S.O.L.I.D.(OCP)

SOLID (single responsibility, open-closed, Liskov substitution, interface segregation, dependency inversion)

Содержание

- Принцип закрытости/открытости OCP
- Паттерн Прототип(Prototype)
- Литература

Принципы

Инициал	Представляет	Название, понятие
S	SRP	**Принцип единственной ответственности** (***The Single Responsibility Principle***) Существует лишь одна причина, приводящая к изменению класса.
O	OCP	**Принцип открытости/закрытости** (***The Open Closed Principle***) «программные сущности … должны быть открыты для расширения, но закрыты для модификации.»
L	LSP	**Принцип подстановки Барбары Лисков** (***The Liskov Substitution Principle***) «объекты в программе должны быть заменяемыми на экземпляры их подтипов без изменения правильности выполнения программы.» (контрактное программирование).
I	ISP	**Принцип разделения интерфейса** (***The Interface Segregation Principle***) «много интерфейсов, специально предназначенных для клиентов, лучше, чем один интерфейс общего назначения.»
D	DIP	**Принцип инверсии зависимостей** (***The Dependency Inversion Principle***) «Зависимость на Абстракциях. Нет зависимости на что-то конкретное.»

Принцип закрытости/открытости OCP

Формулировка:

программные сущности (классы, модули, функции и т.д.) должны быть открыты для расширения, но закрыты для изменения

Какую цель мы преследуем, когда применяем этот принцип?

Программные проекты в течение свой жизни постоянно изменяются. Изменения могут возникнуть, например, из-за новых требований заказчика или пересмотра старых.

В конечном итоге потребуется изменить код в соответствии с текущей ситуацией.

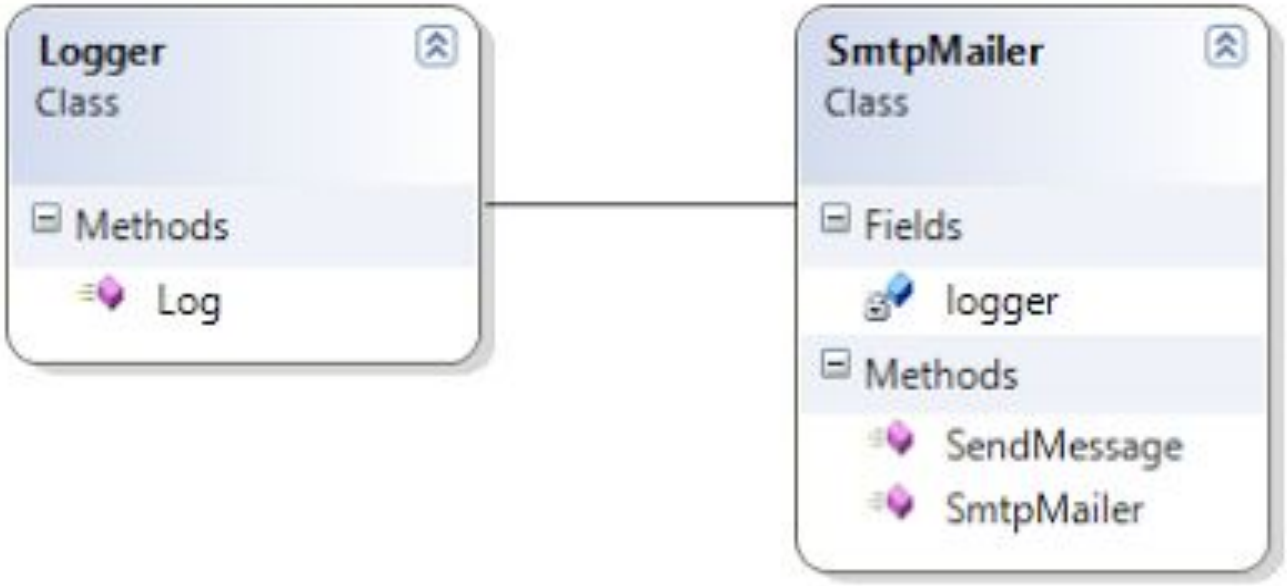
Целью является разработка системы, которая будет достаточно просто и безболезненно меняться. Другими словами, система должна быть гибкой.

Принцип открытости/закрытость как раз и дает понимание того, как создать проект достаточно гибкий в условиях постоянно меняющихся требований.

Без абстракций Проблема

Самый простой пример нарушения принципа открытости/закрытости – использование конкретных объектов без абстракций. Предположим, что у нас есть объект SmtpMailer.

Для записи своих действий он использует Logger, который записывает информацию в текстовые файлы.



Код классаSmtpMailer

```
public class Logger
{
   public void Log(String logText)   {
      // сохранить лог в файле
   }}
public class SmtpMailer
{
   private Logger logger;
   public SmtpMailer()
   {
      logger = new Logger();
   }
   public void SendMessage(String message)
   {
      // отсылка сообщения

      logger.Log(string.Format("Отправлено  ", message));
   }}
```

Изменившаяся ситуация

И тоже самое происходит в других классах, которые используют Logger.

Такая конструкция вполне жизнеспособна до тех, пока мы не решим записывать лог SmptMailer'a в базу данных.

Для этого нам надо создать класс, который будет записывать все логи не в текстовый файл, а в базу данных.

```
public class DatabaseLogger
{
    public void Log(string logText)
    {
        // сохранить лог в базе данных
    }
}
```

Мы должны изменить класс SmptMailer из-за изменившегося бизнес-требования

Изменить класс SmptMailer

```
public class SmtpMailer
{
   private DatabaseLogger logger;

   public SmtpMailer()
   {
      logger = new DatabaseLogger();
   }

   public void SendMessage(String message)
   {
      // отсылка сообщения

      logger.Log(String.Format("Отправлено '", message));
   }
}
```

По принципу единственности ответственности не SmptMailer отвечает за логирование, почему изменения дошли и до него?

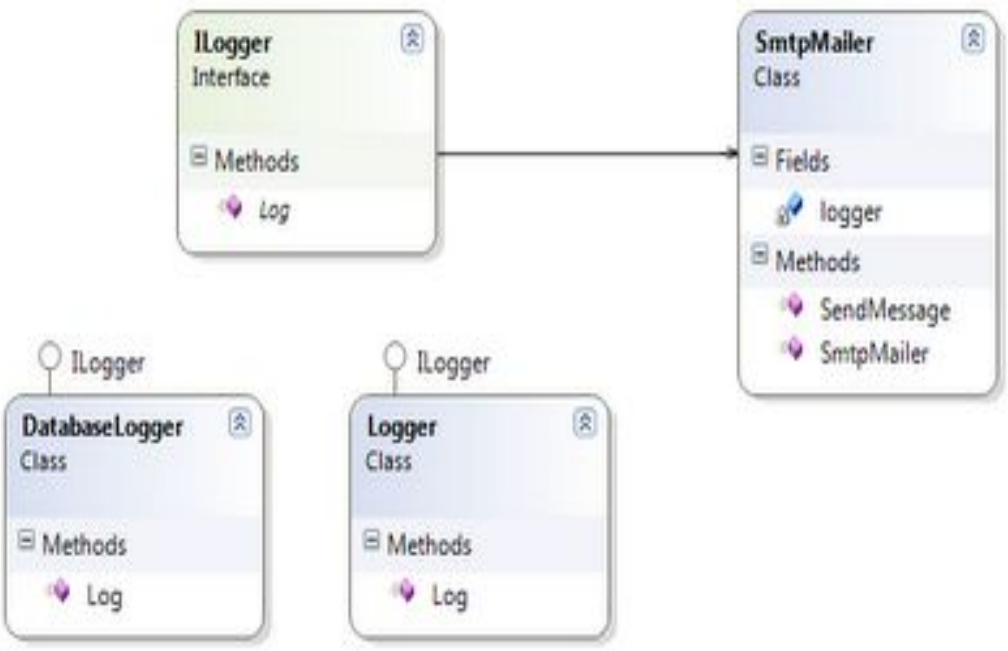
Потому что нарушен наш принцип открытости/закрытости. SmptMailer не закрыт для модификации.

Нам пришлось его изменить, чтобы поменять способ хранения его логов.

Решение проблемы

В данном случае защитить SmtpMailer поможет выделение абстракции.

Пусть SmtpMailer зависит от **интерфейса** ILogger:



Код решения

```
public interface ILogger{
   void Log(String logText);}

public class Logger implements  ILogger{
   public void Log(String logText)
   {
      // сохранить лог в файле    }}

public class DatabaseLogger implements ILogger{
   public void Log(String logText)    {
      // сохранить лог в базе  данных    }}

public class SmtpMailer{
   private  ILogger logger;

   public SmtpMailer(ILogger logger)    {
      this.logger = logger;    }

   public void SendMessage(string message)    {
      // отсылка сообщения

      logger.Log(string.Format("Отправлено “  message));    }}
```

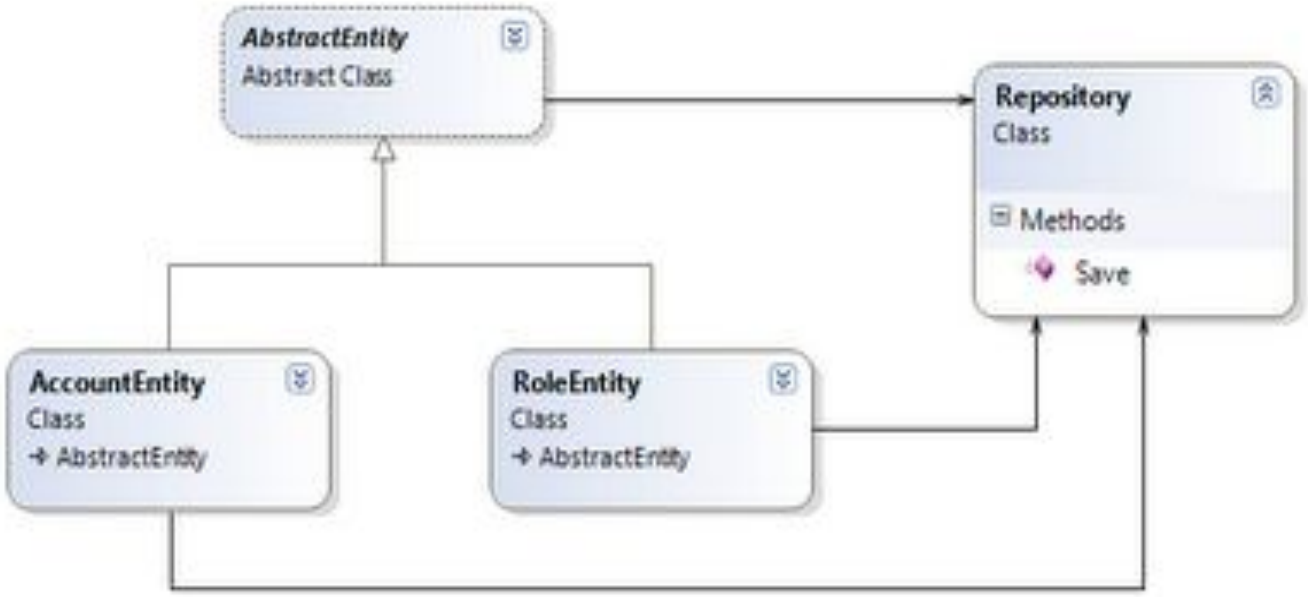
Теперь смена логики логирования уже не будет вести к модификации SmtpMailer'а.

Проверка типа абстракции

Этот пример - самое популярное нарушение проектирования.

У нас есть иерархия объектов с абстрактным родительским классом AbstractEntity и класс Repository, который использует абстракцию.

При этом вызывая метод Save у Repository мы строим логику в зависимости от типа входного параметра



Код

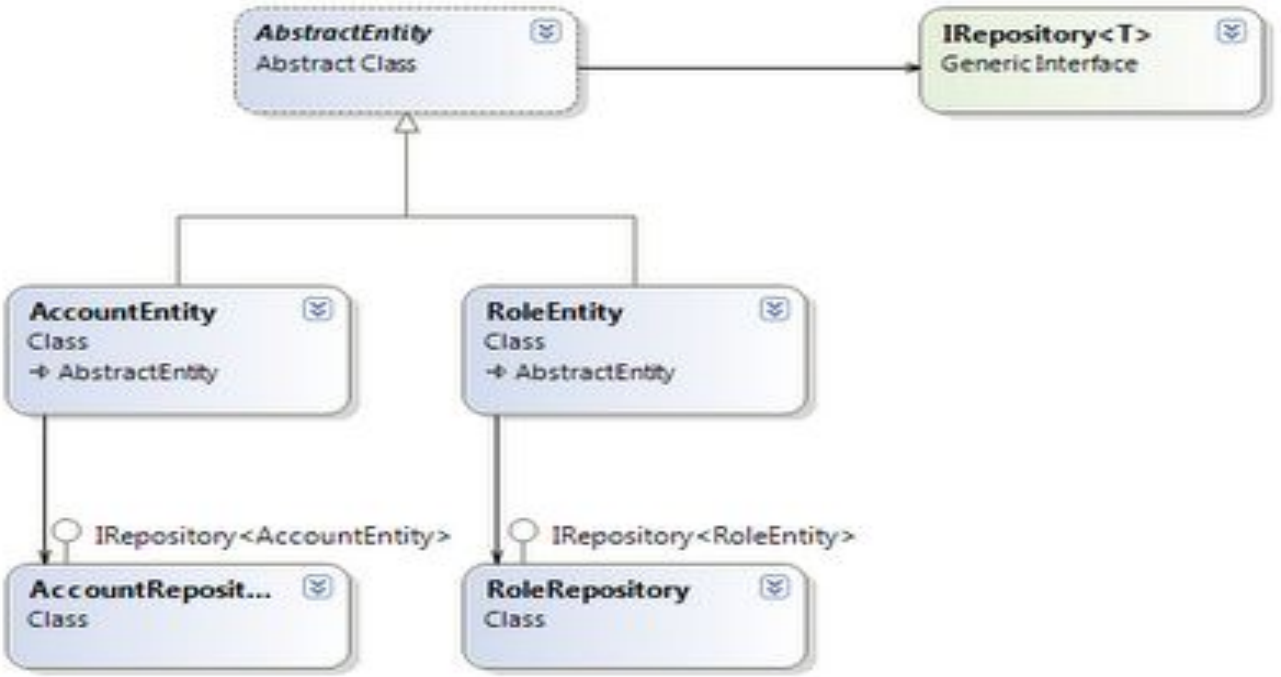
```
public abstract class AbstractEntity{}
public class AccountEntity extends AbstractEntity{}
public class RoleEntity extends AbstractEntity{}
public class Repository{
   public void Save(AbstractEntity entity)    {
      if (entity is AccountEntity)       {
         // специфические действия для AccountEntity
      }
      if (entity is RoleEntity)       {
         // специфические действия для RoleEntity
      }
   }
}
```

Из кода видно, что объект Repository придется менять каждый раз, когда мы добавляем в иерархию объектов с базовым классом AbstractEntity новых наследников или удаляем существующих. Условные операторы будут множится в методе Save и тем самым усложнять его.

Решение

Чтобы решить данную проблему, необходимо логику сохранения конкретных классов из иерархии AbstractEntity вынести в конкретные классы Repository.

Для этого мы должны выделить интерфейс IRepository и создать хранилища AccountRepository и RoleRepository



Код решения

```
public abstract class AbstractEntity{
}
public class AccountEntity extend AbstractEntity{
}
public class RoleEntity extends AbstractEntity{
}
public interface IRepository< T > {
    void Save(T entity);
}
public class AccountRepository implements IRepository<AccountEntity>{
    public void Save(AccountEntity entity)    {
        // специфические действия для AccountEntity
    }}
  public class RoleRepository implements IRepository<RoleEntity>{
    public void Save(RoleEntity abstractEntity)    {
        // специфические действия для RoleEntity    }
}
```

Паттерн Prototype

Встречаются ситуации, когда инициализация объекта некоторого класса занимает много ресурсов/времени.

В таком случае, для того чтобы избежать частого создания объектов этого класса путем инициализации, используют клонирование уже существующих объектов-прототипов, такое решение называют шаблоном прототип.

Прототип — это порождающий паттерн проектирования, который позволяет копировать объекты, не вдаваясь в подробности их реализации

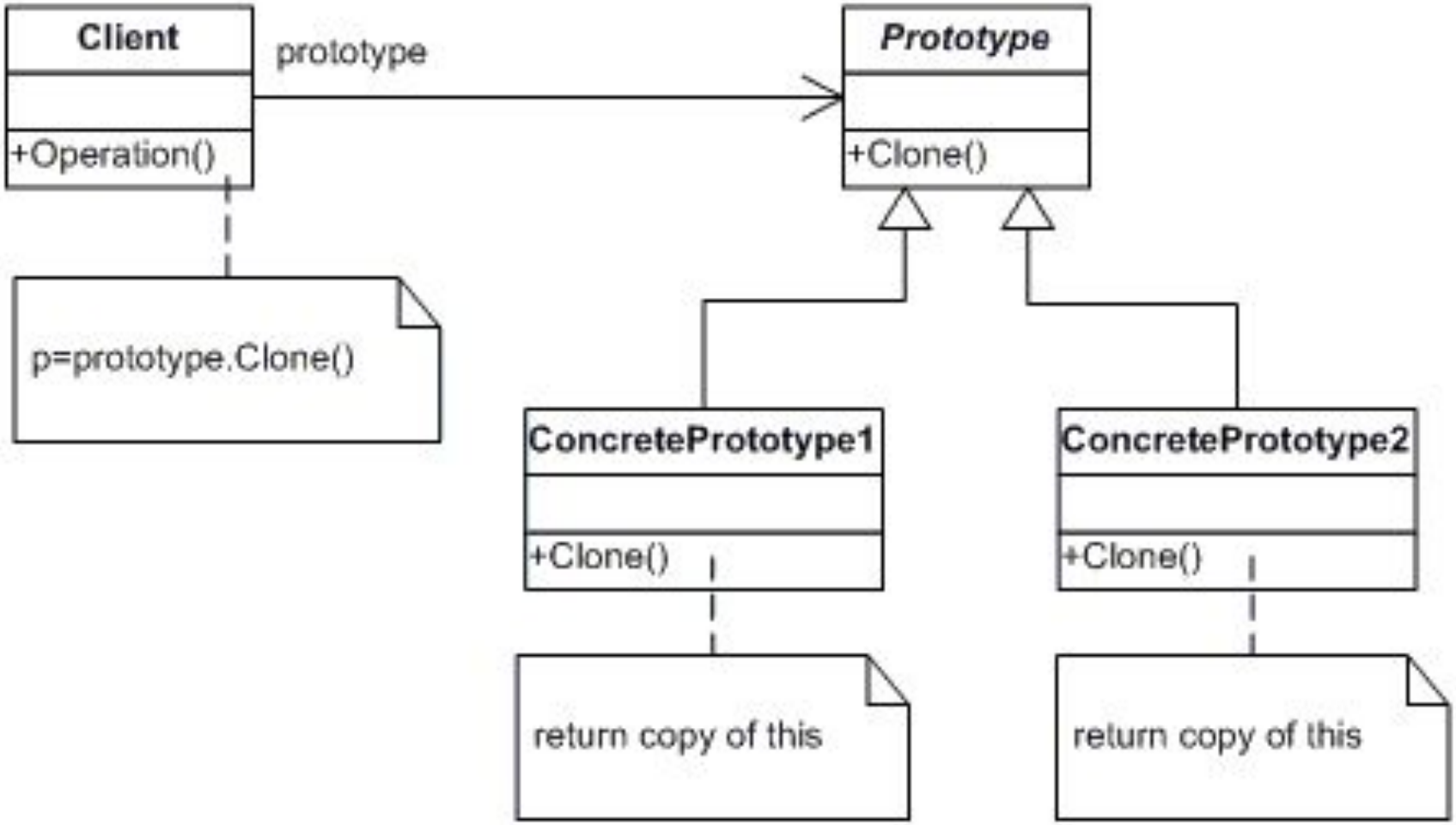
В Java клонирование объекта производится с помощью метода clone(), возвращающим всегда объект типа Object, поэтому требуется приводить результат к требуемому типу:

```
Jobj cloneObj = (Jobj) originalObj.clone();
```

Следует помнить, что клонируются только те объекты, которые реализуют интерфейс Cloneable, а объекты, которые клонированы быть не могут, при попытке клонирования выбрасывают CloneNotSupported Exception.

Кроме того, это protected метод, поэтому вызывать его следует внутри того класса, который содержит данный метод.

Схема шаблона



Пример

Представим что приложению требуется работать с данными из файла книги, создавая объект книги, определенный в классе Book.

Сама операция считывания данных из файла (ресурсоемкая операция) будет выполняться при инициализации книги.

Задачей является реализация клонирования объекта книги для того, чтобы инициализировать его только один раз.

Создаем класс Book, реализующий интерфейс Cloneable, в котором определяем метод clone(), возвращающий объект типа Book, а также описываем считывание файла в контейнер

Код примера

```
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.Vector;
 public class Book implements Cloneable{
   private Vector<String> content;// Контейнер содержимого книги
   public Book(String bookName){
       content = new Vector<String>(); // Инициализация контейнера
try {
         //Считываем файл bookName в контейнер
         BufferedReader file = new BufferedReader(new FileReader(bookName));
         String line;

         while ((line = file.readLine()) != null) {
            content.addElement(line);
         }
         file.close();
      } catch (IOException e) {
         System.out.println("An error while parsing book");
      }
   }
```

Код примера

```
//Определяем метод clone
    public Book clone() {
        Book clone = null;
        try {
            clone = (Book) super.clone();
        } catch (CloneNotSupportedException e) {
            System.out.println("An error cloning book obj:");
            e.printStackTrace();
        }
        return clone;
    }

    // Очистка книги
    public void empty(){
        content.clear();
    }

    // Возвращаем содержимое книги
    public Vector<String> getContent(){
        return content;
    }
}
```

Определен метод empty(), который будем вызывать в качестве примера операции с объектом книги

метод loadCache

Создан класс кэша BookCache, который будет инициализировать объект-прототип при помощи метода loadCache().
Когда клиентской части потребуется копия книги, получить ее можно будет при помощи метода loadCache:

```
import java.util.HashMap;
import java.util.Map;

public class BookCache {
   private static Map<String, Book> cache;

   public static Book getBook(String name){
      Book book = null;
      if(cache.containsKey(name) && cache.get(name)!=null)
         book = cache.get(name).clone(); //Извлекаем прототип из хранилиша и клонируем его

      return book;
   }

   public static void loadCache(){
      Book book = new Book("book.txt"); //Инициализируем прототип
      cache = new HashMap<String, Book>();
      cache.put("book.txt", book); // Сохраняем прототип в хранилище
   }
}
```

Класс для запуска теста

```
import java.util.Vector;

public class RunTestPrototype {
    public static void main(String[] args){
        BookCache.loadCache();

        // Создаем два экземпляра книги и получим их содержимое
        Book book1 = BookCache.getBook("book.txt");
        Book book2 = BookCache.getBook("book.txt");
        Vector<String> content1 = book1.getContent();
        Vector<String> content2 = book2.getContent();
        // Выведем содержимое в консоль
        log("Book 1: ");
        for(String line : content1){
            log(line);
        }
        log("Book 2: ");
        for(String line : content2){
            log(line);
        }
```

Класс для запуска теста

```
        // Теперь опустошим второй экзепляр книги, получим содержимое обеих книг еще раз
        log("Empty-->");
        book2.empty();
        content1 = book1.getContent();
        content2 = book2.getContent();
        // Выведем содержимое в консоль
        log("Book 1: ");
        for(String line : content1){
            log(line);
        }
        log("Book 2: ");
        for(String line : content2){
            log(line);
        }
    }

    private static void log(String msg){
        System.out.println(msg);
    }
}
```

Итог

Хотя мы очищали содержимое второй книги, содержимое первой книги также очищено.

Происходит это поскольку clone() создает неполную копию объекта книги: мы получаем копию ссылки на реальный объект прототипа в памяти, а не ссылку на копию объекта.

Поэтому любое изменение одной такой неполной копии отражается на других копиях.

Заметьте, что в классе Book в методе empty мы изменяем объект вектора, одинаковый для обеих копий:

```
public void empty(){
    content.clear();
}
```

Если хотя бы создать в этом месте новый пустой объект, то content первой копии останется неизменным, поскольку во второй книге ссылка content будет вести на новый объект:

```
public void empty(){
    content = new Vector<String>();
}
```

Пример 2

Если требуется создавать полные копии прототипа, то следует использовать возможности интерфейса Serializable: класс сериализуем, то мы можем записать его как поток байтов и восстановить класс из потока байтов обратно.

В классе Book определим метод deepClone(), в котором выведем объект в поток вывода, а затем считываем байты обратно.

Метод clone() удаляем за ненадобностью. Естественно, добавим указатель на интерфейс Serializable.

Код

```
import java.io.BufferedReader;
import java.io.ByteArrayInputStream;
import java.io.ByteArrayOutputStream;
import java.io.FileReader;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.Serializable;
import java.util.Vector;
//делаем класс сериализуемым
public class Book implements Cloneable, Serializable{
   private Vector<String> content;
   public Book(String bookName){
       content = new Vector<String>();
      try {
         BufferedReader file = new BufferedReader(new FileReader(bookName));
         String line;
          while ((line = file.readLine()) != null) {
            content.addElement(line);          }
         file.close();       } catch (IOException e) {
         System.out.println("An error while parsing book");       }
   }
```

КОД

```
//полное копирование
   public Object deepClone()    {
      try{
         // Операции с потоками
         ByteArrayOutputStream b = new ByteArrayOutputStream();
         ObjectOutputStream out = new ObjectOutputStream(b);
         out.writeObject(this);
         ByteArrayInputStream bIn = new ByteArrayInputStream(b.toByteArray());
         ObjectInputStream oi = new ObjectInputStream(bIn);
         return (oi.readObject());
      } catch (Exception e){
         System.out.println("exception:"+e.getMessage());
         return null;
      }   }
   // Очищаем содержимое
   public void empty(){
      content.clear();    }
      public Vector<String> getContent(){
      return content;    }}
```

поменяем создание копии в BookCache

```
import java.util.HashMap;
import java.util.Map;

public class BookCache {
    private static Map<String, Book> cache;

    public static Book getBook(String name){
        Book book = null;
        if(cache.containsKey(name) && cache.get(name)!=null)
            book = (Book) cache.get(name).deepClone(); //Возвращаем полную копию

        return book;
    }

    public static void loadCache(){
        Book book = new Book("book.txt");
        cache = new HashMap<String, Book>();
        cache.put("book.txt", book);
    }
}
```

Запускаем тест опять и видим, что теперь наши копии совершенно независимы:

3.8. Дублирование объектов

Метод **Object.clone** помогает производить в ваших классах дублирование объектов. При дублировании возвращается новый объект, исходное состояние которого копирует состояние объекта, для которого был вызван метод **clone.** Все последующие изменения, вносимые в объект-дубль, не изменяют состояния исходного объекта.

При написании метода **clone** следует учитывать три основных момента:

Для нормальной работы метода clone необходимо реализовать интерфейс **Cloneable.**

Метод **Object.clone** выполняет простое дублирование, заключающееся в копировании всех полей исходного объекта в новый объект.

Исключение **CloneNotSupportedException** сигнализирует о том, что **метод clone** данного класса не должен вызываться.

Дублирование объектов

Существует четыре варианта отношения класса к методу **clone**:

1. Класс поддерживает **clone**. Такие классы реализуют **Cloneable**
2. Класс **условно поддерживает clone**. Такой класс может представлять собой коллекцию (набор объектов), которая в принципе может дублироваться, но лишь при условии, что дублируется все ее содержимое. Такие классы реализуют **Cloneable,** но при этом допускают возникновение в методе clone исключения **CloneNotSupportedException**, которое может быть получено от других объектов при попытке их дублирования во время дублирования коллекции.
3. Класс разрешает поддержку **clone в подклассах**, но не объявляет об этом открыто. Такие классы не реализуют **Cloneable**, но обеспечивают реализацию clone для правильного дублирования полей, если реализация по умолчанию оказывается неправильной.
4. Класс **запрещает clone**. Такие классы не реализуют **Cloneable**, а метод **clone** в них всегда запускает исключение **CloneNotSupportedException**.

Дублирование объектов

Создать дублируемый класс — объявить о реализации в нем **интерфейса Cloneable**:

public class MyClass extends AnotherClass implements Cloneable{ // ...}

Метод **clone** в интерфейсе **Cloneable** имеет атрибут **public,** следовательно, метод MyClass.clone, унаследованный от Object, также будет public.

После такого объявления можно дублировать объекты MyClass. Дублирование в данном случае выполняется тривиально — Object.clone копирует все поля MyClass в новый объект и возвращает его.

Большинство классов является дублируемыми.

Во многих случаях реализация, принятая по умолчанию, не подходит, поскольку при ее выполнении происходит нежелательное размножение ссылок на объекты.

В таких случаях необходимо переопределить метод **clone** и исправить его поведение. По умолчанию значение каждого поля исходного объекта присваивается аналогичному полю нового объекта

Однако нередко требуется, чтобы ссылки внутри исходного объекта и дубликата были разными, — вероятно, ситуация, при которой дубликат может изменить содержимое массива в исходном объекте или наоборот, окажется нежелательной.

Дублирование объектов

Предположим, имеется простой стек, содержащий целые числа:

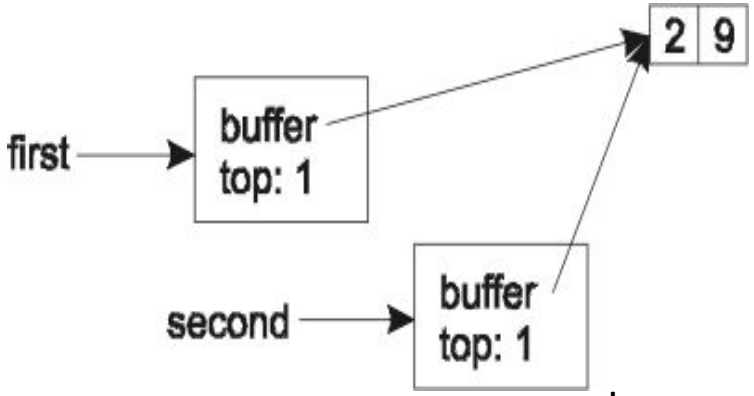
```
public class IntegerStack implements Cloneable {
      private int[] buffer;
private int top;
      public IntegerStack(int maxContents) {;
        buffer = new int[maxContents];
        top = -1;      }
      public void push(int val) {
      buffer[++top] = val;          }
```

Теперь рассмотрим фрагмент программы, который создает объект Integer Stack, заносит в него данные и затем дублирует:

```
IntegerStack first = new IntegerStack(2);
first.push(2);
first.push(9);
IntegerStack second = (IntegerStack)first.clone();
```

При использовании метода **clone**, принятого по умолчанию, данные в памяти будут выглядеть следующим образом:

Клонирование по - умолчанию



Выход заключается в переопределении метода clone и создании в нем отдельной копии массива:

```
public Object clone() {
    try {
IntegerStack nObj = (IntegerStack)super.clone();
nObj.buffer = (int[])buffer.clone();
return nObj;
} catch (CloneNotSupportedExeption e) {
// Не может произойти - метод clone() поддерживается
// как нашим классом, так и массивами
throw new InternalError(e.toString());     }}
```

Литература

- Э. Гамма Р. Хелм Р. Джонсон Дж. Влиссидес Design Patterns Elements of Reusable Object-Oriented Software p.121