Оглавление

[Abstract factory (абстрактная фабрика, порождающий) 3](#_Toc67657232)

[Adapter (адаптер, структурный) 6](#_Toc67657233)

[Bridge (мост, структурный) 7](#_Toc67657234)

[Builder (строитель, порождающий) 9](#_Toc67657235)

[Composite (компоновщик, структурный) 13](#_Toc67657236)

[Decorator (декоратор, структурный) 14](#_Toc67657237)

[Factory method (фабричный метод, порождающий) 16](#_Toc67657238)

[Prototype (прототип, порождающий) 17](#_Toc67657239)

[Proxy (заместитель, структурный) 18](#_Toc67657240)

[Singleton (одиночка, порождающий) 20](#_Toc67657241)

[Адаптер, декоратор и прокси (обобщение) 21](#_Toc67657242)

[Фасад (facade) 21](#_Toc67657243)

[Приспособленец (flyweight) 23](#_Toc67657244)

[Примеры поведенческих шаблонов проектирования 24](#_Toc67657245)

[Цепочка обязанностей (chain of responsibility) 24](#_Toc67657246)

[Итератор (iterator) 26](#_Toc67657247)

[Шаблонный метод (template method) 27](#_Toc67657248)

[Команда (command) 29](#_Toc67657249)

[Интерпретатор (interpreter) 31](#_Toc67657250)

[Посредник (mediator) 32](#_Toc67657251)

[Хранитель (memento) 33](#_Toc67657252)

[Наблюдатель (observer) 35](#_Toc67657253)

[Состояние (state) 36](#_Toc67657254)

[Стратегия (strategy) 38](#_Toc67657255)

[Посетитель (visitor) 39](#_Toc67657256)

[Антипаттерн (anti-pattern) 41](#_Toc67657257)

[Dependency Injection (внедрение зависимости) 44](#_Toc67657258)

[Паттерны в Spring Framework 45](#_Toc67657259)

[Паттерны в Hibernate 45](#_Toc67657260)

# Abstract factory (абстрактная фабрика, порождающий)

Позволяет создавать семейства связанных объектов разного типа при помощи одного интерфейса.

|  |  |
| --- | --- |
| 1.  2.  3.  4. | public interface Car {  String getPrice();  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | public class UsCarImpl implements Car {  @Override  public String getPrice() {  return "300 $";  }  } |
| 1.  2.  3.  4.  5.  6.  7. | public class UsLadaImpl implements Car {  @Override  public String getPrice() {  return "100 $";  }  } |
| 1.  2.  3.  4.  5.  6.  7. | public class UsPorsheImpl implements Car {  @Override  public String getPrice() {  return "200 $";  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | public class UaCarImpl implements Car {  @Override  public String getPrice() {  return "3000 грн";  }  } |
| 1.  2.  3.  4.  5.  6.  7. | public class UaLadaImpl implements Car {  @Override  public String getPrice() {  return "1000 грн";  }  } |
| 1.  2.  3.  4.  5.  6.  7. | public class UaPorsheImpl implements Car {  @Override  public String getPrice() {  return "2000 грн";  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | public interface CarPrice {  Car getLada();  Car getFerrari();  Car getPorshe();  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17. | public class RuCarPrice implements CarPrice {  @Override  public Car getLada() {  return new RuLadaImpl();  }  @Override  public Car getFerrari() {  return new RuCarImpl();  }  @Override  public Car getPorshe() {  return new RuPorsheImpl();  }  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17. | public class UsCarPrice implements CarPrice {  @Override  public Car getLada() {  return new UsLadaImpl();  }  @Override  public Car getFerrari() {  return new UsCarImpl();  }  @Override  public Car getPorshe() {  return new UsPorsheImpl();  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16. | public class Main {  public static void main(String[] args) {  // цены автомобилей в России  CarPrice ruCarPrice = new RuCarPrice();  ruCarPrice.getFerrari().getPrice();  ruCarPrice.getLada().getPrice();  ruCarPrice.getPorshe().getPrice();  // цены автомобилей в США  CarPrice usCarPrice = new UsCarPrice();  usCarPrice.getFerrari().getPrice();  usCarPrice.getLada().getPrice();  usCarPrice.getPorshe().getPrice();  }  } |

Плюсы:  
∙ гарантирует сочетаемость создаваемых продуктов;  
∙ реализует второй принцип SOLID (принцип открытости/закрытости);  
∙ упрощает добавление новых продуктов;  
∙ избавляет клиентский код от привязки к конкретным классам продуктов;  
∙ выделяет код производства продуктов в одно место, упрощая поддержку кода.

Минусы:  
∙ много дополнительных классов;  
∙ требует наличия всех типов продуктов в каждой вариации.

В Java — newInstance() (javax.xml.parsers  
.DocumentBuilderFactory).

# Adapter (адаптер, структурный)

Объект, который обеспечивает взаимодействие двух несовместимых (с разными интерфейсами) объектов.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | class PBank {  private int balance;  public PBank() {  balance = 100;  }  public void getBalance() {  System.out.println(  "PBank balance = " + balance);  }  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | class ABank {  private int balance;  public ABank() {  balance = 200;  }  public void getBalance() {  System.out.println(  "ABank balance = " + balance);  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | class PBankAdapter extends PBank {  private ABank abank;  public PBankAdapter(ABank abank) {  this.abank = abank;  }  public void getBalance() {  abank.getBalance();  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | public class AdapterTest {  public static void main(String[] args) {  PBank pbank = new PBank();  pbank.getBalance();  PBankAdapter abank =  new PBankAdapter(new ABank());  abank.getBalance();  }  } |

Arrays.asList(array) — создает List из массива; при изменении List или массива — изменения будет применяться к обоим контейнерам.

# Bridge (мост, структурный)

Разделяет реализацию и абстракцию (т. е. один класс наследуется от абстрактного, а второй реализует интерфейс), дает возможность изменять их свободно друг от друга. Делает конкретные классы независимыми от классов реализации интерфейса.

|  |  |
| --- | --- |
| 1.  2.  3.  4. | public interface Engine {  void setEngine();  } |
| 1.  2.  3.  4.  5.  6.  7. | public class SportEngine implements Engine {  @Override  public void setEngine() {  System.out.println("спорт");  }  } |
| 1.  2.  3.  4.  5.  6.  7. | public class UnknownEngine implements Engine {  @Override  public void setEngine() {  System.out.println("unknown");  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | public abstract class Car {  protected Engine engine;  public Car(Engine engine) {  this.engine = engine;  }  public void buildBase() {  System.out.println("Делаем корпус");  }  public abstract void setEngine();  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | public class SportCar extends Car {  public SportCar(Engine engine) {  super(engine);  }  @Override  public void setEngine() {  System.out.println(  "Двигатель спортивного автомобиля: ");  engine.setEngine();  }  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | public class UnknownCar extends Car {  public UnknownCar(Engine engine) {  super(engine);  }  @Override  public void setEngine() {  System.out.println(  "Двигатель обычного автомобиля: ");  engine.setEngine();  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13. | public class Main {  public static void main(String[] args) {  Car sportCar = new SportCar(  new SportEngine());  sportCar.setEngine();  System.out.println();  Car unknownCar = new UnknownCar(  new UnknownEngine());  unknownCar.setEngine();  }  } |

В java — newSetFromMap() (Collections).

# Builder (строитель, порождающий)

Интерфейс для создания сложного объекта.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16. | public class Car {  public void buildBase() {  System.out.println("Делаем корпус");  }  public void buildWheels() {  System.out.println("Ставим колесо");  }  public void buildEngine(Engine engine) {  System.out.println(  "Ставим движок "  + engine.getEngineType());  }  } |
| 1.  2.  3.  4. | public interface Engine {  String getEngineType();  } |
| 1.  2.  3.  4.  5.  6.  7.  8. | public class OneEngine implements Engine {  @Override  public String getEngineType() {  return "Первый двигатель";  }  } |
| 1.  2.  3.  4.  5.  6.  7.  8. | public class TwoEngine implements Engine {  @Override  public String getEngineType() {  return "Второй двигатель";  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4. | public interface Builder {  Car buildCar();  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16. | public class OneBuilderImpl implements Builder {  private Car car;  public OneBuilderImpl() {  car = new Car();  }  @Override  public Car buildCar() {  car.buildBase();  car.buildWheels();  Engine engine = new OneEngine();  car.buildEngine(engine);  return car;  }  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18. | public class TwoBuilderImpl implements Builder {  private Car car;  public TwoBuilderImpl() {  car = new Car();  }  @Override  public Car buildCar() {  car.buildBase();  car.buildWheels();  Engine engine = new OneEngine();  car.buildEngine(engine);  engine = new TwoEngine();  car.buildEngine(engine);  return car;  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15. | public class Build {  private Builder builder;  public Build(int i) {  if (i == 1) {  builder = new OneBuilderImpl();  } else if (i == 2) {  builder = new TwoBuilderImpl();  }  }  public Car buildCar() {  return builder.buildCar();  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | public class Main {  public static void main(String[] args) {  Build build = new Build(1);  build.buildCar();  System.out.println();  build = new Build(2);  build.buildCar();  }  } |

Плюсы:  
∙ позволяет создавать продукты пошагово;  
∙ позволяет использовать один и тот же код для создания различных продуктов;  
∙ изолирует сложный код сборки продукта от его основной бизнес-логики.

Минусы:  
∙ усложняет код программы из‑за введения дополнительных классов;  
∙ клиент будет привязан к конкретным классам строителей, т. к. в интерфейсе строителя может не быть метода получения результата.

В java — StringBuilder.

# Composite (компоновщик, структурный)

Объект, который объединяет в себе объекты, подобные ему самому (внутри коллекции).

|  |  |
| --- | --- |
| 1.  2.  3.  4. | public interface Car {  void draw(String color);  } |
| 1.  2.  3.  4.  5.  6.  7.  8. | public class SportCar implements Car {  @Override  public void draw(String color) {  System.out.println(  "Цвет спортивной машины: " + color);  }  } |
| 1.  2.  3.  4.  5.  6.  7.  8. | public class UnknownCar implements Car {  @Override  public void draw(String color) {  System.out.println(  "Цвет обычного автомобиля: " + color);  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19. | public class Drawing implements Car {  private List<Car> cars = new ArrayList<>();  @Override  public void draw(String color) {  for (Car car : cars) {  car.draw(color);  }  }  public void add(Car s) {  this.cars.add(s);  }  public void clear() {  System.out.println();  this.cars.clear();  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17. | public class Main {  public static void main(String[] args) {  Car sportCar = new SportCar();  Car unknownCar = new UnknownCar();  Drawing drawing = new Drawing();  drawing.add(sportCar);  drawing.add(unknownCar);  drawing.draw("green");  drawing.clear();  System.out.println();  drawing.add(sportCar);  drawing.add(unknownCar);  drawing.draw("white");  }  } |

В java — add(Component) (java.awt.Container).

# Decorator (декоратор, структурный)

Класс, расширяющий функциональность другого класса без использования наследования.

|  |  |
| --- | --- |
| 1.  2.  3.  4. | public interface Car {  void draw();  } |
| 1.  2.  3.  4.  5.  6.  7. | public class SportCar implements Car {  @Override  public void draw() {  System.out.println("Спортивная машина");  }  } |
| 1.  2.  3.  4.  5.  6.  7. | public class UnknownCar implements Car {  @Override  public void draw() {  System.out.println("Обычная машина");  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19. | public class Decorator implements Car {  private Car decorator;  private String color;  public Decorator(Car decorator, String color) {  this.decorator = decorator;  this.color = color;  }  @Override  public void draw() {  decorator.draw();  setColor();  }  private void setColor() {  System.out.println("Цвет: " + color);  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | public class Main {  public static void main(String[] args) {  Car sportCar = new SportCar();  Car blueUnknownCar = new Decorator(  new UnknownCar(), "красный");  sportCar.draw();  System.out.println();  blueUnknownCar.draw();  }  } |

java.io.InputStream, OutputStream, Reader и Writer.

# Facade (фасад, структрурный)

Работа с несколькими классами при помощи одного (использование в искомом классе поля с типами других классов).

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6. | public interface Car {  void start();  void stop();  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | public class Key implements Car {  @Override  public void start() {  System.out.println("вставить ключи");  }  @Override  public void stop() {  System.out.println("вытянуть ключи");  }  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | public class Engine implements Car {  @Override  public void start() {  System.out.println("запусить двигатель");  }  @Override  public void stop() {  System.out.println("отсановить двигатель");  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19. | class Facade {  private Key key;  private Engine engine;  public Facade() {  key = new Key();  engine = new Engine();  }  public void startCar() {  key.start();  engine.start();  }  public void stoptCar() {  key.stop();  engine.stop();  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | public class FacadeTest {  public static void main(String[] args) {  Facade facade = new Facade();  facade.startCar();  System.out.println();  facade.stoptCar();  }  } |

В Java — javax.faces.context  
.ExternalContext, который используется внутри ServletContext, HttpSession, HttpServletRequest, HttpServletResponse и т. д.

# Factory method (фабричный метод, порождающий)

Делегирует создание объектов наследникам родительского класса.

|  |  |
| --- | --- |
| 1.  2.  3.  4. | public interface OS {  void getOS();  } |
| 1.  2.  3.  4.  5.  6.  7. | public class LinuxOS implements OS {  @Override  public void getOS () {  System.out.println("Linux");  }  } |
| 1.  2.  3.  4.  5.  6.  7. | public class MacOS implements OS {  @Override  public void getOS () {  System.out.println("Mac");  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | public class FactoryMethod {  public OS getCurrentOS(String inputOs) {  OS os = null;  if (inputos.equals("linux")) {  os = new LinuxOS();  } else if (inputos.equals("mac")) {  os = new MacOS();  }  return os;  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | public class Main {  public static void main(String[] args) {  FactoryMethod factory = new FactoryMethod();  OS linuxOs = factory.getCurrentOS("linux");  linuxOs.getOS();  OS macOs = factory.getCurrentOS("mac");  macOs.getOS();  }  } |

Плюсы:  
∙ избавляет класс от привязки к конкретным классам продуктов;  
∙ выделяет код производства продуктов в одно место, упрощая поддержку кода;  
∙ упрощает добавление новых продуктов в программу;  
∙ реализует принцип открытости/закрытости.

Минус:  
∙ может привести к созданию больших параллельных иерархий классов, т. к. для каждого класса продукта надо создать свой подкласс создателя.

В java — toString() (Object).

# Prototype (прототип, порождающий)

Создание объекта не при помощи конструктора, а при помощи клонирования.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23. | public class Prototype {  private Type type;  public void setType(Type type) {  this.type = type;  }  public enum Type {  ONE, TWO;  }  public Prototype copy() {  return new Prototype();  }  @Override  public String toString() {  return "Prototype {" +  "type = " + type +  '}';  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | public class Main {  public static void main(String[] args) {  Prototype prototype = new Prototype();  prototype.setType(Prototype.Type.ONE);  System.out.println(prototype);  Prototype clone = prototype.copy();  clone.setType(Prototype.Type.TWO);  System.out.println(clone);  }  } |

Пример в java — метод clone() (Object).

# Proxy (заместитель, структурный)

Объект, который является посредником для другого объекта, т. е.  заместитель ограничивает/реализует доступ к объекту.

|  |  |
| --- | --- |
| 1.  2.  3.  4. | public interface Image {  void display();  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17. | public class RealImage implements Image {  private String file;  public RealImage(String file) {  this.file = file;  load(file);  }  @Override  public void display() {  System.out.println("просмотр " + file);  }  private void load(String file) {  System.out.println("загрузка " + file);  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16. | public class Proxy implements Image {  private String file;  private RealImage image;  public Proxy(String file) {  this.file = file;  }  @Override  public void display() {  if (image == null) {  image = new RealImage(file);  }  image.display();  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | public class Main {  public static void main(String[] args) {  Image image = new Proxy("test.jpg");  image.display();  System.out.println();  image.display();  }  } |

javax.persistence.PersistenceContext над EntityManager.

# Singleton (одиночка, порождающий)

Класс, который может иметь только один экземпляр.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24. | public class Singleton {  private static Singleton instance;  private String value;  private Singleton(String value) {  this.value = value;  }  public static Singleton getInstance(  String value) {  if (instance == null) {  instance = new Singleton(value);  }  return instance;  }  @Override  public String toString() {  return "Singleton {"  + "value = '" + value + '\''  + '}';  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17. | public class Main {  public static void main(String[] args) {  Singleton first =  Singleton.getInstance("Hello!");  Singleton second =  Singleton.getInstance("Goodbye!");  // Singleton {value = 'Hello!'}  System.out.println("first = " + first);  // Singleton {value = 'Hello!'}  System.out.println("second = " + second);  // true  System.out.println("first.equals(second) = "  + (first.equals(second)));  }  } |

getDesktop() (java.awt Desktop).

## Адаптер, декоратор и прокси (обобщение)

Если обобщить, то заместитель притворяется оригинальным классом, декоратор расширяет функциональность (умный заместитель), а адаптер заменяет интерфейс оного из несовместимого объекта.

Также можно добавить, что в Java паттерны адаптер и декоратор существуют из‑за отсутствия множественного наследования.

Приспособленец (flyweight) — вместо создания большого количества похожих объектов, объекты используются повторно (создание объекта внутри другого). Экономит память.

В Java — пул строк, а также метод valueOf(int) (java.lang.Integer, Boolean, Byte, Character, Short, Long и BigDecimal).

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | class Flyweight {  private int row;  public Flyweight(int row) {  this.row = row;  System.out.println("ctor: " + this.row);  }  void report(int col) {  System.out.print(" " + row + col);  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14. | class Factory {  private Flyweight[] pool;  public Factory(int maxRows) {  pool = new Flyweight[maxRows];  }  public Flyweight getFlyweight(int row) {  if (pool[row] == null) {  pool[row] = new Flyweight(row);  }  return pool[row];  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14. | public class FlyweightTest {  public static void main(String[] args) {  int rows = 5;  Factory theFactory = new Factory(rows);  for (int i = 0; i < rows; i++) {  for (int j = 0; j < rows; j++) {  theFactory.getFlyweight(i)  .report(j);  }  System.out.println();  }  }  } |

# Примеры поведенческих шаблонов проектирования

Цепочка обязанностей (chain of responsibility) — избегание жесткой зависимости отправителя запроса от получателя; запрос может быть обработан несколькими объектами.

Плюсы:  
∙ уменьшает зависимость между клиентом и обработчиками;  
∙ реализует 2 принципа SOLOD: единственной обязанности и открытости/закрытости.

Минусы:  
∙ Запрос может остаться никем не обработанным.

В Java — log() (java.util.logging.Logger).

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6. | interface Payment {  void setNext(Payment payment);  void pay();  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | class VisaPayment implements Payment {  private Payment payment;  public void setNext(Payment payment) {  this.payment = payment;  }  public void pay() {  System.out.println("Visa Payment");  }  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | class PayPalPayment implements Payment {  private Payment payment;  public void setNext(Payment payment) {  this.payment = payment;  }  public void pay() {  System.out.println("PayPal Payment");  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | public static void main(String[] args) {  Payment visaPayment = new VisaPayment();  Payment payPalPayment =  new PayPalPayment();  visaPayment.setNext(payPalPayment);  visaPayment.pay();  } |

Итератор (iterator) — объект, позволяющий последовательно обходить элементы объекта-агрегата (составного объекта), не раскрывая их внутреннего представления.

Плюсы:  
∙ упрощает классы хранения данных;  
∙ позволяет реализовать различные способы обхода структуры данных;  
∙ позволяет одновременно перемещаться по структуре данных в разные стороны.

Минусы:  
∙ не оправдан, если можно обойтись простым циклом.

В Java — все реализации java.util.Iterator.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6. | interface Iterator {  boolean hasNext();  Object next();  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22. | class Numbers {  public int num[] = {1 , 2, 3};  public Iterator getIterator() {  return new NumbersIterator();  }  private class NumbersIterator  implements Iterator {  int ind;  public boolean hasNext() {  if(ind < num.length) return true;  return false;  }  public Object next() {  if(this.hasNext()) return num[ind++];  return null;  }  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | public static void main(String[] args) {  Numbers numbers = new Numbers();  Iterator iterator = numbers.getIterator();  while (iterator.hasNext()) {  System.out.println(iterator.next());  }  } |

Шаблонный метод (template method) — определяет основу алгоритма (1: 7) и позволяет наследникам переопределять некоторые шаги алгоритма (2, 3), не изменяя его структуру в целом (1).

Плюсы:  
∙ облегчает переиспользование кода.

Минусы:

∙ ограничены скелетом алгоритма;  
∙ можем нарушить 3 принцип SOLID (подстановки Барбары Лисков);  
∙ с ростом количества шагов сложно поддерживать.

В Java — все не абстрактные методы java.io.InputStream, OutputStream, Reader и Writer.

1

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | abstract class Car {  abstract void startEngine();  abstract void stopEngine();  public final void start() {  startEngine();  stopEngine();  }  } |

2, 3

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | class OneCar extends Car {  public void startEngine() {  System.out.println("Start engine.");  }  public void stopEngine() {  System.out.println("Stop engine.");  }  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | class TwoCar extends Car {  public void startEngine() {  System.out.println("Start engine.");  }  public void stopEngine() {  System.out.println("Stop engine.");  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | public static void main(String[] args) {  Car car1 = new OneCar();  car1.start();  System.out.println();  Car car2 = new TwoCar();  car2.start();  } |

Команда (command) — позволяет инкапсулировать различные операции в отдельные объекты.

В Java — все реализации java.lang.Runnable.

|  |  |
| --- | --- |
| 1.  2.  3.  4. | interface Command {  void execute();  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | class StartCar implements Command {  Car car;  public StartCar(Car car) {  this.car = car;  }  public void execute() {  car.startEngine();  }  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | class StopCar implements Command {  Car car;  public StopCar(Car car) {  this.car = car;  }  public void execute() {  car.stopEngine();  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | class Car {  public void startEngine() {  System.out.println("запустить двигатель");  }  public void stopEngine() {  System.out.println(  "остановить двигатель");  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | class CarInvoker {  public Command command;  public CarInvoker(Command command){  this.command = command;  }  public void execute(){  this.command.execute();  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | public static void main(String[] args) {  Car car = new Car();  StartCar startCar = new StartCar(car);  StopCar stopCar = new StopCar(car);  CarInvoker carInvoker =  new CarInvoker(startCar);  carInvoker.execute();  } |

Интерпретатор (interpreter) — решает часто встречающуюся, но подверженную изменениям, задачу.

В Java — все подклассы java.text.Format.

|  |  |
| --- | --- |
| 1.  2.  3.  4. | interface Expression {  String interpret(Context context);  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | class LowerExpression implements Expression {  private String s;  public LowerExpression(String s) {  this.s = s;  }  public String interpret(Context context) {  return context.getLowerCase(s);  }  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | class UpperExpression implements Expression {  private String s;  public UpperExpression(String s) {  this.s = s;  }  public String interpret(Context context) {  return context.getUpperCase(s);  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | class Context {  public String getLowerCase(String s){  return s.toLowerCase();  }  public String getUpperCase(String s){  return s.toUpperCase();  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | public static void main(String[] args) {  String str = "TesT";  Context context = new Context();  Expression lowerExpression =  new LowerExpression(str);  str = lowerExpression.interpret(context);  System.out.println(str);  Expression upperExpression =  new UpperExpression(str);  str = upperExpression.interpret(context);  System.out.println(str);  } |

Посредник (mediator) — обеспечивает взаимодействие множества объектов, формируя при этом слабую связанность и избавляя объекты от необходимости явно ссылаться друг на друга.

В java — execute() (java.util.concurrent.Executor).

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | class Mediator {  public static void sendMessage(  User user,  String msg) {  System.out.println(  user.getName() + ": " + msg);  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15. | class User {  private String name;  public User(String name){  this.name = name;  }  public String getName() {  return name;  }  public void sendMessage(String msg){  Mediator.sendMessage(this, msg);  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6. | public static void main(String[] args) {  User user1 = new User("user1");  User user2 = new User("user2");  user1.sendMessage("message1");  user2.sendMessage("message2");  } |

Хранитель (memento) — позволяет, не нарушая инкапсуляцию, зафиксировать и сохранить внутренние состояния объекта так, чтобы позднее восстановить его в этих состояниях.

В Java — все реализации java.io.Serializable.

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30. | class User {  private String name;  private int age;  public User(String name, int age) {  this.name = name;  this.age = age;  System.out.println(String.format(  "create: name = %s, age = %s",  name,  age));  }  public Memento save(){  System.out.println(String.format(  "save: name = %s, age = %s",  name,  age));  return new Memento(name, age);  }  public void restore(Memento memento){  name = memento.getName();  age = memento.getAge();  System.out.println(String.format(  "restore: name = %s, age = %s",  name,  age));  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12. | class SaveUser {  private List<Memento> list =  new ArrayList<Memento>();  public void add(Memento memento){  list.add(memento);  }  public Memento get(int ind){  return list.get(ind);  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17. | class Memento {  private String name;  private int age;  public Memento(String name, int age){  this.name = name;  this.age = age;  }  public String getName() {  return name;  }  public int getAge() {  return age;  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7. | public static void main(String[] args) {  SaveUser saveUser = new SaveUser();  User user1 = new User("Peter", 17);  User user2 = new User("Ian", 19);  saveUser.add(user1.save());  user1.restore(saveUser.get(0));  } |

Наблюдатель (observer) — определяет зависимость типа «один ко многим» между объектами т. о., что при изменении состояния одного объекта все зависящие от него оповещаются об этом событии.

В Java — все реализации java.util.EventListener (практически во всем Swing).

|  |  |
| --- | --- |
| 1.  2.  3.  4. | interface Observer {  void event(List<String> strings);  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9. | class Director implements Observer {  public void event(List<String> strings) {  System.out.println(  "The list of students has"  + " changed: "  + strings);  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18.  19.  20.  21.  22.  23.  24.  25.  26.  27.  28.  29.  30.  31. | class University {  private List<Observer> observers =  new ArrayList<Observer>();  private List<String> students =  new ArrayList<String>();  public void addStudent(String name) {  students.add(name);  notifyObservers();  }  public void removeStudent(String name) {  students.remove(name);  notifyObservers();  }  public void addObserver(Observer observer){  observers.add(observer);  }  public void removeObserver(  Observer observer) {  observers.remove(observer);  }  public void notifyObservers(){  for (Observer observer : observers) {  observer.event(students);  }  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | public static void main(String[] args) {  University university = new University();  Director director = new Director();  university.addStudent("Vaska");  university.addObserver(director);  university.addStudent("Anna");  university.removeStudent("Vaska");  } |

Состояние (state) — используется в тех случаях, когда во время выполнения программы объект должен менять свое поведение в зависимости от своего состояния.

В Java — execute() (javax.faces.lifecycle.LifeCycle) (контролируется FacesServlet, поведение зависит от текущей фазы (состояния) жизненного цикла JSF).

|  |  |
| --- | --- |
| 1.  2.  3.  4. | interface State {  void doAction();  } |
| 1.  2.  3.  4.  5.  6. | class StartPlay implements State {  public void doAction() {  System.out.println("start play");  }  } |
| 1.  2.  3.  4.  5.  6. | class StopPlay implements State {  public void doAction() {  System.out.println("stop play");  }  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | class PlayContext implements State {  private State state;  public void setState(State state){  this.state = state;  }  public void doAction() {  this.state.doAction();  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10. | public static void main(String[] args) {  PlayContext playContext =  new PlayContext();  State startPlay = new StartPlay();  State stopPlay = new StopPlay();  playContext.setState(startPlay);  playContext.doAction();  playContext.setState(stopPlay);  playContext.doAction();  } |

Стратегия (strategy) — определяет ряд алгоритмов позволяя взаимодействовать между ними. Алгоритм стратегии может быть изменен во время выполнения программы.

В Java — compare() (java.util.Comparator), выполненный среди других методов sort() (Collections).

|  |  |
| --- | --- |
| 1.  2.  3.  4. | interface Strategy {  void download(String file);  } |
| 1.  2.  3.  4.  5.  6.  7.  8. | class DownloadWindownsStrategy  implements Strategy {  public void download(String file) {  System.out.println(  "windows download: " + file);  }  } |
| 1.  2.  3.  4.  5.  6.  7. | class DownloadLinuxStrategy implements Strategy {  public void download(String file) {  System.out.println(  "linux download: " + file);  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11. | class Context {  private Strategy strategy;  public Context(Strategy strategy){  this.strategy = strategy;  }  public void download(String file){  strategy.download(file);  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | public static void main(String[] args) {  Context context = new Context(  new DownloadWindownsStrategy());  context.download("file.txt");  context = new Context(  new DownloadLinuxStrategy());  context.download("file.txt");  } |

Посетитель (visitor) — описывает операцию, которая выполняется над объектами других классов (при изменении класса Visitor нет необходимости изменять обслуживаемые классы).

В Java — java.nio.file.FileVisitor и SimpleFileVisitor.

|  |  |
| --- | --- |
| 1.  2.  3.  4. | interface Car {  void accept(Visitor visitor);  } |
| 1.  2.  3.  4.  5.  6. | class Engine implements Car {  public void accept(Visitor visitor) {  visitor.visit(this);  }  } |
| 1.  2.  3.  4.  5.  6. | class Whell implements Car {  public void accept(Visitor visitor) {  visitor.visit(this);  }  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16. | class SportCar implements Car {  Car[] cars;  public SportCar(){  cars = new Car[]{  new Engine(),  new Whell()};  }  public void accept(Visitor visitor) {  for (int i = 0; i < cars.length; i++) {  cars[i].accept(visitor);  }  visitor.visit(this);  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4.  5.  6.  7.  8. | interface Visitor {  void visit(SportCar sportCar);  void visit(Engine engine);  void visit(Whell whell);  } |
| 1.  2.  3.  4.  5.  6.  7.  8.  9.  10.  11.  12.  13.  14.  15.  16.  17.  18. | class CarVisitor implements Visitor {  public void visit(SportCar computer) {  print("car");  }  public void visit(Engine engine) {  print("engine");  }  public void visit(Whell whell) {  print("whell");  }  private void print(String string) {  System.out.println(string);  }  } |

|  |  |
| --- | --- |
| 1.  2.  3.  4. | public static void main(String[] args) {  Car computer = new SportCar();  computer.accept(new CarVisitor());  } |

# Антипаттерн (anti-pattern)

Подход к решению проблем, являющийся неэффективным, рискованным или непродуктивным.

Полтергейсты (poltergeists) — классы с ограниченной ответственностью и ролью в системе, чье единственное предназначение — передавать информацию в другие классы. Их эффективный жизненный цикл непродолжителен. Полтергейсты нарушают стройность архитектуры программного обеспечения, создавая избыточные (лишние) абстракции, они чрезмерно запутаны, сложны для понимания и трудны в сопровождении. Обычно такие классы задумываются как классы-контроллеры, которые существуют только для вызова методов других классов, зачастую в предопределенной последовательности.

Признаки появления и последствия антипаттерна:

∙ избыточные межклассовые связи;

∙ временные ассоциации;

∙ классы без состояния (содержащие только методы и константы);

∙ временные объекты и классы (с непродолжительным временем жизни);

∙ классы с единственным методом, который предназначен только для создания или вызова других классов посредством временной ассоциации;

∙ классы с именами методов в стиле «управления», такие как startProcess.

Типичные причины:

∙ отсутствие объектно-ориентированной архитектуры (архитектор не понимает объектно-ориентированной парадигмы);

∙ неправильный выбор пути решения задачи;

∙ предположения об архитектуре приложения на этапе анализа требований (до объектно-ориентированного анализа) могут также вести к проблемам на подобии этого антипаттерна.

Внесенная сложность (introduced complexity): необязательная сложность дизайна. Вместо одного простого класса выстраивается целая иерархия интерфейсов и классов. Типичный пример «Интерфейс — Абстрактный класс — Единственный класс реализующий интерфейс на основе абстрактного».

Инверсия абстракции (abstraction inversion): Сокрытие части функциональности от внешнего использования, в надежде на то, что никто не будет его использовать.

Неопределенная точка зрения (ambiguous viewpoint): Представление модели без спецификации ее точки рассмотрения.

Большой комок грязи (big ball of mud): Система с нераспознаваемой структурой.

Божественный объект (god object): Концентрация слишком большого количества функций в одной части системы (классе).

Затычка на ввод данных (input kludge): Забывчивость в спецификации и выполнении поддержки возможного неверного ввода.

Раздувание интерфейса (interface bloat): Разработка интерфейса очень мощным и очень сложным для реализации.

Волшебная кнопка (magic pushbutton): Выполнение результатов действий пользователя в виде неподходящего (недостаточно абстрактного) интерфейса. Например, написание прикладной логики в обработчиках нажатий на кнопку.

Перестыковка (re-coupling): Процесс внедрения ненужной зависимости.

Дымоход (stovepipe system): Редко поддерживаемая сборка плохо связанных компонентов.

Состояние гонки (race hazard): непредвидение возможности наступления событий в порядке, отличном от ожидаемого.

Членовредительство (mutilation): Излишнее «затачивание» объекта под определенную очень узкую задачу таким образом, что он не способен будет работать с никакими иными, пусть и очень схожими задачами.

Сохранение или смерть (save or die): Сохранение изменений лишь при завершении приложения.

# Dependency Injection (внедрение зависимости)

Набор паттернов и принципов разработки программного обеспечения, которые позволяют писать слабосвязный код. В полном соответствии с принципом единой обязанности объект отдает заботу о построении требуемых ему зависимостей внешнему, специально предназначенному для этого общему механизму.

# Паттерны в Spring Framework

Singleton — Creating beans with default scope.

Factory — Bean Factory classes

Prototype — Bean scopes

Adapter — Spring Web and Spring MVC

Proxy — Spring Aspect Oriented Programming support

Template Method — JdbcTemplate, HibernateTemplate etc

Front Controller — Spring MVC DispatcherServlet

Data Access Object — Spring DAO support

Dependency Injection and Aspect Oriented Programming

# Паттерны в Hibernate

Domain Model — объектная модель предметной области, включающая в себя как поведение, так и данные.

Data Mapper — слой мапперов (Mappers), который передает данные между объектами и БД, сохраняя их независимыми друг от друга и себя.

Proxy — применяется для ленивой загрузки.

Factory — используется в SessionFactory.