# Инверсия зависимостей (Dependency Inversion)

# Качество архитектуры

- Плохой дизайн архитектуры:
  - Жесткость - сопротивление изменениям
    - Изменение в одном модуле системы, влечет за собой необходимость внесения каскадных изменений в другие модули
  - Хрупкость - внесение изменений в один модуль приводит к поломке другого модуля, который, на первый взгляд, не связан с тем, который мы меняли
  - Монолитность - сопротивление повторному использованию кода
- Хороший дизайн архитектуры должен быть гибким, устойчивым, и приспособленным к повторному использованию

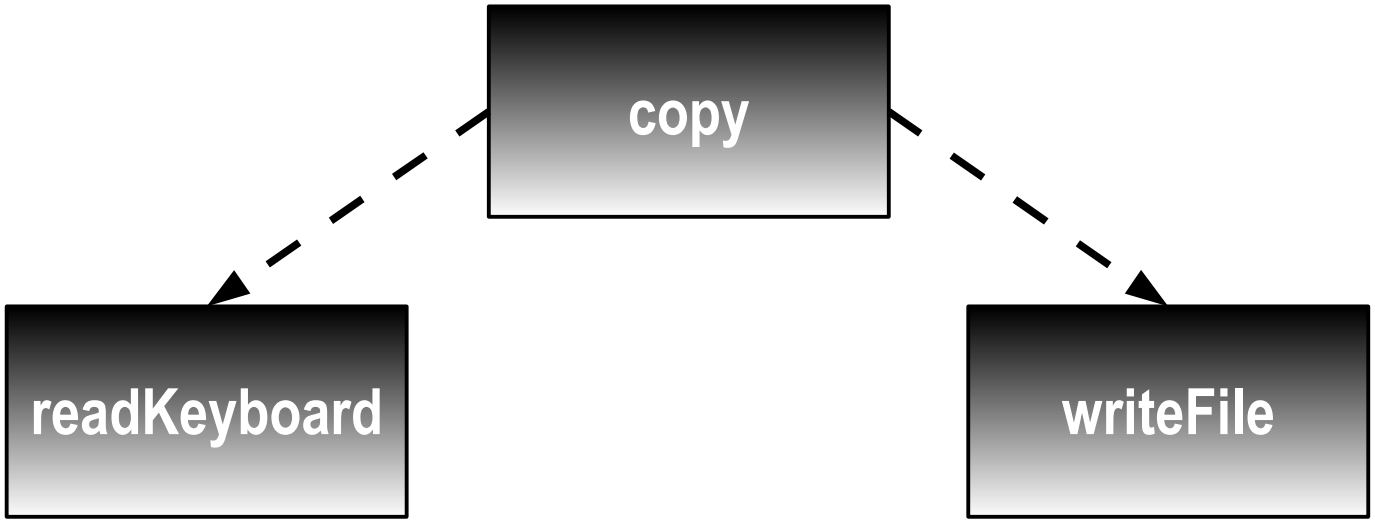
# Принцип инверсии зависимостей

- Инверсия зависимости – особый вид IoC, применяется для удаления зависимостей
  - Зависимости между классами превращаются в ассоциации между объектами
- 1. Модули верхнего уровня не должны зависеть от модулей нижнего уровня. Оба типа модулей должны зависеть от абстракций
  - В процедурном программировании высокоуровневые модули вызывают низкоуровневые => низкая гибкость и закостенелость дизайна
- 2. Абстракция не должна зависеть от реализации. Реализация должна зависеть от абстракции

# Пример инверсии зависимостей

- Модуль copy:

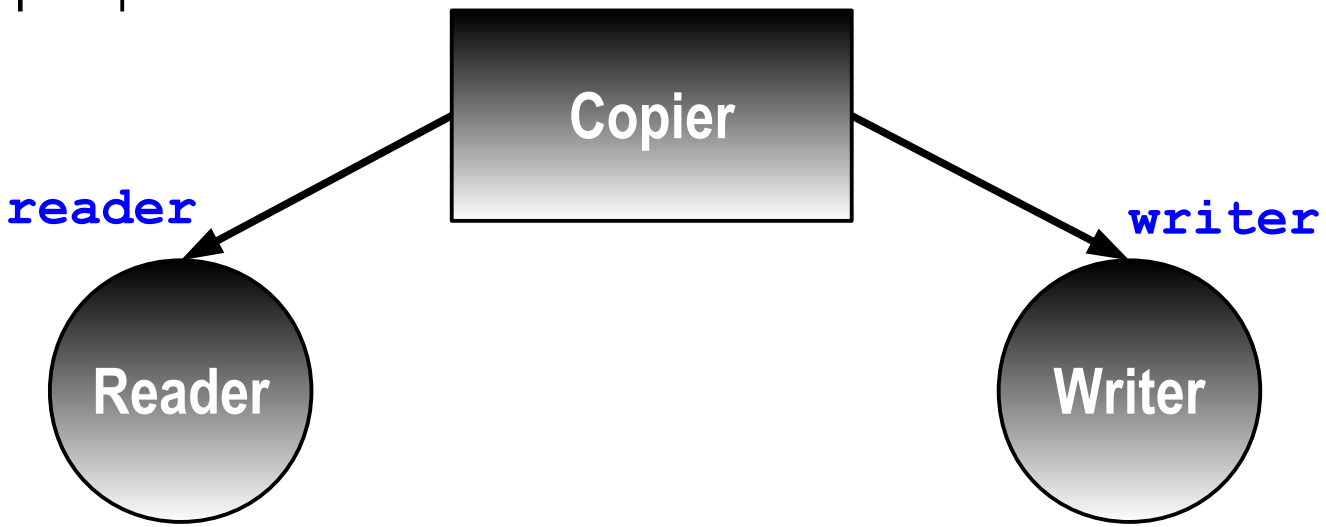
```
while ( (data = readKeyboard()) != -1 )
{
  writeFile("./filename", data);
}
```




> Нельзя повторно использовать в другом контексте
> Сделаем его независимым от объектов источника и назначения данных (инвертируем зависимости)

# Пример инверсии зависимостей

- Модули верхнего уровня не должны зависеть от модулей нижнего уровня. Оба типа модулей должны зависеть от абстракций



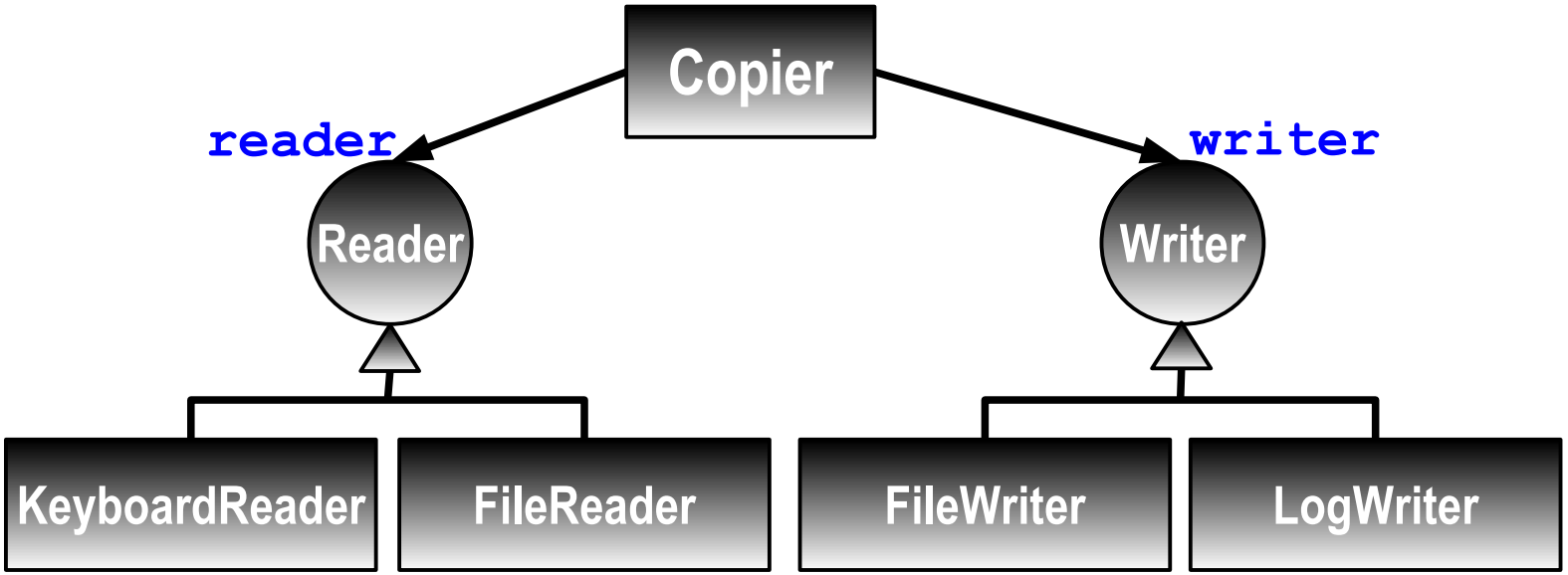

```
interface Reader {
  public int read();
}

interface Writer {
  public void write(int c);
}
```

```
Copier.run():
while ( (data = reader.read()) != -1 )
{
  writer.write(data);
}
```

# Пример инверсии зависимостей

- Абстракция не должна зависеть от реализации. Реализация должна зависеть от абстракции




```
copier = new Copier();

// Копирование данных с клавиатуры в файл
copier.run(new KeyboardReader(), new FileWriter("./filename"));

// Отправка данных из файла системному обработчику логов
copier.run(new FileReader("./filename"), new LogWriter());
```

# Пример инверсии зависимостей

- Свойства полученного кода:
  - модуль может быть использован для копирования данных в контексте отличном от исходного
  - можно добавлять новые устройства ввода/вывода, не меняя при этом модуль Copier
- Снизилась хрупкость кода, повысилась его мобильность и гибкость

# Формы инверсии зависимостей

- Dependency Injection (инъекция зависимостей) — пассивная форма
  - Необходимые сервисы «впрыскиваются» в пользовательский объект контейнером или самим программистом
- Dependency Lookup (поиск зависимостей) — активная форма
  - Зависящий объект сам получает свои зависимости при помощи вспомогательных объектов
  - В системе есть общедоступный объект (локатор или реестр сервисов), который знает обо всех используемых сервисах
- Комбинированная форма
  - Инъекция локатора и поиск зависимостей в нем

# Виды инъекции зависимостей

- С помощью конструктора (constructor injection)
  - Необходимые объекты передаются в конструктор в качестве аргументов

```
class Copier {
    public Copier(Reader reader, Writer writer) { ... }
}
Copier copier = new Copier(new KeyboardReader(),
                           new FileWriter("./filename"));
```

- При помощи set-метода (setter injection)
  - Отдельный set-метод для каждого инъецируемого объекта

```
class Copier {
    public void setReader(Reader reader) { ... }
}
Copier copier = new Copier();
copier.setReader(new KeyboardReader());
```

# Виды инъекции зависимостей

- С помощью интерфейсов (interface injection)
- В поле (field injection)
  - В Java, C# существует возможность получить доступ к private/protected полям объекта

```
class Copier {
    private Reader reader;
}
Copier copier = new Copier();
Class type = copier.getClass();
Field field = type.getDeclaredField("reader");
field.setAccessible(true);
field.set(copier, new KeyboardReader());
```

# Как выполнить инъекцию зависимостей?

- Программно (вручную)
- С помощью контейнера
  - Способы конфигурирования контейнера
    - Программно
    - Конфигурационный XML-файл
    - **Аннотации**

# Аннотации в Java

- Часто требуется связать с классом, методом, полем дополнительную информацию, нужную для
  - Компилятора и других инструментальных средств, работающих с исходным кодом
  - Инструментальных средств обработки скомпилированного кода
    - byte-code enhancement
  - Среды выполнения скомпилированного кода
- Java <5 - инструкции в документационных комментариях, внешние файлы (в т.ч. XML-файлы)
- В Java **5** появился особый тип данных – аннотационный

# Аннотация

- Модификатор, состоящий из
  - имени типа аннотации и
  - пар «элемент-значение»
    - для каждого элемента соответствующего типа аннотации, за исключением элементов со значением по умолчанию
  - Используется в объявлениях пакета, класса, интерфейса, поля, метода, параметра, конструктора или локальной переменной
    - Не более одной аннотации конкретного типа на объявление
  - Сохранность аннотации (retention)
    - Только в исходном коде, либо
    - В скомпилированных классах и интерфейсах
    - Доступность во время выполнения через рефлексию (необязательно)

# Аннотация

- Виды аннотаций по использованию:
  - Нормальные

```
@RequestForEnhancement(
    id = 2868724, synopsis = "Provide time-travel functionality",
    engineer = "Mr. Peabody", date = "4/1/2004")
public static void travelThroughTime(Date destination) { ... }
```

  - Маркерные (без элементов)

```
@Preliminary public class TimeTravel { ... }
```

  - Одноэлементные (элемент с именем value)

```
@Author(@Name(first = "Joe", last = "Hacker"))
public class BitTwiddle { ... }
```

- Типы элементов аннотации:
  - Все примитивные типы, String, Class, перечислимые типы, типы аннотаций и одномерные массивы значений любого из указанных выше типов
  - Значение ~~null~~

# Инверсия зависимостей в EJB

- Окружение компонента (bean environment)
  - Контекст имен java:comp/env - зависимости компонента, не разделяется между компонентами
    - Ссылки на представления используемых EJB-компонентов и такие ресурсы, как
    - Источники данных
    - Очереди сообщений
    - Параметры инициализации
    - …
  - Контекст имен java:comp - службы контейнера

# Инверсия зависимостей в EJB

- Локаторы
  - java.naming.InitialContext
    - метод lookup() ищет по полному JNDI-имени
  - javax.ejb.EJBContext
    - особые методы для доступа к службам контейнера
    - метод lookup() ищет по относительному пути в java:comp/env

```
@EJB(beanInterface=GetPaymentLocal.class, name="GetPayment")
@Stateful
public class StatefulBean implements StatefulLocal {
    public void processRequest(...) {
        InitialContext ic = new InitialContext();
        GetPaymentLocal result = (GetPaymentLocal)
                                 ic.lookup("java:comp/env/GetPayment");
    }
}
```

# Инверсия зависимостей в EJB

- Инъекция зависимостей
  - С помощью set-метода или в поле
  - Описывается в дескрипторе модуля, либо
  - С помощью аннотаций @Resource и @EJB
    - Могут содержать JNDI-имена зависимых ресурсов, либо эти имена указываются в дескрипторе модуля

```
@Stateful
public class StatefulBean implements StatefulLocal {
    @EJB(beanName="GetPayment") private GetPaymentLocal result;
    ...
}
```

# Контекст сессионного компонента

- Объект типа javax.ejb.SessionContext
  - Автоматически создается и поддерживается EJB-контейнером
  - Используется компонентом для обратной связи с контейнером
  - `@Resource SessionContext ctx;`
  - Предоставляемые методы:
    - получение объектов представления, в т.ч. реализации указанного бизнес-интерфейса
    - получение сведений о клиенте компонента
    - программное управление транзакциями
    - поиск зависимостей в контексте имен java:comp/env

# Contexts and Dependency Injection

- JSR-299, обязательный компонент Java EE 6
- Java EE-компонент является CDI-компонентом (bean), если контейнер может управлять ЖЦ его экземпляров в соответствии с моделью ЖЦ контекстов
- Экземпляры CDI-компонента относятся к определенному контексту и определяют состояние и логику приложения
- Контейнер создает и удаляет экземпляры CDI-компонентов и связывает их с подходящим контекстом
- Экземпляры CDI-компонентов можно инъецировать в другие объекты и использовать в EL-выражениях

```java
@Named("guessNumber")
@ConversationScoped
public class GuessNumberBean implements Serializable {
    private int number;
    private int userGuess;
    private int triesCount;
    private boolean hasGuessed;

    // Add business logic below. (Right-click in editor and choose
    // "Insert Code > Add Business Method")
    public int getTriesCount() {
        return triesCount;
    }

    public int getUserGuess() {
        return userGuess;
    }

    public void setUserGuess(int userGuess) {
        this.userGuess = userGuess;
    }

    public void checkGuess() {
        System.out.println("compare " + userGuess + " and " + number);
        if (userGuess == number) {
            hasGuessed = true;
        } else {
            triesCount++;
        }
    }
```

```
@ApplicationScoped
@Named("successCounter")
@Singleton
public class SuccessCounterBean implements SuccessCounterBeanLocal {

    private int totalSuccesses;

    public int getTotalSuccesses() {
        return totalSuccesses;
    }

    @Lock(LockType.WRITE)
    public void addSuccess(@Observes SuccessfulGuessEvent event) {
        System.out.println("one more success");
        totalSuccesses++;
    }
}
```

# Контекст диалога (Conversation)

```
@Named("guessNumber")
@ConversationScoped
public class GuessNumberBean implements Serializable {

    @Inject Conversation conversation;

    public void checkGuess() {
        System.out.println("compare " + userGuess + " and " + number);
        if (userGuess == number) {
            hasGuessed = true;
            conversation.end();
        } else {
            triesCount++;
        }
    }

    @PostConstruct
    private void postConstruct() {
        conversation.begin();
        _reset();
        System.out.println("guess " + number);
    }
```

# События

```
@ConversationScoped
public class GuessNumberBean implements Serializable {
    @Inject
    Event<SuccessfulGuessEvent> guessEvent;

    public void checkGuess() {
        System.out.println("compare " + userGuess + " and " + number);
        if (userGuess == number) {
            hasGuessed = true;
            guessEvent.fire(new SuccessfulGuessEvent());
        } else {
            triesCount++;
        }
    }
```

Инициирование

Обработка

```
public class SuccessCounterBean implements SuccessCounterBeanLocal {
    private int totalSuccesses;

    public int getTotalSuccesses() {
        return totalSuccesses;
    }

    @Lock(LockType.WRITE)
    public void addSuccess(@Observes SuccessfulGuessEvent event) {
        System.out.println("one more success");
        totalSuccesses++;
        System.out.println("fire inter-ejb event");
    }
}
```

# Ограничение операций с контейнером в методах сессионных компонентов

| | Конструктор | Обработчик события ЖЦ | | Бизнес-метод |
|---|---|---|---|---|
| | | SFSB | SSB | |
| Получение объектов представления компонента | – | + | + | + |
| Сведения о клиенте компонента | – | + | – | + |
| Контекст имен java:comp/env | – | + | + | + |
| Вызов других EJB-компонентов | – | + | – | + |
| Доступ к менеджерам ресурсов | – | + | – | + |
| Доступ к менеджеру сущностей | – | + | – | + |

# Ограничения на реализацию бизнес-методов

- Для обеспечения переносимости компонентов
- Основные причины:
  1) Экземпляры компонента могут выполняться в разных JVM
     - > Сохранение значений в статических полях
     - > Использование примитивов синхронизации
  2) Ухудшение управляемости среды выполнения
     - > Работа с загрузчиками классов и менеджерами безопасности, останов JVM, изменение потоков стандартного ввода/вывода
     - > Установка фабрики сокетов и фабрики обработчиков потока данных для URL
     - > Управление потоками и группами потоков

# Ограничения на реализацию бизнес-методов

- Основные причины:
  3) Потенциальное нарушение безопасности
     - Использование рефлексии для получения информации о недоступных членах классов
     - Доступ к пакетам и классам, недоступным компоненту по правилам языка Java
     - Программное создание класса в пакете
     - Доступ и изменение объектов конфигурации безопасности
     - Получение политики безопасности для конкретного источника кода
     - Загрузка машинно-зависимых библиотек
     - Использование возможностей протокола сериализации по подстановке объектов и подклассов
     - ...

# Ограничения на реализацию бизнес-методов

- Основные причины:
  4) Особые ограничения
     - Вывод информации на дисплей и ввод с клавиатуры функциями AWT (Swing)
       - Большинство серверов работает в headless-режиме
     - Доступ к файлам и каталогам в файловой системе средствами пакета java.io
       - Для хранения данных следует использовать менеджеры ресурсов
     - Прослушивание сокетов, прием соединения по сокетам и использование сокетов для широковещания
       - Конфликтует с основной функцией компонента – обслуживанием EJB-клиентов
     - Передача this как параметра при вызове метода и как результата вызова метода
       - Почему?