СБЕРБАНК ТЕХНОЛОГИИ

Шаблоны проектирования

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ

Паттерны это примеры правильных подходов к решению типичных задач проектирования

ШАБЛОНЫ ПРОЕКТИРОВАНИЯ

- Порождающие шаблоны
- Структурные шаблоны
- Поведенческие шаблоны

ПОРОЖДАЮЩИЕ ШАБЛОНЫ

ШАБЛОН	ОПИСАНИЕ	ЧТО ПОЗВОЛЯЕТ МЕНЯТЬ
Абстрактная фабрика	Предоставляет интерфейс для создания семейств связанных или зависимых объектов без указания их конкретных классов	Семейство объектов продуктов
Строитель	Отделяет создание сложного объекта от его представления чтобы один процесс создания мог давать разные представления	Как создаётся сложный объект
Фабричный метод	Определяет интерфейс для создания объекта и позволяет подклассам принимать решение экземпляр какого класса создавать	Подкласс создаваемого объекта
Прототип	Позволяет создавать новые объекты копируя существующий прототип	Класс создаваемого экземпляра
Одиночка	Гарантирует что у класса есть только один экземпляр и предоставляет глобальную точку доступа к нему	Единственный экземпляр класса

Порождающие шаблоны помогают управлять созданием экземпляров объектов.

АБСТРАКТНАЯ ФАБРИКА

Абстрактная фабрика (Abstract Factory):

Предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов, не специфицируя их конкретных классов.

АБСТРАКТНАЯ ФАБРИКА

```
public interface VehicleFactory {
    Body createBody();
    Chassis createChassis();
    Windows createWindows();
}
```

АБСТРАКТНАЯ ФАБРИКА

Интерфейсы деталей транспортного средства будут следующие:

```
public interface Body {
    String getBodyParts();
}
public interface Chassis {
    String getChassisParts();
}
public interface Windows {
    String getWindowParts();
}
```

АБСТРАКТНАЯ ФАБРИКА

Создание конкретных классов:

```
public class VanFactory implements VehicleFactory {
    public Body createBody() {
        return new VanBody();
    }

    public Chassis createChassis() {
        return new VanChassis();
    }

    public Windows createWindows() {
        return new VanWindows();
    }
}
```

АБСТРАКТНАЯ ФАБРИКА

Создание конкретных классов:

```
public class VanFactory implements VehicleFactory {
    public Body createBody() {
        return new VanBody();
    }

    public Chassis createChassis() {
        return new VanChassis();
    }

    public Windows createWindows() {
        return new VanWindows();
    }
}
```

АБСТРАКТНАЯ ФАБРИКА

Создание конкретных классов:

```
public class CarFactory implements VehicleFactory {
    public Body createBody() {
        return new CarBody();
    }

    public Chassis createChassis() {
        return new CarChassis();
    }

    public Windows createWindows() {
        return new CarWindows();
    }
}
```

АБСТРАКТНАЯ ФАБРИКА

Создание транспортного средства из совместимых между собой деталей:

```
public void createVehicle(VehicleFactory vehicleFactory) {
    Body vehicleBody = vehicleFactory.createBody();
    Chassis vehicleChassis = vehicleFactory.createChassis();
    Windows vehicleWindows = vehicleFactory.createWindows();
    ...
}
```

АБСТРАКТНАЯ ФАБРИКА

Используйте паттерн абстрактная фабрика, когда:

- Система не должна зависеть от того, как создаются, компонуются и представляются входящие в неё объекты.
- Входящие в семейство взаимосвязанные объекты должны использоваться вместе и вам необходимо обеспечить выполнение этого ограничения.
- Система должна конфигурироваться одним из семейств составляющих её объектов.
- Требуется предоставить библиотеку объектов, раскрывая только их интерфейсы, но не реализацию.

АБСТРАКТНАЯ ФАБРИКА

Плюсы:

- изолирует конкретные классы;
- упрощает замену семейств продуктов;
- гарантирует сочетаемость продуктов;

Минусы:

- сложно добавить поддержку нового вида продуктов.

Шаблон Строитель (Builder): Отделяет конструирование сложного объекта от его представления, так что в результате одного и того же процесса конструирования могут получаться разные представления

BUILDER

Необходимо написать код по сборке автомобиля в различных комплектациях:

```
public interface VehicleBuilder {
    void buildBody();
    void buildBoot();
    void buildChassis();
    void buildPassengerArea();
    void buildClimateControlSystem();
    void buildSunroof();
    void buildWindows();

    Vehicle getVehicle();
}
```

```
public interface VehicleDirector {
    Vehicle build(VehicleBuilder builder);
}
```

BUILDER

```
public class CarBuilder implements VehicleBuilder {
    private final Vehicle vehicle = new Car();

    public void buildBody() {
        // Add body to car
    }

    public void buildBoot() {
        // Add boot to car
    }
……
    public Vehicle getVehicle() {
        return vehicle;
    }
}
```

BUILDER

```
public class VanBuilder implements VehicleBuilder {
    private Vehicle van = new Van();

    public void buildBody() {
        // Add body to van
    }

    public void buildBoot() {
       // Nothing
    }
…
     public Vehicle getVehicle() {
        return van;
    }
}
```

```
public class StandardDirector implements VehicleDirector {

    public Vehicle build(VehicleBuilder builder) {
        builder.buildChassis();
        builder.buildBody();
        builder.buildPassengerArea();
        builder.buildBoot();
        builder.buildWindows();
        return builder.getVehicle();
    }
}
```

BUILDER

```
public class LuxeDirector implements VehicleDirector {

    public Vehicle build(VehicleBuilder builder) {
        builder.buildChassis();
        builder.buildBody();
        builder.buildPassengerArea();
        builder.buildBoot();
        builder.buildWindows();
        builder.buildClimateControlSystem();
        builder.buildSunroof();
        return builder.getVehicle();
    }
}
```

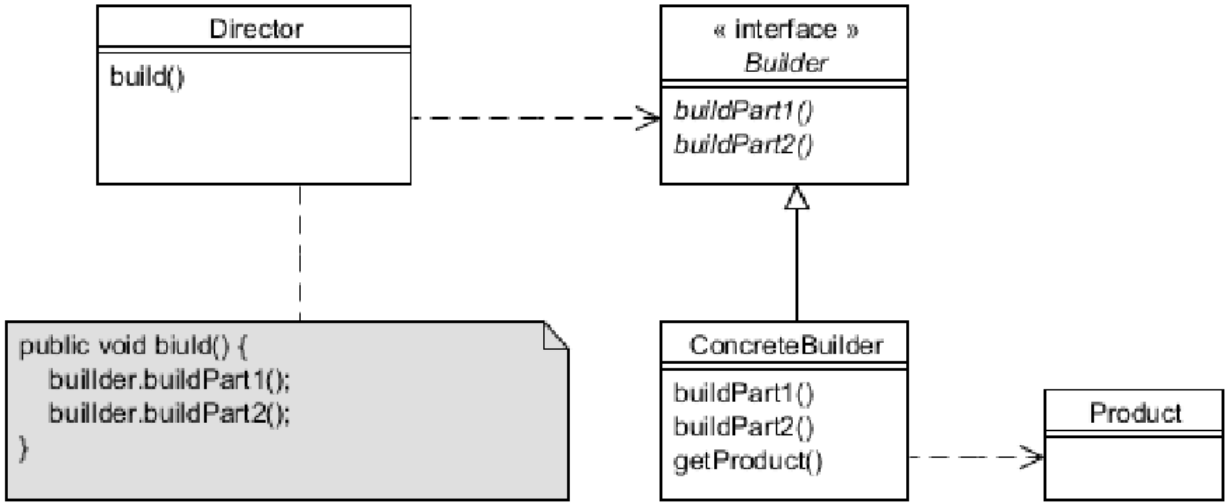
BUILDER

```
VehicleBuilder carBuilder = new CarBuilder();
VehicleBuilder vanBuilder = new VanBuilder();

VehicleDirector standardDirector = new StandardDirector();
Vehicle standardCar = standardDirector.build(carBuilder);
Vehicle standardVan = standardDirector.build(vanBuilder);

VehicleDirector luxeDirector = new LuxeDirector();
Vehicle luxeCar = luxeDirector.build(carBuilder);
Vehicle luxeVan = luxeDirector.build(vanBuilder);
```

BUILDER


Director
build()
« interface »
Builder
buildPart1()
buildPart2()
public void biuld() {
buillder.buildPart1();
buillder.buildPart2();
}
ConcreteBuilder
buildPart1()
buildPart2()
getProduct()
Product

BUILDER

Используйте паттерн строитель, когда:

- алгоритм создания сложного объекта не должен зависеть от того, из каких частей состоит объект и как они стыкуются между собой;
- процесс конструирования должен обеспечивать различные представления конструируемого объекта.

Плюсы:

- позволяет изменять внутреннее представление продукта;
- изолирует код, реализующий конструирование и представление
- дает более тонкий контроль над процессом конструирования

Фабричный Метод (Factory method): Определяет интерфейс для создания объекта и позволяет подклассам принимать решение экземпляр какого класса создавать.

FACTORY METHOD

Две разновидности фабричных методов:

- параметризованные фабричные методы
- метод делегирует создание объектов наследникам родительского класса

FACTORY METHOD

Параметризованный фабричный метод

```
public enum Category {
    CAR, VAN;
}
```

```
public class VehicleFactory {
    public Vehicle createVehicle(Category category) {
        if (Category.CAR == category) {
            return new Car();
        } else {
            return new Van();
        }
    }
}
```

FACTORY METHOD

Делегирование наследникам

```
public interface Iterator<E> {
        boolean hasNext();
        E next();
        …
}
```

```
public interface List<E> extends Collection<E> {
    …
    Iterator<E> iterator();
    …
}
```

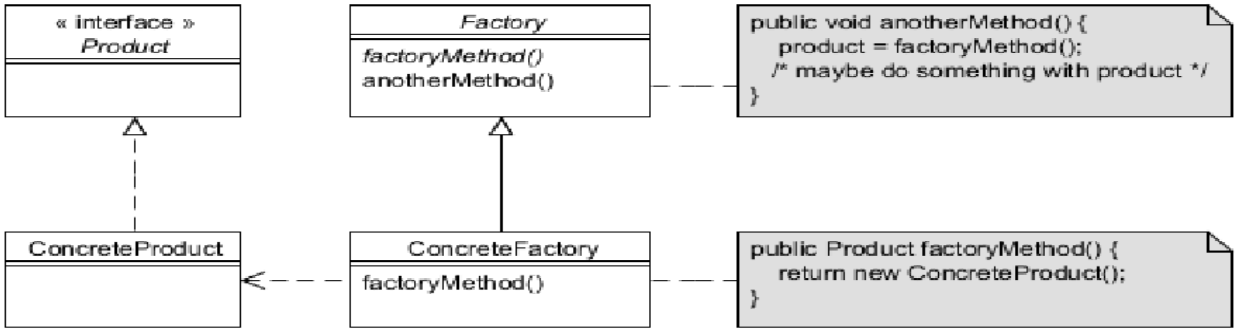
FACTORY METHOD, ДЕЛЕГИРОВАНИЕ НАСЛЕДНИКАМ

Реализация List для работы с большими объемами данных, информацию хранит в файле

```
public class FileList<E> implements List {
    private final File file;

…
    public Iterator iterator() {
        return new FileIterator(file);
    }
…
}
```

FACTORY METHOD



FACTORY METHOD

Используйте паттерн фабричный метод, когда:

- класс не имеет информации о том, какой тип объекта он должен создать
- класс передает ответственность по созданию объектов наследникам;
- необходимо создать объект в зависимости от входящих данных.

Прототип (Prototype): Позволяет создавать новые объекты копируя существующий прототип.

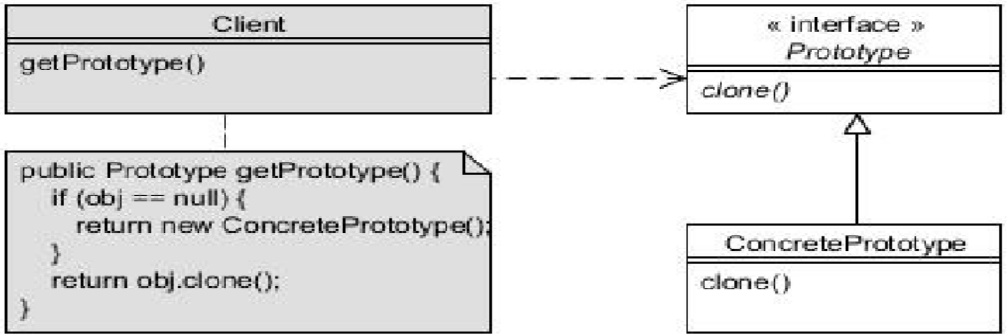
PROTOTYPE

```
public interface Cloneable {
}


 public class ArrayList<E> extends AbstractList<E>
         implements …, Cloneable, … {

     /**
      * Returns a shallow copy of this <tt>ArrayList</tt> instance.  (The
      * elements themselves are not copied.)
      *
      * @return a clone of this <tt>ArrayList</tt> instance
      */
     public Object clone() {
         ...
     }
 }
```

PROTOTYPE


Client
getPrototype()
public Prototype getPrototype() {
if (obj == null) {
return new ConcretePrototype();
}
return obj.clone();
}
« interface »
Prototype
clone()
ConcretePrototype
clone()

Шаблон Одиночка (Singleton): Гарантирует что у класса есть только один экземпляр и предоставляет глобальную точку доступа к нему.

SINGLETON

```
public class SerialNumberGenerator {
    private static final SerialNumberGenerator INSTANCE =
                new SerialNumberGenerator();
    private final AtomicInteger count = new AtomicInteger(0);

    private SerialNumberGenerator() {
    }

    public static SerialNumberGenerator getInstance() {
        return INSTANCE;
    }

    public int getNextSerial() {
        return count.incrementAndGet();
    }
}
```

SINGLETON

```
SerialNumberGenerator generator = SerialNumberGenerator.getInstance();
System.out.println("next serial: " + generator.getNextSerial());
System.out.println("next serial: " + generator.getNextSerial());
System.out.println("next serial: " + generator.getNextSerial());
```

SINGLETON

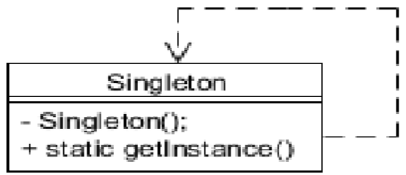
```
public enum SerialNumberGenerator {
    INSTANCE;

    private final AtomicInteger count = new AtomicInteger(0);

    public int getNextSerial() {
        return count.incrementAndGet();
    }
}
```

SINGLETON

```
System.out.println("next serial: " +
        SerialNumberGenerator.INSTANCE.getNextSerial());
System.out.println("next serial: " +
        SerialNumberGenerator.INSTANCE.getNextSerial());
System.out.println("next serial: " +
        SerialNumberGenerator.INSTANCE.getNextSerial());
```


Singleton
- Singleton();
+ static getInstance()

SINGLETON

Используйте паттерн одиночка, когда:

- должен быть ровно один экземпляр некоторого класса, легко доступный всем клиентам;
- единственный экземпляр должен расширяться путем порождения подклассов, и клиентам нужно иметь возможность работать с расширенным экземпляром без модификации своего кода.

ЛИТЕРАТУРА

- Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес, Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб: «Питер», 2007.