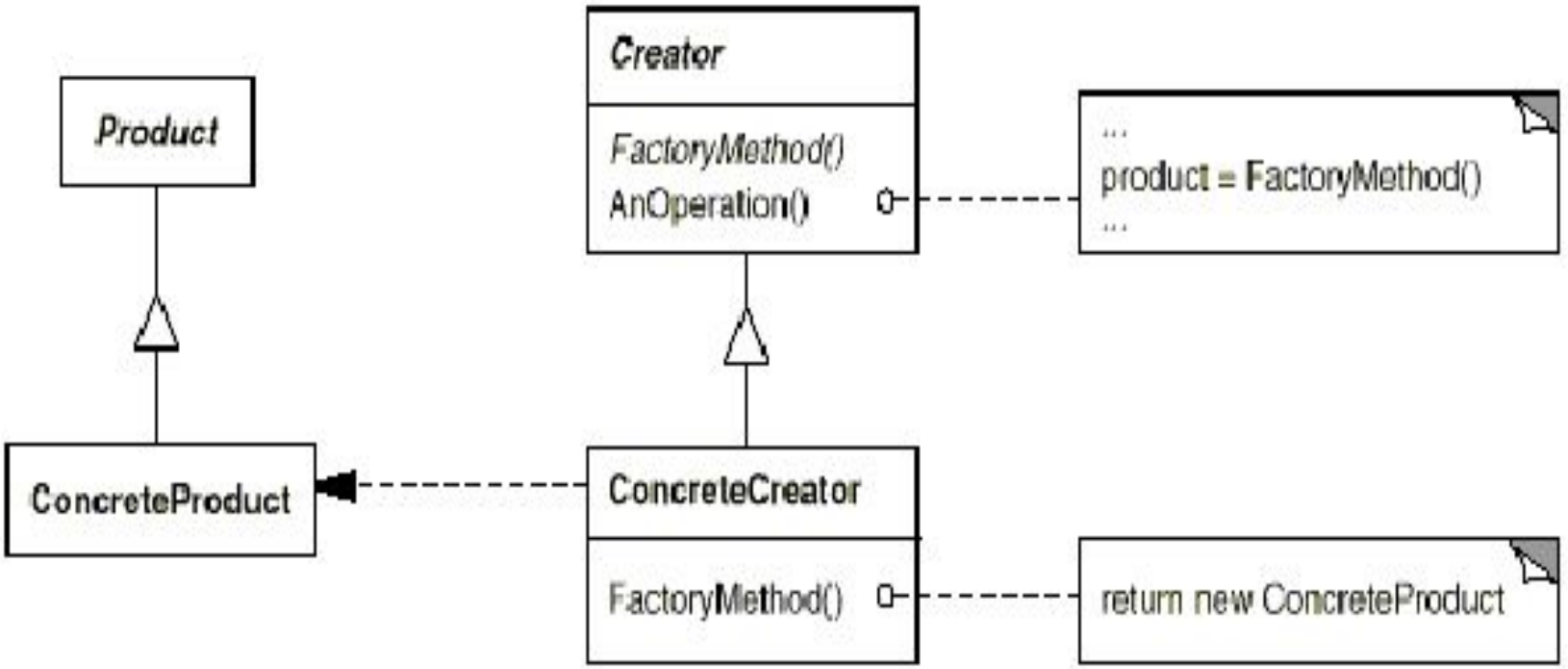
A Client Object Accessing a Service Provider Class Hierarchy Using a Factory Method



2 ways of Implementation

- One of the simplest ways of designing a factory method is to create an abstract class or an interface that just declares the factory method. Different subclasses(or implementer classes in the case of an interface) can be designed to implement the factory method in its entirety.
- Another strategy is to create a concrete creator class with default implementation for the factory method in it. Different subclasses of this concrete class can override the factory method to implement specialized class selection criteria.

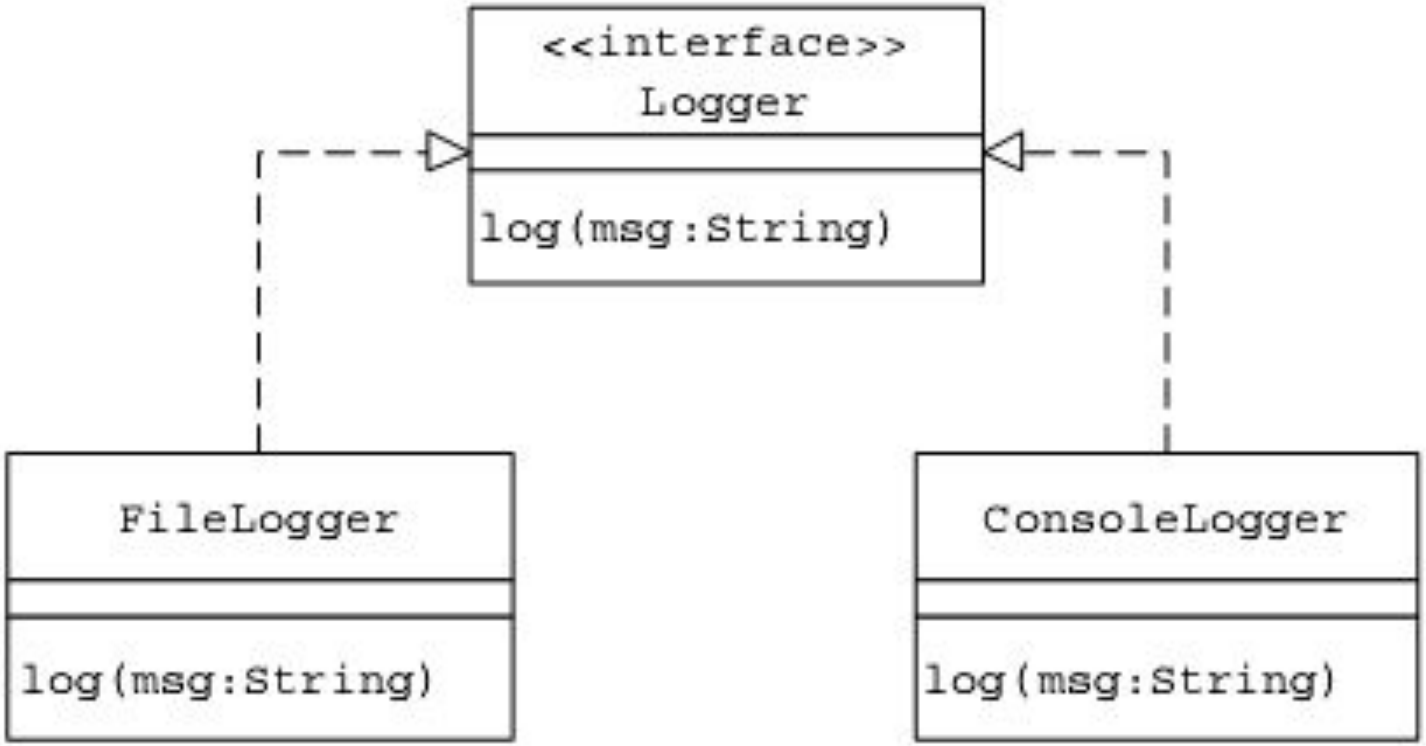
Structure


Product
Creator
FactoryMethod()
AnOperation()
...
product = FactoryMethod()
...
ConcreteProduct
ConcreteCreator
FactoryMethod()
return new ConcreteProduct

Example

- Let us design the functionality to log messages in an application.
- Because the message logging functionality could be needed by many different clients, it would be a good idea to keep the actual message logging functionality inside a common utility class so that client objects do not have to repeat these details.

Message Logging Utility Class Hierarchy



Logger Implementers

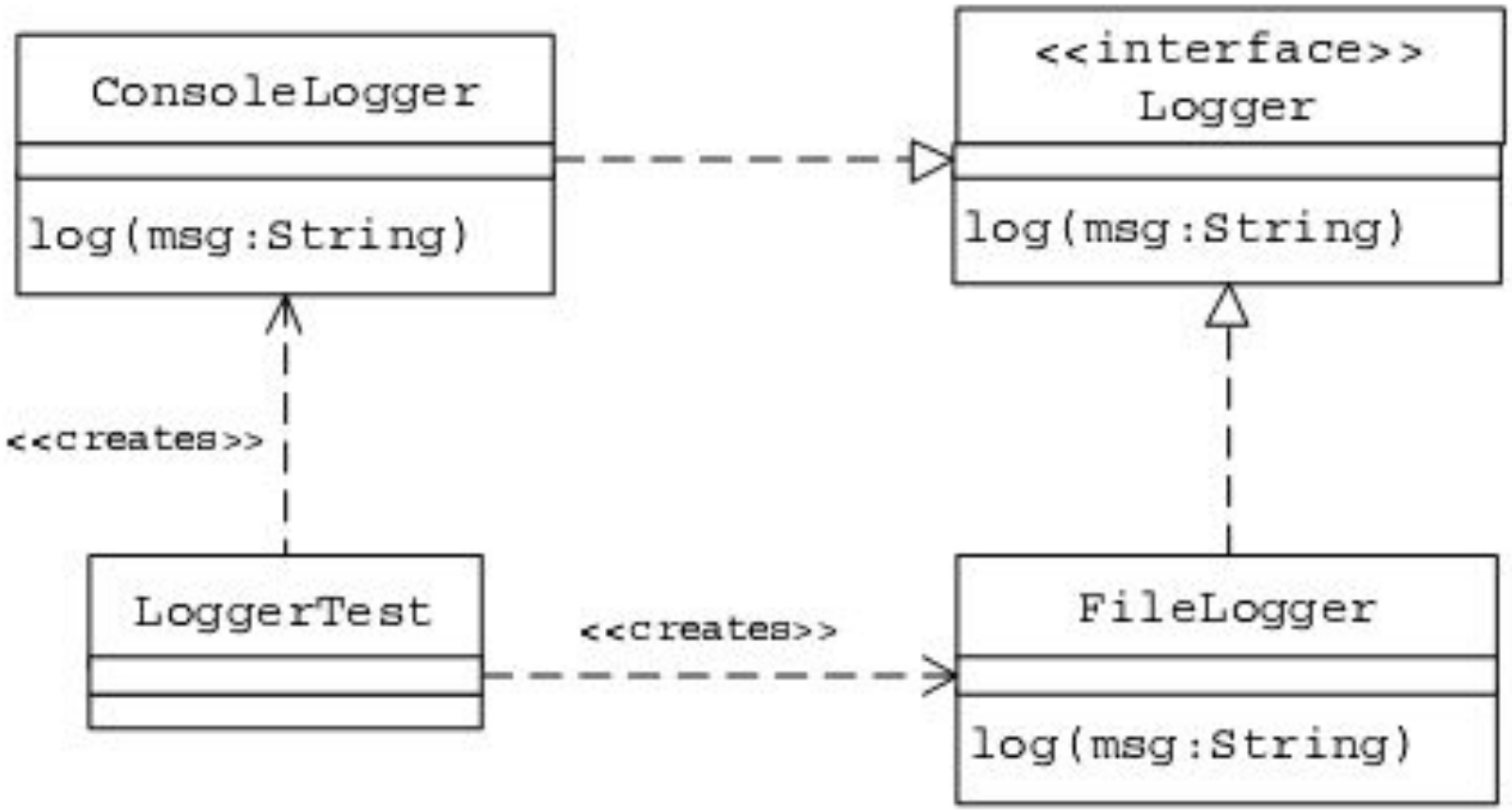
Implementer	Functionality
FileLogger	Stores incoming messages to a log file
ConsoleLogger	Displays incoming messages on the screen

Implementation details

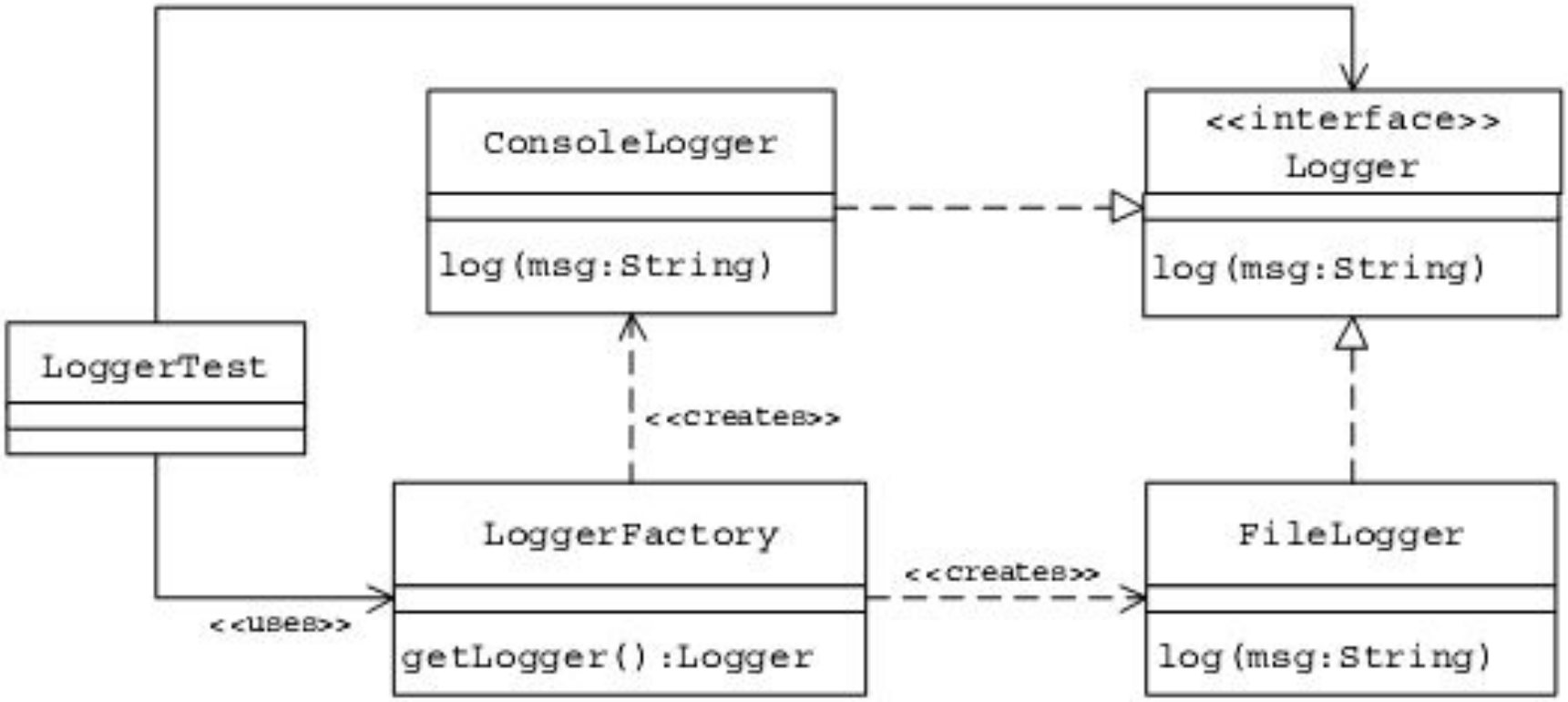
- Let us define a Java interface Logger that declares the interface to be used by the client objects to log messages.
- Each of the Logger implementer classes offers the respective functionality inside the log method declared by the Logger interface.
- Consider an application object LoggerTest that intends to use the services provided by the Logger implementers. Suppose that the overall application message logging configuration can be specified using the logger.properties property file.
- Sample logger.properties file contents

  FileLogging=OFF

Client LoggerTest Accessing Logger Implementers Directly



The Client LoggerTest Accessing the Logger Class Hierarchy after the Factory Method Pattern Is Applied



Sample Code

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

Sample Code

```
public class LoggerFactory {
  public boolean isFileLoggingEnabled() {
    Properties p = new Properties();
   try {

   p.load(ClassLoader.getSystemResourceAsStream("Logg
   er.properties"));
    String fileLoggingValue = p.getProperty("FileLogging");
   if (fileLoggingValue.equalsIgnoreCase("ON") == true)
     return true;
   else
     return false;
   } catch (IOException e) {
   return false;   }
}
```

Sample Code

```
//Factory Method
public Logger getLogger() {
   if (isFileLoggingEnabled()) {
     return new FileLogger();
   } else {
     return new ConsoleLogger();       }    }
public class LoggerTest {
   public static void main(String[] args) {
     LoggerFactory factory = new LoggerFactory();
     Logger logger = factory.getLogger();
     logger.log("A Message to Log");
   }}
```

Self Study

- Add a new logger DBLogger that logs messages to a database.
- Create a subclass of the LoggerFactory class and override the getLogger implementation to implement a different class selection criterion.

Message Flow When a Client Uses the LoggerFactory to Create an Appropriate Logger to Log a Message