##Управление жизненым циклом разработки

1. Управление жизеным циклом проекта. Модели жизненого цикла.

Жизненный цикл проекта.

Каждый проект от возникновения идеи до полного своего завершения проходит ряд последовательных ступеней своего развития. Полная совокупность ступеней развития образует жизненный цикл проекта. **Жизненный цикл** принято разделять на фазы, фазы - на стадии, стадии - на этапы. Проект проходит четыре фазы развития: концепция, разработка, реализация, завершение. Каждый проект имеет окружение. В качестве ближнего окружения большинства проектов выступает «родительская организация», где данный проект является составной частью «жизненного цикла деловой активности организации». Проект может быть тесно связан с выпуском новой продукции или услуг и осуществлением необходимых для этих целей изменений. Таким образом, проект связан еще и с «жизненным циклом продукта».

Рассмотрим более подробно содержание отдельных фаз проекта.

Начальная фаза проекта (разработка концепции) требует выполнения

следующих работ: сбор исходных данных и анализ существующего состояния (предварительное обследование); выявление потребности в изменениях (проекте); определение результата (цели, задачи, результаты; основные требования, ограничительные условия, критерии; уровень риска; окружение проекта, потенциальные участники; требуемое время, ресурсы, средства и др.); определение и сравнительная оценка альтернатив; представление предложений, их апробация и экспертиза; утверждение концепции и получение одобрения для следующей фазы.

Фаза разработки проекта требует выполнения следующих работ: назначение руководителя проекта и формирование команды проекта, в первую очередь ключевых членов команды; установление деловых контактов и изучение целей, мотивации и требований заказчика и владельцев проекта, других ключевых участников; развитие концепции и разработка основного содержания проекта (конечные результаты и продукты; стандарты качества; структура проекта; основные работы; требуемые ресурсы); структурное планирование (декомпозиция проекта; календарные планы и укрупненные графики работ и обеспечения; смета и бюджет проекта; потребность в ресурсах; процедуры УП и техника контроля; определение и распределение рисков.

Фаза реализации проекта требует выполнения следующих работ: организация и проведение торгов, заключение контрактов; полный ввод в действие разработанной системы УП; организация выполнения работ; ввод в действие средств и способов коммуникации и связи участников проекта; ввод в действие системы мотивации и стимулирования команды проекта; детальное проектирование и технические спецификации; оперативное планирование работ; установление системы информационного контроля за ходом работ; организация и управление материально-техническим обеспечением работ, в т.ч. запасами, покупками, поставками; выполнение работ, предусмотренных проектом (в т.ч. производство строительно-монтажных и пуско-наладочных работ); руководство, координация работ, согласование темпов, мониторинг прогресса, прогноз состояния, оперативный контроль и регулирование основных показателей проекта (ход работ, их темпы; качество работ и проекта; продолжительность и сроки; стоимость и другие показатели); решение возникающих проблем и задач.

Фаза завершения проекта требует выполнения следующих работ:

планирование процесса завершения проекта; эксплуатационные испытания

окончательного продукта проекта; подготовка кадров для эксплуатации

создаваемого проекта; подготовка документации, сдача объекта заказчику и ввод в эксплуатацию; оценка результатов проекта и подведение итогов; подготовка итоговых документов; закрытие работ проекта; разрешение конфликтных ситуаций; реализация оставшихся ресурсов; накопление фактических и опытных данных для последующих проектов; расформирование команды проекта.

Большое значение имеют следующие дополнительные элементы проекта:

начальные условия, ограничения и требования к проекту (характеризуют

предысторию и существующее состояние системы; существующее состояние

окружения системы; требования к результатам проекта и способам их достижения; ограничения на цели и результаты проекта, определяющие количественные характеристики и допущенные границы); область допустимых решений проекта; документация проекта; виды обеспечения проекта (функциональное, информационное, математическое, программное, техническое, организационное, правовое, методическое, прочие виды обеспечения); методы и техника управления проектами.

Моделирование жизненного цикла проекта по принципу «водопада»

При моделировании по принципу «водопада» работа над проектом движется линейно через ряд фаз, таких как:

- анализ требований (исследование среды);
- проектирование;
- разработка и реализация подпроектов;
- проверка подпроектов;
- проверка проекта в целом.

Недостатками такого подхода являются накопление возможных на ранних этапах ошибок к моменту окончания проекта и, как следствие, возрастание риска провала проекта, увеличение стоимости проекта.

Моделирование жизненного цикла проекта по итеративной модели

Итеративный подход (англ. *iteration* — повторение) — выполнение работ параллельно с непрерывным анализом полученных результатов и корректировкой предыдущих этапов работы. Проект при этом подходе в каждой фазе развития проходит повторяющийся цикл: Планирование — Реализация — Проверка — Оценка.

Преимущества итеративного подхода:

- снижение воздействия серьезных рисков на ранних стадиях проекта, что ведет к минимизации затрат на их устранение;
- организация эффективной обратной связи проектной команды с потребителем (а также заказчиками, стейкхолдерами) и создание продукта, реально отвечающего его потребностям;
- акцент усилий на наиболее важные и критичные направления проекта;

- непрерывное итеративное тестирование, позволяющее оценить успешность всего проекта в целом;
- раннее обнаружение конфликтов между требованиями, моделями и реализацией проекта;
- более равномерная загрузка участников проекта;
- эффективное использование накопленного опыта;
- реальная оценка текущего состояния проекта и, как следствие, большая уверенность заказчиков и непосредственных участников в его успешном завершении.

Моделирование жизненного цикла проекта по спиральной модели

т.н. модель Боэма (Барри Боэм). Рассматривается зависимость эффективности проекта от его стоимости с течением времени. На каждом витке спирали выполняется создание очередной версии продукта, уточняются требования проекта, определяется его качество и планируются работы следующего витка.

Моделирование жизненного цикла проекта инкрементным методом

Инкрементное построение: разбиение большого объема проектно-конструкторских работ на последовательность более малых составляющих частей.

В экономической теории и практике наиболее распространен вид моделирования, называемый **семиотическим**. При этом главная цель семиозиса - функционирование системы выразительных средств, есть учет ненаблюдаемых свойств объекта, что практически совпадает с функцией моделирования. Семиотичные модели отличает изучение смысловой структуры объекта, его содержательной насыщенности и логики. Наибольшее значение в бизнес-системах имеют семиотичные знаковые модели. Используя широкую совокупность знаковых преобразований: схемы, графики, чертежи, формулы, таблицы, наборы символов, а также законы и закономерности, которыми можно оперировать с выбранными знаковыми элементами, они позволяют исследовать многочисленные процессы и явления, свойственные рыночной среде и развитию бизнес-системы. Среди различных видов знакового моделирования наибольшее употребление находит математическое моделирование, при котором исследование объекта осуществляется посредством модели, построенной с помощью математических методов. Применительно к экономическим задачам оно характеризуется как - экономико-математическое.

2. Гибкие методологии разработки. SCRUM. Kanban.

Общее

"Гибкие" (agile) методы разработки ПО появились как альтернатива формальным и "тяжеловесным" методологиям, наподобие CMM и RUP. Талантливые программисты не желают превращения разработки ПО в рутину, хотят иметь максимум свобод и обещают взамен высокую эффективность. С другой стороны, практика показывает, что "тяжеловесные" методологии в значительном количестве случаев неэффективны. Основными положениями гибких методов, закрепленных в Agile Manifesto в 2007 году являются следующее[1]:

- индивидуалы и взаимодействие вместо процессов и программных средств;
- работающее ПО вместо сложной документации;
- взаимодействие с заказчиком вместо жестких контрактов;

- реакция на изменения вместо следования плану.

Фактически, гибкие методологии говорят о небольших, самоорганизующихся командах, состоящих из высококвалифицированных и энергичных людей, ориентированных на бизнес, то есть, например, разрабатывающих свой собственный продукт для выпуска его на рынок. У этого подхода есть, очевидно, свои плюсы и свои минусы.

Scrum.

Общее описание. Метод Scrum позволяет гибко разрабатывать проекты небольшими командами (7 человек плюс/минус 2) в ситуации изменяющихся требований. При этом процесс разработки итеративен и предоставляет большую свободу команде. Кроме того, метод очень прост – легко изучается и применяется на практике. Его схема изображена на рис. 11.1.

Рис. 11.1.

Вначале создаются требования ко всему продукту. Потом из них выбираются самые актуальные и создается план на следующую итерацию. В течение итерации планы не меняются (этим достигается относительная стабильность разработки), а сама итерация длится 2-4 недели. Она заканчивается созданием работоспособной версии продукта (рабочий продукт), которую можно предъявить заказчику, запустить и подемонстрировать, пусть и с минимальными функциональными возможностями. После этого результаты обсуждаются и требования к продукту корректируются. Это удобно делать, имея после каждой итерации продукт, который уже можно как-то использовать, показывать и обсуждать. Далее происходит планирование новой итерации и все повторяется.

Внутри итерации проектом полностью занимается команда. Она является плоской, никаких ролей Scrum не определяет. Синхронизация с менеджментом и заказчиком происходит после окончания итерации. Итерация может быть прервана лишь в особых случаях.

Роли. В Scrum есть всего три вида ролей.

Владелец продукта (Product Owner) – это менеджер проекта, который представляет в проекте интересы заказчика. В его обязанности входит разработка начальных требований к продукту (Product Backlog), своевременное их изменение, назначение приоритетов, дат поставки и пр. Важно, что он совершенно не участвует в выполнении самой итерации.

Scrum-мастера (Scrum Master) обеспечивает максимальную работоспособность и продуктивную работу команды – как выполнение Scrum-процесса, так и решение хозяйственных и административных задач. В частности, его задачей является ограждение команды от всех воздействий извне во время итерации.

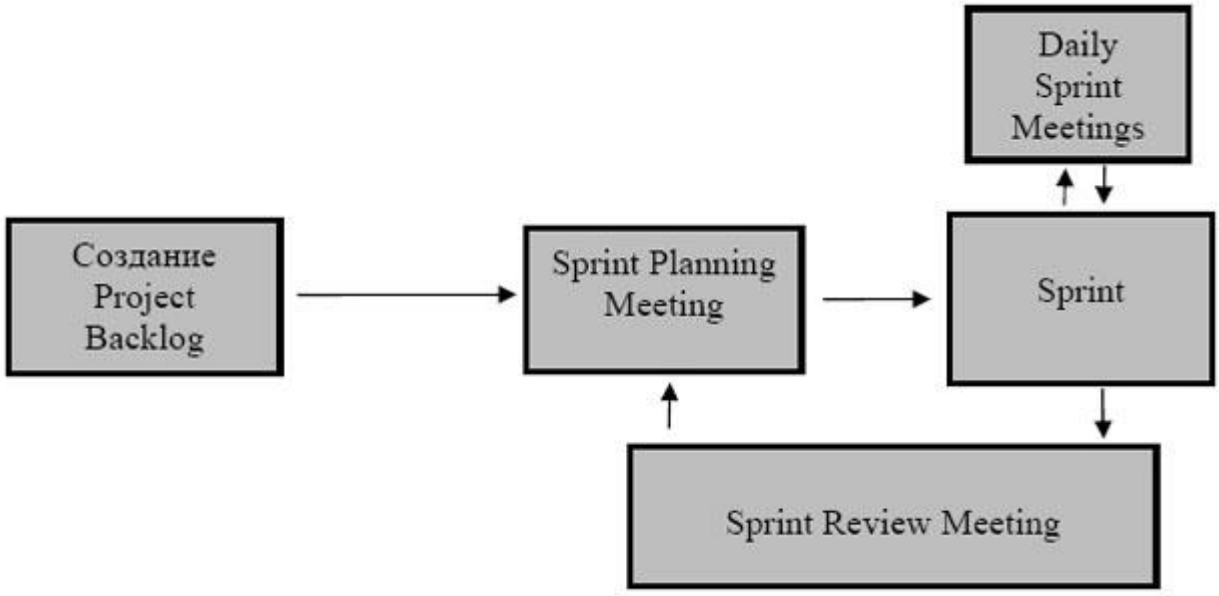
Scrum-команда (Scrum Team) – группа, состоящая из пяти–девяти самостоятельных, инициативных программистов. Первой задачей команды является постановка для итерации реально достижимых и приоритетных для проекта в целом задач (на основе Project Backlog и при активном участии владельца продукта и Scrum-мастера). Второй задачей является выполнение этой задачи во что бы то ни стало, в отведенные сроки и с заявленным качеством. Важно, что команда сама участвует в постановке задачи и сама же ее выполняет. Здесь сочетается свобода и ответственность, подобно тому, как это организовано в MSF. Здесь же "просвечивает" дисциплина обязательств.

Практики. В Scrum определены следующие практики.

Sprint Planning Meeting. Проводится в начале каждого Sprint. Сначала Product Owner, Scrum-мастер, команда, а также представители заказчика и пр. заинтересованные лица определяют, какие требования из Project Backlog наиболее приоритетные и их следует реализовывать в рамках данного Sprint. Формируется Sprint Backlog. Далее Scrum-мастер и Scrum-команда определяют то, как именно будут достигнуты определенные выше цели из Sprint Backlog. Для каждого элемента Sprint Backlog определяется список задач и оценивается их трудоемкость.

Daily Scrum Meeting – пятнадцатиминутное каждодневное совещание, целью которого является достичь понимания того, что произошло со времени предыдущего совещания, скорректировать рабочий план сообразно реалиям сегодняшнего дня и обозначить пути решения существующих проблем. Каждый участник Scrum-команды отвечает на три вопроса: что я сделал со времени предыдущей встречи, мои проблемы, что я буду делать до следующей встречи? В этом совещании может принимать участие любое заинтересованное лицо, но только участники Scrum-команды имеют право принимать решения. Правило обосновано тем, что они давали обязательство реализовать цель итерации, и только это дает уверенность в том, что она будет достигнута. На них лежит ответственность за их собственные слова, и, если кто-то со стороны вмешивается и принимает решения за них, тем самым он снимает ответственность за результат с участников команды. Такие встречи поддерживают дисциплину обязательств в Scrum-команде, способствуют удержанию фокуса на целях итерации, помогают решать проблемы "в зародыше". Обычно такие совещания проводятся стоя, в течение 15-20 минут.

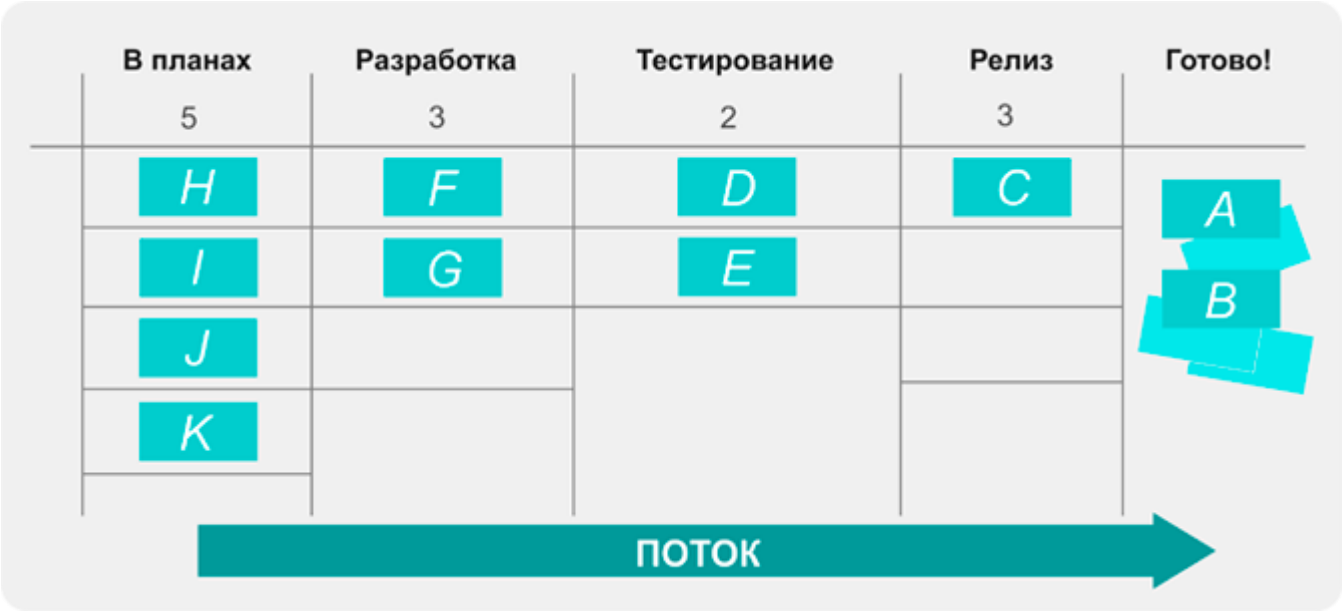
Sprint Review Meeting. Проводится в конце каждого Sprint. Сначала Scrum-команда демонстрирует Product Owner сделанную в течение Sprint работу, а тот в свою очередь ведет эту часть митинга и может пригласить к участию всех заинтересованных представителей заказчика. Product Owner определяет, какие требования из Sprint Backlog были выполнены, и обсуждает с командой и заказчиками, как лучше расставить приоритеты в Sprint Backlog для следующей итерации. Во второй части митинга производится анализ прошедшего спринта, который ведет Scrum-мастер. Scrum-команда анализирует в последнем Sprint положительные и отрицательные моменты совместной работы, делает выводы и принимает важные для дальнейшей работы решения. Scrum-команда также ищет пути для увеличения эффективности дальнейшей работы. Затем цикл повторяется.



Kanban.

Kanban — это система организации производства, которая пришла к нам с фирмы “Тойота”, и характеризуется тремя принципами:

- визуализация процесса работы — работу необходимо выписать на карточки и прикрепить их на стене, подписав столбцы, чтобы видеть этапы работы;
- ограничение количества незавершенной работы (НЗР) — необходимо определить максимальное кол-во работы на каждом этапе;
- измерение времени выполнения задачи.



Визуализация рабочего процесса при помощи Kanban доски

А что же такое Канбан разработка применительно к ПО, и чем она отличается от других гибких методологий, буть то SCRUM или XP?

Во-первых, нужно сразу понять, что Канбан — это не конкретный процесс, а система ценностей. Как, впрочем, и SCRUM с XP. Это значит, что никто вам не скажет что и как делать по шагам.
Во-вторых, весь Канбан можно описать одной простой фразой — **«Уменьшение выполняющейся в**

данный момент работы (work in progress)».
В-третьих, Канбан — это даже еще более «гибкая» методология, чем SCRUM и XP. Это значит, что она не подойдет всем командам и для всех проектов. И это также значит, что команда должна быть еще более готовой к гибкой работе, чем даже команды, использующие SCRUM и XP.

Разница между Канбан и SCRUM:
— В Канбан нет таймбоксов ни на что (ни на задачи, ни на спринты)
— В Канбан задачи больше и их меньше
— В Канбан оценки сроков на задачу опциональные или вообще их нет
— В Канбан «скорость работы команды» отсутствует и считается только среднее время на полную реализацию задачи.

Что нового и полезного дает такая доска с лимитами?

Во-первых, **уменьшение числа параллельно выполняемых задач сильно уменьшает время выполнения каждой отдельной задачи.** Нет нужды переключать контекст между задачами, отслеживать разные сущности, планировать их и т.д. — делается только то, что нужно. Нет нужды устраивать спринт планнинги и 5% воркшопы, т.к. планирование уже сделано в столбце «очередь задач», а детальная проработка задачи начинается ТОЛЬКО тогда, когда задача начинает выполняться.

Во-вторых, **сразу видны затыки.** Например, если тестеры не справляются с тестированием, то они очень скоро заполнят весь свой столбец и программисты, закончившие новую задачу, уже не смогут переместить ее в столбец тестирования, т.к. он заполнен. Что делать? Тут время вспомнить, что «мы — команда» и решить эту проблему. Например, программисты могут помочь тестерам завершить одну из задач тестирования и только тогда передвинуть новую задачу на освободившееся место. Это позволит выполнить обе задачи быстрее.

В-третьих, можно вычислить время на выполнение усредненной задачи. Мы можем помечать на карточке дату, когда она попала в очередь задач, потом дату, когда ее взяли в работу и дату, когда ее завершили. По этим трем точкам для хотя бы 10 задач можно уже посчитать среднее время ожидания в очередь задач и среднее время выполнения задачи. А из этих цифр менеджер или product owner может уже рассчитывать всё, что ему угодно.

Весь Канбан можно описать всего тремя основными правилами:
1. **Визуализируйте производство**
— Разделите работу на задачи, каждую задачу напишите на карточке и поместите на стену или доску.
— Используйте названные столбцы, чтобы показать положение задачи в производстве.
2. **Ограничивайте WIP** (work in progress или работу, выполняемую одновременно) **на каждом** этапе производства.
3. **Измеряйте время цикла** (среднее время на выполнение одной задачи) и **оптимизируйте постоянно процесс**, чтобы уменьшить это время.

#[#Управление](#) требованиями

1. Процесс выявления требований. Способы классификации требований. C- и D- требования. Функциональные и нефункциональные требования.

Сбор требований – это один из самых важных этапов при создании информационных систем и интернет-сайтов в частности. От того, насколько точно и полно будут учтены все пожелания заказчика в процессе проектирования сайта, и будет зависеть итоговый результат: получим ли мы сайт «для

галочки» или это будет эффективный инструмент бизнеса, который будет приносить прибыль своему владельцу.

Методы выявления требований

- Интервью, опросы, анкетирование
- Мозговой штурм, семинар
- Наблюдение за производственной деятельностью, «фотографирование» рабочего дня
- Анализ нормативной документации
- Анализ моделей деятельности
- Анализ конкурентных продуктов
- Анализ статистики использования предыдущих версий системы

Качество требований

- Характеристики качества требований по-разному определены различными источниками. Однако, следующие характеристики являются общепризнанными:

Характеристика	Объяснение
Единичность	Требование описывает одну и только одну вещь.
Завершённость	Требование полностью определено в одном месте и вся необходимая информация присутствует.
Последовательность	Требование не противоречит другим требованиям и полностью соответствует внешней документации.
Атомарность	Требование «атомарно». То есть оно не может быть разбито на ряд более детальных требований без потери завершённости.
Отслеживаемость	Требование полностью или частично соответствует деловым нуждам как заявлено заинтересованными лицами и документировано.
Актуальность	Требование не стало устаревшим с течением времени.
Выполнимость	Требование может быть реализовано в пределах проекта.
Недвусмысленность	Требование кратко определено без обращения к техническому жаргону, акронимам и другим скрытым формулировкам. Оно выражает объективные факты, не субъективные мнения. Возможна одна и только одна интерпретация. Определение не содержит нечётких фраз. Использование отрицательных утверждений и составных утверждений запрещено.
Обязательность	Требование представляет определённую заинтересованным лицом характеристику, отсутствие которой приведёт к неполноценности решения, которая не может быть проигнорирована. Необязательное требование — противоречие самому понятию требования.
Проверяемость	Реализованность требования может быть определена через один из четырёх возможных методов: осмотр, демонстрация, тест или анализ.

Л.Новиков в русской редакции нотации RUP [2.9]приводит следующее определение: "Требование - это условие или возможность, которой должна соответствовать система".

В IEEE Standard Glossary of Software Engineering Terminology (1990) [2.1] данное понятие трактуется шире. Требование - это:

1. *условия или возможности, необходимые пользователю для решения проблем или достижения целей;*

2. *условия или возможности, которыми должна обладать система или системные компоненты, чтобы выполнить контракт или удовлетворять стандартам, спецификациям или другим формальным документам;*
3. *документированное представление условий или возможностей для пунктов 1 и 2.*

Введем еще одно определение. Требования - это исходные данные, на основании которых проектируются и создаются автоматизированные информационные системы. Первичные данные поступают из различных источников, характеризуются противоречивостью, неполнотой, нечеткостью, изменчивостью. Требования нужны в частности для того, чтобы Разработчик мог определить и согласовать с Заказчиком временные и финансовые перспективы проекта автоматизации. Поэтому значительная часть требований должна быть собрана и обработана на ранних этапах создания АИС. Однако собрать на ранних стадиях все данные, необходимые для реализации АИС, удается только в исключительных случаях. На практике процесс сбора, анализа и обработки растянут во времени на протяжении всего жизненного цикла АИС, зачастую нетривиален и содержит множество подводных камней; подробнее о процессе - в лекциях 4 - 8.

КЛАССИФИКАЦИЯ

Существует значительное количество различных методов классификации требований [2.2-2.7], наиболее существенные из которых будут рассмотрены в лекции.

Требования к продукту и процессу

Требования к продукту. В своей основе требования - это то, что формулирует заказчик. Цель, которую он преследует - получить хороший конечный продукт: функциональный и удобный в использовании. Поэтому требования к продукту являются основополагающим классом требований. Более подробно требования к продукту детализируются в следующих ниже классификациях.

Требования к проекту. Вопросы формулирования требований к проекту, т.е. к тому, как Разработчик будет выполнять работы по созданию целевой системы, казалось бы, не лежат в компетенции Заказчика. Без регламентации процесса Заказчиком легко можно было бы обойтись, если бы все проекты всегда выполнялись точно и в срок. Однако, к сожалению, мировая статистика результатов программных проектов говорит об обратном. Заказчик, вступая в договорные отношения с Разработчиком, несет различные риски, главными из которых является риск получить продукт с опозданием, либо ненадлежащего качества. Основные мероприятия по контролю и снижению риска - регламентация процесса создания программного обеспечения и его аудит.

Насколько подробно Заказчику следует регламентировать требования к проекту - вопрос риторический. Ответ на него зависит от множества факторов, таких, как ценность конечного продукта для Заказчика, степень доверия Заказчика к Разработчику, сумма подписанного контракта, увязка срока сдачи продукта в эксплуатацию с бизнес-планами Заказчика и т.д. Однако, со всей определенностью можно сказать следующее:

1. регламентация процесса Заказчиком позволяет снизить его риски;
2. мероприятия Заказчика по регламентации процесса приводят к дополнительным накладным расходам. Требуется найти разумный компромисс между степенью контроля рисков и величиной расходов.

В качестве требований к проекту может быть внесен регламент отчетов Разработчика, совместных семинаров по оценке промежуточных результатов, определены характеристики компетенций участников рабочей группы, исполняющих проект, их количество, указана

методология управления проектом. Ниже приведен пример формулировки требования к оффшорному проекту (Заказчик и Разработчик физически находятся в разных государствах) - в этой ситуации Заказчику требуется жесткий контроль над Разработчиком.

1. Разработчик представляет Заказчику согласованный план работ с детализацией (WorkBreakdownStructure - WBS) с точностью до конкретных исполнителей.
2. Разработчик осуществляет ежедневные сборки, регрессионное тестирование компонентов разрабатываемого продукта и тестирование продукта в целом.
3. Все управленческие и проектные артефакты, исходные коды и тестовые примеры размещаются в режиме online в интегрированной среде разработки Rational ClearCase с возможностью для Заказчика осуществления online-мониторинга на базе web-технологий.

Уровни требований

Требования разделяются по уровням. Уровни требований связаны, с одной стороны, с уровнем абстракции системы, с другой - с уровнем управления на предприятии.

Обычно выделяют три уровня требований.

- На верхнем уровне представлены так называемые бизнес-требования (business requirements). Примеры бизнес-требования: система должна сократить срок оборачиваемости обрабатываемых на предприятии заказов в три раза. Бизнес-требования обычно формулируются топ-менеджерами, либо акционерами предприятия.
- Следующий уровень - уровень требований пользователей (user requirements). Пример требования пользователя: система должна представлять диалоговые средства для ввода исчерпывающей информации о заказе, последующей фиксации информации в базе данных и маршрутизации информации о заказе к сотруднику, отвечающему за его планирование и исполнение. Требования пользователей часто бывают плохо структурированными, дублирующимися, противоречивыми. Поэтому для создания системы важен третий уровень, в котором осуществляется формализация требований.
- Третий уровень - функциональный (functional requirements). Пример функциональных требований (или просто функций) по работе с электронным заказом: заказ может быть создан, отредактирован, удален и перемещен с участка на участок.

Системные требования и требования к программному обеспечению

Существуют различные трактовки понятия "Системные требования" (system requirements).

К. Вигерс формулирует данный термин, как "высокоуровневые требования к продукту, которые содержат многие подсистемы, то есть системе" [2.2]. При этом под системой понимается программная, программно-аппаратная, либо человеко-машинная система. Данная система является сложной, структурированной системой и системные требования являются подмножеством функциональных требований к продукту. В данное подмножество целесообразно относить наиболее важные, существенные требования, которые относятся в целом к системе и не содержат избыточной детализации.

INCOSE (International Council on Systems Engineering) дает более детальное определение системы: "комбинация взаимодействующих элементов, созданная для достижения определенных целей; может включать аппаратные средства, программное обеспечение, встроенное ПО, другие средства, людей, информацию, техники (подходы), службы и другие поддерживающие элементы". Таким образом, происходит разделение между системными требованиями, как обобщающему понятию и требованиями к программному обеспечению, как выделенному подмножеству системных требований, направленных исключительно на

программные компоненты системы. Этот же подход прослеживается в стандарте ГОСТ Р ИСО/МЭК 12207/99 [2.8]: работы, связанные с системой в целом и с программным обеспечением выделяются в отдельные группы в целях удобства оперирования.

В практике компьютерной инженерии бытует другой, более узкий контекст использования данного понятия: под системными требованиями в узком смысле понимаются требования, выдвигаемые прикладной программной системой (в частности - информационной) к среде своего функционирования (системной, аппаратной). Пример таких требований - тактовая частота процессора, объем памяти, требования к выбору операционной системы.

Функциональные, нефункциональные требования и характеристики продукта

Функциональные требования регламентируют функционирование или поведение системы (behavioral requirements). Функциональные требования отвечают на вопрос "что должна делать система" в тех или иных ситуациях. Функциональные требования определяют основной "фронт работ" Разработчика, и устанавливают цели, задачи и сервисы, предоставляемые системой Заказчику.

Функциональные требования записываются, как правило, при посредстве предписывающих правил: "система должна позволять кладовщику формировать приходные и расходные накладные". Другим способом являются так называемые варианты использования (uses cases) - популярный и весьма продуктивный способ представления требований.

Это - основной, определяющий вид требований, который будет рассматриваться на протяжении всего лекционного курса.

Нефункциональные требования, соответственно, регламентируют внутренние и внешние условия или атрибуты функционирования системы. К. Вигерс [2.2]выделяет следующие основные группы нефункциональных требований:

- Внешние интерфейсы (External Interfaces),
- Атрибуты качества (Quality Attributes),
- Ограничения (Constraints).

Среди внешних интерфейсов в большинстве современных АИС наиболее важным является интерфейс пользователя (User Interface, UI). Кроме того, выделяются интерфейсы с внешними устройствами (аппаратные интерфейсы), программные интерфейсы и интерфейсы передачи информации (коммуникационные интерфейсы).

Основные атрибуты качества:

- Применимость,
- Надежность,
- Производительность,
- Эксплуатационная пригодность,

достаточно хорошо раскрыты в модели FURPS (см. ниже).

Ограничения [2.2]- формулировки условий, модифицирующих требования или наборы требований, сужая выбор возможных решений по их реализации. Выбор платформы реализации и/или развертывания (протоколы, серверы приложений, баз данных, ...), которые, в свою очередь, могут относиться, например, к внешним интерфейсам (конец цитаты).

Характеристики продукта. К.Вигерс [2.2]формулирует характеристику, "фичу" (feature), как набор логически связанных функциональных требований, которые обеспечивают возможности пользователя и удовлетворяют бизнес-цели.

Существует и более общий взгляд на данное понятие [2.9]: "features могут быть как относящимися к функциональным, так и к нефункциональным требованиям и могут изменяться от версии к версии продукта".

С.Орлик в [2.6]отмечает, что "с точки зрения инженерии требований, features являются самостоятельным артефактом, который может быть соотнесен как с функциональными требованиями, так и с нефункциональными".

Роль характеристик проявляется в первую очередь в области маркетинга: не всякий потенциальный потребитель продукта станет читать его функциональные описания, а набор ключевых характеристик, характеризующих конкурентные преимущества, можно сделать лаконичным и уместить на одной странице рекламной листовки, либо напечатать на компакт-диске.

C-требования (customer requirements)

Пользовательские требования к системе должны описывать функциональные и нефункциональные системные требования так, чтобы они были понятны даже пользователю, не имеющему специальных технических знаний. Эти требования должны определять только внешнее поведение системы, избегая по возможности определения структурных характеристик системы. Пользовательские требования должны быть написаны естественным языком с использованием простых таблиц, а также наглядных и понятных диаграмм.

D-требования (developer requirements)

Системные требования— это более детализированное описание пользовательских требований. Они обычно служат основой для заключения контракта на разработку программной системы и поэтому должны представлять максимально полную спецификацию системы в целом. Системные требования также используются в качестве отправной точки на этапе проектирования системы.

Спецификация системных требований может строиться на основе различных системных моделей, таких, как объектная модель или модель потоков данных. Различные модели, используемые при разработке спецификации системных требований, рассматриваются в лекции 6.

В принципе системные требования определяют, что должна делать система, не показывая при этом механизма ее реализации. Но, с другой стороны, для полного описания системы требуется детализированная информация о ней, которая по возможности должна включать всю информацию о системной архитектуре. На то существует ряд причин.

1. Первоначальная архитектура системы помогает структурировать спецификацию требований. Системные требования должны описывать подсистемы, из которых состоит разрабатываемая система.

2. В большинстве случаев разрабатываемая система должна взаимодействовать с уже существующими системами. Это накладывает определенные ограничения на архитектуру новой системы.

3. В качестве внешнего системного требования может выступать условие использования для разрабатываемой системы специальной архитектуры.

2. Виды рабочих элементов в TFS. Epics, Features, Backlog Items, Tasks. Особенности каждого рабочего элемента. Жизненный цикл каждого элемента.

Not found

3. Понятие ценности для заказчика. Понятие архитектурной ценности. Модель Остервальдера.

Not found

4. Понятие об UML. Диаграммы для описания требований (Use Case, Activity, State, Classes).

UML (англ. *Unified Modeling Language* — унифицированный язык моделирования) — язык графического описания для объектного моделирования в области разработки программного обеспечения, моделирования бизнес-процессов, системного проектирования и отображения организационных структур.

UML является языком широкого профиля, это — открытый стандарт, использующий графические обозначения для создания абстрактной модели системы, называемой *UML-моделью*. UML был создан для определения, визуализации, проектирования и документирования, в основном, программных систем. UML не является языком программирования, но на основании UML-моделей возможна генерация кода.

Диаграмма прецедентов (use case diagram)

Любые (в том числе и программные) системы проектируются с учетом того, что в процессе своей работы они будут использоваться людьми и/или взаимодействовать с другими системами. Сущности, с которыми взаимодействует система в процессе своей работы, называются **экторами**, причем каждый эктор ожидает, что система будет вести себя строго определенным, предсказуемым образом. Попробуем дать более строгое определение эктора. Для этого воспользуемся замечательным визуальным словарем по UML *Zicom Mentor*:

Эктор (actor) - это множество логически связанных ролей, исполняемых при взаимодействии с прецедентами или сущностями (система, подсистема или класс). Эктором может быть человек или другая система, подсистема или класс, которые представляют нечто вне сущности.

Графически эктор изображается либо " *человечком* ", подобным тем, которые мы рисовали в детстве, изображая членов своей семьи, либо *символом класса с соответствующим стереотипом*, как показано на рисунке. Обе формы представления имеют один и тот же смысл

и могут использоваться в диаграммах. "Стереотипированная" форма чаще применяется для представления системных экторов или в случаях, когда эктор имеет свойства и их нужно отобразить (рис. 2.1).

Внимательный читатель сразу же может задать вопрос: *а почему эктор, а не актер*? Согласны, слово "эктор" немного режет слух русского человека. Причина же, почему мы говорим именно так, проста - эктор образовано от слова **action**, что в переводе означает *действие*. Дословный же перевод слова "эктор" - *действующее лицо* - слишком длинный и неудобный для употребления. Поэтому мы будем и далее говорить именно так.

Рис. 2.1.

Тот же внимательный читатель мог заметить промелькнувшее в определении эктора слово "прецедент". Что же это такое? Этот вопрос заинтересует нас еще больше, если вспомнить, что сейчас мы говорим о *диаграмме прецедентов*. Итак,

Прецедент (use-case) - описание отдельного аспекта поведения системы с точки зрения пользователя (Буч).

Определение вполне понятное и исчерпывающее, но его можно еще немного уточнить, воспользовавшись тем же *Zicom Mentor* 'ом:

Прецедент (use case) - описание множества последовательных событий (включая варианты), выполняемых системой, которые приводят к наблюдаемому эктором результату. Прецедент представляет поведение сущности, описывая взаимодействие между экторами и системой. Прецедент не показывает, "как" достигается некоторый результат, а только "что" именно выполняется.

Прецеденты обозначаются очень простым образом - в виде эллипса, внутри которого указано его название. *Прецеденты и экторы соединяются с помощью линий*. Часто на одном из концов линии изображают *стрелку*, причем *направлена она к тому, у кого запрашивают сервис*, другими словами, чьими услугами пользуются. Это простое объяснение иллюстрирует *понимание прецедентов как сервисов*, пропагандируемое компанией IBM.

Рис. 2.2.

Прецеденты могут включать другие прецеденты, расширяться ими, наследоваться и т. д. *Все эти возможности мы здесь рассматривать не будем*. Как уже говорилось выше, цель этого обзора - просто научить читателя выделять диаграмму прецедентов, понимать ее назначение и смысл обозначений, которые на ней встречаются.

Кстати, к этому моменту мы уже потратили достаточно много времени на объяснение понятий и их условных обозначений. Наверное, пора уже, наконец, привести пример диаграммы прецедентов. Как вы думаете, что означает эта диаграмма (рис. 2.3)?

Рис. 2.3.

Полагаем, здесь все было бы понятно, если бы даже мы никогда не слышали о диаграммах прецедентов! Ведь так? Все мы в студенческие годы пользовались библиотеками (которые теперь для нас заменил Интернет), и потому все это для нас знакомо. Обратите также внимание на примечание, сопоставленное с одним из прецедентов. Следует заметить, что иногда на диаграммах прецедентов *границы системы обозначают прямоугольником*, в верхней части которого может быть указано *название системы*. Таким образом, прецеденты - действия, выполняемые системой в ответ на действия эктора, - помещаются внутри прямоугольника.

А вот еще один пример (рис. 2.4). Думаем, вы сами, без нашей помощи, легко догадаетесь, о чем там идет речь.

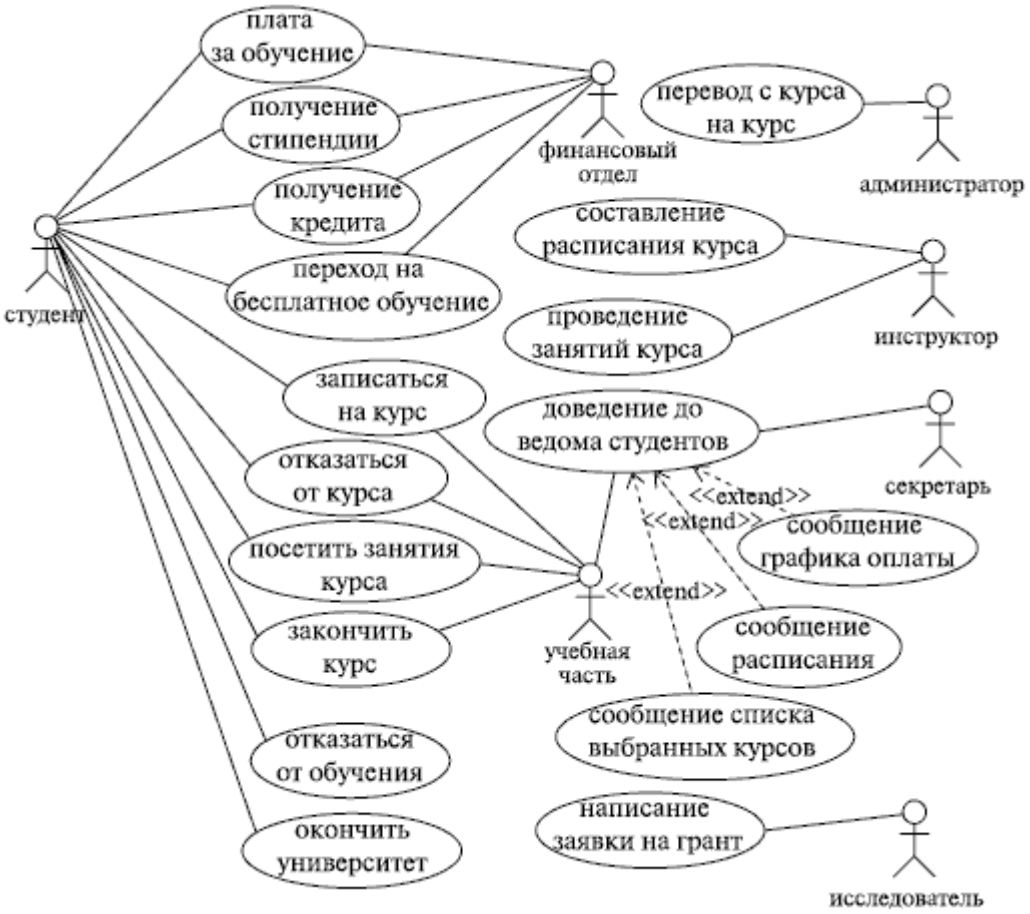


Рис. 2.4.

Из всего сказанного выше становится понятно, что *диаграммы прецедентов* относятся к той группе диаграмм, которые представляют динамические или поведенческие аспекты системы. Это отличное *средство для достижения взаимопонимания между разработчиками, экспертами и конечными пользователями* продукта. Как мы уже могли убедиться, такие диаграммы очень просты для понимания и могут восприниматься и, что немаловажно, обсуждаться людьми, не являющимися специалистами в области разработки ПО.

Подводя итоги, можно выделить такие **цели создания диаграмм прецедентов**:

- определение границы и контекста моделируемой предметной области на ранних этапах проектирования;
- формирование общих требований к поведению проектируемой системы;
- разработка концептуальной модели системы для ее последующей детализации;
- подготовка документации для взаимодействия с заказчиками и пользователями системы.

Диагра́мма де́ятельности (англ. *activity diagram*) — UML-диаграмма, на которой показано разложение некоторой деятельности на её составные части. Под деятельностью (англ. *activity*) понимается спецификация исполняемого поведения в виде координированного последовательного и параллельного выполнения подчинённых элементов — вложенных видов деятельности и отдельных действий англ. *action*, соединённых между собой потоками, которые идут от выходов одного узла ко входам другого.

Диаграммы деятельности используются при моделировании бизнес-процессов, технологических процессов, последовательных и параллельных вычислений.

Диаграммы деятельности состоят из ограниченного количества фигур, соединённых стрелками. Основные фигуры:

1. Прямоугольники с закруглениями — действия
2. Ромбы — решения
3. Широкие полосы — начало (разветвление) и окончание (схождение) ветвления действий
4. Чёрный круг — начало процесса (начальное состояние)
5. Чёрный круг с обводкой — окончание процесса (конечное состояние)

Стрелки идут от начала к концу процесса и показывают последовательность переходов.

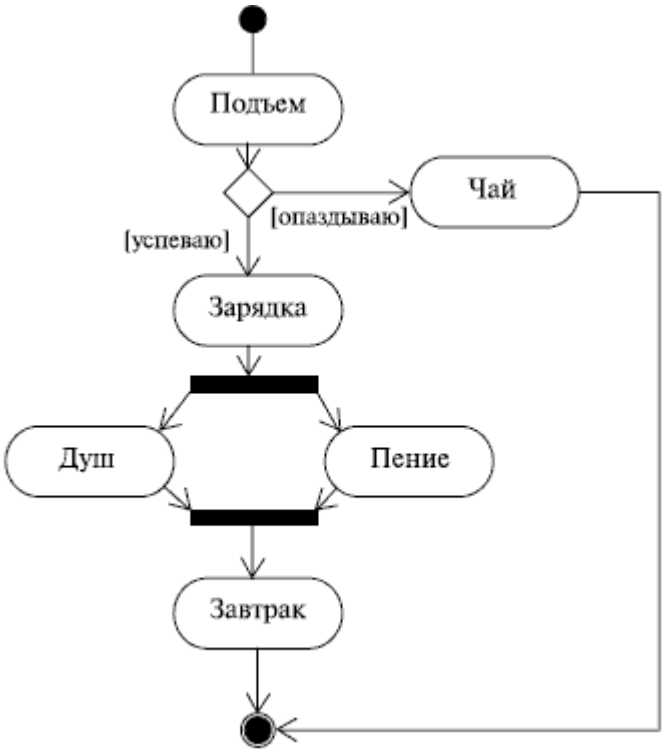


Диаграмма состояний (statechart diagram)

Диаграмма состояний — это, по существу, диаграмма состояний из теории автоматов со стандартизированными условными обозначениями[1] [2], которая может определять множество систем от компьютерных программ до бизнес-процессов. Используются следующие условные обозначения:

- Круг, обозначающий начальное состояние.
- Окружность с маленьким кругом внутри, обозначающая конечное состояния (если есть).
- Скруглённый прямоугольник, обозначающий состояние. Верхушка прямоугольника содержит название состояния. В середине может быть горизонтальная линия, под которой записываются активности, происходящие в данном состоянии.
- Стрелка, обозначающая переход. Название события (если есть), вызывающего переход, отмечается рядом со стрелкой. Охраняющее выражение может быть добавлено перед «/» и заключено в квадратные скобки (***название_события*[охраняющее_выражение]**), что значит, что это выражение должно быть истинным, чтобы переход имел место. Если при переходе производится какое-то действие, то оно добавляется после «/» (***название_события*[охраняющее_выражение]/действие**).
- Толстая горизонтальная линия с либо множеством входящих линий и одной выходящей, либо одной входящей линией и множеством выходящих. Это обозначает объединение и разветвление соответственно.

Диаграмма состояний показывает, как объект переходит из одного состояния в другое. Очевидно, что диаграммы состояний служат для моделирования динамических аспектов системы (как и диаграммы последовательностей, кооперации, прецедентов и, как мы увидим далее, диаграммы деятельности). Часто можно услышать, что *диаграмма состояний показывает автомат*, но об этом мы поговорим подробнее чуть позже. Диаграмма состояний полезна при моделировании жизненного цикла объекта (как и ее частная разновидность - диаграмма деятельности, о которой мы будем говорить далее).

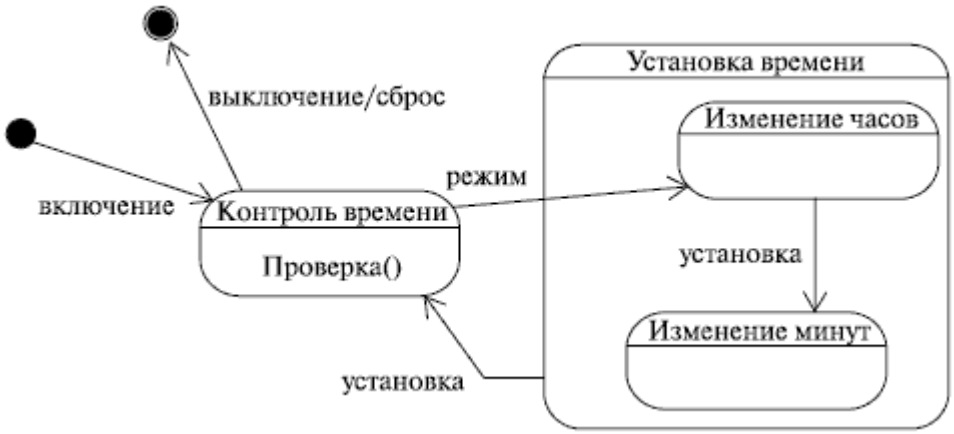


Рис. 2.20.

Не сомневаемся, что вы легко догадались, что это за устройство - конечно же, таймер! Такой прибор может применяться в составе различных реле, например, для отключения телевизора по истечении указанного промежутка времени. Основное его назначение - контроль времени (проверка, не истек ли указанный промежуток), но у него есть еще один режим работы - установка. По истечении указанного промежутка времени или при "сбросе" устройство отключается. В конце концов, о чем мы говорим - вы сами много раз устанавливали слип-таймер в телевизоре или устанавливали опцию "Выключить по завершении" в Nero Burning ROM!

Диаграмма классов (class diagram)

Диаграмма классов (англ. *Static Structure diagram*) — диаграмма, демонстрирующая классы системы, их атрибуты, методы и взаимосвязи между ними. Входит в UML.

Взаимосвязь — это особый тип логических отношений между сущностями, показанных на диаграммах классов и объектов. В UML'е представлены следующие виды отношений:

Ассоциации

Ассоциация показывает, что объекты одной сущности (класса) связаны с объектами другой сущности.

Если между двумя классами определена ассоциация, то можно перемещаться от объектов одного класса к объектам другого. Вполне допустимы случаи, когда оба конца ассоциации относятся к одному и тому же классу. Это означает, что с объектом некоторого класса позволительно связать другие объекты из того же класса. Ассоциация, связывающая два класса, называется бинарной. Можно, хотя это редко бывает необходимым, создавать ассоциации, связывающие сразу несколько классов; они называются n-арными. Графически ассоциация изображается в виде линии, соединяющей класс сам с собой или с другими классами.

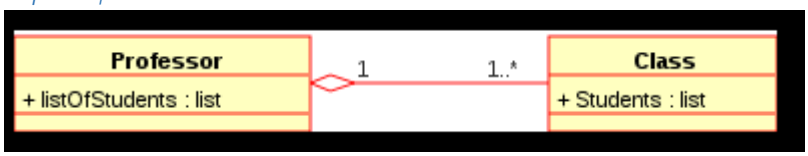
Ассоциации может быть присвоено имя, описывающее природу отношения. Обычно имя ассоциации не указывается, если только вы не хотите явно задать для нее ролевые имена или в

вашей модели настолько много ассоциаций, что возникает необходимость ссылаться на них и отличать друг от друга. Имя будет особенно полезным, если между одними и теми же классами существует несколько различных ассоциаций.

Класс, участвующий в ассоциации, играет в ней некоторую роль. По существу, это "лицо", которым класс, находящийся на одной стороне ассоциации, обращен к классу с другой ее стороны. Вы можете явно обозначить роль, которую класс играет в ассоциации.

Часто при моделировании бывает важно указать, сколько объектов может быть связано посредством одного экземпляра ассоциации. Это число называется кратностью (Multiplicity) роли ассоциации и записывается либо как выражение, значением которого является диапазон значений, либо в явном виде. Указывая кратность на одном конце ассоциации, вы тем самым говорите, что на этом конце именно столько объектов должно соответствовать каждому объекту на противоположном конце. Кратность можно задать равной единице (1), можно указать диапазон: "ноль или единица" (0..1), "много" (0..*), "единица или больше" (1..*). Разрешается также указывать определенное число (например, 3). С помощью списка можно задать и более сложные кратности, например 0 . . 1, 3..4, 6..*, что означает "любое число объектов, кроме 2 и 5".

Агрегация



Агрегация. Простая ассоциация между двумя классами отражает структурное отношение между равноправными сущностями, когда оба класса находятся на одном концептуальном уровне и ни один не является более важным, чем другой. Но иногда приходится моделировать отношение типа "часть/целое", в котором один из классов имеет более высокий ранг (целое) и состоит из нескольких меньших по рангу (частей). Отношение такого типа называют агрегированием; оно причислено к отношениям типа "имеет" (с учетом того, что объект-целое имеет несколько объектов-частей). Агрегация является частным случаем ассоциации и изображается в виде простой ассоциации с незакрашенным ромбом со стороны "целого".

Графически агрегация представляется пустым ромбом на блоке класса, и линией, идущей от этого ромба к содержащемуся классу.

Композиция

Композиция — более строгий вариант агрегации. Известна также как агрегация по значению.

Композиция имеет жёсткую зависимость времени существования экземпляров класса контейнера и экземпляров содержащихся классов. Если контейнер будет уничтожен, то всё его содержимое будет также уничтожено.

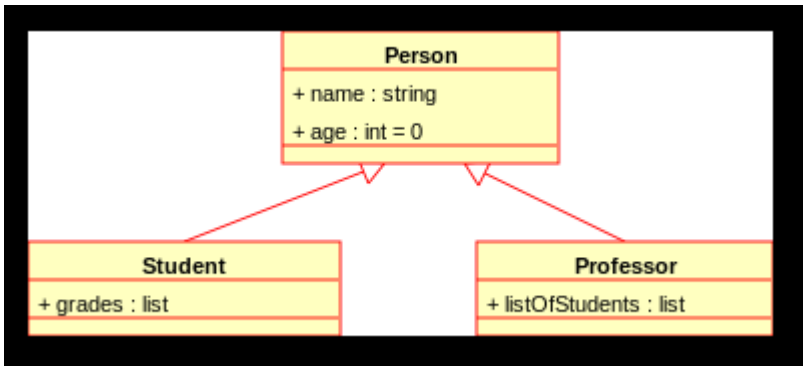
Графически представляется как и агрегация, но с закрашенным ромбиком.

Различия между композицией и агрегацией

Отношения между классом Вуз и классами Студент и Факультет слегка отличаются друг от друга, хотя оба являются отношениями агрегирования. В ВУЗе может быть любое количество студентов (включая ноль), и каждый студент может обучаться в одном или нескольких вузах; вуз может состоять из одного или нескольких факультетов, но каждый факультет принадлежит одному и только одному ВУЗу. Отношение между классами ВУЗ и Факультет называют

композицией, так как при уничтожении модели Вуз модели факультетов, принадлежащих этому вузу, также должны быть уничтожены. Студент и Вуз в отношении агрегации потому, что Студента нельзя удалить при уничтожении Вуза.[1]

Обобщение (наследование)



Обобщение (Generalization) показывает, что один из двух связанных классов (*подтип*) является частной формой другого (*надтипа*), который называется **обобщением** первого. На практике это означает, что любой экземпляр подтипа является также экземпляром надтипа. Например: животные — супертип млекопитающих, которые, в свою очередь, — супертип приматов, и так далее. Эта взаимосвязь легче всего описывается фразой «А — это Б» (приматы — это млекопитающие, млекопитающие — это животные).

Графически обобщение представляется линией с пустым треугольником у супертипа.

Обобщение также известно как наследование или «is a» взаимосвязь (или отношение "является").

Реализация

Реализация — отношение между двумя элементами модели, в котором один элемент (*клиент*) реализует поведение, заданное другим (*поставщиком*). Реализация — отношение целое-часть. Графически реализация представляется так же, как и наследование, но с пунктирной линией.

Зависимость

Зависимость (dependency) — это слабая форма отношения использования, при котором изменение в спецификации одного влечёт за собой изменение другого, причём обратное не обязательно. Возникает, когда объект выступает, например, в форме параметра или локальной переменной.

Графически представляется штриховой стрелкой, идущей от зависимого элемента к тому, от которого он зависит.

Существует несколько именованных вариантов.

Зависимость может быть между экземплярами, классами или экземпляром и классом.

Уточнения отношений

Уточнение имеет отношение к уровню детализации. Один пакет уточняет другой, если в нём содержатся те же самые элементы, но в более подробном представлении. Например, при

написании книги вы наверняка начнете с формулировки предложения, в котором кратко будет представлено содержание каждой главы. Предположим, что резюме к каждой главе в качестве отдельного элемента входит в пакет «Предложение». Допустим также, что «Завершённая книга» — это пакет, элементами которого являются законченные главы. В этом контексте пакет «Завершённая книга» является уточнением пакета «Предложение».

Мощность отношений (Кратность)!

Мощность отношения (мультипликатор) означает число связей между каждым экземпляром класса (объектом) в начале линии с экземпляром класса в её конце. Различают следующие типичные случаи:

нотация	*объяснение*	*пример*
0..1	Ноль или один экземпляр	кошка имеет или не имеет хозяина
1	Обязательно один экземпляр	у кошки одна мать
0..* или *	Ноль или более экземпляров	у кошки может быть, а может и не быть котят
1..*	Один или более экземпляров	у кошки есть хотя бы одно место, где она спит

##Проектирование

1. Понятие о логической и физической архитектурах. Deployment диаграммы. Layer диаграммы.

Архитектура системы — принципиальная организация системы, воплощенная в её элементах, их взаимоотношениях друг с другом и со средой, а также принципы, направляющие её проектирование и эволюцию[1]:3.

Логическая архитектура

Логическая архитектура поддерживает функционирование системы на протяжении всего её жизненного цикла на логическом уровне. Она состоит из набора связанных технических концепций и принципов. Логическая архитектура представляется с помощью методов, соответствующих тематическим группам описаний, и как минимум, включает в себя функциональную архитектуру, поведенческую архитектуру и временную архитектуру.

Функциональная архитектура. Функциональная архитектура представляет собой набор функций и их подфункций, определяющих преобразования, осуществляемые системой при выполнении своего назначения.

Поведенческая архитектура. Поведенческая архитектура — соглашение о функциях и их подфункциях, а также интерфейсах (входы и выходы), которые определяют последовательность выполнения, условия для управления или потока данных, уровень производительности, необходимый для удовлетворения системных требований. Поведенческая архитектура может быть описана как совокупность взаимосвязанных сценариев, функций и/или эксплуатационных режимов.

Временная архитектура. Временная архитектура является классификацией функций системы, которая получена в соответствии с уровнем частоты её исполнения. Временная архитектура включает в себя определение синхронных и асинхронных аспектов функций. Мониторинг решений, который происходит внутри системы, следует той же временной классификации [4]:287.

Физическая архитектура

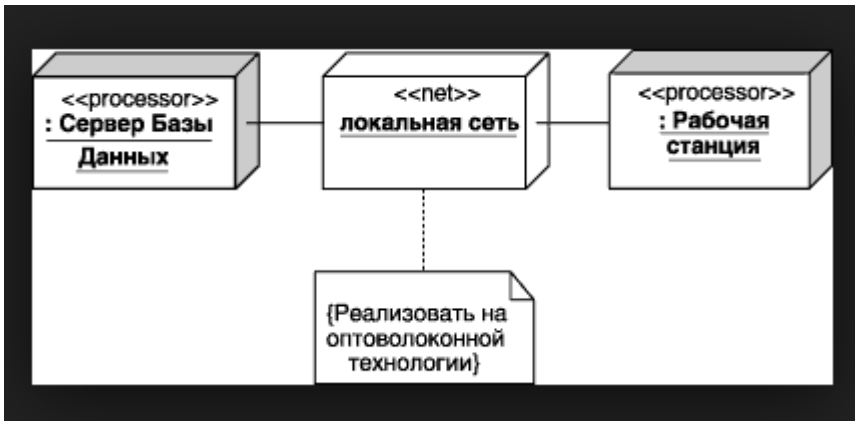
Цель проектирования физической архитектуры заключается в создании физического, конкретного решения, которое согласовано с логической архитектурой и удовлетворяет установленным системным требованиям.

После того, как логическая архитектура определена, должны быть идентифицированы конкретные физические элементы, которые поддерживают функциональные, поведенческие, и временные свойства, а также ожидаемые свойства системы, полученные из нефункциональных требований к системе.

Физическая архитектура является систематизацией физических элементов (элементов системы и физических интерфейсов), которые реализуют спроектированные решения для продукта, услуги или предприятия. Она предназначена для удовлетворения требований к системе и элементам логической архитектуры и реализуется через технологические элементы системы. Системные требования распределяются как на логическую, так и физическую архитектуру. Глобальная архитектура системы оценивается с помощью системного анализа и, после выполнения всех требований, становится основой для реализации системы[4]:296.

Диаграмма развертывания (deployment diagram)

Когда мы пишем программу, мы пишем ее для того, чтобы запускать на компьютере, который имеет некоторую аппаратную конфигурацию и работает под управлением некоторой операционной системы. Корпоративные приложения часто требуют для своей работы некоторой *ИТ-инфраструктуры*, хранят информацию в базах данных, расположенных где-то на серверах компании, вызывают веб-сервисы, используют общие ресурсы и т. д. В таких случаях неплохо бы иметь *графическое представление инфраструктуры, на которую будет развернуто приложение*. Вот для этого-то и нужны **диаграммы развертывания**, которые иногда называют диаграммами размещения.



Диагра́мма развёртывания, **Deployment diagram** в UML моделирует *физическое* развертывание артефактов на узлах.[1] Например, чтобы описать веб-сайт диаграмма развертывания должна показывать, какие аппаратные компоненты («узлы») существуют (например, веб-сервер, сервер базы данных, сервер приложения), какие программные компоненты («артефакты») работают на каждом узле (например, веб-приложение, база данных), и как различные части этого комплекса соединяются друг с другом (например, JDBC, REST, RMI).

Узлы представляются как прямоугольные параллелепипеды с артефактами, расположенными в них, изображенными в виде прямоугольников. Узлы могут иметь подузлы, которые представляются как вложенные прямоугольные параллелепипеды. Один узел диаграммы развертывания может концептуально представлять множество физических узлов, таких как кластер серверов баз данных.

Существует два типа узлов:

1. Узел устройства
2. Узел среды выполнения

Узлы устройств — это физические вычислительные ресурсы со своей памятью и сервисами для выполнения программного обеспечения, такие как обычные ПК, мобильные телефоны. Узел среды выполнения — это программный вычислительный ресурс, который работает внутри внешнего узла и который предоставляет собой сервис, выполняющий другие исполняемые программные элементы.

LAYER DIAGRAMS

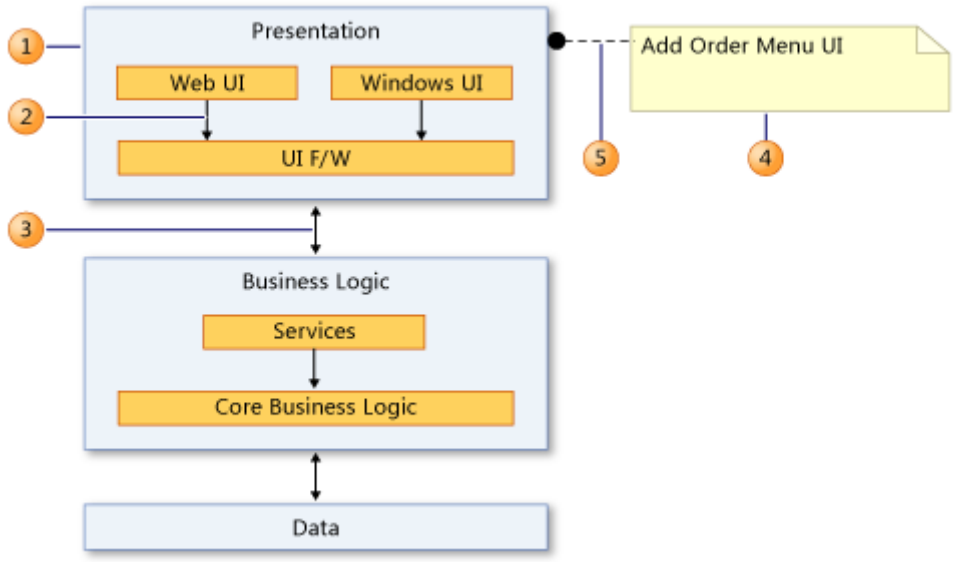
В Visual Studio можно визуализировать высокоуровневую логическую архитектуру системы с помощью *схемы слоев*.Схема слоев упорядочивает физические артефакты системы в логические, абстрактные группы, называемые *слоями*.Слои описывают основные компоненты системы или задачи, выполняемые этими артефактами.Каждый слой может также содержать вложенные слои, описывающие более подробные задачи.

Чтобы узнать, какие версии Visual Studio поддерживают эту функцию, см. раздел Поддержка версий для инструментов моделирования и архитектуры.

Между слоями можно установить предполагаемые или существующие зависимости.Эти зависимости, представленные в виде стрелок, показывают, какие слои могут использовать или в настоящее время используют функциональные возможности, представленные другими слоями.Упорядочивая системы в слои, описывающие различные роли и функции, схемы слоев помогают изучать, повторно использовать и обслуживать код.

С помощью схемы слоев можно также выполнять следующие задачи:

- представлять существующую или предполагаемую логическую архитектуру системы;
- выявлять конфликты между существующим кодом и предполагаемой архитектурой;
- визуализировать влияние изменений на предполагаемую архитектуру при рефакторинге, обновлении или развитии системы;
- дополнительно контролировать предполагаемую архитектуру в процессе разработки и обслуживания кода за счет добавления проверки операций возврата и построения.



В следующей таблице описаны элементы, которые можно использовать на схемах слоев.

Фигура	Элемент	Описание
1	**Уровень**	Логическая группа физических артефактов в системе.Артефактами могут быть пространства имен, проекты, классы, методы и т. п. Для просмотра связанных со слоем артефактов откройте контекстное меню слоя и выберите пункт **Просмотр связей**, чтобы открыть **Обозреватель слоев**. Дополнительные сведения см. в разделе Обозреватель слоев. • Зависимости от запрещенного пространства имен — указывает, что артефакты, связанные с этим слоем, не могут зависеть от указанных пространств имен. • Запрещенные пространства имен — указывает, что артефакты, связанные с этим слоем, не могут принадлежать к указанным пространствам имен. • Обязательные пространства имен — указывает, что артефакты, связанные с этим слоем, должны принадлежать к указанным пространствам имен.
2	**Зависимость**	Указывает, что один слой может использовать функции другого слоя, но не наоборот. • Направление — указывает направление зависимости.
3	**Двусторонняя зависимость**	Указывает, что один слой может использовать функции другого слоя и наоборот. • Направление — указывает направление зависимости.
4	**COMMENT**	Используется для добавления общих примечаний к схеме или ее элементам.

5	**Добавить комментарий для ссылки**	Используется для связи комментариев с элементами на схеме.

2. Правила GRASP и SOLID для проектирования объектно-ориентированных систем.

GRASP (General Responsibility Assignment Software Patterns) — шаблоны проектирования, используемые для решения общих задач по назначению обязанностей классам и объектам.

Ниже следует краткая характеристика девяти известных шаблонов.

Information Expert (Информационный эксперт)

Шаблон ***Информационный эксперт*** определяет базовый принцип назначения обязанностей. Он утверждает, что *обязанности должны быть назначены объекту, который владеет максимумом необходимой информации для выполнения обязанности*. Такой объект называется информационным экспертом. Возможно, этот шаблон является самым очевидным из девяти, но вместе с тем и самым важным.

Если дизайн не удовлетворяет этому принципу, то при программировании получается спагетти-код, в котором очень трудно разбираться. Локализация обязанностей позволяет повысить уровень инкапсуляции и уменьшить степень зацепления. Кроме читабельности кода повышается уровень готовности компонента к повторному использованию.

Creator (Создатель)

См. также: Абстрактная фабрика (шаблон проектирования)

Шаблон ***Создатель*** решает, кто должен создавать объект. Фактически, это применение шаблона Информационный эксперт к проблеме создания объектов. Более конкретно, нужно назначить классу B обязанность создавать экземпляры класса A, если выполняется как можно больше из следующих условий:

- Класс B *содержит* или агрегирует объекты A.
- Класс B *записывает* экземпляры объектов A.
- Класс B *активно использует* объекты A
- Класс B *обладает данными инициализации* для объектов A.

Альтернативой Создателю является шаблон проектирования Фабрика. В этом случае создание объектов концентрируется в отдельном классе.

Controller (Контроллер)

Контроллер берёт на себя ответственность за выполнение операций, приходящих от пользователя и часто выполняет сценарий одного или нескольких вариантов использования (например, один контроллер может обрабатывать сценарии создания и удаления пользователя). Как правило, контроллер не выполняет работу самостоятельно, а делегирует обязанности компетентным объектам.

Иногда класс-контроллер представляет всю систему в целом, корневой объект, устройство или важную подсистему (*внешний контроллер*).

Low Coupling (Низкая связность)

Низкая связность — это принцип, который позволяет распределить обязанности между объектами таким образом, чтобы степень связности* между *системами оставалась низкой. Степень связности— это мера, определяющая, насколько жестко один элемент связан с другими элементами, либо каким количеством данных о других элементах он обладает. Объект с низкой степенью связности зависит от не очень большого числа других объектов и имеет следующие свойства:

- Малое число зависимостей между классами (подсистемами).
- Слабая зависимость одного класса (подсистемы) от изменений в другом классе (подсистеме).
- Высокая степень повторного использования подсистем.

High Cohesion (Высокое зацепление)

Высокое зацепление — это принцип, который задаёт свойство высокого зацепления* внутри *подсистемы. Классы не берут на себя обязанности из разных предметных областей и получаются сфокусированными, управляемыми и понятными. Зацепление — это мера схожести предметной области методов класса, методы разных предметных областей лучше разделить на классы.

Класс с низкой степенью зацепления выполняет много разнородных функций или несвязанных между собой обязанностей. Такие классы создавать нежелательно, поскольку они приводят к возникновению следующих проблем:

- Трудность понимания.
- Сложность при повторном использовании.
- Сложность поддержки.
- Ненадежность, постоянная подверженность изменениям.

Классы с низкой степенью зацепления, как правило, являются слишком «абстрактными» или выполняют обязанности, которые можно легко распределить между другими объектами.

Polymorphism (Полиморфизм)

Полиморфизм позволяет обрабатывать альтернативные варианты поведения на основе типа и заменять подключаемые компоненты системы. Обязанности распределяются для различных вариантов поведения с помощью полиморфных операций для этого класса. Все альтернативные реализации приводятся к общему интерфейсу.

Пример

Интеграция разрабатываемой системы с различными внешними системами учета налогов. Используются локальные программные объекты, обеспечивающие адаптацию (Адаптеры). При отправке сообщения к такому объекту выполняется обращение к внешней системе с использованием ее собственного программного интерфейса. Использование полиморфизма здесь оправдано для возможной адаптации к различным внешним системам.

Pure Fabrication (Чистая выдумка)

Чистая выдумка — это класс, не отражающий никакого реального объекта предметной области, но специально придуманный для усиления связности, ослабления зацепления или увеличения степени повторного использования. Pure Fabrication отражает концепцию сервисов

в модели проблемно-ориентированного проектирования. Пример: есть класс, который нужно записать в базу данных. Мы не можем позволить классу изнутри лезть в базу данных, так как это приведёт к слабой связности внутри класса. И мы создаём новый класс с именем «MyClassDao» или «MyClassRepository», который будет иметь сильную связность внутри и единую ответственность записывать объект класса в базу данных

Indirection (Посредник)

См. также: Посредник (шаблон проектирования)

Посредник поддерживает слабое зацепление между двумя элементами (и возможность повторного использования) путём назначения обязанности посредника между ними промежуточному объекту.

Пример

Model-View-Controller, который позволяет ослабить связь между данными и их представлением с помощью введения класса-посредника (контроллер), отвечающего за поведение.

Protected Variations (Устойчивый к изменениям)

Шаблон **Protected Variations** защищает элементы от изменения других элементов (объектов или подсистем) с помощью вынесения взаимодействия в фиксированный интерфейс. Всё взаимодействие между элементами должно происходить через него. Поведение может варьироваться лишь с помощью создания другой реализации интерфейса.

SOLID это аббревиатура пяти основных принципов проектирования классов в объектно-ориентированном программировании — Single responsibility, Open-closed, Liskov substitution, Interface segregation и Dependency inversion.

Буква	Означает	Описание
S	Single responsibility principle	Принцип единственной обязанности На каждый класс должна быть возложена одна-единственная обязанность.
O	Open/closed principle	Принцип открытости/закрытости Программные сущности должны быть открыты для расширения, но закрыты для изменения.
L	Liskov substitution principle	Принцип подстановки Барбары Лисков *Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом.* *Подклассы не могут замещать поведения базовых классов. Подтипы должны дополнять базовые типы.*
I	Interface segregation principle	Принцип разделения интерфейса Много специализированных интерфейсов лучше, чем один универсальный.
D	Dependency inversion principle	Принцип инверсии зависимостей Зависимости внутри системы строятся на основе абстракций. Модули верхнего уровня не зависят от модулей нижнего уровня. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

##Кодирование

1. Инструменты Code Review. Сценарий проведения Code Review.

2. Понимание централизованных и распределенных систем контроля версий.

3. Сценарии разработки при TDD и BDD. Преимущества и недостатки.

##Тестирование

1. Инструменты модульного тестирования. Continues Integration.

2. Инструменты ручного функционального тестирования. Автоматизация ручного тестирования.

3. Инструменты нагрузочного тестирования.

##Развертывание

1. Понятие о DevOps. Варианты автоматизации развертывания. Docker и Ansible.

##Управление персоналом

1. Компетенции ИТ специалистов. Развитие компетенций. Проверка компетенций.

2. Мотивация персонала. Материальная мотивация. Нематериальная мотивация.

1. Инструменты Code Review. Сценарий проведения Code Review.

Code review - инженерная практика в терминах гибкой методологии разработки. Это анализ (инспекция) кода с целью выявить ошибки, недочеты, расхождения в стиле написания кода, в соответствии написанного кода и поставленной задачи.

К очевидным плюсам этой практики можно отнести:

- Улучшается качество кода
- Находятся «глупые» ошибки (опечатки) в реализации
- Повышается степень совместного владения кодом
- Код приводится к единому стилю написания
- Хорошо подходит для обучения «новичков», быстро набирается навык, происходит выравнивание опыта, обмен знаниями.

Что можно инспектировать

Для ревью подходит любой код. Однако, review **обязательно** должно проводиться для критических мест в приложении (например, механизмы аутентификации, авторизации, передачи и обработки важной информации — обработка денежных транзакций и пр.). Также для review подходят и юнит тесты, так как юнит тесты — это тот же самый код, который подвержен ошибкам, его нужно инспектировать также тщательно, как и весь остальной код, потому что, неправильный тест может стоить очень дорого.

Как проводить review

Вообще, ревью кода должен проводиться в совокупности с другими гибкими инженерными практиками: парное программирование, TDD, CI. В этом случае достигается максимальная эффективность ревью. Если используется гибкая методология разработки, то этап code review можно внести в Definition of Done фичи.

Из чего состоит review

- Сначала design review — анализ будущего дизайна (архитектуры). Данный этап очень важен, так как без него ревью кода будет менее полезным или вообще бесполезным (если программист написал код, но этот код полностью неверен — не решает поставленную задачу, не удовлетворяет требованиям по памяти, времени). Пример: программисту поставили задачу написать алгоритм сортировки массива. Программист реализовал алгоритм bogo-sort, причем с точки зрения качества кода — не придраться (стиль написания, проверка на ошибки), но этот алгоритм совершенно не подходит по времени работы. Поэтому ревью в данном случае бесполезно (конечно — это утрированный пример, но я думаю, суть ясна), здесь необходимо полностью переписывать алгоритм.
- Собственно, сам code review — анализ написанного кода. На данном этапе автору кода отправляются замечания, пожелания по написанному коду.

Также очень важно определиться, за кем будет последнее слово в принятии финального решения в случае возникновения спора. Обычно, приоритет отдается тому, кто будет реализовывать код (как в scrum при проведении planning poker), либо специальному человеку, который отвечает за этот код (как в google — code owner).

Как проводить design review

Design review можно проводить за столом, в кругу коллег, у маркерной доски, в корпоративной wiki. На design reviewтот, кто будет писать код, расскажет о выбранной стратегии (примерный алгоритм, требуемые инструменты, библиотеки) решения поставленной задачи. Вся прелесть этого этапа заключается в том, что ошибка проектирования будет стоить 1-2 часа времени (и будет устранена сразу на review).

Как проводить code review

Можно проводить code review разными способами — дистанционно, когда каждый разработчик сидит за своим рабочим местом, и совместно — сидя перед монитором одного из коллег, либо в

специально выделенным для этого месте, например, meeting room. В принципе существует много способов (можно даже распечатать исходный код и вносить изменения на бумаге).

Тематические review

Можно также проводить тематические code review — их можно использовать как переходный этап на пути к полноценному code review. Их можно проводить для критического участка кода, либо при поиске ошибок. Самое главное — это определить цель данного review, при этом цель должна быть обозримой и четкой:

- "Давайте поищем ошибки в этом модуле" — не подходит в качестве цели, так как она необозрима.
- "Анализ алгоритма на соответствие спецификации RFC 1149" — уже лучше.

Основное отличие тематических review от полноценного code review — это их узкая специализация. Если в code review мы смотрим на стиль кода, соответствие реализации и постановки задачи, поиск опасного кода, то в тематическом review мы смотрим обычно только один аспект (чаще всего — анализ алгоритма на соответствие ТЗ, обработка ошибок).

Преимущество такого подхода заключается в том, что команда постепенно привыкает к практике review (его можно использовать нерегулярно, по требованию). Получается некий аналог мозгового штурма. Мы использовали такой подход при поиске логических ошибок в нашем ПО: смотрели «старый» код, который был написан за несколько месяцев до review (это можно отнести тоже к отличиям от обычного review — где обычно смотрят свежий код).

Результаты review

Самое главное при проведении review — это использование полученного результата. В результате review могут появиться следующие артефакты:

- Описание способа решения задачи (design review)
- UML диаграммы (design review)
- Комментарии к стилю кода (code review)
- Более правильный вариант (быстрый, легкочитаемый) реализации (design review, code review)
- Указание на ошибки в коде (забытое условие в switch, и т.д.) (code review)
- Юнит тесты (design review, code review)

При этом очень важно, чтобы все результаты не пропали, и были внесены в VCS, wiki. Этому могут препятствовать:

- Сроки проекта.
- Лень, забывчивость разработчиков
- Отсутствие удобного механизма внесения изменений review, а также контроль внесения этих изменений.

Для преодоления этих проблем частично может помочь:

- pre-commit hook в VCS
- Создание ветви в VCS, из которой изменения вливаются в основную ветвь только после review
- Запрет сборки дистрибутива на CI сервере без проведения review. Например, при сборке дистрибутива проверять специальные свойства (svn:properties), либо специальный файл с результатами review. И отказывать в сборке дистрибутива, если не все ревьюверы одобрили (approve) код.
- Использование методологии в разработке (в которой code review является неотъемлемой частью).

Инструмены

Возьмем самых распространенных кандидатов.

1. **Phabricator**. Если кто еще не знает, Phabricator — комбайн-набор утилит родом из facebook, содержащий багтрекер (Maniphest), просмотр репозиториев с кодом (Diffusion), инструмент автоматических подписок на рецензии (Herald), Wiki проекта (Phriction). Исходники на PHP. Проект, конечно же, Open Source. Вещь очень интересная и достойна внимания, но, к сожалению, такой комбайн не совсем применим для нашей задачи: у клиента уже есть большая часть инструментов, идущих в наборе с Phabricator (трекер, вики). Есть CLI и API.
2. **Gerrit** (форк Rietveld). Утилита от инженеров google, была создана для девелопмента Android-проекта. Ну очень аскетичная, изначально была заточена под Git и только под Git, исходники написаны на Java (Java EE Servlet), т. ч. стоит озаботиться наличием SDK на сервере и одной из поддерживаемых СУБД (MySQL, PostgreSQL и встроенная H2). Что порадовало из вкусностей — ACL и связанная с ним система оценивания, есть ранжирование оценок. Из минусов, как по мне, несколько сложная система конфигурирования, многое можно сделать напрямую через базу, что идеологически неправильно.
3. **Review Board**. Ну, а это уже бывшая внутренняя утилита VMWare. Написана на Python. Поддерживает разные системы контроля версий (CVS, Subversion, Git, Mercurial, Bazaar, Perforce, ClearCase, Plastic SCM) Поднимается довольно быстро и без бубнов, административная часть тоже имеет достаточно настроек. Кроме WUI, имеет API и консольную часть, поэтому можно считать его довольно гибким и доступным для кастомизации под свои нужды. Правда, изначально был заточен под Pre-commit review, но это уже можно считать пройденным этапом.
4. **Rietveld.** Детище самого Гвидо ван Россума, думаю, не стоит упоминать, на чем этот боард написан. Заточена под хостинг на Google App Engine. Но уже давно есть много методов запускать на своем сервере, как хранилище использовать PostgreSQL. Но немногие идут на этот шаг. Интерфейс аскетичен, нет ничего лишнего.
5. **Crucible.** Один из продуктов Atlassian, конечно же, платный, хотя цена для небольших команд (пять и меньше) очень вкусная. Написан на Java, поддерживает достаточное количество систем контроля версий (CVS, Subversion, Git, Mercurial, Perforce), нативно интегрируется с Jira. Огромное количество настроек, т. е. даже без гуру-навыков в администрировании его можно настроить под свои нужды.

Посмотрев даже на этот кратенький обзор и на нашу тестовую задачу, можно сказать что выбор вполне заслужено мог пасть на Crucible, он очень оправдан, особенно если нет свободных часов на администрирование. Но — команда больше пяти человек, да и лишних денег никто не хочет вкладывать, значит, от этой идеи стоит отказаться.

NDepend – Инструмент для Visual Studio для проведения комплексного анализа .NET кода и обеспечения его высокого качества.

Сценарий проведения Code Review.

Исследование, проведенное Software Engineering Institute, показывает, что программисты делают 15-20 распространенных ошибок. Добавив такие ошибки в чеклист, вы можете быть уверены, что заметите их в момент появления и поможете от них избавиться надолго.

Чтобы вам было от чего отталкиваться, вот вам список типичных пунктов:

Чек-лист CodeReview

Общее

- Работает ли код? Выполняет ли он свои прямые обязанности, корректна ли логика, и т. д.
- Легок ли код для понимания?
- Соответствует ли код вашему стилю написания кода? Обычно это относится к расположению скобок, названиям переменных и функций, длинам строк, отступам, форматированию и комментариям.
- Есть ли в ревью избыточный или повторяющийся код?
- Является ли код независимым, насколько это возможно?
- Можно ли избавиться от глобальных переменных или переместить их?
- Есть ли закомментированный код?
- У циклов есть установленная длина и корректные условия завершения?
- Может ли что-то в коде быть заменено библиотечными функциями?
- Может ли быть удалена часть кода, предназначенного для логгирования или отладки?

Безопасность

- Все ли входные данные проверяются (на корректный тип, длину, формат, диапазон) и кодируются?
- Обрабатываются ли ошибки при использовании сторонних утилит?
- Выходные данные проверяются и кодируются (*прим. пер.: например, от XSS*)?
- Обрабатываются ли неверные значения параметров?

Документация

- Есть ли комментарии? Раскрывают ли они смысл кода?
- Все ли функции прокомментированы?
- Есть ли какое-то необычное поведение или описание пограничных случаев?
- Использование и функционирование сторонних библиотек документировано?
- Все ли структуры данных и единицы измерения описаны?
- Есть ли незавершенный код? Если есть, должен ли он быть удален или помечен маркером типа «TODO»?

Тестирование

- Является ли код тестируемым? Например, он не должен содержать слишком много зависимостей или скрывать их, тестовые фреймворки должны иметь возможность использовать методы кода, и т. д.
- Есть ли тесты и если есть, то достаточны ли они? Например, они покрывают код в нужной мере.
- Юнит-тесты на самом деле проверяют, что код предоставляет требуемую функциональность?
- Все ли массивы проверяются на «выход за границы»?
- Может ли любой тестирующий код быть заменен с использованием существующего API?

Вы также можете захотеть добавить в этот чеклист любые вещи, специфичные для вашего языка и способные вызвать проблемы.
Чеклист сознательно не охватывает всех возможных проблем. Вам не нужен контрольный список, который настолько длинный, что никто его не использует. Лучше всего покрыть им только общие вопросы.

Оптимизируйте свой чеклист.

Этот чеклист можно использовать в качестве отправной точки: вам нужно оптимизировать его под ваши специфичные сценарии и проблемы. Отличный способ сделать это — отмечать вопросы, которые возникают в

вашей команде вовремя ревью кода в течение короткого времени. С этой информацией вы сможете найти самые распространенные ошибки команды, чтобы затем внести их в ваш чеклист.
Не бойтесь выбрасывать из чеклиста те элементы, которые вам не подходят (вы можете захотеть оставить редко встречающиеся, но все же критические пункты — например, связанные с безопасностью).

Начните использовать и держите в актуальном состоянии

Как правило, любые пункты чеклиста должны быть конкретными и, если возможно, давать вам однозначный ответ на ваш вопрос. Это помогает избежать непоследовательности суждений.
Еще одна хорошая идея — поделиться списком с вашей командой и получить их согласие с его содержимым. Периодически просматривайте и сам чеклист для проверки, что каждый пункт в нем все еще актуален.

Вооружившись хорошим чеклистом, вы можете увеличить количество дефектов, обнаруживаемых вовремя ревью кода. Это позволит вам улучшить стандарт кодирования и избежать плохого качества ревью.

Понимание централизованных и распределенных систем контроля версий.

О Контроле Версий

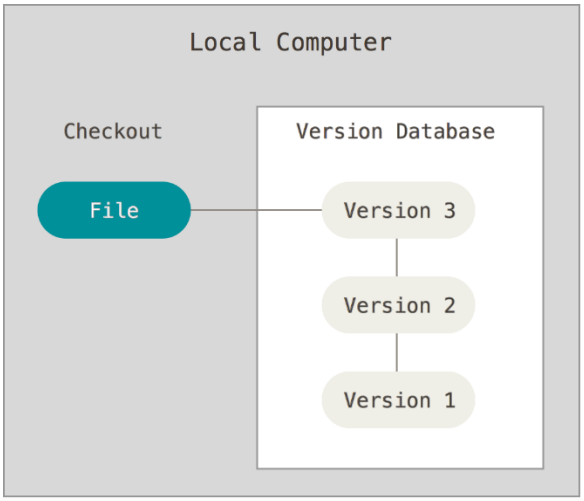
Что такое "контроль версий", и почему вы должны заботиться? Контроль версий - это система, которая записывает изменения в файл или набор файлов, с течением времени, так что вы можете вспомнить конкретные варианты позже. Для примеров в этой книге мы будем использовать программное обеспечение, исходный код в виде файлов под контролем версий, хотя в действительности вы можете сделать это с практически любого типа файлов на компьютере.

Если вы графический или веб-дизайнер и хотите хранить каждую версию изображения или макета (который вы наверняка хотите), система контроля версий (СКВ) - это очень мудрая вещь, чтобы использовать. Она позволяет вернуть файлы обратно в Предыдущее состояние, откатить весь проект вернуться к предыдущему состоянию, сравнивать изменения с течением времени, см., который последним изменил что-то, что может быть причиной неполадки, которые появились проблемы и когда, и больше. Пользуясь СКВ, как правило, означает, что если вы все испортите или потеряете файлы, Вы можете легко восстановить. Кроме того, вы получите все это за очень небольшие накладные расходы.

Локальные Системы Контроля Версий

Многих людей версия-контроль методом выбора является копирование файлов в другой каталог (возможно с временной меткой каталога, если они умные). Такой подход очень распространен, ведь это так просто, но он также невероятно подвержен ошибкам. Легко забыть в каком каталоге вы находитесь и случайно написать не тот файл, или скопировать файлы не туда, куда хотел.

Чтобы решить эту проблему, программисты уже давно разработали локальные СКВ с простой базой данных, в которой хранятся все изменения нужных файлов контроля.



Одной из наиболее популярных СКВ такого типа была система rcs, которая до сих пор распространяется на многие компьютеры. Даже популярный Мак ОС х Операционная система включает в `PBC` команду, когда вы устанавливаете инструменты разработчика. PBC работает, сохраняя наборов исправлений (то есть различия между файлами) в специальном формате на диске; он может заново создать какой-либо файл выглядел как в любой момент времени путем суммирования всех патчей.

Централизованные Системы Управления Версиями

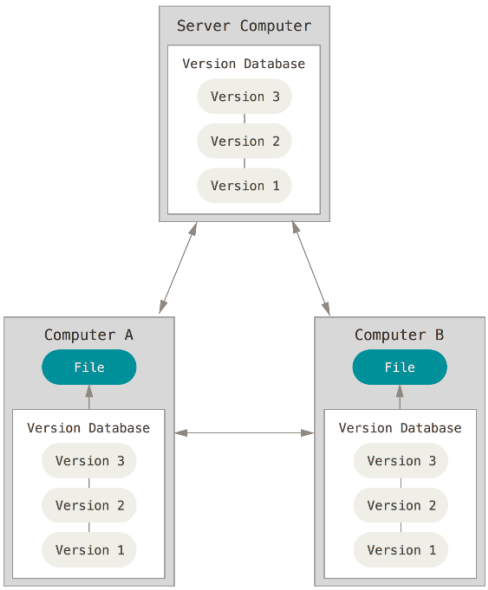
Следующей основной проблемой оказалась необходимость сотрудничать с разработчиками других систем. Чтобы справиться с этой проблемой, централизованных систем контроля версий (CVCSs) были разработаны. Эти системы, такие как CVS, subversion и perforce, есть сервер, на котором хранятся все файлы под версионным контролем, и ряд клиентов, которые получают копии файлов из него, что центральное место. В течение многих лет, это было стандартом для контроля версий.

Такой подход имеет множество преимуществ, особенно над локальными СКВ. Например, все знают, до известной степени, что все остальные на проекте делает. Администраторы имеют мелкозернистый контроль над тем, кто и что может делать, и это гораздо легче администрировать CVCS, чем иметь дело с локальными базами данных на каждого клиента.

Однако эта установка также имеет несколько серьезных недостатков. Наиболее очевидным является единой точкой отказа, что централизованный сервер представляет. Если сервер выключается на час, то в течение часа никто не может взаимодействовать, и сохранить новой версии они работают. Если жесткий диск с центральной базой данных на повреждения, и надлежащие резервные копии не сохранились, вы теряете абсолютно все – всю историю проекта ни один, кроме снимков людей посчастливилось иметь на своих локальных машинах. Локальные системы контроля версий подвержены той же проблеме: если вся история проекта хранится в одном месте, вы рискуете потерять все.

Распределенные Системы Контроля Версий

Это где распределенными системами контроля версий (DVCSs) шага. В таких системах как git, а не mercurial, Bazaar или помощью darcs клиенты не просто проверить последний снимок файлов: они полностью зеркало репозитория. Таким образом, если какой-либо сервер умрет, и эти системы сотрудничают через него, любой клиентский репозиторий может быть скопирован обратно на сервер, чтобы восстановить его. Каждый клон-это действительно полное резервное копирование всех данных.



Кроме того, многие из этих интернет-систем довольно хорошо с несколькими удаленными репозиториями, с которыми можно работать, так что вы можете взаимодействовать с различными группами людей различными способами одновременно в одном проекте. Это позволяет установить несколько типов процессов, которые невозможно в централизованных системах, таких как иерархические модели.

Другой вариант инфы

Централизованные системы контроля версий

Соответственно, появились централизованные системы контроля версий.То есть, у нас появился сервер, где находится последняя версия всего дерева проекта, всей разработки. В качестве разработки у нас может быть пользовательская документация, любые файлы, сценарное тестирование, допустим (как только что говорили) – для него тоже надо хранить версии, чтобы понимать, где новые изменения.
Открытыми системами этого типа являются CVS и SVN. До вчерашнего дня я думал, что **CVS (ConcurrentVersionsSystem)** уже можно выкинуть на свалку истории, но, оказывается, есть в нем некоторые моменты, которые все-таки до сих пор применимы и в наших сегодняшних жизненных реалиях.

Subversion(SVN– сокращение от названия клиента для командной строки, входящей в состав пакета) – это эволюционное развитиеCVS. По сравнению со своим прототипом что-то нашло, а что-то потеряло.

Хранилище 1С тоже является централизованной системой хранения версий. В принципе, в хранилище у нас есть и сервер, и права доступа, и две чашки кофе вы можете успеть выпить, пока захватите рекурсивно весь корень конфигурации, да? А если по http-доступу еще или по tcp – ваше счастье… Придется долго ждать.

Работа централизованных систем версий проста. Для каждого программиста-новичка понятна.

1. Получил из хранилища
2. Захватил
3. Отредактировал
4. Поместил обратно.

Такая простая схема работы сцентрализованными системами быстро завоевала признание людей, так как позволила быстро вникнуть в разработку и получить последнюю версию/копию.

Но у централизованных систем есть и некоторые недостатки:

1. Первый недостаток – это конечно то, что **сервер** фактически является **«единой точкой отказа»**. В связи с этим совет: **делайте backup хранилища**. На самом деле, хотя бы базы данных.Но хранилища тоже, пожалуйста, делайте. Потому что если с сервером что-то случится – восстановиться можно будет только из backup-а.
2. Следующая особенность – **пессимистическая блокировка**. Некоторые считают, что, когда в централизованной системе разработчик захватывает объект, он,таким образом, всем объявляет свое намерение, что собирается в этом объекте что-то изменить. К тому же, многие руководители думают, что они, таким образом, контролируют этого разработчика. То есть, раз разработчик захватил какой-то там документ – значит, мы знаем, что он над ним работает, и он его к какому-то периоду времени сделает. По факту, если это уже опытный специалист, и вы ему доверяете, то он, конечно, всегда обоснует, почему он месяц назад объект захватил и до сих пор не поместил его в хранилище. Но все равно – эта пессимистическая блокировка **накладывает на нас ограничение, которого можно было бы избежать**. В дальнейшем в распределенных системах обошли этот путь.
   Также, если у вас идет глобальный рефакторинг, или идет какое-то исправление модуля и срочно необходимо исправить ошибку – тоже может возникнуть проблема. **В худшем случае – база стоит, ничего не работает**. В лучшем – возникает дилемма, 50% кода уже исправлена, то ли помещать в хранилище неоттестированный код, то ли сохранять его куда-то в файл. И у нас опять появляется в файловой системе «Копия конфигурации такой-то» и мы забыли/не забыли ее вернуть обратно в конфигурацию.

3. И есть еще в централизованных системах такое соглашение – это за или против, я даже сомневаюсь – что **последняя версия** у нас самая оттестированная и **самая рабочая**. По факту, наверное, так и есть. Но **только если мы тестируем на пользователях**. То есть, нам сказали, что есть ошибки, мы их исправили и только тогда положили объект в хранилище. Только в этом случае мы можем быть уверены, что в хранилище у нас более-менее рабочая копия.
В любых других случаях – для простого программиста у нас тестов нет. **Разработчики** типовых конфигураций **не покрывают код тестами**. То есть, дописал свое, запустил полное тестирование, проверил, как работает – такого нет. Будет ли? Время покажет.

Вехи истории: централизованные системы

За	Против
Контроль и учет (пессимистическая блокировка).	Забыл выложить и сделал 2 задачи сразу.
Единый сервер.	Backup.
Последняя версия самая рабочая.	Последняя версия самая "оттестированная"?

Распределенные системы контроля версий

Следующим этапом развития систем контроля версий стали, конечно же, распределенные системы.**Основная идея** этих **распределенных систем контроля ревизий** – **это ветвление**.

Ветвления – это некие параллельные процессы, когда есть один родитель и две истории от него отпочковываются. Самое хорошее у них то, что с помощью одного этого родителя вы в дальнейшем можете объединить эти истории.

Условный такой **пример ветвления**, который все знают – **это конфигурация на поставке**. Файл обновлений является «патчем» для конфигурации поставщика. При обновлении конфигурации поставщика на ее основе воссоздается новая конфигурация поставщика, которую вы не видите, но она есть. И, запустив действие «Показать дважды измененные», вы можете увидеть упрощенный вариант «трехстороннего сравнения». То есть, можно сказать так: **в конфигураторе «трехстороннее сравнение» есть, но оно скрыто от разработчика**. Те, у кого есть «Снегопат», могут что-то в этой ситуации подкорректировать и все-таки посмотреть трехстороннее сравнение. Но от остальной массы разработчиков трехстороннее сравнение в конфигураторе скрыто.

Преимущество распределенных систем в том, что у них **нет единого сервера**, и **каждая копия** хранилища **может** в любой момент **выступить в качестве сервера**. Но за эту особенность нам приходится платить местом на жестком диске. Это, конечно, проблема. Но, учитывая современные возможности и небольшую стоимость соответствующего оборудования, эта проблема незначительная. Сами распределенные системы поддерживают режим работы как в серверном режиме (как централизованные системы контроля версий), так и в децентрализованном режиме. Причем, согласно моему практическому опыту, для распределенных систем сервер все-таки нужен.

Ключевые особенности

Итак – что отличает распределенные системы контроля версий от всех остальных?

- Во-первых, как только перед вами встанет задача с кем-то обменяться информацией – для вас станет необходимостью организовать какой-то сервер. Пускай он будет развернут даже на вашей локальной машине, но **для обмена сервер необходим**.
- Во-вторых, по сравнению с централизованными системами, где обычно можно было бы на определенные объекты поставить ограничение доступа, **в распределенных системах нет смысла говорить о какой-то блокировке на определенный файл**. В централизованной системе, например, в хранилище, у нас есть возможность на определенные объекты поставить доступ на чтение/запись, а в распределенной системе, поскольку вся копия проекта копируется разработчику на машину, разработчик все равно может этот файл изменить, прочитать и т.д. Это, наверное, одна из проблем… Но при желании, если вам необходимо какому-то удаленному разработчику выгрузить определенную подсистему, чтобы он ваше основное дерево конфигурации не смотрел, вы можете выделить эту подсистему в отдельную конфигурацию. Способы решения есть. Например, можно какую-то часть дерева истории выгрузить, а потом обратно загрузить уже изменения, которые разработчик внес.
- Еще один важный момент, о котором я говорил: в централизованных системах на сервере должен всегда быть оттестированный код. **В децентрализованных системах**, когда вечером вы уходите домой, вы **можете** спокойно **поместить в свою локальную копию неработающий код**. Поверьте, это очень удобно, когда поместили свои изменения и записали для них какой-то естественный комментарий. От комментария многое зависит. Если вы с пустым комментарием что-то помещаете в, то же хранилище, то я бы это не приветствовал. Понятно, что вы тут что-то изменили… а что по факту? С учетом скорости хранилища — это практически невозможно узнать.
- Ветвление, естественно, является палкой о двух концах. Оно, как линия жизни вашего проекта, может уйти куда-то в сторону, а может держаться генеральной линии. Это надо прочувствовать на себе. И **основной принцип – ветвиться часто, но в меру**.

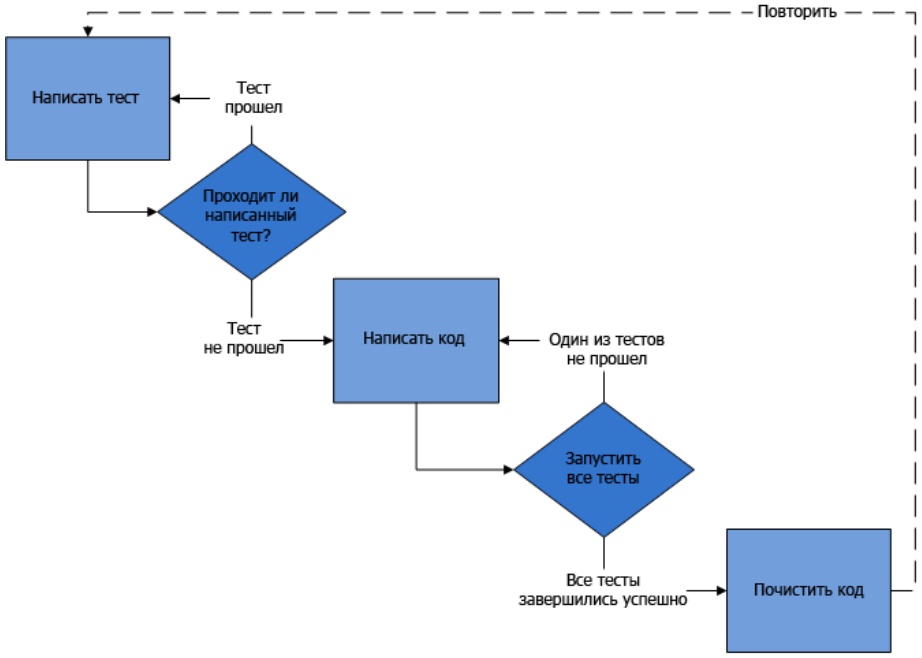
Режимы работы распределенных систем

Распределенные системы поддерживают различные режимы работы.

- Если вы разворачиваете проект для одного себя, вы **можете использовать базу** хранения версий **локально**.
- **Можно** по старинке – **с центральным сервером. Причем**, вы можете **копировать** не полностью весь репозиторий на него, а **только нужные вам для хранения истории файлы**. Допустим, выгрузка хранилища 8.3 занимает 1.5 Гб. Разработчику первый раз выгрузить содержимое репозитория по каналу интернета очень тяжело. В то же время, можно выкачать только последние изменения, с которыми он работает. Так, поддерживается тот же принцип централизованной системы, как и в хранилище.
- Без центрального сервера тоже можно работать. Но, в любом случае, какой-то сервер, хотя бы на локальной машине, организовывать все равно нужно.
- И наконец, самый предпочтительный и самый удобный способ – с центральным сервером, с локальными фиксациями и ветками. Однако настройка будет наиболее трудоемкой.

Сценарии разработки при TDD и BDD. Преимущества и недостатки.

Разработка через тестирование(TDD) (англ. *test-driven development, TDD*) — техника разработки программного обеспечения, которая основывается на повторении очень коротких циклов разработки: сначала пишется тест, покрывающий желаемое изменение, затем пишется код, который позволит пройти тест, и под конец проводится рефакторинг нового кода к соответствующим стандартам. Кент Бек, считающийся изобретателем этой техники, утверждал в 2003 году, что разработка через тестирование поощряет простой дизайн и внушает уверенность (англ. *inspires confidence*).



Цикл разработки через тестирование

Приведенная последовательность действий основана на книге Кента Бека «Разработка через тестирование: на примере» (англ. *Test Driven Development: By Example*).[1]

Добавление теста

При разработке через тестирование, добавление каждой новой функциональности (англ. *feature*) в программу начинается с написания теста. Неизбежно этот тест не будет проходить, поскольку соответствующий код ещё не написан. (Если же написанный тест прошёл, это означает, что-либо предложенная «новая» функциональность уже существует, либо тест имеет недостатки.) Чтобы написать тест, разработчик должен чётко понимать предъявляемые к новой возможности требования. Для этого рассматриваются возможные сценарии использования и пользовательские истории. Новые требования могут также повлечь изменение существующих тестов. Это отличает разработку через тестирование от техник, когда тесты пишутся после того, как код уже написан: она заставляет разработчика сфокусироваться на требованиях до написания кода — тонкое, но важное отличие.

Запуск всех тестов: убедиться, что новые тесты не проходят

На этом этапе проверяют, что только что написанные тесты не проходят. Этот этап также проверяет сами тесты: написанный тест может проходить всегда и соответственно быть бесполезным. Новые тесты должны не проходить по объяснимым причинам. Это увеличит уверенность (хотя не будет гарантировать полностью), что тест действительно тестирует то, для чего он был разработан.

Написать код

На этом этапе пишется новый код так, что тест будет проходить. Этот код не обязательно должен быть идеален. Допустимо, чтобы он проходил тест каким-то неэлегантным способом. Это приемлемо, поскольку последующие этапы улучшат и отполируют его.

Важно писать код, предназначенный именно для прохождения теста. Не следует добавлять лишней и, соответственно, не тестируемой функциональности.

Запуск всех тестов: убедиться, что все тесты проходят

Если все тесты проходят, программист может быть уверен, что код удовлетворяет всем тестируемым требованиям. После этого можно приступить к заключительному этапу цикла.

Рефакторинг

Когда достигнута требуемая функциональность, на этом этапе код может быть почищен. Рефакторинг — процесс изменения внутренней структуры программы, не затрагивающий её внешнего поведения и имеющий целью облегчить понимание её работы, устранить дублирование кода, облегчить внесение изменений в ближайшем будущем.

Повторить цикл

Описанный цикл повторяется, реализуя всё новую и новую функциональность. Шаги следует делать небольшими, от 1 до 10 изменений между запусками тестов. Если новый код не удовлетворяет новым тестам или старые тесты перестают проходить, программист должен вернуться к отладке. При использовании сторонних библиотек не следует делать настолько небольшие изменения, которые буквально тестируют саму стороннюю библиотеку[3], а не код, её использующий, если только нет подозрений, что библиотека содержит ошибки.

Плюсы

Исследование 2005 года показало, что использование разработки через тестирование предполагает написание большего количества тестов, в свою очередь, программисты, пишущие больше тестов, склонны быть более продуктивными.[5] Гипотезы связывающие качество кода с TDD были неубедительны.[6]

Программисты, использующие TDD на новых проектах, отмечают, что они реже ощущают необходимость использовать отладчик. Если некоторые из тестов неожиданно перестают проходить, откат к последней версии, которая проходит все тесты, может быть более продуктивным, нежели отладка.

Разработка через тестирование предлагает больше, чем просто проверку корректности, она также влияет на дизайн программы. Изначально сфокусировавшись на тестах, проще представить, какая функциональность необходима пользователю. Таким образом, разработчик продумывает детали интерфейса до реализации. Тесты заставляют делать свой код более приспособленным для тестирования. Например, отказываться от глобальных переменных, одиночек (singletons), делать классы менее связанными и легкими для использования. Сильно связанный код или код, который требует сложной инициализации, будет значительно труднее протестировать. Модульное тестирование способствует формированию четких и небольших интерфейсов. Каждый класс будет выполнять определенную роль, как правило небольшую. Как следствие, зависимости между классами будут снижаться, а зацепление повышаться. Контрактное программирование (англ. *design by contract*) дополняет тестирование, формируя необходимые требования через утверждения (англ. *assertions*).

Несмотря на то, что при разработке через тестирование требуется написать большее количество кода, общее время, затраченное на разработку, обычно оказывается меньше. Тесты защищают от ошибок. Поэтому время, затрачиваемое на отладку, снижается многократно. Большое количество тестов помогает уменьшить количество ошибок в коде. Устранение дефектов на более раннем этапе разработки, препятствует появлению хронических и дорогостоящих ошибок, приводящих к длительной и утомительной отладке в дальнейшем.

Тесты позволяют производить рефакторинг кода без риска его испортить. При внесении изменений в хорошо протестированный код риск появления новых ошибок значительно ниже. Если новая функциональность приводит к ошибкам, тесты, если они, конечно, есть, сразу же это покажут. При работе с кодом, на который нет тестов, ошибку можно обнаружить спустя значительное время, когда с кодом работать будет намного сложнее. Хорошо протестированный код легко переносит рефакторинг. Уверенность в том, что изменения не нарушит существующую функциональность, придает уверенность разработчикам и увеличивает эффективность их работы. Если существующий код хорошо покрыт тестами, разработчики будут чувствовать себя намного свободнее при внесении архитектурных решений, которые призваны улучшить дизайн кода.

Разработка через тестирование способствует более модульному, гибкому и расширяемому коду. Это связано с тем, что при этой методологии разработчику необходимо думать о программе как о множестве небольших модулей, которые написаны и протестированы независимо и лишь потом соединены вместе. Это приводит к меньшим, более специализированным классам, уменьшению связанности и более чистым интерфейсам. Использование mock-объектов также вносит вклад в модуляризацию кода, поскольку требует наличия простого механизма для переключения между mock- и обычными классами.

Поскольку пишется лишь тот код, что необходим для прохождения теста, автоматизированные тесты покрывают все пути исполнения. Например, перед добавлением нового условного оператора разработчик должен написать тест, мотивирующий добавление этого условного оператора. В результате получившиеся в результате разработки через тестирование тесты достаточно полны: они обнаруживают любые непреднамеренные изменения поведения кода.

Тесты могут использоваться в качестве документации. Хороший код расскажет о том, как он работает, лучше любой документации. Документация и комментарии в коде могут устаревать. Это может сбивать с толку разработчиков, изучающих код. А так как документация, в отличие от тестов, не может сказать, что она устарела, такие ситуации, когда документация не соответствует действительности — не редкость.

Минусы

- Существуют задачи, которые невозможно (по крайней мере, на текущий момент) решить только при помощи тестов. В частности, TDD не позволяет механически продемонстрировать адекватность разработанного кода в области безопасности данных и взаимодействия между процессами. Безусловно, безопасность основана на коде, в котором не должно быть дефектов, однако она основана также на участии человека в процедурах защиты данных. Тонкие проблемы, возникающие в области взаимодействия между процессами, невозможно с уверенностью воспроизвести, просто запустив некоторый код.
- Разработку через тестирование сложно применять в тех случаях, когда для тестирования необходимо прохождение функциональных тестов. Примерами может быть: разработка интерфейсов пользователя, программ, работающих с базами данных, а также того, что зависит от специфической конфигурации сети. Разработка через тестирование не предполагает большого объёма работы по тестированию такого рода вещей. Она сосредотачивается на тестировании отдельно взятых модулей, используя mock-объекты для представления внешнего мира.
- Требуется больше времени на разработку и поддержку, а одобрение со стороны руководства очень важно. Если у организации нет уверенности в том, что разработка через тестирование улучшит качество продукта, то время, потраченное на написание тестов, может рассматриваться как потраченное впустую.
- Модульные тесты, создаваемые при разработке через тестирование, обычно пишутся теми же, кто пишет тестируемый код. Если разработчик неправильно истолковал требования к приложению, и тест, и тестируемый модуль будут содержать ошибку.
- Большое количество используемых тестов могут создать ложное ощущение надежности, приводящее к меньшему количеству действий по контролю качества.
- Тесты сами по себе являются источником накладных расходов. Плохо написанные тесты, например, содержат жёстко вшитые строки с сообщениями об ошибках или подвержены ошибкам, дороги при поддержке. Чтобы упростить поддержку тестов следует повторно использовать сообщения об ошибках из тестируемого кода.
- Уровень покрытия тестами, получаемый в результате разработки через тестирование, не может быть легко получен впоследствии. Исходные тесты становятся всё более ценными с течением времени. Если неудачные архитектура, дизайн или стратегия тестирования приводят к большому количеству не пройденных тестов, важно их все исправить в индивидуальном порядке. Простое удаление, отключение или поспешное изменение их может привести к не обнаруживаемым пробелам в покрытии тестами.

BDD

В подходе BDD нет ничего нового или революционного. Это просто эволюционное ответвление подхода TDD, где слово "тест" заменено словом "должен". Если отложить в сторону слова, то многие найдут понятие "должен" более естественным для процесса разработки, чем понятие "тест". Мышление в терминах функциональности (того, что код должен делать), приводит к подходу, когда сначала пишутся классы для проверки спецификации, которые, в свою очередь, могут оказаться очень эффективным инструментом реализации.

Суть BDD — в описании системы архитектуры приложения в терминах эксперта предметной области, а не программиста, что позволяет ускорить процесс получения обратной связи и убрать традиционные языковые барьеры между создателями ПО и его пользователями.

С помощью BDD тестировать систему (или, как сейчас принято говорить, описывать сценарии взаимодействия) может не только сам программист, пишущий код, но и PM, не разбирающийся в деталях реализации, но хорошо знающий систему с точки зрения пользователя. Для новичков BDD-скрипты — самый простой и натуральный пусть ознакомиться с документацией проекта.

В последнее время BDD применяется чаще в вебе, по большей части из-за того, что его модель сценариев органично вписывается в принцип «запрос-ответ».

Принципы ТДР

Разработка через тестирование-это методология разработки программного обеспечения, который по существу означает, что для каждой единицы программного обеспечения, разработчик программного обеспечения должен:

- определить набор тестов для группы *первой*;
- затем реализовать блок;
- наконец, убедитесь, что выполнение устройства делает тесты успешно.

Данное определение довольно неспецифическая, что позволяет использовать тесты в условиях высокого уровня программных требований, низкоуровневые технические детали или что-нибудь между ними. Один способ смотреть на ТДР, следовательно, заключается в том, что это продолжение разработки по tdd, который делает выбор более конкретным, чем по tdd.

Разработки на основе поведения указывает, что испытания любого элемента программного обеспечения должны быть определены в терминах желаемого поведения блока. заимствования из гибкой разработки программного обеспечения "желаемое поведение" в данном случае состоит из требований бизнеса — то есть желаемого поведения, который имеет ценность для бизнеса по каким-предприятие введено в эксплуатацию программное обеспечение узла под строительство. В ТДР практике это называется ДИСМОРФОФОБИЕЙ будучи "снаружи-в" активности.

Плюсы и минусы

Идея написания спецификаций на «естественном языке» манит своей внешней красотой и простотой. Мысль о том, что не умеющий программировать product owner станет сам рисовать Fitnesse-таблички и писать Cucumber-спецификации, выглядит очень привлекательно, возникает надежда переложить на него часть работы. Более того, исполнимые спецификации можно использовать как направляющие для разработки, и наряду с *test driven development* возникают подходы с похожими названиями — *Behavior driven development* и даже *acceptance test driven development*.

Однако здесь есть два больших подводных камня.

Помните бородатый анекдот про морскую свинку, которая, вопреки своему названию, и не плавает, и не хрюкает? Когда я слышу про автоматизированное приёмочное тестирование в контексте agile, у меня всегда возникает ассоциация с этой морской свинкой. Автоматизированное? Да, с этим не поспоришь. Но при этом и не приёмочное, и не тестирование. Для тестирования это слишком просто, «программирование в табличках» — адская пытка, паттерн given-when-then не даёт возможности сделать хоть сколько-нибудь сложные автоматизированные тесты, а при ручном тестировании он и вовсе не нужен. Ну а идея автоматизировать приёмку вообще слабо вписывается в концепцию agile: если «приёмочные тесты» будут пройдены, а *product owner* недоволен — продукт будет считаться успешно сданным или нет?

Второй подводный камень связан с большими скрытыми (или по крайней мере замалчиваемыми) затратами. Да, спецификации на «естественном языке» выглядят красиво. Но, увы, работа по реализации фикстур и шагов сценариев — это уже чистое программирование, причём объём и сложность этой работы в разы превышает размеры табличек и спецификаций. И это немедленно разрушает ореол простоты.

Так стоит ли вообще вкладывать усилия в эту деятельность? Кому BDD и ATDD приносит пользу — заказчику, программистам, тестировщикам? Как разрабатывать тесты, чтобы потраченные усилия всё же не пропали даром? Я постараюсь дать свои ответы на эти вопросы, и с удовольствием выслушаю вашу точку зрения.

Инструменты модульного тестирования. Continues Integration.

Модульное тестирование, или **юнит-тестирование** (англ. *unit testing*) — процесс в программировании, позволяющий проверить на корректность отдельные модули исходного кода программы.

Идея состоит в том, чтобы писать тесты для каждой нетривиальной функции или метода. Это позволяет достаточно быстро проверить, не привело ли очередное изменение кода к *регрессии*, то есть к появлению ошибок в уже оттестированных местах программы, а также облегчает обнаружение и устранение таких ошибок.

Цель модульного тестирования — изолировать отдельные части программы и показать, что по отдельности эти части работоспособны.

Для каждого языка существуют свои специализированные средства модульного тестирования. Лично я использую *JUnit* для Java, *rspec* для Ruby, *NUnit* для. Net, *Midje*для Clojure и *CppUTest* для C и C++.

Впрочем, какой бы инструмент модульного тестирования вы ни выбрали, все они должны обладать набором базовых свойств и функций.

1. Тесты должны запускаться легко и быстро. Не важно, как именно это делается – при помощи плагинов IDE или простых утилит командной строки; главное, чтобы разработчики могли запускать эти тесты по своему усмотрению. Команда запуска должна быть тривиально простой. Например, я запускаю свои тесты CppUTest в TextMate простым нажатием клавиш *command+M*. Я связал эту комбинацию с запуском *makefile*, который автоматически выполняет тесты и выводит короткий однострочный отчет в том случае, если все тесты прошли успешно. IntelliJ поддерживает *JUnit* и *rspec*, поэтому от меня требуется лишь нажать кнопку. Для выполнения *NUnit* я использую плагин Resharper.

2. Программа должна выдавать четкий визуальный признак прохождения/не прохождения теста. Не важно, будет ли это зеленая полоса на графическом экране или консольное сообщение «Все тесты прошли». Суть в том, чтобы вы могли быстро и однозначно определить, что все тесты прошли. Если для определения результата тестирования приходится читать многострочный отчет или, того хуже, сравнивать выходные данные двух файлов, – условие не выполнено.

3. Программа должна выдавать четкий визуальный признак прогресса. Не важно, будет ли это графический индикатор или строка постепенно появляющихся точек – главное, чтобы пользователь видел, что процесс идет, а тестирование не остановилось и не прервалось.

4. Программа должна препятствовать взаимодействию между отдельными тестовыми сценариями. *JUnit* решает эту проблему, создавая новый экземпляр тестового класса для каждого тестового метода; таким образом предотвращается использование переменных экземпляров для взаимодействия между тестами. Другие инструменты запускают тестовые методы в случайном порядке, чтобы исключить возможную зависимость от конкретного порядка выполнения тестов. Независимо от конкретного выбора механизма, программа должна принять меры к обеспечению независимости тестов. Зависимые тесты – опасная ловушка, которую необходимо избегать.

5. Программа должна по возможности упрощать написание тестов. Например, *JUnit* предоставляет удобный API для проверки условий (assertions), а также использует рефлексию и атрибуты Java для того, чтобы отличать тестовые функции от обычных. В результате хорошие IDE могут автоматически идентифицировать тесты, а вы избавляетесь от хлопот с подготовкой тестов.

Continues Integration(Непрерывная интеграция)

В разработке программного обеспечения, непрерывная интеграция — практика частой сборки и тестирования проекта с целью выявления ошибок на ранней стадии. Непрерывная интеграция — автоматизированный процесс, в котором, как правило, используется специальное серверное ПО, отвечающее за поиск изменений в коде в системе контроля версий, сборку, развертывание и тестирование приложения.

Основные принципы

Каждое изменение должно интегрироваться

Слово continuous в термине Continuous Integration означает «непрерывный/непрекращающийся». Это означает, что в идеале сборка вашего проекта должна идти буквально все время. Каждое изменение в системе контроля версий (например, CVS) должно интегрироваться без пропусков или задержек. Организация ночных сборок — это хорошая практика, но это не continuous integration. Ведь результаты такой ночной сборки будут доступны лишь на следующий день, когда их актуальность для разработчиков уже значительно снижена. На практике довольно часто реализуют оба процесса и непрерывную интеграция и ночные сборки — более редкую интеграцию. В очень крупных проектах это требование иногда невозможно соблюсти, но интеграция каждые сутки — это предел за который не стоит уходить. Принцип непрерывной интеграции не выполним без другого условия — «Сборка должна идти быстро».

Быстрая сборка

«Сборка должна идти быстро» — точнее не более 10 минут. Если после одного небольшого коммита ваш интеграционный сервер будет уходить в 2-х часовое пыхтение на сборку, тестирование и разворачивание от этого будет мало пользы. Разработчики будут уже далеко, над решение других проблем, им будет сложно вернуться и понять причины сбоя, если таковой был. Ведь суть непрерывной интеграции в получении быстрого feedback. Вдобавок, поздний ответ с сервера может отвлечь их от другого дела.

В случае если все этапы процесса никак не удается втиснуть в приемлемые временные рамки можно разделить его на несколько частей. При каждом коммите производить лишь саму сборку и минимальный набор тестов (smoke tests), чтобы уменьшить время. А по ночам проводить полный цикл интеграции, результаты которого команда будет анализировать с утра. Но это скорее вынужденная мера, а не пример для подражания.

Сделайте тесты

Тесты просто необходимо включать в continuous integration процесс, в противном случае вы не можете быть уверены в качестве и работоспособности своего проекта. Чем тестов больше, тем лучше, в разумных пределах конечно. Основными двумя ограничителями на количество тестов будет:

- время интеграции — сборка по-прежнему должна оставаться быстрой, основное тестирование можно перенести «на ночь»,
- наличие автоматизированных тестов — не все тесты требуют автоматизации, нет смысла делать автоматизированные тесты только для самих тестов, они должны быть целесообразны.

Чем лучше ваши тесты, тем раньше находятся ошибки и раньше исправляются. Как известно, чем раньше ошибка исправлена, тем дешевле ее исправление. Это одно из основных преимуществ практики непрерывной интеграции — снижение стоимости исправления ошибок (не всех конечно). Попутно наличие хорошего набора тестов в процессе интеграции дает больше уверенности в том, что проект работает правильно.

Именно присутствие тестов одно из отличий интеграции от нажатия кнопки Build в вашей любимой IDE.

Интеграция на специальной машине

Организовывать процесс необходимо на специально выделенной машине. Такая машина по своей конфигурации и набору прикладных программ должна максимально соответствовать окружению, в котором проект будет развернут (production enviroment). Очевидно, что полного совпадения достичь практически невозможно — маловероятно, что эксплуатироваться программа будет на машине с установленными средствами сборки, тестирования и проч. Но точное совпадение версий операционных систем (и сервис паков) необходимо.

При этом это не должна быть машина разработчика или кого-то еще, это должна быть выделенная машина (можно виртуальная). Ведь зачастую проект, собранный на машине одного разработчика, не собирается на машине другого. Выделение машины для целей интеграции позволяет уменьшить риск, связанный с конфигурацией программного и аппаратного обеспечения.

Инструменты ручного функционального тестирования. Автоматизация ручного тестирования.

Ручное тестирование – это процесс поиска дефектов в работе программы, когда тестировщик проверяет работоспособность всех компонентов программы, как если бы он был пользователем. Часто, для точности проверки, тестировщик использует заранее заготовленный план тестирования, в котором отмечены наиболее важные аспекты работы программы.

Ручное тестирование – это ключевой этап разработки программного обеспечения. Тестер может не придерживаться строго плану тестирования, а отклоняться от него для более полного тестирования, приближенного к использованию программы обычным пользователем.

Крупные проекты придерживаются строгой методологии тестирования в целях выявления максимального количества дефектов. Системный подход к тестированию включает в себя несколько этапов:

1. Выбор методологии тестирования, приобретение необходимого оборудования (компьютеры, программное обеспечение), принятие людей на должность тестеров;
2. Составление тестов с описанием выполнения и ожидаемым результатом;
3. Передача наборов тестов тестерам, которые вручную выполняют тесты и записывают результаты;
4. Передача результатов тестов разработчикам в подробном докладе с описанием всех выявленных проблем для обсуждения и исправления дефектов.

Для тестирования могут быть использованы **статический** и **динамический подходы**. **Динамические подход** включает в себя запуск программного обеспечения. **Статистическое тестирование** включает в себя проверку синтаксис и другие особенности кода программы.

Тестирование может быть **функциональным** и не **функциональным**. **Функциональное тестирование** - это проверка рабочей области программного обеспечения. **Не функциональное тестирование** - проверка производительности, совместимости и безопасности тестируемой системы.

Функциональное тестирование — это тестирование ПО в целях проверки реализуемости функциональных требований, то есть способности ПО в определённых условиях решать задачи, нужные пользователям. Функциональные требования определяют, что именно делает ПО, какие задачи оно решает.

Функциональные требования включают в себя:

- Функциональная пригодность (англ. *suitability*).
- Точность (англ. *accuracy*).
- Способность к взаимодействию (англ. *interoperability*).
- Соответствие стандартам и правилам (англ. *compliance*).
- Защищённость (англ. *security*).

Наиболее популярными инструментами функционального тестирования являются:

1. **HP Quicktest Professional** (QuickTest Pro или QTP) – это основной инструмент автоматизации функционального тестирования компании Hewlett-Packard (Mercury Interactive). Инструмент может автоматизировать функциональные и регрессионные тесты через запись действий пользователя при работе с тестируемым приложением, а потом исполнять записанные действия с целью проверки работоспособности ПО. Используется для тестирования веб-приложений, клиент-серверных приложений, .NET-приложений, Java-приложений. (HP также разработало уже начинающее устаревать средство WinRunner; оно отличается тем, что приходится работать с кодом, написанным на специальном языке TSL).

2. **Testing Anywhere** – это универсальный и легкий в использовании инструмент автоматизации, который позволяет автоматизировать разные типы тестирования. Запись GUI сценариев и простота использования без особых навыков программирования дают значительное преимущество в создании тестов, а встроенный редактор тестов позволяет модифицировать записанные тесты.

Программный продукт является платным, предназначен для тестирования веб- и windows-приложений, поддерживает разные платформы: Windows 2000, Windows 2003 Server, Windows XP, Windows Vista, Windows 2008 Server, Windows 7 и выше.
Язык тестов: визуальное проектирование.
Поддерживаемые технологии: VB.NET, C#, C++, Win32, VB6, AJAX, ActiveX, JavaScript, HTML, Delphi, Java, Perl, 32 bit apps, 64 bit apps, Oracle Forms, PHP, Python, Macromedia Flash 1.0-8.0, Adobe Flash 9.0 & later.
Кроме функционального, выполняет также модульное и нагрузочное тестирование.

3. **Canoo WebTest** – это бесплатный мультиплатформенный **Open Source** инструмент для автоматизации тестирования веб-приложений.
   Canoo Webtest – "легкий" инструмент автоматизации, созданный исключительно для тестирования веб-приложений и веб-сайтов. Под словом "легкий" подразумевается несколько понятий: размер инструмента, доступность в обучении, высокая скорость выполнения скриптов. Скрипты представляют из себя XML-файлы, имеющие определенный набор шагов – функций. Функции, как правило, простые и понятные, сложные алгоритмы не предусмотрены.
   Достоинства инструмента:
   - Позволяет получить рабочие тесты за очень короткое время.
   - Скрипты легко читаемы и доступны для понимания даже новичку.
   - Скрипты выполняются быстро, т. к. Webtest напрямую общается с тестируемым веб-приложением, минуя браузеры.
   - Хороший репортинг. Хорошо проработанный репорт дает возможность практически мгновенно определять состояние теста и увидеть в чем ошибка.
   - Может запускаться где угодно. WebTest написан на Java и работает везде, где может использоваться JDK.
   - Запуск без открытия браузера.
   - Легко расширяется.
   - Легко интегрируется.

   Минусы:

   - При использовании XML-скриптов нельзя сделать сложные тесты с циклами, ветвлениями и т.п. (для этого надо использовать связку Groovy-Webtest).
   - Не поддерживаются приложения или страницы на flash.
   - Если ваше приложение содержит сложные javascript вставки, то как минимум часть функционала вы не сможете покрыть.
   - Не принимает плохо сформированый HTML.

ПоддерживаемыеОС: (Mac OS, Microsoft Windows, Linux, Solaris, Unix.)

4. **Ranorex** – это инструмент автоматизации тестирования графических интерфейсов Windows и веб-приложений (в т. ч. .NET-приложения, Java-приложения). Основная функциональность данного инструмента – распознавание GUI-объектов.
   Продукт является платным.
   В состав **Ranorex** входит приложение **Ranorex Spy**, который исследует и анализирует объекты приложений. **Ranorex** Spy может распознать объекты для следующий технологий:
   - .NET, WPF (framework versions 1.1, 2.0, 3.5);
   - Win32 applications (MFC, Delphi);
   - Infragistics, DevExpress, QT;
   - Java SWT;
   - Adobe Flash/Flex.

   Ranorex Recorder в составе продукта обладает значительно расширенной функцией записи и воспроизведения.

   Технические характеристики программы:

   Поддерживаемые технологии: .NET (C#, VB.NET), WPF (XAML), Win32 MFC, QT, Java (SWT), Web technologies, (AJAX, Javascript, Adobe Flash/Flex, Silverlight).

ПоддерживаемыеОС: Windows 2000, Windows 2003 Server, Windows XP, Windows Vista, Windows 2008 Server, Windows 7 ивыше.

Языктестов: C#, VB.NET, Python (IronPython).

5. **TestPlan** – бесплатный инструмент, может выполнять тесты на FireFox, Internet Explorer или без запуска браузера. TestPlan 1.0 выпущен с лицензией GPLv3.

6. **Selenium** – это набор инструментов для автоматизации тестирования. В основе Selenium лежит среда для тестирования веб-приложений, реализованная на JavaScript и выполняющая проверки непосредственно средствами браузера, что в некоторой степени гарантирует адекватность такого тестирования. В рамках проекта Selenium выпускается 4 инструмента, каждый из которых имеет свои особенности и область применения:
   - *Selenium Core* – среда выполнения тестов, которая выполняет требуемые тесты непосредственно в браузере, тем самым полностью эмулируя действия пользователя.
   - *Selenium IDE* – реализовано в виде расширения к браузеру Firefox. Позволяет записывать, редактировать и отлаживать тесты. Selenium IDE включает в себя так же среду выполнения тестов Selenium Core.
   - *Selenium RC* – инструмент для создания и интеграции автоматических тестов практически на любом из популярных языков программирования.
   - *Selenium GRID* – инструмент для синхронного и контролируемого запуска тестов на разных платформах и машинах.

   Программный продукт обладает характеристиками:

   Поддерживаемые технологии: DHML, JavaScript, Ajax.

   ПоддерживаемыеОС: Mac OS, Microsoft Windows, Linux, Solaris.

   Языктестов: HTML, Java, C#, Perl, PHP, Python, и Ruby.

   Тестируемые приложения: веб-приложения.

7. **Rational Robot** – уже почти устаревшее средство от компании IBM с весьма неудобным языком. Предназначено для функционального тестирования на основе объектно-ориентированной технологии. С помощью Robot тестируются свойства объектов, в том числе и скрытые. С его помощью можно протестировать логику работы приложения через пользовательский интерфейс (использование GUI-скриптов), а также выполнить нагрузочное тестирование (VU – virtual users скрипты).
   Помимо Robot, компания IBM предлагает целое семейство продуктов для тестирования: Rational Functional Tester (включает Rational Manual Tester – средство проведения ручного тестирования), Rational Perfomance Tester, Rational XDE Tester, Rational Purify Plus, Quantify, Rational Test Manager, Rational Clear Quest.

8. **Borland (Segue) SilkTest** – средство, предоставляет широкие возможности для ручной работы со стандартными и нестандартными объектами на объектно-ориентированном языке 4Test.
   SilkTest – это универсальная среда тестирования, но в большинстве случаев SilkTest используют для GUI тестирования приложения в режиме “черного ящика”.
   Программа платная.
   Поддерживаемые технологии: DHTML, XML, Microsoft HTC/HTA, Java, Java GUI , Web-браузеры, .NET GUI, Microsoft .NETCLR 1.x, 2.0, VisualBasic6/Active X, PowerBuilder 9.0, 10.0, 10.2 и 10.5, Win32, MFC, Motif (в Solaris и Linux), SAP GUI 6.2 и выше, пользовательские объекты GUI.
   ПоддерживаемыеОС: Microsoft Windows 98 SE, Windows ME, Windows 2000, Windows XP или Windows Server 2003, Red Hat Enterprise Linux WS 2.1 или 3.0, Sun Solaris 9 или 10.

Автоматизированное тестирование программного обеспечения — часть процесса тестирования на этапе контроля качества в процессе разработки программного обеспечения. Оно использует программные средства для выполнения тестов и проверки результатов выполнения, что помогает сократить время тестирования и упростить его процесс.

С автоматизацией тестирования, как и со многими другими узконаправленными IT - дисциплинами, связано много неверных представлений. Для того, чтобы избежать неэффективного применения автоматизации, следует обходить ее недостатки и максимально использовать преимущества. Далее мы перечислим и дадим небольшое описание для основных нюансов автоматизации и дадим ответ на основной вопрос данной статьи – когда автоматизацию всё-таки стоит применять.

Преимущества автоматизации тестирования:

- Повторяемость – все написанные тесты всегда будут выполняться однообразно, то есть исключен «человеческий фактор». Тестировщик не пропустит тест по неосторожности и ничего не напутает в результатах.
- Быстрое выполнение – автоматизированному скрипту не нужно сверяться с инструкциями и документациями, это сильно экономит время выполнения.
- Меньшие затраты на поддержку – когда автоматические скрипты уже написаны, на их поддержку и анализ результатов требуется, как правило, меньшее время чем на проведение того же объема тестирования вручную.
- Отчеты – автоматически рассылаемые и сохраняемые отчеты о результатах тестирования.
- Выполнение без вмешательства – во время выполнения тестов инженер-тестировщик может заниматься другими полезными делами, или тесты могут выполняться в нерабочее время (этот метод предпочтительнее, так как нагрузка на локальные сети ночью снижена).

Недостатки автоматизации тестирования (их тоже немало):

- Повторяемость – все написанные тесты всегда будут выполняться однообразно. Это одновременно является и недостатком, так как тестировщик, выполняя тест вручную, может обратить внимание на некоторые детали и, проведя несколько дополнительных операций, найти дефект. Скрипт этого сделать не может.
- Затраты на поддержку – несмотря на то, что в случае автоматизированных тестов они меньше, чем затраты на ручное тестирование того же функционала – они все же есть. Чем чаще изменяется приложение, тем они выше.
- Большие затраты на разработку – разработка автоматизированных тестов — это сложный процесс, так как фактически идет разработка приложения, которое тестирует другое приложение. В сложных автоматизированных тестах также есть фреймворки, утилиты, библиотеки и прочее. Естественно, все это нужно тестировать и отлаживать, а это требует времени.
- Стоимость инструмента для автоматизации – в случае если используется лицензионное ПО, его стоимость может быть достаточно высока. Свободно распространяемые инструменты как правило отличаются более скромным функционалом и меньшим удобством работы.

- Пропуск мелких ошибок - автоматический скрипт может пропускать мелкие ошибки, на проверку которых он не запрограммирован. Это могут быть неточности в позиционировании окон, ошибки в надписях, которые не проверяются, ошибки контролов и форм с которыми не осуществляется взаимодействие во время выполнения скрипта.

Для того чтобы принять решение о целесообразности автоматизации приложения нужно ответить на вопрос «перевешивают ли в нашем случае преимущества?» - хотя бы для некоторой функциональности нашего приложения. Если вы не можете найти таких частей, либо недостатки в вашем случае неприемлемы – от автоматизации стоит воздержаться.

При принятии решения стоит помнить, что альтернатива – это ручное тестирование, у которого есть свои недостатки.

Инструменты нагрузочного тестирования

Нагрузочное тестирование – определение или сбор показателей производительности и времени отклика программно-технической системы или устройства в ответ на внешний запрос с целью установления соответствия требованиям, предъявляемым к данной системе (устройству).

Зачем производится нагрузочное тестирование:

- Проверка и оптимизация конфигурации оборудования, виртуальных машин, серверного программного обеспечения;
- Оценка максимальной производительности, которую способен выдерживать проект с типовыми сценариями нагрузки на доступных ресурсах;
- Влияние модулей проекта на производительность, сценарии обработки пиковой нагрузки;
- Оценка стабильности при максимальных нагрузках при проведении 24-часовых тестов с учетом внешних факторов (импорты, резервное копирование и т.п.);
- Выявление ограничений конфигурации, определение методов дальнейшего масштабирования и оптимизации.

Вообще, существует огромное количество инструментов для нагрузочного тестирования, как **opensource**, так и коммерческих. Остановимся на наиболее часто используемых и расскажем об их основных возможностях.

Инструменты:

1. Apache HTTP server benchmarking tool (Бесплатный)

Самый часто используемый, т.к входит в состав Apache.

```
ab  [options]  [http[s]://]hostname[:port]/path
```

где основные необходимые **options**:

-c *concurrency* - количество одновременных запросов к серверу (по умолчанию 1);
-n *requests* - общее количество запросов (по умолчанию 1).

Плюсы:

- Есть везде, где есть Apache;
- Не требует никакой дополнительной настройки;
- Очень простой инструмент.

Минусы:

- Очень простой инструмент;
- Тестирует только производительность веб-сервера: опрашивает только один URL, не поддерживает сценарии нагрузки, невозможно имитировать пользовательскую нагрузку и оценить работоспособность проекта со всех сторон - как с точки зрения инфраструктуры, так и с точки зрения разработки.

2. Joe Dog Siege (Бесплатный)

Немного сложнее **ab** и необходимые задачи выполняет гораздо лучше.

В файле-сценарии задаются URL-ы и запросы тестирования. Если сценарий большой по объему, то можно вынести все запросы в отдельный файл и в команде указать этот файл при тестировании:

В команде указывается количество пользователей **-c**, количество повторений **-r** и задержку между хитами**-d**.

Результат можно выводить в log-файл или сразу в консоль в режиме реального времени:

Также можно взять из access-log веб-сервера URL-ы, по которым ходили реальные пользователи и эмулировать нагрузку реальных пользователей.

Плюсы:

- Многопоточный;
- Можно задавать как количество запросов, так и продолжительность (время) тестирования - т.е можно эмулировать пользовательскую нагрузку;
- Поддерживает простейшие сценарии

Минусы:

- Ресурсоемкий;

- Мало статистических данных и не очень хорошо эмулирует такие пользовательские сценарии, как ограничение скорости запросов пользователя;
- Не подходит для серьезного и масштабного тестирования в сотни и тысячи потоков, т.к он сам по себе ресурсоемкий, а на большом количестве запросов и потоков очень сильно нагружает систему.

3. Apache JMeter(Бесплатный)

Основные возможности:

- Написан на Java;
- HTTP, HTTPS, SOAP, Database via JDBC, LDAP, SMTP(S), POP3(S), IMAP(S);
- Консоль и GUI;
- Распределенное тестирование;
- План тестирования – XML-файл;
- Может обрабатывать лог веб-сервера как план тестирования;
- Визуализация результатов в GUI.

Результаты выводятся в графическом виде:

Плюсы:

- Кроссплатформенный, т.к написан на Java;
- Очень гибкий, используется много протоколов, не только веб-сервер, но и базы;
- Управляется через консоль и gui интерфейс;
- Использование напрямую логов веб-сервера Apache и Nginx в качестве сценария с возможностью варьирования нагрузки по этим профилям;
- Достаточно удобный и мощный инструмент.

Минусы:

- Ресурсоемкий;
- На длительных и тяжелых тестах часто падает по разным причинам;
- Стабильная работа зависит от окружения и конфигурации сервера.

4. Tsung(Бесплатный)

Основные возможности:

- Написан на Erlang;
- HTTP, WebDAV, SOAP, PostgreSQL, MySQL, LDAP, Jabber/XMPP;
- Консоль (GUI через сторонний плагин);
- Распределенное тестирование (миллионы пользователей);
- Фазы тестирования;
- План тестирования – XML;
- Запись плана с помощью Tsung recorder’a;
- Мониторинг тестируемых серверов (Erlang, munin, SNMP);
- Инструменты для генерации статистики и графиков из логов работы.

С помощью собственных скриптов, которые обрабатывают логи работы, можно выводить различные отчеты по тестированию:

- **tsung_stats.pl**
- **tsung_stats.pl + Gnuplot:**

Плюсы:

- Т.к на писан на Erlang, то хорошо масштабируется, зависит от выделяемых ресурсов;
- Может выполнять роль распределенной системы, на большом количестве машин;
- Большое количество тестируемых систем - не только веб-серверы и БД, но и, к примеру, XMPP-сервер: может отправлять сообщения, тесты с авторизацией;
- Управление через консоль, но, благодаря поддержке плагинов, можно управлять и с помощью стороннего плагина с gui-интерфейсом;
- Наличие в комплекте инструмента Tsung Recorder - своего рода, proxy-сервер, через который можно ходить по тестируемому сайту и записывать сразу как профиль нагрузки;
- Генерация различных графиков тестирования с помощью скриптов.

Минусы:

- Нет gui-интерфейса;
- Только *nix системы.

5. WAPT(Платный)

Основные возможности:

- Windows
- Платный (есть триал на 30 дней / 20 виртуальных пользователей);
- Запись плана тестирования из десктопных и мобильных браузеров;

- Зависимости в планах тестирования (последующий URL в зависимости от ответа сервера);
- Имитации реальных пользователей (задержки между соединениями, ограничение скорости соединений).

Отчет можно вывести как таблицей, так и графиком:

Плюсы:

- Очень гибкий, большое количество настроек и тестов;
- Эмуляция медленных каналов соединений пользователей;
- Подключение модулей;
- Запись сценариев тестирования прямо из браузера, как с десктопного, так и с мобильного;
- Генерация различных графиков тестирования с помощью скриптов.

Минусы:

- Доступен только для Windows;
- Платный.

6. Инструменты тестирования в продуктах «1С-Битрикс»(Входит в лицензию продукта)

Основные возможности:

- Простой и понятный функционал, доступен сразу из административного интерфейса продукта «1С-Битрикс»;
- Задается количество потоков, можно изменять количество потоков в процессе теста;
- Удобен для быстрых сценариев по проверке текущей конфигурации сервера.

Отчет о тестировании выводится в виде таблицы и графиков:

Понятие о DevOps. Варианты автоматизации развертывания. Docker и Ansible.

1. Что такое DevOps и откуда оно взялось?

Термином «DevOps» обычно называют возникшее профессиональное движение, которое выступает за совместные рабочие отношения между разработчиками и ИТ-подразделением, в результате получая более быстрое выполнение планируемых работ (например, высокие темпы развертывания), одновременно увеличивая надежность, стабильность, устойчивость и безопасность production-среды. Почему разработчики и ИТ-подразделения (далее для краткости админы)? Потому что, как правило, поток ценностей (value stream) находится между бизнесом (где определяются требования) и заказчиком (куда поставляется результат).

Считается, что движение DevOps зародилось в 2009 году, как объединение многочисленных смежных и взаимодополняющих сообществ: Velocity Conference, на которой особенно яркими было выступление «10 Deploys A Day» John Allspaw и Paul Hammond, подход «инфраструктура как код»(Mark Burgess и Luke Kanies)," Agile infrastructure "(Andrew Shafer) и системное администрирование в Agile (Patrick DeBois), подход Lean Startup Эрика Райса с его непрерывной интеграции, а так же широкая доступность облачных и PaaS технологий.

Один из соавторов DevOps Cookbook, Джон Уиллис, написал фантастический пост об этом событии.

2. Чем DevOps отличаются от Agile?

Одним из принципов Agile является предоставление рабочего программного обеспечения меньшими и более частыми релизами, в отличие от подхода «большого взрыва», который присущ водопадной методологии. Так что Agile стремятся иметь потенциально готовый продукт в конце каждого спринта (как правило, раз в две недели).

Высокие темпы развертывания приводят к тому, что перед админами накапливается большая гора задач. Clyde Logue, основатель StreamStep, говорит об этом так: «Agile сыграл важную роль в разработке для восстановления доверия у бизнеса, но он нечаянно оставил IT Operations позади. DevOps это способ восстановления доверия ко всей ИТ-организации в целом ».

DevOps особенно хорошо дополняет Agile, так как он расширяет и дополняет процессы непрерывной интеграции и выпуска продукта, давая уверенность в том, что код готов к выпуску и несет ценность для клиента.

DevOps позволяет сформировать гораздо более непрерывный поток работы в ИТ-подразделение. Если код не поставляется в прод, в том виде как он был разработан (например, разработчики поставляют код каждые две недели, но развертывается он только один раз в два месяца), операции по установке будут накапливаться перед админами, клиенты не получат важный функционал для себя и сам процесс установки в таком случае приводит к хаосу и дезорганизации.

DevOps, в том числе, изменяет культуру, так же как изменяет процессы и метрике разработчиков и админов.

3. Каковы принципы DevOps?

В DevOps Cookbook и The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win, мы описываем лежащие в основе принципы, с помощью которых все DevOps паттерны могут быть получены с помощью подхода «Три пути». Они описывают ценности и философию, которые являются основой процессов, процедур, практик, а также обязательных шагов.

Первый Путь подчеркивает производительность всей системы в целом, в отличие от производительности отдельного звена или отдела — это может быть, как большое подразделение (например, разработка или ИТ отдел) так и отдельные люди (например, разработчик, системный администратор).

В центре внимания находится все бизнес-потоки по созданию ценности, которые включены в IT. Другими словами, он начинается тогда, когда определяются основные требования (например, для бизнес или ИТ), они

закончены в разработке, а затем перешли в ИТ-отдел, в которых ценность сервиса затем и доставляется заказчику в виде сервиса.

Результаты следования Первому Пути на практике состоят в том, что известные баги никогда не передаются на следующий этап работ, никогда не развивается локальная оптимизация, приводящая к созданию глобальной деградации, происходит непрерывное улучшение и стремление достичь глубокого понимания системы (в соответствии с Демингом).

Второй Путь заключается в создании петли обратной связи, идущей справа налево. Целью практически любой инициативы по совершенствования процесса является сокращение и усиление обратной связь, чтобы необходимые поправки могли внедряться постоянно.

Итоги Второго Пути: понимание и реакция на всех клиентов, как внутренних, так и внешних, сокращение и усиление всех петель обратной связи, и углубление знаний о среде там, где это нужно.

Третий путь заключается в создании культуры, которая влияет на две вещи: постоянное экспериментирование, которое требует принятия рисков и извлечение уроков из успехов и неудач, а также понимание того, что повторения и практики являются предпосылкой к мастерству.

Нам нужны оба этих принципа в равной мере. Эксперименты и принятие рисков, является тем, что гарантирует, что мы продолжим улучшения, даже если это означает, что мы можем зайти в слишком опасные дебри. И мы должны учиться навыками, которые могут помочь нам выйти из опасной зоны, когда мы зашли слишком далеко.

Итоги Третьего Пути включают выделение времени для улучшения повседневной работы, создание ритуалов, которые поощряют команду в принятии рисков и возможному созданию неисправностей в системе с целью повышения устойчивости в перспективе.

4. Области внедрения DevOps

В книге „DevOps Cookbook“, мы выделили 4 моделей внедрения DevOps:

- **Модель 1**: Углубление процессов разработки в поставку: это включает расширение непрерывной интеграции и выпуска на боевые сервера, интеграция тестирования и информзащиты в рабочие процессы, что дает готовый к поставке код, настроенные среды, и так далее.
- **Модель 2**: Создание обратной связи от прода до разработки: включает создание полной хронологии событий в разработке и администрировании, с целью помощи в разрешении проблем, а также предоставление доступа команде разработки к анализу проблем на проде, одновременно с созданием разработчиками сервисов самообслуживания, везде где это возможно, и создание информационных радиаторов, показывающих изменение в поведении системы при вносе изменений.
- **Модель 3**: Объединение разработки и администрирования: состоит во включении команды разработки в цепочку разрешения проблем, назначение разработчиков на разрешение проблем на проде, а также взаимные тренинги между разработчиками и администраторами, чтобы уменьшить количество эскалаций.
- **Модель 4**: Включение ИТ команды в разработку: состоит во включении или тесной связью между IT и разработкой, создание многоэтапных пользовательских историй (включая развертывание, управление кодом в производстве и т.д.), и определение нефункциональных требования, которые могут быть использованы во всех проектах.

5. В чем ценность DevOps?

Я полагаю, что есть три бизнес преимущества, которые организации получают от перехода на DevOps: быстрый выход на рынок (например, сокращение времени цикла и более высокие темпы развертывания), повышение качества (например, повышение доступности, меньше сбоев и т.д.), и увеличение организационной эффективности (например, больше времени тратится на деятельность, связанную с увеличением ценности продукта по сравнению с потерями, увеличение количества функционала, переданного заказчику).

Быстрое время выхода на рынок

В 2007 году в IT Process Institute, мы тестировали более 1500 ИТ организаций и пришли к выводу, что высокопроизводительные ИТ организации были в среднем на 5-7х порядков более производительными, чем их не высокопроизводительные сверстники. Они делали в 14 раз больше изменений, получая проблемы только половине случаев, при этом имея скорость исправления проблем с первого раза в 4 раза выше, а также имели в 10 раз более короткие периоды простоя. У них было в 4 раза меньше результатов повторного аудита, они в 5 раз чаще находили нарушения за счет автоматизированной системы внутреннего контроля, и в 8 раз лучше укладывались в сроки проекта.

Одной из характеристик высокопроизводительных команд состоит в том, что они уходят далеко вперед от толпы. Другими словами, лучшее становится еще лучше. Это происходит в области темпов развертывания приложения. Accenture недавно провели исследование о том, что делают интернет-компании, и Amazon пошла на рекорд, заявив, что они делают более 1000 развертываний в день, поддерживая скорость успешных изменения 99,999%!

Возможность поддержания высоких темпов развертывания (например, быстрое время цикла) трансформируется в бизнес-ценность двумя основными способами: как быстро может организация перейти от идеи к чему-то, что может быть передано заказчику и сколько экспериментов организация может проводить одновременно.

Без сомнения, если я работаю в организации, где я могу сделать только одно развертывание каждые девять месяцев, и мой конкурент может сделать 10 установок в день, у меня есть значительные, структурные конкурентные недостатки.

Высокие темпы развертывания позволяют проводить эксперимент быстро и практически непрерывно. Скотт Кук, основатель Intuit, был одним из самых ярых сторонников „безудержной инновационной культуры“ на всех уровнях организации. Один из моих любимых примеров приводится ниже:

»Каждый сотрудник [должен быть в состоянии] проводить быстрые, высоко-скоростные эксперименты… Дэн Маурер руководит нашим потребительским подразделением, в том числе запуск веб-сайта TurboTax. Когда он начинал работу над этим проектом, мы сделали около семи экспериментов за год. За счет внедрения инновационной культуры, они сейчас делают 165 экспериментов в течение трех месяцев налогового сезона. Бизнес результат? Коэффициент конверсии веб-сайта составляет до 50 процентов. Результат для сотрудников? Люди просто любят его, потому что теперь их идеи могут попасть рынок ".

Для меня самой шокирующей частью истории Скотта Кука является то, что они делали все эти эксперименты в пик сезона подачи налоговых деклараций! Большинство организаций делают остановку изменений во время пикового сезона. Но если вы можете увеличить коэффициент конверсии, и, следовательно, продаж, в пик сезона, когда ваш конкурент не может, то это подлинное конкурентное преимущество.

Предпосылки для этого включают возможность находится в состоянии сделать много мелких изменений быстро, без перерыва в обслуживании клиентов.

Уменьшение количества потерь IT отдела

Mike Orzen и я как-то говорили об огромных потерях в IT стриме, вызванными длительными сроками простоя, медлительной передачей работ, незапланированными работами и переделками. Для статьи Michael Krigsman мы оценили, сколько полезной деятельности мы могли бы вернуть путем применения принципов DevOps.

Мы подсчитали, что, если бы мы могли просто сократить вдвое количество IT потерь, а также перераспределять эти деньни таким образом, что сможем вернуться в пять раз больше чем было вложено, мы бы производили $ 3 триллионов долларов полезной деятельности в год. Это ошеломляющее количество денег и возможностей, которым мы позволяем ускользнуть из наших рук. Это 4,7 процента от ежегодного мирового ВВП, или больше, чем весь объем производства Германии.

Я думаю, это важно, особенно когда я думаю о мире, в котором будут жить трое моих детей. Потенциальное экономическое влияния на производительность труда, уровень жизни и процветание ставит это на первое место.

Однако, существует еще другой тип потерь. Люди в большинстве ИТ-организаций часто чувствуют себя обойденными вниманием и полными разочарования. Люди чувствуют, как будто они в ловушке вечно повторяющегося фильма ужасов и беспомощны, чтобы изменить финал. Менеджмент отрекается от своей ответственности за IT, что часто приводит к бесконечным войнам между разработкой, ИТ и отделами информационной безопасности. И все становится только хуже, когда аудиторы вскрывают все это. Жизнь ИТ-специалистов часто деморализована и полна разочарований, что, как правило, приводит к чувству бессилия и наполняет стрессами, которые проникают в каждый аспект жизни.

Варианты автоматизации развертывания (http://www.ibm.com/developerworks/ru/library/j-ap02109/)

Развертывание – это еще один аспект создания программного продукта, который хорошо поддается автоматизации. Это позволяет получить все преимущества надежного процесса: повысить точность, скорость и контроль над выполнением нужных действий.

Варианты автоматизации развертывания:

- помещение всех конфигурационных файлов в центральный **репозиторий**, благодаря чему можно генерировать рабочие версии приложений при помощи скриптов развертывания;
- **развертывание при помощи скриптов**, при котором все этапы развертывания выполняются специальными скриптами без вмешательства человека;
- **принцип одной команды**, упрощающий процесс развертывания и обеспечивающий его автономное выполнение (headless execution);
- **принцип токенизации конфигурации**, позволяющий использовать переменные подстановки в конфигурационных файлах;
- **принцип внешнего конфигурирования**, который позволяет *единожды* задавать параметры, которые различаются в разных средах развертывания;
- **принцип проверки по шаблону**, при помощи которого легче гарантировать единообразие свойств всех сред развертывания;
- **принцип автономного выполнения**, благодаря которому обеспечивается безопасный доступ к нескольким компьютерам в процессе автоматического развертывания;

- **принцип унифицированного развертывания**, в основе которого лежит стремление использовать один скрипт для развертывания приложения в различных средах.

- **Принцип двоичной согласованности**, гарантирующий, что ни один артефакт не будет потерян в процессе развертывания приложения в целевых окружениях.
- **Принцип одноразового контейнера**, в соответствии с которым среда для развертывания должна переводиться в заранее известное состояние. Благодаря этому снижается вероятность ошибок.
- **Принцип удаленного развертывания**, гарантирующий взаимодействие с несколькими удаленными серверами с центрального компьютера или кластера.
- **Принцип обновления базы данных**, целями которого являются организация и централизованное управление скриптовым процессом для выполнения последовательных изменений в базе данных.
- **Принцип тестирования развертывания**, в соответствии с которым перед развертыванием и после него должны выполняться специальные проверки, чтобы убедиться, что приложение работает как ожидается.
- **Принцип отката среды**, который возвращает приложение и базу данных в исходное состояние в случае неудачного развертывания.
- **Принцип защиты файлов**, предписывающий ограничивать доступ к определенным файлам, использующимся системой сборки приложения.

Docker — программное обеспечение для автоматизации развёртывания и управления приложениями в среде виртуализации на уровне операционной системы, например **LXC**(система виртуализации на уровне операционной системы для запуска нескольких изолированных экземпляров операционной системы Linux на одном узле). Позволяет «упаковать» приложение со всем его окружением и зависимостями в контейнер, который может быть перенесён на любой Linux-системе с поддержкой cgroups вядре, а также предоставляет среду по управлению контейнерами.

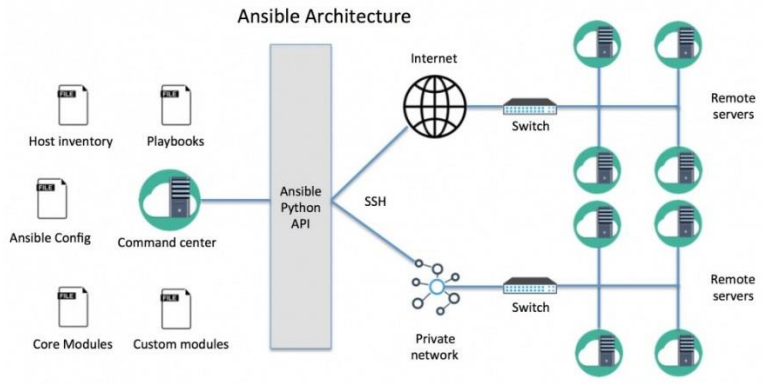
Ansible – популярный инструмент для автоматизации настройки и развертывания ИТ-инфраструктуры. Основные задачи, которые решает Ansible:

- **Управление конфигурациями**. Максимально быстрая и правильная настройка серверов до описанной конфигурации.
- **Провижнинг**. Управление процессом развертывания новых облачных серверов (например, через API, с помощью Docker или LXC).
- **Развертывание**. Инсталляция и обновление ваших приложений без простоя наилучшим образом.
- **Оркестрация**. Координация компонентов вашей инфраструктуры для выполнения развертываний. Например, проверка, что веб-сервер отключен от балансировщика нагрузки, до апгрейда ПО на сервере.
- **Мониторинг и уведомления**.
- **Логгирование**. Централизованный сбор логов.

По сравнению с другими популярными инструментами автоматизации ит-инфраструктуры, Ansible не требует установки клиентских приложений на обслуживаемые сервера, что может сократить время настройки перед развертыванием инфраструктуры. Для работы Ansible подключается к обслуживаемым серверам по SSH.

Важность подобных инструментов только увеличивается в облаке с появлением возможности быстро создавать необходимые сервера, развертывать необходимое ПО, использовать и удалять, когда необходимость отпала, оплачивая только используемые ресурсы.

Основная идея **Ansible** – наличие одного или нескольких управляющих серверов, из которых вы можете отправлять команды или наборы последовательных инструкций (playbooks) на удаленные сервера, подключаясь к ним по SSH.



Файл **Host inventory** содержит информацию об обслуживаемых серверах, где команды будут исполнены. **Файл конфигурации** Ansible может быть полезен для указания настроек вашего окружения.

Почему Ansible?

После 4 лет активного использования Chef, инфраструктура как код стала действительно утомительной для меня. Я проводил больше времени за кодом, который управлял моей инфраструктурой, но не за самой инфраструктурой. Любое изменение, не важно насколько маленькое, потребует большое количество усилий для незначительной выгоды.

С Ansible, с одной стороны есть данные описывающие инфраструктуру, с другой ограничения между взаимодействием других компонентов. Эта модель намного проще, она позволяет намного быстрее двигаться дальше, позволяя сосредоточится на том, что делает моя инфраструктура. Как и в Unix модели, Anisible предоставляет модули с единичной ответственностью, которые могут быть объедены бесконечным множеством способов.

Ansible не имеет зависимостей кроме как Python и SSH. Ей не нужно устанавливать агентов на удаленные машины, она не оставляет всякого рода мусора после своей работы. Более того, она поставляется со стандартной библиотекой модулей для управления всем чем угодно, начиная от пакетного менеджера в облаке, заканчивая базами данных.

Почему Docker?

Docker позиционирует себя как самый надежный и удобный способ развертывания процесса на машине. Это может быть все что угодно, от mysqld до redis, заканчивая Rails приложением. Таким же эффективным способом, как git снимки и распределение кода, Docker делает тоже самое с процессами. Он гарантирует, что будет предоставлено все, что нужно для запуска этого процесса, независимо от машины, на которой он запущен.

Основная, но понятная ошибка, сравнивание Docker контейнера с VM. Здесь применяется принцип единственной ответственности, запуск одного процесса на контейнер дает простоту изменяемости и поддержки. Эта модель выдержала испытание временем в философии Unix, это дает прочный фундамент для действий.

Настройка

Не покидая свой терминал, я могу получить от Ansible настроенную машину на одном из следующих хостингов: AWS, Linode, Rackspace или DigitalOcean. Если быть более конкретным, я с помощью Ansible создаю новый дроплет с 2ГБ памяти на DigitalOcean в регионе Amsterdam 2 за 1 минуту 25 секунд. В течение еще 1 минуты и 50 секунд я могу получить настроенную систему с Docker'ом и некоторыми другими установками. Теперь, имея базовую систему, я могу развернуть свое приложение. Заметьте, я не настраивал какой либо язык программирования или базу данных. Docker сам позаботился об этом.

Ansible исполняет все команды на удаленных машинах через SSH. Мои SSH ключи, лежащие в локальном ssh-agent'е будут удаленно расшарены через SSH сессию Ansible. Когда, на удаленной машине, код моего приложения будет клонирован или обновлен никаких данных для авторизации в git не потребуется, для авторизации будет использован проброшенный ssh-agent с локальной машины.

Компетенции ИТ специалистов. Развитие компетенций. Проверка компетенций.

Когда мы говорим о подготовке ИТ-специалистов, то речь обычно идет о профессиональной подготовке.

Ее оценку можно разделить на несколько составляющих:

- общетеоретические знания по информатике,
- теоретические знания в конкретной области,
- практические знания в конкретной области,
- навыки.

Все это в той или иной степени дается системой образования, а также дополнительной работой и самообучением самого специалиста. Стандартные оценки в зачетной книжке далеко не отражают детальный уровень подготовки специалиста, необходимо иметь более развернутую и детальную систему оценки профессиональных знаний. Некоторые компании уже используют такие развернутые подходы.

Профессиональная подготовка (программист баз данных, как пример):

- общетеоретические знания по информатике:
  - дискретные структуры
  - архитектура и организация ЭВМ
  - операционные системы
  - управление информацией
  - программная инженерия
  - человеко-машинное взаимодействие
- теоретические знания в конкретной области:
  - теория баз данных
  - основы программирования
  - алгоритмы и теория сложности
  - языки программирования
- практические знания в конкретной области:
  - знание Oracle 9i
- навыки:
  - опыт работы с Oracle 9i.

Система образования практически ограничивается профессиональной подготовкой. Практически в образовании не уделяется достаточного внимания ни тестированию, ни развитию профессиональных способностей.

Однако интересы бизнеса далеко не ограничиваются только профессиональной подготовкой. По существу, компания заинтересована в получении конкретных результатов деятельности (РД). Личные качества, знания и навыки определяют способности - возможные линии поведения специалиста (или потенциал компетенций). Компетенции в терминологии HR-специалистов – это набор поведенческих образцов или линий поведения. Выполнение задач (Performance) – реализация компетенций,- зависит от внутренних скрытых и явных ситуационных факторов. Это – система мотивации, корпоративная культура, взаимоотношения с коллегами, домашняя ситуация и т.п. Как эти факторы повлияют на поведение зависит от восприимчивости к ситуационным факторам. Для одного важна только зарплата, для другого признание коллег и т.п. Одни и те же действия могут дать разные результаты в зависимости от внешних ситуационных факторов (экономической ситуации, действий конкурентов и пр.)

На основе сформированного профиля компетенций возможно определить соответствующие подходы к тестированию и подбору кадров ИТ-специалистов.

Какими компетенциями помимо профессиональных должен обладать ИТ-специалист, чтобы работать успешно? Часть компетенций - это профессионально важные качества (ПВК), т.е. личностные качества, способствующие успешной работе, но не связанные с ее спецификой. ПВК не формируются на работе, люди, которые приходят в компанию уже несут их в себе, выраженными в большей или меньшей

степени. Естественно в зависимости от специфики конкретной ИТ-специальности, и от специфики данной конкретной компании этот список будет меняться. Для того, чтобы определить необходимые для данной специальности компетенции проводиться целый комплекс мероприятий, которые называются профессиографированием (job map, job disruption, job specification). Инструментарий: экспертное интервью, наблюдение, работа с нормативной документацией и др.

Следующий вопрос **“Как их протестировать?”** - Вообщем-то ничего особо сложного в этом нет, существует довольно широкий оценочный инструментарий**,** обычно обозначаемый термином assessment. Ассесмент включает в себя такие процедуры как психологическое и проф.тестирование, ситуационно-поведенческие тесты (ролевые и деловые игры, конкретные ситуации), структурированное интервью и оценку на 360 градусов. Сначала прописывается профиль компетенций под специальность, затем под этот профиль подбирается адекватный оценочный инструментарий. По все тому же известному принципу – Цена / качество. Если Вы располагаете соответствующими ресурсами, можно провести комплексную оценочную процедуру, в которой присутствует весь (или почти весь выше перечисленный оценочный инструментарий), такая процедура называется ассесмент центром).

Можно ли развивать эти качества? – конечно можно, более того обязательно нужно, между прочим – это прямая обязанность HR’ов и тренинг-менеджеров определять недостаток в компетенциях ваших ИТ-специалистов и разрабатывать для них командные и индивидуальные программы развития. Конечно, некоторые из компетенций как-то клиенториентированность, ответственность, обучаемость развивать сложно, но опять-таки можно, если это делать системно и последовательно. Инструментарий также давно известен бизнес-тренинг и бизнес-коучинг. Мы не случайно используем термин бизнес, поскольку в последнее время очень много развелось тренеров-драйверов и коучей-учителей жизни, имеющих о работе весьма приблизительное представление. Будьте внимательны при выборе ваших тренеров.

Мотивация персонала. Материальная мотивация. Нематериальная мотивация.

Мотивация персонала — один из способов повышения производительности труда. Мотивация труда персонала является ключевым направлением кадровой политикилюбого предприятия. Наиболее эффективной системой мотивации сотрудников, является «мотивация на результат». Результаты работы сотрудников определяются с помощью KPI (Ключевых показателей эффективности). KPI и мотивация персонала позволяют существенно улучшить эффективность и производительность работы компании. Большинство теоретиков систем мотивации приходили к выводу, что только мотивация на результат является совершенной системой, т.к. обосновывает бизнесу выплаты вознаграждений, а сотрудникам дает возможность получать и увеличивать доход в четкой зависимости к приложенным усилиям

Выделяют следующие виды мотивации персонала:

- Материальная мотивация
- Социальная мотивация
- Психологическая мотивация

Материальная мотивация – это количество денежных выплат работодателем сотруднику. Под материальной мотивацией понимаются исключительно денежные компенсации, которые выплачиваются наличным или безналичным расчетом:

- основной оклад (зарплата),
- премии, надбавки, оплата сверхурочной работы, процент от продаж. Какие привилегии не относятся к материальной мотивации?
- Социальные гарантии (оплата больничного листа, отпускные выплаты и проч.).
- Предоставление или компенсация деньгами соцпакета (проезд, мобильная связь, спортзал, питание и пр.).
- Компенсация за неиспользованный отпуск, командировочные.

Три задачи, которые нужно решить до составления положения о мотивации персонала

Первая задача. Предстоит проведение мониторинга окладов своих сотрудников – для понимания подходящих зарплат для таких сотрудников.

Вторая задача. Нужно уточнить, что работодатель ожидает от своих работников, насколько сотрудники соответствуют подобным требованиям и ожиданиям.

Третья задача. Формирование инструментов, способствующих сохранению в штате сотрудников и достижению поставленных целей компании.

При решении первой задачи удастся понять – действительно ли зарплата сотрудников в компании соответствует уровню рынка труда. Если же зарплаты в компании оказываются ниже по сравнению с рынком, возможны сложности в сохранении своих сотрудников. Чтобы подобная ситуация не произошла, следует пообщаться с руководством, составив график гармонизации выплат для работников. Предстоит совместная работа с бухгалтерией и отделом финансов. Благодаря такому взаимодействию удастся сохранить деятельность компании в пределах своего бюджета, возможно выявление тонкостей и различных нюансов, которые нужно учитывать.

Решение второй задачи – понимание, что работодатель желает получить от сотрудников. Справиться с этим на самом деле несложно. Для этого нужно поговорить с руководителями своей компании, линейными менеджерами, изучая финансовую отчетность организации в течение нескольких последних лет. Необходимо понять эффективность своего персонала. При наличии проблем следует задаться вопросом – что стоит предпринять для устранения проблем за счет мотивации персонала.

В частности, при общении с линейными менеджерами может быть выявлена проблема – множество рабочего времени сотрудников направлено на свои личные цели. Генеральный Директор отметил, что штат сотрудников компании расширяется, однако пропорционального роста прибыли компании не заметно. Вполне вероятно, что проблема заключается в недостаточной мотивации сотрудников для более эффективной и продуктивной работы.

Третья задача – формируются инструменты материальной мотивации, в соответствии с которыми совмещаются интересы работников и их компании. Нужно исходить из понимания – не обязательно нужно увеличивать оклад, к примеру, фиксированную часть. Можно подумать о предоставлении дополнительных выплат за достижение определенных результатов и достижение целей. Фактически, активно применяется механизм премий и надбавок. Благодаря этому отношение сотрудников к служебным обязанностям будет более мотивированным.

Материальная мотивация персонала – важная составляющая в работе компаний со своим штатом сотрудников. Здесь множество актуальных вопросов и нюансов – какую зарплату и премии предлагать? будет ли этого достаточно достойному сотруднику? Быть может, стоило предложить меньше – всё равно бы согласился? И при попытке ответить на каждый вопрос количество нюансов и сомнений только увеличивается. Это вполне нормальная ситуация в бизнесе, поэтому уделим этому вопросу особое внимание.

Положение о мотивации относится к числу основных документов для компании с указанием списка сотрудников, размера зарплат и за что их получают. Однако причина не только в этом. При грамотной и внимательной разработке данного документа может быть сформирована соответствующая оплата труда и выдача премий, когда сотрудники будут заинтересованы в максимально рациональном расходовании своего рабочего времени, мотивированы в повышении качества работы, выявлении брака, снижении расходов и совершенствовании своих навыков.

Как ранжировать персонал при использовании материальной мотивации

Первый уровень сотрудников – от которых никаким образом не зависят продажи. Им просто выплачивается оклад за проделанную работу.

Второй уровень – сотрудники обслуживающего персонала. Данная категория представлена клиентским сервисом, службой поддержки и пр. Для этих сотрудников можно предусмотреть небольшой процент с прибыли, поскольку их работа тоже в определенной мере влияет на лояльность покупателей.

Третий уровень – люди, занимающиеся продажами. Они занимаются непосредственными продажами, добиваясь прибыли для компании. Им можно устанавливать средний процент выплат – порядка 8-10% с прибыли компании.

Четвертый уровень – руководители отделов продаж. Получают сравнительно низкий оклад, но рассчитывают на большой процент. Хотя процент может быть установлен меньше, чем у продажников – к примеру, 5%. Но ведь в подчинении руководителя не один продажник, а 5 или больше сотрудников – их результаты будут суммироваться. Поэтому берется процент от общего количества продаж своего отдела.

Система нематериальной мотивации персонала – 5 основных правил создания

1. Нематериальная мотивация должна решать тактические задачи вашего бизнеса

В первую очередь, используемые стимулы должны быть направлены на решение конкретных задач, которые стоят перед вашим бизнесом. К примеру, если вы развиваете филиальную сеть, то вы должны сформировать команды, которые смогут работать по стандартам, принятым в главном офисе. Соответственно ваша нематериальная мотивация должна быть направлена на обучение ваших сотрудников, например, посещение тренингов по эффективным коммуникациям и командообразованию.

2. Нематериальная мотивация должна охватывать все категории работников

В большинстве случаев, когда мы говорим про мотивацию, упор делается на тех людей в компании или подразделения, которые приносят прибыль. Однако, не нужно забывать, что помимо них есть еще бухгалтера, секретари, производственники. К таким людям могут быть применены не только мотивационные программы, а просто признание труда, похвала.

В компаниях малого бизнеса, где руководитель наизусть знает каждого работника, зажечь огонь в глазах каждого сотрудника достаточно легко. Задача многократно усложняется, когда речь заходит о крупной фирме. Генеральный Директор уже не может воздействовать на каждого. На данном этапе в игру вступают линейные руководители, у которых в подчинении находятся небольшие группы людей, как правило, 7-10 человек. Линейные руководители постоянно общаются с людьми из своего коллектива и поэтому знают, что может стимулировать каждого.

3. Нематериальная мотивация должна учитывать этап развития компании

В небольшом семейном бизнесе главный мотиватор – энтузиазм. Когда же компания переходит на следующий этап своего развития, когда работников становится больше и часть процессов формализуется, мотивационные программы должны быть ориентированы на признание заслуг каждого работника, но также важно и учитывать возможность коллективного признания услуг, например какого-нибудь отдела или подразделения компании.

4. Правильный выбор методов нематериальной мотивации персонала

Нам часто кажется, что то, что нас мотивирует, будет мотивировать и других. Но это не так. Для того чтобы подобрать правильные методы мотивации, вам необходимо первоначально собрать информацию об истинных потребностях сотрудников. И в данном случае вам поможет пирамида потребностей Абрахама Маслоу. С ее помощью система нематериальной мотивации персонала приобретает понятный вид. Итак, важно определить какие потребности для ваших сотрудников ведущие, и разработать соответствующие факторы мотивации.

- **Физиологические потребности**. Если для сотрудника важна данная группа, то необходимо обеспечить ему комфортный уровень заработной платы.
- **Потребность в защите и безопасности**. Для таких людей важно организовать дружелюбную атмосферу в коллективе. Соответственно должна быть минимизирована информация о негативных составляющих работы: банкротство, увольнения.
- **Социальные потребности.** Для сотрудников из данной категории важно получать поддержку от коллег и руководства, также им важно постоянно находится в кругу людей.

- **Потребность в уважении и самоуважении.** Этих сотрудников нужно одаривать постоянным вниманием. Им важно осознавать, что их действия будут оценены по достоинству.
- **Потребность в самореализации.** Это главная потребность для креативных сотрудников. Таким людям важно заниматься творческой работой. Они способны решать самые сложные, нестандартные проблемы.

И помните, что любой Ваш сотрудник постоянно чего-то хочет. И когда достигнуто желаемое, то потребности переходят на более высокий уровень.

5. Эффект новизны

Поощрения не должны становиться обычным явлением, потому что единообразные мотивационные программы только угнетают ваших сотрудников. Поэтому, раз в полгода стоит придумать какую-то новую мотивационную программу.

Способы нематериальной мотивации персонала

Можно придумать большое количество различных **способов нематериальной мотивации** Ваших сотрудников, но мы постарались Вам дать только самые действенных из них. Итак, вот они.

- Мотивирующие совещания
- Конкурсы и соревнования
- Поздравления со знаменательными датами
- Скидки на услуги
- Информирование о достижениях
- Поощрительные командировки
- Оценки коллег
- Помощь в семейных делах