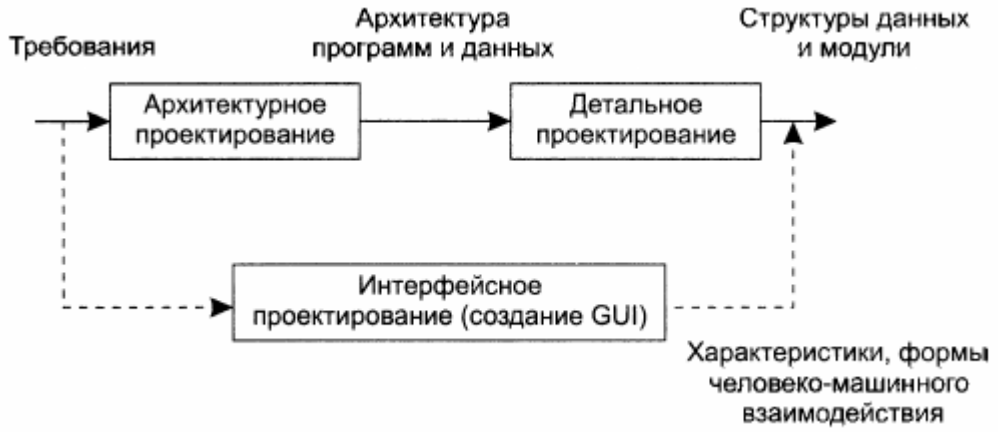
Особенности этапа проектирования

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 (product line) — 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.

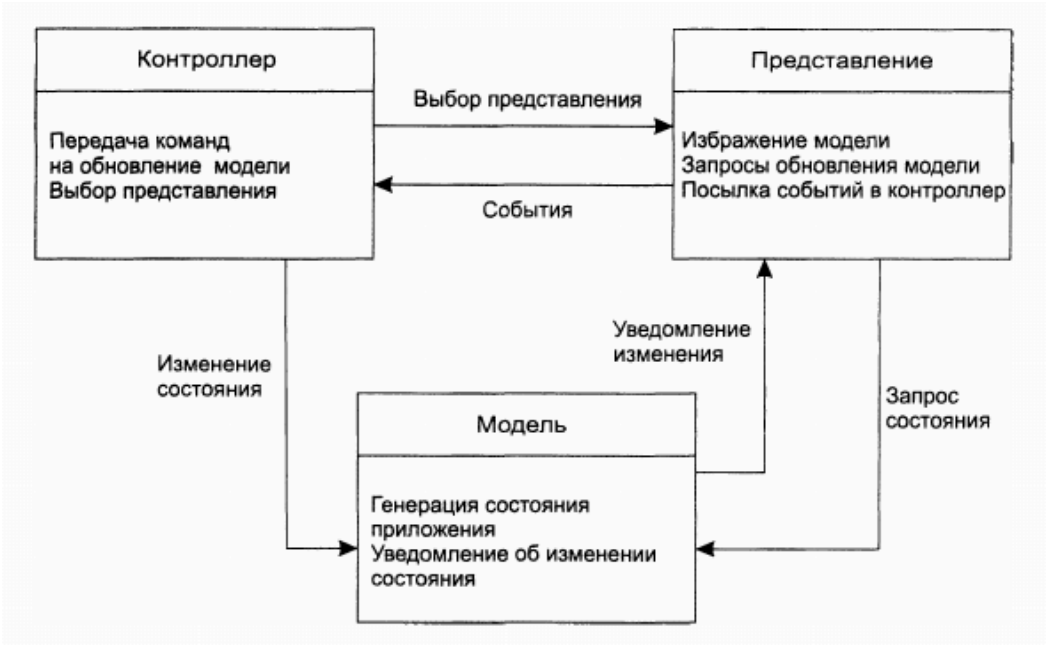


Архитектура программной системы может быть основана на определенном архитектурном стиле или шаблоне/паттерне. Архитектурный шаблон — это описание типовой организации системы. Архитектурные шаблоны фиксируют сущность архитектуры, которая использовалась в различных программных системах. Идея

шаблонов как способа многократного использования знаний о программных системах в настоящее время находит широкое применение.

Архитектурный шаблон можно рассматривать как обобщенное описание хорошей практики, опробованной и проверенной в различных системах и средах. Таким образом, архитектурный шаблон 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 стили и шаблоны 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 сформулировал 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.

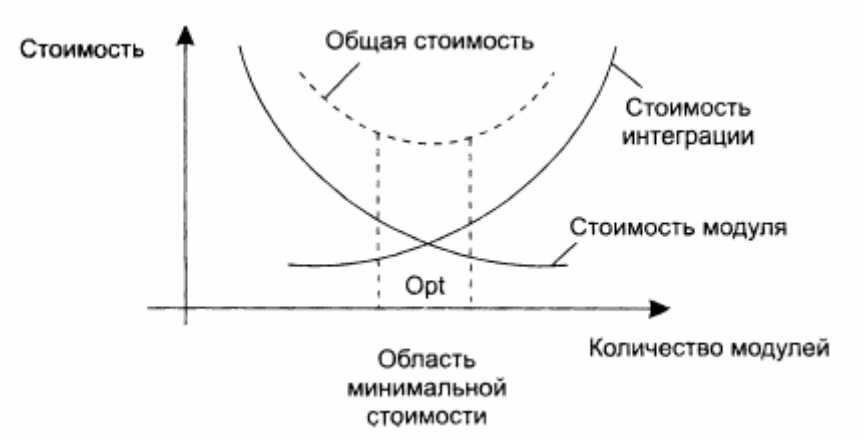
Модульность

Модульность — это наиболее общая демонстрация концепции 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 чёрный 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 «Подготовиться к выходу» могут быть элементы «обуть ботинки» и «надеть одежду». 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

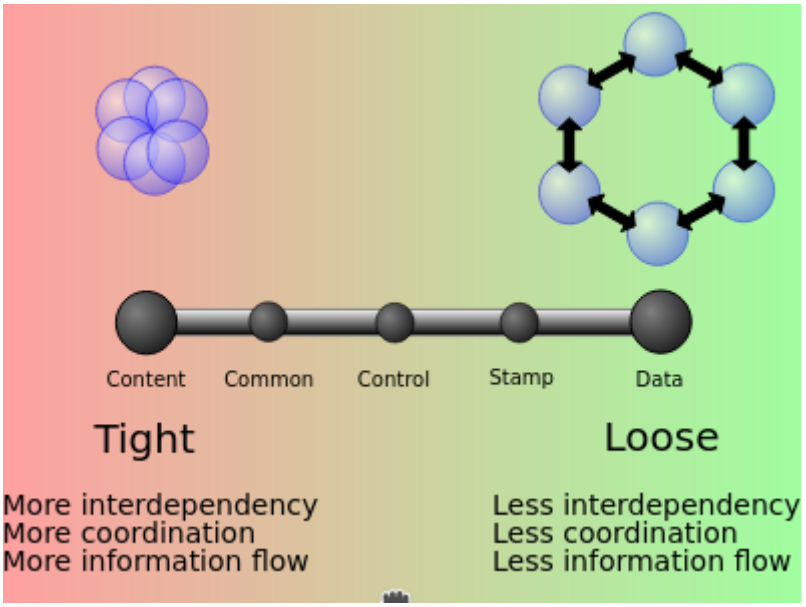
требованиях приводят к изменениям в меньшем количестве модулей) и повысить переиспользуемость модулей (разработчикам проще найти требуемую функциональность и нужно меньше модулей, чтобы её использовать).

Сопряжение

Сопряжение (coupling), или внешняя связность — 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 несколько типов сопряжения, ниже они приведены 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 (например, используют общую глобальную переменную).
3. **Внешнее сопряжение**. Модули используют какой-то единый внешний формат данных, протокол или устройство.
4. **Сопряжение по управлению**. Модуль A явно управляет функционированием модуля B (с помощью флагов, посылая ему управляющие данные и т.п.).
5. **Сопряжение по структуре данных**. Модули A и B ссылаются на одну и ту же глобальную структуру данных, но используют разные её части (например, работают с разными атрибутами в рамках одной таблицы СУБД).
6. **Сопряжение по данным**. Модуль A вызывает модуль B. Все входные и выходные параметры вызываемого модуля — простые элементы данных.
7. **Сопряжение по сообщениям**. Обмен сообщениями или событийно-ориентированная схема позволяет добиться наиболее низкого сопряжения модулей.



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 появляется 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. Некоторые относят к такому виду проектирования Agile-подходы к разработке, но это не совсем корректно.

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).

ООП и объекты

При объектно-ориентированном подходе любая программа представляется в виде набора взаимодействующих между собой объектов. Традиционно считается, что объект — это набор данных и процедур их обработки, вместе представляющий некую независимую сущность. Это в явном виде постулируют большинство уважаемых источников.

- `Objects may contain data, in the form of fields, often known as attributes; and code, in the form of procedures, often known as methods -- Wikipedia.`
- `An object stores its state in fields and exposes its behavior through methods -- Oracle.`
- `Each object looks quite a bit like a little computer – it has a state, and it has operations that you can ask it to perform -- Thinking in Java.`
- `An object is some memory that holds a value of some type -- The C++ Programming Language.`

Технически это верно, поскольку когда мы объявляем классы, мы определяем поля и методы, да и при создании объектов для них выделяется определённое количество байтов в оперативной памяти. Но при таком определении очень велик соблазн начать относиться к объектам как к “структурам данных с методами”, что влечёт яростное непонимание объектно-ориентированной парадигмы среди начинающих программистов. Объекты -- это “живые организмы”, представляющие собой сущности реального мира, с собственным жизненным циклом и собственным поведением. В книге Object Thinking даётся такое определение объекта:

- `An object is the equivalent of the quanta from which the universe is constructed.`

Оно, разумеется, гораздо более абстрактно и не содержит практических указаний для применения на практике, однако гораздо более концептуально верно отражает объектно-ориентированную парадигму. Ни у кого не возникает сомнений, что чистое функциональное программирование очень сильно отличается от процедурного (стоит начать писать, например, что-нибудь нетривиальное на Haskell, чтобы в этом убедиться). Чистое объектно-ориентированное программирование отличается от процедурного ничуть не меньше, но это не все осознают, особенно если начинают знакомиться с ООП на примере C++, который имеет обратную совместимость с C и по сути является мультипарадигменным, позволяя замечательно писать код и в процедурном стиле тоже. В этом смысле Java или C# заставляют писать более “чистый” объектно-ориентированный код, но и в них тоже есть свои особенности.

Итак, объекты отражают в коде сущности предметной области, моделируя их атрибуты и поведение. Объекты общаются друг с другом посредством посылки-приёма сообщений. В популярных объектно-ориентированных языках в простых программах это часто реализуется посредством того же вызова функций, что опять же может запутать неопытных программистов. В ООП важное отличие концепции сообщений от обычных вызовов функций -- объект сам решает, как реагировать на сообщение, и тот код, который вызовется как реакция на какое-то сообщение, вообще говоря, заранее (во время компиляции) неизвестен. Кроме того, у объекта есть интерфейс — набор сообщений, которые он может обрабатывать. То есть, один объект для другого можно представлять себе как чёрный ящик, у которого есть какие-то входы, за которые его можно “дёрнуть” и попросить что-то сделать. В ООП такую точку зрения на объект называют инкапсуляцией, подразумевая под ней одновременно и сокрытие деталей реализации (наружу видно только интерфейс объекта, или набор интерфейсов), и сбор всех данных и методов их обработки в одном месте — в самом объекте.

Рассмотрим некоторые характеристики объектов.

Состояние объекта характеризуется перечнем (обычно статическим) всех свойств данного объекта и текущими (обычно динамическими) значениями каждого из этих свойств. Например, одним из свойств торгового автомата является способность принимать монеты. Это статическое (фиксированное) свойство, в том смысле, что оно -- существенная характеристика торгового автомата. С другой стороны, этому свойству соответствует динамическое значение, характеризующее количество принятых монет. Сумма увеличивается по мере опускания монет в автомат и уменьшается, когда продавец забирает деньги из автомата.

К числу свойств объекта относятся присущие ему или приобретаемые им характеристики, черты, качества или способности, делающие данный объект самим собой. Например, для лифта характерным является то, что он сконструирован для поездок вверх и вниз, а не горизонтально. Перечень свойств объекта является, как правило, статическим, поскольку эти свойства составляют неизменяемую основу объекта. Но при этом в ряде случаев состав свойств объекта может изменяться. Примером может служить робот с возможностью самообучения. Робот первоначально может рассматривать некоторое препятствие как статическое, а затем обнаруживает, что это дверь, которую можно открыть. В такой ситуации по мере получения новых знаний изменяется создаваемая роботом концептуальная модель мира. В соответствии с этой концепцией, например, в некоторых языках программирования типа JavaScript’a объектам могут добавляться новые свойства прямо во время работы

программы.

Все свойства имеют некоторые значения. Эти значения могут быть простыми количественными характеристиками, а могут ссылаться на другой объект. Состояние лифта может описываться числом 3, означающим номер этажа, на котором лифт в данный момент находится. Состояние торгового автомата описывается в терминах других объектов, например, имеющихся в наличии напитков. Конкретные напитки -- это самостоятельные объекты, отличные от торгового автомата (их можно пить, а автомат нет, и совершать с ними иные действия).

С состоянием объекта также связано понятие **инварианта класса** -- утверждение, которое должно быть истинно применительно к любому объекту данного класса в любой момент времени (за исключением переходных процессов в методах объекта). По сути эти утверждения задают целостность объектов и являются их важнейшей характеристикой.

Объекты не существуют изолированно, а подвергаются воздействию или сами воздействуют на другие объекты. **Поведение объекта** -- это то, как объект действует и реагирует, это его наблюдаемая и проверяемая извне деятельность. Операцией называется определенное воздействие одного объекта на другой с целью вызвать соответствующую реакцию. Например, клиент может активизировать операции append и pop для того, чтобы управлять объектом-очередью (добавить или изъять элемент).

Важным свойством объектов также является их **идентичность** -- качество, которое позволяет отличать один объект от других. Каждый объект, созданный в процессе работы объектно-ориентированной системы, уникален и идентифицируется независимо от значений его полей. Возможны две ситуации:

- два различных объекта могут иметь абсолютно одинаковые поля;
- поля данного объекта могут изменяться в процессе выполнения системы, но это не влияет на идентификацию объекта.

В зависимости от выбранного подхода выражение "a обозначает тот же объект, что и b" будет иметь разный смысл. Например, в первом случае два разных объекта Point с одинаковыми значениями полей x и y имеет смысл считать одинаковыми, второй вариант может описывать один и тот же объект, данные которого менялись с течением времени.

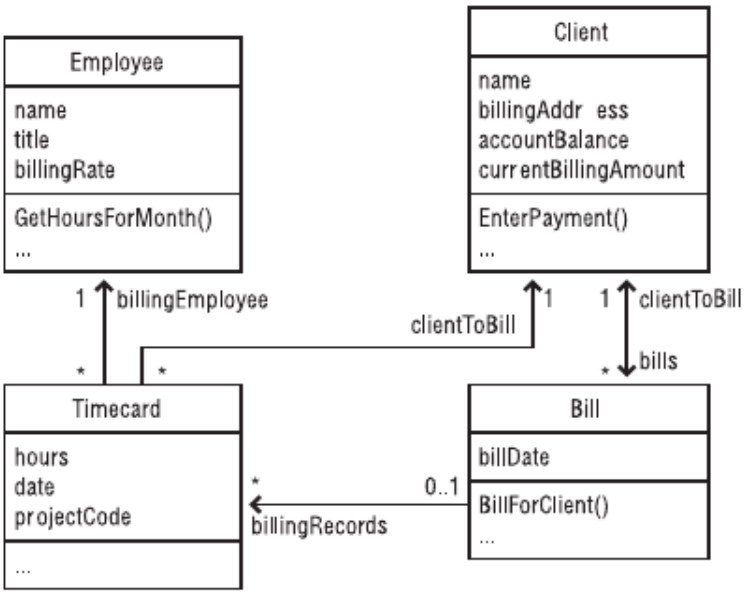
Объектно-ориентированное проектирование

Так как процесс проектирования не является детерминированным, главным аспектом проектирования качественного ПО становится умелое применение набора эффективных эвристических принципов. Выше мы рассматривали некоторые обобщённые принципы процесса декомпозиции, тут же поговорим более подробно про объектно-ориентированное проектирование. Рассмотрим ряд таких принципов — подходов, способных привести нас к более удачным решениям.

Определите объекты реального мира

Первый, ”общепринятый” подход к проектированию основан на определении объектов реального мира и идентификации соответствующих им программных объектов.

Определение объектов и их атрибутов. Как мы говорили выше, объекты отражают сущности реального мира. Например, система расчета зарплаты может быть основана на таких сущностях, как сотрудники, клиенты, карты учета времени и счета. Каждый объект имеет характеристики, релевантные для создаваемой программы. Например, в системе расчета повременной оплаты объект «сотрудник» обладал бы такими атрибутами, как имя/фамилия, должность и уровень оплаты. С объектом «счет» были бы связаны такие атрибуты, как сумма, имя/фамилия клиента, дата и т. д. Объектами системы GUI были бы разнообразные окна, кнопки, шрифты и инструменты рисования.



Такого рода установление однозначного соответствия между объектами программы и объектами реального мира -- не самый лучший способ определения объектов, но для самого начала и он вполне подойдёт, особенно когда больше нет никаких идей, с чего начать.

Определение действий, которые могут быть выполнены над каждым объектом. Объекты могут поддерживать самые разные операции. В этой системе расчета оплаты объект «сотрудник» мог бы поддерживать изменение должности или уровня оплаты, объект «клиент» — изменение реквизитов счета и т. д.

Определение связей между объектами. Двумя универсальными действиями, которые объекты могут выполнять друг над другом, являются композиция и наследование. Какие объекты могут включать другие (и какие?) объекты? Какие объекты могут быть унаследованными от других (и каких?) объектов? Например, объект «карта учета времени» может включать в себя объект «сотрудник» и объект «клиент», а объект «счет» может включать карты учета времени. Кроме того, счет может сообщать, что клиент оплатил услуги, а клиент -- оплачивать указанную в счете сумму.

Противостояние наследования и композиции -- очень важный момент, мы про него ещё поговорим подробнее.

Определение интерфейса каждого объекта. Для каждого объекта надо определить формальный синтаксический интерфейс на уровне языка программирования. Данные и методы, которые объект предоставляет в распоряжение остальным объектам, называются «открытым интерфейсом». Части объекта,

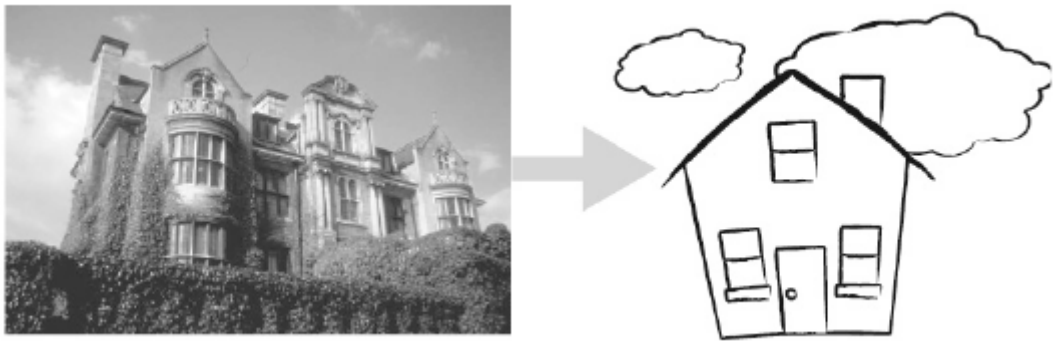
доступные производным от него объектам, называются «защищенным интерфейсом» объекта. Проектируя программу, нужно аккуратно продумывать интерфейсы обоих типов.

Завершая проектирование высокоуровневой объектно-ориентированной организации системы, используют два вида итерации: высокоуровневую, направленную на улучшение организации классов, и итерацию на уровне каждого из определенных классов, направленную на детализацию проекта каждого класса.

Определите согласованные абстракции

Абстракция позволяет задействовать концепцию, игнорируя ее некоторые детали и работая с разными деталями на разных уровнях. Имея дело с составным объектом, вы имеете дело с абстракцией. Если вы рассматриваете объект как «дом», а не как комбинацию стекла, древесины и гвоздей, вы прибегаете к абстракции. Если вы рассматриваете множество домов как «город», вы прибегаете к другой абстракции.

Абстракция выделяет существенные характеристики некоторого объекта, отличающие его от всех других видов объектов и, таким образом, четко определяет его концептуальные границы с точки зрения наблюдателя.



Базовые классы представляют собой абстракции, позволяющие концентрироваться на общих атрибутах производных классов и игнорировать детали конкретных классов при работе с базовым классом. Удачный интерфейс класса — это абстракция, позволяющая сосредоточиться на интерфейсе, не беспокоясь о внутренних механизмах работы класса. Интерфейс грамотно спроектированного метода обеспечивает такую же выгоду на более низком уровне детальности, а интерфейс грамотно спроектированного пакета или подсистемы — на более высоком.

С точки зрения сложности, главное достоинство абстракции в том, что она позволяет игнорировать нерелевантные детали. Большинство объектов реального мира уже является абстракциями некоторого рода. Как было сказано выше, дом — это абстракция окон, дверей, обшивки, электропроводки, водопроводных труб, изоляционных материалов и конкретного способа их организации. Дверь же — это абстракция особого вида организации прямоугольного фрагмента некоторого материала, петель и ручки. А дверную ручку можно считать абстракцией конкретного способа упорядочения медных, никелевых или стальных деталей.

Мы используем абстракции на каждом шагу. Если бы, открывая или закрывая дверь, вы должны были иметь дело с отдельными волокнами древесины, молекулами

лака и стали, вы вряд ли смогли бы войти в дом или выйти из него. Абстракция — один из главных способов борьбы со сложностью реального мира.

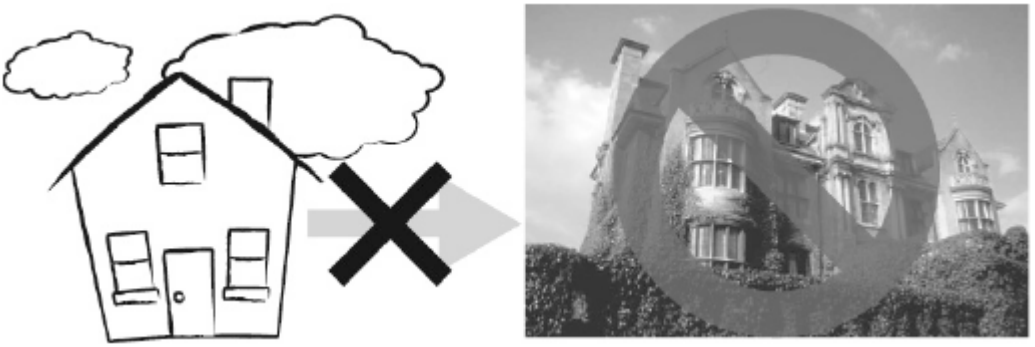
Разработчики ПО иногда создают системы на уровне волокон древесины и молекул лака и стали, из-за чего такие системы становятся слишком сложными и плохо поддаются осмыслению. Если программисты не создают более общие абстракции, разработка системы может завершиться неудачей. Благоразумные программисты продумывают и создают абстракции на уровне интерфейсов методов, интерфейсов классов и интерфейсов пакетов (иначе говоря, на уровне дверной ручки, уровне двери и на уровне дома), что способствует более быстрому и безопасному программированию.

Инкапсулируйте детали реализации

Когда абстракция нас покидает, на помощь приходит инкапсуляция. Абстракция говорит: «Вы можете рассмотреть объект с общей точки зрения». Инкапсуляция добавляет: «Более того, вы не можете рассмотреть объект с иной точки зрения».

Инкапсуляция -- это процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение. Инкапсуляция служит для того, чтобы изолировать контрактные обязательства абстракции от их реализации.

Продолжим нашу аналогию: инкапсуляция позволяет вам смотреть на дом, но не дает подойти достаточно близко, чтобы узнать, из чего сделана дверь. Инкапсуляция позволяет вам знать о существовании двери, о том, открыта она или заперта, но при этом вы не можете узнать, из чего она сделана (из дерева, стекловолокна, стали или другого материала), и уж никак не сможете рассмотреть отдельные волокна древесины. Инкапсуляция помогает управлять сложностью, блокируя доступ к ней.



Абстракция и инкапсуляция дополняют друг друга: абстрагирование направлено на наблюдаемое поведение объекта, а инкапсуляция занимается внутренним устройством.

Выбирайте между наследованием и композицией

Наследование упрощает программирование, позволяя создать универсальные методы для выполнения всего, что основано на общих свойствах сущностей, и затем написать специфические методы для выполнения специфических операций над конкретными типами этих сущностей. Некоторые операции, такие как Open() или

Close(), будут универсальными для всех дверей: внутренних, входных, стеклянных, стальных — каких угодно.

Наследование — одно из самых мощных средств объектно-ориентированного программирования. При правильном применении оно может принести большую пользу, однако в обратном случае и ущерб будет немалым. Например, мы не всегда можем хотеть применять наследование (скажем, когда нам не нужен интерфейс класса, а только его реализация), язык не предоставляет нам для этого технических возможностей (к примеру, если запрещено множественное наследование) или в принципе применение наследования неразумно (например, если есть большое количество разных объектов и способов реализаций каждого из них). Связь наследования нельзя разорвать во время работы программы, к тому же не любая связь между двумя классами объектов описывается отношением “является” (“is a”). К тому же, многие считают, что наследование разрушает инкапсуляцию, так как подклассу становятся доступны скрытые детали реализации предка, да и очень легко неаккуратно поменять предка так, что потомки поломаются.

Зависимость от реализации может повлечь за собой проблемы при попытке повторного использования подкласса. Если хотя бы один аспект унаследованной реализации непригоден для новой предметной области, то приходится переписывать родительский класс или заменять его чем-то более подходящим. Такая зависимость ограничивает гибкость и возможности повторного использования. С проблемой, конечно, можно справиться, если наследовать только абстрактные классы, поскольку в них обычно совсем нет реализации или она минимальна, но такое не всегда возможно.

Другой механизм расширения функциональности классов -- агрегация/композиция, которая позволяет использовать один объект как часть состояния другого. Композиция задаёт отношение “является частью” (“has a”) и позволяет гибко управлять зависимостью во время выполнения программы, не создавая новых типов сущностей. Композицию можно применить, если объекты соблюдают интерфейсы друг друга. Для этого, в свою очередь, требуется тщательно проектировать интерфейсы, так чтобы один объект можно было использовать вместе с широким спектром других. Но и выигрыш велик. Поскольку доступ к объектам осуществляется только через их интерфейсы, мы не нарушаем инкапсуляцию. Во время выполнения программы любой объект можно заменить другим, лишь бы он имел тот же тип. Зависимость от реализации резко снижается.

Композиция объектов влияет на дизайн системы и еще в одном аспекте. Отдавая предпочтение композиции объектов, а не наследованию классов, вы инкапсулируете каждый класс и даете ему возможность выполнять только свою задачу. Классы и их иерархии остаются небольшими, и вероятность их разрастания до неуправляемых размеров невелика. С другой стороны, дизайн, основанный на композиции, будет содержать больше объектов (хотя число классов, возможно, уменьшится), и поведение системы начнет зависеть от их взаимодействия, тогда как при другом подходе оно было бы определено в одном классе.

Разница между композицией и агрегацией заключается в том, что в случае композиции целое явно контролирует время жизни своей составной части (часть не существует без целого), а в случае агрегации целое хоть и содержит свою составную часть, время их жизни не связано (например, составная часть передается через параметры конструктора или метод-сеттер).

Интересной особенностью разных отношений между классами является то, что логичность их использования может зависеть от точки зрения проектировщика, от того, с какой стороны он смотрит на задачу и какие вопросы он задает себе при ее анализе. Именно поэтому одну и ту же задачу можно решить десятком разных способов, при этом в одном случае мы получим сильно связанный дизайн с большим количеством наследования и/или композиции, а в другом случае – эта же задача будет разбита на более автономные строительные блоки, объединяемые между собой с помощью агрегации.

Скрывайте “лишнюю” информацию

Сокрытие информации — одна из самых конструктивных идей в мире разработки ПО. При сокрытии информации каждый класс (пакет, метод) характеризуется аспектами проектирования или конструирования, которые он скрывает от остальных классов. Секретом может быть источник вероятных изменений, формат файла, реализация типа данных или область, изоляция которой требуется для сведения к минимуму вреда от возможных ошибок. Класс должен скрывать эту информацию и защищать свое право на «личную жизнь». Небольшие изменения системы могут влиять на несколько методов класса, но не должны распространяться за его интерфейс.

Один из важнейших аспектов проектирования класса — принятие решения о том, какие свойства сделать доступными вне класса, а какие оставить секретными. Класс может включать 25 методов, предоставляя доступ только к пяти из них и используя остальные 20 внутренне. Класс может использовать несколько типов данных, не раскрывая сведений о них. Интерфейс класса должен сообщать как можно меньше о внутренней работе класса. В этом смысле класс во многом похож на айсберг, большая часть которого скрыта под водой.

Связанные с сокрытием информации секреты относятся к двум общим категориям:

- секреты, которые скрывают сложность, позволяя программистам забыть о ней при работе над остальными частями программы;
- секреты, которые скрывают источники изменений с целью локализации результатов возможных изменений.

В число источников сложности входят сложные типы данных, файловые структуры, булевы тесты, запутанные алгоритмы и т. д. Источники изменений будут описаны в следующем разделе.

В некоторых случаях скрыть информацию невозможно, однако большинство барьеров, препятствующих сокрытию информации, является умственными и обусловлены привыканием разработчика к другим методикам.

Избыточное распространение информации. Зачастую сокрытию информации препятствует избыточное распространение информации по системе. Так, жесткое кодирование литерала 100 во многих местах программы децентрализует ссылки на него. Лучше скрыть эту информацию в одном месте — скажем, при помощи константы MAX_EMPLOYEES, для изменения значения которой придется изменить только одну строку кода.

Ошибочное представление о данных класса как о глобальных данных. Возможна еще одна преграда на пути к эффективному сокрытию информации: вы можете рассматривать данные класса как глобальные данные, избегая их из-за соответствующих проблем. Всем известно, что дорога в ад программирования вымощена глобальными переменными, однако использовать данные класса гораздо безопаснее.

Глобальные данные имеют два главных недостатка: методы, обращающиеся к глобальным данным, не знают о том, что другие методы тоже обращаются к этим данным, или же методы знают об этом, но не знают, что именно другие методы делают с глобальными данными. Данные класса этих недостатков не имеют. Непосредственный доступ к данным класса ограничен несколькими методами этого же класса, которые знают и о том, что другие методы также работают с данными, и о том, что это за методы.

Конечно, это предполагает, что система включает грамотно спроектированные небольшие классы. Если программа использует огромные классы, включающие десятки методов, различие между данными класса и глобальными данными стирается, и данные класса приобретают многие недостатки, характерные для глобальных данных.

Кажущееся снижение производительности. Наконец, отказ от сокрытия информации может объясняться стремлением избежать снижения производительности кода. Разработчикам кажется, что опосредованный доступ к данным снизит производительность программы в период выполнения из-за дополнительных затрат на создание объектов, вызовы методов и т. д. Эти волнения преждевременны. Пока вы не оцените производительность системы и не найдете узкие места, лучшим способом подготовки к повышению производительности на уровне кода является модульное проектирование. Позже, определив в коде «горячие точки», вы оптимизируете отдельные классы и методы, не затрагивая остальную часть системы.

Определяйте области вероятных изменений

Как мы уже говорили на прошлой лекции, обеспечение легкости адаптации программы к возможным изменениям относится к самым сложным аспектам проектирования. Его цель заключается в изоляции нестабильных областей, позволяющей ограничить следствия изменений одним методом, классом или пакетом. Если при проектировании системы не принималась во внимание возможность будущих изменений, то есть вероятность, что в будущем ее придется полностью перепроектировать. Это может повлечь за собой переопределение и новую реализацию классов, модификацию клиентов и повторный цикл тестирования. Перепроектирование отражается на многих частях системы, поэтому непредвиденные изменения всегда оказываются дорогостоящими.

1. **Определите элементы, изменение которых кажется вероятным.** Если вы выработали адекватные требования, они включают список потенциальных изменений и оценки вероятности каждого из них. В этом случае определить вероятные изменения легко. Если требования не описывают потенциальные изменения, ниже вы найдете некоторый список областей, которые меняются чаще всего независимо от типа проекта.

2. **Отделите элементы, изменение которых кажется вероятным.** Создайте отдельный класс для каждого нестабильного компонента, определенного в п.1, или разработайте классы, включающие несколько нестабильных компонентов, изменение которых скорее всего будет одновременным.
3. **Изолируйте элементы, изменение которых кажется вероятным.** Спроектируйте интерфейсы между классами так, чтобы они не зависели от потенциальных изменений. Спроектируйте интерфейсы так, чтобы изменения ограничивались только внутренними частями классов. Изменение класса должно оставаться незаметным для любых других классов. Интерфейс класса должен защищать его секреты.

Обдумывая потенциальные изменения системы, проектируйте ее так, чтобы влияние изменений было обратно пропорционально их вероятности. Если вероятность изменения высока, убедитесь, что систему будет легко адаптировать к нему. С большим влиянием на несколько классов системы можно смириться лишь в случае крайне маловероятных изменений. Грамотные проектировщики также принимают во внимание цену предвосхищения изменений. Если изменение маловероятно, но его легко предугадать, рассмотрите его внимательнее, чем более вероятное изменение, которое трудно спланировать.

Ниже описано несколько областей, изменяющихся чаще всего.

Бизнес-правила. Необходимость изменения ПО часто объясняется изменениями бизнес-правил. Оно и понятно: может измениться система налогообложения, могут быть пересмотрены условия контрактов и т. д. Если вы соблюдаете принцип сокрытия информации, логика, основанная на этих правилах, не будет распространена на всю программу. Она будет скрыта в одном темном уголке системы, пока не придет время ее изменить.

Зависимости от оборудования. Примерами модулей, зависимых от оборудования, могут служить интерфейсы между программой и разными типами мониторов, принтеров, клавиатур, дисководов, звуковых плат и сетевых устройств. Изолируйте зависимости от оборудования в отдельной подсистеме или отдельном классе. Это облегчает адаптацию программы к новой аппаратной среде, а также помогает разрабатывать ПО для нестабильных версий устройств. Вы можете разработать ПО, моделирующее взаимодействие с конкретным устройством, и создать подсистему аппаратного интерфейса, использующую эту модель, пока устройство нестабильно или недоступно. Когда устройство будет готово к работе, подсистему интерфейса можно будет отключить от модели и подключить к устройству.

Ввод-вывод. На чуть более высоком в сравнении с аппаратными интерфейсами уровне проектирования частой областью изменений является ввод-вывод. Если ваше приложение создает собственные файлы данных, его усложнение вполне может потребовать изменения формата файлов. Аспекты формата ввода-вывода данных, относящиеся к пользовательскому уровню, такие как позиционирование и число полей на странице, их последовательность и т. д., изменяются не менее часто. В общем, анализ всех внешних интерфейсов на предмет возможных изменений — благоразумная идея.

Нестандартные возможности языка. Большинство версий языков поддерживает нестандартные расширения, облегчающие работу программистов. Расширения — палка о двух концах, потому что в другой среде — будь то другая аппаратная

платформа, реализация языка другим производителем или новая версия языка, выпущенная тем же производителем, — они могут оказаться недоступны. Если вы применяете нестандартные расширения языка, скройте работу с ними в отдельном классе, чтобы его можно было заменить при адаптации приложения к другой среде. Аналогично, используя библиотечные методы, доступные не во всех средах, скройте их за интерфейсом, поддерживающим все нужные среды.

Сложные аспекты проектирования и конструирования. Скрывайте сложные аспекты проектирования и конструирования, потому что их частенько приходится реализовывать заново. Отделите их и минимизируйте влияние, которое может оказать их неудачное проектирование или конструирование на остальные части системы.

Переменные статуса. Переменные статуса характеризуют состояние программы и изменяются чаще, чем большинство других видов данных. Так, разработчики, определившие переменную статуса ошибки как булеву переменную, вполне могут позднее прийти к выводу, что для этого лучше было бы использовать перечисление со значениями ErrorType_None, ErrorType_Warning и ErrorType_Fatal. Использование переменных статуса можно сделать более гибким и понятным минимум двумя способами.

- В качестве переменных статуса примените не булевы переменные, а перечисления. Диапазон поддерживаемых переменными статуса состояний часто приходится расширять, что в случае перечисления требует лишь перекомпиляции программы, а не масштабной ревизии всех фрагментов кода, выполняющих проверку переменной.
- Вместо непосредственной проверки переменной используйте методы доступа. Так вы сохраните возможность реализации более сложного механизма определения состояния. Например, если вы захотите проверять комбинацию переменной статуса ошибки и переменной текущего функционального состояния, вам будет легко реализовать это, если проверка будет скрыта в методе, и гораздо сложнее, если механизм проверки будет жестко закодирован во многих местах программы.

Структуры данных. Объявляя массив из 100 элементов, вы раскрываете информацию, которую никто знать не должен. Защищайте право на личную жизнь! Сокрытие информации не всегда требует создания целого класса. Иногда для этого достаточно именованной константы: например, MAX_EMPLOYEES позволяет скрыть число 100 или интерфейс коллекции может спрятать конкретную используемую структуру.

Поддерживайте сопряжение слабым

Сопряжение (coupling) характеризует силу связи класса или метода с другими классами или методами. Наша цель — создать классы и методы, имеющие немногочисленные, непосредственные, явные и гибкие отношения с другими классами, что еще называют «слабым сопряжением» (loose coupling)».

Ниже описаны критерии, позволяющие оценить сопряжение модулей.

Объем. Объем связи характеризует число соединений между модулями. Чем их меньше, тем лучше, поскольку модуль, имеющий более компактный интерфейс, легче связать с другими модулями. Метод, принимающий один параметр, слабее сопряжен с

вызывающими его модулями, чем метод, принимающий шесть параметров. Класс, имеющий четыре грамотно определенных открытых метода, слабее сопряжен с модулями, которые его используют, чем класс, предоставляющий 37 открытых методов.

Видимость. Тут под видимостью мы понимаем заметность связи между двумя модулями. Программирование не служба в ЦРУ — никто не похвалит вас за удачную маскировку. Оно больше похоже на рекламу: вам следует делать связи между модулями как можно более крикливыми. Передача данных посредством списка параметров формирует очевидную связь, и это удачный вариант. Передача информации другому модулю в глобальных данных является замаскированной и потому неудачной связью. Описание связи, осуществляемой через глобальные данные, в документации делает ее более явной и является чуть более удачным подходом.

Гибкость. Гибкость характеризует легкость изменения связи между модулями. Идеальная связь должна быть как можно гибче. Гибкость частично определяется другими аспектами связанности, но в то же время отличается от них. Например, если у вас есть какой-то класс, который использует для удалённого вызова какую-то определённую RPC библиотеку, то и всем остальным, кто захочет с ним взаимодействовать, придётся её использовать. Понятно, что можно реализовать протокол самостоятельно и не использовать эту библиотеку, но никто этого делать не будет. Чем проще вызывать модуль из других модулей, тем слабее он сопряжен, и это хорошо, потому что такой модуль более гибок и прост в сопровождении. Создавая структуру программы, делите ее на блоки с учетом их взаимосвязанности.

Стремитесь к максимальной связности

Связность (cohesion) характеризует то, насколько хорошо все методы класса или все фрагменты метода соответствуют главной цели, — иначе говоря, насколько сфокусирован класс. Классы, состоящие из очень похожих по функциональности блоков, обладают высокой степенью связности, и наша эвристическая цель состоит в том, чтобы целостность была как можно выше. Связность — полезный инструмент управления сложностью, потому что чем лучше код класса соответствует главной цели, тем проще запомнить все, что код выполняет.

Формализуйте контракты классов

На более детальном уровне полезную информацию можно получить, рассматривая интерфейс каждого класса как контракт с остальными частями программы. Обычно контракт имеет форму «Если вы обещаете предоставить данные x, y и z и гарантируете, что они будут иметь характеристики a, b и c, я обязуюсь выполнить операции 1, 2 и 3 с ограничениями 8, 9 и 10». Обещания клиентов классу обычно называются предусловиями (preconditions), а обязательства класса перед клиентами — постусловиями (postconditions). Контракты помогают управлять сложностью, потому что хотя бы теоретически объект может свободно игнорировать любое поведение, не описанное в контракте.

Проектируйте систему для тестирования

На некоторые интересные идеи можно натолкнуться, спросив, как будет выглядеть система, если спроектировать ее для обеспечения максимальной легкости тестирования. Отделять ли пользовательский интерфейс от остальной части программы, чтобы протестировать его независимо? Организовывать ли каждую подсистему так, чтобы минимизировать ее зависимость от других подсистем? Проектирование для тестирования часто приводит к разработке более формализованных интерфейсов классов, что обычно выгодно.

Подумайте об использовании “грубой силы”

Грубая сила — один из мощнейших эвристических инструментов. Не стоит ее недооценивать. Работоспособное решение проблемы методом грубой силы лучше, чем элегантное, но не работающее решение. Создавать элегантные решения зачастую долго и сложно. Гораздо практичнее сначала получить плохое, но работающее решение, а потом уже делать его красивым и элегантным, чем потратить кучу времени на создание красивого решения сразу и пропустить дедлайн.

Рисуйте диаграммы

Диаграммы — еще один мощный эвристический инструмент. Все знают, что лучше один раз увидеть, чем сто раз услышать. Диаграммы позволяют представить проблему на более высоком уровне абстракции, и никакие описания их не заменят. Иногда проблему следует рассматривать на детальном уровне, а иногда целесообразно иметь дело с более общими аспектами, и в этом случае нужны диаграммы разных уровней абстракции.

Принципы SOLID

Есть также чуть более формализованные принципы, позволяющие разрабатывать более структурированные и модульные объектно-ориентированные системы. Такие системы обеспечивают удобство сопровождения и безболезненность внесения изменений в их код. Пять важных принципов дизайна классов в ОО-проектировании, сформулированных Робертом Мартином, по их первым буквам получили аббревиатуру SOLID.

Single responsibility principle, Принцип единственной обязанности. Означает он, что каждый объект должен иметь одну обязанность и эта обязанность должна быть полностью инкапсулирована в класс. Все, что этот класс делает, должно быть направлено исключительно на обеспечение этой обязанности. Например, представьте себе модуль, который составляет и печатает отчёт. Такой модуль может измениться по двум причинам. Во первых, может измениться само содержимое отчёта. Во вторых, может измениться формат отчёта. Оба этих фактора изменяют модуль по разным причинам: в одном случае изменение содержательное, а во втором — косметическое. Принцип единственной обязанности говорит, что оба аспекта этой проблемы на самом деле являются двумя разными обязанностями, и в таком случае должны находиться в

разных классах или модулях. Объединение двух сущностей, изменяющихся по разным причинам и в разное время, считается плохим проектным решением. Если у объекта много ответственности, то и меняться он будет очень часто. Таким образом, если класс имеет больше одной ответственности, то это ведет к хрупкости дизайна и ошибкам в неожиданных местах при изменениях кода.

Open/closed principle, Принцип открытости/закрытости: “программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения”. Это означает, что такие сущности могут позволять менять свое поведение без изменения их исходного кода. Оригинальная идея, описанная Бертраном Мейером в своей книге 1988 году, была в том, что однажды разработанная реализация класса в дальнейшем требует только исправления ошибок, а новые или изменённые функции требуют создания нового класса. Этот новый класс может переиспользовать код исходного класса через механизм наследования. Производный подкласс может реализовывать или не реализовывать интерфейс исходного класса, который изменяться не должен.

В настоящее время под принципом OCP обычно обычно понимают требование закрытия интерфейса классов, но сохранения реализации в открытом виде. Закрытый интерфейс вашего класса позволит другим программистам приступить к его использованию до того, как вы закончите реализацию. Открытая реализация позволит вам и другим программистам исправлять ошибки в коде. Интерфейс класса при этом обычно описывают при помощи абстрактных базовых классов, их использование позволяет полностью заменить старую реализацию на новую при необходимости.

Liskov substitution principle, Принцип подстановки Барбары Лисков: “*Пусть q(x) является свойством, верным относительно объектов x некоторого типа T. Тогда q(y) также должно быть верным для объектов y типа S, где S является подтипом типа T*”. Для ООП это можно переформулировать так: “*код, который использует базовый тип, должен иметь возможность использовать подтипы базового типа, не зная об этом*”.

Таким образом, идея Лисков о «подтипе» даёт определение понятия замещения — если S является подтипом T, тогда объекты типа T в программе могут быть замещены объектами типа S без каких-либо изменений желательных свойств этой программы. Поведение наследуемых классов не должно противоречить поведению, заданному базовым классом, то есть поведение наследуемых классов должно быть ожидаемым для кода, использующего переменную базового типа.

Этот принцип является важнейшим критерием для оценки качества принимаемых решений при построении иерархий наследования. Например, предположим, что у нас есть класс Квадрат и класс Прямоугольник. Кто из них является базовым классом, а кто подклассом? Можно подумать, что прямоугольник — более общая фигура с полями высота и ширина, а квадрат — ее конкретизация, в которой ширина всегда равна высоте. Однако, прямоугольник обладает следующим свойством — при изменении высоты ширина никак не меняется (и наоборот). Если у нас будет код, который будет использовать это свойство, и мы подставим по интерфейсу прямоугольника в этот код квадрат, у нас все развалится. Так что проектирование объектно-ориентированных иерархий — это дело непростое.

Также можно сделать вывод, что в соответствии с принципом подстановки Лисков, выделение общего кода двух разных классов в базовый класс — плохая затея.

В дальнейшем это сильно сужает возможности для расширения всех этих классов.

Interface segregation principle, Принцип разделения интерфейса: “Клиенты не должны зависеть от методов, которые они не используют”. Т.е. слишком “толстые” интерфейсы необходимо разделять на более мелкие и специфические, чтобы клиенты маленьких интерфейсов знали только о методах, которые необходимы им в работе. В итоге, при изменении метода интерфейса не должны меняться клиенты, которые этот метод не используют. Предположим, если класс Модем, который умеет звонить и передавать данные. Есть программа дозвона, которая использует этот класс, чтобы звонить, и есть программа посылки данных, которая запускается после программы дозвона, и которая использует класс Модем, чтобы передавать данные. При этом первой нужны только те методы, которые реализуют дозвон, а второй эти методы совсем не интересны. Поэтому лучше явно выделить два интерфейса и заставить класс Модем явно их реализовывать. Ну а каждая программа будет обращаться с этим классом по нужному ей интерфейсу.

Dependency inversion principle, Принцип инверсии зависимостей: Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций. Пример: пусть есть два класса — Switcher и Lamp. Switcher знает про класс Lamp и умеет ее включать/выключать. Однако, включать и выключать можно не только лампу, но и много чего еще. Но в коде переключателя есть явное указание класса Lamp, что сильно мешает его масштабированию на работу с другими классами. Чтобы для каждого класса, который нужно включать и выключать, не переписывать класс Switcher заново, можно выделить интерфейс Switchable с методами включить() и выключить(), и заставить все такие классы типа лампы реализовывать этот интерфейс, а класс Switcher работать с такими классами именно по интерфейсу. Тогда он даже и знать не будет, кого же он переключает. Такой код становится гораздо более гибким и переиспользуемым.

Закон Деметры

Ещё одно правило хорошего проектирования носит название закона Деметры и в краткой форме звучит как “Не разговаривай с незнакомцами!”. В более длинной формулировке: объект A не должен иметь возможность получить непосредственный доступ к объекту C, если у объекта A есть доступ к объекту B, и у объекта B есть доступ к объекту C.

Направлен этот принцип в первую очередь против использования конструкций типа `book.pages.last.text`, когда объект получает слишком много информации о внутреннем устройстве других объектов, нарушая сокрытие данных и начиная зависеть не от интерфейса, а от реализации.

Более формально Закон Деметры для функций требует, что метод M объекта O должен вызывать методы только следующих типов объектов:

- собственно самого O;
- аргументов M;
- других объектов, созданных в рамках M;
- объектов-полей O;
- глобальных переменных, доступных O, в пределах M.

То есть хорошим кодом было бы создание внутри класса Book метода lastPageText(), который бы пробросил по цепочке этот запрос дальше. И если такого рода цепочек вызовов в коде много разных, потребуется создание большого количества малых методов-адаптеров для передачи вызовов метода к внутренним компонентам.

Стоит заметить, что это правило часто ошибочно называют “правилом одной точки”, запрещая конструкции типа `book.pages().last().text()`. К примеру, что-то такое написано на странице в Википедии. При этом если посмотреть оригинал статьи, то там явно говорится, что этот принцип направлен на повышение модульности и сохранения инкапсуляции, и что объектам не стоит обращаться к внутренней структуре других объектов. И если метод pages() создаст и вернёт объект типа Page, у которого потом будет вызван метод last() и т.д. -- в этом нет никакого противоречия описанным выше правилам.

Но при этом надо всё же понимать, что конструкции типа `book.pages().last().text()` повышает сопряжение (coupling) между классами, заставляя пользователей класса Book знать про интерфейс класса Page. Решение о том, заменять подобные цепочки вызовов специальными методами или нет, должно приниматься осознанно в каждом конкретном случае, а не просто потому, что Википедия так сказала.

Чаще всего подобные цепочки методов означают, что объект пытается сделать что-то, что не лежит в его области ответственности: например, взять у объекта данные вместо того, чтобы попросить его выполнить какое-либо действие. Небольшое перепроектирование обязанностей почти всегда помогает избавиться от этой проблемы.

Литература

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