Creational Design Patterns

Mohsen Afsharchi

Definition

A design pattern is a documented best practice or core of a solution that has been applied successfully in multiple environments to solve a problem that recurs in a specific set of situations.

Architect Christopher Alexander describes a pattern as "a recurring solution to a common problem in a given context and system of forces." In his definition, the term *context* refers to the set of conditions/situations in which a given pattern is applicable and the term *system of forces* refers to the set of constraints that occur in the specific context.

More about Patterns

- A design pattern is an effective means to convey/communicate what has been learned about high-quality designs. The result is:
  - A shared language for communicating the experience gained in dealing with these recurring problems and their solutions.
  - A common vocabulary of system design elements for problem solving discussions. A means of reusing and building upon the acquired insight resulting in an improvement in the software quality in terms of its maintainability and reusability.
- A design pattern is not an invention. A design pattern is rather a documented expression of the best way of solving a problem that is observed or discovered during the study or construction of numerous software systems.

More about Patterns

- Design patterns are not theoretical constructs. A design pattern can be seen as an encapsulation of a reusable solution that has been applied successfully to solve a common design problem.
- Though design patterns refer to the best known ways of solving problems, not all best practices in problem resolution are considered as patterns. A best practice must satisfy the Rule of Three to be treated as a design pattern. The Rule of Three states that a given solution must be verified to be a recurring phenomenon, preferably in at least three existing systems.

Creational Patterns

- Deal with one of the most commonly performed tasks in an OO application, the creation of objects.
- Support a uniform, simple, and controlled mechanism to create objects.
- Allow the encapsulation of the details about what classes are instantiated and how these instances are created.
- Encourage the use of interfaces, which reduces coupling.

Creational Patterns

Factory Method	When a client object does not know which class to instantiate, it can make use of the factory method to create an instance of an appropriate class from a class hierarchy or a family of related classes. The factory method may be designed as part of the client itself or in a separate class. The class that contains the factory method or any of its subclasses decides on which class to select and how to instantiate it.
Singleton	Provides a controlled object creation mechanism to ensure that only one instance of a given class exists.
Abstract Factory	Allows the creation of an instance of a class from a suite of related classes without having a client object to specify the actual concrete class to be instantiated.
Prototype	Provides a simpler way of creating an object by cloning it from an existing (prototype) object.
Builder	Allows the creation of a complex object by providing the information on only its type and content, keeping the details of the object creation transparent to the client. This allows the same construction process to produce different representations of the object.

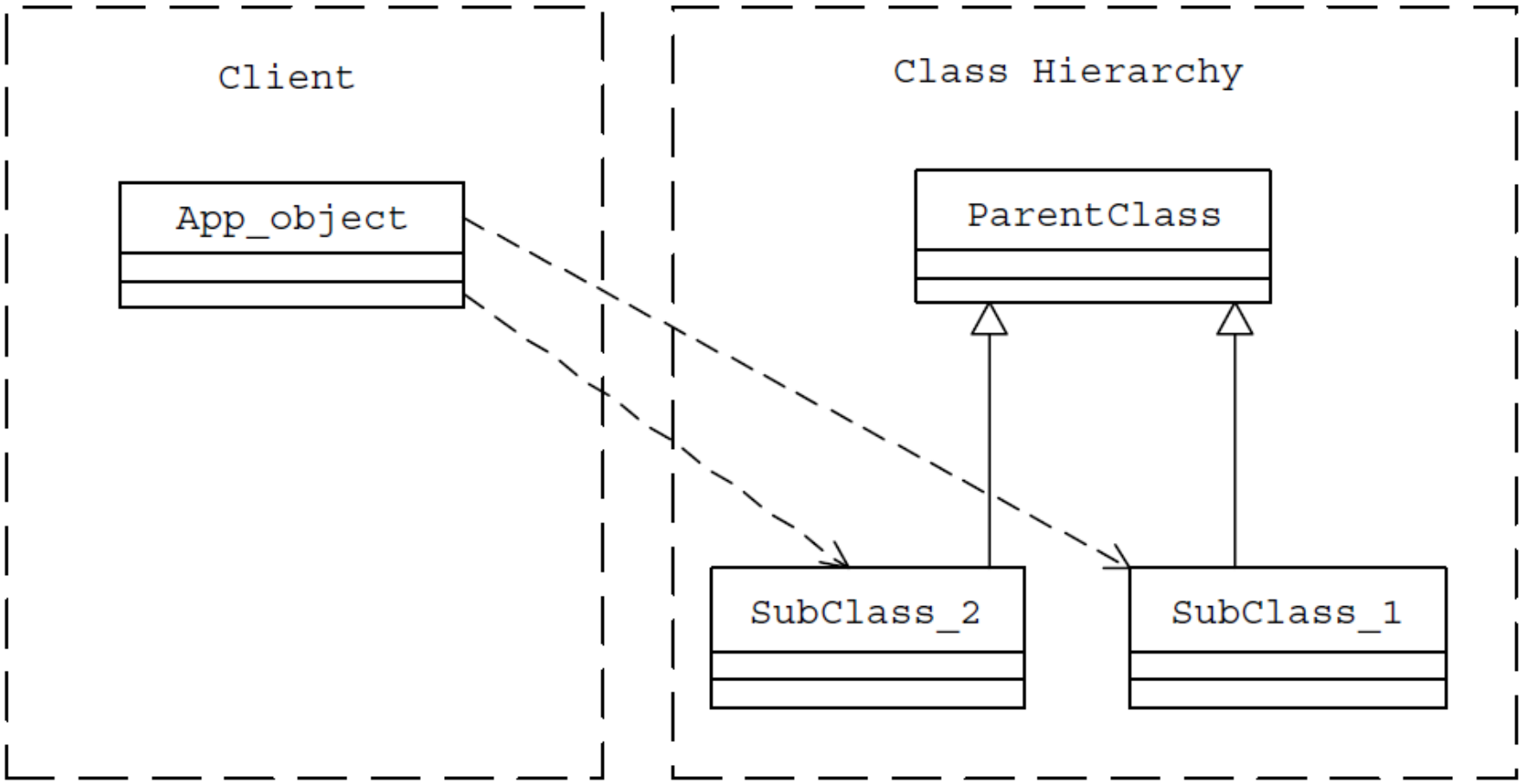
Factory Pattern

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.

At times, an application object may only know that it needs to access a class from within the class hierarchy, but does not know exactly which class from among the set of subclasses of the parent class is to be selected. The choice of an appropriate class may depend on factors such as:

- The state of the running application
- Application configuration settings
- Expansion of requirements or enhancements

Ordinary Design



Problems

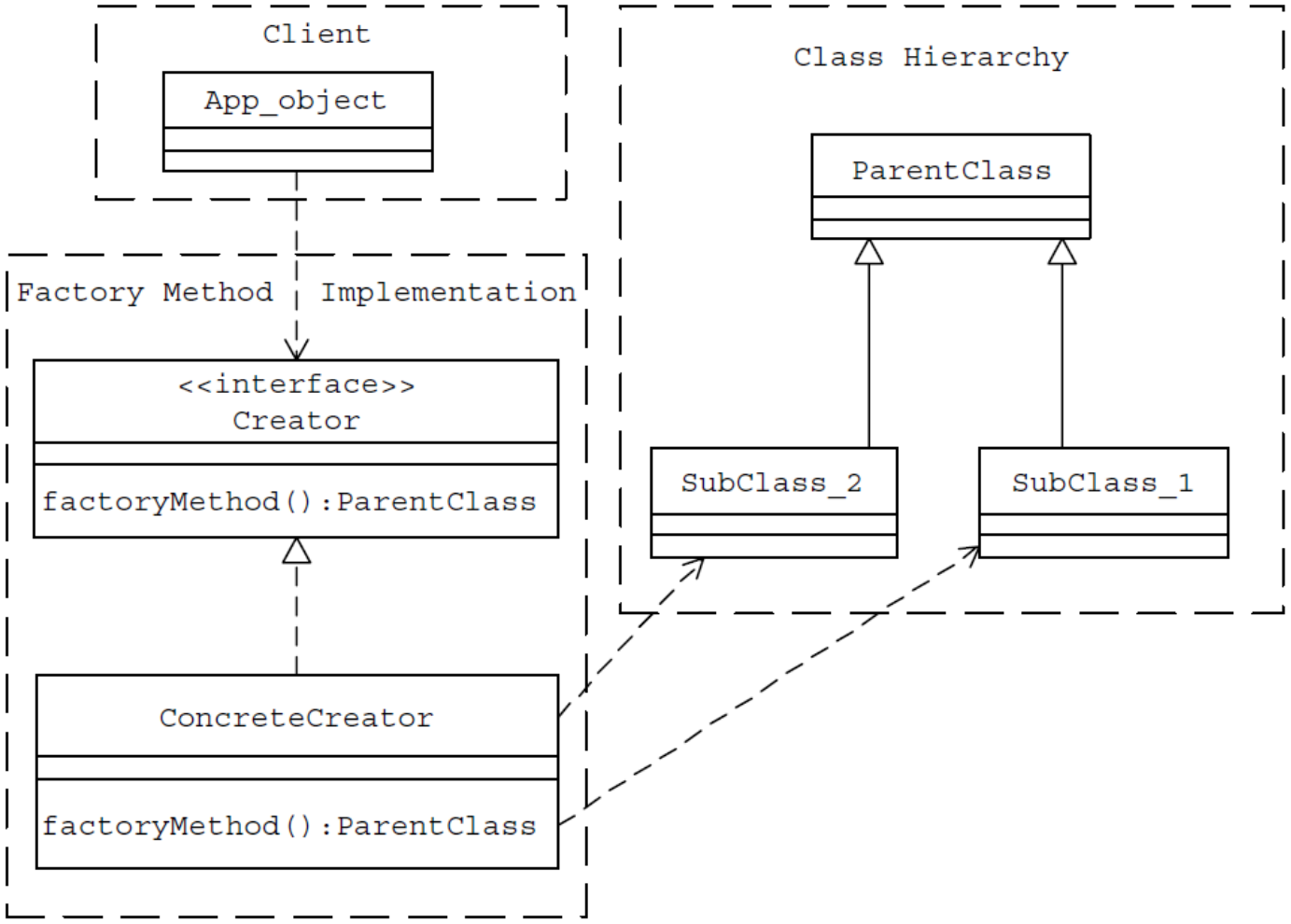
- Because every application object that intends to use the services offered by the class hierarchy needs to implement the class selection criteria, it results in a high degree of coupling between an application object and the service provider class hierarchy.
- Whenever the class selection criteria change, 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.

Factory Method

the Factory Method pattern recommends encapsulating the functionality required, to select and instantiate an appropriate class, inside a designated method referred to as a *factory method*. Thus, a factory method can be defined as a method in a class that:

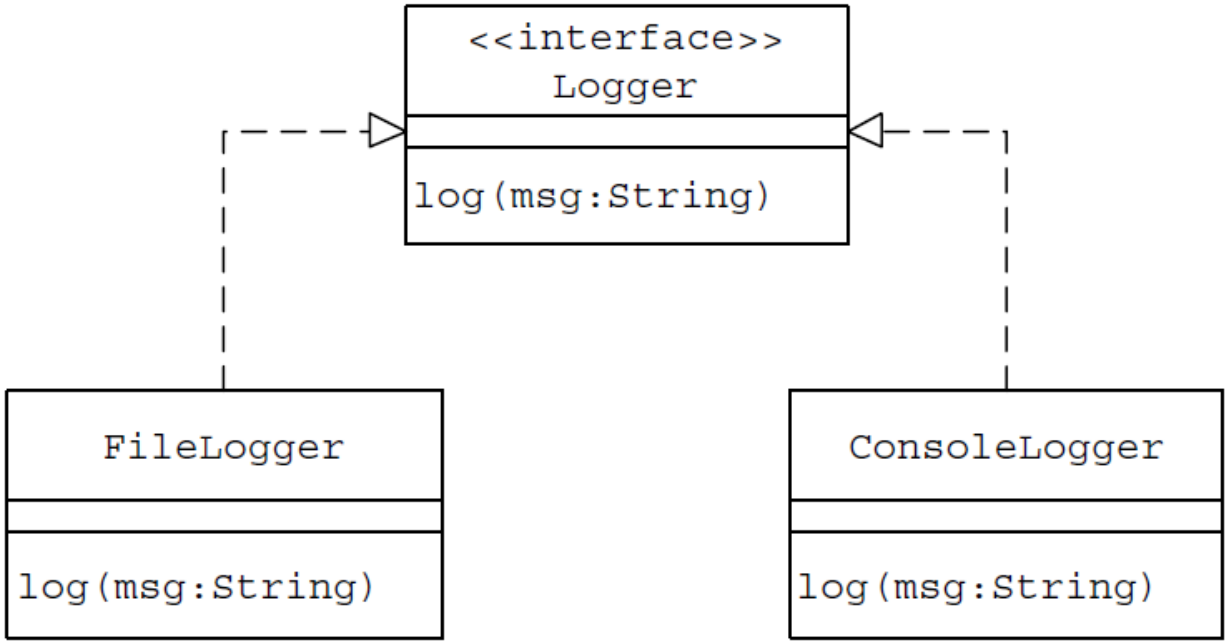
- Selects an appropriate class from a class hierarchy based on the application context and other influencing factors
- Instantiates the selected class and returns it as an instance of the parent class type

Factory Method



Factory Method (Example)

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



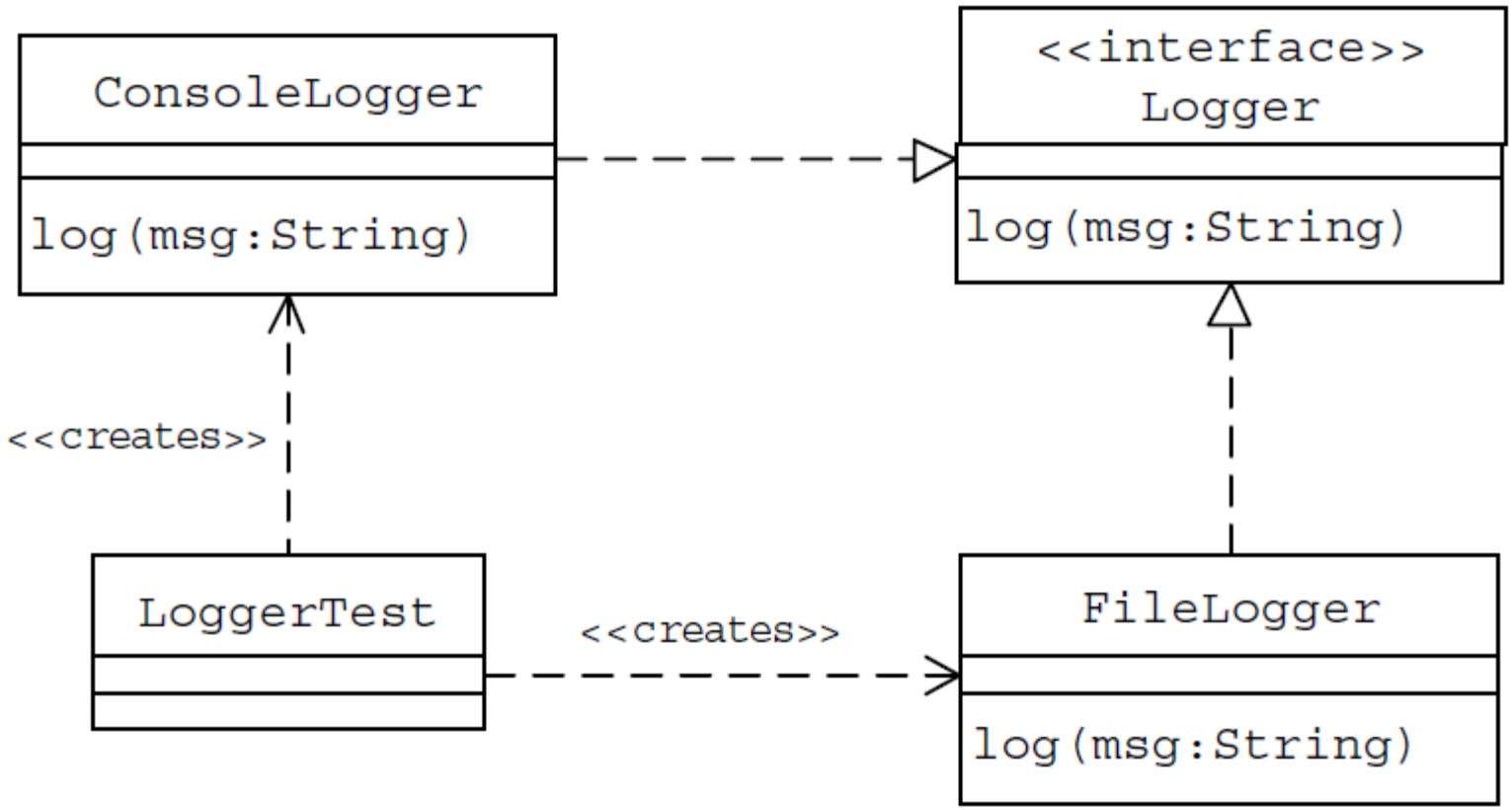
Factory Method (Example)

```
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 `logger.properties` file contents

```
FileLogging=OFF
```

Factory Method (Example)

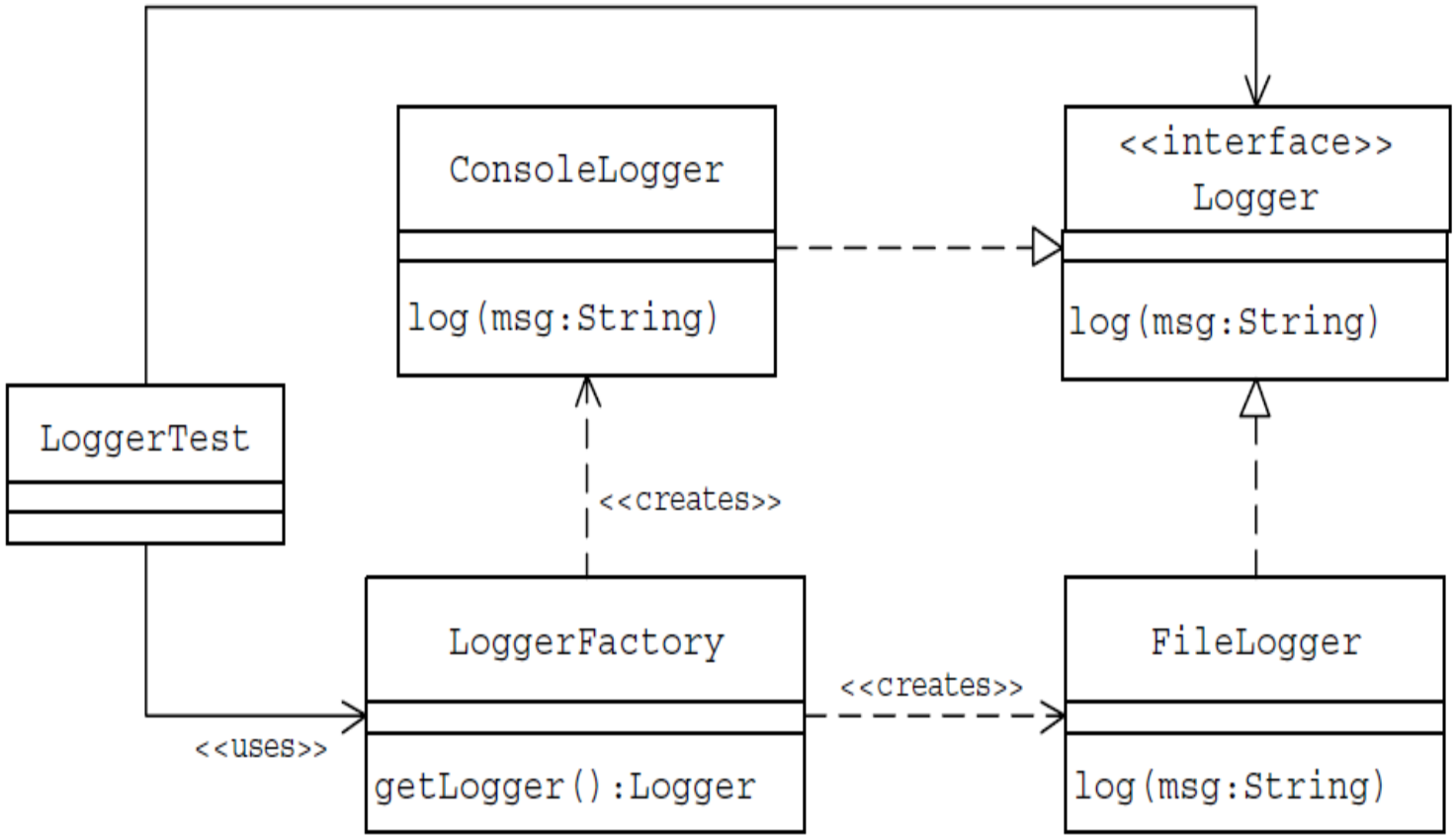


Factory Method (Example)

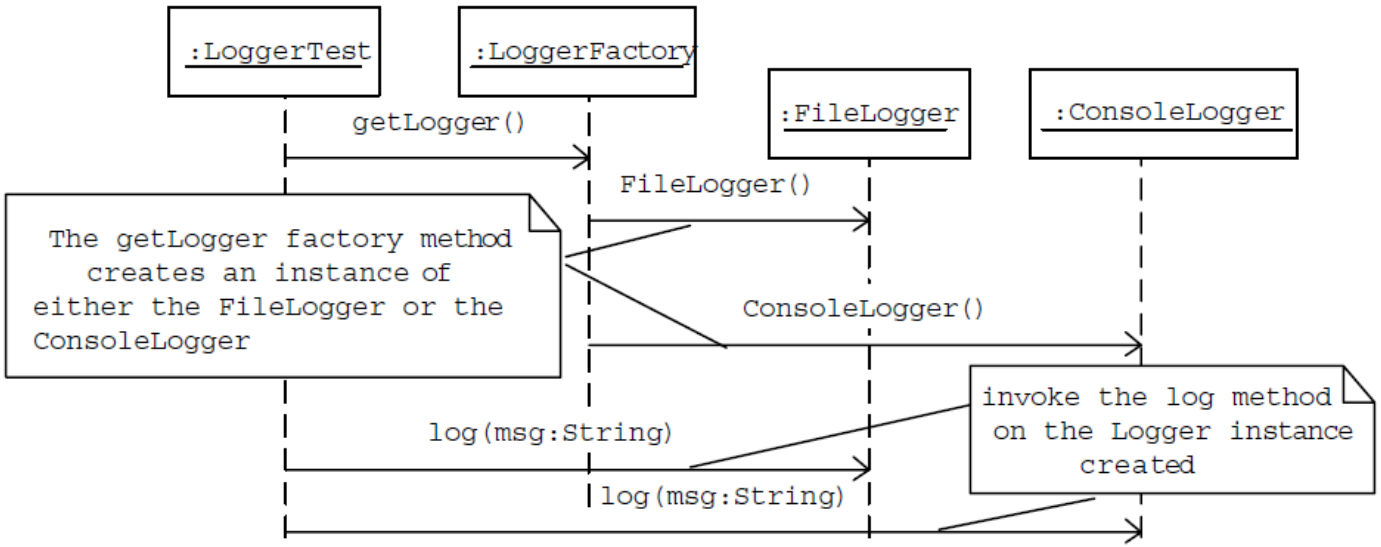
```
public class LoggerFactory {
  public boolean isFileLoggingEnabled() {
    Properties p = new Properties();
    try {
      p.load(ClassLoader.getSystemResourceAsStream(
        "Logger.properties"));
      String fileLoggingValue =
        p.getProperty("FileLogging");
      if (fileLoggingValue.equalsIgnoreCase("ON") == true)
        return true;
      else
        return false;
    } catch (IOException e) {
      return false;
    }
  }

  //Factory Method
  public Logger getLogger() {
    if (isFileLoggingEnabled()) {
      return new FileLogger();
    } else {
      return new ConsoleLogger();
    }
  }
}
```

Factory Method (Example)



Factory Method (Example)

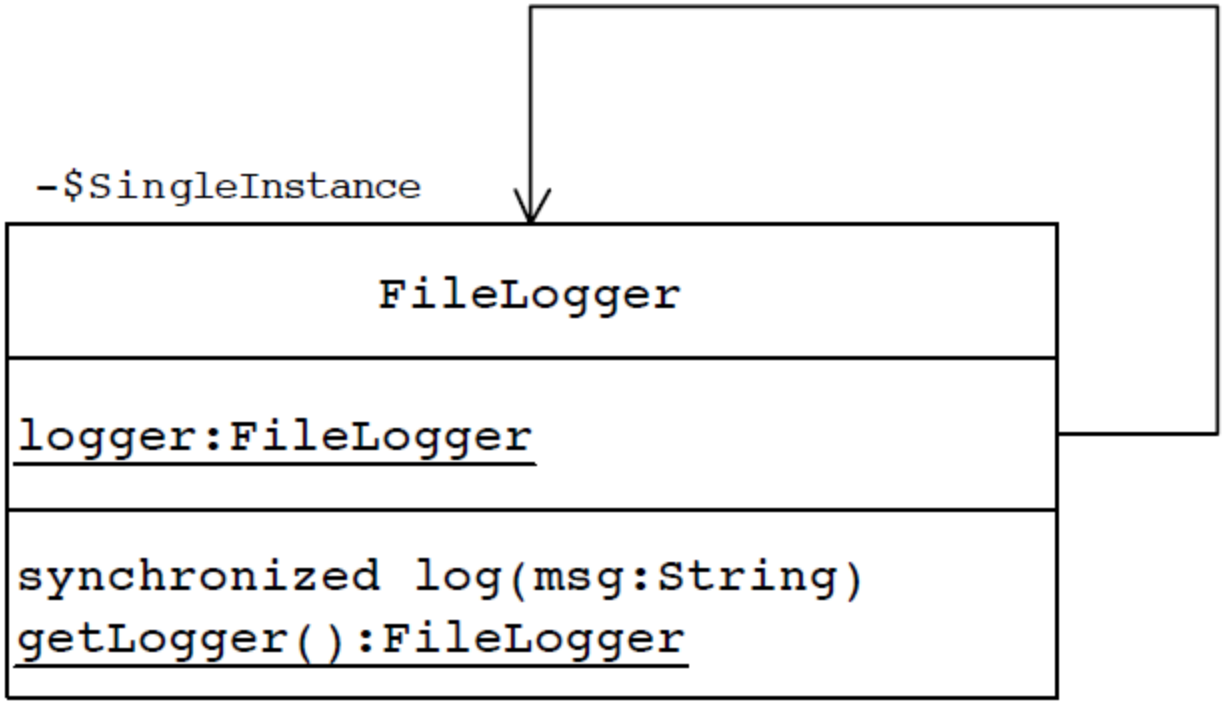



```
public class LoggerTest {
  public static void main(String[] args) {
    LoggerFactory factory = new LoggerFactory();
    Logger logger = factory.getLogger();
    logger.log("A Message to Log");
  }
}
```

Singleton Pattern

- Having an instance of the class in a global variable seems like an easy way to maintain the single instance. All client objects can access this instance in a consistent manner through this global variable. But this does not prevent clients from creating other instances of the class. For this approach to be successful, all of the client objects have to be responsible for controlling the number of instances of the class.
- This widely distributed responsibility is not desirable because a client should be free from any class creation process details.
- **The responsibility for making sure that there is only one instance of the class should belong to the class itself.**

Singleton (Example)



FileLogger Class as a Singleton

Singleton (Example)

```
public class FileLogger implements Logger {
  private static FileLogger logger;
  //Prevent clients from using the constructor
  private FileLogger() {
  }
  public static FileLogger getFileLogger() {
    if (logger == null) {
      logger = new FileLogger();
    }
    return logger;
  }
  public synchronized void log(String msg) {
    FileUtil futil = new FileUtil();
    futil.writeToFile("log.txt",msg, true, true);
  }
}
```

Singleton (Example)

- **Make the Constructor Private**
- **Static Public Interface to Access an Instance**

```
//client code
public class LoggerTest{
  public static void main(String[] args){
     LoggerFactory factory=new LoggerFactory();
     //factory method call
     Logger logger=factory.getLogger();
     logger.log("A Message to Log");
  }
}
```

Singleton (Example)

```
public class LoggerFactory {
  public boolean isFileLoggingEnabled() {
    Properties p = new Properties();
    try {
      p.load(ClassLoader.getSystemResourceAsStream(
        "Logger.properties"));
      String fileLoggingValue =
        p.getProperty("FileLogging");
      if (fileLoggingValue.equalsIgnoreCase("ON") == true)
        return true;
      else
        return false;
    } catch (IOException e)
      return false;
    }
  }
```

```
public Logger getLogger() {
  if (isFileLoggingEnabled()) {
    return FileLogger.getFileLogger();
  } else {
    return new ConsoleLogger();
  }
}
}
```

Builder Pattern

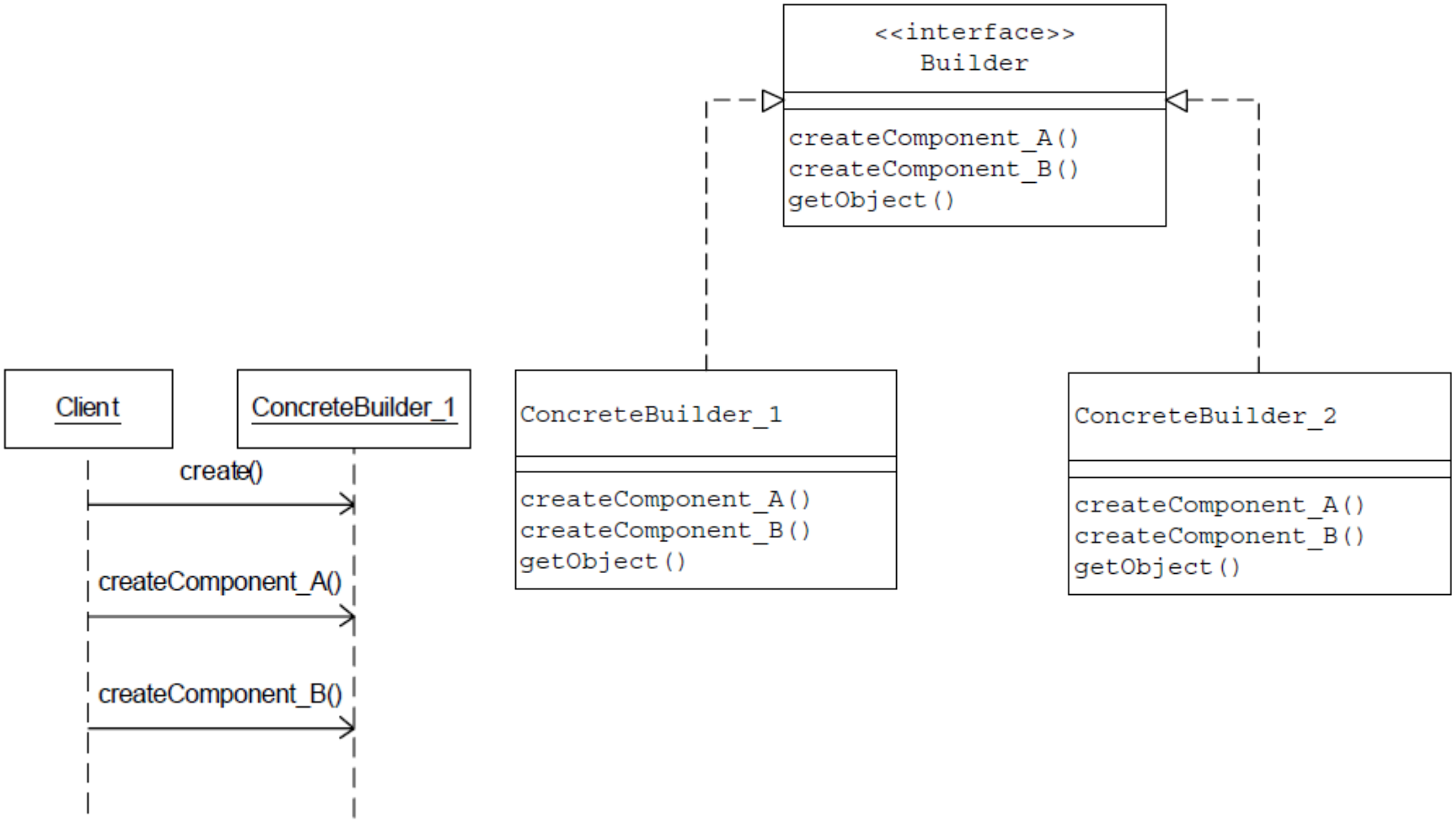
- This design may not be effective when the object being created is complex and the series of steps constituting the object creation process can be implemented in different ways producing different representations of the object.

- Using the Builder pattern, the process of constructing such an object can be designed more effectively. The Builder pattern suggests moving the construction logic out of the object class to a separate class referred to as *a builder* class.

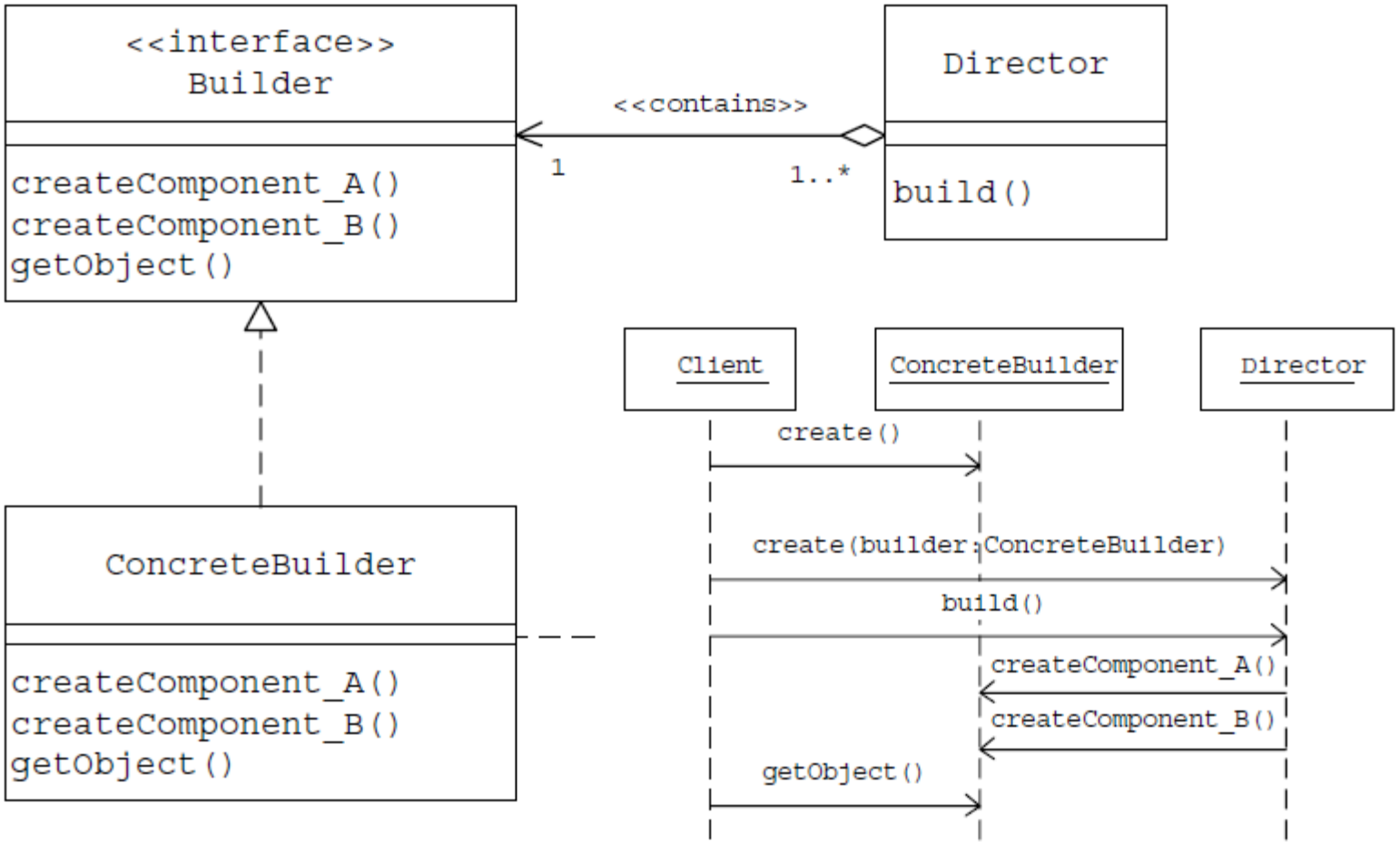
Builder

- The design turns out to be more modular with each implementation contained in a different builder object.
- Adding a new implementation (i.e., adding a new builder) becomes easier.
- The object construction process becomes independent of the components that make up the object. This provides more control over the object construction process.

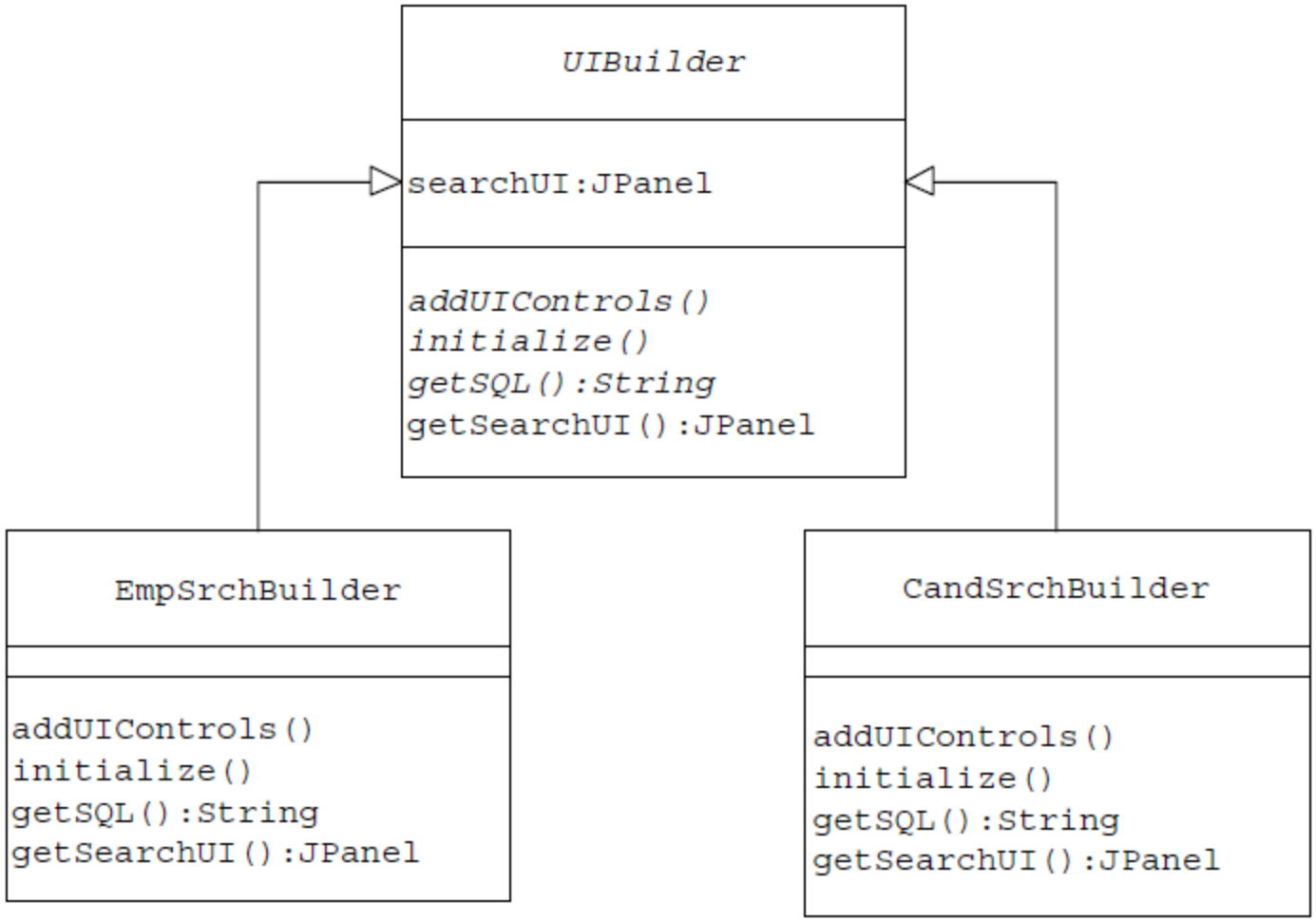
Builder


<<interface>>
Builder
createComponent_A()
createComponent_B()
getObject()
ConcreteBuilder_1
createComponent_A()
createComponent_B()
getObject()
ConcreteBuilder_2
createComponent_A()
createComponent_B()
getObject()
Client
ConcreteBuilder_1
create()
createComponent_A()
createComponent_B()

Director Object



Builder (Example)



Abstract UIBuilder Class

```
public abstract class UIBuilder {
  protected JPanel searchUI;
  //add necessary UI controls and initialize them
  public abstract void addUIControls();
  public abstract void initialize();
  //return the SELECT sql command for the specified criteria
  public abstract String getSQL();
  //common to all concrete builders.
  //returns the fully constructed search UI
  public JPanel getSearchUI() {
    return searchUI;
  }
}
```

Concrete Builder

```
class EmpSrchBuilder extends UIBuilder {
          …
          …
  public void addUIControls() {
    searchUI = new JPanel();
    JLabel lblUserName = new JLabel("Name :");
    JLabel lblCity = new JLabel("City:");
    JLabel lblRenewal = new JLabel("Membership Renewal :");
    GridBagLayout gridbag = new GridBagLayout();
    searchUI.setLayout(gridbag);
    GridBagConstraints gbc = new GridBagC
    searchUI.add(lblUserName);
    searchUI.add(txtUserName);
```

```
class CandSrchBuilder extends UIBuilder {
          …
          …
  public void addUIControls() {
    searchUI = new JPanel();
    JLabel lblUserName = new JLabel("Name :");
    JLabel lblExperienceRange =
      new JLabel("Experience(min Yrs.):");
    JLabel lblSkill = new JLabel("Skill :");
    cmbExperience.addItem("<5");
    cmbExperience.addItem(">5");
    GridBagLayout gridbag = new GridBagLayout();
    searchUI.setLayout(gridbag);
```

Concrete Builder

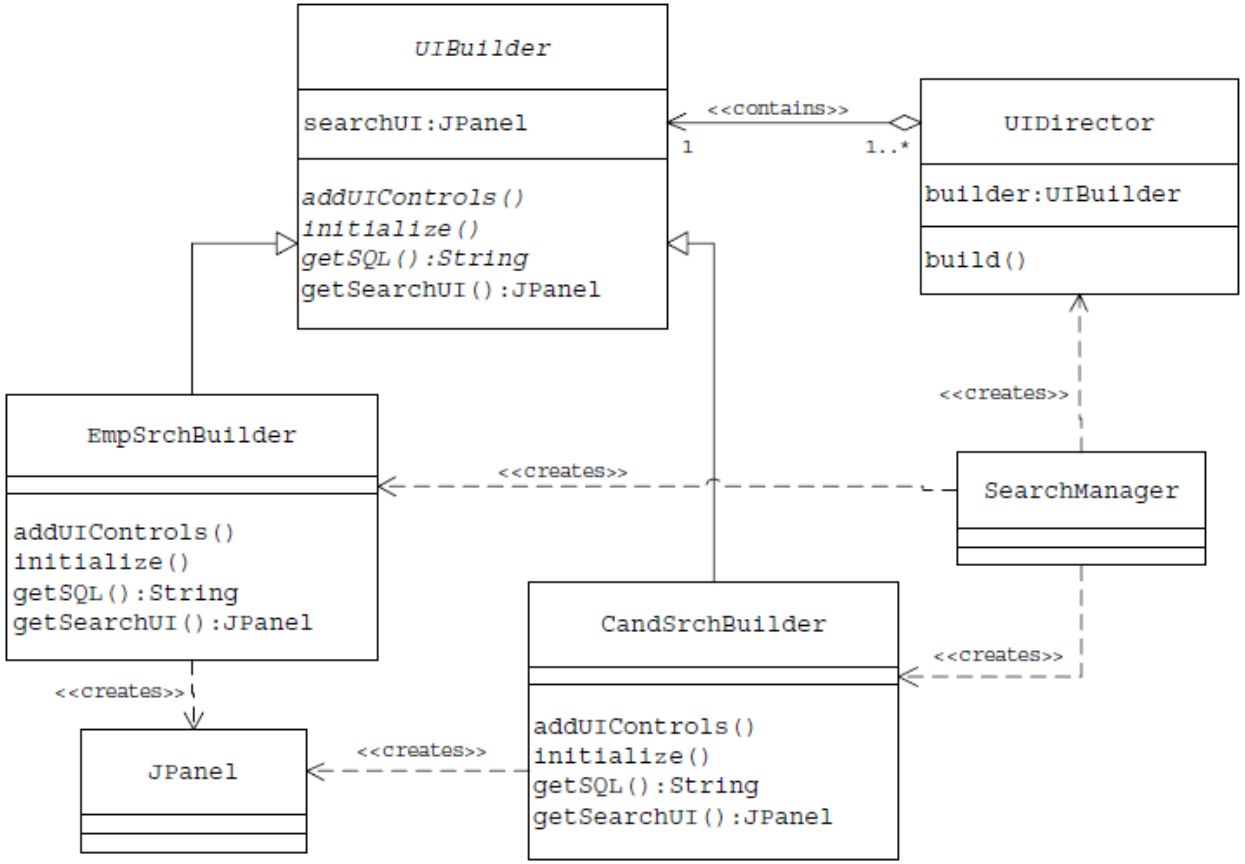
Builder	*Responsibility*
`EmpSrchBuilder`	• Builds a `JPanel` object with the necessary UI controls for the employer search • Initializes UI controls • Returns the fully constructed `JPanel` object as part of the `getSearchUI` method • Builds the required SQL select command and returns it as part of the `getSQL` method
`CandSrchBuilder`	• Builds a `JPanel` object with the necessary UI controls for the candidate search • Initializes UI controls • Returns the fully constructed `JPanel` object as part of the `getSearchUI` method • Builds the required SQL select command and returns it as part of the `getSQL` method

Director Class

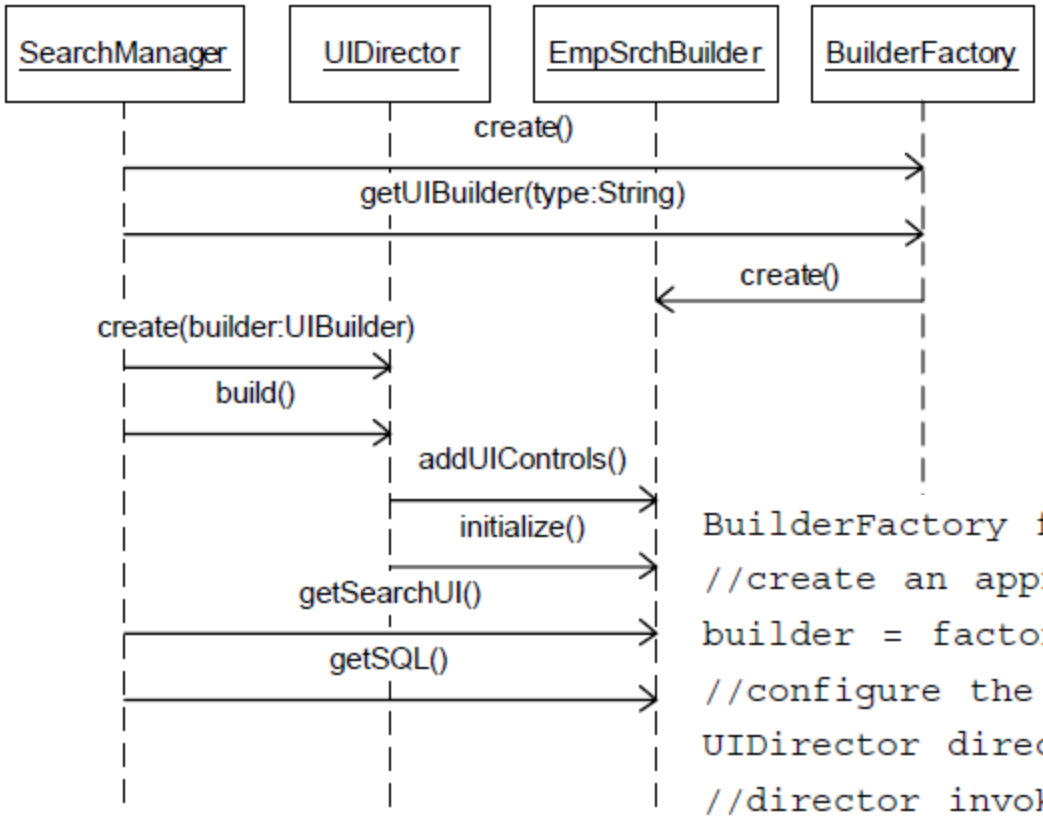
```
public class UIDirector {
  private UIBuilder builder;
  public UIDirector(UIBuilder bldr) {
    builder = bldr;
  }
  public void build() {
    builder.addUIControls();
    builder.initialize();
  }
}
```

```
class BuilderFactory {
  public UIBuilder getUIBuilder(String str) {
    UIBuilder builder = null;
    if (str.equals(SearchManager.CANDIDATE_SRCH)) {
      builder = new CandSrchBuilder();
    } else if (str.equals(SearchManager.EMPLOYER_SRCH)) {
      builder = new EmpSrchBuilder();
    }
    return builder;
  }
}
```

Finally


UIBuilder
searchUI:JPanel
addUIControls()
initialize()
getSQL():String
getSearchUI():JPanel
<<contains>>
1
1..*
UIDirector
builder:UIBuilder
build()
<<creates>>
EmpSrchBuilder
addUIControls()
initialize()
getSQL():String
getSearchUI():JPanel
<<creates>>
SearchManager
<<creates>>
CandSrchBuilder
addUIControls()
initialize()
getSQL():String
getSearchUI():JPanel
<<creates>>
<<creates>>
JPanel
<<creates>>

Final Cut

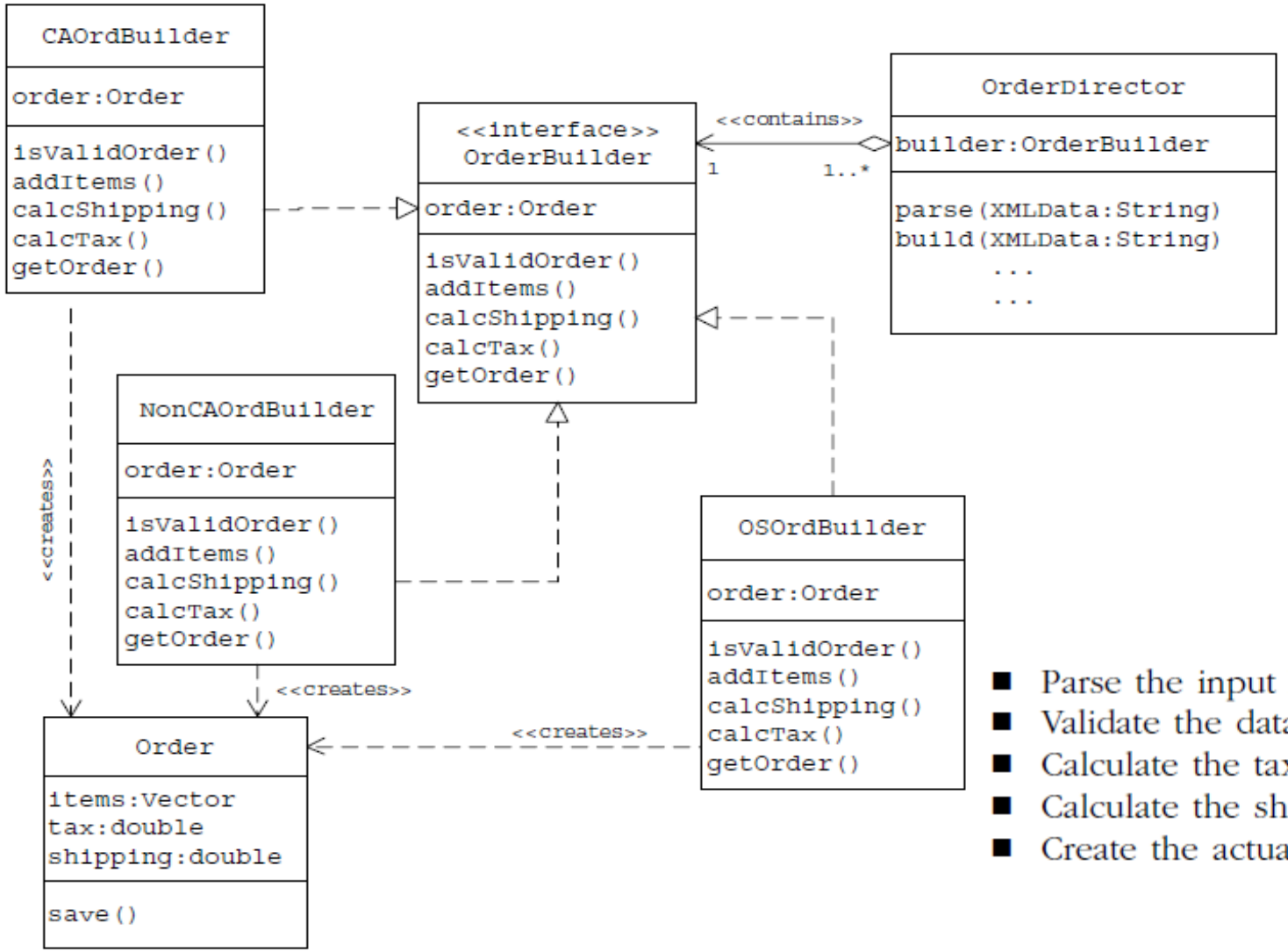



```
BuilderFactory factory = new BuilderFactory();
//create an appropriate builder instance
builder = factory.getUIBuilder(selection);
//configure the director with the builder
UIDirector director = new UIDirector(builder);
//director invokes different builder
//methods
director.build();
//get the final build object
JPanel UIObj = builder.getSearchUI();
```

Builder (Example 2)

S. No	Order Type	Details
1	Overseas orders	• Orders from countries other than the United States. Additional shipping and handling is charged for these orders. • Overseas orders are accepted only if the order amount is greater than $100.
2	California orders	• U.S. orders with shipping address in California and are charged additional sales tax. • Orders with $100 or more order amount receive free regular shipping.
3	Non-California orders	• U.S. orders with shipping address not in California. Additional sales tax is not applicable. • Orders with $100 or more order amount receive free regular shipping.

Director



- Parse the input XML string
- Validate the data
- Calculate the tax
- Calculate the shipping
- Create the actual object with: