Lesson 7 Builder Flyweight and Façade

The Builder Pattern

1. Intent

   Separate the construction of a complex object from its representation so that the same construction process can create different representations.

2. Motivation

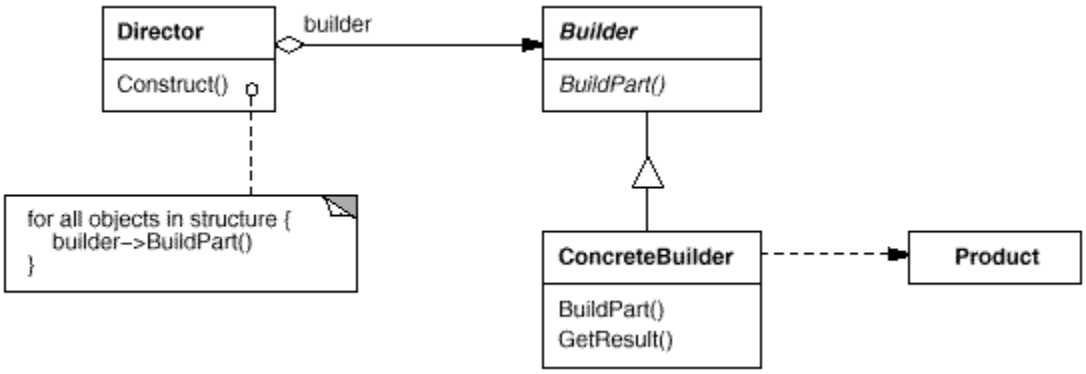
   a. Your application needs to create complex objects with many attributes of both primitive and reference types.
   b. Instead of having a constructor that takes all of them as parameters, you choose a step-by-step approach to construct the complex object.
   c. You want to avoid creating a partially initiated object instance.
   d. Your construction process stays the same for creating a series of related products.

3. Applicability

   Use the Builder pattern when:

   a. The algorithm for creating a complex object should be independent of the parts that make up the object and how they're assembled.
   b. The construction process must allow different representations for the object that's constructed.

4. Structure



5. Participants

a. Builder
- Specifies an abstract interface for creating parts of a Product object.

b. ConcreteBuilder
- Constructs and assembles parts of the product by implementing the Builder interface.
- Defines and keeps track of the representation it creates.
- Provides an interface for retrieving the product.

c. Director
- Constructs an object using the Builder interface.

d. Product
- Represents the complex object under construction. ConcreteBuilder builds the product's internal representation and defines the process by which it's assembled.
- Includes classes that define the constituent parts, including interfaces for assembling the parts into the final result.

6. Consequences

a. It lets you vary a product's internal representation. The Builder interface only provides the director the steps for constructing the product without telling how the product gets assembled.
b. It isolates code for construction and representation. Clients do not need to know anything about the classes that define the product's internal structure. Each Concrete Builder contains all the code to create and assemble a particular kind of product.
c. It gives you finer control over the construction process (step-by-step vs construction in one shot as other creational patterns do), thus gives you finer control over the construction process and consequently the internal structure of the resulting product.

7. How to implement the Builder pattern?

a. Product class

```
public class Meal {

	private String name;
	private Drink drink;
	private MainDish mainDish;
	private SideDish sideDish;

	…

}
```

b. Builder interface

```
public interface MealBuilder {
	public void buildDrink();
	public void buildMainDish();
	public void buildSideDish();
	public Meal getMeal();
```

```
}
```

c. Concrete Builder classes

```
public class ItalianMealBuilder implements MealBuilder {

	private Meal meal;

	public ItalianMealBuilder() {
		meal = new Meal("Italian Combo");
	}

	@Override
	public void buildDrink() {
		Drink drink = new Drink("red wine");
		meal.setDrink(drink);
	}

	@Override
	public void buildMainDish() {
		MainDish main = new MainDish("pizza");
		meal.setMainDish(main);
	}

	@Override
	public void buildSide() {
		SideDish side = new SideDish("bread");
		meal.setSideDish(side);
	}

	@Override
	public Meal getMeal() {
		return meal;
	}

}

public class MexicanMealBuilder implements MealBuilder {

	private Meal meal;

	public MexicanMealBuilder() {
		meal = new Meal("Tequila Set");
	}

	@Override
	public void buildDrink() {
		Drink drink = new Drink("Tequila");
		meal.setDrink(drink);
	}

	@Override
	public void buildMainDish() {
		MainDish main = new MainDish("Tortas");
		meal.setMainDish(main);
```

```
	}

	@Override
	public void buildSide() {
		SideDish side = new SideDish("Arroz con leche");
		meal.setSideDish(side);
	}

	@Override
	public Meal getMeal() {
		return meal;
	}
}
```

d. Director class

```
public class MealDirector {

	private MealBuilder mealBuilder = null; //or multiple meal
builders

	public MealDirector(MealBuilder mealBuilder) {
		this.mealBuilder = mealBuilder;
	}

	public void constructMeal() {
		mealBuilder.buildDrink();
		mealBuilder.buildMainDish();
		mealBuilder.buildSide();
	}

	public Meal getMeal() {
		return mealBuilder.getMeal();
	}

}
```

e. Client

```
public class BuilderClient {
	public static void main(String[] args) {
		MealDirector md = new MealDirector(new
ItalianMealBuilder());
		md.constructMeal();
		Meal meal = md.getMeal();
		System.out.println(meal);
		//see the difference between the 2 code blocks
		md = new MealDirector(new MexicanMealBuilder());
		md.constructMeal();
		meal = md.getMeal();
		System.out.println(meal);
	}
}
```

Lab 7-1

For some applications, the main purpose is to create objects that contain information they need to drive the business. One example is a customer services application used at call centers. Other than servicing the customers, the business needs to get the data during the call – a 'request' object that has all details regarding the conversation.

```
public class Request {
	private String reqId;
	private Customer requester;
	private Agent agent;
	private String reqContent;
	private String respContent;
	private boolean isAnswered;
	private boolean isSaleLead;
	private boolean needCallBack;
}
```

During the call, the representative answers questions and records them. At the end of the call, a request object is created with all necessary state information set before saving it in the database. Obviously, request object creation is not a one-shot job. You can build it step-by-step using the Builder Pattern.

Logical steps to create a request object that has 3 parts: 'opening', 'response content', and 'closing'.

1. When a call is connected, the request id and agent information becomes available.
   ```
   public class Call {
   	private String requestId;
   	private Agent agent;

   	public void callPop(Agent agent){
   		this.agent = agent;
   		this.requestId = ""+(new Date()).getTime();
   	}

   }
   ```
2. The agent will ask for the caller's phone number to retrieve the caller's information from the database (or if it is a new customer, agent will create user profile)
3. Agent will get/collect what the caller is requesting during the call (as request content)
4. Agent will answer the questions for the request (as response content)
5. If Agent cannot answer a question, a consultation to supervisor or even a third-party (like a credit agency) can be done during the call.
6. Agent will set 'isAnswered' and 'needCallback' accordingly and close the call (saving all information in the database)

Modify the 'Request' class so that it has parts 'opening', 'response content' and 'closing' in addition to other necessary attributes. Then use the builder pattern to create 'Request' objects.

The Flyweight Pattern

1. Intent

   Use sharing to support large numbers of fine-grained objects efficiently.

2. Motivation

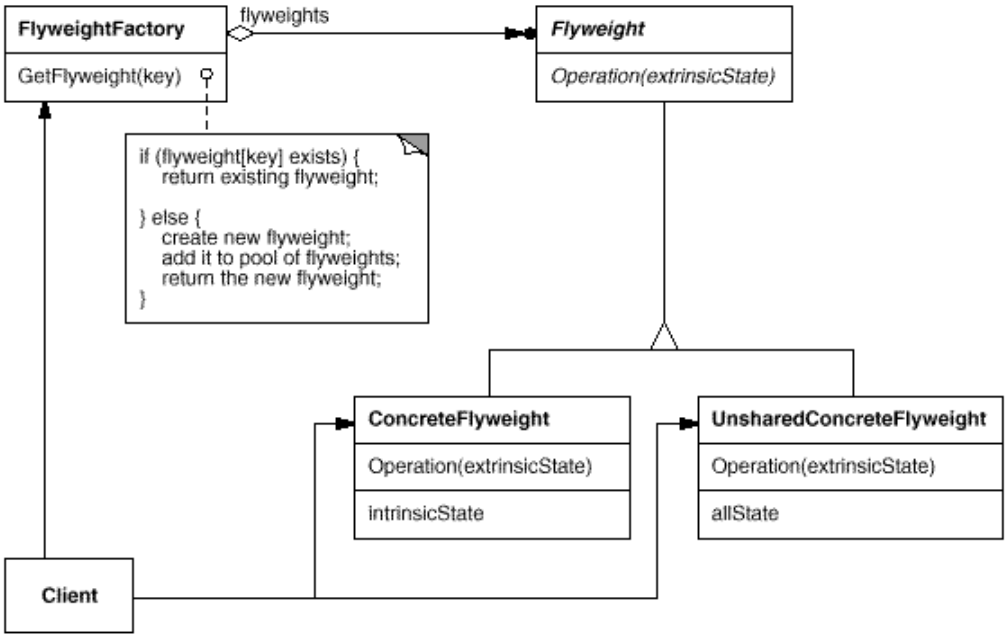
   a. Your application deals with a potentially large number of objects of the same type, which may take a huge amount of memory space.
   b. For these objects, their states can be easily externalized.
   c. For each type of objects, you can choose to process them one by one, not all at once.
   d. Performance of the application is not critical sometimes.

3. Applicability

   The Flyweight pattern's effectiveness depends heavily on how and where it's used. Apply the Flyweight pattern when all of the following are true:

   a. An application uses a large number of objects.
   b. Storage costs are high because of the sheer quantity of objects.
   c. Most object state can be made extrinsic.
   d. Many groups of objects may be replaced by relatively few shared objects once extrinsic state is removed.
   e. Creating a new object has to be much more expensive than object retrieval from the factory.

4. Structure



5. Participants

a. Flyweight
- declares an interface through which flyweights can receive and act on extrinsic state.

b. ConcreteFlyweight
- implements the Flyweight interface and adds storage for intrinsic state, if any.
- must be sharable. Any state it stores must be intrinsic.

c. UnsharedConcreteFlyweight
- The Flyweight interface enables sharing; it doesn't enforce it. It's common for UnsharedConcreteFlyweight objects to have ConcreteFlyweight objects as children at some level in the flyweight object structure.

d. FlyweightFactory
- creates and manages flyweight objects.

- ensures that flyweights are shared properly. When a client requests a flyweight, the FlyweightFactory object supplies an existing instance or creates one, if none exists.

e. Client
- maintains a reference to flyweight(s).
- computes or stores the extrinsic state of flyweight(s).

6. Consequences

Flyweights may introduce run-time costs associated with transferring, finding, and/or computing extrinsic state, especially if it was formerly stored as intrinsic state. However, such costs are offset by space savings, which increase as more flyweights are shared.

7. How to implement the Flyweight pattern?

a. Flyweight Interface

```
public interface Flyweight {
	public void draw(Graphics g, String string,
		Color color, int x, int y);
}
```

b. Concrete Flyweight

```
public class ConcreteFlyweight implements Flyweight{
	Color color; //only the intrinsic state

	public ConcreteFlyweight(Color color) {
		this.color = color;
	}

	public void draw(Graphics g, String string,
		Color color, int x, int y){
		g.drawString(string, x, y);
	}
}
```

c. Unshared concrete flyweight

```
public class ConcreteHeavyweight implements Flyweight{
	private Color color = null; //intrinsic state
	private String string = ""; //extrinsic state
	private int x, y; //extrinsic state

	public ConcreteHeavyweight(String string,
			Color color, int x, int y) {
		this.string = string;
		this.color = color;
		this.x = x;
		this.y = y;
	}

	public void draw(Graphics g, String str,
			Color color, int i, int j) {
		g.setColor(color);
		g.drawString(str, i, j);
	}

}
```

d. Flyweight Factory

```
public class FlyweightFactory {
	static ConcreteFlyweight byColor[] = new ConcreteFlyweight[6];
	static {
		byColor[0] = new ConcreteFlyweight(Color.red);
		byColor[1] = new ConcreteFlyweight(Color.blue);
		byColor[2] = new ConcreteFlyweight(Color.yellow);
		byColor[3] = new ConcreteFlyweight(Color.orange);
		byColor[4] = new ConcreteFlyweight(Color.black);
		byColor[5] = new ConcreteFlyweight(Color.white);
	}

	public static ConcreteFlyweight getInstance(Color color) {
		int i = Math.abs(color.hashCode() % 6);
		ConcreteFlyweight line = byColor[i];

		return line;
	}

	public static ConcreteFlyweight getInstance(int i) {
		return byColor[i];
	}
}
```

e. Flyweight client

```
	Graphics g = panel.getGraphics();
	for (int i = 0; i < NUMBER_OF_LINES; ++i) {
		Color color = getRandomColor();
		ConcreteFlyweight line =
				flyweightFactory.getInstance(color);
```

```
            line.draw(g, "Hello", getRandomX(),
                        getRandomY());

            }
```

f. Heavyweight client

```
      Graphics g = panel.getGraphics();
      for (int i = 0; i < NUMBER_OF_LINES; ++i) {
            Color color = getRandomColor();
            ConcreteHeavyweight sh = new ConcreteHeavyweight(new
                        String("Hello"), color,
                  getRandomX(), getRandomY());
            sh.draw(g, "", color, 0,0);
      }
```

Lab 7-2

Suppose you are working for a health center that provides medical aid services to residents in neighboring cities and towns. In the original design, a core class 'Customer' looked like this –

```
public class Customer {
      private int customerId;
      private String firstName;
      private String lastName;
      private Address residenceAddress;
      private HealthProfile profile;
      //city map with a red dot representing residence location
      private Image locationMap;
}
```

One issue with it is when a large number of customers get online during the same period of time, customers experience longer response times. One reason, among others, was because of the huge memory consumption caused by the large number of concurrent users. Your job is to roll out a fix to the issue. Use the Flyweight pattern to reduce the space need of concurrent requests from online customers.

The Façade Pattern

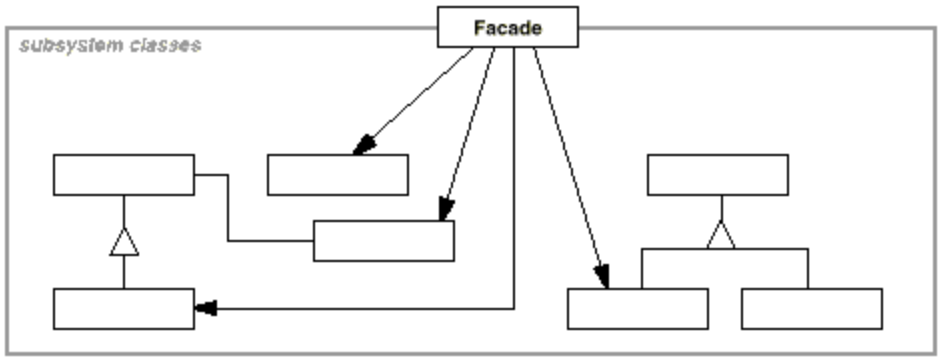
1. Intent

   Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

2. Motivation

   e. Structuring a system into subsystems helps reduce complexity.
   f. Subsystems are groups of classes, and/or other subsystems.
   g. The interface exposed by the classes in a subsystem or set of subsystems can become quite complex.
   h. One way to reduce this complexity is to introduce a facade object that provides a single, simplified interface to the more general facilities of a subsystem.

3. Structure



4. Consequences

a. It hides the implementation of the subsystem from clients, making the subsystem easier to use.
b. It promotes weak coupling between the subsystem and its clients. This allows you to change the classes that comprise the subsystem without affecting the clients.
c. It reduces compilation dependencies in large software systems.
d. However, it does not prevent sophisticated clients from accessing the underlying classes.
e. And Facade does not add any functionality, it just simplifies interfaces.

5. How to implement the Façade pattern?

Let's see an example "LoanManager" class

```
public class LoanManagerBefore {
      public void Process(LoanApplication application) {
            boolean personalDetailsAreValid = new PersonalDetailVerifier()
                        .verifyDetails(application);

            boolean professionalDetailsAreValid = new
ProfessionalDetailVerifier()
                        .verifyDetails(application);

            boolean creditHistoryVerified = new CreditHistoryVerifier()
                        .verify(application);

            boolean isPreferredDeveloper = new DeveloperVerifier()

      .isPreferredDeveloper(application.getDeveloperName());

            boolean isProjectApproved = isPreferredDeveloper ? true
                        : new ProjectVerifier()
.isBlackListedProject(application.getProjectName());

            boolean isEligible = personalDetailsAreValid
                        && professionalDetailsAreValid &&
creditHistoryVerified   && isProjectApproved;

            if (isEligible) {
```

```
                  NotificationService notificationService = new
NotificationService();
                  notificationService.notify(application);
            }
      }
      }
```

LoanManager refactored using the Façade pattern

```
public class LoanManagerRefactored {
      public void Process(LoanApplication application) {
            ILoanApplicationVerifier loanApplicationVerifier = new
            LoanApplicationVerifier();

            boolean isEligible =
            loanApplicationVerifier.isEligible(application);

            if (isEligible) {
                  NotificationService notificationService = new
                  NotificationService();
                  notificationService.notify(application);
            }
      }
}
```

Lab 7-3

Refactor the demo class with the Façade pattern to simplify the client code.

```
public class JDBCdemo {
      public static void main(String[] arg) {
            Connection conn = null;
            PreparedStatement prep = null;
            CallableStatement call = null;
            ResultSet rset = null;
            try {
                  Class.forName("<driver>").newInstance();
                  conn = DriverManager.getConnection("<database>");
                  String sql = "SELECT * FROM <table> WHERE <column name> =
?";
                  prep = conn.prepareStatement(sql);
                  prep.setString(1, "<column value>");
                  rset = prep.executeQuery();
                  if (rset.next()) {
                        System.out.println(rset.getString("<column name"));
                  }
                  sql = "{call <stored procedure>( ?, ? )}";
                  call = conn.prepareCall(sql);
                  call.setInt(1, 1972);
                  call.registerOutParameter(2, java.sql.Types.INTEGER);
                  call.execute();
```

```
                System.out.println(call.getInt(2));
            } catch (SQLException e) {
                e.printStackTrace();
            } catch (InstantiationException e) {
                e.printStackTrace();
            } catch (ClassNotFoundException e) {
                e.printStackTrace();
            } catch (IllegalAccessException e) {
                e.printStackTrace();
            } finally {
                if (rset != null) {
                    try {
                        rset.close();
                    } catch (SQLException ex) {
                        ex.printStackTrace();
                    }
                }
                if (prep != null) {
                    try {
                        prep.close();
                    } catch (SQLException ex) {
                        ex.printStackTrace();
                    }
                }
                if (call != null) {
                    try {
                        call.close();
                    } catch (SQLException ex) {
                        ex.printStackTrace();
                    }
                }
                if (conn != null) {
                    try {
                        conn.close();
                    } catch (SQLException ex) {
                        ex.printStackTrace();
                    }
                }
            }
        }
}
```