Архитектура приложений

Основные понятия

Понятие архитектуры и дизайна

“Архитектура отражает важные проектные решения по формированию системы, где важность определяется стоимостью изменений” **(Гради Буч)**

“Если вы думаете, что хорошая архитектура стоит дорого, попробуйте плохую архитектуру” **(Джозеф Йодер)**

“Архитектура - набор верных решений, которые хотелось бы принять на ранних этапах работы над проектом, но которые не более вероятны, чем другие” **(Ральф Джонсон)**

Понятие архитектуры и дизайна

Архитектура программной системы - это форма, которая придается системе ее создателями. Эта форма образуется декомпозицией (делением) системы на компоненты, их организацией и определением способов взаимодействия между ними.

Цель формы - упростить разработку, развертывание и сопровождение программной системы, содержащейся в ней.

Главная стратегия - сделать связанность компонентов системы максимально низкой таким образом, чтобы компоненты могли быть легко заменяемыми и расширяемыми.

Важно понимать, что архитектура системы слабо влияет на работу этой системы. Существует много систем с ужасной архитектурой, которые прекрасно работают.

Главное предназначение архитектуры — поддержка жизненного цикла системы . Хорошая архитектура делает систему легкой в освоении, простой в разработке, сопровождении и развертывании . Конечная ее цель — минимизировать затраты на протяжении срока службы системы и максимизировать продуктивность программиста .

Понятие архитектуры и дизайна

Программной системе, которую трудно развивать, едва ли стоит рассчитывать на долгую и здоровую жизнь . Поэтому архитектура системы должна делать ее простой в развитии для тех, кто ее разрабатывает .

Чтобы достичь высокой эффективности, программная система должна легко разворачиваться . Чем выше стоимость развертывания, тем ниже эффективность системы . Соответственно, целью архитектуры является создание системы, которую можно развернуть в одно действие .

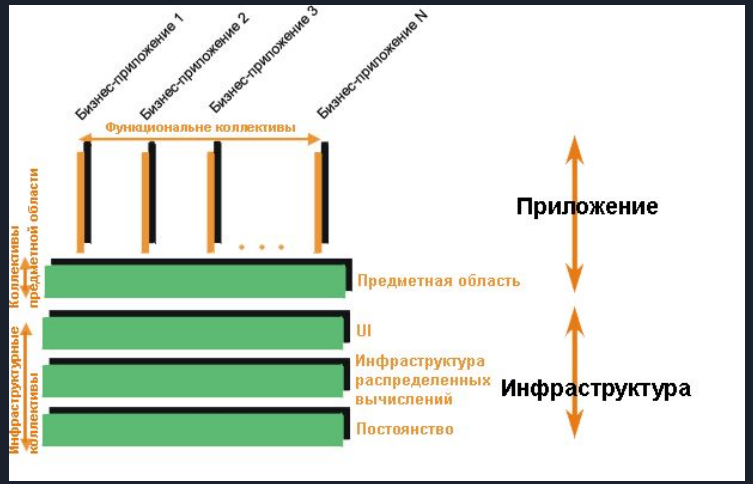
Из всех аспектов программной системы сопровождение является самым дорогостоящим . Бесконечный парад новых особенностей и неизбежная череда дефектов и исправлений требуют огромных человеческих трудозатрат.

Основная стоимость сопровождения складывается из стоимости исследования и рисков. Продуманная архитектура значительно снижает эти затраты . Разделив систему на компоненты и изолировав их за устойчивыми интерфейсами, можно осветить пути к будущим особенностям и существенно уменьшить риск непреднамеренных ошибок .

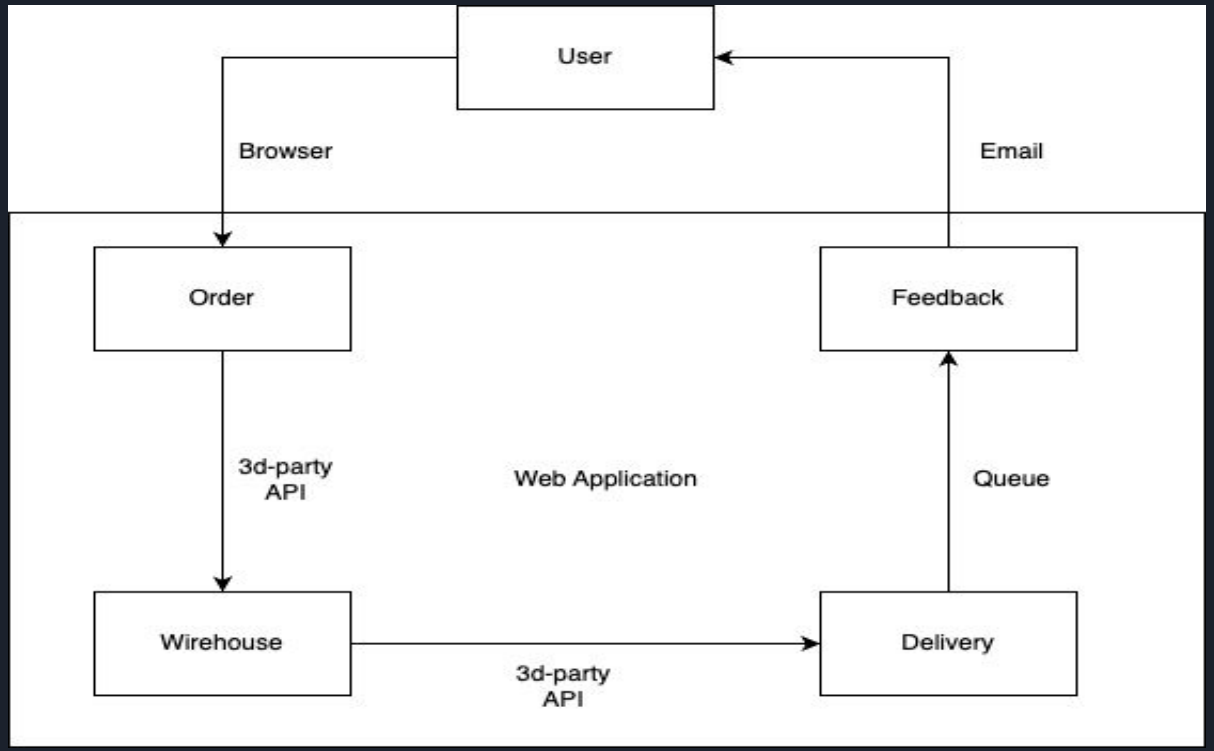
Понятие архитектуры и дизайна

Хорошая архитектура должна обеспечивать:

- Разнообразие вариантов использования и эффективную работу системы .
- Простоту сопровождения системы .
- Простоту разработки системы .
- Простоту развертывания системы .



Примеры


User
Browser
Email
Order
Feedback
3d-party
API
Web Application
Queue
Wirehouse
Delivery
3d-party
API

Корпоративные приложения

К числу корпоративных приложений относятся, бухгалтерский учет, ведение медицинских карт пациентов, экономическое прогнозирование, анализ кредитной истории клиентов банка, страхование, внешнеэкономические торговые операции и т.п.

Корпоративными приложениями не являются средства обработки текста, регулирования расхода топлива в автомобильном двигателе, управления лифтами и оборудованием телефонных станций, автоматического контроля химических процессов, а также операционные системы, компиляторы, игры и т.д.

Корпоративные приложения обычно подразумевают необходимость долговременного (иногда в течение десятилетий) хранения данных. Данные зачастую способны "пережить" несколько поколений прикладных программ, предназначенных для их обработки, аппаратных средств, операционных систем и компиляторов. В продолжение этого срока структура данных может подвергаться многочисленным изменениям в целях сохранения новых порций информации без какого-либо воздействия на старые. Даже в тех случаях, когда компания осуществляет революционные изменения в парке оборудования и номенклатуре программных приложений, данные не уничтожаются, а переносятся в новую среду.

Корпоративные приложения

Корпоративные приложения подразумевают работу с большим объемом данных и множеством пользователей, которые работают с системой параллельно.

Система должна обеспечить возможность одновременной работы нескольких пользователей с одним и тем ресурсом, не блокируя и не вызывая коллизий.

Система должна иметь уровни доступа, соответствующие бизнес-правилам.

Система должна предусмотреть возможность интеграции с внешними сервисами.

Система должна быть отказоустойчивой и иметь возможность быстрого восстановления данных при утере.

Система должна быть масштабируемой. Добавление нового функционала не должно влечь за собой расходы большого количества ресурсов на рефакторинг смежных подсистем.

Типы корпоративных приложений

1. Бизнес - пользователь (B2C).

Подобная система предназначена для обслуживания большого количества пользователей одновременно, поэтому проектное решение должно быть не только эффективным по критерию использования ресурсов, но и масштабируемым: все, что требуется для повышения пропускной способности такой системы — это приобретение дополнительного аппаратного обеспечения.

Для подобной системы важным является так называемый Uptime, процент времени, в течение которого система находится в рабочем состоянии. К идеалу в 100% стремятся все системы, однако на текущий момент к такому показателю приблизились только гиганты рынка, такие как Google, Amazon, Facebook и др. Максимальный Uptime, который обеспечивают эти системы составляет 99,95%.

Также b2c системы должны иметь отличный интерфейс для работы пользователей (UX): ведь задача бизнеса заключается в том, чтобы как можно больше пользователей воспользовались его услугами или приобрели товары.

Типы корпоративных приложений

2. Бизнес - бизнес - (B2B).

Примерами такой системы могут быть системы обслуживания корпоративных клиентов в банке, платежная система - эквайринг, сервисы оптовых закупок. Т.е. в такой системе пользователь - какой-либо бизнес, которому необходима услуга или товар, которые обеспечивает система.

Для подобной системы обычно характерно отсутствие большой посещаемости: работающих бизнесов гораздо меньше, чем обычных потребителей. Однако есть и обратная сторона: потеря одного клиента - бизнесмена гораздо существеннее, чем потеря одного потребителя.

Поэтому, при создании таких систем особенно важным является бесперебойная и выверенная работа основного функционала системы.

Также важным для пользователя является обеспечение системой безопасности данных: ведь зачастую посредством системы обеспечивается функционирование бизнеса клиента, т.е. взаимодействие с его пользователями.

Термины масштабируемой системы

Время отклика (response time) — промежуток времени, который требуется системе, чтобы обработать запрос извне, подобный щелчку на кнопке графического интерфейса или вызову функции API сервера.

Быстрота реагирования (responsiveness) — скорость подтверждения запроса (не путать с временем отклика— скоростью обработки).
Эта характеристика во многих случаях весьма важна, поскольку интерактивная система, пусть даже обладающая нормальным временем отклика, но не отличающаяся высокой быстротой реагирования, всегда вызы- вает справедливые нарекания пользователей.
Если, прежде чем принять очередной запрос, система должна полностью завершить обработку текущего, параметры времени отклика и быстроты реагирования, по сути, совпадают.
Если же система способна подтвердить получение запроса раньше, ее быстрота реагирования выше. Например, приме-нение динамического индикатора состояния процесса копирования повышает быстроту реагирования экранного интерфейса, хотя никак не сказывается на значении времени отклика.

Термины масштабируемой системы

Время задержки (latency) — минимальный интервал времени до получения какого- либо отклика (даже если от системы более ничего не требуется). Параметр приобретает особую важность в распределенных системах. Если я "прикажу" программе ничего не делать и сообщить о том, когда именно она закончит это "ничегонеделание", на персональном компьютере ответ будет получен практически мгновенно.
Если же программа выполняется на удаленной машине, придется подождать, вероятно, не менее нескольких секунд, пока запрос и ответ проследуют по цепочке сетевых соединений. Снизить время задержки разработчику прикладной программы не под силу. Фактор задержки — главная причина, побуждающая минимизировать количество удаленных вызовов.

Пропускная способность (thmughput) — количество данных (операций), передаваемых (выполняемых) в единицу времени. Если, например, тестируется процедура копирования файла, пропускная способность может измеряться числом байтов в секунду.
В корпоративных приложениях обычной мерой производительности служит число транзакций в секунду (transactions per second — tps), но есть одна проблема — транзакции различаются по степени сложности. Для конкретной системы необходимо рассматривать смесь "типо- вых" транзакций.

Термины масштабируемой системы

В контексте рассмотренных терминов под производительностью можно понимать один из двух параметров — время отклика или пропускную способность, в частности тот, который в большей степени отвечает природе ситуации.

Иногда бывает трудно судить о производительности, если, например, использование некоторого решения повышает пропускную способность, одновременно увеличивая время отклика.

С точки зрения пользователя, значение быстроты реагирования может оказаться более важным, нежели время отклика, так что улучшение быстроты реагирования ценой потери пропускной способности или возрастания времени отклика вполне способно повысить производительность.

Термины масштабируемой системы

Загрузка (load) — значение, определяющее степень "давления" на систему и измеряемое, скажем, количеством одновременно подключенных пользователей. Параметр загрузки обычно служит контекстом для представления других функциональных характеристик, подобных времени отклика. Так, нередко можно слышать выражения наподобие следующего: "время отклика на запрос составляет 0,5 секунды для 10 пользователей и 2 секунды для 20 пользователей".

Чувствительность к загрузке (load sensitivity) — выражение, задающее зависимость времени отклика от загрузки.

Предположим, что система A обладает временем отклика, равным 0,5 секунды для 10-20 пользователей, а система B — временем отклика в 0,2 секунды для 10 пользователей и 2 секунды для 20 пользователей. Это дает основание утверждать, что система A обладает меньшей чувствительностью к загрузке, нежели система B. Можно воспользоваться и термином ухудшение (degradation), чтобы подчеркнуть факт меньшей устойчивости параметров системы B.

Термины масштабируемой системы

Эффективность (efficiency) — удельная производительность в пересчете на одну единицу ресурса. Например, система с двумя процессорами, способная выполнить 30 tps, более эффективна по сравнению с системой, оснащенной четырьмя аналогичными процессорами и обладающей продуктивностью в 40 tps.

Мощность (capacity) — наибольшее значение пропускной способности или загрузки. Это может быть как абсолютный максимум, так и некоторое число, при котором величина производительности все еще превосходит заданный приемлемый порог.

Способность к масштабированию (scalability) — свойство, характеризующее поведение системы при добавлении ресурсов (обычно аппаратных). Масштабируемой принято считать систему, производительность которой возрастает пропорционально объему приобщенных ресурсов (скажем, вдвое при удвоении количества серверов).

Термины масштабируемой системы

Вертикальное масштабирование (vertical scalability, scaling up) - это увеличение мощности отдельного сервера (например, за счет увеличения объема оперативной памяти).

Горизонтальное масштабирование (horizontal scalability, scaling out)— это наращивание потенциала системы путем добавления новых серверов.

Проектируя корпоративную систему, часто следует уделять больше внимания обеспечению средств масштабирования, а не наращиванию мощности или повышению эффективности. Производительность масштабируемой системы всегда удастся увеличить, если такая потребность действительно возникнет.

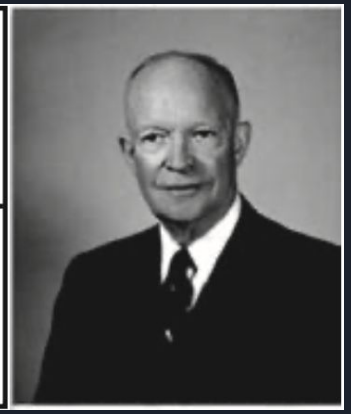
Зачастую использовать более новое оборудование просто выгоднее, нежели заставлять программу работать на устаревшей технике. Если корпоративное приложение поддается масштабированию, добавить несколько серверов дешевле, чем приобрести услуги нескольких программистов.

Приоритеты: срочное и важное

Рассмотрим матрицу президента Дуайта Дэвида Эйзенхауэра для определения приоритета между важностью и срочностью

У меня есть два вида дел, срочные и важные.
Срочные дела, как правило, не самые важные, а важные — не самые срочные (с)

Важные Срочные	Важные Не срочные
Не важные Срочные	Не важные Не срочные



Приоритеты: срочное и важное

Срочное действительно редко бывает важным, а важное — срочным .

Первая ценность программного обеспечения — поведение — это нечто срочное, но не всегда важное .

Вторая ценность — архитектура — нечто важное, но не всегда срочное .

Конечно, имеются также задачи важные и срочные одновременно и задачи не важные и не срочные . Все эти четыре вида задач можно расставить по приоритетам .

1. **Срочные и важные .**
2. **Не срочные и важные .**
3. **Срочные и не важные .**
4. **Не срочные и не важные .**

Приоритеты: срочное и важное

Обратите внимание, что архитектура кода — важная задача — оказывается в двух верхних позициях в этом списке, тогда как поведение кода занимает первую и *третью* позиции .

Руководители и разработчики часто допускают ошибку, поднимая пункт 3 до уровня пункта 1 .
Иными словами, они неправильно отделяют срочные и не важные задачи от задач, которые по-настоящему являются срочными и важными . Эта ошибочность суждений приводит к игнорированию важности архитектуры системы и уделению чрезмерного внимания не важному поведению .

Разработчики программного обеспечения оказываются перед проблемой, обусловленной неспособностью руководителей оценить важность архитектуры .

Но именно для ее решения они и были наняты . Поэтому разработчики должны всякий раз подчеркивать приоритет важности архитектуры перед срочностью поведения .

Архитектура. Парадигмы программирования

Структурное программирование

Первой, получившей всеобщее признание (но не первой из придуманных), была парадигма структурного программирования, предложенная Эдсгером Вибе Дейкстрой в 1968 году .
Дейкстра показал, что безудержное использование переходов (инструкций goto) вредно для структуры программы .
Он предложил заменить переходы более понятными конструкциями if/then/else и do/while/until .

Подводя итог, можно сказать, что:

Структурное программирование накладывает ограничение на прямую передачу управления.

Архитектура. Структурное программирование

Программа любой сложности содержит слишком много деталей, и стоит упустить из виду одну маленькую деталь, и программа, которая кажется работающей, может завершаться с ошибкой в самых неожиданных местах .

В качестве решения Дейкстра предложил применять математический аппарат *доказательств* .
Оно заключалось в построении евклидовой иерархии постулатов, теорем, следствий и лемм . Дейкстра полагал, что программисты смогут использовать эту иерархию подобно математикам .

Иными словами, программисты должны использовать проверенные структуры и связывать их с кодом, в правильности которого они хотели бы убедиться .

Для этого необходимо разработать методику написания доказательств на простых алгоритмах .

Архитектура. Структурное программирование

В ходе исследований Дейкстра обнаружил, что в некоторых случаях использование инструкции goto мешает рекурсивному разложению модулей на все меньшие и меньшие единицы и тем самым препятствует применению принципа «разделяй и властвуй», что является необходимым условием для обоснованных доказательств .

В связи с этим Дейкстра воспользовался результатами исследований Бёма и Якопини, доказавшими, что любую программу можно написать, используя всего три структуры: последовательность, выбор и итерации .

Вывод: управляющие структуры, делающие доказуемой правильность модуля, в точности совпадали с набором структур, минимально необходимым для написания любой программы .

Это и есть структурное программирование .

Архитектура. Структурное программирование

Структурное программирование сводится к функциональной декомпозиции.

Декомпозиция - это разделение программы на ряд функций верхнего уровня, а их в свою очередь - на еще более низкого, и так до момента, пока каждый “лист”, т.е. единичная функция не станет атомарной.

Этим подходом пользуются в современном мире при планировании и при оценке работ.

Программные архитекторы стремятся определить модули, компоненты и службы, неправильность которых легко можно было бы доказать (протестировать) .

Для этого они используют ограничения, напоминающие ограничения в ООП, хотя и более высокого уровня .

Архитектура. Парадигмы программирования

Объектно-ориентированное программирование

Второй парадигмой, получившей широкое распространение, стала парадигма, появившаяся в 1966-м, и предложенная Оле-Йоханом Далем и Кристеном Нюгором .
Эти два программиста заметили, что в языке ALGOL есть возможность переместить кадр стека вызова функции в динамическую память (кучу), благодаря чему локальные переменные, объявленные внутри функции, могут сохраняться после выхода из нее .

В результате функция превращалась в конструктор класса, локальные переменные — в переменные экземпляра, а вложенные функции — в методы . Это привело к открытию полиморфизма через строгое использование указателей на функции .

Подводя итог, можно сказать, что:

Объектно-ориентированное программирование накладывает ограничение на косвенную передачу управления.

Архитектура. ООП

Что такое ОО (П)?

- комбинация данных и функций
- способ моделирования реального мира
- инкапсуляция, наследование и полиморфизм

ОО дает, посредством поддержки полиморфизма, абсолютный контроль над всеми зависимостями в исходном коде.

Факт поддержки языками ОО надежного и удобного механизма полиморфизма означает, что *любую зависимость исходного кода, где бы она ни находилась, можно инвертировать* .

Это позволяет архитектору создать архитектуру со сменными модулями (плагинами), в которой модули верхнего уровня не зависят от модулей нижнего уровня.

Низкоуровневые детали не выходят за рамки модулей плагинов, которые можно развертывать и разрабатывать независимо от модулей верхнего уровня.

Архитектура. Парадигмы программирования

Функциональное программирование

Третьей парадигмой, начавшей распространяться относительно недавно, является самая первая из придуманных .
Фактически изобретение этой парадигмы предшествовало появлению самой идеи программирования .
Парадигма функционального программирования является прямым результатом работы Алонзо Чёрча, который в 1936 году изобрел лямбда-исчисление (λ-исчисление), исследуя ту же математическую задачу, которая примерно в то же время занимала Алана Тьюринга .
Его λ-исчисление легло в основу языка LISP, изобретенного в 1958 году Джоном Маккарти .
Основополагающим понятием λ-исчисления является неизменяемость — то есть невозможность изменения значений символов . Фактически это означает, что функциональный язык не имеет инструкции присваивания . В действительности большинство функциональных языков обладает некоторыми средствами, позволяющими изменять значение переменной, но в очень ограниченных случаях .

Подводя итог, можно сказать, что:

Функциональное программирование накладывает ограничение на присваивание.

Архитектура. Функциональное программирование

Пример: вывод квадратов первых 25 целых чисел

```
public class Squint {
  public static void main(String args[]) {
    for (int i=0; i<25; i++)
      System.out.println(i*i);
  }
}
```

Java

```
(println (take 25 (map (fn [x] (* x x)) (range))))
```

Lisp

```
(println ;_____________________ Вывести
  (take 25 ;___________________ первые 25
    (map (fn [x] (* x x)) ;__ квадратов
      (range)))) ;____________ целых чисел
```

Архитектура. Функциональное программирование

println, take, map и range — это функции . В языке Lisp вызов функции производится помещением ее имени в круглые скобки . Например, (range) — это вызов функции range .

Выражение (fn [x] (* x x)) — это анонимная функция, которая, в свою очередь, вызывает функцию умножения и передает ей две копии входного аргумента . Иными словами, она вычисляет квадрат заданного числа .

Алгоритм:

- функция range возвращает бесконечный список целых чисел, начи-
  ная с 0;
- этот список передается функции map, которая вызывает анонимную функцию для вычисления квадрата каждого элемента и производит бес- конечный список квадратов;
- список квадратов передается функции take, которая возвращает новый список, содержащий только первые 25 элементов;
- функция println выводит этот самый список квадратов первых 25 целых чисел .

Архитектура. Функциональное программирование

Вывод: переменные в функциональных языках *не изменяются* .

Все состояния гонки (race condition), взаимоблокировки (deadlocks) и проблемы параллельного обновления обусловлены изменяемостью переменных .

Если в программе нет изменяемых переменных, она никогда не окажется в состоянии гонки и никогда не столкнется с проблемами одновременного изменения . В отсутствие изменяемых блокировок программа не может попасть в состояние взаимоблокировки .

Один из самых общих компромиссов, на которые приходится идти ради неизменяемости, — деление приложения или служб внутри приложения на изменяемые и неизменяемые компоненты .

Неизменяемые компоненты решают свои задачи исключительно функциональным способом, без использования изменяемых переменных . Они взаимодействуют с другими компонентами, не являющимися чисто функциональными и допускающими изменение состояний переменных.

Архитектура. Парадигмы программирования

Итоги:

1. *Объектно-ориентированное программирование накладывает ограничение на косвенную передачу управления.*
2. *Структурное программирование накладывает ограничение на прямую передачу управления.*
3. *Функциональное программирование накладывает ограничение на присваивание.*

Парадигмы говорят нам не столько *что делать*, сколько *чего нельзя делать* .

Три парадигмы вместе отнимают у нас инструкции goto, указатели на функции и оператор присваивания .

Если есть четкие ограничения - это лучше, чем невероятные возможности.