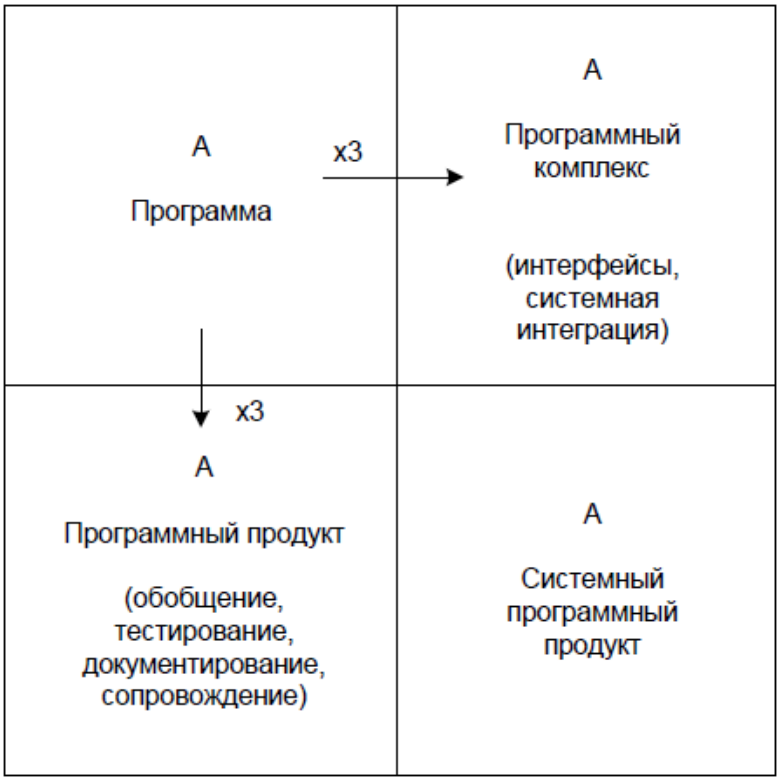
О программах и программных продуктах

Большая часть программ, за которые люди готовы платить деньги, довольно сложна. Даже небольшое по промышленным меркам приложение требует порядка двух-трёх человеко-лет на разработку и имеет размер в несколько десятков тысяч строк кода. Большие программные системы могут иметь несколько миллионов строк кода и занимать сотни и тысячи человеко-лет разработки. При этом каждый из вас понимает, что может писать гораздо больше 100 строчек кода в день, а “за выходные” вообще может сделать практически всё. Тут весь вопрос в том, что же по факту создаётся.



На картинке выше (книга Брукса хоть и устарела местами, но во многом не теряет своей актуальности) слева вверху показана программа -- законченный кусок кода, который его автор может скомпилировать, запустить на своем устройстве и который выполняет нужную задачу. Это именно то, чем неопытные программисты привыкли оценивать свою производительность. Но за такой код мало кто готов платить или использовать его в своих проектах.

При перемещении по стрелке вниз программа превращается в программный продукт. Это программа, которую любой человек может запускать, тестировать, исправлять и развивать. Она может использоваться в различных операционных средах и со многими наборами данных. Чтобы стать общеупотребительным программным продуктом, программа должна быть написана в обобщенном стиле. В частности, диапазон и вид входных данных должны быть настолько обобщенными, насколько это допускается базовым алгоритмом. Затем программу нужно тщательно протестировать, чтобы быть уверенным в ее надежности. Для этого нужно подготовить достаточное количество контрольных примеров для проверки диапазона допустимых значений

входных данных и определения его границ, обработать эти примеры и зафиксировать результаты. Наконец, развитие программы в программный продукт требует создания подробной документации, с помощью которой каждый мог бы использовать ее, делать исправления и расширять. Ну и сама программа должна быть написана так, чтобы её потенциально можно было адекватно расширять и сопровождать. По разным эмпирическим оценкам программный продукт стоит в 3-10 раз дороже, чем просто отлаженная программа с такой же функциональностью.

При пересечении вертикальной границы программа становится компонентом программного комплекса. Последний представляет собой программную систему, в состав которой входит набор взаимодействующих программ, согласованных по программным интерфейсам и форматам данных, и все вместе решающие какие-то более сложные задачи, чем каждая по отдельности. Хорошим примером тут можно считать консольные утилиты *nix, аналоги которых, кстати, вы будете разрабатывать на практических занятиях. Чтобы стать частью программного комплекса, синтаксис и семантика ввода и вывода программы должны удовлетворять точно определенным интерфейсам. Программа должна быть также спроектирована таким образом, чтобы использовать заранее оговоренный объём ресурсов — объем памяти, устройства ввода/вывода, процессорное время. Наконец, программу нужно протестировать вместе с прочими системными компонентами во всех сочетаниях, которые могут встретиться. Это тестирование может оказаться большим по объему, поскольку количество тестируемых случаев растет экспоненциально. Оно также занимает много времени, так как скрытые ошибки выявляются при неожиданных взаимодействиях отлаживаемых компонентов. Компонент программного комплекса стоит примерно так же в 3-10 раз дороже, чем автономная программа с теми же функциями. И этот порядок может увеличиться, если в системе много компонентов.

В правом нижнем углу рисунка находится системный программный продукт. От обычной программы он отличается во всех перечисленных выше отношениях. И стоит, соответственно, на 1-2 порядка дороже. Но это действительно полезный объект, который является целью большинства системных программных проектов.

Об архитектуре

Когда компонентов программы (классов, функций или чего-то ещё) становится больше десятка, организация взаимодействия по принципу “как получится” уже приводит к мешанине, в которой довольно трудно разобраться. Поэтому обычно приложение делят на подсистемы, которые взаимодействуют друг с другом по определённым интерфейсам, подсистемы, если это требуется, делят на подсистемы ещё раз и т.д. Вот такое вот описание разделения приложения на компоненты и их взаимодействия и понимают чаще всего под архитектурой системы. Архитектура ПО — это то, как его части организованы вместе, предоставляя в результате решение некоторой технической или бизнес-проблемы, которую заказчик или пользователь этого ПО хочет решить. Архитектура — это эволюционирующий свод ключевой информации об устройстве проекта.

Архитектура бывает разных уровней детализации, от общей концепции приложения до детальной структуры взаимодействия классов. Например, если мы хотим написать свой instant messenger, то он, скорее всего, будет состоять из сервера

и клиента — вот уже крупнозернистая архитектура с двумя компонентами. Дальше, если мы продолжим декомпозицию, у сервера должна быть компонента для работы с сетью, некое хранилище данных и т.д., у клиента — тоже компонента работы с сетью, GUI.

Решения, принимаемые на стадии формирования архитектуры, являются очень важными, поскольку будут лежать в основе всей системы. По сути архитектура системы — это ее фундамент, все остальные решения, которые будут приниматься в процессе разработки, будут вкладываться в существующую архитектуру. А это значит, что стоимость изменения базовых архитектурных решений в ходе проекте будет очень высока.

Еще один важный вопрос, касаемый архитектуры, состоит в том, что вашим клиентам/пользователям архитектура вашего ПО абсолютно по барабану. Вы можете сколько угодно рассказывать, какой красивый и богатый внутренний мир вашего кода, пользователям важно, чтобы ваше ПО делало то, что от него требуется, и делало хорошо и быстро (как писал Алан Купер, если бы в магазинах продавали дырки в стене, никто бы не покупал дрели). Задача проектировщиков архитектуры — делать это так, чтобы код можно было потом легко и успешно поддерживать и дорабатывать, не желая ежеминутно убить всех вокруг. И вот именно для этого нужна архитектура.

О метафоре строительства

В литературе для процесса разработки ПО очень популярна метафора строительства зданий. Эта инженерная область развивается уже тысячи лет, и за это время в ней выработались очень чётко регламентированные процессы и роли, которые в этих процессах участвуют. В упрощённом виде вначале собирают требования о здании (размер, назначение, место строительства, бюждет и т.п.), проектируется высокоуровневая архитектура здания, удовлетворяющая этим требованиям, она уточняется и оформляется в виде чертежей и другой проектной документации, по которым потом происходит строительство, происходит сдача и последующее использование объекта. Вот и в разработке ПО начинают следовать примерно такой же схеме: определяют требования, создают высокоуровневую архитектуру, разрабатывают на её основе алгоритмы, пишут код, реализующий алгоритмы, ну и в конце размещают и используют готовую систему. Можно проследить и другие аналогии:

- процесс разработки архитектуры учитывает пожелания и требования будущих пользователей;
- возможно разделение труда: те, кто разрабатывают архитектуру, не обязательно будут потом воплощать её в жизнь. Такой подход в мире ПО может неплохо работать, когда создание софта или его частей отдаётся на аутсорсинг, но в рамках одного проекта такой подход почти никогда не работает, поскольку архитектор, который не пишет код, очень часто становится оторванным от жизни и теряет связь с реальностью и коллективом;
- процесс создания имеет много ключевых точек, в которых можно отслеживать прогресс выполнения (со зданиями это очевидно, в мире ПО также создаются разного рода мокапы, прототипы, демо-версии и релиз кандидаты).

Но есть и ряд более интересных и полезных выводов, которые можно сделать из данной метафоры и которые помогают лучше понять понятие архитектуры и процесс разработки вообще.

Архитектура как самостоятельная сущность

Во-первых, нужно признать саму концепцию архитектуры как набора принципиальных решений об устройстве ПО, связанного, но существующего при этом отдельно от её реализации в коде и других артефактах разработки. Архитектура — это вполне отдельная сущность проекта, которая может проектироваться, развиваться, документироваться, тестироваться, сравниваться с другими вариантами архитектуры и проверяться на соответствие физическому её воплощению. Архитектура есть у любого приложения. Она может быть хорошая или плохая, но она есть всегда, **даже если о ней никто специально не думал**. Вопросы, на которые это нас наталкивает:

- откуда берётся архитектура?
- чем она характеризуется?
- какими свойствами обладает?
- что такое хорошая и плохая архитектура?

Архитектура и качества системы

В частности, последние три вопроса наталкивают нас на второй вывод: свойства архитектуры приводят к определённым свойствам конечного продукта. Например, если мы хотим защищаться от врагов с мечами и копьями, то проектируем и строим крепость с высокими отвесными стенами и узкими редкими окнами, а если хотим защищаться от врагов с пушками, то крепость будет иметь низкие, но толстые наклонные стены с землёй внутри. Если хотим надёжно хранить конфиденциальные данные пользователей, формируем DMZ и прячем всё критичное туда. Для создания устойчивости к отказам (обеспечения бесперебойности работы) в архитектуру вводятся избыточные (резервные) подсистемы. При проектировании архитектуры нужно держать в голове и учитывать требования к целевому продукту.

Архитектура и архитектор

Третий вывод: есть особые люди, архитекторы, которые занимаются проектированием этой самой архитектуры. Это человек или группа людей, которые принимали ключевые решения по вопросам того, как устроено приложение, и кто поддерживает (или меняет) эти решения в процессе разработки. Существующий профессиональный стандарт “Архитектор программного обеспечения” говорит нам про архитектора следующее:

> Архитектор ПО ответственен за планирование, установку и интеграцию программного обеспечения и информационных систем. Роль архитектора ПО заключается в том, чтобы понимать, интегрировать и использовать информационные решения с технологической точки зрения. Убедиться, что технические решения, процедуры и модели развития являются современными и соответствуют нормам. Следить за технологическим

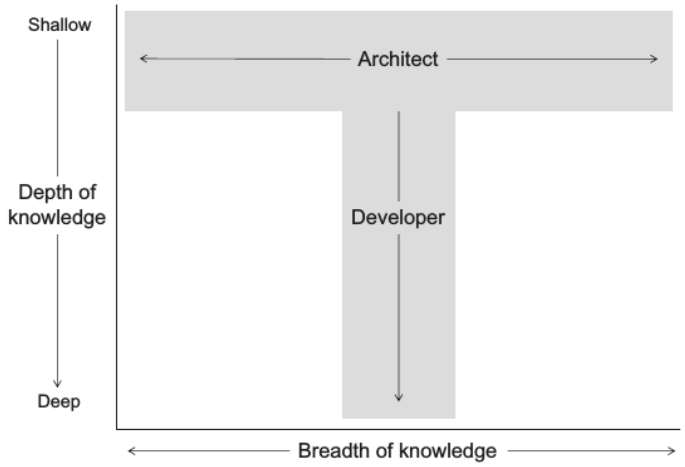
развитием и интегрировать новые решения. Реагировать как руководитель группы технических экспертов и разработчиков.

Вот некоторые из трудовых функций, которые стандарт относит к профессиональной деятельности архитектора.

- **Оценка требований к программному средству**:
  - оценка осуществимости функционирования и сопровождения программного средства,
  - оценка возможности тестирования требований,
  - оценка архитектуры с точки зрения прослеживаемости требований,
  - анализ на критичность изменения требований проекта.
- **Оценка возможности создания архитектурного проекта**:
  - оценка возможности создания архитектурного проекта программного средства,
  - определение целей архитектуры программного средства,
  - определение ключевых сценариев для архитектуры программного средства.
- **Утверждение и контроль методов и способов взаимодействия программного средства со своим окружением**:
  - согласование с заказчиком версии архитектуры программного средства,
  - техническое исследование возможных вариантов архитектуры компонентов, включающее описание вариантов и технико-экономическое обоснование выбранного варианта,
  - выбор модели обеспечения необходимого уровня производительности компонентов, включая вопросы балансировки нагрузки,
  - выбор технологий и средств разработки программного обеспечения.
- **Оценка, выбор и создание вариантов архитектуры программного средства:**
  - разбиение программного средства на компоненты,
  - определение качественных характеристик каждого компонента,
  - оценка и выбор слоев программных компонентов,
  - оценка и выбор шаблонов (стилей) проектирования для каждого слоя или компонента,
  - определение внешних-внутренних интерфейсов каждого из компонентов,
  - определение перечня возможных протоколов взаимодействия компонентов,
  - оценка и выбор модели обеспечения отказоустойчивости программных компонентов,
  - определение структуры данных каждого компонента и программного средства в целом,
  - оценка и выбор технологии доступа к данным,
  - корректировка системных требований в части необходимых инфраструктурных ресурсов,
  - определение стандартов для разработки документации,
  - …
- **Документирование архитектуры программных средств.**

- **Контроль реализации программного средства**
    - идентификация и регистрация возможных проблем из-за деталей реализации компонентов программных средств,
    - координация процесса создания и сборки программного средства из компонентов.
- **Контроль сопровождения программных средств:**
    - идентификация возможных проблем, путей их решения,
    - разработка решений для повторного использования компонентов.

Это и навыки, которым надо специально учиться, и навыки, которые приходят лишь с опытом. Одного лишь опыта программирования для успешной работы здесь недостаточно. В итоге, чтобы качественно выполнять роль архитектора, нужна очень серьёзная подготовка не только в инженерных аспектах, но и чувство эстетики, и глубокое понимание того, какие люди используют продукт и как они это делают во всех возможных аспектах (то есть архитектор должен обладать отличным пониманием предметной области, в которой создаётся продукт). А так как работа эта подразумевает активное соприкосновение с предметной областью и заказчиком, то и социальные навыки в этой работе тоже важны.



О чём стоит задуматься тут:

- кто принимал ключевые решения об архитектуре?
- отдавал ли он себе отчёт в своих действиях, когда принимал эти решения, и о последствиях этих решений?
- рассматривались ли альтернативы при принятии этих решений?
- может ли он чётко выделить и объяснить эти ключевые решения другим?
- что нужно делать для поддержания целостности архитектуры с течением времени?

Свод знаний по разработке ПО

Четвёртый вывод: проектирование архитектуры -- это область знания с вполне определёнными процессами и объёмом знаний, содержащим в себе информацию о том, как получать продукт, соответствующий определённым качествам. Если бы строительство каждого здания начиналось с изобретения всех основных принципов строительства или открытия необходимых материалов, прогресс бы никуда не

двигался. Спроектировав какое-то здание, мы по сути спроектировали целый класс типовых зданий, некое типовое решение, которое может быть применено для строительство целого микрорайона. Разумеется, здания в этом районе будут как-то отличаться друг от друга, но концепт у них будет один, для их строительства будут использоваться одинаковые инструменты, материалы, возможно даже переиспользуемые компоненты, да и процесс строительства может быть сильно ускорен по сравнению с уникальным проектом. Формируются понятия архитектурных стилей, объединяющий в себе предыдущий опыт, включая ключевые характеристики типового решения и требования/ограничения, в рамках которых это решение имеет смысл. С разработкой ПО ситуация полностью аналогичная.

Особенности мира ПО

Но есть и различия между строительством зданий и разработкой ПО, причём некоторые довольно существенны.

Во-первых, мы находимся среди зданий всю свою жизнь, люди уже тысячи лет строят здания, и даже у самого далёкого от строительства человека есть интуитивное понимание, что некоторые вещи можно строить в определённых местах или определённым способом, а некоторые нельзя. С программным обеспечением человечество имеет дело чуть больше полсотни лет, и интуиция по поводу ПО у людей ещё даже близко так не развита. А поэтому к проектированию ПО стоит относиться гораздо более внимательно, меньше полагаться на собственную интуицию, а больше на исследования и опыт других, более опытных в этой области людей (впрочем, и они тоже могут ошибаться).

Другое крайне важное отличие состоит в том, что можно очень многое сказать о здании, его внутренней архитектуре и качествах, просто смотря на него. План здания помогает архитектору и заказчику оценить пространство, возможности перемещения, виды. Становятся очевидными противоречия, можно заметить упущения. С ПО всё гораздо сложнее, поскольку по своей природе программный продукт невидим и невизуализуем. Если есть куча кирпичей, каждый из которых можно пощупать и при должном воображении представить, как будет выглядеть здание, из них построенное, то с программами это не так. Программы состоят из инструкций программного кода, из которого потом могут появляться сетевые сообщения, файлы на диске или образы на экране. Более того, может появляться такое поведение, которое вообще внешне никак не заметно (если, например, разрабатываются сервисы/демоны или вирусы). Программы неосязаемы, у них нет собирательного образа, с которым ассоциируется в голове человека понятие абстрактной компьютерной программы. Образ программного обеспечения не встраивается естественным образом в наше мышление. Поэтому у него нет готового геометрического представления подобно тому, как местность представляется картой, кремниевые микросхемы — диаграммами, компьютеры — схемами соединений. Ситуацию немного спасают графические нотации, но лишь тоже до опредённого предела. Модель по определению проще моделируемой сущности, и чтобы адекватно отобразить более-менее нетривильную программу, требуется целый набор графических языков, а модели, с их помощью созданные, обладают большим количеством перекрёстных ссылок.

Третье отличие: программные объекты постоянно подвержены изменениям, в том числе (в отличие от большинства объектов физического мира) и уже после их изготовления. Программы часто меняются, потому что 1) это становится необходимо (исправляются ошибки или меняется понимание их предназначения или окружающей инфраструктуры) и 2) это в принципе возможно за невысокую цену (тут как повезёт, на самом деле, часто заказчику бывает тяжело объяснить, что для реализации такой-то фичи придётся половину кода переписывать). Программы — это продукт мысли, воплощённые в программном коде. Поменяв несколько символов текста и осуществив процедуру сборки (и, возможно, размещения) программы, мы можем получить абсолютно другое поведение, чем у этой программы было раньше. И всё это чаще всего в полностью автоматическом режиме и практически без каких-либо затрат. Со многими физическими объектами такого в принципе осуществить невозможно, а если и возможно, это может быть очень долго и дорого. Да, здания тоже перестраиваются, но если бы можно было бы поменять внешний вид или добавить своему дому пару лишних этажей простым нажатием клавиш, вид наших улиц бы сильно преобразился.

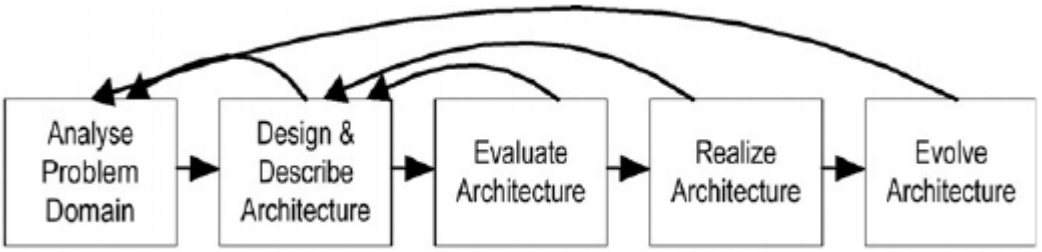
Четвёртое отличие: сложность программных объектов зависит от их размеров в гораздо большей степени, чем для любых других создаваемых человеком объектов. Разрабатывать сложное ПО невыразимо трудно. В сложных системах огромное количество компонент, которые могут сочетаться друг с другом неким нелинейным способом, генерируя ещё большее количество сочетаний, и сложность целого растет значительно быстрее, чем линейно. К тому же масштабирование программного объекта — это не просто увеличение в размере или количестве каких-то типовых элементов, это обязательно увеличение числа различных элементов. В итоге программные системы очень сложно разрабатывать, понимать, описывать, тестировать.

Сложность программ является существенным, а не второстепенным свойством. Поэтому описания программных объектов, абстрагирующиеся от их сложности, часто абстрагируются от их сущности. Математика и физические науки за три столетия достигли больших успехов, создавая упрощенные модели сложных физических явлений, получая из этих моделей свойства и проверяя их опытным путем. Это удавалось благодаря тому, что сложности, игнорировавшиеся в моделях, не были существенными свойствами явлений. И это не действует, когда сложности являются сущностью. Многие классические трудности разработки программного обеспечения проистекают из этой сложности сущности и ее нелинейного роста при увеличении размера. Сложность служит причиной трудности процесса общения между участниками команды разработчиков, что ведет к ошибкам в продукте, превышению стоимости разработки, затягиванию выполнения графиков работ. Сложность служит причиной трудности перечисления, а тем более понимания, всех возможных состояний программы, а отсюда возникает ее ненадежность. Сложность функций служит причиной трудностей при их вызове, из-за чего программами трудно пользоваться. Сложность структуры служит причиной трудностей при развитии программ и добавлении новых функций так, чтобы не возникали побочные эффекты. Сложность структуры служит источником невизуализуемых состояний, в которых нарушается система защиты. Сложность служит причиной не только технических, но и административных проблем. Из-за сложности трудно осуществлять контроль, в

результате страдает концептуальная целостность. Трудно найти и держать в голове все “движущиеся части” системы.

Об архитектуре и жизненном цикле ПО

Часто фазу проектирования явно выделяют в моделях жизненного цикла ПО: её называют “проектирование”, “анализ и дизайн” или как-то иначе. Принимая архитектуру как продукт на выходе такой фазы, мы ограничиваем её лишь некоторыми (скорее всего высокоуровневыми и самыми очевидными) проектными решениями, о которых велик соблазн забыть и которые в будущем могут противоречить другим решениям, принятым по ходу разработки и поддержки. К тому же такой подход жестко и навсегда фиксирует принятые решения, что может быть критично, если меняются требования, условия, понимание задачи или разработчики сталкиваются с непредвиденными ранее сложностями. Ну и если в проекте не классический водопад, что вряд ли у вас будет в наше время. В этом смысле к проектированию имеет смысл относиться как к одной из практик при разработке ПО по аналогии, например, с тестированием, версионированием кода или continuous integration.



Рассмотрим далее более подробно, как происходит работа с архитектурой на разных этапах жизненного цикла.

Разработка требований

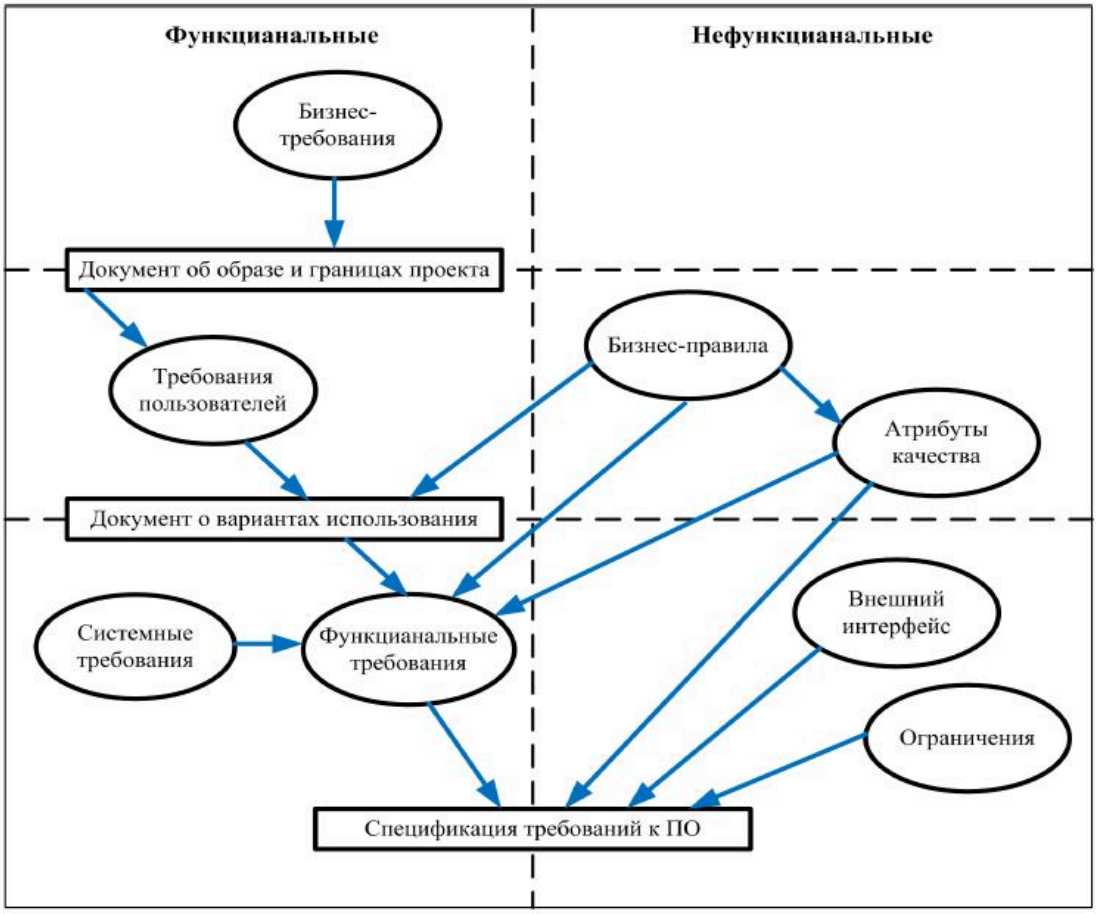
Традиционно процесс разработки ПО начинается с этапа выявления требований — набора того, что, как и в каких условиях и ограничениях разрабатываемая система должна делать. Вполне очевидно, что это информация, которая будет напрямую влиять на архитектуру системы.

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.

Говоря об атрибутах качества, стандарт ISO 9126 определяет следующие основные характеристики качества ПО.

- **Функциональность** (Functionality) определяется способностью ПО решать задачи, которые соответствуют зафиксированным и предполагаемым потребностям пользователя, при заданных условиях использования ПО. Т.е. эта характеристика отвечает за то, что ПО работает исправно и точно, функционально совместимо, соответствует стандартам отрасли и защищено от несанкционированного доступа.
- **Надежность** (Reliability) – способность ПО выполнять требуемые задачи в обозначенных условиях на протяжении заданного промежутка времени или указанное количество операций. Атрибуты данной характеристики – это завершенность и целостность всей системы, способность самостоятельно и корректно восстанавливаться после сбоев в работе, отказоустойчивость.
- **Удобство использования** (Usability) – возможность легкого понимания, изучения, использования и привлекательности ПО для пользователя.
- **Эффективность** (Efficiency) – способность ПО обеспечивать требуемый уровень производительности в соответствие с выделенными ресурсами, временем и другими обозначенными условиями.
- **Удобство сопровождения** (Maintainability) – легкость, с которой ПО может анализироваться, тестироваться, изменяться для исправления дефектов, для реализации новых требований, для облегчения дальнейшего обслуживания и адаптироваться к имеющемуся окружению.
- **Портативность** (Portability) – характеризует ПО с точки зрения легкости его переноса из одного окружения (software/hardware) в другое.

Каждая характеристика раскрывается набором атрибутов, приведённых на рисунке ниже.

Большая часть из этих атрибутов качества могут быть описаны и реализованы как обычные функциональные требования, а значит имеют непосредственное влияние на архитектуру системы. Более того, архитектура даёт осязаемый инструмент для управления этими атрибутами.

Традиционная точка зрения в разработке требований заключается в том, что требования должны быть ориентированы на потребности пользователей и свободны от каких-либо архитектурных решений. Работой с требованиям занимаются отдельные люди, бизнес-аналитики, которые потом передают извлечённые знания техническим специалистам. Но за пределами огромных проектов такое редко бывает. В небольших проектах архитектор проекта чаще всего активно привлекается к общению с заказчиками и разработке требований (извлечению, анализу, проверке) на начальном этапе проекта, да и процесс управления требований в ходе разработки проекта также не обходит архитектора стороной, поскольку имеет непосредственное влияние на структуру системы. Это позволяет архитектору как изначально иметь больше информации для проектирования системы, так и “держать руку на пульсе” по ходу её создания.

Возвращаясь к примеру про строительство, когда мы планируем или проектируем дом или квартиру, мы думаем в терминах комнат, размерах и типах окон и о том, хотим мы газовую плиту или электрическую. Никто не говорит о средствах предоставления предоставления защиты от осадков, средствах предоставления адекватного освещения или средств для приготовления горячей пищи. Мы имеем достаточный опыт, чтобы сразу оперировать конкретными вещами, которые нам нужны, и это позволяет сделать более точную оценку по срокам и цене. Сравнивая архитектуру других зданий и проецируя собственный опыт на новый проект, мы можем чётко сказать то, что нам нужно.

В программировании примерно так же. Если стараться отделить требования от того, как они будут воплощены в жизнь, то часто и требования то сложно будет

сформулировать. Понимание о том, что возможно в рамках пользовательских интерфейсов, на что способны вычислительные устройства, какого рода сервисы могут быть предоставлены помогает с одной стороны привязать наши “хотелки” к реальной жизни, а с другой — более здраво оценивать работу. То есть существующие архитектурные решения могут давать словарь при общении с заказчиком о том, что ему нужно. Но при этом надо отдавать себе отчёт, что подобная схема может работать прямо противоположно в случае какого-то рода инноваций, а разработка ПО сейчас — гораздо более инновационная область, чем строительство. Чтобы залезть на сарай, достаточно лестницы. Чтобы залезть на двухэтажный дом, нужна большая лестница. Но на небоскрёб таким образом не залезть никак, нужен принципиально другой механизм — лифт. А чтобы попасть на Луну, нужен ещё один принципиальный шаг в развитии устройств. Так что если создаётся более-менее типовое решение, то существующие подходы могут и даже должны использоваться при обсуждении решения, но если создаётся решение в новой области (greenfield development), то существующие решения могут лишь всё запутать.

Проектирование

Это, собственно, фаза, на которой и происходит проектирование архитектуры — определение ключевых решений, которые лягут в основу разрабатываемой системы. Но, как мы говорили выше, высокоуровневые решения, которые определяют общее направление развития системы, принимаются не только на этой стадии. Понятно, что архитектурные решения могут приниматься и потом по ходу реализации проекта, но обычно это стадия, где архитектурным вопросам уделяют больше всего внимания.

В зависимости от выбранной модели жизненного цикла данная стадия может выполняться произвольное число раз. В классической водопадной схеме на этой стадии на основе требований, полученных на прошлой фазе, формируется пакет проектной документации, который отдаётся на следующую стадию для реализации в коде. В модных сейчас итеративно-инкрементальных моделях фаза проектирования часто выделяется до начала проекта, а потом в том или ином виде случается на каждой итерации (которые обычно занимают 2-4 недели) и может раз за разом существенно влиять на архитектуру проекта (особенно учитывая серьёзную необходимость в практически непрерывном рефакторинге в agile-методологиях).

Вот примеры задач, с которыми приходится сталкиваться архитектору на стадии проектирования:

- определение базового архитектурного стиля (например, глобальная многоуровневая или микросервисная архитектура) и структуры приложения;
- определение функционального поведения (“такие-то этапы должны идти друг за другом”);
- осуществление декомпозиции на модули (“такие-то элементы будут организованы и связаны друг с другом таким-то образом”) и планирование их взаимодействия (“взаимодействие модулей будет осуществляться только через сообщения”);
- выбор стратегии и деталей реализации:
    - используется ли кодогенерация?
    - используются ли сторонние фреймворки?

    - используются ли сторонние библиотеки?
    - какая часть функциональности будет реализовывать “вручную”?
- вопросы распределённости/децентрализованности приложения;
- вопросы безопасности и прочие нефункциональные требования;
- вопросы локализации;
- вопросы размещения;
- вопросы обновления;
- и т.д.

В работу архитектора в серьёзных проектах входит в том числе и продумывание всего того, что в проектах поменьше часто люди забывают делать или уделяют не так много внимания: как, где и каким образом будет осуществляться логирование, обработка ошибок и т.п. Эти чисто реализационные решения должны продумываться до написания кода, иначе потом чаще всего оказывается, что эти механизмы становится крайне сложно добавить.

Ещё один момент, который не даёт покоя проектировщикам на этой стадии -- адаптируемость системы для будущих изменений при её развитии. Если система не получится мертворождённой, то в неё заведомо потребуется вносить какие-то изменения, в результате чего каждый архитектор пытается делать свою архитектуру как можно более гибкой и адаптируемой. В некотором смысле назначение архитектуры как раз и состоит в том, чтобы обеспечить безболезненное внесение изменений при развитии системы. Однако, тут есть три возможных варианта развития событий, сложно сказать, какой из которых хуже.

- *Аналитический паралич* -- проектировщик боится принять те или иные решения, так как они могут не дать возможности реализовать те или иные решения в будущем. Он честно пытается предусмотреть все возможные последствия, в результате процесс проектирования сильно затягивается. В попытке зафиксировать все возможные варианты архитектурная документация распухает, и есть реальный шанс вообще не добраться до этапа разработки. Или разработчики не выдерживают и начинают реализовывать систему, исходя из своих соображений об архитектуре.
- *Сверхобобщённая архитектура* -- никакие решения не фиксируются, оставляя как можно больше на стадию разработки или даже сопровождения. Архитектура строится наиболее гибкой и практически полностью состоит из точек расширения. Такую систему крайне сложно понимать и изменять, связи между компонентами зачастую неявные ввиду их абстрактности. Здесь явно прослеживается противостояние между понимаемостью и гибкостью системы, компромисс между которыми должен находить проектировщик каждой системы.
- *Архитектурное невежество* -- игнорирование потенциальных изменений системы в будущем, архитектура затачивается сугубо на реализацию известных требований и не предусматривает никаких точек изменения/расширения. Разумеется, невозможно знать точно, какие изменения потребуется вносить, но провести анализ имеющейся информации, выявить наиболее вероятных кандидатов и оставить возможности для их безболезненной реализации вполне реально. Особенно учитывая, что есть вполне известные классы изменений (мы поговорим про них на следующих лекциях).

Реализация

Задача этой стадии — создание исполняемого кода в соответствии с разработанной архитектурой. При этом надо отдавать себе отчёт в том, что в живом и развивающемся проекте могут меняться как окружение разрабатываемой системы и требования к ней, так и понимание разработчиками того, как эти требования должны лучшим образом воплощаться в код. В этом контексте выделяют понятие архитектуры предписывающей (prescriptive architecture, PA) и описательной (descriptive architecture, DA). Первая фиксирует то, как архитектура была задумана проектировщиком, в вторая — то, что в итоге было построено по коду, то есть что в итоге получилось при реализации PA. В идеальном мире PA и DA совпадают, но если PA и DA начинают расходиться, говорят о деградации архитектуры. Она опять же бывает двух видов:

- architectural drift — добавляем в DA то, чего ещё нет в PA, и при этом оно там ничего не ломает (например, добавили связь между двумя компонентами для целей оптимизации производительности);
- architectural erosion — добавляется в DA что-то, что ломает PA.

В принципе и то, и другое дорого и этого стоит избегать, но если очень надо, то важно не терять изменения в архитектуре и аккуратно их фиксировать.

Тестирование

Тестирование традиционно — это деятельность, которая позволяет находить ошибки в программе (ну и далее, понятно, исправлять их, тем самым повышая качество кода). Обычно тестированию подвергается код, который проверяется на функциональную корректность и что-нибудь типа производительности. В классическом водопаде эта фаза идёт сразу после разработки, однако в более гибких процессах эта деятельность идёт параллельно с разработкой (иногда даже немного опережая разработку). При этом можно тестировать не только код, но и дизайн архитектуры, и даже требования (особенно если эти документы создавались не на доске или листе бумаги, а с использованием автоматизированных средств). Ведь чем раньше находятся ошибки в фундаментальных основах приложения, тем дешевле стоит их исправление.

Архитектурная модель может быть проверена на целостность, синтаксическую и семантическую корректность: наличие значений ключевых свойств, корректность связей между элементами, отсутствие антипаттернов, использование алгоритмов control flow analysis/data flow analysis, поиск проблем многопоточной работы, потенциальных проблем с безопасностью, примерная оценка размера и сложности как отдельных компонент, так и системы в целом.

Архитектура также должна быть проверена на соответствие требованиям. Даже если требования были оформлены как набор описаний на естественном языке, подобную проверку можно осуществить вручную.

Архитектура может быть использована как исходные данные для формирования стратегий тестирования кода. В частности, определяя то, как структурно устроен код, архитектура представляет собой набор исходных данных для интеграционного и системного тестирования.

Развитие и поддержка

Эта фаза по сути наступает после выхода системы в релиз. В неё входят как исправления ошибок, так и последующие доработки для выпуска новых версий. При этом происходит “откат” на одну из предыдущих стадий (возможно даже на стадию выявления требований, если всё совсем плохо), и всё повторяется заново.

Самый большой риск для архитектуры на данном этапе — потенциальная деградация. Особенно при ad-hoc изменениях типа исправления ошибок очень велик соблазн забыть про архитектуру и внести изменения только в код. Это плохо не только потому, что архитектура начинает “отставать” от кода (а хуже никакой документации может быть только некорректная документация), но и потому, что ad-hoc изменения без учёта архитектуры могут не учитывать какие-то важные моменты и привносить новые ошибки. В этом смысле всегда рекомендуется откатываться к фазе проектирования, на котором пересматривать архитектуру с учётом планируемых изменений.

Другая проблема -- внесение в код изменений, которые нарушают заложенные в нём архитектурные принципы. Это может происходить и сознательно, когда на краткосрочной перспективе время важнее качества кода, либо несознательно из-за отсутствия достаточной квалификации разработчиков или непрозрачности принятых изначально решений.

Литература

1. Ф. Брукс. Мифический человеко-месяц
2. Richard N. Taylor, Software Architecture: Foundations, Theory, and Practice
3. M. A. Babar, A. Brown, I. Mistrik. Agile Software Architecture