

# Правильная декомпозиция ПО

JSP & Servlets  
14 уровень, 6 лекция

ОТКРЫТА

## Иерархическая декомпозиция

Никогда не стоит сразу начинать писать классы вашего приложения. Сначала его нужно спроектировать. Проектирование должно закончиться продуманной архитектурой. И чтобы получить эту архитектуру, тебе нужно последовательно выполнить декомпозицию системы.

Декомпозицию надо проводить иерархически — сначала систему разбивают на крупные функциональные модули/подсистемы, описывающие ее работу в самом общем виде. Затем полученные модули анализируются более детально и делятся на подмодули либо на объекты.

Перед тем как выделять объекты, разделите систему на основные смысловые блоки хотя бы мысленно. В небольших приложениях обычно сделать это очень просто: пару уровней иерархии бывает вполне достаточно, так как система вначале делится на подсистемы/пакеты, а пакеты делятся на классы.



Эта мысль не так банальна, как кажется. Например, в чем заключается суть такого распространенного “архитектурного шаблона” как Модель-Вид-Контроллер (MVC)?

Всего-навсего в **отделении представления от бизнес-логики**. Сначала любое пользовательское приложение делится на два модуля — один отвечает за реализацию самой бизнес логики (Модель), а второй — за взаимодействие с пользователем (Пользовательский Интерфейс или Представление).

Затем выясняется, что модули должны как-то взаимодействовать, для этого в них добавляют Контроллер, задача которого управлять взаимодействием модулей. Также в мобильной (классический) версии MVC в него добавляют паттерн Наблюдатель, чтобы View мог получать события из модели и изменять отображаемые данные в реальном времени.

Типичными модулями верхнего уровня, полученными в результате первого деления системы на наиболее крупные составные части, как раз и являются:

- Бизнес-логика;
- Пользовательский интерфейс;
- База данных;
- Система обмена сообщениями;
- Контейнер объектов.

При первом разбиении обычно все приложение разбивается на 2-7 (максимум 10 частей). Если разбить на большее количество частей, то потом возникнет желание их сгруппировать, и мы получим опять-таки 2-7 модулей верхнего уровня.

## Функциональная декомпозиция

Деление на модули/подсистемы лучше всего производить **исходя из тех задач, которые решает система**. Основная задача разбивается на составляющие ее подзадачи, которые могут решаться/выполняться автономно, независимо друг от друга.

Каждый модуль должен отвечать за решение какой-то подзадачи и выполнять соответствующую ей **функцию**. Помимо функционального назначения модуль характеризуется также набором данных, необходимых ему для выполнения его функции, то есть:

**Модуль = Функция + Данные**, необходимые для ее выполнения.

Если декомпозиция на модули выполнена правильно, то взаимодействие с другими модулями (отвечающими за другие функции) будет минимальным. Оно может быть, но его отсутствие не должно быть критически важным для вашего модуля.

Модуль — это не произвольный кусок кода, а отдельная функционально осмысленная и законченная программная единица (подпрограмма), которая обеспечивает решение некоторой задачи и в идеале может работать самостоятельно или в другом окружении и быть переиспользуемой. Модуль должен быть некой "целостностью, способной к относительной самостоятельности в поведении и развитии". (Кристофер Александер)

Таким образом, грамотная декомпозиция основывается, прежде всего, на **анализе функций** системы и необходимых для выполнения этих функций данных. Функции в этом случае — это не функции класса и модули, ведь это не объекты. Если у тебя в модуле всего пара классов, значит ты перестарался.

## Сильная и слабая связность

Очень важно не перестараться с разбиением на модули. Если дать новичку монолитное Spring-приложение и попросить разбить его на модули, то он вынесет каждый Spring Bean в отдельный модуль и будет считать, что его работа закончена. Но это не так.

Главным критерием качества декомпозиции является то, насколько модули **сфокусированы на решении своих задач и независимы**.

Обычно это формулируют следующим образом: "Модули, полученные в результате декомпозиции, должны быть максимально сопряжены внутри (high internal cohesion) и минимально связаны друг с другом (low external coupling)."

**High Cohesion, высокая сопряженность** или "сплоченность" внутри модуля, говорит о том, модуль сфокусирован на решении одной узкой проблемы, а не занимается выполнением разнородных функций или несвязанных между собой обязанностей.

Сопряженность — cohesion, характеризует степень, в которой задачи, выполняемые модулем, связаны друг с другом.

Следствием High Cohesion является **принцип единственной ответственности** (Single Responsibility Principle — первый из пяти принципов SOLID), согласно которому любой объект/модуль должен иметь лишь одну обязанность и не должно быть больше одной причины для его изменения.

**Low Coupling**, слабая связанность, означает что модули, на которые разбивается система, должны быть, по возможности, **независимы** или слабо связаны друг с другом. Они должны иметь возможность взаимодействовать, но при этом как можно меньше знать друг о друге.

Каждый модуль не должен знать, как устроен другой модуль, на каком языке он написан и как он работает. Часто для организации взаимодействия таких модулей используют некий контейнер, в который эти модули и загружаются.

При правильном проектировании, при изменении одного модуля, не придется править другие или эти изменения будут минимальными. Чем слабее связанность, тем легче писать/понимать/расширять/чинить программу.

Считается, что хорошо спроектированные модули должны обладать следующими свойствами:

- **Функциональная целостность и завершенность** — каждый модуль реализует одну функцию, но реализует хорошо и полностью, модуль самостоятельно выполняет полный набор операций для реализации своей функции.
- **Один вход и один выход** — на входе программный модуль получает определенный набор исходных данных, выполняет содержательную обработку и возвращает один набор результатных данных, то есть реализуется стандартный принцип IPO — вход—>процесс—>выход.
- **Логическая независимость** — результат работы программного модуля зависит только от исходных данных, но не зависит от работы других модулей.

- **Слабые информационные связи с другими модулями** — обмен информацией между модулями должен быть по возможности минимизирован.

Новичку очень сложно понять, как снизить связность модулей еще сильнее. Частично это знание приходит с опытом, частично — после чтения умных книг. Но лучше всего помогает анализ архитектур уже существующих приложений.

## Композиция вместо наследования

Грамотная декомпозиция — это своего рода искусство и сложная задача для большинства программистов. Простота тут обманлива, а ошибки обходятся дорого.



Бывает, что выделенные модули оказываются сильно сцеплены друг с другом и их не удастся разрабатывать независимо. Или не ясно за какую функцию каждый из них отвечает. Если ты столкнулся с похожей проблемой, то скорее всего разбиение на модули произвели неправильно.

Всегда должно быть понятно, **какую роль выполняет каждый модуль**. Самый надежный критерий того, что декомпозиция делается правильно, это если модули получаются самостоятельными и ценными подпрограммами, которые могут быть использованы в отрыве от всего остального приложения (а значит, могут быть переиспользуемы).

Делая декомпозицию системы, желательно проверять ее качество, задавая себе вопросы: "Какую задачу выполняет каждый модуль?", "Насколько модули легко тестировать?", "Возможно ли использовать модули самостоятельно или в другом окружении?", "Как сильно изменения в одном модуле отразятся на остальных?".

Нужно стараться, чтобы модули были предельно **автономны**. Как было сказано раньше, это **является ключевым параметром правильной декомпозиции**. Поэтому проводить ее нужно таким образом, чтобы модули изначально слабо зависели друг от друга. Если это у тебя получилось, то ты молодец.

Если нет, то тут тоже не все потеряно. Имеется ряд специальных техник и шаблонов, позволяющих дополнительно минимизировать и ослабить связи между подсистемами. Например, в случае MVC для этой цели использовался шаблон "Наблюдатель", но возможны и другие решения.

Можно сказать, что техники для уменьшения связанности, как раз и составляют основной "инструментарий архитектора". Только необходимо понимать, что речь идет обо всех подсистемах и **ослаблять связанность нужно на всех уровнях**

**иерархии**, то есть не только между классами, но также и между модулями на каждом иерархическом уровне.

[← Предыдущая лекция](#)

[Следующая лекция →](#)

 **+20** 

Комментарии

популярные

новые

старые

JavaCoder

Введите текст комментария



У ЭТОЙ СТРАНИЦЫ ЕЩЕ НЕТ НИ ОДНОГО КОММЕНТАРИЯ

ОБУЧЕНИЕ

- Курсы программирования
- Курс Java
- Помощь по задачам
- Подписки
- Задачи-игры

СООБЩЕСТВО

- Пользователи
- Статьи
- Форум
- Чат
- Истории успеха
- Активности

КОМПАНИЯ

- О нас
- Контакты
- Отзывы
- FAQ
- Поддержка

JavaRush — это интерактивный онлайн-курс по изучению Java-программирования с нуля. Он содержит 1200 практических задач с проверкой решения в один клик, необходимый минимум теории по основам Java и мотивирующие фишки, которые помогут пройти курс до конца: игры, опросы, интересные проекты и статьи об эффективном обучении и карьере Java-девелопера.

ПОДПИСЫВАЙТЕСЬ

ЯЗЫК ИНТЕРФЕЙСА

Русский

▼

СКАЧИВАЙТЕ НАШИ ПРИЛОЖЕНИЯ

ДОСТУПНО В  
Google Play

Загрузите в  
App Store

