В этом семестре занимаются курсом проектирования программного обеспечения, мы практически пытаемся в рамках этого курса подсобрать ваш практический опыт программирования, как-то его систематизировать, структурировать в плане осознания, что же такое проектирование, что такое программирование.

И в следующем семестре, на четвертом курсе, у нас с вами будет еще предмет основы разработки веб-приложений, где мы, собственно, продолжим с вами, фактически, двигаться в том же направлении, но будем больше заниматься как-то так детальными технологическими и техническими аспектами разработки современных веб-приложений и будем применить уже те архитектурные вопросы, которые мы с вами будем, собственно, обсуждать в рамках текущего курса.

Сегодня у нас с вами вводная лекция. Поговорим про курс, про архитектуру. Рассмотрим организационные моменты. И, собственно, уже начнем работать вплотную с нашим предметом.

Цель нашего курса, в общем-то, довольно такая абстрактная, потому что, на самом деле, если ставить ее серьезно, то она очень была бы оптимистична. Потому что серьезно поставить цель за один семестр на вашем уровне опыта программирования, разобраться вплотную действительно что такое архитектура и научиться проектировать сложные информационные системы было бы, наверное, опрометчиво.

Поэтому цель у нас с вами такая максимально, насколько это возможно, осознать вопрос, что такое архитектура, какой она бывает, зачем о ней думать, зачем хороший разработчик, в общем-то, должен себя всегда мучить вопросами, а как получше спроектировать то или иное программное обеспечение, почему нельзя использовать одни и те же рецепты, проектируя различные программы, с одной стороны, а с другой стороны, почему есть набор стандартных практик и методик, которые настолько универсальны, что они не меняются уже десятилетиями. Какие бы современные технологии, какие бы современные языки программирования ни выходили, есть весьма уже устоявшийся список наработанных опытом поколений разработчиков рекомендаций, размышлений на тему того, о чем надо думать, когда проектируется та или иная программа.

На курсе мы с вами будем рассматривать как вопросы, связанные с тем, как непосредственно проектировать программные компоненты и как из программных

компонентов собирать программную информационную систему, так и более высокоуровневые вещи рассмотрим, связанные с жизненным циклом, связанные с организацией процессов разработки. То есть вещи, которые, возможно, вам будут не совсем так органически понятны, нежели как те вещи, связанные непосредственно с написанием программного кода, с которого мы начнем.

Но, тем не менее, те понятия, которые, как можно раньше, хорошие программный инженер должен впитать и осознать, чтобы уже непосредственно и на практике это самостоятельно применять и понимать, что и как применяется в тех процессах, в тех командах, в тех компаниях, где вы будете работать или кто-то из вас уже работает.

Собственно, по составу курса, у нас с вами курс очень маленький, легколесный, всего 8 лекций, потому что мигающие у нас с вами занятия, и 8 семинаров. На лекциях будем в основном такие вещи, довольно объемно-понятийные разбирать, в рамках наших с вами диалогов. На семинарах более конкретно будем рассматривать либо какие-то вещи, которые не в ваш лекционный курс, какие то понятия и темы, либо повторять уже пройденный ранее материал, либо заниматься обычными практическими вещами. Решаем кейсы, поработаем в командах, пообсуждаем лабораторные работы. Всего у нас будет 9 лабораторных работ. Мне понравился опыт наш предыдущего года, когда мы их выложили с самого начала. Собственно, я думаю, что мы в течение недели с коллегами их согласуем и выложим.

До конца лета, скорее всего, где-то в середине следующей недели уже будет полный список лабораторных работ на курс. Соответственно, вы сможете все прочитать, оценить, спрогнозировать, проектировать свое время, которое вы будете посвящать работе над лабораторными работами.

Фактически все лабораторные работы у нас представляют из себя небольшие шаги, этапы жизненного цикла разработки программы и некоторых программных решений. **В рамках нашего курса вам нужно будет выбрать самостоятельно тему вашего проекта, на базе которого вы будете выполнять лабораторные работы. Это может быть абсолютно любая тема.** Это может быть тема, как совмещенная, например, с курсовым проектом по базе данных, также может быть отвлеченная тема, в общем-то все равно

какая, а требования к темам в лабораторных работах представлены, вы сможете ознакомиться и выбрать тему.

В рамках первого лабораторного работы, мы используем согласование темы проекта, первичный этап анализа. И фактически мы с вами в рамках лабораторных работ будем выполнять стандартные фазы жизненного цикла программного обеспечения. Анализ, проектирование, разработка разных компонентов, вплоть до разработки разных компонентов, которые мы сможем подменять друг относительно друга и прочих различных моментов, которые возникают в жизненных циклах разных программных систем.

Собственно, у нас с вами запланировано по нагрузке каждую неделю время на сдачу лабораторных работ, и я вам настоятельно рекомендую этим активно пользоваться и ваших преподавателей тоже к этому мотивировать, которые будут у вас вести семинары и лабораторные работы. Никто за вами бегать не будет, но, соответственно, никто не будет сидеть с вами часами перед зачетом, чтобы допринять незданную лабораторную работу. Поэтому рекомендация стандартная, люди взрослые. Если начать выполнять лабораторную работу сразу и каждую неделю показывать какой-то срез ваших результатов, то семестр у вас пройдет незаметно, легко, и к зачету вы придете со всеми сданными лабораторными работами и, собственно, получите зачет автомат.

Условия получения зачета, опять-таки, я публикую чуть позже, в течение недели, но у нас бальная система, соответственно, за сданные в сроку лабораторные работы вы получаете баллы. Если лабораторная работа сдана не в срок, то, соответственно, баллов вы получаете меньше. Если сдать все лабораторные работы в срок, плюс еще там можно выполнять дополнительные задания с дополнительными баллами, посещать лекции и семинары, то зачет получится автоматически.

Если вы сдадите не все лабораторные работы, если не будет хватать баллов, то будет возможность сдачи устного зачета. Соответственно, классический устный зачет: по билету три вопроса, чтобы донабрать необходимое количество баллов. Но вообще без лабораторных работ сдать зачет не получится. Минимум 5 или 6 у нас лабораторных работ нужно выполнить так или иначе, чтобы получить зачет.

Также с этого года будет душное нововведение. Будут проверяться конспекты с лекцией и со семинаров.

Я говорю из-за конспекции, вне записи лекции, потому что я не планирую вам надиктовывать никакого материала, я буду вам рассказывать материал. А вы будете тренировать очень важную в жизни навык — конспектирование полезных мыслей. Соответственно, мы с интересом, с коллегами и посмотрим, как вы развили навык конспектирования, какие мысли вы считаете важными для вас, для вашей будущей профессии, какие вы считаете не важными. Соответственно, если конспекции выпусты, то наш разговор с вами будет такой же наполненный, как ваш пустой лист.

На самом деле, я впервые за 10 лет преподавательской практики решил поработать с конспектами студентов, потому что я всегда думал, что студенты взрослые, они сами в состоянии принять решение, записывать им что-то или не записывать. Но, наблюдав за тем, как это все обычно происходит, понял, что здесь нужно ввести некоторое административное воздействие.

Наше административное воздействие будет заключаться в том, что ваши конспекты будут иметь для вас первостепенную ценность, потому что без них не будет доступа к зачёту. Соответственно, люди, которые не посещают лекции принципиально, смогут опуститься к зачёту, только если перепишут чей-нибудь конспект в полном объёме с указанием авторства.

Я думаю, что большинство из вас подумает и решится настоятельно писать отчет в конспекте течения семестра, чтобы не делать себе супер задачу на и без того перегруженную зачетную неделю в рамки переписывания. Так что моя просьба к вам — ходить на лекции, ходить на семинары, работать на лекциях и на семинарах. Это будет нам с вами полезно.

Если у вас будут какие-то вопросы, вы будете реально вовлечены в процесс нашего с вами предмета, вы можете всегда эти вопросы задавать, мы можем с вами их обсуждать. И это, в общем-то, то, зачем я здесь нахожусь. Если бы нашего с вами взаимодействия здесь не было, не предполагается, то необходимости в моем здесь присутствии вообще никакой нет. Я мог бы просто видеоролик записать, поставить, вы бы его скипнули, и мы бы с вами довольно-таки разошлись. Так как я здесь физически нахожусь, давайте это время использовать.

Курс в основном дискуссионный, потому что проектирование программного обеспечения это не какой-то rocket science, это не какая-то дисциплина наукообразная. Здесь нет аксиом, теорем, каких-то строгих выводов. В основном все, о чем мы будем говорить, это некоторые рекомендации сводов методик, не более того.

Ну, поговорим немножко, конечно, о стандартах, но стандарты это лишь вершинка айсберга, потому что одно дело, сочиненные кем-то там согласованные стандарты, другое дело, как это все на практике применять для разработки разных информационных систем. Вот, поэтому в нашем курсе достаточно широкий люфт для обсуждений, в общем-то, и для разговоров, чем я нам с вами предлагаю воспользоваться. Как правило, лекции.

На лекциях, соответственно, если у вас возникает вопрос, вы поднимаете руку, мы с вами это обсуждаем прямо в течение лекции, это будет уместно. Так и в рамках семинаров. Соответственно, на семинарах у нас будет больше возможности что-то обсудить, потому что будет меньше людей, более камерная обстановка.

Буду считать, что вы меня услышали, и что вы увидите в цельных аспектах лекции. Соответственно, за собой оставляю право иногда прогуливаться и заглядывать, у кого есть фразки, а у кого нет. Так, с этим хорошо. Второй момент — это присутствие на лекции. Так как было говорено, что за присутствие на лекции, соответственно, вычисляются баллы.

заставить всех регулярно в 100 количестве присутствовать, я не могу, у меня нет таких рычагов, но замотивировать вас дополнительно путем набирания баллов я могу. Соответственно, каждое занятие, каждую лекцию прошу с первого парта пускать в источник, куда будет записываться по фамилии, кто здесь присутствует.

Никаких санкций за отсутствие нет, кроме отсутствия дополнительных баллов.

С организационными вопросами мы закончили. Начинаем говорить о том, о чем мы будем говорить в течение всего этого курса. И будем возвращаться к этим же вопросам на самом деле и в следующую осень. А мы будем с вами говорить про архитектуру ПО.

Название нашего курса, проектирование программного изменения, оно довольно такое многозначное, как и большинство понятий в IT и в рамках этой многозначности можно было бы выбрать одну из направляющих возможных сопл, о чем в этом курсе можно было бы говорить.

Можно было бы сконцентрироваться на методологии разработки программного обеспечения, идти в вопросы стандартизации, формализации, различных языков моделирования, различных графических нотаций с одной стороны. С другой стороны, можно сфокусироваться на сущностном вопросе, а сущностный вопрос, когда мы говорим про проектирование, это вопрос архитектуры.

Наш курс ориентирован на сущностном вопросе.

Вопросы, связанные с документированием, с моделированием, с графическими нотациями и прочими вещами, они у нас с вами идут в служебном режиме и подкрепляются практическими заданиями в рамках лабораторных работ. Соответственно, в рамках лабораторных работ, даже самый первый, вам нужно будет подготовить некоторый набор документаций. Он не очень большой, но, тем не менее, позволяет понять, как это все обычно происходит.

Итак, начинаем говорить про архитектуру. В общем-то, понятие самой архитектуры, программное обеспечение, как и большинство терминов в IT, давайте не иметь строгого определения. Пришло оно к нам из других сфер человеческий деятельности. Под архитектурой, мы обычно принимаем некоторые вопросы строительства зданий, но этот термин был позаимствован не просто так, потому что на самом деле строительство программного обеспечения во многом похоже на строительство зданий.

Одно из самых известных зданий во всем мире, которая выжила до наших дней. Это колизей в Риме. Ему почти 2000 лет. И за время своей жизни это здание успело потерпеть самые разные способы своего применения. Было амфитеатром, было ареной, было кладбищем, церковью, крепостью, каменномолнией, селитровым заводом, специальным священным местом. И в конце концов, венцом его карьеры стало главным туристическим аттракционом.

И за это время, естественно, само здание претерпевало огромное количество изменений. Это были изменения, как связанные с окружающей средой. Мы можем это видеть, потому что половинки здания, собственно, нет. Потому что

произошло землетрясение, оно было разрушено.

Это была разная враждебно-отбивающаяся среда, когда были нашествия варваров, и они растаскивали его по кусочкам. Это была, собственно, такая полувраждебная среда, когда по возрождению его просто использовали как каменоломню для того, чтобы возводить другие здания, а это, в общем-то, разобрали.

И, тем не менее, само здание функционально сохранилось. Оно, естественно, изменило свою сферу применения. Оно было модифицировано огромное количество раз под разные сферы применения. Достроено, перестроено.

И даже сейчас, когда его используют как туристический аттракцион, соответственно, там тоже были написаны специальные изменения, чтобы туристам было удобно и комфортно находиться. Встроен лифт и прочие разные блага цивилизации, магазин соединений и прочее.

Почему я этот пример привожу? Потому что, на самом деле, когда мы говорим про разработку ПО, то в большинстве случаев, я подчеркиваю в большинстве, потому что все-таки не во всех случаях, мы должны относиться к разработке программного обеспечения так же, как относились строители этого замечательного здания 2000 лет назад и его построили.

Как построить программу таким образом, чтобы вне зависимости от любых внешних воздействий, от изменения окружающей среды, от изменения требований, срабатывание всевозможных рисков, наша программа продолжалась существовать, нам не нужно было ее сносить полностью и строить запад. На этот вопрос разработчики пытаются ответить, когда встают перед задачей проектирования программного обеспечения.

Основная цель проектирования — это спроектировать такую структуру программного обеспечения, которая бы обеспечила длительный жизненный цикл ПО, который будет устойчив к внешнему воздействию, к изменениям требований и будет позволять себя модифицировать.

Конечно, это касается не любых случаев разработки программного обеспечения, потому что если мы говорим про быстрое прототипирование для проверки гипотез, например, в стартапе, то большие вложения в процесс проектирования программного обеспечения, они, как правило, не окупаются, потому что во время

проверки гипотез наша задача очень быстро выполнит некоторый прототип, проверит этот прототип, если прототип не прошел проверки нашей гипотезы, то мы выкидываем этот прототип и разрабатываем что-то за. В этом случае при перепроектировании мы только тратим ресурсы. Поэтому при разработке прототипов стараемся стадию проектирования сделать минимально возможной, определить базовые какие-то точки и, используя стандартные процедуры, стандартные методики, быстро собрать прототип. Неважно, будет ли он отвечать всем требованиям, будет ли он отвечать требованиям эффективности, отвечать различным претензиям, которые мы с вами будем обсуждать. В рамках прототипа это неважно. В рамках прототипа важна скорость проверки методов.

Но если во всех остальных случаях, когда мы разрабатываем не прототипы, когда мы уже запрототипировали, мы проверили гипотезу и решили разрабатывать полноценное программное обеспечение, будь то заказная разработка, будь то заказная коммерческая разработка, государственная разработка, проектная разработка, будь то продуктовая разработка, в общем, любая разработка, программа, продукт, жизненный цикл, которого превышает время его разработки, тогда можно думать о проектировании и думать об архитектуре.

Причем, что по закону подлости, вообще по закону жизни, как бы мы ни думали об архитектуре, как бы мы ни проектировали, у нас все равно, конечно, ничего с вами не получится.

Потому что в реальности при проектировании мы работаем с огромным количеством рисков. Задача проектировщика программного обеспечения — это работа с рисками. Мы должны проанализировать, какие риски у нас могут быть, проанализировать, какие риски более вероятны, чем другие.

В общем-то, отсортировать их в порядке возможной реализуемости и в зависимости от этой наиболее вероятной реализуемости заложить те или иные риски.

Понятно, что если у нас будет неограниченный бюджет, неограниченные ресурсы, неограниченное время, мы могли бы за это неограниченное время полностью проанализировать предметную область, полностью проанализировать сферу применения программного обеспечения, полностью вытащить оттуда весь набор возможных рисков, их полностью заложить в наше программное

обеспечение, чтобы оно было устойчиво к срабатыванию любого из этих рисков.

От того, что заказчик сначала хотел веб-приложение, а потом захотел мобильное, до того, что полностью поменяются все функциональные требования и нам придется полностью переделывать какую-то значимую часть логики. Это тоже все можно заложить и сделать в пределе программное обеспечение максимально гибким.

Но я думаю, что понятно, что это все утопическая история. Чем более гибкое мы делаем программное обеспечение, чем более устойчивым к изменениям, чем более мы туда закладываем срабатывание самых разных рисков, тем оно получается дороже и дольше в разработке и сложнее в том числе в разработке.

Поэтому задача проектирования архитектуры программного обеспечения — это задача тонкого поиска баланса. То есть, с одной стороны, мы можем сделать максимально жесткую структуру, которая будет удовлетворять конкретно сформализованным требования. Это один полюс. Другой полюс — это то, что мы посмотрим на эти требования критически, поймем, что в них недозаложены какие-то важные дополнительные требования, недозаложены риски, их формализуем, заложим все эти риски, сделаем его максимально диким.

Эти два полюса как раз те два полюса, между которыми проектировщик и разработчик программного обеспечения всегда находятся. Потому что если мы сделаем его максимально жестким, то срабатывается любой риск, нам нужно его выкидывать и встать, грубо говоря, заново.

Если мы делаем ПО максимально гибким, то нам нужно вкладывать огромное количество ресурсов в то, чтобы его проектировать и разработать. Так как у нас никогда не бывает бесконечного количества ресурсов, и так как мы не хотим постоянно переписывать ПО, потому что в конце концов это тоже нас подталкивает к тому, что нам нужно бесконечное количество ресурсов, чтобы постоянно переписывать одно и то же. Мы должны найти тот самый баланс, где ПО будет, с одной стороны, и недостаточно жестким, он будет достаточно гибким для того, чтобы его можно было поддерживать и модифицировать в долгосрочный период. А с другой стороны, не настолько гибким, чтобы это у нас, собственно, заняло какое-то бесконечное количество времени.

Собственно, что такое архитектура программного обеспечения? Это сово-

купность важнейших решений об организации программной системы.

То есть это не какая-то конкретная картинка или описание технологий или взаимодействия мозга. Это именно совокупность различных важных решений.

Причем в зависимости от задачи, в зависимости от типа проекта, в зависимости от технологического ограничения условий, у нас такие решения могут быть очень и очень разными и очень по-разному формализованы. Тем не менее, для любой программной системы мы можем такой список важнейших решений формализовать и представить.

Обычно, в совокупности этих решений входит выбор структурных элементов, или иных интерфейсов, с помощью которых они взаимодействуют, а также поведение этих элементов в рамках своего сотрудничества.

Ну, если мы рассматриваем какую-то простейшую процедурную программу, написанную в структурном стиле, например, на языке C, то такими структурными элементами что у нас будет? Функция. Если мы говорим про процедурный язык, то это некоторая программа, некоторая процедура вот таким структурным элементом.

Интерфейс этого структурного элемента — это входные аргументы и выходные значения функций.

Ну а правила взаимодействия это собственно правила декомпозиции и правила вызова функций друг другу. Соответственно мы можем для программы некоторое это все формализовать прописать в том числе наши подходы как мы используем. В объект-ориентированном языке программирования c++, java, c-sharp у нас структурным элементом будет класс и, соответственно, интерфейсы классов, т.е. тот набор методов и свойств, которые они предоставляют во внешнюю среду, мы также можем формализовать и прописать логику поведения этих структурных элементов.

Соответственно, следующее важнейшее решение в этом, как мы соединяем эти элементы структуры во все более крупные подсистемы и системы. Здесь стандартный абсолютно для человеческого мышления подход — **иерархическая декомпозиция**. Мы начинаем говорить с каких-то маленьких частей, они объединяются в более крупные части, более и более крупные, и наконец они вместе соединяются в системе. Но это если мы всю работу проделали и всю

работу провели.

На практике, конечно, человеку сложно, как правило, проводить декомпозицию снизу вверх человек мыслит другими категориями, человек мыслит все-таки от задачи поэтому, имея верхнеуровневую задачу, наша декомпозиция обычно идет сверху вниз: мы имеем большую задачу и постепенно ее разбиваем на некоторые подзадачи, соответственно, в рамках проектирования большой системы это идет постепенное нарезание такого пирога, причем оно идет иерархически. То есть неважно, что мы используем, в рамках парадигмы разработки, какой язык программирования мы используем, у нас так или иначе есть возможность иерархической декомпозиции, что мы сначала что-то большое смотрим, потом часть поменьше, и каждую часть поменьше отдельно раскладываем на составные части.

Такая иерархическая структура нам позволяет на каждом уровне находиться на том уровне визуализации, на том уровне абстракции, который для этого уровня удобен. И, соответственно, не опускаться в детали и не подниматься слишком высоко. Потому что если мы все детали вывернем на один плоский уровень, то есть наверх, на огромную информационную систему, например, социальной сети, мы на уровне всей информационной системы попытаемся формализовать сразу все действия, которые там происходят, но просто в этом утонем. Естественно, проще как-то разбить на куски, систему обмена сообщениями, систему постинговых изображений, постов и прочее, и уже декомпозировать и детализировать именно эти большие блоки.

Наконец, одно из решений — это общий архитектурный стиль, то есть общий подход, в рамках которого разработчики и проектировщики будут эту архитектуру а. строить, б. поддерживать. Здесь очень важный момент, к которому мы будем с вами постоянно обращаться, это то, что архитектура, вообще ПО — это не статическая вещь, она не зафиксирована где-то в единицу времени.

Она является некоторым процессом, некоторой функцией. Если подумать, у нас в жизни не так много вещей, которые могли бы жестко стабилизировать в какую-то единицу времени и иметь просто статические средства. Естественно, что архитектура, которая, по сути дела, отражает некую структуру, некую структурную связанность отличных программных компонентов и превращает

эти программные компоненты в единую программную систему. Это понятие, которое находится в постоянном изменении, оно живое, оно двигается. Нельзя нарисовать диаграмму архитектуры на первом этапе проекта в первый месяц, ее положить на полочку и через два года считать, что эта диаграмма хоть сколько-нибудь совпадает с тем, что у нас получилось.

Поэтому, естественно, что архитектура находится в постоянном изменении. И если у нас фиксировано наше важнейшее решение о том, как эти изменения будут претворять жизнь, то, в общем-то, можно считать, что мы занимаемся архитектурой, мы занимаемся проектированием.

Здесь немножко забегание вперед, можно подумать о разных моделях жизненных циклов о том, что есть, например, водопадная классическая модель когда мы сделали проектирование, дальше садимся кодить, что мы напроектировали но, я думаю, вам это уже на каких-то курсах говорили, вы сами это понимаете, что такой жесткий подход с невозможностью откатиться, исправить ошибки более ранних этапов он неэффективен как и неэффективен обратная сторона медали, когда мы живем в океане, где мы ничего не проектируем, нет никаких этапов, и мы пытаемся просто кодить поток сознания.

Естественно, истина, как всегда, где-то посерединке. Поэтому всегда выделяется этап разработки программной системы, но их никогда не изолируют друг от друга, это было бы невозможно. Соответственно, если вы прошли уже этап базового проектирования, собрали некоторую общую архитектуру системы, то в дальнейшем, продолжая ее разрабатывать, постоянно внося изменения в код, вы должны думать о нашем архитектурном решении. Во-первых, насколько наши изменения все еще попадают под нашу архитектурную концепцию, насколько архитектурная концепция удовлетворяет реальное положение дела в разработке. И, соответственно, принимать все решения и претворять их в жизни, соответственно, с этой архитектурной нашей идеей, архитектурной концепцией, которую мы установили, но при необходимости иметь возможность ее изменить.

Здесь, конечно, всегда тонкий момент, потому что принятие решения об изменении каких-то серьезных, в общем-то, столпов наших процессов, то есть мы установили некоторый архитектурный стиль, а потом вдруг понимаем, что он нам не подходит для этого проекта. Мы ошиблись, мы применили какую-то

стандартную историю, а история у нас вышла не стандартная, или наоборот.

Подходить к принятию решения о том, что нам нужно полностью поменять архитектурный стиль нужно очень осторожно, потому что это несет за собой небольшое количество рисков, издержек. Цена ошибки может быть очень велика, потому что цена ошибки фактически будет равняться всей проделанной работе и остатку работы.

Как правило, такие решения применяются опытными специалистами если мы говорим про средние и большие команды всегда эти позиции архитектора, технических руководителей. Если это малые команды, то, соответственно, стараются принять такое решение за общаду так или иначе, как коллегиально.

Но, тем не менее, мы с вами сейчас попытались дать какое-то очень расплывчатое определение, сложно назвать это определением, скорее описание того, что можно понимать по словам архитектуры программного обеспечения, и какие с этим есть связанные размышления.

Продолжим говорить про размышления. У меня здесь выписан набор цитат разных деятелей этой индустрии, которые по-разному пытаются определить понятие архитектуры.

Гради Бурч пишет такое: Архитектура отражает важные проектные решения по формированию системы, где важность определяется стоимостью изменений.

Принимая архитектурные решения, мы фактически себе закладываем те рельсы, ту структуру, по которой у нас дальше будет двигаться проект, по которой у нас будет идти разработка.

и если мы пропускаем какие-то ошибки и будем их переделывать, то мы можем оценить, какая стоимость этих ошибок, потому что если мы фундаментально ошиблись в чем-то и наша ошибка будет означать переписывание всего, то эта стоимость, соответственно, большая мы приходим к тому, что нам надо выкинуть то, что мы уже сделали, и заново это проработать.

Если у нас какая-то небольшая ошибка, то есть мы неправильно спроектировали какой-то интерфейс, то, в общем-то, эта ошибка достаточно легковесная и легко исправить. Стоимость ее, в общем-то, не такая большая. Вот, собственно, в этом размышлении здесь говорится о том, что наше решение мы можем все

взвесить по степени важности. И в мерилом важности решения будет являться стоимость тех изменений, которые нам нужно будет сделать, если мы ошибемся в этом решении.

Джоссеф Йодер пишет «Если вы думаете, что хорошая архитектура стоит дорого, попробуйте плохую архитектуру».

Ну, это, в общем-то, довольно тоже такая простая и ясная мысль, если ее представить.

Потому что, естественно, хорошая архитектура позволяет достаточно фундаментальный подход к первым этапам разработки некоторого программного проекта. Нам нужно хорошенько заниматься анализом требований, анализом предметов области, все это формализовать, перейти к проектированию, и потратить достаточно продолжительное время, причем на всех квалифицированных специалистов, ведущих разработчиков, архитекторов, технических руководителей, для того, чтобы сформировать концепцию архитектуры и провести проектирование.

И фактически для каких-то проектов это может выглядеть дикостью, потому что мы, например, тратим три месяца проектного времени на то, чтобы сделать какие-то действия, результатов которых будет не прототип, не какой-то кусочек программы, а просто стопка бумаги. Стопка бумаги, где мы опишем бумаги, в общем-то все формализованные требования, и мы опишем архитектуру программной системы.

Это кажется дорогим удовольствием, причем, естественно, мы там все это очень серьезно запроектируем, и это еще и угрожает нам санкциями по разработке, потому что так мы писали туда что попало, получали какие то прототипы, а тут нам нужно закладывать уровни абстракции, писать много инфраструктурного кода, разделять все на уровни, прописывать огромное количество транспортных моделей для передачи данных между различными уровнями.

Это нам предстоит лабораторных работ, и всегда часть студентов возмущается, зачем писать столько ненужного инфраструктурного кода по перекладыванию одного и того же в одно и то же. Хорошая архитектура иногда предполагает достаточно высокие докладные расходы именно на инфраструктурный уровень,

который она обеспечивает гибкость системы. Если нам гибкость системы важна как критерий качества программного продукта, то мы должны, в общем-то, идти на эти инфраструктурные накладные расходы.

И, в общем-то, Джозеф Фьюдор говорит, что если вам кажется, что вот этот вот подход, когда мы долго проектируем большую сложную архитектуру, это очень дорого, то просто попробуйте этого не делать. И посчитайте, во сколько обойдется это. И он совершенно прав. Но прав, конечно, для больших, сложных информационных систем.

Если попытаться посчитать реальную стоимость от проектирования разработки маленьких программных продуктов или каких-то прототипов, то, конечно, как я говорил в начале, длительный этап проектирования будет, наверное, избыточным или, наверное, будет лишним, и, наверное, это будет нецелесообразным.

Но если мы говорим про большие сложные программные системы с циклом разработки год-полтора-два и, соответственно, с численным циклом до пяти лет, то здесь, в общем-то, уже срабатывают все стандартные размышления и рекомендации вокруг архитектуры. Потому что те накладные расходы, которые мы понесем, вкладываясь в процесс анализа и проектирования, они нам окупятся тем, что мы сделаем систему, которую мы сможем просто и дешево в будущем поддерживать, модифицировать и исправлять в ней ошибки.

Если мы этого не сделаем на первом этапе, не сделаем у них накладных расходов, они нас нагонят в будущем, причем гораздо большими темпами, когда нам нужно будет потом долгосрочно поддерживать эту программную систему, выносить туда изменения, модифицировать, исправлять решения и так далее. И чем хуже она будет спроектирована, тем больше затрат понесет это в дальнейшем, вплоть до того, чтобы в конце концов мы неизбежно упрёмся в момент, когда вносить изменения в плохо строительную систему окажется не то, что дорого или сложно, окажется просто невозможно и нам придётся всё выкидывать и писать это заново.

Стандартная, кстати говоря, история у многих разработчиков, особенно у начинающих, когда они видят легоси, старый код, в котором много людей писало, и он уже такой весь некрасивый, как говорится, с душком, они корчат нос и говорят, давайте это все выкинем и напишем заново.

Конечно, они напишут так, что через 3 года другой человек, который посмотрит в их код, скажет, а, ну это другое дело, с этим можно работать.

На самом деле у старых программ всегда огромное количество проблем. У старых программ их не то, что там огромное количество, их бесконечное количество. Потому что это все продукт человеческого творчества. Человек ошибается, человек делает какие-то ставки, принимает решения, управляет рисками, что-то там реализует, и в общем-то никто не гарантирует, что эти решения хорошие или просто некачественные. Конечно нет.

Но, тем не менее, хорошо спроектированная программа отличается от плохо спроектированной программы тем, что плохо спроектированная программа, когда она превращается в легаси, ее становится просто физически невозможно поддерживать. А хорошо спроектированная программа, когда она так же протухает со временем, потому что не надо тешить себе иллюзию, что можно спроектировать так, что программа через 5 лет будет выглядеть бодрячком. Нет, она так не будет выглядеть. Она обрастет костылями, обрастет всякими гнусными отступлениями от архитектурной концепции.

Тем не менее, хорошо спроектированная программа может жить десятилетиями.

И таких кейсов в истории общественной техники очень много, когда хорошо сориентированные программные комплексы реально живут десятилетиями. И за это время у них вносят постоянные изменения, поддерживают, модифицируют, исправляют ошибки и так далее.

Да, с течением времени они все равно становятся не такими красивыми с точки зрения красоты кода и архитектуры. Но тем не менее, если в них была заложена гибкость и готовность к изменениям, то за счет этого они могут жить очень долго, отличаясь от программ, где этот этап не был произведен.

Раз Джонсон пишет: архитектура — это набор верных решений, которые хотелось бы принять на ранних этапах работы над проектом, но которые не более вероятны, чем другие.

в общем-то в этой фразе заключена главная роль проектировщика программного обеспечения, потому что главная проблема, с чем работают архитекторы, почему в общем-то профессия архитектора и программного обеспечения это

как правило такая актёр-градувериесть профессии, то есть нельзя просто пойти курсы на архитектора и стать архитектором, нельзя закончить бакалавриат об архитектуре ПО и стать архитектором.

Да, архитектором можно стать только когда вы поработаете долгое время программистом. В общем-то, наступите на все грабли, которые возможно. Постепенно начнете все больше и больше участвовать в вопросах проектирования. И, наконец, станете самостоятельно принимать решения, основываясь на том, на чем обычно опытные люди принимают решения. На собственном опыте, на интуиции, на размышлениях и на управлении рисками.

Естественно, что мы можем думать, когда какой-то профессионал-архитектор принимает какое-то решение, это решение принято максимально объективно. Но фактически принятие решений архитекторам никогда не может быть чистого, объективного, простого процесса. Всегда будет какая-то червоточинка, где нужно будет принять решение. Мы закладываем этот риск или не закладываем? А может ли вот это произойти? Может ли заказчик потребовать вот эту функцию и вот эту функцию через год? А через пять лет? А через десять? А какие клиентские устройства будут через два года, через пять? Что нам это принесет? Как это поменяется на нашу природу? А что будет с системой связи через год, через два, через три? А что будет с системой связи во всем мире или в нашей конкретной стране? Ну и так далее.

Здесь есть над чем подумать и какие риски применить. В общем-то проблема в том что мы должны принять, в идеале где мы должны принять максимально верное решение, но поймем, насколько оно верное, только когда случится тот риск, на базе которого, который мы накладывали, мы принимали это решение. То есть мы приняли какое-то решение, вот сделать вот этот кусочек программы системы гибким, заложить все, что здесь, нам нужно будет менять требования.

И прошло пять лет, и вот заказчик придумал заменить здесь эту требование. Мы такие «Вау, как хорошо мы придумали, мы заложили этот риск, мы это требование сейчас сюда легко внесем, потому что в этом месте программа гибкая, она здесь позволяет это внести».

А могло случиться так, что эта гибкость не сработала, мы ее добавили просто так, она никогда нам не понадобится в этом месте, соответственно, нам не

нужно будет делать модификации, а то нужно будет делать модификации в тех местах, о которых мы подумали, что нет, это можно закодить жестко, потому что здесь вряд ли что-то изменится, вряд ли, не знаю, поставщики этой библиотеки прекратили свое существование, перестали предоставлять услуги здесь. Мы не заложимся, а это произойдет, и нам можно будет с этим что-то делать.

Как говорит один мой коллега, который по профессии математики занимается основными исследованиями, скептически смотрит на наши инженерные штучки, всякие архитектурные, программистские, и когда смотрит на очередную перепроектированную программу, он говорит так. Ну, конечно, замечательную программу, как всегда сделали, она берется во всех местах, кроме тех, в которых нужно.

Это в общем-то про вопрос, куда мы закладываем гибкости, как мы можем определить, что гибкость нужна именно здесь, а не здесь.

Когда мы принимаем верное решение, любое решение, и мы хотим, чтобы оно было верным, у нас все решения могут быть равнозначными, потому что мы точно не знаем, какой риск сработает, какую проблему нам нужно будет решать завтра. И единственное, на что здесь архитектор, проектировщик может полагаться, это на понимание статистики, теории вероятности, на какой-то свой опыт, дальше идет пространство вероятностей, которое может сработать в ближайшие годы.

В том деле, архитектура — это гипотеза, которую требуется доказать реализацией и оценкой. Это продолжение предыдущей мысли, что только полноценная реализация нашей гипотезы и последующие оценки этой реализации, то есть применения ее на практике, может нам сказать, верные были те решения, которые мы туда вложили или нет.

Поэтому, когда проектируется архитектура сложных программной системы, мы никогда не можем сказать, что мы спроектировали идеальную архитектуру, здесь все хорошо. Такого не бывает никогда. Мы можем сказать, мы сделали неплохую гипотезу, которая статистически вроде покрывает стандартные риски, которые могут случиться с этой программой.

Но то, насколько эта архитектура, насколько эта гипотеза архитектурная была верна для данной задачи, для данного проекта, покажет только время.

Покажет только то, насколько мы сможем качественно реализовать задумку и то, что случится дальше при эксплуатации этой программной системы.

Роберт Мартин поспешай не торопясь.

Здесь речь идет о том, что не нужно нежно при проектировании, при разработке программной системы. Нужно стараться все делать быстро, но тем не менее. В этой скорости разработки, скорости проектирования ни в коем случае не нужно слишком сильно торопиться, потому что ошибки которые мы можем запустить когда мы торопимся они соответственно очень дорого будут стоить, потому что это как с фундаментом дома когда мы проектировали программную систему если мы не додумали и не заложили фундамент у нас дальше все может очень быстро рухнуть.

Такое торопление в принятии решения с точки зрения задач проектирования очень опасно. В какой то момент мы можем принять решение о том, что нам нужно переделать архитектуру по концепции. Значит, нам нужно перепроектировать какой-то модуль или систему целиком.

И если мы к этому придем с таким самонавейным ощущением, что в этот раз мы сделали как надо, потому что предыдущие ребята, которые это сделали, сделали все плохо, а мы-то знаем, как надо сделать и хорошо, то с этой самонадеянностью мы можем закрутить еще более плохое решение, несогласованное, не учитывающее каких-то особенностей, чем было принято раньше. Потому что в этом вообще одна из больших проблем долгосрочной поддержки любых сложных технологических инженерных решений.

Мы с вами как программисты всегда работаем в мире математических моделей даже если это так не представляется, но это некоторые модели, которые пытаются какими-то местами пристроиться к реальной объективной реальности со всеми своими тонкостями, особенностями и спецификами.

И если мы полностью, целиком уходим в идеальный мир, в платоновский мир нашей математической модели, живем в мире программ, то нам могут показаться, что есть какие-то более красивые, более изящные решения.

Но те решения, которые в реальной системе были приняты и обкатаны на практике, они могут быть такими не потому, что тот, кто придумал изначально гипотезу, не додумался до какого-то красивого и изящного решения, а потому

что реальность в этом месте имеет такую особенность, что чтобы эту особенность учесть, нам нужно заложить здесь некоторое отступление от некоторого идеала и сделать какую-то конкретную специфику.

У нас может быть очень красивая архитектура, но нам нужно, чтобы наша программа взаимодействовала с какими-то очень старыми программными системами, с очень старыми железками. Там установлен, не знаю, DOS, Windows XP, на котором нам нужно запускать наш web-интерфейс в интернет-эксплорер 6 и прочее.

И вот эта реальная специфика реального мира, с реальными ограничениями, с реальными особенностями, она всегда, конечно, имеет очень большое влияние на принятие архитектурных решений, и не учитывать это нельзя ни в коем случае.

Программы архитектуры с системой размышлений, с которой мы сейчас с вами познакомились, они в целом-то на самом деле неизменны на всю протяжение развития IT уже более полувека. И это хорошая новость. Хорошая новость в том, что несмотря на очень быстрый технический прогресс. Несмотря на огромные уровни неопределенности и риска, с которыми мы работаем как разработчики информационных систем, тем не менее есть некоторые довольно универсальные рецепты и универсальные соображения, которые не меняются десятилетиями. Ну, просто, наверное, потому что они действительно универсальные, и программирование должно очень сильно измениться, чтобы отойти от этих универсальных идей.

Следующий такой постулат будет то, что низкоуровневые детали, низкоуровневые структуры имеют равное значение. Я про это начал говорить в предыдущей мысли. Что мы можем увлечься высокоуровневой архитектурой, высокоуровневой структурой, есть такая профессия solution архитектора. Это не технический архитектор, который занимается проектированием конкретных программных узлов, а архитектор решений, у которого есть задача проектировать высокоуровневую архитектуру, автоманизацию какого-то скоростного процессора или предприятия. И фактически он рисует пять квадратиков, соединяет их с стрелочками другого цвета.

Можно увлечься высокоуровневым проектированием, при этом не обращая

внимания на специфику деталей. И наоборот, можно очень сильно увлечься деталями, заниматься проектированием низкоуровневых деталей, конкретных реализаций классов, алгоритмов, которые реализуют методы классов. В общем-то, очень глубоко в такую реализацию и потерять внимание сумм.

В общем-то, не тот вариант, естественно, не является хорошим, мы теряем понимание целого, потому что целый у нас рождается именно с учетом и наших верхнеуровневых смыслов, то есть то, от чего мы это делаем, какие функциональные задачи мы решаем, какие риски в плане поддержки, гибкости, модифицированности мы закладываем с одной стороны, а с другой стороны как мы это все наложим на реальность, на реальный состав нашей команды разработчиков, что у нас есть программист на assembly, программист на C, программист на Bitcoin. Давайте разработаем мобильное приложение, такой командой.

И на много-много низкоуровневых детализированных аспектов. Мы все это должны учитывать, чтобы говорить об архитектуре. Наша главная цель, когда мы говорим про программную архитектуру, это сокращение издержек на разработку, на развитие и на поддержку программной системы.

Главная задача это максимальная экономия, максимальное повышение эффективности. Использовать ресурсы для того, чтобы в заложенных ресурсах мы могли добиться наилучшего результата в плане некоторых метрик качества по разработке, поддержке, модифицированности и так далее.

Потому что если у нас такой цели нет, то мы приходим к изначальному выходу: либо мы просто делаем максимально жесткую реализацию конкретных функциональных требований и вообще ничего дополнительного не закладываем, но тогда мы приходим к тому, что нам нужен бесконечный ресурс на то, чтобы полностью переделывать это решение как раз когда меняются требования или окружающая среда, либо мы разрабатываем некий гипотетический универсальный комбайн, то есть какая-то универсальная программа, которая может сделать все. Сейчас такая IT сфера вообще живет с такими холмиками хайпов. Сейчас очередной хайп на базе генеративного искусственного интеллекта появилась такая идея, что мы близки к созданию программ, которые могут всё делать. Потому что лишний интеллектуальный агент или чат бота, в принципе, может

всё. Мы можем дать ему любую задачу, и она ту задачу нам сделает.

И постепенно все разработчики этих генеративных сетей, они туда добавляют новые всякие модулярности. Он уже с картинками работает, с текстом, со звуком, с видео, с интернетом. Туда добавляют все больше и больше данных, которые поддерживаются с агентством, с табличными данными, с документами и прочее, прочее. И вот это, действительно, в рамках вот этого трека движения мы двигаемся к созданию некоторой универсальной программы. То есть, грубо говоря, программа, которая могла бы автоматизировать и решать большое количество разных задач.

Но, естественно, как у любого универсальной программы, мы здесь приходим, что никакое универсальное решение им будет эффективнее, лучше, точнее и качественнее, чем какое-то узкоспециализированное решение.

Потому что универсальное решение всегда будет допускать ошибки, всегда будет допускать неточность, а если мы говорим про такой гигантский черный ящик, который обучен на неизвестных датасетах мировых, соответственно, принимает какие-то решения, ну, оно никаких решений не принимает, это просто статистическая модель, то, естественно, никто никогда не даст гарантию 100 , что будет этот правильный ответ какой-то негативной модели, будет какой-то ответ, с какой-то метрики качества, приближающейся к 80, 90, 95 максимум, но не более того.

Поэтому как ассистент-помощник такая универсальная программа может быть, а как программа, действительно универсальная, автоматизирующая любой бизнес-процесс, пока не видится, чтобы это было в возможно в ближайшие перспективы. Но, тем не менее, все меняется, все развивается.

Обратимся к истокам, с чего началось развитие вычислительной техники. Напомню, что задача первостепенной программной обеспечения была в том, чтобы мы могли гибко менять поведение компьютера. Потому что первые вычислительные машины, вычислительные как таковые, они собирались под конкретную задачу.

То есть нужно решить дифференциальное уравнение. Мы, в общем, собираем такую электротехнику, такие электросхемы, так их соединяем друг с другом, ставим там лампочки, полупроводники и так далее. И получаем какое-то

конкретное, запечатленное в железе, физическое решение нашей задачи.

Сразу меняется постановка задачи. Нам нужно устройство, которое будет обнажать матрицы, чтобы что-то считать. Мы сразу собираем устройство, которое будет считать матрицы.

В общем-то хороший подход, мы можем решить любую задачу, но все максимально эффективно, потому что никаких лишних расчетов, никаких лишних вычислений, движений тока и так далее, у нас будет конкретно при запуске нашего оборудования решаться поставленная конкретная задача.

Естественно, что задач человек продуцирует так много и так часто меняются требования к этим задачам, что под каждую задачу собирает свою собственную железку, это либо долго, либо слишком долго. Поэтому, в конце концов, в развитой вычислительной технике пришло к тому, что мы научились создавать некоторые наборы инструкций, которые поддерживают наши вычислители, и составлять программы из инструкций, которые могут пользоваться этими разными инструкциями.

Таким образом, на базе некоторого универсального вычислителя, комбинируя его разные возможности в разные инструкции, разрабатывает решение самых разных проблем, самых разных задач. И, в общем-то, вот эта изначальная задача программного обеспечения — гибко менять поведение компьютера — мы иногда про нее забываем, а про нее забывает нельзя, потому что то, что программа должна быть гимкой, это заложено в саму суть понятия программы, то есть в программу как сущность для того, чтобы получить ее технику, для того, чтобы гибко менять поведение.

Поэтому жестко написанная программа, которую мы написали один раз, зафиксировали, сохранили ее и вот она работает но это нонсенс для программы, программа не для этого создавала мы можем тогда такую жестко записанную программу просто прошить физически в микросхему, вот она там будет работать. Такое бывает тоже бывает редко чаще всего все программы с которыми мы работаем они конечно же должны быть гибкими.

И вот задача проектирования, задача размышления об архитектуре, это вопросы поиска вот этой самой степени гибкости нашей программы. К сожалению, или к счастью, абсолютно гибкую программу мы написать не можем.

Вопрос. Что с точки зрения разработки программы-системы более важно? Правильная работа системы или простота её изменения?

Если она неправильно работает, то ее можно легко изменить правильный ответ здесь именно, как и кажется, правильная работа должна быть более ценной но на самом деле для большой сложности у лицом этой системы самым ценным качеством является простота изменения этой системы потому что если изменять систему легко и она при этом неправильно работает совершенно верно мой коллега сказал то мы это можем легко исправить.

Мы внесли изменения, и система теперь работает правильно. Завтра она опять будет неправильно работать, потому что завтра другой день. У нас, например, какой-то алгоритм был зависим от нашему календаря, и у нас все поломалось. Мы будем опять его исправлять. Мы никогда не можем сказать, что наша программа работает абсолютно правильно, потому что максимум, что мы можем сделать, это прогнать какое-то огромное количество тестов, которые никогда не будут давать 100 покрытия. Мы можем 100 покрыть ветки код, но не можем 100 покрыть все возможные вторные данные. Поэтому у нас всегда есть какая-то вероятность. Вот у нас есть 100 покрытие тестами, например, по коду. Не 100 покрытие по данным, потому что это невозможно. И мы считаем, что в данный момент времени у нас программа на данном наборе тестов показала правильный результат.

Будет ли это считаться, что она правильно работает? Нет, это вообще ни о чем не говорит это нам дает какую-то призрачную уверенность, что мы можем сказать пользователю да, программа работает, я проверял, точно работает пользователь спускает программу, нажимает на кнопку, программа падает потому что на его компьютере не установлен какой-то программный компонент, который использует ваша программа.

Кто об этом должен был подумать? Ну явно не пользователь. По крайней мере мы заложили, что у него какой-то древний компьютер на котором 10 тысяч раз было переустановлена ОС, там на пиратской версии отсутствует какая-то часть стандартных компонентов и так далее, мы не можем заложиться на все лисы, которые могут сработать мы не можем на них предусмотреть все, поэтому говорить, что программа правильно работает мы не можем никогда у нас никогда

не бывает такой возможности.

Мы можем сказать, что мы ее проверили в каком-то наговоре тестов, и в данный момент времени тесты зелененькие. Но будут ли они зелененькие через 5 минут, даже этого мы не можем сказать. Поэтому вообще правильность работы — это очень условный параметр. Мы должны, конечно, стремиться максимально себя здесь утешать, ну то есть какими-то вот такими вещами, типа тестирования. Это очень важный момент.

По тестированию мы с вами будем говорить тоже в рамках и нашего курса по проектированию, и у вас будет отдельный курс в следующем семестре, осенью, по тестированию программного обеспечения, где вы будете подробно разбираться с этим вопросом. Тестирование очень важно, но оно носит такую же вероятностную природу, как и проектирование. Мы не можем сказать наверняка, что это вот так. В программе сколько-то тестов, мы не можем на основе этого сделать вывод, что программа работает.

Поэтому, если мы выбираем между правильностью работы и красотой изменений, конечно, нужно ставить красоту изменений, потому что такая ставка нам позволит более быстро вносить изменения в нашу программу, и всякий раз, когда мы будем обнаруживать, причем мы можем это случайно обнаружить, что наша программа работает не совсем правильно, быстро это исправляем.

Если у нас простоты изменений в программе не будет, то любое изменение, которое нам потребуется, оно потребуется в любом случае. В любом случае мы столкнулись с ситуацией, когда программа не будет работать на каком-то кейсе. Любое изменение будет для нас очень дорогим и сложным.

А обнаруживая, что что-то неправильно работает, программа может годами работать. И таких кейсов огромное количество. Программа реально работает с продакшнами. Годами ее пользуются конкретные пользователи. Они пользуются 95 функционала этой программы. Они все прощелкивают, все прекрасно работает.

Но вдруг приходит какой-то новый пользователь, который эти же самые действия, которые все делали в протяжении пяти лет, сделал в обратном порядке. Ну, просто потому, что ему никто не сказал, что все делают обычно так. Он сделал по-другому. И это тоже было бы правильно. Но наша программа под этим не была рассчитана, и она сломалась. Могли ли мы это все раньше там

выловить? Ну, могли бы, если бы мы серьезно вложились опять-таки ресурсами в тестирование и, в общем-то, прогнали нашу программу всячески.

Можем ли мы в рамках тестирования действительно продумать все сценарии? мы гипотетически можем, на практике это очень слабо реализуемо.

От чего мы здесь страхуемся? Какие у нас риски требуют простоты изменения? В первую очередь, это изменение требований. Здесь сложно говорить в первую и вторую очередь. Все эти понятия, наверное, в разной степени значимы для разных проектов, но в равной степени важны для рассмотрения проблематики.

Это изменение требований, когда к нашему программу не встречу меняются требования. И это может быть очень слабо формализованный процесс. Мы очень слабо, как правило, как разработчики, управляем этим процессом. Потому что здесь могут быть очень разные краевые ситуации. Например, стартап, который двигается в agile схеме, и на каждую итерацию команда разработчиков, в голове с лидером, который придумывает эту инновацию, генерирует новый поток изменений.

Требования могут меняться буквально каждую итерацию. Каждый месяц у нас будут принципиально новые требования. Потому что, не знаю, в этом месяце стартап представлен на какой-то конференции, и по обратной связи помним, что надо все переделывать.

Другой раз пришли к бизнес-агенту, он сказал нет, делайте все обратно. И опять меняются требования. И в рамках такой гибкой разработки требования вообще текут. Причем очень гибкая разработка появилась, мы с вами про это тоже будем говорить. Потому что стало понятно, что фиксировать в современном мире какой-то жесткий отрезок требований практически невозможно. Они очень динамически меняются. Поэтому процесс изменения требований вообще континуальный.

Даже если мы берем ситуацию совершенно обратную. Например, проектная заказная разработка какой-то сложной системы по государственному заказу. Там долгие всегда идут согласования, пишется огромное техническое задание.

Пару лет пишется, согласовывается, запускается в работу, два года система разрабатывается, потом внедряется в опытную эксплуатацию, потом внедряется...

От момента согласования, придумки, что такая система нужна, до момента того, что она дойдет до реальных пользователей, может дойти до 5 лет. Это абсолютно нормально с точки зрения больших, сложных, государственных, информационных систем.

И вот когда она доходит до конкретных пользователей, вдруг оказывается, что когда 5 лет назад собирали требования, конкретных пользователей-то никто не спросил. Потому что начальник подумал, ну я же знаю, чем они там занимаются, ну я вот сделаю так. И на всех этапах это все требования все были согласованы, потому что согласованы другие начальники.

Разработчики это все разработали, в общем-то, все было внедрено. Система внедрена, система работает, в нее потрачены деньги. А получается, что, например, она на 70 удовлетворяет реальных требований конкретных пользователей, которые работают на местах, а на 30 совсем не удовлетворяет.

Ну и дальше идет, в общем-то, такой вот процесс уже притирки системы к реальному использованию. И даже в рамках таких жестких, довольно с точки зрения формальности структур, все равно вступает дело в цикл разработки, потому что иначе ничего не будет ехать. То есть постепенно в программу влетают какие-то неформализованные, незаложенные требования, которые нужно так или иначе туда отпилить, просто потому что без них никто ими пользоваться не будет.

Изменение окружения и условий эксплуатации. Если изменение требований, оно поступает от конечных пользователей, от целевой аудитории, от заказчика, от разных факторов, которые принимают участие в процессе. Это так называются ордерные процессы различными.

То изменение окружения и условий эксплуатации, мы здесь говорим не об авторах, а о некоторых явлениях, о некоторых событиях, которые совершенно, как казалось бы, не относятся к нашему проекту, но тем не менее напрямую его затрагивают. В нашей текущей состоянии обстановки, на протяжении последних двух лет, как раз все разработчики, так или иначе, столкнулись с изменениями.

Условия кружили, потому что ушли какие-то лендеры, ушли разработчики из нашей страны, и пришлось довольно оперативно большому количеству команд адаптироваться к этим изменениям. Ну или кто-то спрогнозировал, что какие-то

такие процессы произойдут. Кто-то мог, но большинство, естественно, такого развития событий не прогнозировало, и пришлось попотеть буквально всем, чтобы подстроиться, пристроиться к новой ситуации и к новому изменению технологического ландшафта в стране в целом. Но все решаемо, и эти решаемые проблемы, тем более наш отечественный бизнес умеет приспосабливаться вообще ко всему, поэтому в общем-то все будет хорошо. Все уже нормально.

Когда мы проектируем систему, мы должны закладывать, что реально может измениться и фактически многие вещи, про которые мы с вами будем говорить связанные с профилированием и закладыванием гибкости, разбиваются на закладывание гибкости в этих двух поинтах: на изменение тегов и нить, это закладывание гибкости в части, которая реализует функциональные задачи

Но есть тонкая грань, потому что если требования, которые влияют на функциональность системы, изменяются кардинально, то есть полностью, то с точки зрения современных подходов к разработке ПО считается, что все-таки это у нас новое ПО появляется. То есть, если у нас требования меняются так, что от старой функции программы не остается ничего, у нас появляются новые функции, то, наверное, мы будем решать новые программы.

Ну вот все, что касается изменений окружающей среды, условий эксплуатации и технических различных аспектов. Например, мы планировали, что на устройствах конечных пользователей будет стоять Windows, а сейчас у всех будет стоять Linux.

Или вы планировали, что у нас будет, соответственно, приложение под айфоны, а сейчас мы понимаем, что у целевой аудитории будет только Android. И прочие-прочие вещи. Вот как раз про технологические риски. Задача проектирования входит в максимальные закладки гибкости. Потому что есть вещи, от которых мы можем очень легко подстраховаться, и от которых всегда будем подстраховаться.

Но мы про это будем с вами конкретно говорить на примере даже наших лабораторных работ.

И наконец, это исправление ошибок. Ошибки могут быть разные мы их можем вносить уже в этапе поддержки, в этапе разработки где-то не дотестировали, где-то ошибка случайная, где-то ошибка потому что мы что-то не додумали,

где-то ошибка потому что мы неправильно спроектировали.

Фактически у разных видов ошибок мы имеем разную стоимость ее исправления, разную ценность этой ошибки, важность функционирования системы, ну и соответственно закладка гибкости вот эту непосредственную ошибку, она имеет разную стоимость. Работа с этой стоимостью тоже входит в задачи проектирования архитектуры.

В общем-то, продолжаем мысли про то, а как вообще такую архитектуру спроектировать. Здесь нужно еще понимать, что в любом сложном большом проекте, где есть от команды разработчиков и команды разработчиков, в том числе входят технические руководители, архитекторы и все разработчики разных коллег, есть некоторая команда менеджмента, будь то внутренний менеджмент проекта, будь то внешний менеджмент заказчика, будь то продукт менеджмент, который пытается соединить идеи продукта с целевой аудиторией, некоторое управляющее воздействие.

И здесь нужно понимать, что в этих двух групп, во-первых, нужно разделять эти две группы, хотя они делают действительный проект, но нужно разделять эти две группы по фокусу, о чем они должны концентрироваться.

Потому что менеджмент должен следить как раз за самым важным, за бизнес-value, это функциональность и затраты на эту функциональность.

То есть с точки зрения бизнеса принимается решение, что нужна такая-то функциональность в продукте и оценивается, что это будет по деньгам. А если это будет недорого, а вам не нравится, если это будет слишком дорого, то может быть эта функция не так важна. Это в общем-то задача принятия решения уровня некоторого абстрактного руководства.

А задача фокусировки командной разработки это собственно задача проектирование архитектуры и работы с технологиями. И если команда разработки не будет думать об архитектуре, то никто о ней, об этой архитектуре думать не будет.

Здесь поинт включается в том, что команда менеджмента любого типа, я ее перечислил в разные вариации, как можно представить эту команду менеджмента. Абстрактная. Она никогда не будет думать об архитектуре, потому что для этой команды архитектура сама по себе имеет никакой ценности.

Если разработчики приходят и говорят, вот мы, значит, в течение трех месяцев будем заниматься архитектурой. Мы ее будем проектировать, а потом будем разрабатывать инфраструктурные компоненты, которые будут нам делать каркас. То менеджмент спрашивает, так, а какие функции мы получим за эти три месяца? Разработчик говорит, нет, никаких. Типа, там просто будут интерфейсы голые и все. В смысле, интерфейсы не пользовательские, а интерфейсы в плане интерфейсов для программных компонентов.

Естественно, мэр такой ответ не устроит, он это не поймет. Сражаться за архитектуру и наказывать необходимость инвестировать в архитектуру всегда должна техническая команда разработчиков. Будут архитекторы, будут рядовые разработчики, будут ведущие разработчики.

Вообще-то все, кто понимает, зачем нужна здесь архитектура, они в проекте должны отвечать за то, чтобы пробивать затраты на архитектуру. Потому что с точки зрения менеджмента, который реально занимается бизнесовыми задачами, архитектура не нужна. Но она им понадобится, когда им нужно будет внедрять туда из людей главных проектов. Потому что это не их задача, это не их уровень ответственности. Их уровень ответственности управлять функциональными требованиями и ресурсами.

Поэтому закладывать архитектуру нужно с технической командой, нужно с умом, нужно об этом всегда думать. И об этом никто не подумает за техническую команду.

И понятно, что есть постоянная битва, постоянный торг. И даже если вдруг в какой-то момент приходит понимание, что, например, в каких-то местах у нас архитектура устарела, или так получилось, что в спешке для реализации функциональных требований мы отошли от изначального вектора архитектурного проектирования. Мы ушли куда-то далеко. И назрел рефактор для того, чтобы нашу программу, систему подбить в единое целое, вернуть в изначальную концепцию.

Но продавить финансирование рефакторинга с точки зрения общения между технической и руководящей командой может быть очень сложно. Это можно только на примере конкретных функциональных требований. Обстрактный рефакторинг никому не нужен. На примере конкретных функциональных

требований, для которых рефакторинг жизненно необходим. Причем ее надо умело размазать по набору функциональных требований и продать эту идею в комплекте.

По опыту бывает так, что даже когда это назревает, у вас, по сути дела, не рефакторинг, а переписывание всей большой, сложной, гигантской системы, что даже само по себе задачу переписывания абстрактно поставить невозможно. Никто никогда не согласует на нее с точки зрения, например, журналистской разработки почти никогда денег не выделит. Будет это государство, будут это коммерческие заказчики и так далее.

Коммерческий заказчик даже еще тем более, потому что он более активно считает деньги вложенные.

Поэтому, как правило, любое переписывание, любое рефакторинг всегда проводит с собой какие-то новые функции. Если есть функции, они могут оправдывать заботу об архитектуре. Это такое немножко философское размышление, политическое некоторое, но о нем тоже нужно думать, потому что архитектура это не что-то, что обязательно есть. Для того, чтобы архитектура была, техническая команда должна об этом заботиться и должна постоянно отстаивать интересы архитектуры.

Пустили интересно, можно очень быстро и очень просто. Например, гибкие циклы разработки, и прибегает продукт менеджер и говорит, нет, срочно нам нужно внедрить эту функцию, и если мы не на ней, то у нас все пользователи разбегутся. А эта функция требует такого гигантского куска кода, чтобы ее грамотно разделить в нашу архитектуру.

У нас времени этого нет, мы его впиливаем на костылях, так и вешаем если оно так и останется висеть, то все сломается, через какое-то время мы не сможем поддерживать систему потому что эта штука просто приклеена на подорожники.

Но с точки зрения продукт менеджера все хорошо. Он попросил фичу, фича есть. Зачем мне возвращаться? Зачем вообще возвращаться, ее перепроектировать и переписывать? Это вот такой вот несколько политический аспект, но он так или иначе очень тесно связан с вопросами архитектуры.

Говоря об архитектуре, уместно начать, хотя мы когда проектируем архитектуру, мы всегда начинаем сверху, вниз.

Но изучая разные аспекты архитектурного планирования, нам с вами будет проще начать снизу вверх, потому что нам будет понятнее говорить о тех вещах, с которыми мы непосредственно руками работаем и постепенно повышаем уровень абстракции.

Здесь нельзя не обойти разные процессы программирования, потому что так или иначе, несмотря на то, что сейчас мы проезжаем век мультипарадигмальных языков, которые впитывают все хорошее и все плохое повсюду и подряд, тем не менее, есть три устоявшихся парадигмы, которые так или иначе языками программирования применяются. Это, собственно, структурное программирование, фактически, строительный блок, структурный элемент структурного программирования, это как мы с вами вначале обсудили, это функция, либо процедура, либо программа, в общем-то, некоторое действие, некоторое действие, которое абстрактировано от данных, в которое мы можем соединять иерархически некоторую древоподобную структуру.

В объектно-ориентированном программировании у нас, на самом деле, строительным типичным является класс, либо из класса объект, и мы здесь уже переходим к более сложному понятию, а сложное оно до того, что у нас с вами появляется не просто какое-то абстрактно висящее действие, которое связано с данными лишь, скажем так, всего едино, да, по адресным данным, данных параметров, а это непосредственно упаковка действий и данных вместе, что рождает нам такую сущность как план, на базе которого уже можно строить довольно сложные программные архитектуры, потому что с точки зрения архитектуры структурные программы очень простые они всегда иерархически декомпозируют все, там сам архитектурный подход может быть довольно линейным, примитивным и все, за чем мы должны следить, это фактически за тем, насколько мы управляем структурным программированием, насколько у нас хороша декомпозиция.

Думать, конечно, про зависимость тоже, но про это мы с вами поговорим. С точки зрения ООП, здесь уже появляется гораздо большее количество измерений, в которых мы должны принимать решения, гораздо больше возможностей ошибиться, гораздо больше, собственно, степени свободы, в которой мы принимаем решения.

Тем интереснее, тем более гибкие программные структуры мы можем устраивать с помощью ООП. И да, ООП наконец-то стала ведущей проектами разработки.

Ну и, наконец, функциональное программирование весь спектр различных языков программирования, которые так или иначе обыгрывают свой собственный декларативный подход, аплодический подход, функциональный подход, которые на базе некоторой глобальной идеи пытаются представить свой уровень декомпозиции функциональное программирование и свой уровень декомпозиции это математические функции, но соответственно со всеми вытекающими моментами связанными с тем, что например у нас там отсутствуют переменные как таковые и поэтому само себе программирование имеет другую специфику тем не менее сама создача декомпозиции она во многом схожа с декомпозицией в структурном, либо в объект-ориентированном программировании.