Inversion of Control
инверсия управления

Dependency Injection (внедрение зависимостей)

Инверсия зависимости

Зависимости между классами превращаются в ассоциации между объектами.

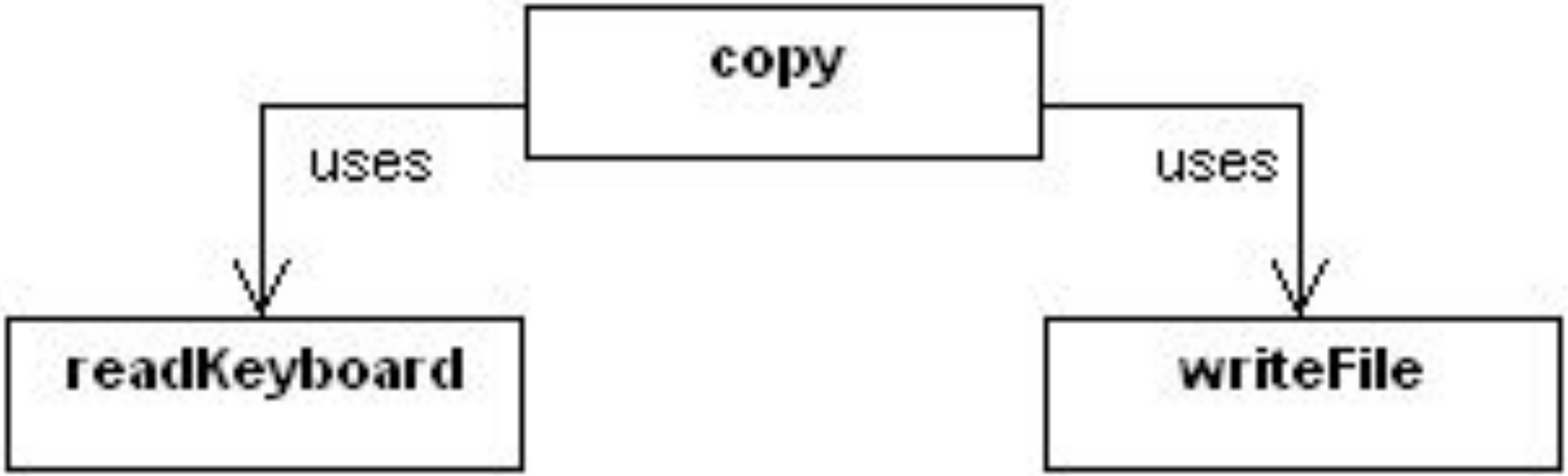
Ассоциации между объектами могут устанавливаться и меняться во *время выполнения* приложения.

Это позволяет сделать модули менее связанными между собой

2 принципа инверсии зависимостей:

- Модули верхнего уровня не должны зависеть от модулей нижнего уровня. Оба типа модулей должны зависеть от абстракций;
- Абстракция не должна зависеть от реализации. Реализация должна зависеть от абстракции.

Схема программы копирования данных



Copy может выглядеть примерно следующим образом:

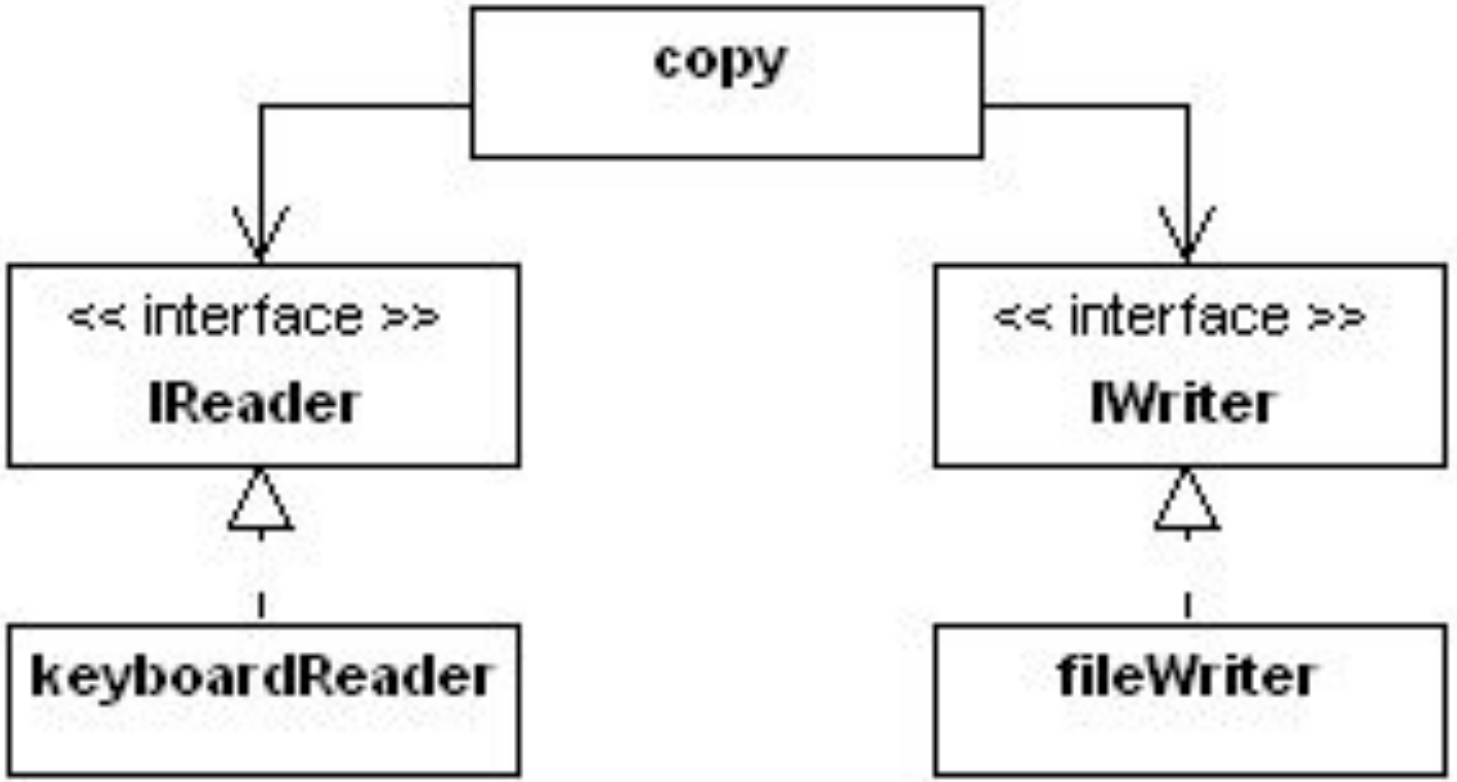
```
 1: public static class CopyManager
 2: {
 3:     public static void Copy()
 4:     {
 5:         var keyboard = new Keyboard();
 6:         var file = new File();
 7:         byte[] buffer;
 8:
 9:         while ((buffer = keyboard.Read()) != null)
10:         {
11:             file.Write(buffer);
12:         }
13:     }
14: }
```

МИНУС

Низкоуровневые классы ***Keyboard*** и ***File*** обладают высокой гибкостью. Можно легко использовать их в контексте, отличном от класса CopyManager.

Однако сам *класс* CopyManager не может быть повторно использован в другом контексте. Например, для отправки данных из файла системному обработчику логов.

Используя принцип инверсии зависимостей, можно сделать *класс* CopyManager **независимым от объектов источника** и назначения данных.



```
1: public interface IReader
2: {
3:     public byte[] Read();
4: }
5:
6: public interface IWriter
7: {
8:     public void Write(byte[] arg);
9: }
```

Класс CopyManager должен полагаться только на выработанные абстракции и не делать никаких предположений *по* поводу индивидуальных особенностей объектов ввода/вывода:

```
 1: public static class CopyManager
 2: {
 3:     public static void Copy(IReader reader, IWriter writer)
 4:     {
 5:         byte[] buffer;
 6:
 7:         while ((buffer = reader.Read()) != null)
 8:         {
 9:             writer.Write(buffer);
10:         }
11:     }
12: }
```

Теперь код обладает следующими качествами:

- класс может быть использован для копирования данных в контексте, отличном от данного;
- возможно добавлять новые устройства ввода/вывода, не меняя при этом класс Copy.

Формы инверсии зависимостей

Существует две формы инверсии зависимостей:

активная и **пассивная**.

- При использовании пассивной формы объекты зависимости внедряются в зависимый *объект*. Зависимому объекту не надо прилагать никаких усилий, все нужные сервисы он получает через свой *интерфейс*.
- Активная форма, в отличие от пассивной, предполагает, что зависящий *объект* будет сам получать свои зависимости при помощи вспомогательных объектов.

Пассивная *инверсия* зависимостей (*Dependency Injection*):

- внедрение через конструктор
- внедрение через устанавливаемое свойство
- внедрение через интерфейс
- внедрение через поле

Активная *инверсия* зависимостей (*Dependency* Lookup):

- **pull-подход:** предполагается наличие в системе общедоступного объекта, который знает обо всех используемых сервисах. В качестве такого объекта может выступать объект, реализующий *паттерн* **Service Locator**. Локатор реализует *паттерн* **синглетона**, благодаря чему доступ к нему можно получить из любого места приложения.
- **push-подход**: данная методика отличается от pull-подхода тем, как объект узнает об объекте-локаторе. При использовании pull-подхода класс сам получал локатор посредством класса-синглетона. Push-подход характеризуется тем, что объект-локатор (или как его иногда называют контекст) передается в класс извне (обычно через конструктор).

IoC контейнер

- **IoC *контейнер*** – это специальный *объект*-сборщик, который на основании схемы зависимостей между классами и абстракциями может создать *граф* объектов. Любой IoC *контейнер* реализует принцип инверсии зависимостей.

MEF

Она позволяет строить модульные приложения с минимальным уровнем связности частей (parts) приложения.

Эта библиотека включает в себя не только *Dependency Injection контейнер*, но большой *объём* инфраструктуры: множество механизмов поиска элементов композиции в сборках, удалённых XAP файлах, механизм пометки элементов композиции с помощью *.Net* атрибутов и т.д.