GeekBrains

Архитектура ПО

Методичка к уроку 4

Компоненты. Принципы связности и сочетаемости компонентов

Оглавление

Введение

Если речь идёт об архитектуре ПО (или ведёт её архитектор ПО), под компонентом имеется в виду то же, что часто называют программным модулем. Это достаточно произвольный и абстрактный элемент структуры системы, определённым образом выделенный среди окружения, решающий некоторые подзадачи в рамках общих задач системы и взаимодействующий с окружением через определённый интерфейс.

На диаграммах компонентов в языке UML часто изображают компоненты, являющиеся единицами сборки и конфигурационного управления: файлы с кодом на каком-то языке, бинарные файлы, какие-либо документы, входящие в состав системы. Иногда там же появляются компоненты, представляющие собой единицы развёртывания системы, — это компоненты уже в третьем, следующем смысле.

На этом уроке

1. Разберём понятие программного компонента.
2. Разберём понятие интерфейса.
3. Рассмотрим UML-диаграмму компонентов.
4. Рассмотрим принцип «проектирования по контракту» (Design by Contract).
5. Изучим принципы связности компонентов.
6. Поговорим о принципах сочетаемости компонентов.
7. Изучим способ версионирования компонентов Semantic Versioning.

Основные понятия компонентного подхода в разработке ПО

Вспомним из прошлой лекции:

💡 Программные структуры нижнего уровня — это классы и модули.

Классы — шаблоны, на основе которых создаётся конкретный программный объект. Класс описывает свойства и методы, определяющие поведение объектов этого класса.

Модули — наборы отдельных, не зависящих друг от друга блоков кода при построении программного приложения.

Принцип модульности упрощает проектирование программного обеспечения (ПО) и распределение процесса разработки между группами разработчиков. При разбиении ПО на модули для каждого модуля указывают реализуемую им функциональность, а также связи с другими модулями. Удобство использования модульной архитектуры заключается в возможности обновления (замены) модуля без необходимости изменения остальной системы.

Программные структуры среднего уровня — компоненты. Это множества классов и языковых конструкций, объединённых по общему признаку и организованных в соответствии с определёнными правилами и ограничениями.

Компоненты — это единицы развёртывания. Они представляют наименьшие сущности, развёртываемые в составе системы:

- В Java — это jar-файлы
- В Ruby — gem-файлы
- В .NET — библиотеки DLL
- В компилируемых языках — комплексы двоичных файлов
- В интерпретируемых языках — комплексы файлов с исходным кодом, например, Python (.py)

Компоненты развёртывания являются блоками, из которых строится компонентное программное обеспечение.

Компонент представляет собой структурную единицу программной системы, обладающую чётко определённым интерфейсом, который полностью описывает её зависимости от окружения. Такой компонент может быть независимо поставлен или не поставлен, добавлен в состав некоторой системы или удалён из неё, в том числе может включаться в системы других поставщиков.

Основные понятия компонентного подхода к разработке

Понятие программного компонента (software component) является одним из ключевых в современной инженерии ПО. Этим термином обозначают несколько различных вещей, часто не уточняя подразумеваемого в каждом конкретном случае смысла.

В архитектуре ПО под компонентом понимают то же, что часто называют программным модулем. Это элемент структуры системы, решающий подзадачи в

рамках общих задач системы и взаимодействующий с окружением через интерфейс — архитектурный компонент или компонент архитектуры.

На диаграммах компонентов в языке UML изображаются компоненты, являющиеся единицами сборки и конфигурационного управления: файлы с кодом на каком-то языке, бинарные файлы, какие-либо документы, входящие в состав системы. Иногда там же появляются компоненты, представляющие собой единицы развёртывания системы, — это компоненты уже в третьем, следующем смысле.

Компоненты развёртывания являются блоками, из которых строится компонентное программное обеспечение. Эти же компоненты имеются в виду, когда говорят о компонентных технологиях, компонентной или компонентно-ориентированной (component based) разработке ПО.

Компонент представляет собой структурную единицу программной системы, обладающую чётко определённым интерфейсом, который полностью описывает её зависимости от окружения.

Понятие компонента сборки и конфигурационного управления связано со структурными элементами системы, с которыми приходится иметь дело инструментам, осуществляющим сборку и конфигурационное управление, а также людям, работающим с этими инструментами. Для этих видов компонентов интерфейсы взаимодействия с другими такими же элементами системы могут быть не определены.

Компонент — выделенная структурная единица с чётко определённым интерфейсом. В этом смысле он — единица развёртывания. Он может быть присоединён к остальной системе, когда она уже некоторое время работает, и должен после этого выполнять все свои функции, если в исходной системе уже были все компоненты, от которых он зависит. Он может быть удалён из неё. Естественно, после этого могут перестать работать те компоненты, которые зависят от него. Он может быть встроен в программные продукты третьих партий и распространяться вместе с ними. В то же время никакая его часть не обладает этими свойствами.

В идеале такой компонент может быть добавлен или полностью замещён другой реализацией тех же интерфейсов прямо в ходе работы системы, без её остановки. Хотя и не все разновидности компонентов обладают этим свойством, его наличие признаётся крайне желательным.

Всё это означает, что такой компонент должен быть работоспособен в любой среде, где имеются необходимые для его работы другие компоненты. Это требует

наличия определённой инфраструктуры, которая позволяет компонентам находить друг друга и взаимодействовать по определённым правилам.

Компоненты отличаются от классов объектно-ориентированных языков:

1. Класс определяет не только набор реализуемых интерфейсов, но и саму их реализацию, поскольку он содержит код определяемых операций. Контракт компонента, чаще всего, не фиксирует реализацию его интерфейсов.
2. Класс написан на определённом языке программирования. Компонент же не привязан к определённому языку, если, конечно, его компонентная модель этого не требует. Компонентная модель является для компонентов тем же, чем для классов является язык программирования.

Обычно компонент является более крупной структурной единицей, чем класс. Реализация компонента часто состоит из нескольких тесно связанных друг с другом классов.

💡 Понятие компонента является более узким, чем понятие программного модуля.

Основное содержание понятия модуля — наличие чётко описанного интерфейса между модулем и окружением. Понятие компонента добавляет атомарность развёртывания, а также возможность поставки или удаления компонента отдельно от всей остальной системы.

💡 Возможность включения и исключения компонента из системы делает его отдельным элементом продаваемого ПО.

Компоненты, хотя и не являются законченными продуктами, могут разрабатываться и продаваться отдельно, если они следуют правилам определённой компонентной модели и реализуют достаточно важные для покупателей функции, которые те хотели бы иметь в рамках своей программной системы.

В архитектуре ПО ПК (программный модуль) — произвольный и элемент структуры системы, решающий некоторые подзадачи и взаимодействующий с окружением через определённый интерфейс.

На диаграммах UML ПК (единицы сборки) — файлы с кодом, бинарные файлы, какие-либо документы, входящие в состав системы.

Компоненты развёртывания — блоки, из которых строится компонентное программное обеспечение.

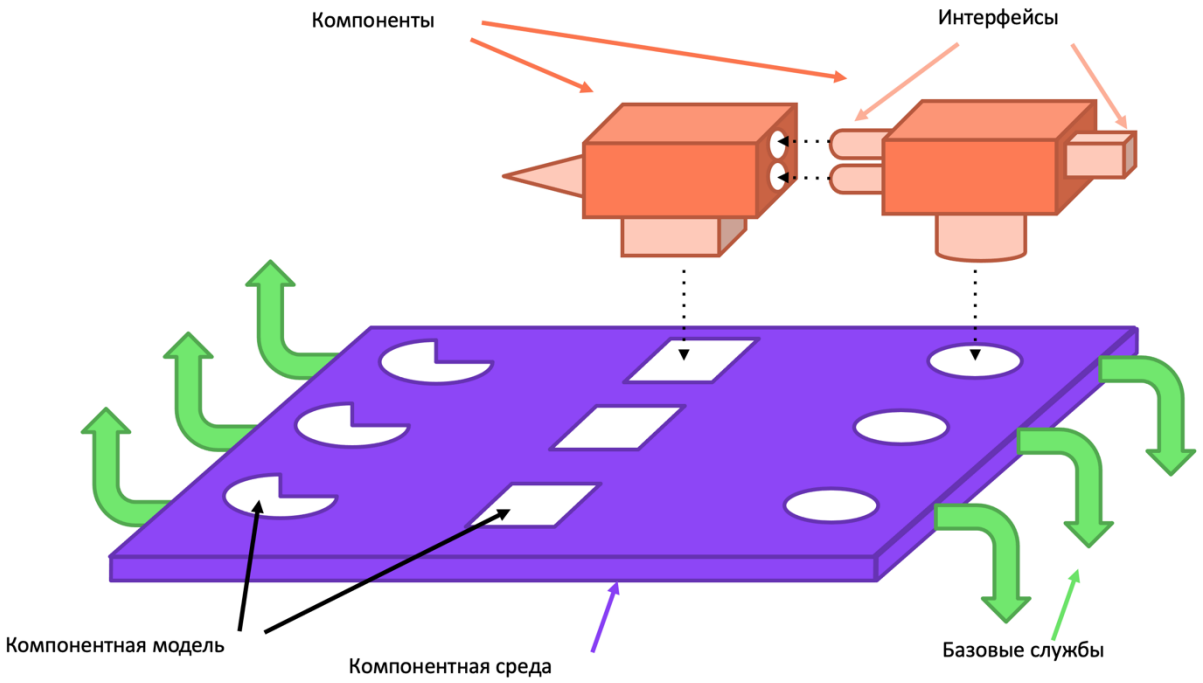
Компонент развёртывания — структурная единица программной системы, обладающая чётко определённым интерфейсом. Такой компонент, независимо от того, удалён он или добавлен в состав некоторой системы, может включаться в состав систем других поставщиков.

Компонентный каркас (component framework)

Состав:

Компонентная модель (component model) — набор правил определения интерфейсов компонентов и их реализаций, а также правил, по которым компоненты работают в системе и взаимодействуют друг с другом. Такие правила регламентируют жизненный цикл компонента.

Набор базовых служб (basic services), обеспечивающих, например, обнаружение компонентов в сетевой среде, передачу данных между компонентами через сетевое взаимодействие.

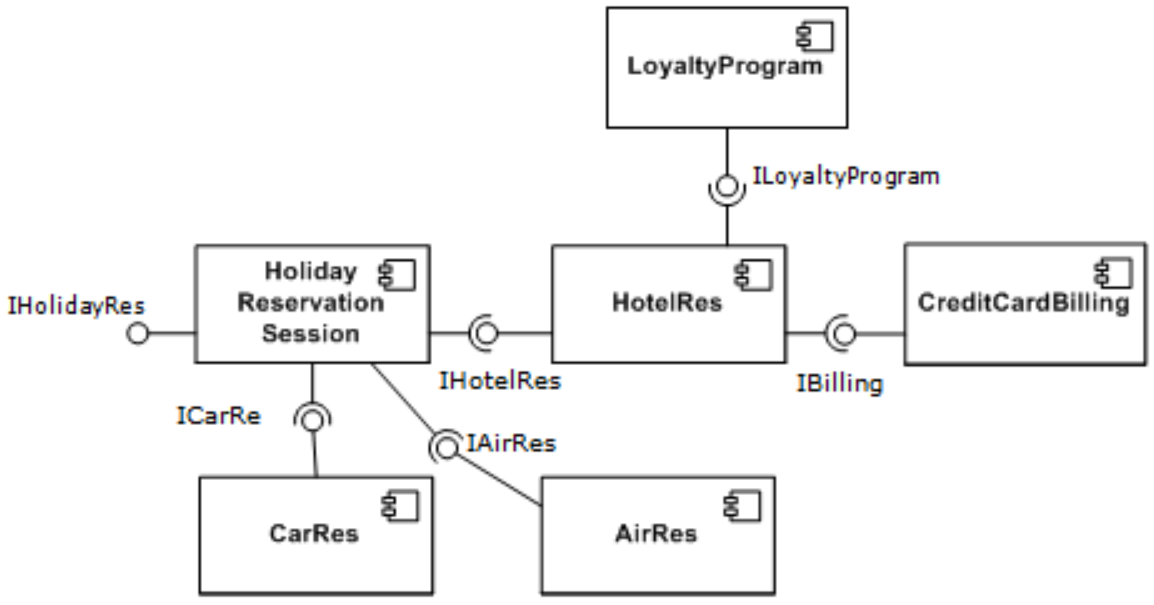


Компонентно-ориентированная разработка

Программная инженерия на основе компонентов (Component-based software engineering — CBSE), также называемая компонентно-ориентированной разработкой (Component-based development — CBD), является направлением программной инженерии, которое подчёркивает разделение проблем в отношении широкой функциональности, доступной в данной программной системе. Это основанный на повторном использовании подход к определению, реализации и компоновке свободно связанных независимых компонентов в системы. Эта практика направлена на получение одинаково широкого спектра преимуществ как в краткосрочной, так и в долгосрочной перспективе — для самого программного обеспечения и для организаций, которые спонсируют такое программное обеспечение.

Специалисты по программной инженерии рассматривают компоненты как часть стартовой платформы для сервис-ориентации. Компоненты играют эту роль, например, в веб-сервисах, а в последнее время — в сервис-ориентированных архитектурах (SOA), где компонент преобразуется веб-сервисом в сервис и впоследствии наследует дополнительные характеристики, выходящие за рамки характеристик обычного компонента.

Компоненты могут производить или потреблять события и могут быть использованы в архитектурах, управляемых событиями (EDA).



Все системные процессы помещаются в отдельные компоненты таким образом, чтобы все данные и функции внутри каждого компонента были семантически связаны (так же, как и содержимое классов). Из-за этого принципа часто говорят, что компоненты являются модульными и сплочёнными.

Что касается общесистемной координации, то компоненты взаимодействуют друг с другом через интерфейсы. Когда компонент предлагает услуги остальной части системы, он принимает предоставленный интерфейс, который определяет услуги, которые могут использовать другие компоненты, и то, как они могут это делать. Этот интерфейс можно рассматривать как подпись компонента — клиенту не нужно знать о внутренней работе компонента (реализации), чтобы использовать его. Этот принцип приводит к тому, что компоненты называются инкапсулированными. Иллюстрации UML в этой статье представляют предоставляемые интерфейсы в виде леденца-символа, прикреплённого к внешнему краю компонента.

Возможность повторного использования — важная характеристика высококачественного программного компонента. Программисты должны разрабатывать и реализовывать программные компоненты таким образом, чтобы многие различные программы могли использовать их повторно. Более того, тестирование юзабилити на основе компонентов должно рассматриваться, когда программные компоненты непосредственно взаимодействуют с пользователями.

Чтобы написать программный компонент, который можно эффективно использовать повторно, требуются значительные усилия и осведомлённость.

Компонент должен быть:

- Полностью документирован.
- Тщательно протестирован.
- Надёжен, снабжён всесторонней проверкой валидности входных данных.
- Способен передавать обратно необходимые сообщения об ошибках или коды возврата.
- Разработан с осознанием того, что он будет использоваться в непредвиденных целях.

В 1960-х годах программисты создавали библиотеки научных подпрограмм, которые можно было многократно использовать в широком спектре инженерных и научных приложений. Хотя эти библиотеки подпрограмм эффективно

использовали чётко определённые алгоритмы, они имели ограниченную область применения. Коммерческие сайты регулярно создавали прикладные программы из многократно используемых модулей, написанных на языке ассемблера, COBOL, PL/1 и других языках второго и третьего поколения, используя как системные, так и пользовательские библиотеки приложений.

Компонентная разработка (CBD) — это процедура, которая подчёркивает проектирование и разработку компьютерных систем с помощью повторно используемых программных компонентов. С CBD акцент смещается с программирования на составление программных систем.

Методы разработки на основе компонентов включают процедуры для разработки программных систем путём выбора идеальных готовых компонентов, а затем их сборки с использованием чётко определённой программной архитектуры. Благодаря систематическому повторному использованию крупнозернистых компонентов, CBD намеревается обеспечить лучшее качество и производительность.

Объектно-ориентированное моделирование приводит к множеству мелкозернистых классов, объектов и отношений. Среди этих небольших устройств очень трудно обнаружить многоразовые детали. Идея, стоящая за КБР, заключается в интеграции связанных частей и их повторном использовании. Эти интегрированные части известны как компоненты.

Методы разработки на основе компонентов состоят из нетрадиционных процедур разработки, включая оценку компонентов, извлечение компонентов и т.д. Важно, чтобы CBD выполнялся в инфраструктуре промежуточного программного обеспечения, которая поддерживает процесс, например, Enterprise Java Beans.

Основные цели КБР

Экономить время и деньги при создании больших и сложных систем. Разработка сложных программных систем с помощью готовых компонентов помогает существенно сократить время разработки программного обеспечения. Функциональные точки или аналогичные методы могут быть использованы для проверки доступности существующего метода.

Повышать качество программного обеспечения. Качество компонентов является ключевым фактором повышения качества программного обеспечения.

Обнаруживать дефекты в системах. Стратегия CBD поддерживает обнаружение ошибок путём тестирования компонентов. Однако поиск источника дефектов — сложная задача в КБР.

Среди преимуществ КБР:

- Минимизированная доставка
    - Поиск по каталогам компонентов
    - Переработка готовых компонентов
- Улучшенная эффективность
    - Разработчики концентрируются на разработке приложений
- Улучшенное качество
    - Разработчики компонентов могут выделить дополнительное время для обеспечения качества
- Минимизированные расходы

Конкретные процедуры КБР:

- Разработка компонентов
- Компонент публикации
- Поиск компонентов, а также поиск
- Компонентный анализ
- Сборка компонентов

Понятие программного компонента как единицы развёртывания

Программный компонент — программная часть системы компонент программного обеспечения.

Применительно к информационным системам распределёнными обычно называют системы с логикой приложения, распределённой между несколькими компонентами системы, каждый из которых выполняется на отдельном компьютере. Однако определение распределённой системы как совокупности компьютеров считается условным, поскольку программное обеспечение, установленное на одном компьютере, может отвечать за выполнение функций, относящихся к разным уровням. Для описания и реализации распределённых систем вводится понятие программного компонента.

Программная компонента — это единица программного обеспечения, исполняемая на одном компьютере в пределах одного процесса и

предоставляющая некоторый набор сервисов. Эти сервисы используются через внешний интерфейс программной компоненты другими компонентами как выполняющимися на этом же компьютере, так и на удалённых компьютерах. Ряд компонент пользовательского интерфейса предоставляют свой сервис конечному пользователю.

Интерфейсы

Интерфейс (Interface) — программная/синтаксическая структура, определяющая отношение между объектами, которые разделяют определённое поведенческое множество и не связаны никак иначе. При проектировании классов разработка интерфейса тождественна разработке спецификации: есть множество методов, которые каждый класс, использующий интерфейс, должен реализовывать.

Интерфейсы, наряду с абстрактными классами и протоколами, устанавливают взаимные обязательства между элементами программной системы, что является фундаментом концепции программирования по контракту (design by contract, DbC). Интерфейс определяет границу взаимодействия между классами или компонентами, специфицируя определённую абстракцию, которую осуществляет реализующая сторона.

Интерфейс в ООП является строго формализованным элементом объектно-ориентированного языка и широко используется в исходном коде программ.

Интерфейсы позволяют наладить множественное наследование объектов и в то же время решить проблему ромбовидного наследования.

Описание и использование интерфейсов

Описание ООП-интерфейса, если отвлечься от деталей синтаксиса конкретных языков, состоит из двух частей: имени и методов интерфейса.

- Имя интерфейса строится по тем же правилам, что и другие идентификаторы используемого языка программирования. Разные языки и среды разработки имеют различные соглашения по оформлению кода, в соответствии с которыми имена интерфейсов могут формироваться по некоторым правилам, которые помогают отличать имя интерфейса от имён других элементов программы.
- Методы интерфейса. В описании интерфейса определяются имена и сигнатуры входящих в него методов, то есть процедур или функций класса.

Использование интерфейсов возможно двумя способами:

- Класс может реализовывать интерфейс. Реализация интерфейса заключается в том, что в описании класса данный интерфейс указывается как реализуемый, а в коде класса обязательно определяются все методы, которые описаны в интерфейсе, в полном соответствии с сигнатурами из описания этого интерфейса. То есть, если класс реализует интерфейс, для любого экземпляра этого класса существуют и могут быть вызваны все описанные в интерфейсе методы. Один класс может реализовать несколько интерфейсов одновременно.
- Возможно объявление переменных и параметров методов как имеющих тип «интерфейс». В такую переменную или параметр может быть записан экземпляр любого класса, реализующего интерфейс. Если интерфейс объявлен как тип возвращаемого значения функции, это означает, что функция возвращает объект класса, реализующего данный интерфейс.

Как правило, в объектно-ориентированных языках программирования интерфейсы, как и классы, могут наследоваться друг от друга. В этом случае интерфейс-потомок включает все методы интерфейса-предка и, возможно, добавляет к ним свои собственные.

Таким образом, с одной стороны, интерфейс — это «договор», обязательный для выполнения классом, реализующим данный интерфейс. С другой стороны, интерфейс — это тип данных, потому что его описание достаточно чётко определяет свойства объектов, чтобы наравне с классом типизировать переменные.

Интерфейсы и абстрактные классы

Интерфейс, с формальной точки зрения, — это просто чистый абстрактный класс, то есть класс, в котором не определено ничего, кроме абстрактных методов.

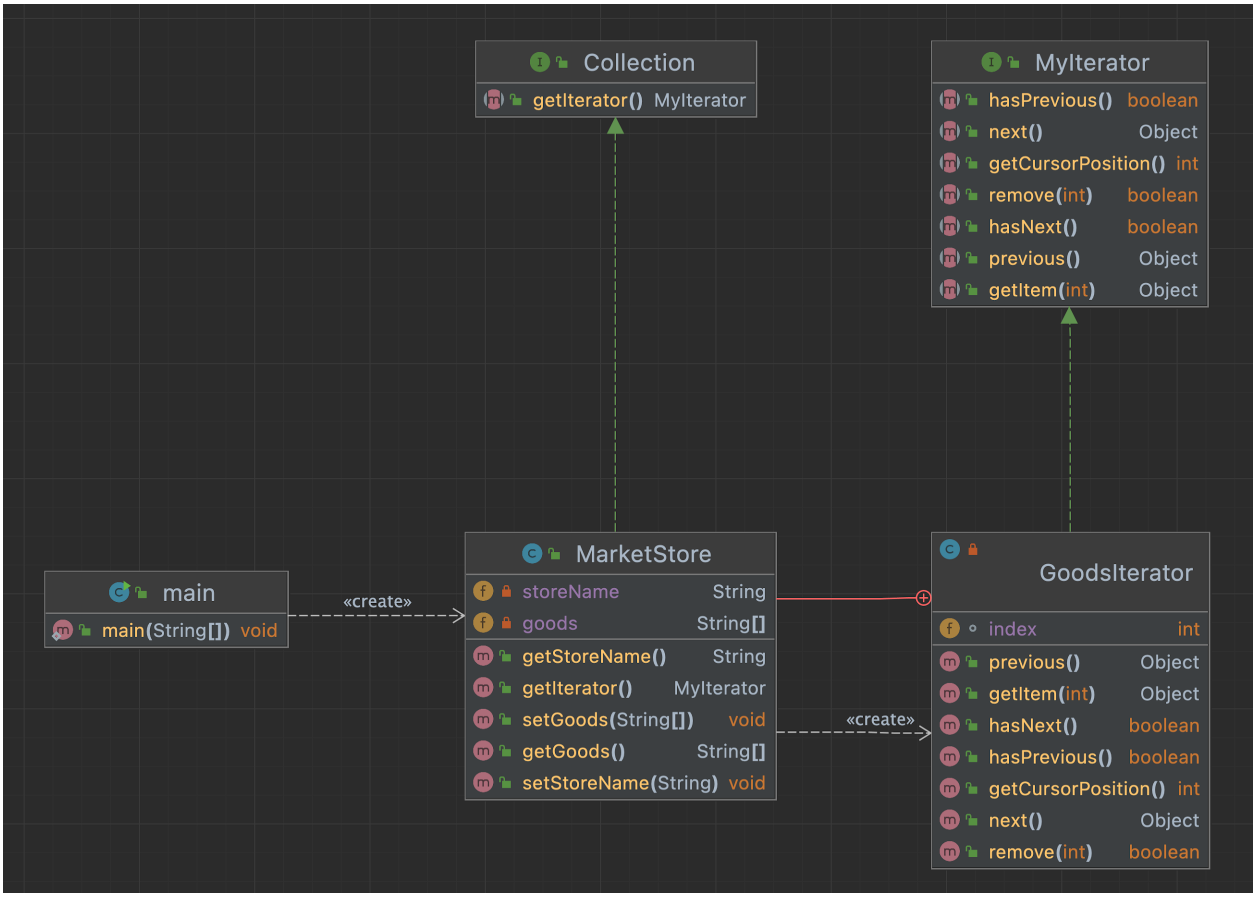
Если язык программирования поддерживает множественное наследование и абстрактные методы (как, например, C++), то необходимости во введении в синтаксис языка отдельного понятия «интерфейс» не возникает. Данные сущности описываются с помощью абстрактных классов и наследуются классами для реализации абстрактных методов.

Пример Java:

```
public interface MyIterator {

    boolean hasNext();
    Object next();

    boolean hasPrevious();
    Object previous();
}
```

```
private class GoodsIterator implements MyIterator {

     int index;

    @Override
    public boolean hasNext() {
        if(index < goods.length){
            return true;
        }
        return false;
    }

    @Override
    public Object next() {
        return goods[index++];
    }

    @Override
    public boolean hasPrevious() {
        if(index > 0){
            return true;
        }
        return false;
    }

    @Override
    public Object previous() {
        index--;
        return goods[index];
```

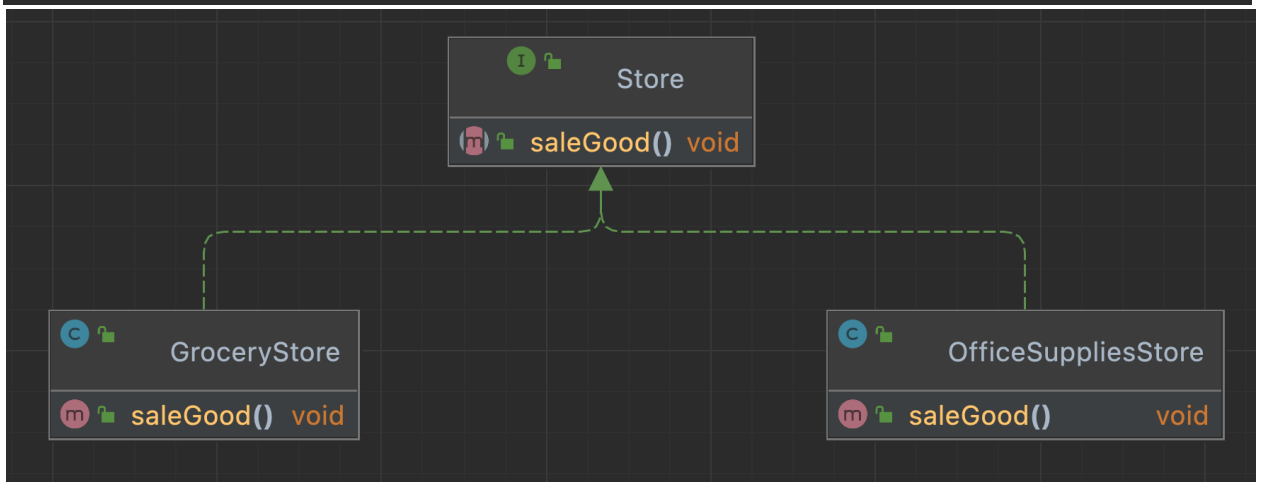
```
    }
}
```




```
public interface StoreFactory {
    Store createStore();
}
```

```
public class GroceryStore implements  Store {
    @Override
    public void saleGood() {
        System.out.println("Tasty food.");
    }
}
```

```java
public class OfficeSuppliesStore implements Store {
    @Override
    public void saleGood() {
        System.out.println("Great tools!");
    }
}
```



Понятие контракта, контрактное программирование

Интерфейс компонента включает набор операций, которые можно вызвать у любого компонента, реализующего данный интерфейс, и набор операций, которые этот компонент может вызвать в ответ у других компонентов.

Это называется **интерфейсным контрактом** или **программным контрактом** компонента.

Проектирование по контракту является сильной методикой, основанной на документировании прав и обязанностей программных модулей для обеспечения корректности программы.

Термин предложил Бертран Мейер (род. 21 ноября 1950, Париж) — разработчик языка Eiffel. Контрактное программирование (1985 год) выросло из формальной верификации, формальной спецификации и логики Хоара. Контрактное программирование — это не только простая метафора, указывающая способ проектирования.

Контрактное программирование (*design by contract* (DbC), *programming by contract*, *contract-based programming*) — это метод проектирования программного обеспечения, который предполагает, что разработчик должен определить формальные, точные и верифицируемые спецификации интерфейсов для компонентов системы.

При этом кроме обычного определения абстрактных типов данных также используются *предусловия*, *постусловия* и *инварианты*. Данные спецификации называются «контрактами» в соответствии с концептуальной метафорой условий и ответственности в гражданско-правовых договорах.

Условия, облегчающие применение контрактного программирования:

- наличие наследования и возможность динамического связывания,
- способность обрабатывать исключения,
- возможность автоматического документирования ПО.

Логика Хоара (***Hoare logic***, также **Floyd—Hoare logic**, или **Hoare rules**) — формальная система с набором логических правил, предназначенных для доказательства корректности компьютерных программ. Была предложена в 1969 году английским учёным в области информатики и математической логики Хоаром, позже развита самим Хоаром и другими исследователями.

Проектирование по контракту (Design by Contract) — это установление отношений между классом и его клиентами в виде формального соглашения, недвусмысленно определяющего права и обязанности сторон. Только через точное определение требований и ответственности для каждого модуля можно надеяться на достижение существенной степени доверия к большим программным системам.

Основная идея контрактного программирования — это модель взаимодействия элементов программной системы, основыванная на идее взаимных *обязательств* и *преимуществ*. Как и в бизнесе, *клиент* и *поставщик* действуют в соответствии с определённым *контрактом*. Контракт некоторого метода или функции может включать в себя:

- **Предусловия** — конкретные обязательства, которые любой клиентский модуль должен выполнить перед вызовом метода. Преимущество для поставщика в том, что он может не проверять выполнение предусловий.
- **Постусловия —** конкретные свойства, которые должны присутствовать после выполнения метода. Входят в обязательства поставщика.
- Обязательства по выполнению конкретных свойств — **инвариантов**, которые должны выполняться при получении поставщиком сообщения, а также при выходе из метода.

В терминах контрактного программирования метод (или функция) обязуется выполнить контракт:

- Если на вход поступили данные, удовлетворяющие входному условию контракта, то метод гарантирует соблюдение выходного условия.
- Если входные данные НЕ удовлетворяют первому условию, то ничего не гарантируется.
- При этом соблюдаются некоторые обобщённые условия.

Спецификация программы и её реализация неразрывно связаны друг с другом. Спецификация говорит о том, *что* должна делать система, а реализация определяет, *как* это будет достигнуто. Оба эти аспекта программного продукта должны развиваться одновременно, чего в большинстве программных проектов не происходит.

Если спецификация и реализация находятся в разных источниках, мы сталкиваемся с классической проблемой дублирования информации, и в этом случае рассогласование такой информации является всего лишь вопросом времени. В проектировании по контракту выражение спецификации осуществляется с помощью механизма утверждений (assertions).

Основой этой концепции являются утверждения. Это булевы выражения, описывающие состояние программы. Можно выделить три основных типа утверждений: предусловия, постусловия и инварианты класса.

Предусловия

Предусловия — это требования подпрограммы, т.е. то, что обязано быть истинным для выполнения подпрограммы. Если данные предусловия нарушены, то подпрограмма не должна вызываться ни в коем случае. Вся ответственность за передачу «правильных» данных лежит на вызывающей программе.

Этот подход противоречит многим концепциям, приводимым в учебниках. Там постоянная проверка вынесена во главу угла. Проектирование по контракту утверждает обратное — лишние проверки могут только навредить. Вообще, принцип проектирования по контракту «смотрит» на проектирования с позиции «Сложность — главный враг качества».

Предусловие операции должно быть выполнено при её вызове, иначе корректность результатов не гарантируется. Если эта операция вызывается у компонента, то обязанность позаботиться о выполнении предусловия лежит на

клиенте, вызывающем операцию. Если же эта операция вызывается компонентом у другого компонента, он сам обязуется выполнить это предусловие.

Постусловия

С постусловием всё наоборот: постусловие вызванной у компонента операции должно быть выполнено после её вызова, и это — обязанность компонента. Постусловие операции определяет, какие её результаты считаются корректными. В отношении вызываемых компонентом операций выполнение их постусловий должно гарантироваться теми компонентами, у которых они вызываются, а вызывающий компонент может на них опираться в своей работе.

Постусловия выражают состояния «окружающего мира» на момент выполнения подпрограммы. Т.е. это условия, которые гарантируются самой подпрограммой. Кроме того, наличие постусловия в подпрограмме гарантирует её завершение, например, не будет бесконечного цикла.

Инварианты класса

Инварианты — это глобальные свойства класса. Они определяют более глубокие семантические свойства и ограничения целостности, характеризующие класс. Класс гарантирует, что данное условие всегда истинно с точки зрения вызывающей программы.

Интерфейсный контракт для каждой операции самого компонента (или используемой им) определяет ***предусловие*** и ***постусловие её*** вызова.

Входное условие называется предусловием (precondition). Выходное условие — постусловием (postcondition). Дополнительно обеспечивается поддержка инвариантов.

Инвариант (invariant) — условие, которое не должно нарушаться из-за выполнения метода, т.е. должно гарантироваться выполнение инварианта до и после выполнения метода. Во время выполнения инвариант может быть временно нарушен.

Пример. Снятие денег в банкомате

- Balance — сумма на счёте клиента
- Cash — сумма, которую клиент хочет снять
- Amount — денежный запас банкомата

Предусловия:

- Cash > 0;
- Cash ≤ 1 000 000;

Постусловия:

- Balance = old(Balance) — Cash;
- Amount = old(Amount) — Cash;

Инвариант:

- Amount ≥ 0;
- Balance ≥ 10 000;

Многие языки программирования позволяют учитывать такие обязательства. Контрактное программирование подразумевает эти требования критическими для корректности программ, поэтому они должны быть утверждены при проектировании. Таким образом, контрактное программирование предписывает начинать писать код с написания формальных утверждений корректности (assertions).

В объектно-ориентированном программировании контракт метода обычно включает следующую информацию:

- возможные типы входных данных и их значение;
- типы возвращаемых данных и их значение;
- условия возникновения исключений, их типы и значения;
- присутствие побочного эффекта метода;
- предусловия, которые могут быть ослаблены (но не усилены) в подклассах;
- постусловия, которые могут быть усилены (но не ослаблены) в подклассах;
- инварианты, которые могут быть усилены (но не ослаблены) в подклассах;
- (иногда) гарантии производительности, например, временная сложность или сложность по памяти.

При использовании контрактов сам код не обязан проверять их выполнение. Обычно в таких случаях в коде делают *жёсткое падение* («*fail-fast*») и тем самым облегчают отладку выполнения контрактов.

В конечном варианте кода это поведение может быть сохранено либо проверки могут быть убраны для повышения производительности.

Юнит-тесты проверяют модуль изолированно от других, проверяя, что модуль удовлетворяет предположениям контракта, а также свои контракты выполняют используемые им модули. Интеграционные тесты проверяют, что модули работают корректно вместе.

💡 **Основная идея:**
«Если клиент, вызывающий подпрограмму, выполняет все предусловия, то вызываемая подпрограмма обязуется, что после её выполнения все постусловия и инварианты будут истинными».

Как и в реальной жизни, при нарушении одного из пунктов контракта наступает заранее обговорённая и согласованная мера. В мире разработки ПО это может быть возбуждение исключения или завершение работы программы. В любом случае вы будете точно знать, что нарушение условий контракта есть ошибка.

Контрактное программирование может:

- повысить уровень повторного использования кода, поскольку обязательства модуля чётко документированы
- обеспечить автоматическое тестирование кода;
- использоваться как способ документации программного обеспечения.

Поддержка DbC на уровне языка

Языки, изначально поддерживающие средства для контрактного программирования:

- Ada 2012
- Clojure
- Eiffel
- Kotlin
- Scala
- C# (с .NET Framework 4.0)

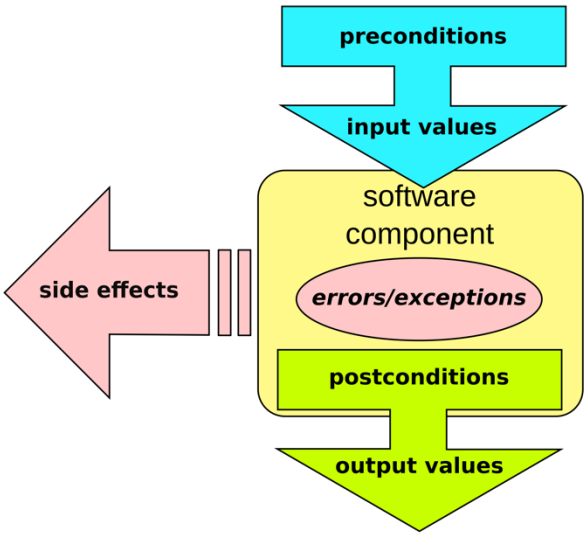
Поддержка DbC с помощью сторонних библиотек

- C и C++ посредством CTESK, библиотеки Contract++, препроцессора *DBC for C*, *GNU Nana* или компилятора C++ от Digital Mars.
- Go посредством DbC.
- Java посредством JavaTESK, iContract2, Contract4J, jContractor, Jcontract, CodePro Analytix, STclass, Jass preprocessor, OVal with AspectJ, Java Modeling Language (JML), SpringContracts для Spring Framework, или Modern Jass, Custos с использованием AspectJ, JavaDbC с использованием AspectJ.
- JavaScript посредством Cerny.js, dbc-code-contracts или ecmaDebug.
- PHP с помощью PhpDeal.
- Python — с помощью пакета zope.interface, PyDBC, PyContracts или Contracts for Python.
- Ruby — с помощью DesignByContract (от Brian McCallister), Ruby DBC или ruby-contract.
- Rust — с помощью библиотеки Hoare.

Проектирование по контракту

Контракт — это поведение и данные, которые согласованы между клиентом и сервисом. Сопровождение контрактов особенно сложно в распределённых архитектурах, в первую очередь из-за разделённых сервисов и систем, принадлежащих разным командам и отделам.

Одним из главных принципов проектирования по контракту является отсутствие проверок предусловий внутри тела программы. Это правило противоречит принципам защитного программирования, в котором «открытые методы класса предполагают, что данные небезопасны и отвечают за их проверку и исправление. Если данные были проверены открытыми методами класса, закрытые методы могут считать, что данные безопасны». При этом внутри открытых методов рекомендуется применять обработчики ошибок (например, генерировать исключение или возвращать соответствующий код ошибки в случае неверных входных данных), а в закрытых методах применять утверждения (assertions) (поскольку это характеризует программные ошибки).



Контракты — сугубо декларативное описание требований

Проектирование по контракту исходит из того, что «чем меньше и конкретнее задача, тем проще разработка, поддержка и сопровождение кода».

Сложность — основная проблема качества. Добавляя избыточные проверки, вы добавляете больше кода. Больше кода — больше сложности, отсюда и больше источников условий, при которых всё может пойти не так.

Структура контракта

Пример структуры контракта между компонентами:

- Набор операций.
- Параметры операций, область допустимых значений, интерпретация.
- Допустимые возвращаемые значения и их интерпретация.
- Сообщения об ошибках и исключениях.
- Предусловия, например, требование по начальной инициализации сервера).
- Постусловия, например, что должны быть выполнены какие-либо операции).
- Инварианты, например, уникальность имён).
- Побочные эффекты (изменение глобальных данных).

- Дополнительные гарантии, например, производительность.
- Статусы состояния.

Использование контрактов

В ООП-языках программирования контракты могут являться часть объектной модели.

При наследовании:

- Предусловия у наследников могут быть ослаблены.
- Постусловия у наследников могут быть усилены.
- Инварианты у наследников могут быть усилены.

Проверка всех условий во время исполнения:

- В случае нарушения — остановка.
- Возможность отключения проверок в релизах.

Статические проверки:

- Вывод с помощью дедуктивных методов.
- Проверка контрактов с помощью методов статического анализа.

Документирование:

- Предусловия + постусловия + интерфейс класса — документирование методов.
- Инварианты + интерфейс класса — документирование классов.

Тестирование:

Контракты могут служить основой для автоматизированного тестирования:

- Предусловия и инварианты — ограничения на генерируемые тесты.
- Постусловия и инварианты — основа для построения тестовых скриптов.

Компонентная диаграмма (Component Diagram)

Для создания конкретной физической системы необходимо реализовать все элементы логического представления в конкретные материальные сущности.

Физическая система (*physical system*) — существующий прототип модели системы в реальном мире.

Диаграмма компонентов — описывает физическое представление системы, связывая его программными компонентами.

Цель диаграммы компонентов — показать взаимосвязь между различными компонентами системы.

Диаграммы компонентов используются для визуализации организации компонентов системы и зависимостей между ними. Они позволяют получить высокоуровневое представление о компонентах системы.

Компонентами могут быть программные компоненты, такие как база данных или пользовательский интерфейс; или аппаратные компоненты, такие как схема, микросхема или устройство; или бизнес-подразделение, такое как поставщик, платёжная ведомость или доставка.

Преимущества диаграмм компонентов

Хотя диаграммы компонентов могут показаться сложными на первый взгляд, они неоценимы, когда дело доходит до создания вашей системы. Диаграммы компонентов могут помочь вашей команде:

- Представить себе физическую структуру системы.
- Обратить внимание на компоненты системы и их взаимосвязь.
- Подчеркнуть поведение сервиса, поскольку оно связано с интерфейсом.
- Используются в компонентно-ориентированных разработках для описания систем с сервис-ориентированной архитектурой.
- Показать структуру самого кода.
- Может использоваться для фокусировки на отношениях между компонентами, скрывая при этом детализацию спецификации.
- Помочь в информировании и разъяснении функций создаваемой системы заинтересованным сторонам.

Как использовать диаграммы компонентов

Диаграмма компонентов в UML даёт представление о вашей программной системе «с высоты птичьего полёта». Понимание точного поведения сервиса, который обеспечивает каждая часть вашего программного обеспечения, сделает вас лучшим разработчиком. Диаграммы компонентов могут описывать программные системы, реализованные на любом языке программирования или в любом стиле.

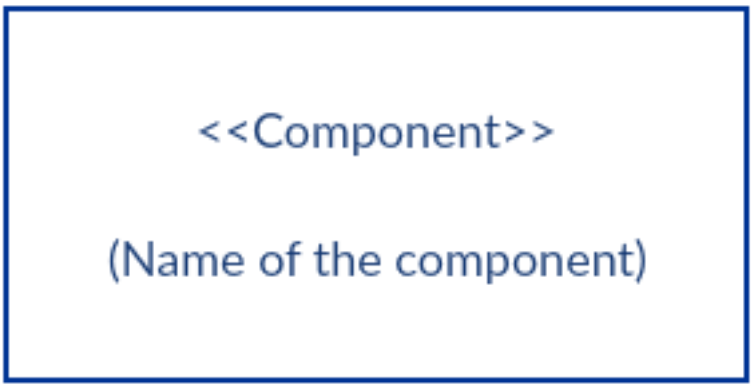
UML — это набор соглашений для объектно-ориентированных диаграмм, который имеет широкий спектр применения. В диаграммах компонентов унифицированный язык моделирования предписывает соединять компоненты и пакеты линиями, представляющими коннекторы сборки и коннекторы делегирования. Чтобы узнать больше о UML и его применении, ознакомьтесь с нашим руководством «Что такое UML?».

Символы на диаграмме компонентов

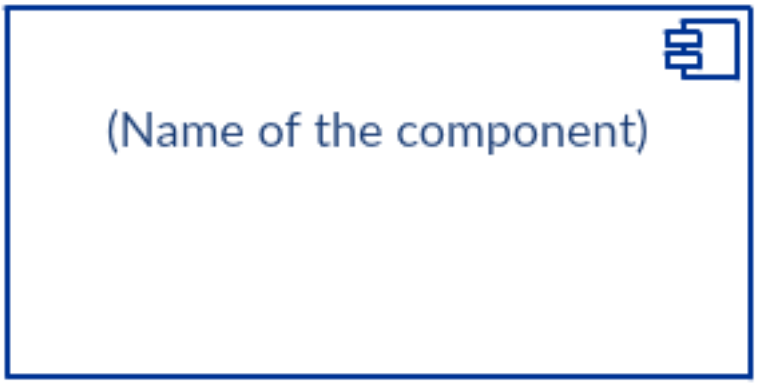
Компонент

Существует три способа использования символа компонента.

1. Прямоугольник со стереотипом компонента (текст <<Component>>). Стереотип компонента обычно используется над именем компонента, чтобы не перепутать форму со значком класса.

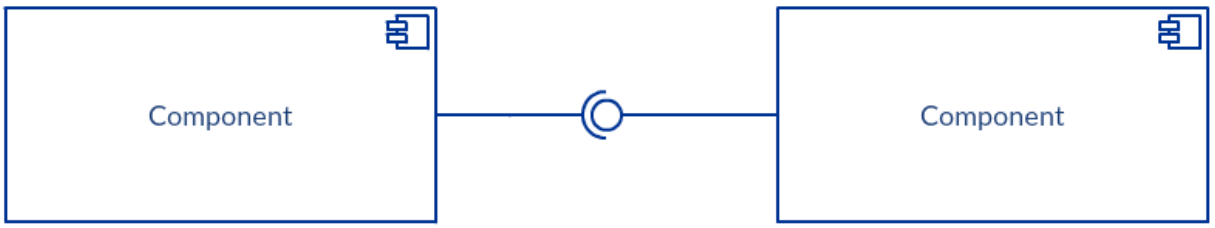


2. Прямоугольник со значком компонента в правом верхнем углу и названием компонента.



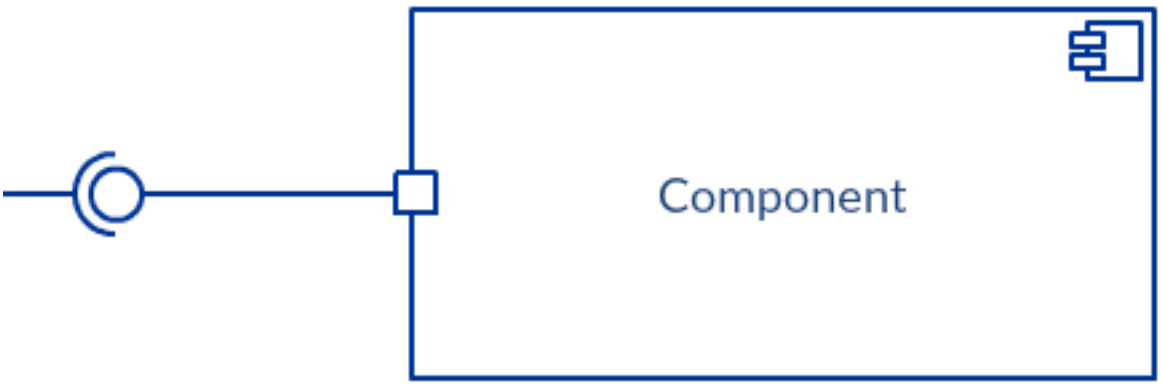
3. Прямоугольник со значком компонента и стереотипом компонента.

Предоставляемый интерфейс и требуемый интерфейс



Интерфейсы на компонентных схемах показывают, как компоненты соединены друг с другом и как взаимодействуют. Соединитель сборки позволяет соединить требуемый интерфейс компонента (представленный полукругом и сплошной линией) с предусмотренным интерфейсом (представленный окружностью и сплошной линией) другого компонента. Это показывает, что один компонент предоставляет услугу, которая требуется другому.

Порт



Порт (представленный маленьким квадратом в конце требуемого интерфейса или предоставляемого интерфейса) используется, когда компонент делегирует интерфейсы внутреннему классу.

Зависимости

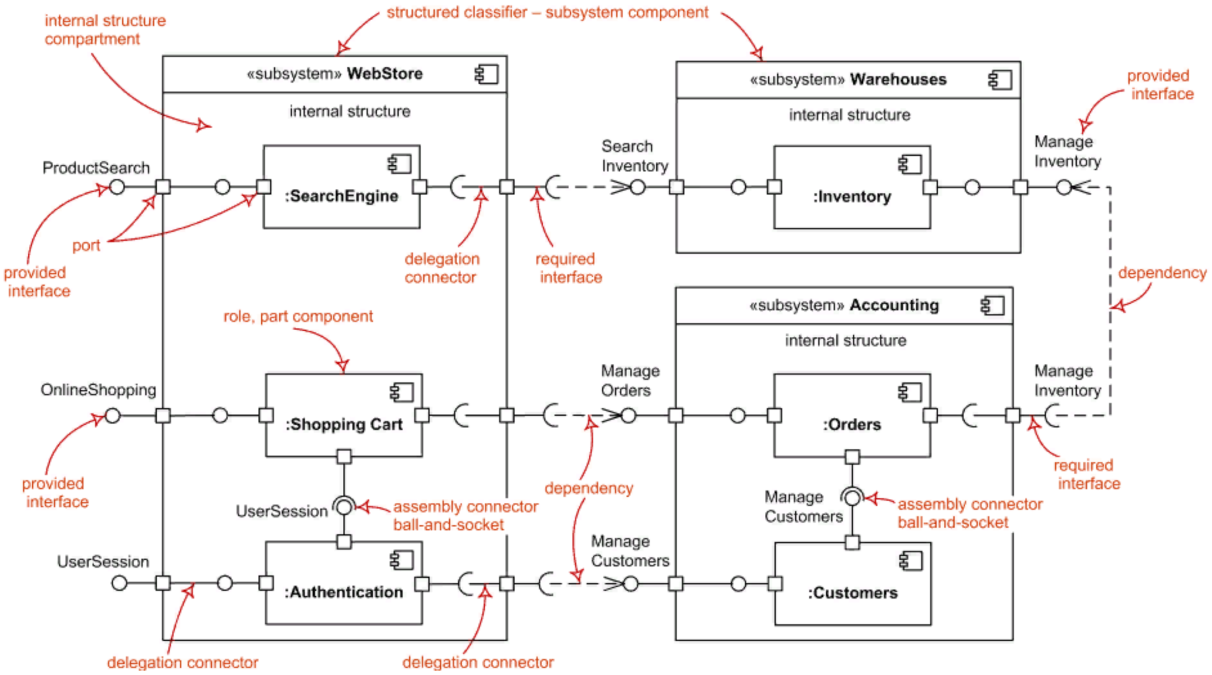
Компонент A зависит от компонента B.

Хотя вы можете показать более подробную информацию об отношениях между двумя компонентами с помощью нотации ball-and-socket (предусмотренный интерфейс и требуемый интерфейс), вы можете также использовать стрелку зависимостей, чтобы показать отношения между двумя компонентами.

В качестве собственных имён *компонентов* принято использовать имена исполняемых файлов, динамических библиотек, веб-страниц, текстовых файлов или файлов справки, файлов баз данных или файлов с исходными текстами программ, файлов скриптов и другие.

Пример компонентных диаграмм

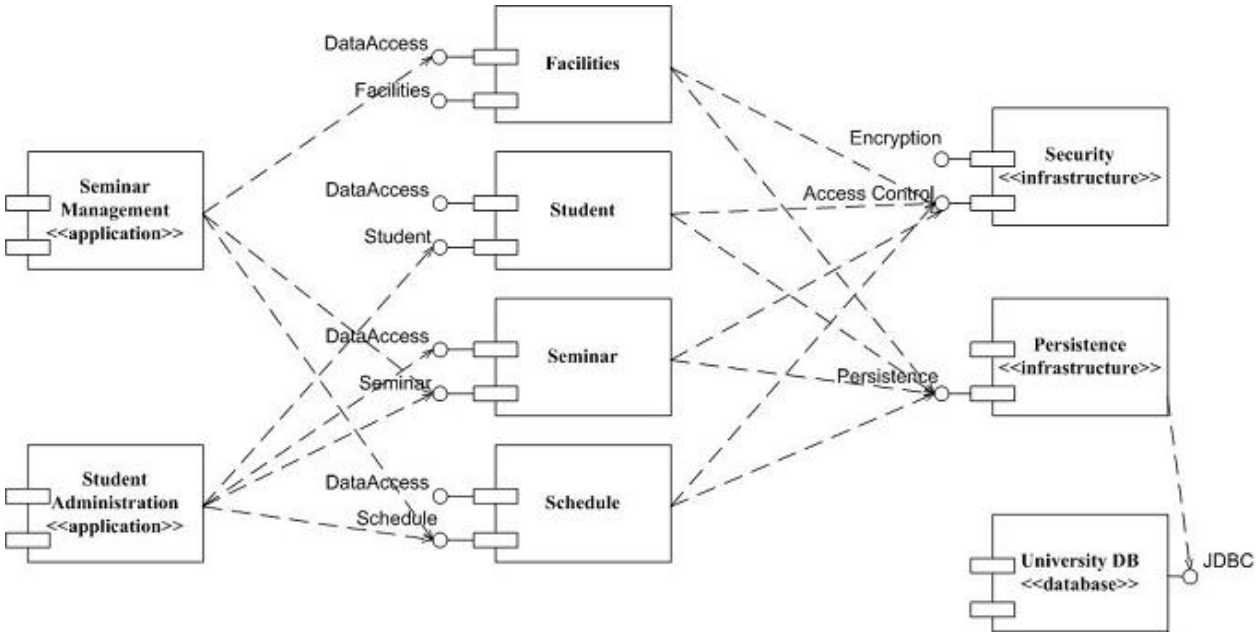
Общая нотация Component Diagramm UML 2



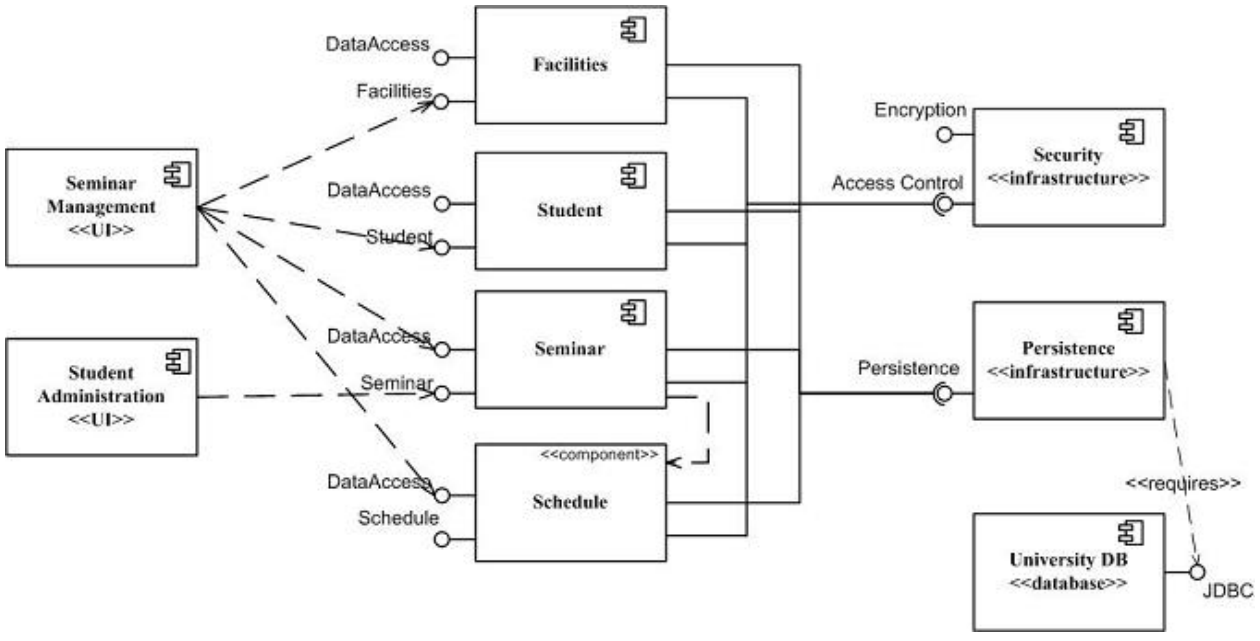
Источник: https://www.uml-diagrams.org/component-diagrams.html

Поскольку *компонент* как элемент модели может иметь различную физическую реализацию, иногда его изображают в форме специального графического символа, иллюстрирующего конкретные особенности реализации.

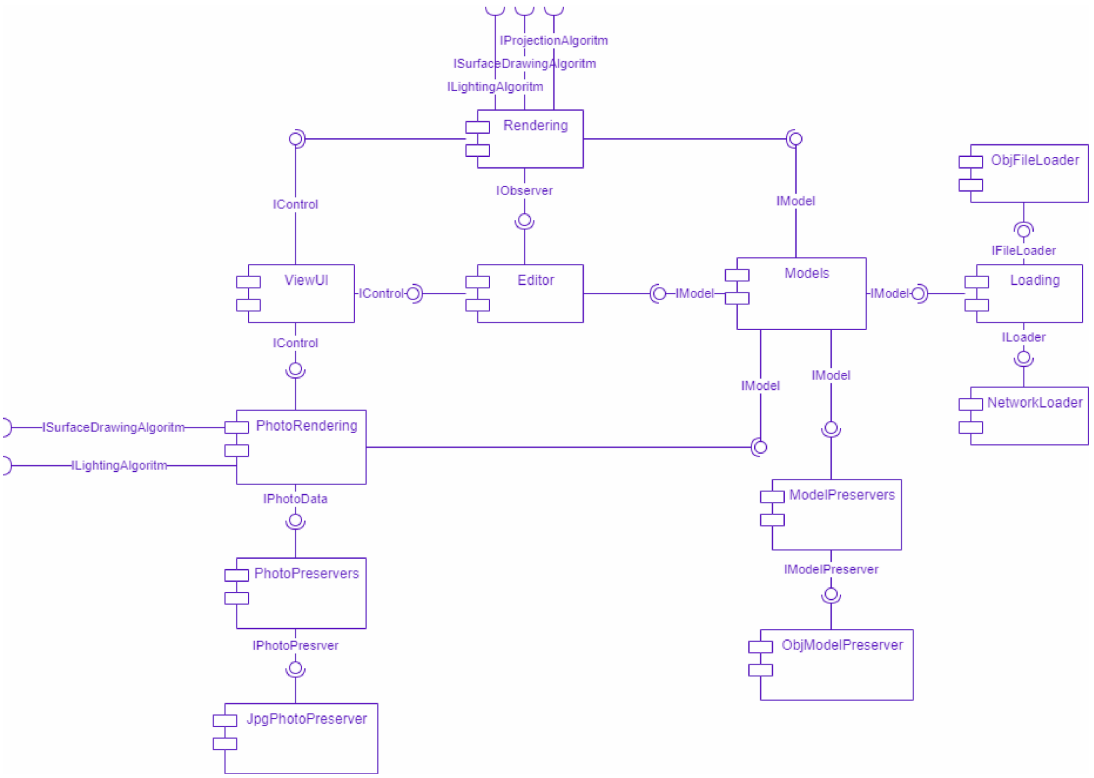
Примеры Component Diagram


DataAccess
Facilities
Facilities
Seminar Management <<application>>
DataAccess
Student
Student
DataAccess
Seminar
Seminar
Student Administration <<application>>
DataAccess
Schedule
Schedule
Encryption
Security <<infrastructure>>
Access Control
Persistence <<infrastructure>>
Persistence
University DB <<database>>
JDBC

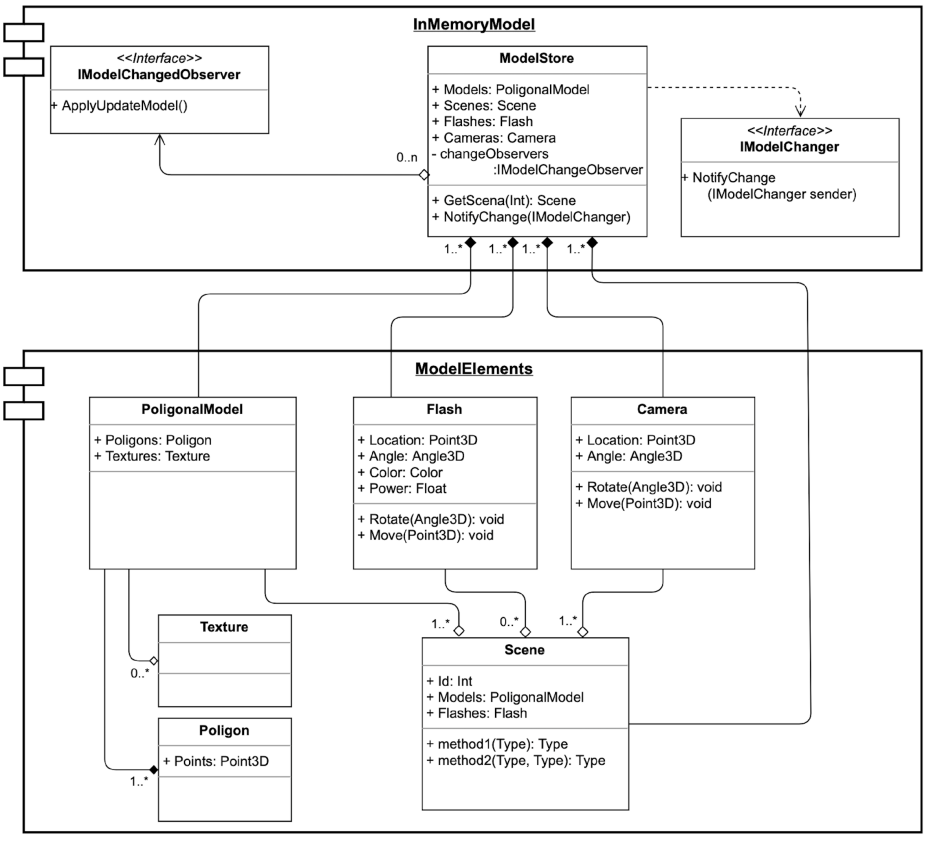
UML 1


DataAccess
Facilities
Facilities
Seminar Management <<UI>>
DataAccess
Student
Student
Student Administration <<UI>>
DataAccess
Seminar
Seminar
<<component>>
DataAccess
Schedule
Schedule
Encryption
Security <<infrastructure>>
Access Control
Persistence <<infrastructure>>
Persistence
<<requires>>
University DB <<database>>
JDBC

UML 2



Пример №1 Диаграммы компонентов в нотации UML 1



Принципы связности программных компонентов (REP, CCP, CRP)

Изначально, **программные компоненты** — динамически связываемые объектные файлы, которые подключаются во время выполнения (связывающий загрузчик). В настоящий момент понятие компонента трактуется более широко.

Принципы связности программных компонентов позволяют принять решение, какие классы и модули стоит включать в программный компонент, а какие — нет:

- REP: Reuse или Release Equivalence Principle — принцип эквивалентности повторного использования и выпусков.
- CCP: Common Closure Principle — принцип согласованного изменения.
- CRP: Common Reuse Principle — принцип совместного повторного использования.

Принцип эквивалентности повторного использования и выпусков (REP)

«Единица повторного использования есть единица выпуска».

«Выпуск»:

- номер версии;
- описание новой версии;
- Change Log (Лог изменений).

Основная идея этого принципа заключается в том, что компоненты должны отдельно выпускаться, версионироваться и отслеживаться для обеспечения возможности повторного использования кода. Автор должен организовать классы в повторно используемые компоненты и затем отслеживать их с помощью выпуска. Без номеров релизов не было бы способа убедиться, что все повторно используемые компоненты совместимы друг с другом.

Принцип REP гласит, что единица повторного использования, компонент, не может быть меньше единицы выпуска. Всё, что предназначено для повторного использования, должно управляться некой системой учёта выпусков. Нельзя,

чтобы разработчик просто написал какой-то класс и объявил его повторно используемым.

О возможности повторного использования можно говорить, когда внедрена некая система учёта выпусков и обеспечены гарантии извещения, надёжности и поддержки, в которых так нуждаются потенциальные пользователи. Объединение в один выпуск должно иметь **смысл** для автора и пользователей.

Важно, чтобы классы и модули, составляющие компонент, принадлежали связной группе. Компонент не включает случайную смесь классов и модулей, должна быть какая-то тема или цель, общая для всех модулей. Помимо этого классы и модули, объединяемые в компонент, требуется выпускать вместе.

Рассмотрим более подробно.

Что такое повторное использование? Считается ли повторным использованием перенос кода из одной программы в другую? Повторное использование — «заимствование» программного модуля у кого-то другого или это связка его с собственными библиотеками? Нет, эти ситуации — не повторное использование кода.

Эти ситуации — просто ситуации копирования кода. Они имеют серьёзный недостаток: мы владеем кодом, который копируем.

Если он не работает в нашей программной среде или системе, требуется изменить его. Когда в коде есть ошибки, их надо исправить. Если автор оригинала находит некоторые ошибки в коде и исправляет их, мы должны выяснить это, а также узнать, как внести соответствующие изменения в собственную копию. В конце концов, скопированный код будет настолько отличаться от оригинала, что его с трудом получится распознать.

В то время как копирование кода облегчает некоторую начальную разработку, это не сильно помогает на самом дорогостоящем этапе жизненного цикла программного обеспечения — сопровождении и техническом обслуживании.

Повторное использование кода происходит, когда не требуется смотреть на исходный код, кроме общедоступных частей заголовочных файлов. Происходит только связывание. Всякий раз, когда эти библиотеки исправляются или расширяются, можно получить новую версию, чтобы затем интегрировать в свою систему.

Есть возможность рассматривать код, который используется повторно, как отдельный, независимый продукт. Он поддерживается и распространяется

автором. Автор несёт ответственность за его содержание. Тот, кто использует этот код, считается заказчиком.

Когда библиотеки, которые используются повторно, изменяются автором, все заказчики должны быть уведомлены об этом. Они могут решить использовать старую версию библиотеки на некоторое время.

Таким образом, нельзя повторно использовать то, что не выпущено. Когда кто-то повторно использует что-то в выпущенной библиотеке, он фактически становится клиентом всей библиотеки. Независимо от того, повлияют ли изменения на клиента или нет, ему придётся интегрироваться с каждой новой версией библиотеки, когда она выйдет, чтобы воспользоваться более поздними улучшениями и исправлениями.

Принцип согласованного изменения (CCP)

«В один компонент должны включаться классы, изменяющиеся по одним причинам и в одно время. В разные компоненты должны включаться классы, изменяющиеся в разное время и по разным причинам».

Варианты мест изменения кода в приложении:

- все в одном компоненте;
- по многим компонентам.

Логичнее видеть изменения сосредоточенными в одном компоненте, а не искать их по всему проекту. Таким образом, нужно выпустить только один изменённый компонент. Другие компоненты, которые не зависят от изменённого, не нуждаются в повторной проверке или повторном выпуске.

По сути, это принцип единственной ответственности (SRP) из SOLID, но перефразированный для компонентов. Так же, как принцип SRP, гласящий, что класс не должен иметь нескольких причин для изменения. Принцип согласованного изменения (CCP) требует, чтобы компонент не имел нескольких причин для изменения.

Этот принцип говорит нам, какие классы следует объединять в группы. Многократно используемые классы с большей вероятностью будут зависеть друг от друга, поэтому классы редко используются повторно в разделении. CRP утверждает, что классы компонента должны быть неразделимы, что означает, что если компонент зависит от этого компонента, то он должен зависеть от всех его классов, а не от нескольких. Короче говоря, классы, которые не являются тесно связанными друг с другом, не должны находиться в одном компоненте.

Это важно, поскольку в случае изменения кода мы предпочли бы, чтобы все изменения произошли в одном компоненте, а не были распределены по многим компонентам. Таким образом, если изменения ограничиваются одним компонентом, то нам нужно переразвернуть только один изменённый компонент. Другие компоненты, которые не зависят от изменённого компонента, не нужно перепроверять или переразвёртывать.

Принцип требует собирать все классы, чтобы изменить по общей причине. Если два класса тесно связаны, физически или концептуально, настолько, что всегда будут изменяться вместе, они должны принадлежать одному компоненту. Это позволит уменьшить трудозатраты, имеющие отношение к повторному выпуску, тестированию и развёртыванию программного обеспечения.

Этот принцип тесно связан с принципом открытости и закрытости (Open Closed Principle). Фактически он означает «закрытость» в смысле принципа OCP. Принцип открытости и закрытости (OCP) требует, чтобы классы были закрыты для изменений, но открыты для расширения. Так как 100% закрытость невозможна, она должна носить скорее стратегический характер. Требуется проектировать свои классы так, чтобы они закрывались для изменений, наиболее типичных из ожидаемых по опыту.

Принцип согласованного изменения (CCP) развивает эту идею, предписывая объединять в один компонент только закрытые для одного вида изменений классы. То есть увеличивается вероятность, что изменение требований повлечёт изменения минимального количества компонентов.

Принцип совместного повторного использования (CRP)

«Не вынуждайте пользователей компонента зависеть от того, чего им не требуется».

Принцип совместного повторного использования (Common Reuse Principle) — ещё один принцип, позволяющий определять, какие классы и модули включаются в компонент. Он указывает, что в компонент войдут классы и модули, используемые совместно.

Общий принцип повторного использования (Common Reuse Principle, CRP) помогает нам, говоря, что мы не должны заставлять наших пользователей зависеть от вещей, которые они не собираются использовать. Поэтому мы должны помнить, что модули в компоненте будут использоваться вместе. Это означает, что мы хотим убедиться, что классы, которые мы помещаем в компонент, неразделимы — невозможно зависеть от одних и не зависеть от

других. В противном случае мы будем перераспределять больше компонентов, чем необходимо, и тратить значительные усилия.

Классы редко используются по отдельности. Обычно многократно используемые классы взаимодействуют с другими классами, считающиеся частью многократно используемой абстракции. Принцип CRP указывает, что такие классы включаются в один компонент. Но принцип CRP говорит не только об этом, он также сообщает, каким классам нельзя объединяться.

Когда образуется зависимость от компонента, желательно, чтобы она распространялась на все классы в этом компоненте. То есть, классы, включаемые в компонент, должны быть неотделимы друг от друга — чтобы не зависеть ни от одних, ни от других. Иначе придётся повторно развёртывать больше компонентов, чем требуется, и тратить существенно больше усилий.

Простейший пример — класс коллекции и связанные с ним итераторы. Эти классы используются вместе, потому что они тесно связаны друг с другом. Соответственно, им следует находиться в одном компоненте.

Если в подобный компонент, помимо коллекции и итераторов, добавить алгоритмы вычисления арифметических выражений (даже использующих коллекции), другие пользователи, использующие такой компонент только для коллекций, будут зависеть от нужного им кода. При обновлении алгоритма вычисления выражений они станут получать новую версию всего компонента и будут вынуждены обновлять своё решение целиком.

Противоречие

В целом, принцип эквивалентности повторного использования/выпуска заключается в группировке вещей для повторного использования, принцип общего закрытия сосредоточен на группировке для повышения сопровождаемости, а принцип повторного использования Closure говорит нам о разделении вещей, чтобы они зависели только от того, что вам нужно.

Три принципа связности компонентов вступают в противоречие друг с другом.

Принципы эквивалентности повторного использования (REP) и согласованного изменения (CCP) считаются включительными: оба стремятся сделать компоненты как можно крупнее.

Принцип повторного использования (CRP) — исключительный, стремящийся сделать компоненты как можно мельче. Задача архитектора — разрешить это противоречие.

На рисунке изображена диаграмма противоречий, показывающая, как три принципа связности влияют друг на друга. Рёбра на диаграмме описывают цену нарушения принципа на противоположной вершине.

Диаграмма противоречий принципов связности

Как же решить это противоречие? Требуется найти золотую середину, отвечающую текущим нуждам разработки конкретной системы, а также подумать об изменениях, которые могут произойти в будущем. Например, на ранних этапах разработки проекта принцип CCP намного важнее, чем REP, удобство разработки важнее удобства повторного использования.

Обычно, в начале разработки наибольшую важность имеет правая сторона треугольника, когда проблемой будет повторное использование. Но по мере развития и интеграции в другие проекты фокус начинает смещаться влево. Таким образом, организация компонентов в проекте изменяется с течением времени. Это больше связано с тем, как проект разрабатывается, развивается и используется, нежели с функциональными возможностями системы.

Принципы сочетаемости программных компонентов

Если принципы связности компонентов говорили о том, какие классы и модули включать или не включать в компоненты, то принципы сочетаемости указывают, как связывать компоненты между собой.

Принцип ацикличности зависимостей

«Циклы в графе зависимостей компонентов недопустимы».

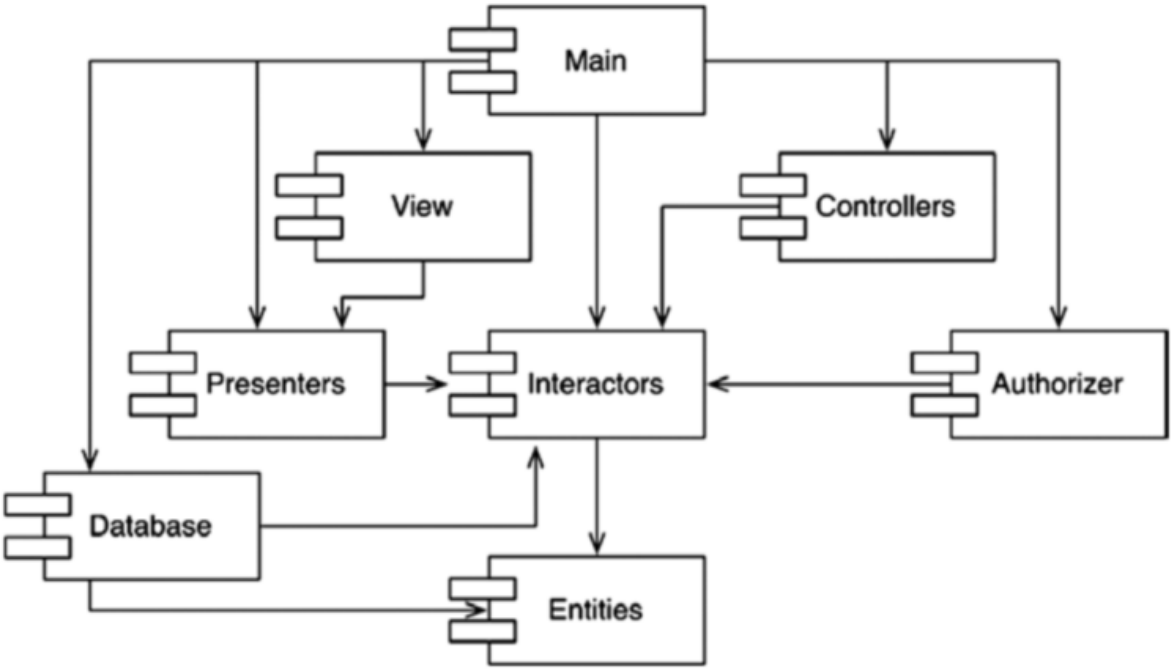
Проект требуется разделять на компоненты, выпускаемые независимо. Компоненты становятся единицами работы, ответственность за которые возлагается на некоторого разработчика или на группу. Когда разработчики добиваются работоспособности компонента, они выпускают новую версию для использования другими разработчиками.

Этой версии присваивается номер, и она помещается в каталог, доступный другим разработчикам. Затем разработка компонента продолжается, с изменением локальных копий. А все пользователи компонента используют выпущенную ранее версию.

Когда появляется новая версия компонента, другие команды могут выбирать — сразу же задействовать новую версию или подождать. Если принято решение подождать, они просто продолжают использовать предыдущую версию. Но как только они решат, что готовы, то начинают использовать новую версию.

Изменения в одном компоненте не оказывают немедленного влияния на другие команды. Каждая команда сама решает, когда начать адаптацию своего компонента для использования новой версии другого компонента. Интеграция происходит небольшими шагами. Нет единого момента времени, когда все разработчики должны собраться вместе и интегрировать всё, что они создали.

Посмотрим на диаграмму компонентов. Она демонстрирует типичную структуру компонентов, собранных в приложение. В этом случае назначение приложения не так важна, как сама структура зависимостей компонентов. При этом структура имеет вид ориентированного (направленного) графа. Компоненты — это узлы, а зависимости между ними — ориентированные рёбра.



Ориентированный ациклический граф

Зависимости между компонентами располагаются так, что с какого компонента ни начни движение, обратно не вернёшься.

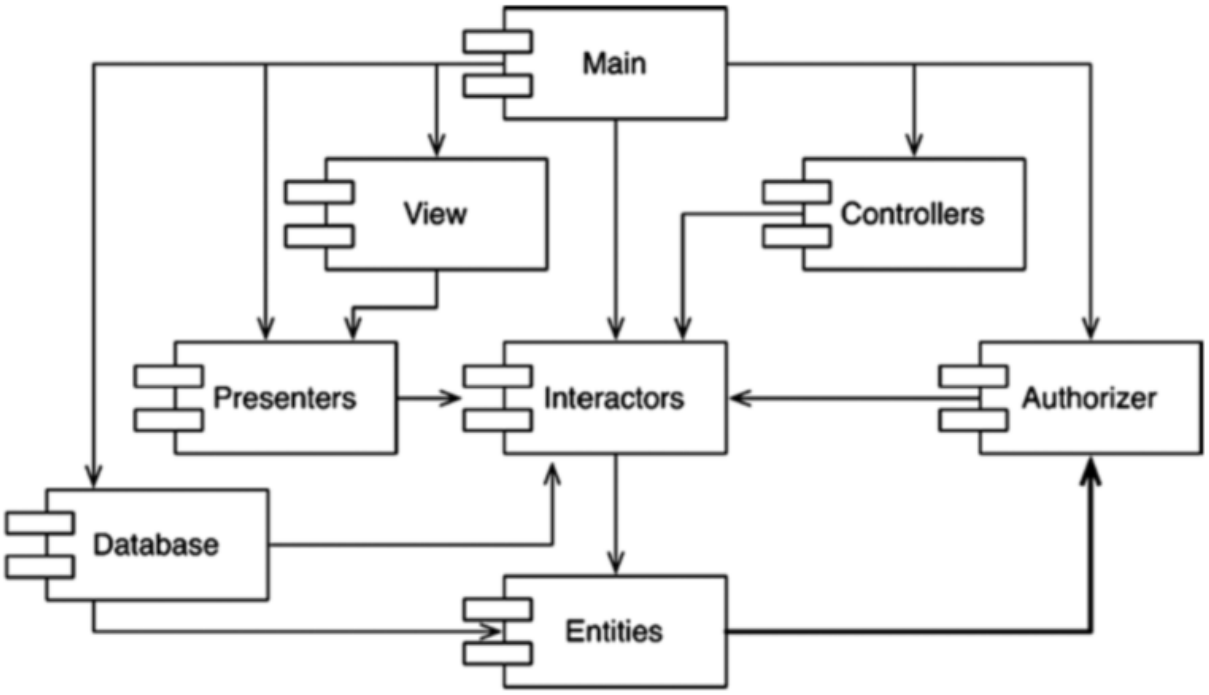
Если одна из команд выпустит новую версию компонента Presenters, то будут затронуты только компоненты View и Main. В этот момент разработчики, развивающие эти компоненты, должны решить, когда начать их интеграцию с новой версией Presenters.

Выпуск новой версии компонента Main не затронет никакой другой компонент в системе. Это означает, что выпуск новой версии Main оказывает минимальное влияние.

Когда разработчики компонента Presenters пожелают протестировать его, им просто потребуется собрать собственную версию Presenters с версиями компонентов Interactors и Entities, используемыми в конкретный момент. Никакой другой компонент в системе им для этого не понадобится.

Когда придёт время выпустить новую версию всей системы, процесс будет протекать снизу вверх. Сначала компилируется, тестируется и выпускается компонент Entities. Затем те же действия будут выполнены с компонентами Database и Interactors. За ними последуют Presenters, View, Controllers, и затем Authorizer. И наконец, очередь дойдёт до Main. Мы знаем, в каком порядке собирать систему, потому что понимаем, как связаны зависимостями отдельные её части.

Предположим, что появление новых требований вынудило нас изменить один из классов в компоненте Entities, поэтому он стал использовать класс из компонента Authorizer. Например, класс User из Entities стал использовать класс Permissions из Authorizer. В результате образовалась циклическая зависимость, как показано на рисунке ниже.



Ориентированный граф с циклами

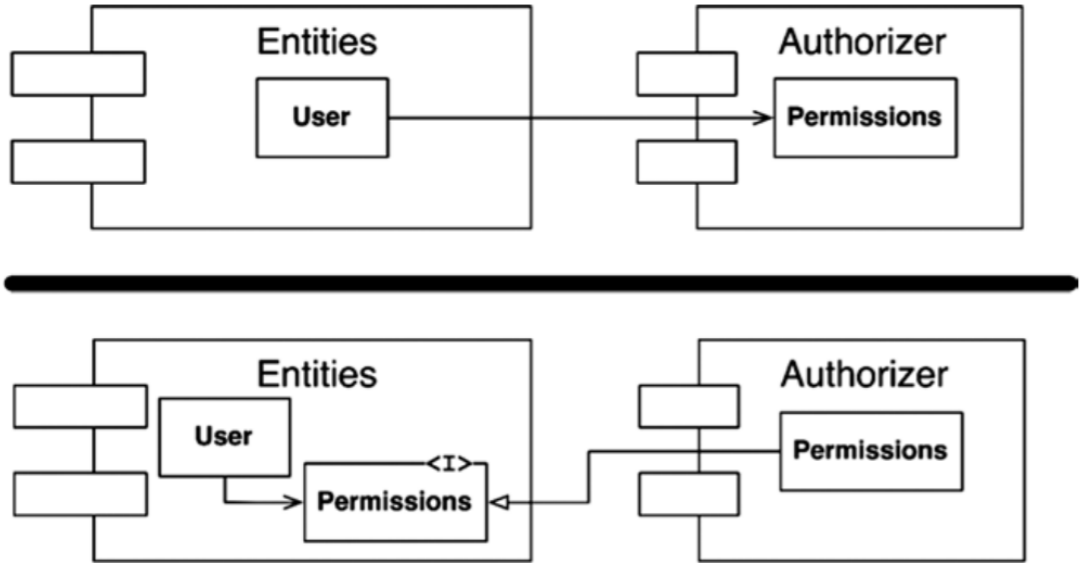
Циклы в зависимостях делают код более хрупким. И возникает множество сложностей:

- с разработкой, например, Entities, Authorizer и Interactors фактически превращаются в один большой компонент;
- тестированием, где степень зависимости компонентов вызывает беспокойство, если она вообще возможна;
- сборкой — нет правильного порядка сборки.

Образовавшуюся циклическую зависимость всегда можно разорвать и привести граф зависимостей к форме ациклического ориентированного графа. Для этого используются два основных механизма:

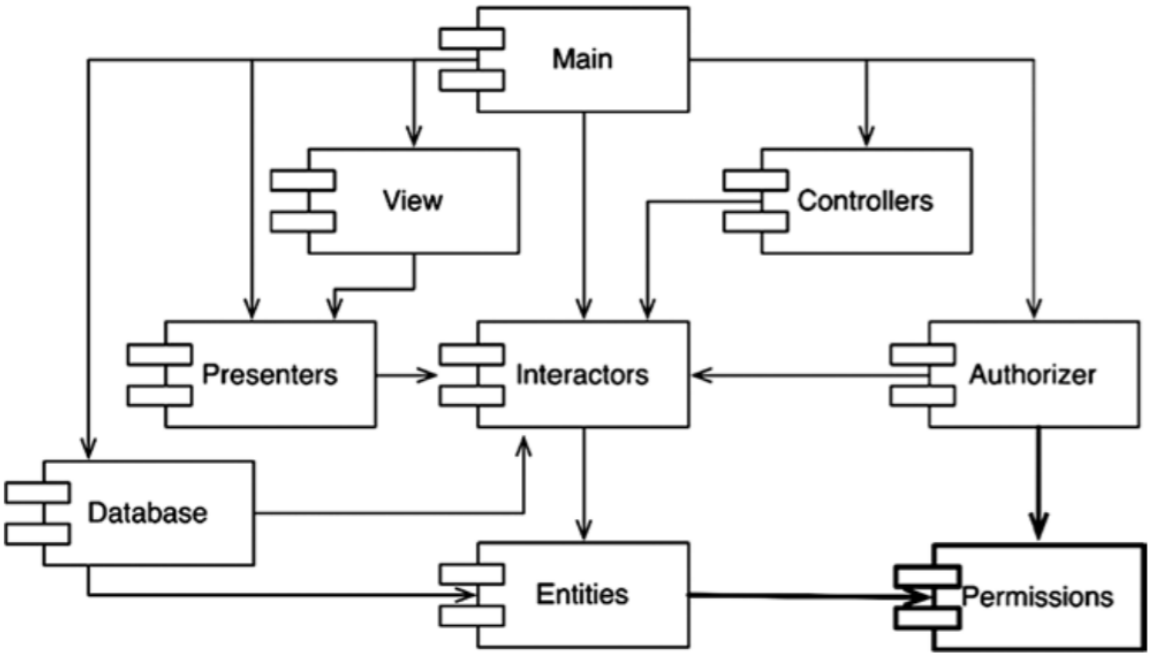
1. Применить принцип инверсии зависимостей — Dependency Inversion Principle (DIP). В этом случае логичнее создать интерфейс, определяющий методы, требуемые классу User, затем поместить этот интерфейс в Entities

и унаследовать его в Authorizer. Это обратило бы зависимость между Entities и Authorizer и разорвало цикл.



Инверсия зависимости между Entities и Authorizer

2. Создать новый компонент, от которого зависят Entities и Authorizer. Поместить в новый компонент класс (ы), от которых они оба зависят.



Новый компонент, от которого зависят Entities и Authorizer

Принцип устойчивых зависимостей

«Зависимости должны быть направлены в сторону устойчивости».

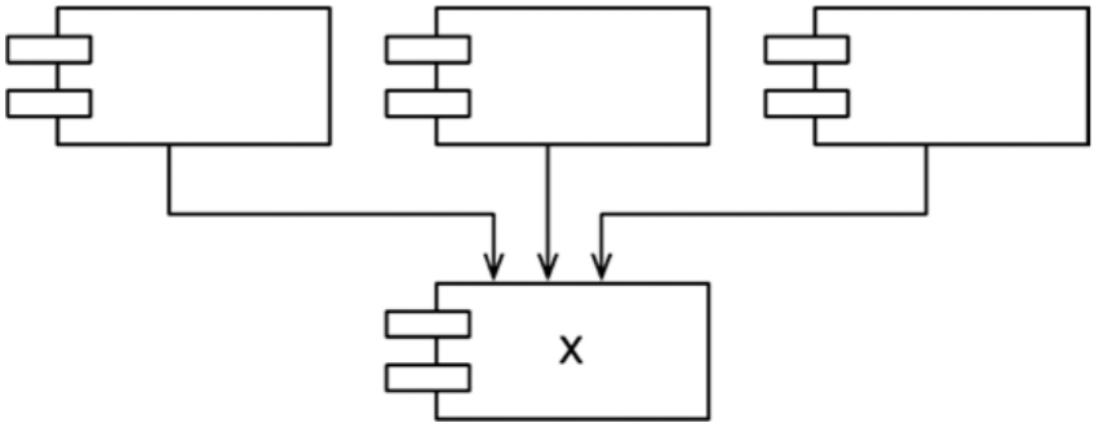
Дизайн не может оставаться статичным. Некоторая изменчивость всё равно необходима, если предполагается сопровождать дизайн. Следуя принципу согласованного изменения (CCP), мы создаём компоненты, чувствительные к одним изменениям и невосприимчивые к другим.

Компоненты, с большим трудом поддающиеся изменению, не должны зависеть от любых изменчивых компонентов. Иначе изменчивый компонент тоже станет трудно изменять.

Устойчивость — это «способность сохранять своё состояние при внешних воздействиях». Если текущее состояние при этом не сохраняется, то его называют неустойчивым.

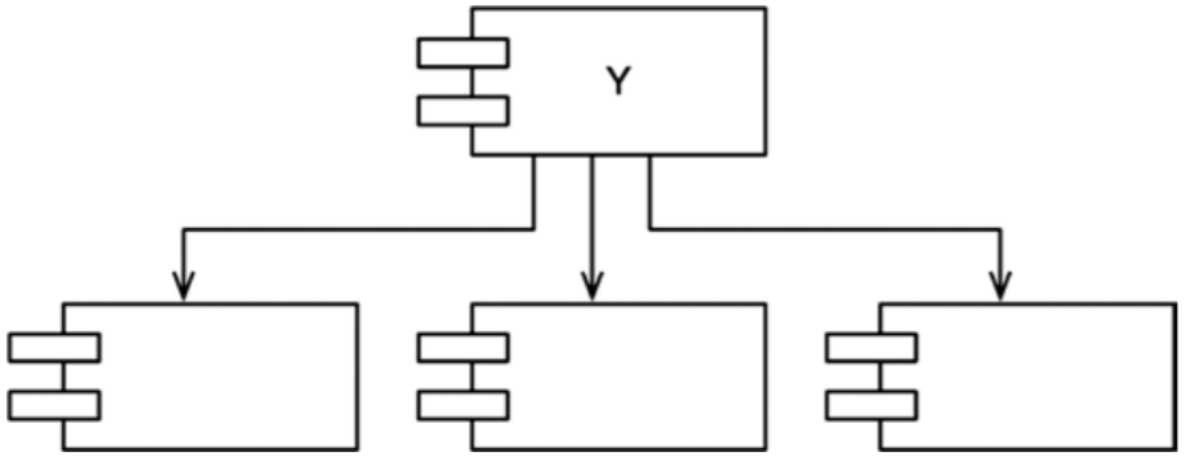
Есть один верный способ сделать программный компонент сложным для изменения — создать много других компонентов, зависящих от него. Компонент с множеством входящих зависимостей очень устойчив, потому что согласование изменений со всеми зависящими компонентами требует значительных усилий.

На рисунке ниже представлена диаграмма с устойчивым компонентом X. От него зависят три других компонента, то есть имеются три веские причины не изменять его. X несёт ответственность за эти три компонента. Сам компонент X, напротив, ни от чего не зависит, то есть на него нет никаких внешних воздействий, которые могли бы привести к изменению. X — независим.



X — устойчивый компонент

На рисунке ниже изображён очень неустойчивый компонент Y. От него не зависит никакой другой компонент, поэтому он лишён ответственности. Есть также три компонента, от которых зависит Y, поэтому необходимость его изменения проистекает из трёх внешних источников. Y — зависим.



Y — очень неустойчивый компонент

Один из способов определить устойчивость — подсчитать количество входящих и исходящих зависимостей этого компонента. Эти числа позволят вычислить меру его устойчивости.

1. **Fan-in (число входов):** количество входящих зависимостей. Такая метрика определяет количество классов вне конкретного компонента, которые зависят от классов внутри компонента.
2. **Fan-out (число выходов):** количество исходящих зависимостей. Эта метрика определяет количество классов внутри конкретного компонента, зависящих от классов за его пределами.
3. **Метрика неустойчивости I:** $\frac{Fan-out}{Fan-in + Fan-out}$. Значение этой метрики изменяется в диапазоне [0, 1]. I = 0 соответствует максимальной устойчивости компонента, I = 1 — максимальной неустойчивости.

Принцип устойчивых зависимостей (SDP) говорит, что метрика неустойчивости компонента должна быть выше метрик неустойчивости компонентов, от которых зависит.

Если все компоненты в системе будут иметь максимальную устойчивость, такую систему невозможно будет изменить. Это нежелательная ситуация. В действительности структура компонентов должна проектироваться так, чтобы в ней были и устойчивые, и неустойчивые компоненты.

Принцип устойчивости абстракций

«Устойчивость компонента пропорциональна его абстрактности».

Некоторые части программных систем меняются редко. Эти части представляют высокоуровневые архитектурные и другие важные решения. Никто не желает, чтобы такие решения были изменчивыми. Поэтому программное обеспечение, инкапсулирующее высокоуровневые правила, находится в устойчивых компонентах (I = 0), а неустойчивые (I = 1) — содержать только изменчивый код, который можно легко и быстро изменить.

Однако, если высокоуровневые правила поместить в устойчивые компоненты, это усложнит изменение исходного кода, реализующего их. Это может сделать всю архитектуру негибкой. Чтобы компонент с максимальной устойчивостью стал гибким и сохранял устойчивость при изменениях, требуется соблюдать принципа открытости и закрытости, то есть использовать абстрактные классы.

Принцип устойчивости абстракций — Stable Abstractions Principle (SAP) — устанавливает связь между устойчивостью и абстрактностью. Он говорит, что устойчивый компонент должен быть абстрактным, чтобы его устойчивость не препятствовала расширению, а неустойчивый — конкретным, так как неустойчивость позволяет легко изменять его код.

То есть стабильный компонент должен состоять из интерфейсов и абстрактных классов, чтобы его легко было расширять. Устойчивые компоненты, доступные для расширения, обладают достаточной гибкостью. Это позволяет не накладывать чрезмерные ограничения на архитектуру.

Принципы устойчивости абстракций (SAP) и устойчивых зависимостей (SDP) вместе соответствуют принципу инверсии зависимостей (DIP) для компонентов. Это верно, принцип SDP требует, чтобы зависимости были направлены в сторону устойчивости. При этом принцип SAP утверждает, что устойчивость подразумевает абстрактность. То есть зависимости должны быть направлены в сторону абстрактности.

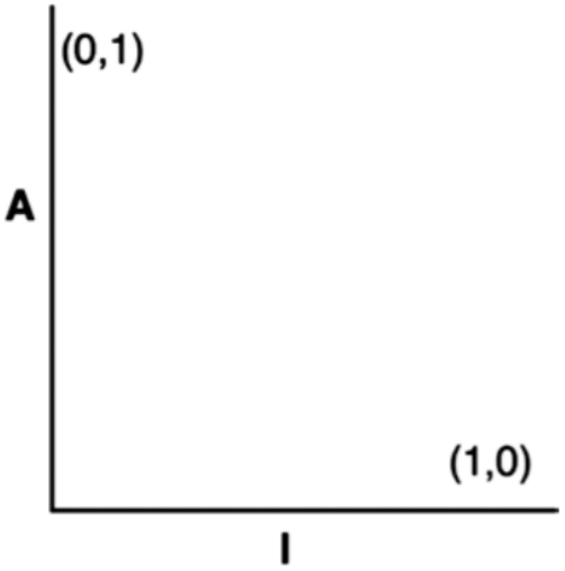
Мерой абстрактности компонента служит метрика *A*. Значение этой метрики определяется простым отношением количества интерфейсов и абстрактных классов к общему числу классов в компоненте.

1. *Nc*: число классов в компоненте.
2. *Na*: число абстрактных классов и интерфейсов в компоненте.

3. *A*: абстрактность. *A* = *Na* ÷ *Nc*.

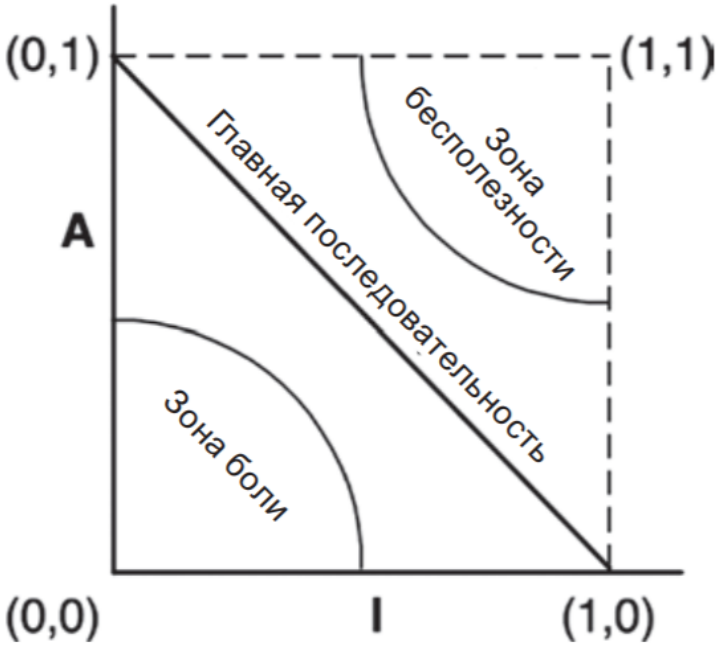
Значение метрики *A* изменяется в диапазоне от 0 до 1. 0 означает полное отсутствие абстрактных классов в компоненте, а 1 — что компонент не содержит ничего, кроме абстрактных классов.

Зависимость между устойчивостью (I) и абстрактностью (A)



Если нанести на график «хорошие» компоненты обоих видов, обнаружится, что максимально устойчивые и абстрактные находятся слева вверху, в точке с координатами (0, 1), а максимально неустойчивые и конкретные — справа внизу, в точке (1, 0).

Предположим, что на графике A/I имеется несколько точек, определяющих оптимальные позиции для компонентов. Вывести это множество можно, определив области, где компоненты не должны находиться.



Область вокруг точки (0, 0) — это зона исключения, которую называют зоной боли. Компоненты этой области нежелательны, потому что их нельзя расширить, они неабстрактные, и очень трудно изменить из-за большой устойчивости.

Область вокруг точки (1, 1) — это зона бесполезности. Компоненты этой области также нежелательны, потому что они максимально абстрактны и не имеют входящих зависимостей. Часто это оставшиеся абстрактные классы, которые так и не реализовались.

Наиболее изменчивые компоненты находятся далеко от зон исключения. Точки, максимально удалённые от обеих зон, лежат на прямой, соединяющей точки (1, 0) и (0, 1) — главной последовательности.

Компонент, располагающийся на главной последовательности, не «слишком абстрактный» для своей устойчивости и не «слишком неустойчив» для своей абстрактности.

Самые желательные позиции для компонента — конечные точки главной последовательности. Хорошие архитекторы стремятся разместить большинство компонентов в этих точках.

Чтобы компонент располагался на или вблизи главной последовательности, следует определить метрику, выражающую удалённость компонента от идеала.

D — расстояние. $D = |A + I - 1|$

Метрика D вычисляется для всех компонентов. Любой компонент со значением метрики D, далёким от нуля, требует пересмотра и реструктуризации.

Такая метрика не считается точной и однозначной оценкой состояния системы, но позволяет на качественном уровне формализовать процесс работы с компонентами и ввести некоторую «измеримость» этих процессов.

При составлении диаграммы зависимостей, расчёт устойчивости компонентов также участвует в улучшении структуры связи между компонентами.

Закон Деметры

Принцип наименьшего знания

Закон Деметры Law of Demeter (LoD) или Principle of Least Knowledge: «Разговаривайте только с друзьями, не разговаривайте с незнакомцами».

Друзья модуля, например, класса:

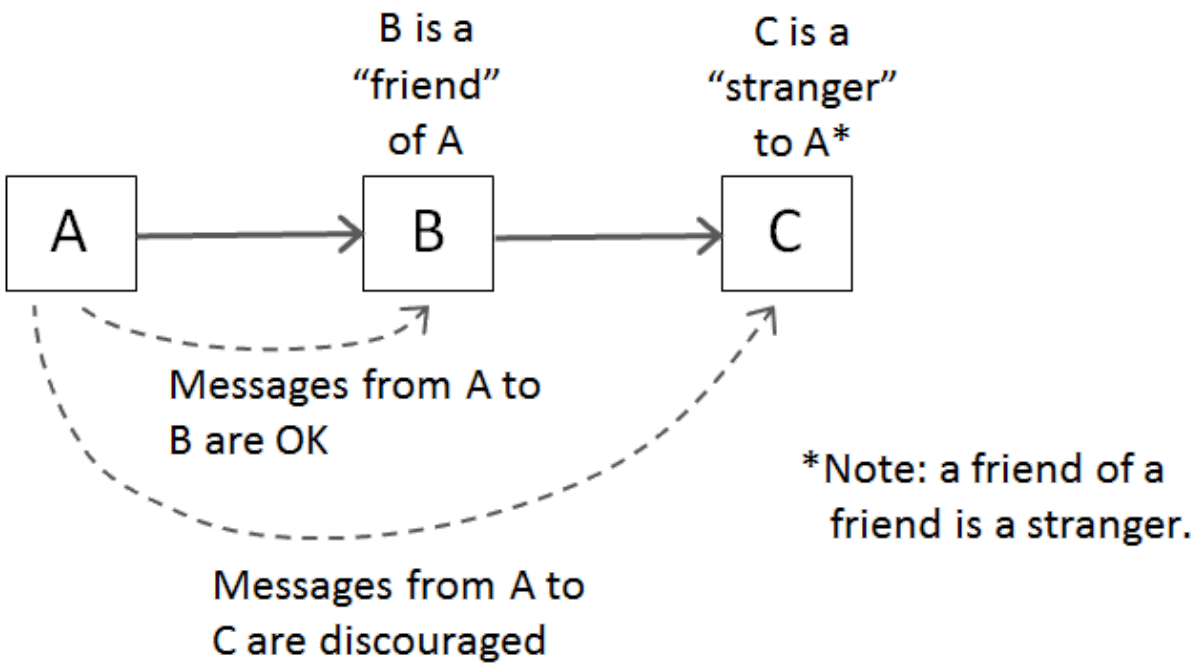
- Составные части данного модуля
- Составные части модуля, которому принадлежит данный
- Объекты, переданные данному модулю
- Объекты, которые модуль сделал сам

Альтернативная формулировка закона Деметры предложил Peter van Rooijen:

- 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.

Каждый программный модуль должен:

- Обладать ограниченным знанием о других модулях, то есть знать о модулях, которые имеют к нему «непосредственное» отношение
- Взаимодействовать только с известными ему модулями-«друзьями», не взаимодействовать с незнакомцами.
- Обращаться только к непосредственным «друзьям».



Способ версионирования программных продуктов

Ещё одной особенно сложной задачей в распределённой архитектуре является создание, сопровождение и версионирование компонентов.

Семантическое управление версиями (также называемое **SemVer**) — это система управления версиями, которая **развивается** в течение последних нескольких лет.

SemVer — сокращение от Semantic Versioning. Это общепринятый способ версионирования программных продуктов.

Это всегда было проблемой для разработчиков программного обеспечения, менеджеров по выпуску и потребителей. Универсальный способ создания версий проектов разработки программного обеспечения — лучший способ отслеживать, что происходит с программным обеспечением, поскольку новые плагины, дополнения, библиотеки и расширения создаются практически каждый день.

Версия выглядит как X.Y.Z, где X — мажорная версия, Y — минорная, Z — версия патча.

Мажорная увеличивается, если происходят несовместимые изменения пользовательского API — такое обновление сломает программу.

Минорная увеличивается, если происходят функционально заметные, но совместимые изменения — такое обновление может поменять поведение;

Патч увеличивается при незаметных для пользователя изменениях — такое обновление может улучшить качество работы.

Примеры:

- Мажорная — из библиотеки была удалена часть функций, некоторые стали принимать другие параметры: 4.3.12 → 5.0.0.
- Минорная — в приложении добавлен новый режим работы, старый остался без изменений: 1.2.1 → 1.3.0.
- Патч — исправлен баг, улучшена производительность, произведён рефакторинг: 1.1.1 → 1.1.2.

Особый смысл имеет мажорная версия 0. До выпуска 1.0.0 продукт считается на этапе начала разработки. Его поведение может быть нестабильным, а совместимость может ломаться.

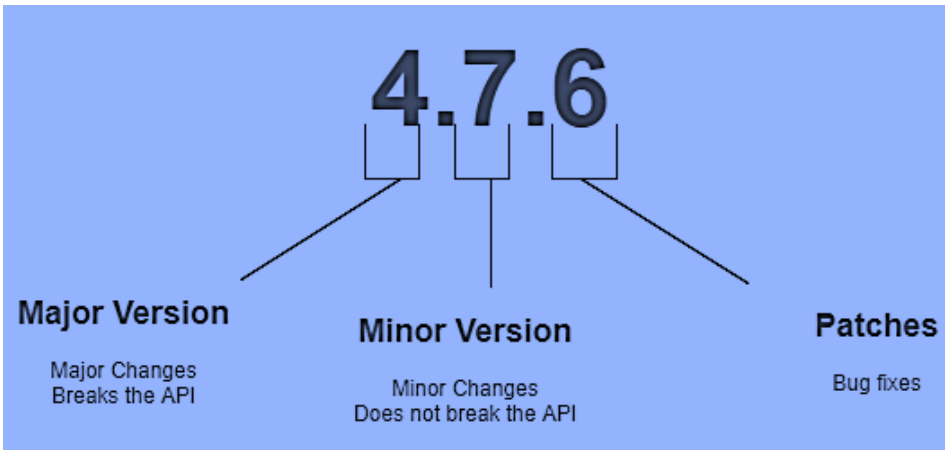
Вдобавок к трём версиям иногда используется лейбл (метка) сборки — строка после символа-.

Например,1.2.3-rc– типичное обозначение версии-кандидата в релиз. Лейблы могут делиться точкой на компоненты:1.2.3-rc.3– более *поздняя* версия, чем 1.2.3-rc.2.

Семантическое управление версиями представляет собой трёхкомпонентный номер в формате **XYZ** , где:

- X — основная версия
- Y — минорная версия
- Z — патч

Итак, SemVer имеет вид **Major.Minor.Patch**.



Цель SemVer в том, чтобы придать здравый смысл управлению быстро меняющимися целями выпуска программного обеспечения. Как обсуждалось выше, для идентификации версии программного обеспечения требуются 3 числа, т. Е. Major, Minor и Patch. Например, если мы берём версию 5.12.2, то она имеет основную версию 5, вспомогательную версию 12 и версию патча 2. Ниже приведены сценарии, когда вам следует увеличить значения X, Y и Z.

- Увеличьте значение **X** при нарушении существующего API.
- Увеличьте значение **Y** при реализации новых функций в обратно-совместимом способе.
- Увеличьте значение **Z** при исправлении ошибок.

Допустимые идентификаторы находятся в наборе **[A-Za-z0-9]** и не могут быть пустыми. Метаданные перед выпуском идентифицируются добавлением дефиса в конец последовательности SemVer. Таким образом, предварительный релиз для версии 1.0.0 может быть 1.0.0-alpha.1. Затем, если потребуется другая сборка, она станет 1.0.0-alpha.2 и так далее. Надо обратить внимание, что имена не могут содержать начальные нули, но в именах для идентификаторов перед выпуском разрешены дефисы.

Особенности:

- Первая версия начинается с 0.1.0, а не с 0.0.1, так как никаких исправлений ошибок не было, скорее мы начнём с набора функций в качестве первого черновика проекта.
- До версии 1.0.0 это только фаза разработки, на которой вы сосредоточены на выполнении работы.
- SemVer не охватывает библиотеки с тегом **0. *. *** . Первая стабильная версия v1.0.0.

Преимущества SemVer:

- Можно отслеживать каждый переход на этапе разработки программного обеспечения.
- Управление версиями может объяснить разработчикам, какие изменения произошли, и возможные обновления, которые должны произойти в программном обеспечении.
- Это помогает держать артефакты приложения (сервисы, компоненты) понятными в структуре большой системы.
- Это помогает другим людям понимать ваш продукт, который они используют в качестве зависимости.

Глоссарий

Класс — это шаблон, на основе которого создаётся конкретный программный объект. Он описывает свойства и методы, определяющие поведение объектов этого класса.

Компонент — множество классов и языковых конструкций, объединённых по общему признаку и организованных в соответствии с конкретными правилами и ограничениями.

Независимость развёртывания компонентов — когда реализация компонента меняется, достаточно повторно развернуть только этот компонент.

Независимость разработки компонентов — система состоит из модулей, которые разрабатываются и развёртываются независимо, разными командами.

Сценарий использования, вариант использования, прецедент использования (англ. use case) — в разработке программного обеспечения и системном проектировании это описание поведения системы, когда она взаимодействует с кем-то (или чем-то) из внешней среды.

Компоненты развёртывания — блоки, из которых строится компонентное программное обеспечение.

Компонент развёртывания — структурная единица программной системы, обладающая чётко определённым интерфейсом. Такой компонент независимо, удалён или добавлен в состав некоторой системы, может включаться в состав систем других поставщиков.

Компонентная модель (component model) — набор правил определения интерфейсов компонентов и их реализаций, а также правил, по которым компоненты работают в системе (регламентирующие жизненный цикл компонента) и взаимодействуют друг с другом.

Набор базовых служб (basic services), обеспечивающих, например, обнаружение компонентов в среде (сетевой), передачу данных между компонентами (через сетевое взаимодействие).

Дополнительные материалы

1. Статья «Чистая архитектура на PHP. Как её измерять и контролировать?».
2. Статья «Зависимости между компонентами».

1. Введение в проблемно-ориентированное проектирование.
2. Вернон Вон, «Реализация методов предметно-ориентированного проектирования».

Используемые источники

3. Роберт Мартин, «Чистая архитектура. Искусство разработки программного обеспечения».
4. Use-Case-Driven Development
5. Эрик Эванс ,«Предметно-ориентированное проектирование»
6. Doug Rosenberg, Matt Stephens “Use Case Driven Object Modeling with UML: Theory and Practice”
7. Крэг Ларман, «Применение UML и шаблонов проектирования».
8. Диаграмма компонентов — руководство на LucidChart
9. Контрактное программирование: Вики-статья и статья на Хабре.
10. UML — справка по Component diagram