ПРИНЦИПЫ ПРОЕКТИРОВАНИЯ

Д. Мигинский

Базовые принципы

Разделение ответственностей (SoC, DRY)

Бритва Оккама (KISS)

Проблема:
принципы слишком общие, они не содержат подсказок, как им удовлетворить

Способ решения:
Накопление и формализация опыта в форме более частных принципов и образцов проектирования

Принципы и образцы проектирования

Принцип проектирования (дизайна) – утверждение о конечных характеристиках «хорошей» архитектуры.

Принцип может не содержать указаний, как этого достичь.
Принцип допускает исключения, но не приветствует их.
Некоторые принципы обладают «широким спектром действия», другие применимы только в рамках определенных парадигм.

Образец проектирования – способ решения некоторой типичной задачи проектирования.
В отличие от принципов, в образцах всегда описывается способ как достичь результата.

Некоторые принципы также обладают и свойствами образцов проектирования.

Существующие системы принципов

GRASP (General Responsibility Assignment Software Patterns) – система 9-ти то ли образцов, то ли принципов, связанных с распределением ответственностей между классами/пакетами.
Преимущества: претендует на полноту (в основном за счет 2-х из 9-и принципов)
Недостатки: уровень обобщения (абстракции) формулировки различных принципов слишком различается, отчего система не формирует согласованной картины.

SOLID(первые буквы названий принципов) – система из 5-ти принципов, сформулированная Р. Мартином.
Преимущества: все принципы формулируются примерно на одном уровне абстракции, относительно ортогональны.
Недостатки: система не полная.

Также существует множество самостоятельных, несистематизированных принципов

Понятие контракта

Контракт – совокупность соглашений о функциональности элемента программы (модуля, класса и т.д.).

Контракт определяет обязательства 2-х сторон:

Сервер предоставляет некоторую функциональность, гарантирует выполнение определенных правил, а также требует выполнения определенных правил клиентом.
Сервер обязуется возвратить корректный результат, если клиент передаст ему корректные параметры.

Понятие контракта введено в контрактном программировании (Б. Мейер) в середине 80-х. Цель КП – объединить ООП и формальные методы доказательства корректности программ (в первую очередь, на основе логики Хоара)

Структура контракта

В контракт входят:

- набор операций;
- параметры операций, область допустимых значений, интерпретация;
- допустимые возвращаемые значения, их интерпретация;
- сообщения об ошибках (исключения и т.д.);
- предусловия *(напр. перед выполнением любой операции сервер должен быть инициализирован);*
- постусловия *(напр. после выполнения операции init можно осуществлять другие операции);*
- инварианты *(напр. имя пользователя всегда уникально);*
- побочные эффекты (изменение глобального состояния, базы данных и т.д.);
- дополнительные гарантии (напр. производительность).

Принципы, связанные с контрактом

- Liskov Substitution Principle (LSP)
- Open/Closed Principle (OCP)
- Command-Query Separation (CQS)

Принцип подстановки Барбары Лисков (Liskov Substitution Principle, LSP)

Формулировка в терминах контракта:
Контракт производного класса не должен противоречить контракту базового класса.

Следствия:

- клиент, оперирующий контрактом базового класса не должен знать, каким конкретно подклассом этот контракт реализуется;
- подкласс не может усиливать предусловия;
- подкласс не может ослаблять постусловия;
- подкласс не может изменять инварианты;

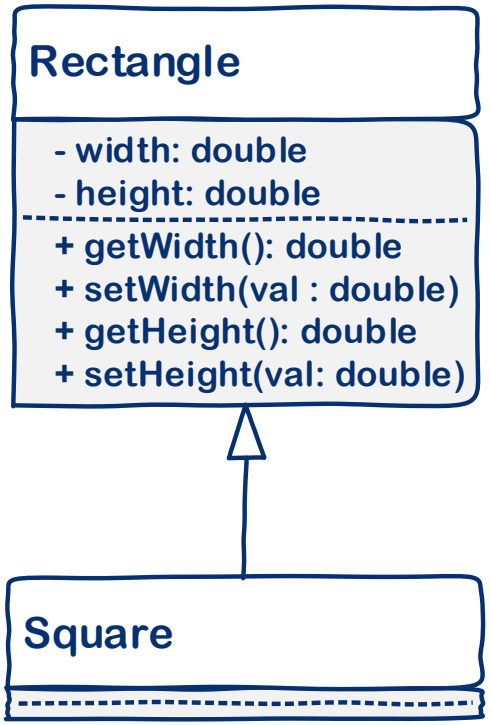
Оригинальная формулировка:
Let ***q(x)*** be a property provable about objects ***x*** of type ***T***. Then ***q(y)*** should be provable for objects ***y*** of type ***S*** where ***S*** is a subtype of ***T***.

«Классическое» нарушение LSP

В рамках контракта Rectangle клиент предполагает, что размеры устанавливаются независимо.

При использовании экземпляра Square и изменении одной из сторон клиент может делать ошибочные выводы о состоянии (например, площади прямоугольника/квадрата)

В чем конкретно состоит нарушение?



Суть нарушения

Square определяет новый инвариант, равенство сторон прямоугольника.

Проблема вызвана наличием мутаторов в контракте Rectangle.

С точки зрения геометрии (в математики все объекты неизменяемы) наследование корректно.

Утиная типизация и LSP

Утиный тест:
Если нечто выглядит как утка, плавает как утка и крякает как утка, то это, скорее всего, и есть утка.

LSP «в обратную сторону»:
Если все объекты класса ***S*** содержат все операции и удовлетворяет всем ограничениям контракта класса ***T***, класс ***S*** является подклассом класса ***T***.

Замечания:

- LSP – формальное определение отношения генерализации;
- в большинстве языков отношение наследования не является ни необходимым, ни достаточным для выполнения LSP.

Принцип открытости/закрытости (Open/Closed Principle, OCP)

Оригинальная формулировка:

Программные сущности (классы, модули, функции и т.д.) должны быть открыты для расширения и закрыты для модификации.

(Б. Мейер, 1988)

Оригинальная трактовка:

После разработки класса все изменения, которые в него вносятся, должны касаться только исправления ошибок. Расширение функциональности должно делаться через подклассы.

Виды изменений в коде

- Исправление ошибок
- Изменения кода, связанные с изменением требований (не обязательно функциональными)
  - Расширение функциональности
- Рефакторинг: изменение дизайна коде без изменения функциональности

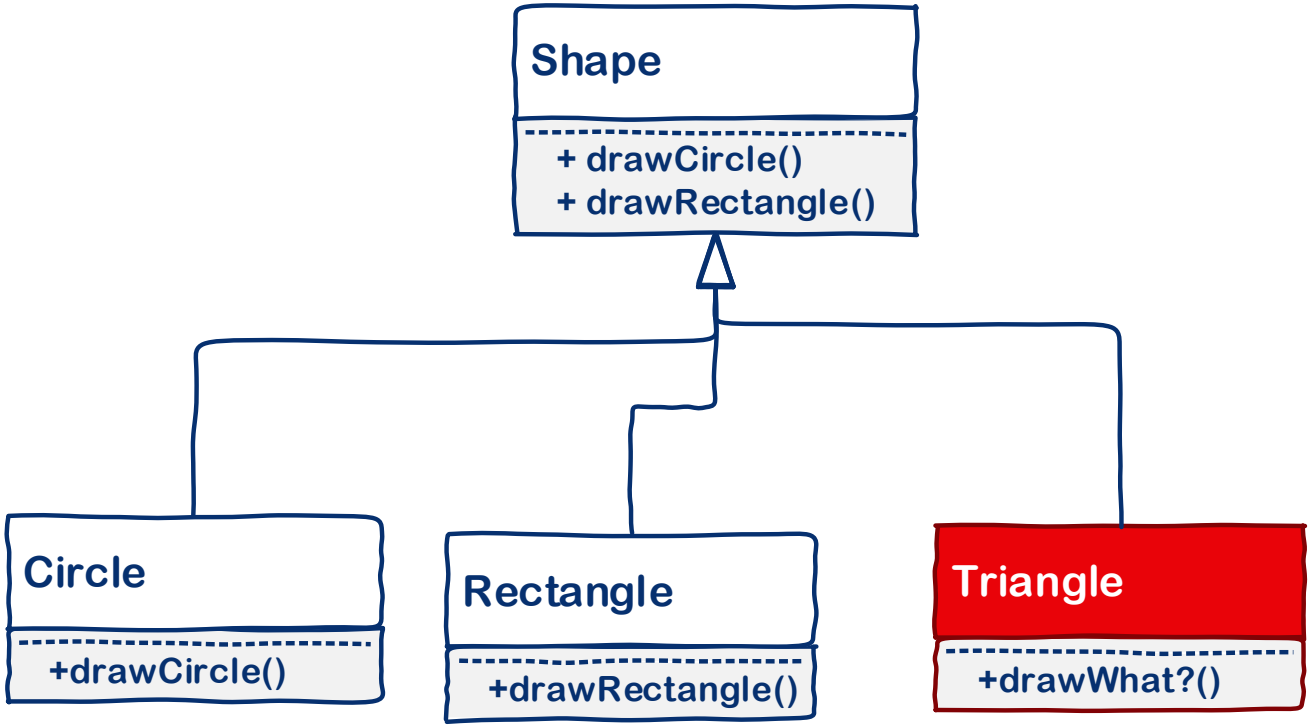
Замечание: говоря об изменениях кода этот и все последующие принципы не рассматривают рефакторинг

OCP в компонентном программировании

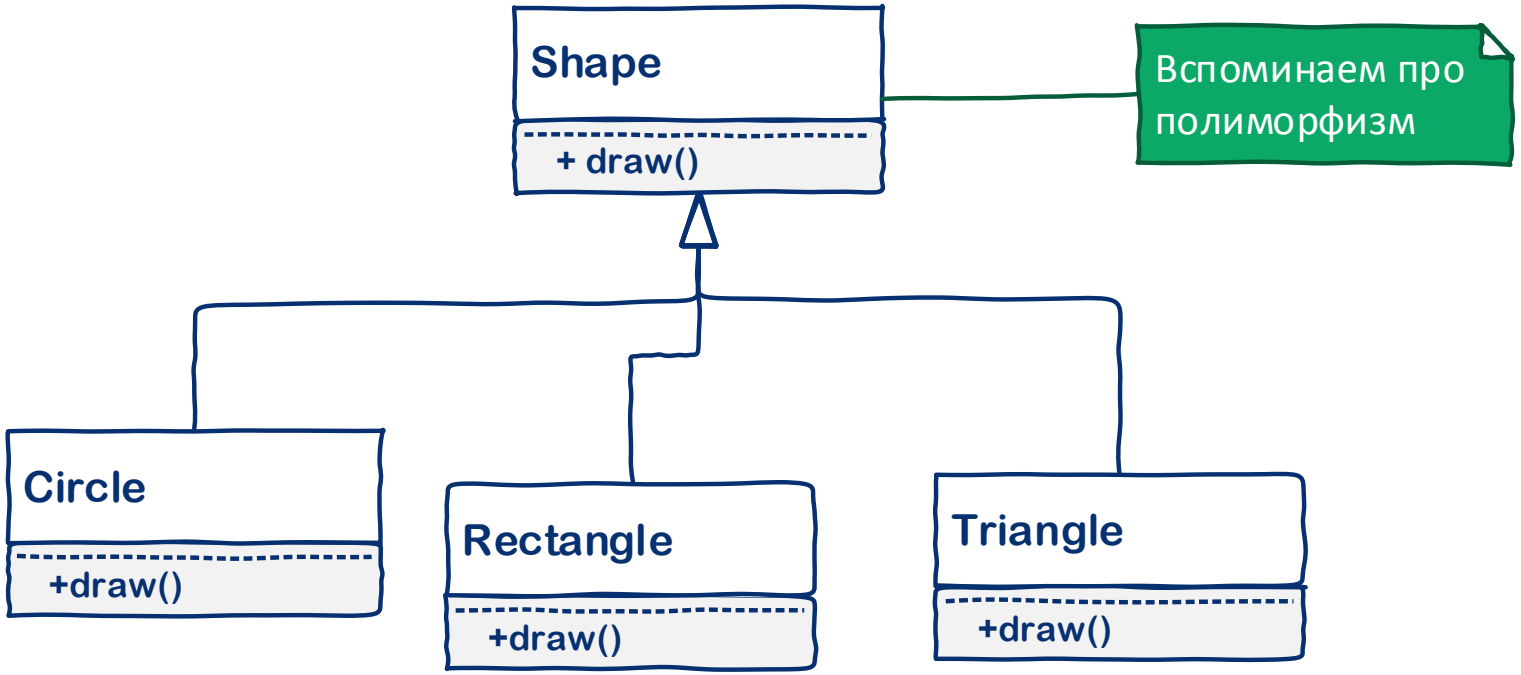
Компонентное программирования – форма ООП, где запрещается наследование в общем виде, однако разрешено разделение определения и реализации контракта (как правило – реализация интерфейса).
Прямое взаимодействие одного класса с другим запрещено, допускается только взаимодействие через интерфейс.

В этом случае OCP формулируется в отношении контракта (интерфейса), но не касается конкретных классов.

Нарушение OCP



Решение в рамках OCP



Побочные эффекты и чистые выражения

Чистое выражение (referentially-transparent expression) – выражение, которое может быть заменено на свое значение без влияния на выполнение программы.

Побочные эффекты – любые изменения состояния программы, помимо порождения результата (или возбуждения исключения).

К побочным эффектам относятся любые изменения состояния объектов, глобальных переменных, ввод-вывод.

Чистые выражения = выражения, не имеющие побочных эффектов.

Разделение команд и запросов (Command-Query Separation, CQS)

Альтернативное название:
Command-Query Responsibility Segregation (CQRS)

Формулировка (близкая к оригинальной):
Операция либо имеет побочные эффекты (команда), либо возвращает значение (запрос), являясь чистой функцией.

(Б. Мейер, 1988)

Допустимые отклонения от CQS (фактически, не нарушают CQS)

- Команда, возвращающая статус операции
- Кэширующий запрос
- Иногда: запрос, выставляющий внешний признак ошибки (напр. в переданную по ссылке переменную)

Где CQS неприменим?

- Транзакционные операции (в частности, atomic-типы)
- Интерфейсы, формирующие встраиваемые языки (например, конструкторы запросов). *Примечание: в этом случае часто используются т.н. fluent-интерфейсы (М. Фаулер)*

Согласованность и связанность

Согласованность (cohesion) – степень «сфокусированности» обязанностей модулей системы.

Связанность (coupling) – степень того, насколько сильно модули зависят друг от друга.

Примечания:

- *понятия введены Л. Константином в 1974;*
- *обычно говорят о высокой или низкой согласованности, сильной или слабой связанности хотя есть попытки ввести количественные метрики.*

Согласованность

High Cohesion (принцип): все программные единицы (функции, классы, модули) должны иметь высокую согласованность.

В зависимости от выбранного критерия согласованность варьируется от высокой до низкой.

«Хорошие» критерии:

- Функциональная согласованность (functional cohesion, ORR)
- Совместное использование (procedural cohesion, CRP)
- Совместные изменения (CCP)
- Последовательное использование (sequential cohesion): результаты одной функции передаются в другую и т.д.
- Оперирование одними и теми же данными (communicational cohesion)

Функциональная согласованность: правило единственной ответственности

One Responsibility Rule:

Класс (в общем случае также функция, модуль и т.д.) должен делать что-то одно, должен делать это хорошо и должен делать только это.

(Б. Мейер, 1988)

Примечание: при выполнении ORR обеспечивается самая высокая согласованность - функциональная

Типичная проблема распределения ответственностей

Было: 100 классов/интерфейсов.

Применяем: ORR

Получаем: 50 пакетов по 2 класса/интерфейса

Проблема: число сущностей как было необозримо, так и осталось

Что делать: снижать согласованность, но аккуратно – см. принципы далее

Принцип эквивалентности выпуска и переиспользования

Использование: я написал код и сам его использую.

Переиспользование: я написал код, отдал кому-то, и он его использует (часто это называют просто использованием).

Reuse/Release Equivalence Principle (REP):

Единица переиспользования есть единица выпуска.

Переиспользованы могут быть только компоненты (прим.: компонент(-а) – единица физической организации системы; модуль – логическая единица), которые прошли формальную процедуру выпуска (release). Такая единица называется пакетом (можно рассматривать, как определение пакета).

http://www.objectmentor.com/resources/articles/granularity.pdf

Следствие: «высокая» технология copy-paste неприменима к чужому коду (прим.: да и к своему тоже – см. SoC)

Примечание: далеко не все пакеты проходят формальную процедуру выпуска и используются сами по себе (могут быть частью более крупного пакета). Но проектировать их следует так, чтобы они были готовы к этому.

Принцип совместного переиспользования

Common reuse principle (CRP):

Программные сущности внутри пакета используются вместе. Если используется хотя бы одна из них, то следует считать, что используются все.

http://www.objectmentor.com/resources/articles/granularity.pdf

Интерпретация: если при типичном использовании пакета в действительности задействована только малая часть его составляющих, то принцип не соблюдается (тянем за собой балласт).

Примечание: при выполнении CRP обеспечивается procedural cohesion

Принцип согласованного изменения

Common closure principle (CRP):

Программные сущности внутри пакета должны изменяться согласованно. Если мы изменяем одну сущность, то мы меняем весь пакет.

http://www.objectmentor.com/resources/articles/granularity.pdf

Интерпретация: если при типичном ожидаемом изменении пакета (например, при изменении контракта пакета, от которого зависит данный) изменению подвергается только небольшая часть его программных сущностей, то принцип не соблюдается.

Примечание: соблюдение этого принципа нельзя измерить заранее, можно только предвидеть

Что дает высокая согласованность?

- Существенное количество ошибок локализуются (ORR).
- По проявлению ошибки (bug) обычно легко определить модуль/класс, где она локализуется (maintainability).
- При необходимости внесения изменений, они легко локализуются (частично extensibility).
- Пакеты частично подготовлены к переиспользованию (частично reusability)

Чего не хватает?

- Если ошибка комплексная, не локализуется в одном пакете, то бороться с ней нужно, учитывая зависимости между пакетами
- Пока расширяемость обеспечивается только через изменения: нет механизмов для выполнения OCP.
- Использование пакета потенциально «тянет» за собой множество дополнительных зависимостей.

Таким образом, не хватает принципов организации зависимостей.

Связанность

Связанность (coupling) – степень того, насколько сильно модули зависят друг от друга. Бывает сильной и слабой.

Как измерять: можно посчитать количество и исходящих связей (вместе или по отдельности) между пакетами, классами, отдельными функциями. *Примечание: но лучше не измерять, а следовать принципам.*

Loose coupling (принцип): связанность должны быть слабой, т.е. связей между программными сущностями должно быть как можно меньше, и они должны быть как можно слабее.

Виды связанности

Хорошие (слабые):

- Модуль **A** определяет контракт (и, зачастую, реализует его), модуль **B** использует контракт.
- Модуль **A** определяет контракт (без реализации), модуль **B** его реализует.

Плохие (сильные):

- Модули **A** и **B** используют совместно некоторые данные (без строго определенного контракта, обеспечивающего целостность данных при совместном доступе).
- Модули **A** и **B** используют глобальные данные.
- Модуль **B** использует внутренние функции и/или структуры данных модуля **A**, не предусмотренные внешним контрактом (нарушение инкапсуляции).

Замечания по зависимостям

1. В общем случае все зависимости считаются транзитивными.

2. Следовательно, циклических зависимостей не должно быть.

Закон Деметры

Law of Demeter (LoD) / Principle of Least Knowledge:
Разговаривайте только с друзьями, не разговаривайте с незнакомцами.

Друзья модуля (напр. класса):

- Составные части данного модуля
- Составные части модуля, которому принадлежит данный
- Объекты, которые были переданы данному модулю
- Объекты, которые модуль сделал сам

Закон Деметры: альтернативная формулировка

- Вы можете играть сами с собой.
- Вы можете играть в свои игрушки (но не можете раздавать их).
- Вы можете играть в игрушки, которые сделали сами.
- Вы можете играть в игрушки, которые вам дали.

(Peter van Rooijen)

Принцип разделения интерфейсов

Interface Segregation Principle (ISP):

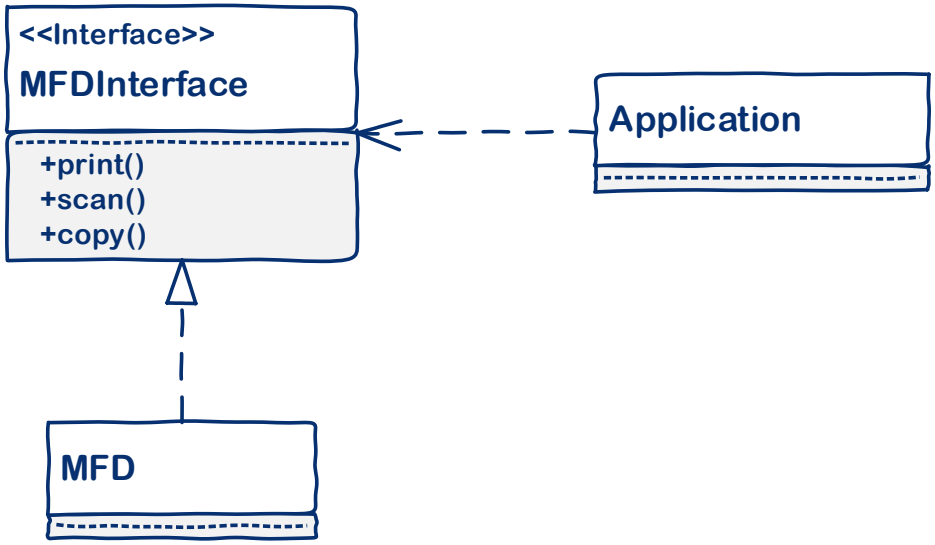
Клиент не должен зависеть от частей контракта, которые он не использует.

(Р. Мартин)

Следствие:

Для каждого (типичного вида) клиента должны быть отдельная проекция контракта (т.е. свой интерфейс)

Нарушение ISP

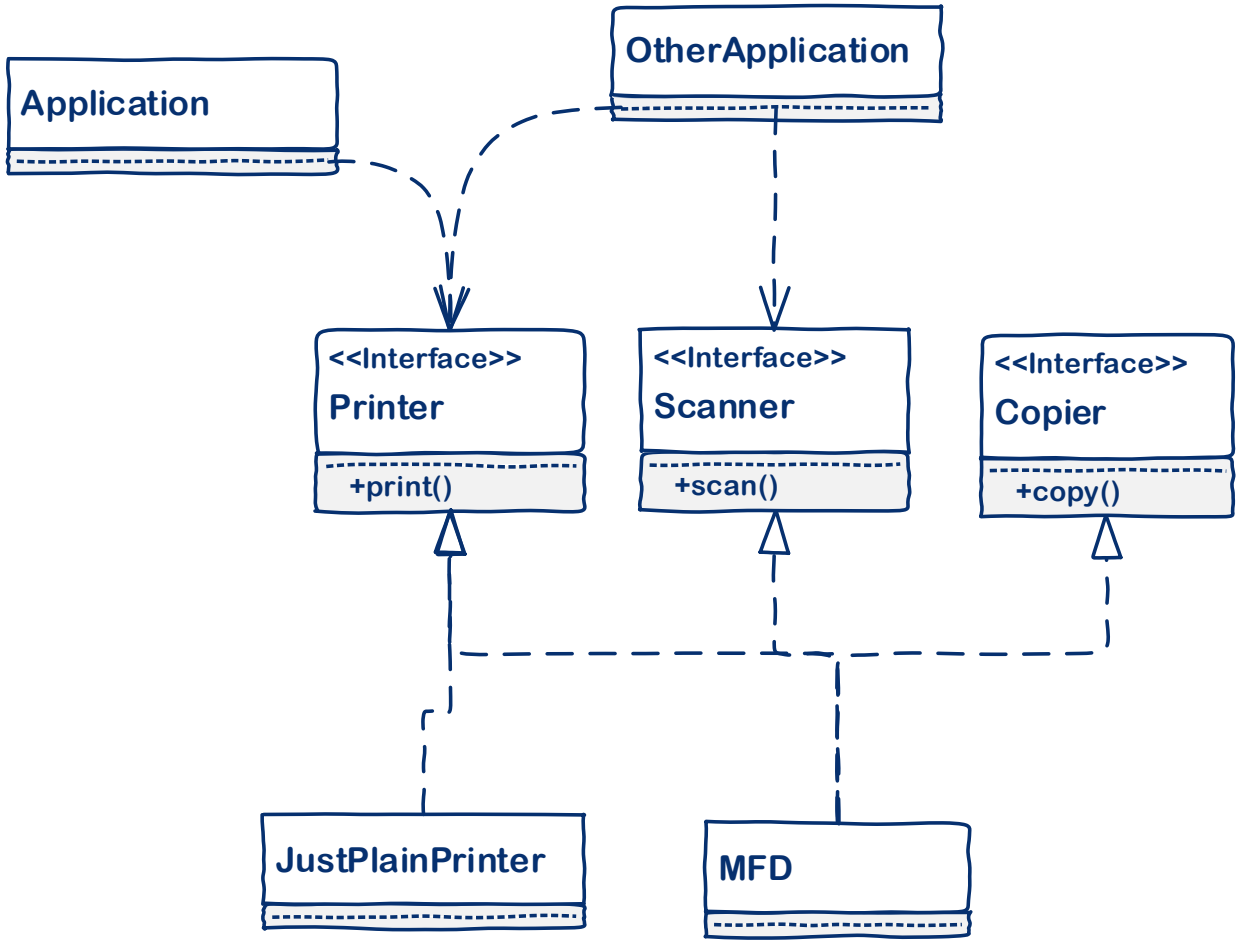


Application нуждается только в принтере.

Зачем ему знать про сканер и копир?

Если добавить в MFD еще функции факса, то Application необходимо как минимум пересобрать.

Решение, соответствующее ISP



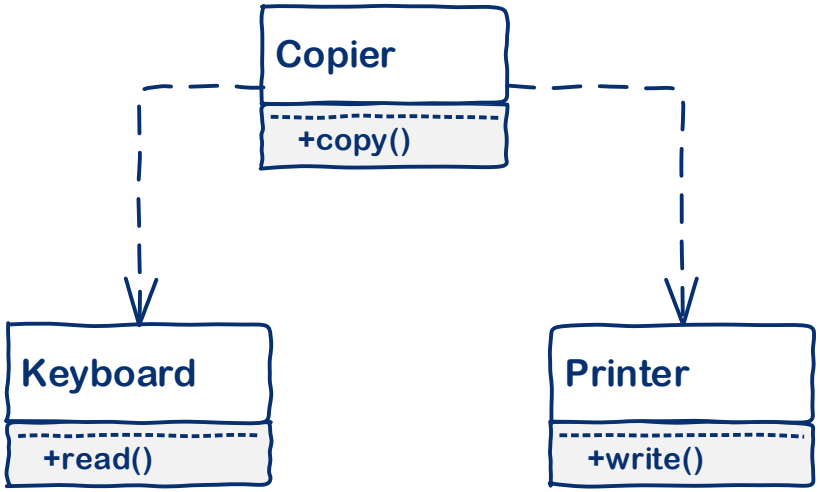
Принцип обращения зависимостей

Dependency Inversion Principle (DIP):

- Модули высокого уровня не должны зависеть от модулей низкого уровня. И те и другие должны зависеть от абстракций.
- Абстракции не должны зависеть от деталей реализации. Детали должны зависеть от абстракций.

(Р. Мартин)

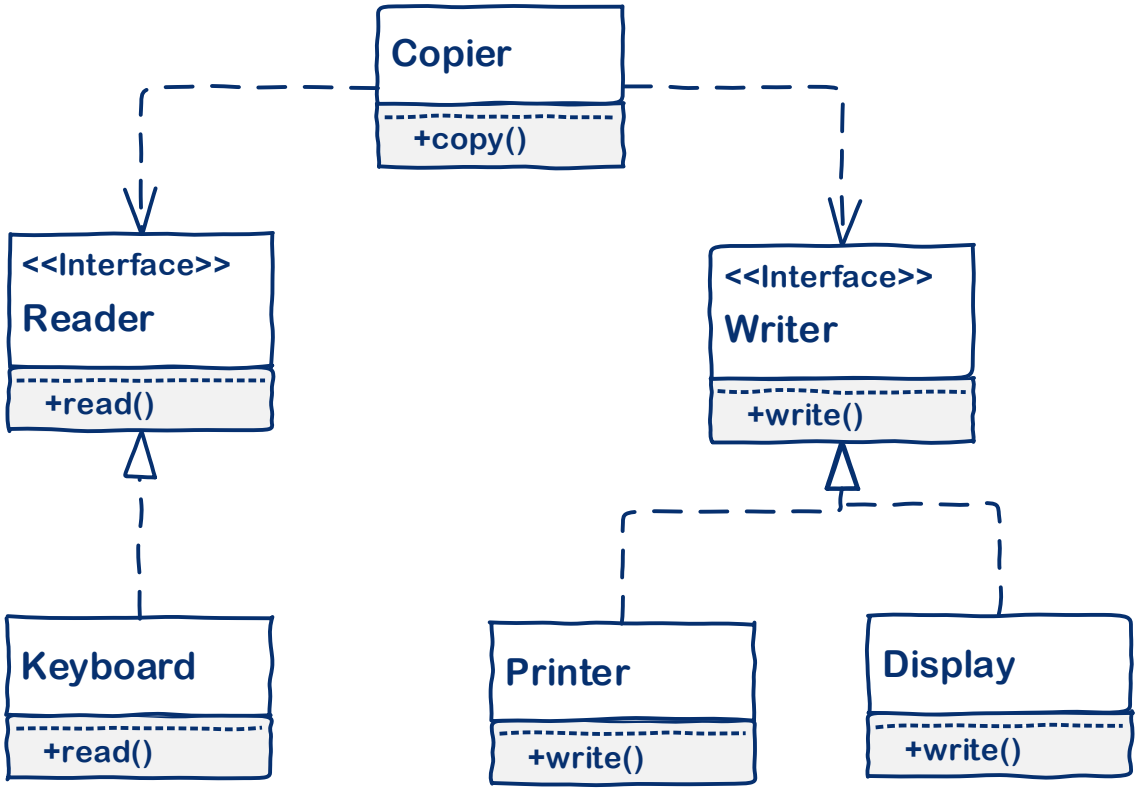
Нарушение DIP



Copier – модуль высокого уровня, **Printer** и **Keyboard** – низкого.

Если заменить **Printer** на **Display**, то логика **Copier** по сути не изменится, однако в такой форме его повторно использовать нельзя.

Решение, соответствующее DIP



Метрика абстрактности

Модуль может декларировать (или наследовать) некоторый контракт, и часть этого контракта реализовать.

Чем выше часть нереализованного контракта, тем выше абстрактность.

Количественная метрика может быть построена на основе абстрактных классов, абстрактных функций и т.д.

Метрика стабильности

Стабильность – мера сложности изменения модуля. При увеличении количества модулей, зависящих от модуля **A** возрастает стабильность **A**.

Количественная метрика может быть построена, как некоторое соотношение входящих и исходящих связей (между пакетами, классами, функциям, с учетом их веса или нет).

Следствие: если **B** зависит от **A**, то стабильность **A** не меньше, чем стабильность **B** (Stable Dependencies Principle, SDP).

Замечание: высокая стабильность есть нежелательное свойство модуля. Однако, без стабильных модулей невозможно построить ни одну сложную систему.

Соотношение стабильности и абстрактности

Принцип стабильных абстракций (Stable Abstractions Principle, SAP): стабильность модуля должна быть пропорциональна абстракции.

Отклонения:

- Абстрактный и нестабильный модуль: никому не нужен.
- Неабстрактный и стабильный модуль: расширение невозможно (контракт реализован полностью), модификация затруднена.