Урок 1

Архитектура Java-приложений

Что такое (понятие) Архитектура программного обеспечения

Представление об Архитектуре программного обеспечения как о концепции берет свое начало в исследовательских работах Эдсгера Дейкстры и Дэвида Парнаса на рубеже 60-70-х годов прошлого века. Эти ученые выявили критическую важность структуры программной системы. В 90-х годах потребность изучения Архитектуры получила дополнительные стимулы в результате роста сложности программного обеспечения, что привело к формированию учения об Архитектуре как об обособленной отрасли научной мысли. Знания об Архитектуре ПП (программного продукта) – предмет целенаправленного поиска решения возникающих при создании ПП задач.

Можно говорить об Архитектуре как о совокупности трудно изменяемых подходов (архитектурных решений) к проектированию ПП, или как о максимально общем виде (структуре) проектируемого решения, без деталей реализации.

Например, Мартин Фаулер в [2] дает такое определение: «Обычно это согласие в вопросе идентификации главных компонентов системы и способов их взаимодействия, а также выбор таких решений, которые интерпретируются как основополагающие и не подлежащие изменению в будущем. Если позже оказывается, что нечто изменить легче, чем казалось вначале, это «нечто» исключается из Архитектурной категории.

Пример из реальной жизни: кирпич, из которого построено здание, легко заменяем по аналогии с легкой сменой реализации интерфейса или движка СУБД, в то время как общий облик здания достаточно крайне трудноизменяем.

Очевидно, что спорткомплекс в виде небоскреба едва ли будет удобно строить или эксплуатировать в дальнейшем.

Впервые аналогию между разработкой программного обеспечения и классической Архитектурой провел Кристофер Александер в [3]. Опираясь на его мысль, можно составить следующую мотивирующую таблицу.

Традиция	**Программирование**	**Функция**	**Мотивация, в единицу времени**
Архитектор	Архитектор	Задает А-системы	$$$
Проектировщик	Developer (Senior)	Проектирует систему, используя крупные блоки – шаблоны	$$
Строитель	Coder/программист (Junior)	Реализует	$

Мораль: «Плох тот солдат, который не мечтает стать генералом», – Суворов А.В.

Стандарт IEEE 1471 дает формальное определение следующим образом. Архитектура – это базовая организация системы, воплощенная в ее компонентах, их отношениях между собой и с окружением, а также принципы, определяющие проектирование и развитие системы [7][8].

Как в обычной жизни, где никто не строит здание без Архитектурного проекта, так и у нас: подходить к созданию ПП без Архитектуры – крайне легкомысленный шаг, за исключением простых задач, которые можно реализовать без проекта.

Преимущества Архитектуры как абстракции сложной системы

1. Архитектура – это база для анализа поведения системы еще до ее реализации, что позволяет устранить узкие места еще на этапе проектирования.
2. Позволяет повторно использовать найденные ранее архитектурные решения, что, в свою очередь:
    a. Повышает скорость разработки.
    b. Повышает качество разработки.
    c. Снижает риски и вероятность провала.
    d. Снижает стоимость разработки.
    e. Увеличивает совместимость различных систем.
    f. Снимает необходимость снова и снова изобретать очередной велосипед.
3. Обеспечивает раннее принятие проектных решений, оказывающих в дальнейшем огромное влияние на разработку, внедрение и поддержку ПП, что позволяет:
    a. Выдерживать график разработки (внедрения, развертывания и т.д.).
    b. Управлять стоимостью разработки.
    c. Распределить во времени принятие решений, «съесть слона по частям» во времени.
4. Облегчает взаимодействие разработчиков с кругом заинтересованных лиц (едва ли сведущих в программировании), что способствует созданию системы, более полно отвечающей истинным требованиям заказчика.

Критерии хорошей Архитектуры

С точки зрения современных разработчиков [5][9], список критериев хорошей Архитектуры выглядит так:

- Эффективность системы. В первую очередь программа, конечно же, должна решать поставленные задачи и хорошо выполнять свои функции, причем в различных условиях.
- Гибкость системы. Любое приложение приходится менять со временем – меняются требования, добавляются новые. Чем быстрее и удобнее можно внести изменения в существующий функционал, чем меньше проблем и ошибок это вызовет, тем гибче и конкурентоспособнее система.
- Расширяемость системы. Возможность добавлять в систему новые сущности и функции, не нарушая ее основной структуры. Архитектура должна позволять легко наращивать дополнительный функционал по мере необходимости. Причем так, чтобы внесение наиболее вероятных изменений требовало наименьших усилий.

- Тестируемость. Код, который легче тестировать, будет содержать меньше ошибок и надежнее работать.

- Сопровождаемость. Хорошая Архитектура должна давать возможность новым членам команды относительно легко и быстро разобраться в системе. Проект должен быть хорошо структурирован. По возможности в системе лучше применять стандартные, общепринятые решения, привычные для большинства программистов.

- Масштабируемость процесса разработки. Возможность сократить срок разработки за счет добавления к проекту новых человеческих ресурсов.

В противовес критериям хорошо спроектированных Архитектур используются и критерии «плохого дизайна», выдвинутые Сергеем Тепляковым в [6].

- Его тяжело изменить, поскольку любое изменение влияет на слишком большое количество других частей системы (Жесткость, Rigidity).

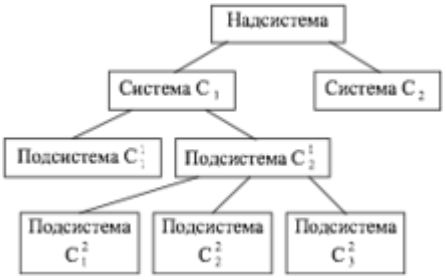
- При внесении изменений неожиданно ломаются другие части системы (Хрупкость, Fragility).

- Код тяжело использовать повторно в другом приложении, поскольку его слишком трудно «выпутать» из текущего приложения (Неподвижность, Immobility).

Формирование, выбор Архитектуры. Декомпозиция

Используя принцип «от простого к сложному» (а более точно в данном контексте – «разделяй и властвуй»), необходимо разбить проектируемую систему на части, реализация которых вызывает меньше вопросов либо уже известна исходя из накопленного опыта. Иными словами, нужно произвести декомпозицию системы.

При декомпозиции руководствуются следующими правилами:



1. Каждое разбиение системы порождает свой уровень сложности – иерархическая декомпозиция.
2. На всех уровнях иерархической декомпозиции система бьется по какому-то одному признаку, например:
    a. Функциональная декомпозиция.
    b. Структурная декомпозиция.
    c. ВременнАя декомпозиция.
    d. И т.п.
3. Полученные подсистемы в сумме должны полностью описывать исходную систему, при этом не пересекаясь.

Вопросы проектирования Архитектуры

При разработке Архитектуры приложения необходимо найти ответы на следующие вопросы:

1. Какие части Архитектуры являются фундаментальными?
2. Какие части Архитектуры скорее всего будут изменяться в дальнейшем под натиском внешних факторов?
3. Проектирование каких частей Архитектуры можно отложить?
4. Основные варианты развития изменений и методы верификации предположений?
5. Что может привести к перепроектированию Архитектуры?

Не пытайтесь создать слишком сложную Архитектуру и не делайте предположений, которые не можете проверить. Лучше оставляйте свои варианты открытыми для изменения в будущем.

Некоторые аспекты дизайна должны быть приведены в порядок на ранних стадиях процесса, потому что их возможная переработка может потребовать существенных затрат. Такие области необходимо выявить как можно раньше и уделить им достаточное количество времени.

Основные принципы проектирования Архитектуры

При проектировании Архитектуры руководствуйтесь следующими основными принципами:

1. Создавайте с расчетом на будущее. Продумайте, как со временем может понадобиться изменить приложение, чтобы оно отвечало вновь возникающим требованиям и задачам, и предусмотрите необходимую гибкость.
2. Создавайте модели для анализа и сокращения рисков. Используйте средства проектирования и системы моделирования, такие как Унифицированный язык моделирования (Unified Modeling Language, UML), и средства визуализации, когда необходимо выявить требования, принять архитектурные и проектные решения и проанализировать их последствия. Тем не менее, не создавайте слишком формализованную модель – она может ограничить возможности для выполнения итераций и адаптации дизайна.
3. Используйте модели и визуализации как средства общения при совместной работе. Для построения хорошей Архитектуры критически важен эффективный обмен информацией о дизайне, принимаемых решениях и вносимых изменениях. Используйте модели, представления и другие способы визуализации Архитектуры для эффективного обмена информацией и связи со всеми заинтересованными сторонами, а также для обеспечения быстрого оповещения об изменениях в дизайне.
4. Выявляйте ключевые инженерные решения. В самом начале проекта уделите достаточное количество времени и внимания для принятия правильных решений, – это обеспечит создание более гибкого дизайна, внесение изменений в который не потребует полной его переработки.
5. Рассмотрите возможность использования инкрементного и итеративного подхода при работе над Архитектурой.
6. Начинайте с базовой Архитектуры, правильно воссоздавая полную картину, после чего проработайте возможные варианты в ходе итеративного тестирования и доработки Архитектуры. Не пытайтесь сделать все сразу: проектируйте настолько, насколько это необходимо для начала тестирования вашего дизайна на соответствие требованиям и допущениям.
7. Усложняйте дизайн постепенно, в процессе многократных пересмотров, чтобы убедиться, прежде всего, в правильности принятых крупных решений и лишь затем сосредотачиваться на деталях. Общей ошибкой является быстрый переход к деталям при ошибочном представлении о правильности крупных решений из-за неверных допущений или неспособности эффективно оценить свою Архитектуру.
8. При тестировании Архитектуры дайте ответы на следующие вопросы:
    a. Какие допущения были сделаны в этой Архитектуре?
    b. Каким явным или подразумеваемым требованиям отвечает данная Архитектура?
    c. Основные риски при использовании такого архитектурного решения?
    d. Каковы меры противодействия для снижения основных рисков?
    e. Является ли данная Архитектура улучшением базовой Архитектуры или одним из возможных вариантов Архитектуры?

Эрозия Архитектуры

Эрозия Архитектуры наблюдаются при разрыве между планируемой и фактически полученной Архитектурой приложения при его реализации. Разрыв между планируемой и фактически полученной Архитектурами иногда называется «технический долг» [15]. Причины его возникновения могут быть заложены как в непродуманной Архитектуре, так и под давлением внешних факторов. Например,

давление бизнеса, когда заказчик требует реализации дополнительного функционала, разработка которого зачастую требует перепроектирования системы. В то же время и сами разработчики склонны к созданию предпосылок для возникновения «технического долга»:

1. Отсутствие тестирования – поощрение быстрой разработки и рискованных исправлений.
2. Использование временных, «уродливых», но быстро реализуемых решений для реализации нового функционала и исправления ошибок.
3. Отсутствие документации приводит к сложностям при необходимости доработки решения или внесению изменений спустя некоторый промежуток времени.
4. Применение сильно связанных компонентов приводит к отсутствию гибкости под натиском изменяющихся требований бизнеса.
5. Отсутствие взаимодействия, – когда база знаний не распространяется по организации и страдает эффективность бизнеса, или младшие разработчики неправильно обучены их наставниками.
6. Разрозненная разработка может вызвать создание «технического долга» вследствие необходимости слияния изменений воедино.
7. Откладывание важных изменений на потом, внесение важных (и необходимых) изменений как можно раньше дешевле последующего полного перепроектирования.

Восстановление Архитектуры. Перепроектирование

Восстановление Архитектуры – это набор методов выделения архитектурной информации на основе анализа нижележащих слоев дизайна, в том числе из программного кода.

Виды Архитектуры приложений

Архитектура клиент/сервер

Клиент/серверная Архитектура обычно разбивает приложение на две части по функциональному признаку. Клиентская часть создает запросы к серверу, сервер авторизует клиента, обрабатывает запрос и передает обратно ответом результат. Описание языка взаимодействия клиента и сервера называется протоколом. Типичные представители Архитектуры клиент/сервер: WWW, FTP, электронная почта и т.п.

Основные преимущества Архитектуры клиент/сервер:

1. Высокая безопасность. Все данные хранятся на сервере, который полностью контролирует доступ к ним (в теории).
2. Более простое администрирование вследствие централизованной авторизации пользователей сервером.
3. Простота обслуживания. Роли и ответственность вычислительной системы распределены между несколькими серверами, общающимися друг с другом по сети. Благодаря этому клиент гарантированно остается неосведомленным и не подверженным влиянию событий, происходящих с сервером (ремонт, обновление либо перемещение).

Тем не менее Архитектура клиент/сервер имеет и недостатки:

1. Тенденцию тесного связывания данных и бизнес-логики приложения на сервере, что может иметь негативное влияние на расширяемость и масштабируемость системы.
2. Зависимость от центрального сервера, что негативно сказывается на надежности системы.

Многослойная Архитектура

Многослойная Архитектура описывает приложение как набор слоев. Каждый слой реализует какую-то одну свою архитектурную задачу, используя при этом «сервисы» (в широком понимании этого слова) непосредственно лежащего под ним слоя, но не через слой ниже. В качестве примера многослойной Архитектуры из смежных областей можно рассмотреть сетевую модель OSI [4]:

	Единица данных	Уровень	Функция	Примеры протоколов
ОС	Поток	Прикладной	Прикладная задача	HTTP, SMTP, DNS, etc.
		Представления	Представление данных, шифрование, etc.	MIME, SSL
		Сеансовый	Взаимодействие хостов (на уровне ОС)	NetBIOS, именов. пайпы
	Сегмент	Транспортный	Соединение конец-в-конец, контроль передачи данных	TCP, UDP
Сеть	Пакет	Сетевой	Логическая адресация и маршрутизация пакетов	IP, ICMP
	Фрейм	Канальный	Физическая адресация	IEEE 802.3, ARP, DHCP
	Бит	Физический	Кодирование и передача данных по физическому каналу	IEEE 802.3

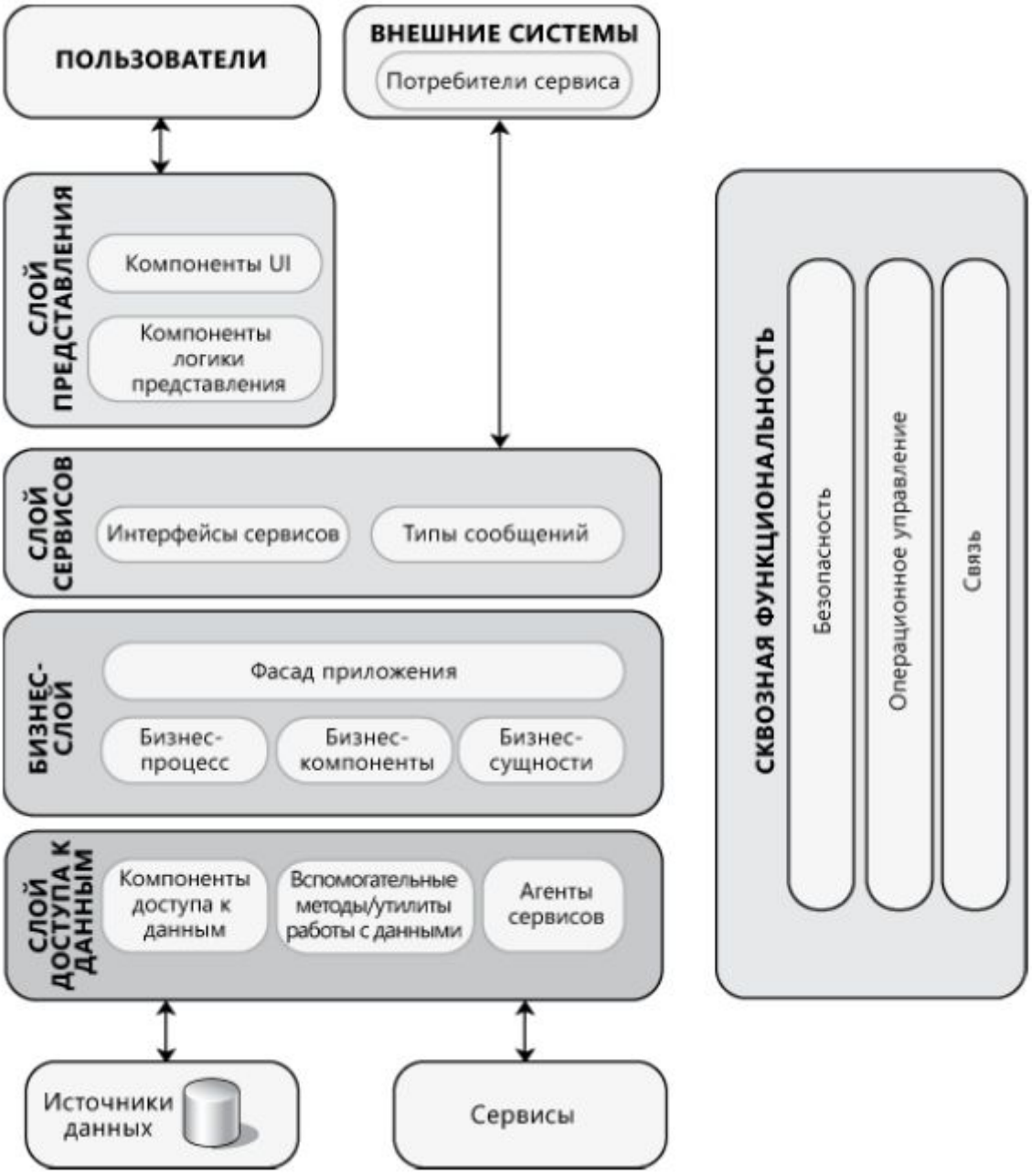
Источник: http://www.doctorrouter.ru/wp-content/uploads/2014/04/86ea4e.png

Преимущества расслоения:

1. Каждый слой самодостаточен и индивидуально тестируем.
2. Каждый слой независим от соседних слоев.
3. Многовариантность реализации отдельно взятого слоя.
4. Стандартизация слоя позволяет многократно его использовать в разных приложениях.
5. Каждый слой может служить основой для нескольких различных слоев более высокого уровня.

Недостатки:

1. Слои разбивают приложение по функциям, но не по смыслу. Добавление/модификация сущности, например, поля БД может спровоцировать каскад изменений в других слоях.
2. Избыточное расслоение может понизить производительность системы.



Архитектура типичного представителя многослойной Архитектуры, источник [12].

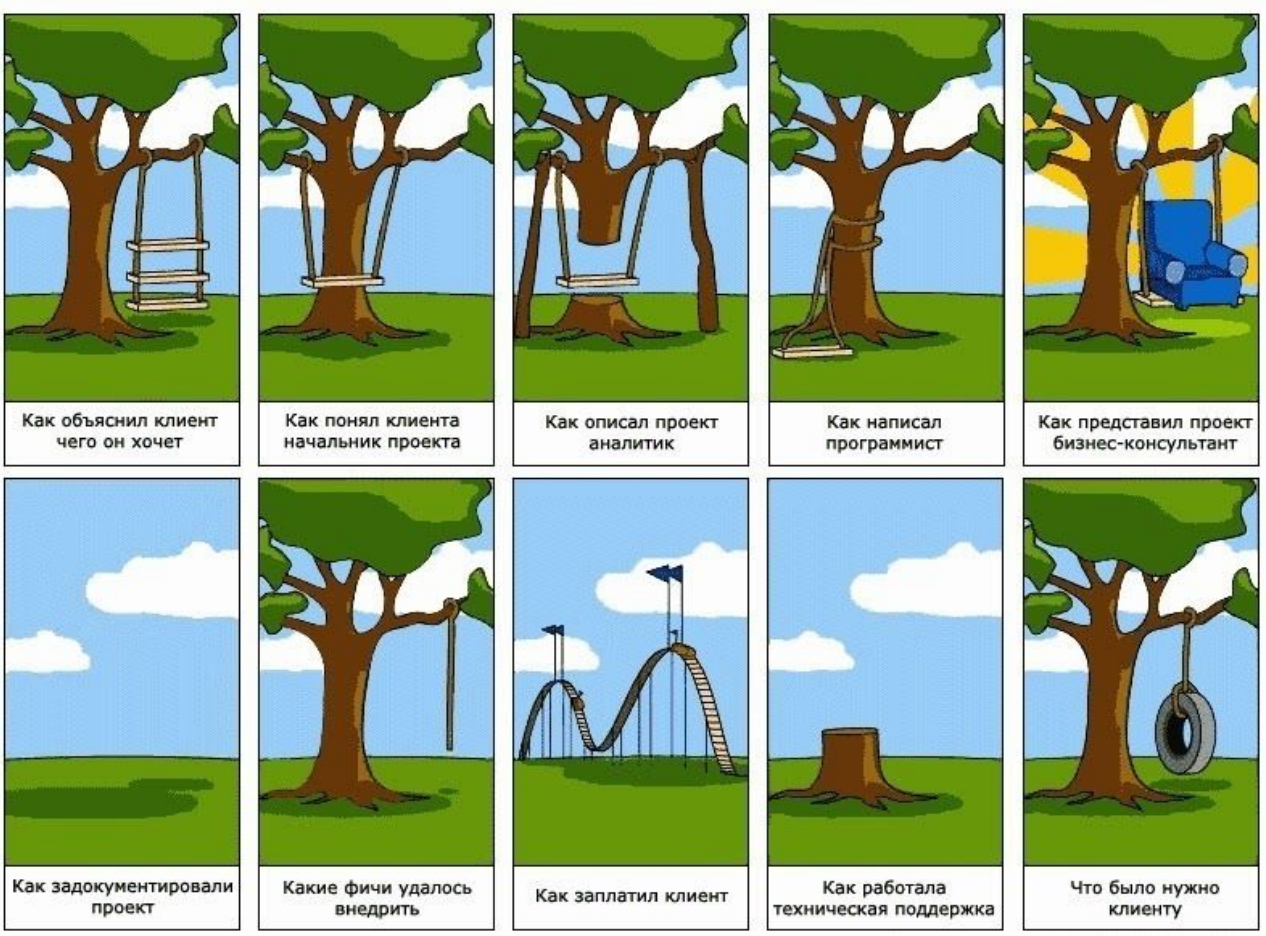
Проектирование на основе предметной области

Проектирование на основе предметной области (Domain Driven Design, DDD) – это объектно-ориентированный подход к построению Архитектуры приложения с использованием понятий и языка предметной области. Этот язык разрабатывается на основе сущностей предметной области и в дальнейшем используется как разработчиками, так и экспертами данной области, не являющимися программистами, но владеющими всей полнотой знаний в этой сфере: например, экспертами со стороны заказчика.

Преимущества проектирования на основе предметной области:

1. Использование языка предметной области значительно упрощает обмен информацией.
2. Гибкая модель предметной области, написанная на языке предметной области, упрощает модификацию и развитие при изменении внешних условий.

3. Объекты модели предметной области прекрасно тестируются благодаря слабой связанности.



В целом DDD направлен на борьбу с широко известным примером (см. рисунок) недопонимания между специалистами узкой направленности и заинтересованными лицами.

Сервисно-ориентированная Архитектура (SOA)

Сервисно-ориентированная Архитектура разделяет функции на отдельные слабо связанные блоки (сервисы), которые могут располагаться как на одном узле сети, так и на разных. Сервисы имеют стандартизованные интерфейсы и могут обмениваться между собой информацией, используя стандартные протоколы SOAP (Simple Object Access Protocol), REST, CORBA или Jini. Слабая связанность сервисов позволяет использовать в разработке отдельных сервисов разные стеки технологий. В рамках одного распределенного приложения могут успешно и взаимозаменяемо сосуществовать сервисы, созданные с использованием разных программных и аппаратных платформ: например, написанные на разных языках (Java, C#, etc.), каждый из сервисов внутри может использовать свою предметно-ориентированную Архитектуру. Распределение функций приложения по слабосвязанным небольшим блокам (сервисам) позволяет очень быстро подключать и модифицировать различные функции системы.

Архитектура - Шина сообщений

Шина служб предприятия (ESB) реализует систему связи между взаимодействующими программными приложениями в сервис-ориентированной Архитектуре (SOA). Основной принцип сервисной шины – концентрация обмена сообщениями между различными системами через единую точку, в которой при необходимости обеспечиваются транзакционный контроль, преобразование данных и сохранность сообщений. Все настройки обработки и передачи сообщений предполагаются также

сконцентрированными в единой точке и формируются в терминах служб. Таким образом, при замене какой-либо информационной системы, подключенной к шине, нет необходимости в перенастройке остальных систем.

Компонентная Архитектура

Компонентная Архитектура [14] описывает подход к проектированию и разработке систем с использованием методов проектирования программного обеспечения. Основное внимание в этом случае уделяется разложению дизайна на отдельные функциональные или логические компоненты, предоставляющие четко определенные интерфейсы, содержащие методы, события и свойства. В данном случае обеспечивается более высокий уровень абстракции, чем при объектно-ориентированной разработке, и не происходит концентрации внимания на таких вопросах, как протоколы связи или общее состояние.

CAP-теорема (теорема Брюера)

Сформулирована Эриком Брюером в 2000 году [18].

Невозможно в распределенной вычислительной системе обеспечить более двух из трех следующих свойств:

- *Согласованность (Consistency) – любая операция чтения получает самую последнюю запись или ошибку.*
- *Доступность (Availability) – любой запрос получает корректный ответ, однако без гарантии, что ответом будет последняя запись.*
- *Устойчивость к разделению (Partition tolerance) – система остается работоспособной, несмотря на потерю произвольного количества сообщений между узлами.*

Следствия

Распределенные системы могут быть CA, CP или AP класса.

Система CA класса обеспечивает согласованность и доступность данных, но жертвует разделением. Типичные представители – СУБД, LDAP.

Система CP класса в каждый момент обеспечивает целостный результат и способна функционировать в условиях распада, но жертвует доступностью и может не выдавать отклик на запрос.

Система AP класса не гарантирует целостность, но остается доступной и устойчивой к распаду. Типичный представитель – DNS, NoSQL.

Мораль: не бывает одновременно быстро, качественно и недорого.
При проектировании не стоит гоняться за тремя зайцами – поймайте хотя бы двух.

Практическое задание

1. Установить IBM Rational Software Architect Designer, Modelio Open Source или другую схожую систему.
    a. https://www.ibm.com/developerworks/downloads/r/architect/index.html
    b. https://www.modelio.org/downloads/download-modelio.html
    c. http://staruml.io/
2. Выбрать Архитектуру домашнего CRM или ERP приложения. Обосновать выбранное решение.

Дополнительные материалы

1. https://sm-dev.edutone.net/Architect/Leanpub.Software.Architecture.for.Developers.May.2014.pdf

Используемая литература

1. http://ru.wikipedia.org/wiki/Архитектура_системы.
2. Мартин Фаулер, Шаблоны корпоративных приложений, М. «Вильямс», 2016.
3. Кристофер Александер, Язык шаблонов … , М. Студия Артемия Лебедева, 2014.
4. http://ru.wikipedia.org/wiki/Сетевая_модель_OSI
5. https://habrahabr.ru/post/276593/
6. http://sergeyteplyakov.blogspot.ru/2013/01/blog-post_29.html
7. https://www.ibm.com/developerworks/ru/library/eeles/
8. http://www.iso-architecture.org/42010/defining-architecture.html
9. https://dev.by/lenta/apalon/nemnogo-pro-arhitekturu
10. https://habrahabr.ru/post/257677/
11. http://dit.isuct.ru/Publish_RUP/core.base_rup/guidances/concepts/software_architecture_4269A354.html
12. Руководство Microsoft по проектированию Архитектуры приложений http://download.microsoft.com/documents/rus/msdn/pы_приложений_полная_книга.pdf
13. Л.Басс и другие, Архитектура программного обеспечения на практике, Питер, 2006.
14. https://habrahabr.ru/post/170623/
15. https://ru.wikipedia.org/wiki/Технический_долг
16. https://en.wikipedia.org/wiki/CAP_theorem
17. http://www.julianbrowne.com/article/viewer/brewers-cap-theorem
18. https://people.eecs.berkeley.edu/~brewer/cs262b-2004/PODC-keynote.pdf
19. Предметно-ориентированное проектирование (DDD). Структуризация сложных программных систем, Эрик Эванс, Вильямс, 2010.