Архитектура и требования

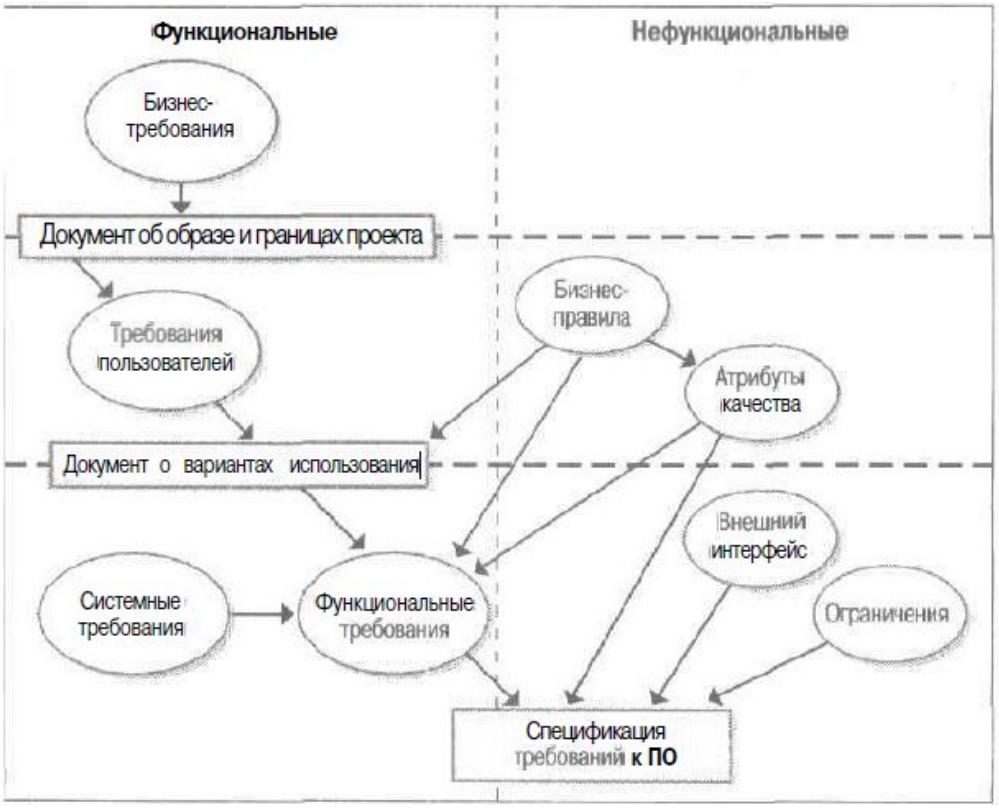
Как мы уже говорили ранее, результатом проектирования архитектуры является понимание правильной организации системы, выделение основных структурных компонент системы и формирование отношений между ними. Архитектура очень сильно связана с требованиями и ограничениями, предъявляемыми к системе, и про них тоже стоит сказать несколько слов.

Требования к ПО состоят из нескольких уровней: бизнес-требования, требования пользователей, функциональные требования и нефункциональные требования.

Функциональные требования собственно фиксируют, *что* система должна делать, нефункциональные -- то, *как* система должна это делать.

Бизнес-требования отражают высокоуровневые цели организации и заказчиков системы и проясняют, зачем вообще эта система создаётся с точки зрения заказчиков. На архитектуру эти требования могут не всегда оказывать прямое влияние, но на сам процесс разработки — легко и зачастую довольно сильное. Например, система в таком-то функционале должна работать через месяц, иначе на крупной пресс-конференции, которая уже назначена, будет нечего показать и вашу компанию ждет позор и забвение.

Требования пользователей фиксируют цели и задачи, которые пользователям позволит решить система. Из этих требований можно понять, какие будут типы пользователей у системы, как они будут с ней взаимодействовать и что от неё ожидать.



Требования на качество — как быстро система должна работать, насколько много в ней должно быть ошибок, насколько она должна быть надёжна. Например, процент времени, который система готова обслуживать клиентов, или время, за которое работоспособность системы будет восстановлена, если на её главный сервер упадёт самолёт.

Ограничения делятся на технические (если вам сказали писать под айфон, то язык реализации сам собой получится Objective C или Swift) и на бизнес-правила (разного рода корпоративные политики, промышленные стандарты и прочие нормативные документы. По факту они существуют снаружи границ любой системы, но всё равно их нужно учитывать при проектировании).

Все эти ограничения и определяют архитектуру системы, причём наибольший эффект на архитектуру имеют отнюдь не функциональные требования. Написать работающую программу несложно, сложно написать работающую программу в срок, причём так, чтобы она была не очень глючной, достаточно быстрой и потом была сопровождаема не только вами. Удовлетворить функциональным требованиям при выполнении всех ограничений можно кучей различных способов, так что остаётся некое место для манёвра в плане выбора атрибутов качества, которые зачастую противоречат друг другу, например, переиспользуемость как правило вредит скорости работы.

Всё сказанное выше в целом верно для любой парадигмы программирования. Наиболее мейнстримная сейчас парадигма — это ООП, в нём по традиции делается основной упор на таких атрибутах качества, как переиспользуемость и расширяемость программ. Например, во встроенных системах приоритеты другие, скажем, скорость работы будет важнее — расширяемость и сопровождаемость не очень важны для программ, жёстко вшитых в кристаллы.

Особенности этапа проектирования

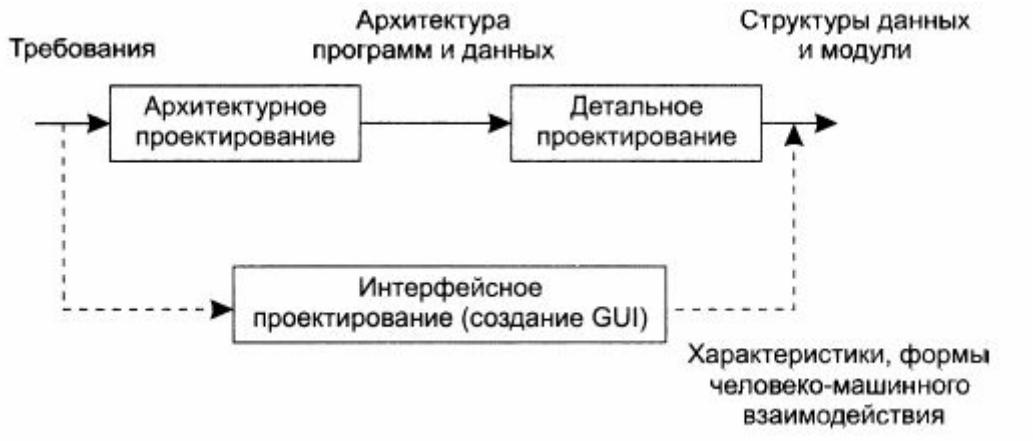
Проектирование — итерационный процесс, в рамках которого требования к системе транслируются в инженерные представления о системе. Вначале эти представления дают только концептуальную информацию (на высоком уровне абстракции), по следующие уточнения приводят к формам, которые близки к текстам на языках программирования.

Обычно в проектировании выделяют две ступени: архитектурное проектирование и детальное проектирование. Архитектурное проектирование формирует абстракции высокого уровня, детальное проектирование уточняет эти абстракции, добавляет подробности алгоритмического уровня. Кроме того, во многих случаях выделяют интерфейсное проектирование, цель которого — сформировать графический интерфейс пользователя.

В ходе архитектурного проектирования создается структурная организация системы, которая будет отвечать всем функциональным и нефункциональным требованиям. Успех этого творческого процесса зависит от типа разрабатываемой системы, подготовки и опыта системного архитектора, а также от конкретных требований к системе.

Каждая программная система уникальна, но системы в одной и той же прикладной области часто имеют сходные архитектуры, отражающие

фундаментальные положения этой области. Например, семейство продуктов — это приложения, которые строятся вокруг основной архитектуры с вариантами, удовлетворяющими специфическим требованиям клиента. Проектируя системную архитектуру, нужно выяснить общие черты создаваемой системы, а также более широкой категории приложений и решить, что можно позаимствовать из этой прикладной архитектуры.

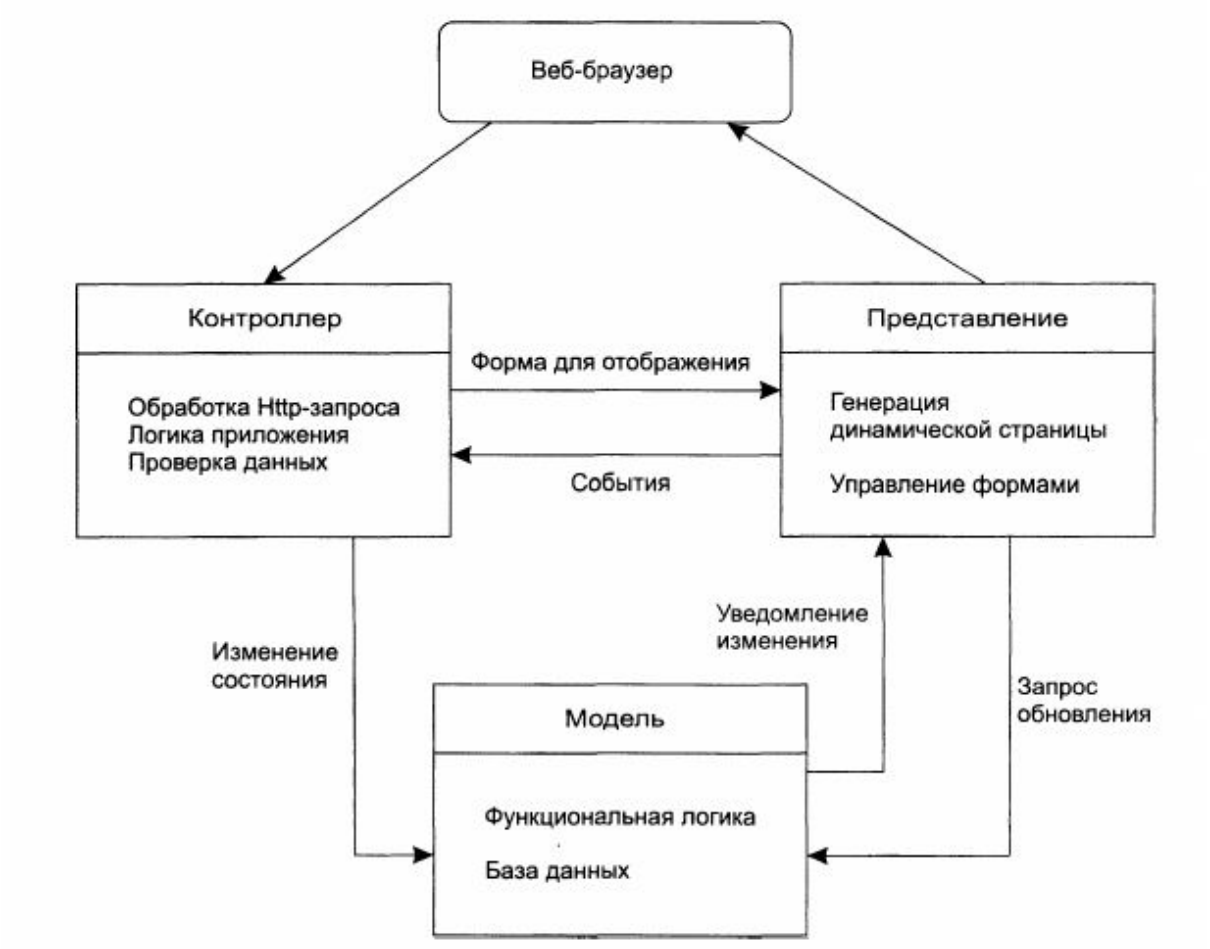


Архитектура программной системы может быть основана на определенном архитектурном паттерне. Архитектурный паттерн — это описание типовой организации системы. Архитектурные паттерны фиксируют сущность архитектуры, которая использовалась в различных программных системах. Идея паттернов как способа многократного использования знаний о программных системах в настоящее время находит широкое применение.

Архитектурный паттерн можно рассматривать как обобщенное описание хорошей практики, опробованной и проверенной в различных системах и средах. Таким образом, архитектурный паттерн должен описать системную организацию, которая была успешна в предыдущих системах.

Самый типичный пример -- шаблон Model-View-Controller, который весьма успешно используется в разработке ПО уже более 30 лет.

С его помощью архитектуру абстрактной веб-системы можно было бы описать как-то так:



Про архитектурные паттерны мы будем подробно говорить позже.

Декомпозиция

После того, как определили, из каких подсистем будет состоять наша система, имеет смысл начать детальное проектирование разбиения каждой из них на модули. Результатом декомпозиции решения является формирование структуры подсистемы — набора модулей и отношений их взаимодействия. В основе декомпозиции лежит принцип разделения понятий (separation of concerns), который в 1972 году сформулровал Э. Дейкстра: *Любую сложную проблему проще понять, разделив ее на части, каждую из которых можно решать и оптимизировать независимо.*

Рассмотрим несколько принципов проектирования, применяемых при декомпозиции.

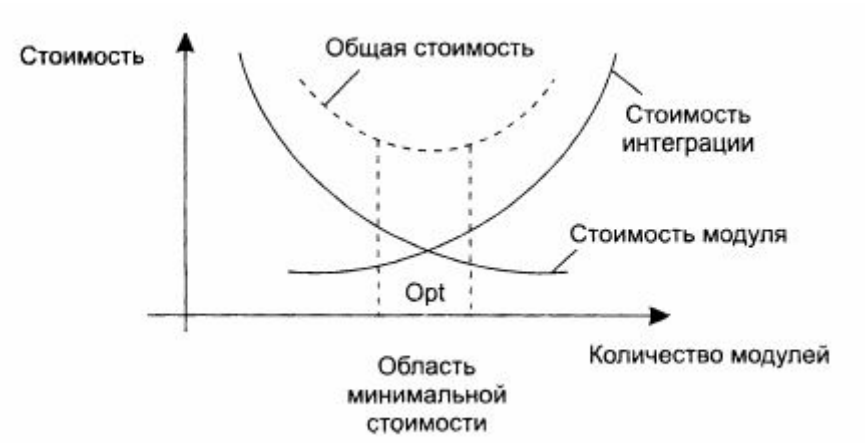
Модульность

Модульность — это наиболее общая демонстрация разделения понятий. Программная система делится на именуемые и адресуемые компоненты, часто называемые модулями, которые затем интегрируются для совместного решения проблемы.

Модуль — фрагмент программного текста, являющийся строительным блоком для физической структуры системы. Как правило, модуль состоит из интерфейсной части и части-реализации.

Модульность — свойство ПО, обеспечивающее интеллектуальную возможность создания сколь угодно сложной программы. Монолитное ПО, то есть состоящее из одного модуля, не поддается восприятию программным инженером. Огромное количество альтернативных ветвей, множество ссылок, переменных, наконец, общая сложность во многих случаях недоступны для понимания. Мы вынуждены разбивать «монолит» на модули в расчете на упрощение понимания и, как следствие, на уменьшение стоимости создания ПО.

Но тогда можно предположить, что путем последовательного дробления ПО можно добиться, что система станет тривиальной, а затраты на её разработку станут ничтожно малы. Однако это отражает лишь часть реальности — ведь здесь не учитываются затраты на дальнейшую интеграцию модулей. Как показано на рисунке ниже, с увеличением количества модулей (и уменьшением их размера) такие затраты также растут.



Таким образом, существует оптимальное количество модулей Opt, которое приводит к минимальной стоимости разработки. Но на практике у нас нет необходимых инструментов и опыта для гарантированного предсказания Opt. Есть лишь общее понимание, что оптимальный модуль должен удовлетворять следующим критериям:

- снаружи он проще, чем внутри;
- его проще использовать, чем построить;
- его можно переписать недели за две.

Информационная закрытость

Принцип информационной закрытости (Д. Парнас, 1972) утверждает, что содержание модулей должно быть скрыто друг от друга. Модуль должен определяться и проектироваться так, чтобы его содержимое (логика и данные) было недоступно тем модулям, которые не нуждаются в такой информации (клиентскому коду).

Информационная закрытость означает:

- все модули независимы, обмениваются только информацией, необходимой для работы;
- доступ к операциям и структурам данных модуля ограничен.

Достоинства информационной закрытости:

- обеспечивается возможность разработки модулей различными независимыми коллективами;
- обеспечивается легкая модификация системы (вероятность распространения ошибок очень мала, так как большинство данных и процедур скрыто от других частей системы).

Идеальный модуль играет роль черного ящика, содержимое которого невидимо клиентам. Он прост в использовании, его легко развивать и корректировать в процессе сопровождения программной системы. Но для обеспечения таких возможностей система внутренних и внешних связей модуля должна отвечать особым требованиям. Обсудим характеристики внутренних и внешних связей модуля.

Связность

Связность модуля (cohesion) — это внутренняя характеристика, определяющая меру зависимости его частей друг от друга. Чем выше связность модуля, тем лучше результат проектирования, то есть тем «черней» его ящик, тем меньше «ручек управления» на нем находится и тем проще эти «ручки». Слабая связность означает, что ни в одном месте программы нет смысла использовать все методы класса. Например, в класс, который осуществляет загрузку/выгрузку данных, не имеет смысла добавлять метод для расчета какой-либо сложной тригонометрической функции.

Выделяют 7 уровней связности, от самого сильного к самому слабому.

1. **Функциональная связность**. Части модуля вместе реализуют одну задачу. Задача может быть предельно простой, может быть сложной, то есть распадаться на многие части, но с точки зрения внешнего клиента — это всегда единое действие. Приложения, построенные из функционально связных модулей, легче всего сопровождать.

2. **Информационная (последовательная) связность**. Элементы модуля образуют конвейер для обработки данных: выходные данные одного элемента используются как входные данные в другого. Например, в модуле «Обработка массива» могут быть элементы «инициировать массив», «упорядочить массив», «распечатать массив». Сопровождать модули с информационной связностью почти так же легко, как и функционально связные модули. Правда, возможности повторного использования здесь ниже, чем в случае функциональной связности.
3. **Коммуникативная связность**. Части модуля связаны по данным (работают с одной и той же структурой данных). Например, в модуле «Анализ текста» могут быть элементы «подсчитать количество гласных», «подсчитать количество согласных», «подсчитать количество слов». Модули с коммуникативной и информационной связностью подобны в том, что содержат элементы, связанные по данным. Их удобно использовать, потому что лишь немногие элементы в этих модулях связаны с внешней средой. Главное различие между ними — информационно связный модуль работает подобно сборочной линии; его обработчики действуют в определенном порядке; в коммуникативно связном модуле порядок выполнения действий безразличен.
4. **Процедурная связность**. Части модуля связаны порядком выполняемых ими действий, реализующих некоторый сценарий поведения. Зависимости по данным между элементами нет. Например, в модуле «Обуться» могут быть элементы «надеть обувь», «зашнуровать обувь». При достижении процедурной связности мы попадаем в пограничную область между хорошей сопровождаемостью (для модулей с более высокими уровнями связности) и плохой сопровождаемостью (для модулей с более низкими уровнями связности). Например, имея задачу друг за другом посчитать среднее значение двух массивов одинаковой длины, программист может “заоптимизировать” и поместить вычисления в один цикл. Для процедурной связности этот случай типичен — независимый (на уровне постановки задачи) код стал зависимым (на уровне реализации).
5. **Временная связность**. Части модуля не связаны, но необходимы в один и тот же период работы системы. Например, в модуле «Утро» могут быть элементы «умыться», «одеться», «позавтракать». Или более технический пример — инициализация различных компонент системы при её старте. Недостаток: сильная взаимная связь с другими модулями, отсюда и сильная чувствительность к внесению изменений. Модуль со связностью по времени испытывает те же трудности, что и процедурно связный модуль. Программист соблазняется возможностью совместного использования кода (действиями, которые связаны только по времени), модуль становится трудно использовать повторно.
6. **Логическая связность**. Части модуля объединены по принципу функционального подобия. Например, модуль состоит из разных подпрограмм обработки ошибок, и при использовании такого модуля клиент выбирает только одну из подпрограмм.

7. **Связность по совпадению**. В модуле отсутствуют явно выраженные внутренние связи. Например, разработчик не знал куда пристроить этот код, да и на обед спешил, так что положил код в какой-то модуль, а потом и забыл.

Тип связности	Сопровождаемость	Роль модуля
Функциональная	Лучшая	«Черный ящик»
Информационная (последовательная)		Не совсем «черный ящик»
Коммуникативная		«Серый ящик»
Процедурная	Худшая	«Белый» или «просвечивающийся ящик»
Временная		«Белый ящик»
Логическая		
По совпадению		

Типы связности 5-7 — результат неправильного планирования проектного решения, а тип связности 4 — результат небрежного планирования проектного решения приложения.

В итоге сильная связность позволяет понизить сложность модулей (в них меньше операций, и они проще), повысить сопровождаемость системы (изменения в требованиях приводят к изменениям в меньшем количестве модулей) и повысить переиспользуемость модулей (разработчикам проще найти требуемую функциональность и нужно меньше модулей, чтобы её использовать).

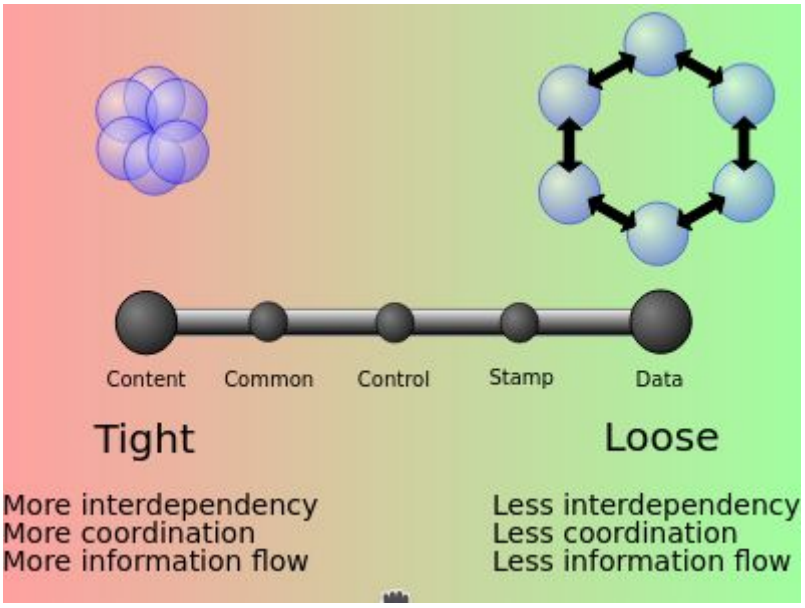
Сопряжение

Сопряжение (coupling) — это взаимная зависимость реализации модулей между собой, то есть количество изменений, которые придётся внести в один модуль при изменении другого. Слабое сопряжение означает, что изменения, вносимые в один модуль повлекут за собой небольшие изменения в другие модули. Например, если в классе есть публичная переменная, широко используемая в программе, то изменение типа этой переменной повлечёт за собой изменение большой части кода программы. Уменьшить это сопряжение можно, например, за счет реализации метода-геттера и сеттера. В большинстве случаев это позволит изменить только методы доступа и при необходимости добавить новые.

Сопряжение — внешняя характеристика модуля, которую желательно уменьшать. Выделяют 6 типов сопряжения, от сильного к слабому.

1. **Сопряжение по содержанию**. Один модуль прямо ссылается на содержание другого модуля (не через его точку входа). Например, коды их команд перемежаются друг с другом.
2. **Сопряжение по общей области**. Модули разделяют одну и ту же глобальную область данных.
3. **Сопряжение по внешним ссылкам**. Модули A и B ссылаются на одну и ту же глобальную структуру данных.

4. **Сопряжение по управлению**. Модуль A явно управляет функционированием модуля B (с помощью флагов, посылок сообщений и т.п.), посылая ему управляющие данные.
5. **Сопряжение по структуре данных**. Модули A и B ссылаются на одну и ту же глобальную структуру данных, но используют разные её части.
6. **Сопряжение по данным**. Модуль A вызывает модуль B. Все входные и выходные параметры вызываемого модуля — простые элементы данных.



Одно из главных правил объектно-ориентированного проектирования — "low coupling, high cohesion". Данное правило означает, что каждый класс должен быть сфокусирован на решении одной конкретной задачи и иметь ровно столько связей с другими классами, сколько нужно для решения этой задачи.

Сложность

В простейшем случае сложность программной системы определяется как сумма мер сложности ее модулей. Сложность модуля может вычисляться различными способами. Том МакКейб (1976) при оценке сложности программных систем предложил исходить из топологии внутренних связей. Для этой цели он разработал метрику цикломатической сложности:

$V(G) = E - N + 2P$,

где E — количество дуг, N — количество вершин, а P — количество компонент связности в графе потока управления программы.

Нисходящая и восходящая разработка

При проектировании, реализации и тестировании компонентов структурной иерархии, полученной при декомпозиции, применяют два подхода: восходящий и нисходящий.

При использовании восходящего подхода сначала проектируют и реализуют компоненты нижнего уровня, затем предыдущего и т. д. По мере завершения тестирования и отладки компонентов осуществляют их сборку, причем компоненты нижнего уровня при таком подходе часто помещают в библиотеки компонентов.

Подход имеет следующие недостатки:

- увеличение вероятности несогласованности компонентов вследствие неполноты спецификаций;
- наличие издержек на проектирование и реализацию тестирующих программ, которые нельзя преобразовать в компоненты;
- позднее проектирование интерфейса, а соответственно невозможность продемонстрировать его заказчику для уточнения спецификаций и т. д.

Исторически восходящий подход появился раньше, что связано с особенностью мышления программистов, которые в процессе обучения привыкают при написании небольших программ сначала детализировать компоненты нижних уровней (подпрограммы, классы). Это позволяет им лучше осознавать процессы верхних уровней. При промышленном изготовлении программного обеспечения восходящий подход в настоящее время практически не используют.

Нисходящий подход предполагает, что проектирование и последующая реализация компонентов выполняется «сверху-вниз», т. е. вначале проектируют компоненты верхних уровней иерархии, затем следующих и так далее до самых нижних уровней. Как только вы убедитесь, что некоторая часть задачи может быть реализована в виде отдельного модуля, постарайтесь больше не думать об этом, т. е. не уделяйте слишком много внимания тому, как именно он будет реализован.

В той же последовательности выполняют и реализацию компонентов. При этом в процессе программирования компоненты нижних, еще не реализованных уровней заменяют специально разработанными отладочными модулями-«заглушками», что позволяет тестировать и отлаживать уже реализованную часть. Когда компонент завершён, используемые им «заглушки» по одной заменяются реальным кодом (разумеется, к каждой из «заглушек» может применяться тот же самый подход с декомпозицией на более мелкие задачи). Такая последовательность действий гарантирует, что на каждом этапе разработки программист одновременно имеет дело с обозримым и понятным ему множеством участков кода (которые он может удержать в голове и не сойти с ума), и может быть уверен, что общая структура всех более высоких уровней программы верна, так что про них можно пока не думать вовсе.

Нисходящий подход обычно используют и при объектно-ориентированном программировании. В соответствии с рекомендациями подхода вначале проектируют и реализуют пользовательский интерфейс программного обеспечения, затем разрабатывают классы некоторых базовых объектов предметной области, а уже потом, используя эти объекты, проектируют и реализуют остальные компоненты. Нисходящий подход обеспечивает:

- максимально полное определение спецификаций проектируемого компонента и согласованность компонентов между собой;
- раннее определение интерфейса пользователя, демонстрация которого заказчику позволяет уточнить требования к создаваемому программному обеспечению;
- возможность нисходящего тестирования и комплексной отладки.

В итоге при декомпозиции программа разбивается на модули. Модуль (независимо от используемого языка) — это логически обособленный кусок функциональности программы, обладающий интерфейсом и реализацией. Интерфейс — это та функциональность, которую модуль предоставляет клиентам, реализация — это то, как модуль реализует эту функциональность. Клиенты модуля видят только интерфейс, и чем меньше они знают о реализации, тем лучше для них — это так называемая концепция сокрытия деталей реализации. Это удобно, поскольку позволяет менять реализацию модуля, не внося никаких изменений в код, который его использует. Пример модуля — модуль сортировки или работы со списками. Например, интерфейс модуля сортировки — объявление функции qsort(), реализация — реализация функции qsort() и функций, которые нужны, чтобы qsort() работала. Определение разумного интерфейса модуля — такого, чтобы с одной стороны, предоставить максимум возможностей клиентам, и с другой стороны, не раскрывать как можно больше деталей реализации — сложная задача проектирования.

Некоторые соображения по проектированию модулей:

- Модуль должен быть максимально минимизирован.
- Модуль не должен давать и предполагать побочных эффектов от других модулей.
- Модуль не зависит от реализации других модулей.
- Модуль реализует независимую (и по возможности единственную) функциональность.
- Модуль может быть отдельно запрограммирован и протестирован.
- Модуль спроектирован с учетом принципа сокрытия данных.
- Реализация модуля может быть изменена при сохранении интерфейсов
- Процесс разработки идет посредством последовательной детализации (на каждом этапе существует вариант программы, который можно тестировать).

Литература

1. К. Вигерс. Разработка требований к программному обеспечению
2. С. А. Орлов, Б. Я. Цилькер. Технологии разработки программного обеспечения