Введение в программную инженерию

Клименков С.В.
2019-2020 уч. год
v.1.4.0.0 от 22.06.2020

TODO list:

- четко объяснить инкрементный, итеративный и эволюционный подход к разработке.
- придумать связанные реальные примеры в риск анализе, что бы они были более демонстративными и понятными.
- 6-я глава целиком.

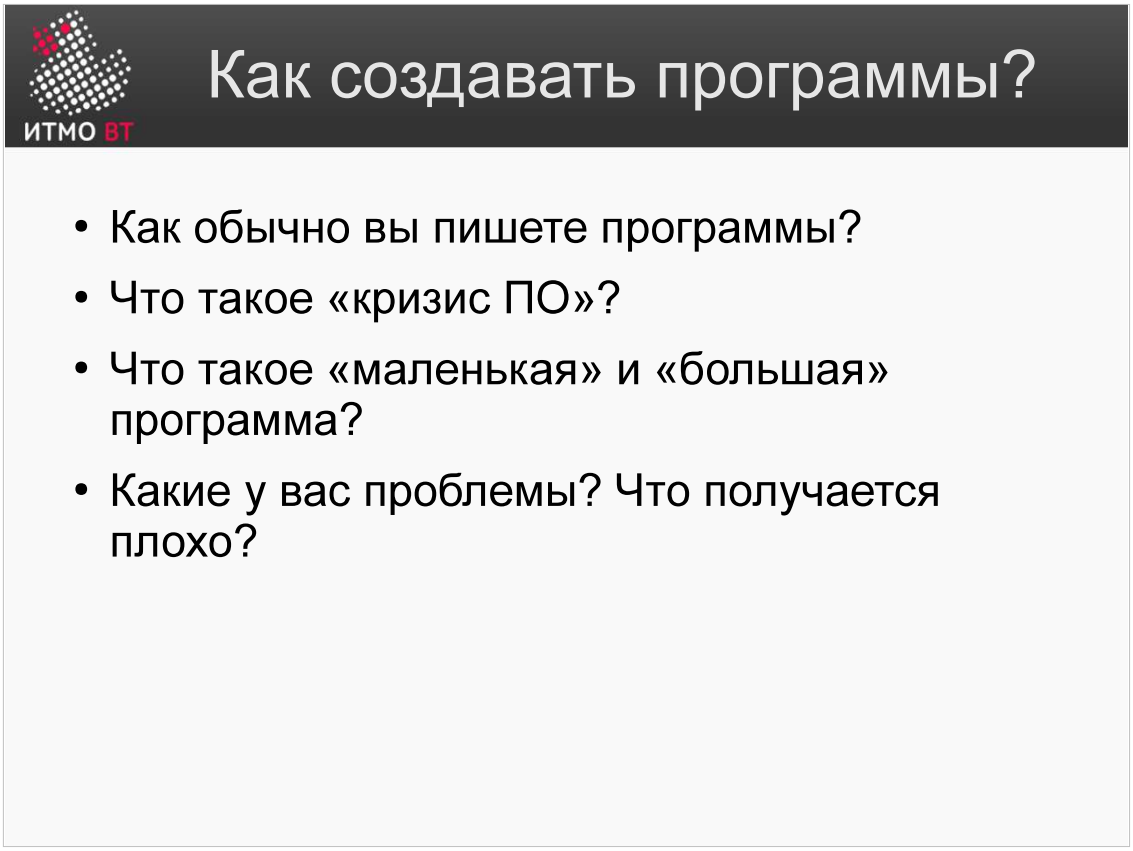
Литература

- Орлов С.А. Программная инженерия. Учебник для вузов. 5-е издание обновленное и дополненное. Стандарт третьего поколения. — СПб.:Питер, 2018. — 640 с.: ил.
- Арлоу Д., Нейштадт И. UML 2 и Унифицированный процесс. Практический объектноориентированный анализ и проектирование, 2е издание. – Пер. с англ. – СПб: Символ Плюс, 2007. – 624 с., ил
- Соммервилл И. Инженерия программного обеспечения. 6-е издание.: Пер. с англ.: - М.: Вильямс, 2002
- Вольфсон Б. Гибкие методологии разработки. СПб.: Питер, 2012. — 112 с.
- Гецци К., Джазаейри М., Мандриоли Д. Основы инженерии программного обеспечения. 2-е изд.: Пер. с англ. – СПб.: БХВ - Петербург, 2005. – 832 с.: ил.
- Брукс Ф. Мифический человеко-месяц или как создаются программные системы. СПб: Символ-Плюс, 2006. – 304 с., ил.
- Рудаков, А.В. Технология разработки программных продуктов [- 9-е изд., стер.. - М.: Академия, 2014. - 208 с.
- ГОСТ Р ИСО/МЭК 12207-2010. Информационная технология. Системная и программная инженерия. Процессы жизненного цикла программных средств

ИТМО ВТ

1

Основные концепции



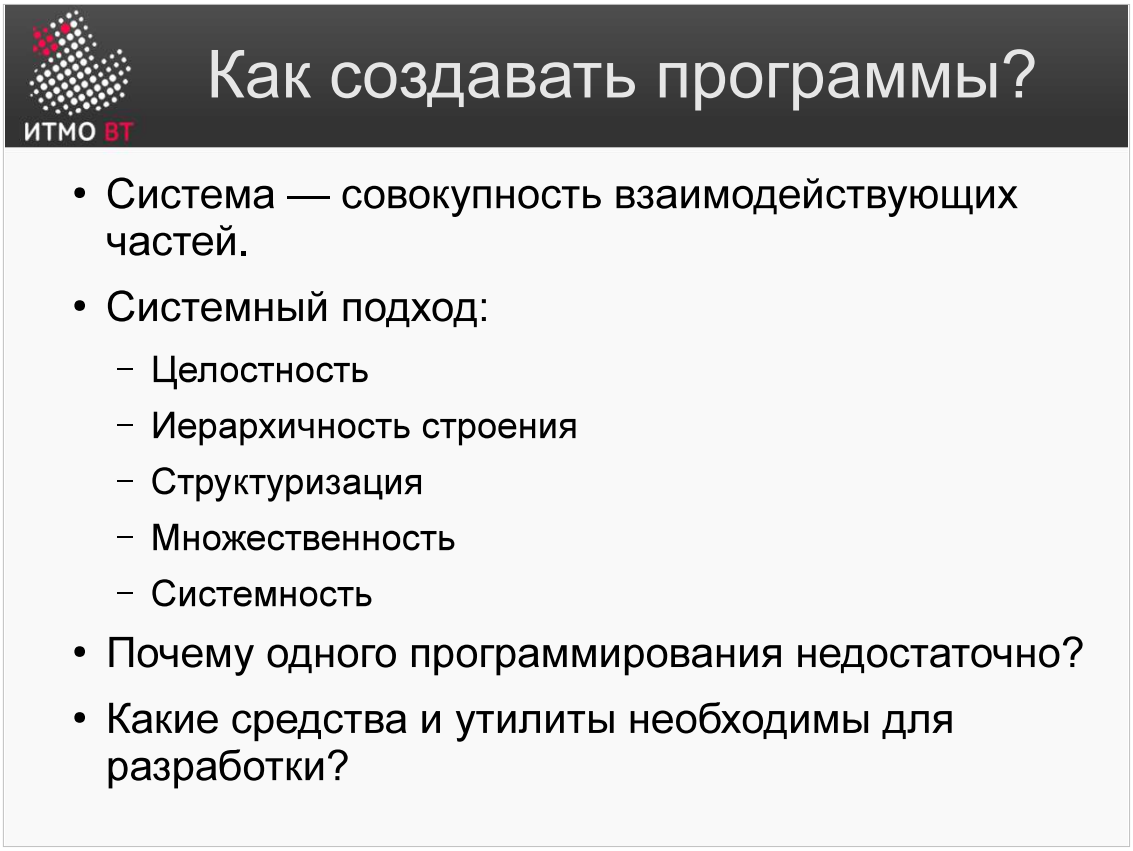
Как обычно вы пишете программы?

Вероятно, вы знакомитесь с заданием, обдумываете крупноблочную структуру, разрабатываете алгоритмы, запускаете и проверяете работоспособность программы, вводя тестовые данные и убеждаетесь либо в том, что программа работает, либо в необходимости доработок. Это методика разработки ПО по методу "code-and-fix" известна любому студенту, в ней при помощи отладки программы, мы пытаемся добиться её работоспособности, причем за счёт минимальных действий — заработало и хорошо, пора сдавать лабу!

Как оценить, какая программа является «маленькой», и у вас хватит сил ее разработать, а какая «большой», и вы потеряетесь внутри вашего кода? Что будет, если вам потребуется создать слишком большую для вас программу? Почему по мере роста программы ваших усилий становится недостаточно, чтобы её написать?

С этими вопросами вынужденно сталкиваются все начинающие программисты, которые начинают продавать свои разработки. По мере роста спроса на ваше ПО появится необходимость в повышении скорости разработки, что повлечет автоматизацию и увеличение штата сотрудников. Проблемой является то, что при увеличении количества разработчиков (что неизбежно для написания больших продуктов), начинают лавинообразно возрастать и организационные задачи. Необходим найм и увольнение сотрудников, управление их ролями и обязанностями. Брукс, в своей известной книге "Мифический человеко-месяц", наглядно показал, что увеличение штата не всегда вызывает прямо пропорциональное увеличение скорости работы, а иногда приводит и к обратному эффекту.

Все эти проблемы известны как "кризис программного обеспечения" и впервые были описана в 1960-х годах. Успех мелких команд в разработке ПО составляет 70%, а больших — 30%, и именно из-за организационных сложностей. Как попытка решения этой проблемы была создана дисциплина (точнее, комплекс дисциплин), известных как программная инженерия.



Часто в вычислительной технике можно слышать словосочетание «системный подход». Что же такое системный подход и система?

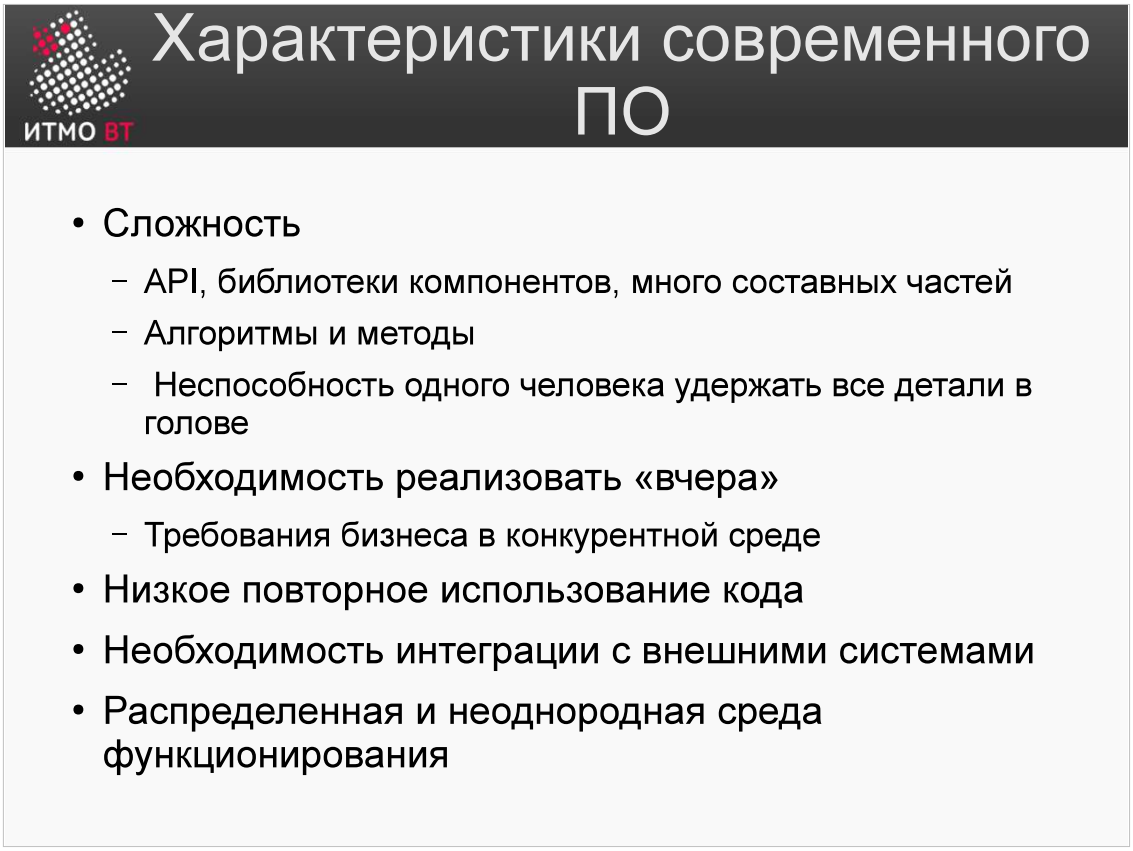
Система — совокупность взаимосвязанных и взаимодействующих элементов, образующих целостность или единство. Системный подход это такой способ организации наших действий, который выявляет закономерности и взаимосвязи элементов системы с целью её более эффективного использования. Основными принципами системного подхода являются:

- Целостность, позволяющая рассматривать одновременно систему как единое целое и в то же время как подсистему для вышестоящих уровней.
- Иерархичность строения, то есть наличие множества элементов, расположенных на основе подчинения элементов низшего уровня элементам высшего уровня.
- Структуризация, позволяющая анализировать элементы системы и их взаимосвязи в рамках конкретной организационной структуры, которая, как правило, задает процесс функционирования системы.
- Множественность, которая позволяет использовать множество кибернетических, экономических и математических моделей для описания элементов и системы в целом.
- Системность — это свойство объекта обладать всеми признаками системы.

Программное обеспечение в совокупности с аппаратными средствами вычислительной техники представляет собой сложную систему и одного программирования оказывается недостаточно. Для успешной разработки оказывается необходима организация работы самой команды, бизнес-анализ предметной области, тестирование, организация продаж и реклама, обучение, поиск финансирования, и решение других необходимых зачач, напрямую не связанная с программированием. У среднего проекта само по себе программирование составляет всего около 25% от общего объёма работ, выполняемого в процессе разработки ПО.

Для облегчения процесса разработки рутинную работу необходимо автоматизировать. Какие средства и утилиты вы используете при разработке?

В качестве примеров можно назвать интегрированные среды разработки, система контроля версий, автоматический сборщик, системы непрерывной интеграции, системы автоматического тестирования.

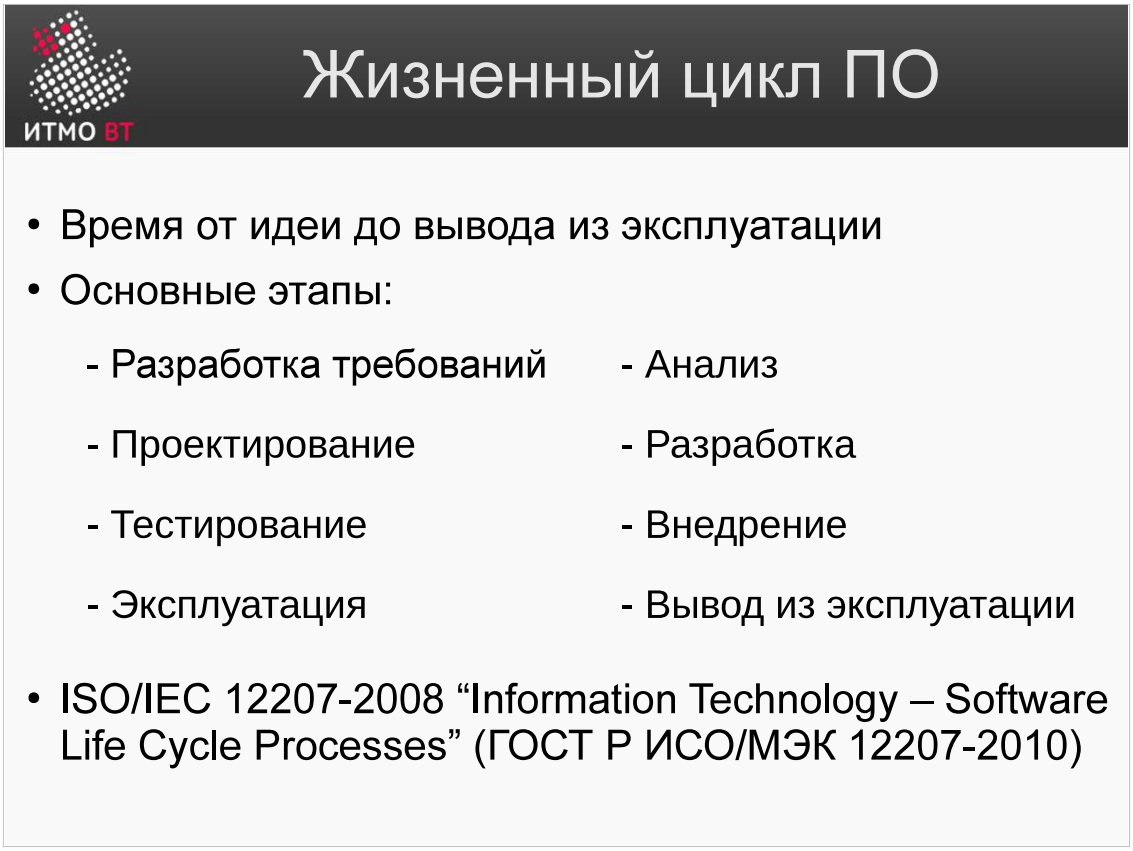


Современное ПО обладает представленными на слайде характеристиками. Основными из них являются:

- сложность ПО. Продукты изготавливаются из компонентов, и качество конечного продукта зависит от качества этих компонентов. Дополнительно на качество влияют применяемые алгоритмы и методы. Современные API (прикладные программные интерфейсы) и технологии достаточно сложны, и в одиночку человек не поспевает за их развитием. Документация для технологий и программных интерфейсов зачастую отстает от самих технологий.
- Изменения требований заказчиков и необходимость реализовать "вчера". Из-за этих причин качество труда разработчиков неизбежно снижается, так как они не успевают выработать правильную стратегию решения. Давление рынка настолько сильное, что нет времени создать правильное, вдумчиво разработанное ПО. Идет гонка, чтобы быстрее его продать. Именно поэтому в современном ПО столько дефектов. Бизнес всегда следует требованиям рынка, которые постоянно меняются, и программисты вынуждены переписывать ПО, чтобы рынок мог заработать на их труде, что определяет высокие требования к скорости работы программистов.

Проблемой также является низкое повторное использование кода, когда снова и снова разрабатывают API, принципиально ничем не отличающееся от предыдущих. Например, при смене моды на язык программирования приходится снова создавать похожие на ранее разработанные библиотеки функций. Фактически, за последние 30-40 лет в отрасли разработки ПО не появилось ничего принципиально нового, кроме, возможно, единичных технологий, таких как квантовые компьютеры.

Дополнительную сложность вносит необходимость интеграции с внешними системами. Современные системы более не существуют в одиночку, а вычисления стали распределёнными, что требует специальной организации вычислений. Например, в распределенных средах необходимо синхронизировать данные и операции. В настоящее время всё программное обеспечение всегда функционирует в неоднородной, распределённой и многопоточной среде с большим количеством запросов. Всё это накладывает дополнительные ограничения на разрабатываемое ПО.



В силу рассмотренных выше обстоятельств, современное ПО является сложным. Как же нам учитывать эту сложность?

После появления первых компьютеров и программ появилось понятие "Жизненный цикл ПО". Впервые об этом заговорили в 1956 году, когда Герберт Бенингтон[1] написал статью, где высказывал на основе своего опыта своё мнение относительно разработки больших программных систем (тогда большими считались системы от 100 000 до 1 000 000 ассемблерных строк). Он показал, что все основные задачи при разработке ПО можно свести в определенные группы или этапы и потом обсуждать характеристики задач отдельно, внутри каждого этапа.

Основные этапы разработки ПО представлены на слайде:

Разработка программного обеспечения начинается с определения требований. Требования формулирует не только сам заказчик — так как его запросы могут быть неточными, или быть неверно истолкованы, их также необходимо согласовать с разработчиками. Правильно сформированные требования очень важны как основа для дальнейшей работы, так как они позволят понять, что именно будет создано. Требования заказчика поступают к аналитикам, которые предлагают способы решения поставленной задачи.

После анализа происходит проектирование архитектуры и шаблонов реализации ПО, а затем проектное задание передается разработчикам (этап разработки). Одновременно с этим формируются подходы и производится тестирование, и, после его успешного завершения, программный продукт внедряется и эксплуатируется.

В процессе эксплуатации осуществляется поддержка пользователей, исправление дефектов и проблем ПО и обновление продукта.В конце срока эксплуатации производятся процедуры и работы, связанные с выводом ПО из использования (в том числе связанные с переходом на новую систему). Например, при смене операционной системы необходимо обеспечить сохранность накопленных файлов. Таким образом, старое ПО используется до того момента, пока им пользуется хотя бы один пользователь.

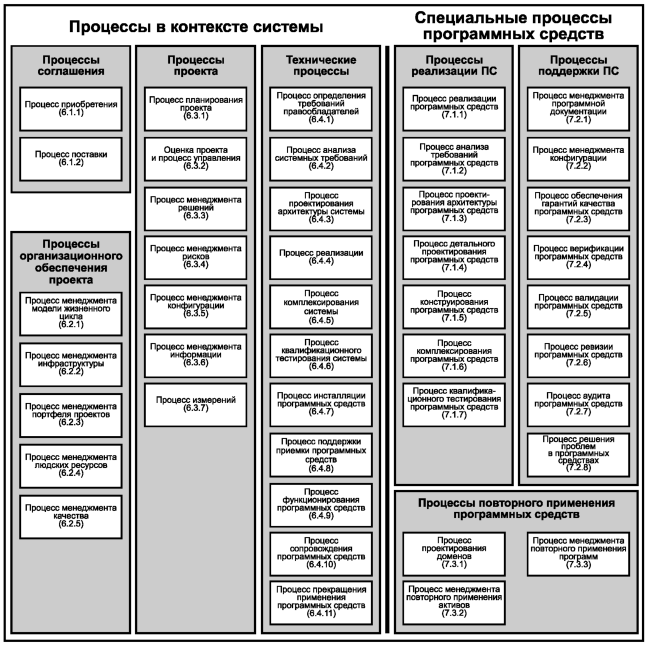
В современном мире все этапы и процессы жизненного цикла стандартизованы.

1- Benington, H.D.. Production of Large Computer Programs – Symposium on advanced programming methods fordigital computers sponsored by the Navy Mathematical Computing Advisory Panel and the Office of Naval Research in June 1956.

Группы процессов ЖЦ (ISO/IEC 12207:2010)

- Процесс ЖЦ

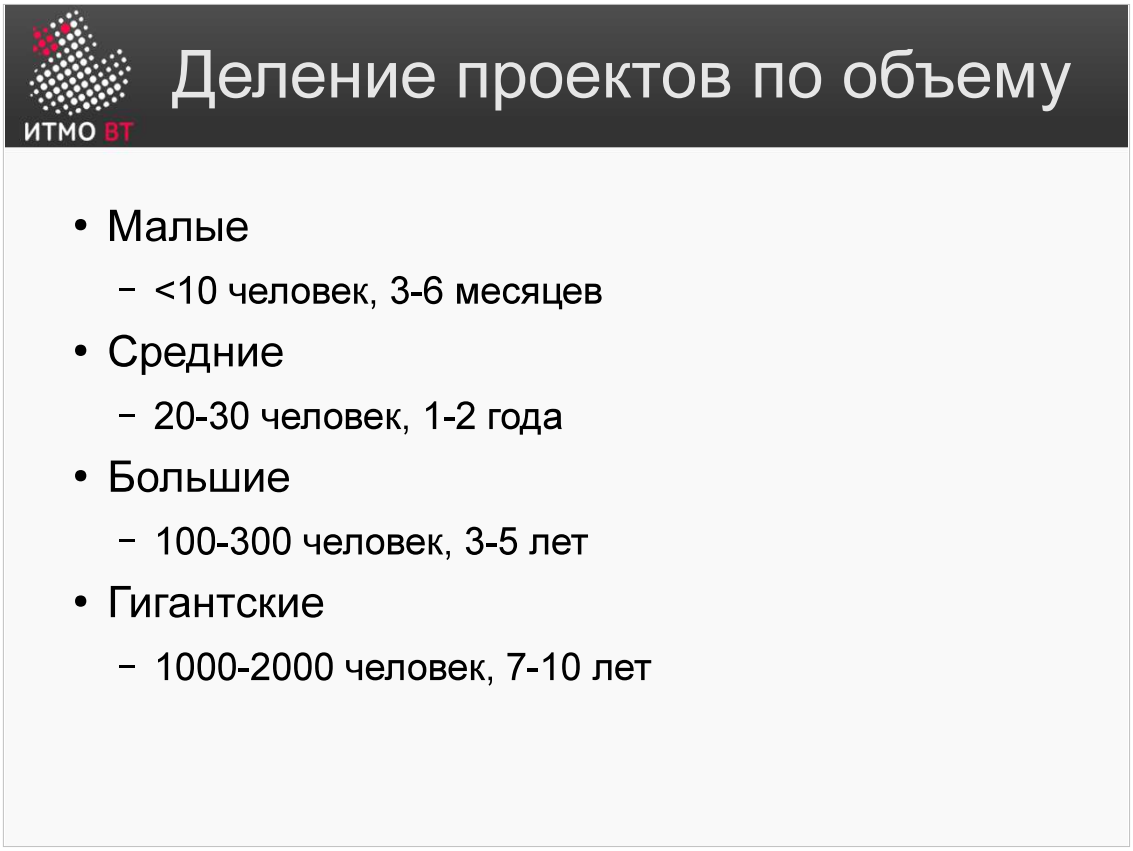
  Входные данные и ресурсы → **совокупность действий** → Выходные данные и ресурсы
- Процессы:
  - Согласования (2)
  - Орг. обеспечения (5)
  - Проектов (7)
  - Тех. процессов (11)
  - Реализации ПС (7)
  - Поддержки ПС (8)
  - Повт. исп. ПС (3)



Итак, жизненный цикл ПО — это время существования программы от момента начального замысла до окончательного вывода программы из эксплуатации. Все этапы жизненного цикла описаны в стандартах ISO, и компании, которым необходимо быть сертифицированными ISO, должны следовать этим стандартам.

Жизненный цикл ПО, согласно стандартам ISO, помимо самого написания программы, которое представлено процессами реализации программных средств, составляющих всего 7 из 43 описанных в стандарте процессов, должен включать в себя и иные процессы, включая процесс приобретения ПО, процесс регистрации изменений в ПО, процесс менеджмента, процесс повторного использования программы и т. д. Общая структура процессов вы можете увидеть на слайде.

Этих процессов много, и они сложны! Каждый процесс в стандарте ISO описывается единообразно, по схеме "Входные данные и ресурсы — совокупность действий — выходные данные и ресурсы"; при этом про совокупность действий по обработке данных и ресурсов в стандарте не указано ничего конкретного, кроме самого факта необходимости присутствия этих процессов. Входные и выходные данные могут включать требования по времени, деньгам и т.п. Любой крупной компании-разработчику следует следовать этой схеме для обеспечения надлежащего качества разработки.



Вернемся к вопросу о объеме работ, необходимым для разработки ПО. Как можно заметить из слайда, деление по объему весьма условно. Помимо объема, есть и внутренние характеристики, такие, как сложность и бюджет. Считается что:

Малым проектом является такой проект, в котором занято меньше10 человек на срок от 3-х до 6-ти месяцев). Основными тратами в разработке ПО считается труд программистов, что даст нам, с учетом средней зарплаты в 120 000 руб./месяц, расходы на разработку в месяц составят 1 200 000 руб, дав в итоге 7,2 млн. руб. за шесть месяцев без учета аренды офиса, трат на оборудование и налогов. Итого бюджет малого проекта в среднем можно оценить как 5-15 млн. руб.

Средние (20-30 человек, 1-2 года), большие (100-300 человек, 3-5 лет) и гигантские проекты (1000-2000 человек, 7-10 лет), разумеется, стоят гораздо больше.

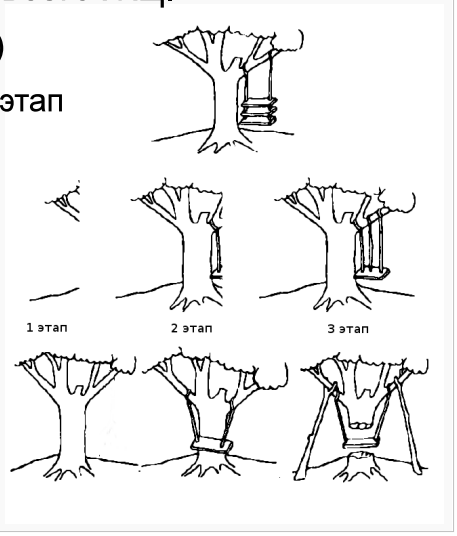
Впрочем, главным является то, что увеличение проекта по размеру не приводит к такому же росту производительности. Растут издержки, а в проектах с объемом выше среднего нужны люди, которые непосредственно разработкой программного обеспечения не занимаются (Например, директор, бухгалтер, секретари, продавцы и пр.).

Кроме того, растет и несогласованность между людьми при выполнении заданий, растет бюрократизация ежедневных задач. Это является основным тормозом работы, и скорость разработки в крупных компаниях существенно снижается.

Модели ЖЦ

Модель ЖЦ ПО – это структура, определяющая последовательность выполнения и взаимосвязи процессов, действий и задач на протяжении всего ЖЦ.

- Последовательная (однократная)
  - Определены все требования, один этап разработки
- Инкрементная
  - Определены все требования, несколько этапов
- Эволюционная
  - Определены не все требования, несколько этапов
- Формальных преобразований



За время существования программной инженерии было предложено множество различных моделей жизненного цикла ПО. Модель ЖЦ ПО – это структура, определяющая последовательность выполнения и взаимосвязи процессов, действий и задач на протяжении всего ЖЦ.

Существует несколько групп моделей жизненного цикла. Простейший из циклов — последовательный. В этом случае стадии разработки, то есть анализ, собственно разработка, тестирование и т.д., выполняются последовательно и один раз. Результаты разработки в этом случае не меняются.

Данная модель получила название водопадной или каскадной. Для такой модели легко спрогнозировать сроки, хотя, скорее всего, они все равно не совпадут с фактическими. Кроме того, легко рассчитать и предъявить заказчику стоимость.

Итак, в последовательном цикле все требования определены в самом начале, есть один большой этап разработки. Уязвимость заключается в том, что требования заказчика могут измениться, а обратная связь при такой модели появляется поздно. В то же время, такие продукты обязаны существовать — например, к ним можно отнести задачи вроде посылки миссии на Марс, где разработанное ПО предназначено для одной, конкретной модели марсохода.

В инкрементной модели ЖЦ продукт разбивается на несколько частей, у которых номенклатура функциональных требований является основой для разбиения. Важно, что части приблизительно одинаковы с архитектурной точки зрения.

В эволюционной модели ЖЦ разрабатываются прототип, который с течением времени архитектурно и функционально развивается.

В реальной разработке обычно используют инкрементно-эволюционную модель, потому что в чистом виде инкрементных или эволюционных продуктов существует мало (хотя, например, популярный в настоящее время подход к разработке SCRUM — это эволюционная модель с коротким циклом производства).

Также необходимо упомянуть о модели формальных преобразований, хотя она слабо представлена на рынке разработки. В ней создаются модели ПО, которые последовательно преобразуются друг в друга, а затем в программный код по определенным формальным принципам.

Современные подходы к разработке ПО принято называть методологией, хотя это не совсем терминологически корректно.

Метод — это последовательность шагов, которая должна привести к заданному результату.

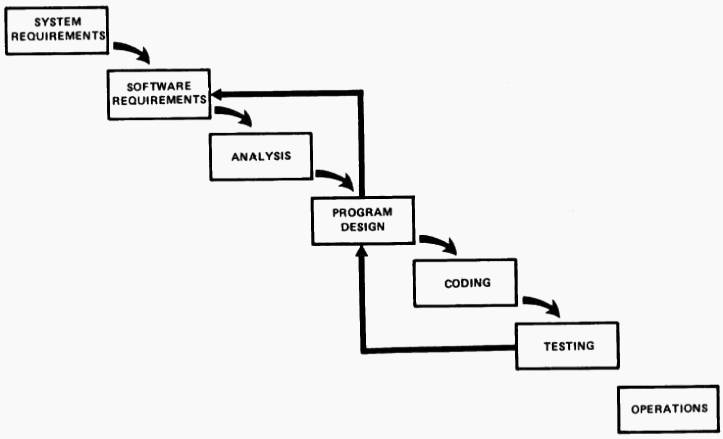
Методология — наука, изучающая методы. Тем не менее, методы, приведённые на слайде, часто называют методологиями.

На основании моделей ЖЦ ПО существует большое количество методологий разработки ПО, каждая из которых имеет свои положительные и отрицательные особенности. Число таких методологий исчисляется сотнями, поэтому далее мы приведем и опишем только те, которые внесли что-то новое в инженерию ПО. Список наиболее известных представлен на слайде.

Водопадная (каскадная) модель

ИТМО ВТ

- Разработана в 60-х, критически описана Ройсом в 70-х.



Royce W. Managing the development of large software systems

Данная модель существует в разных вариантах. Её авторство часто ошибочно приписывают Уинстону Ройсу, хотя последний лишь описал, а не создал её.

В своей статье Ройс описывает различные подходы к построению программных систем. Он последовательно выстраивает и проводит анализ нескольких моделей. Первая модель предназначена для небольших программ, она включает в себя две стадии — анализ и кодирование. Обычно такие продукты используются теми, кто их разрабатывает, и служат для внутренних целей. Во второй модели добавлены две стадии анализа требований, стадии дизайна, тестирования и эксплуатации в том порядке, в котором они выполняются. Эта модель и получила название каскадной или водопадной модели.

В следующей модели введено понятие итерации между фазами разработки, отмечается возможность отката к предыдущей фазе, но подчёркивается сложность возврата в более удаленные фазы, так как такой возврат рискован и подвержен ошибкам. Например, как показано на слайде, тестирование — это первая фаза, на которой может быть обнаружено, что ожидаемые характеристики (функциональность, производительность, объём данных и пр.) отличаются от утвержденных в результате анализа. Соответственно либо требования, либо дизайн системы должны быть изменены. Это может привести к глобальной переделке системы и двукратному (100%) увеличению сроков и стоимости её разработки.

Стандартная последовательность шагов в каскадной модели такова:
1) Определяются системные требования.
2) Определяются требования к ПО.
3) Требования анализируются.
4) Проектируется программа.
5) Разрабатывается код.
6) Проводится тестирование.
7) ПО вводится в эксплуатацию.

В своем методе Ройс предлагает взять вышеописанный подход за основу и расширить его пятью дополнительными шагами, которые преобразуют рискованный процесс разработки в процесс разработки, поставляющий требуемый продукт.

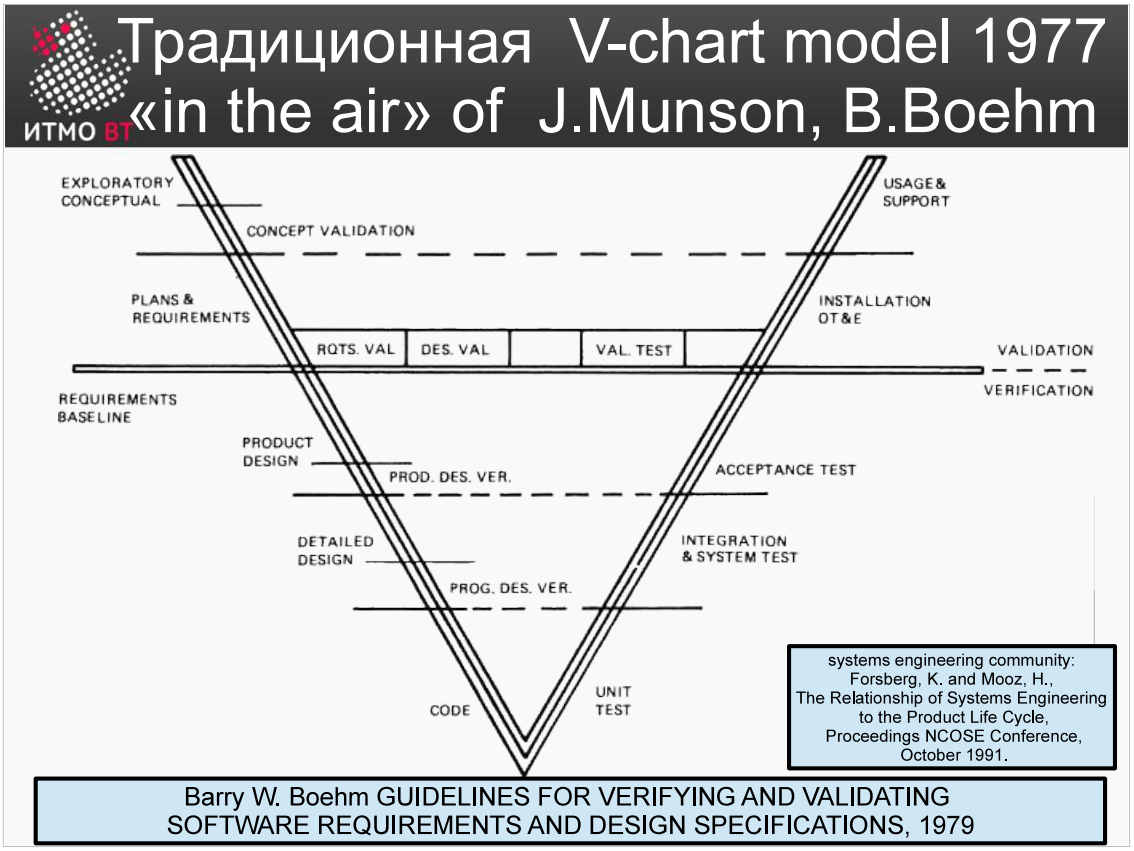
Первый шаг — "сначала дизайн программы". Ройс определил стадию предварительного дизайна системы между программными требованиями и анализом. Несмотря на невозможность учесть все требования без выполненного анализа, дизайнер программы может убедиться, что требуемые характеристики могут быть реализованы. Предварительный дизайн выполняется только дизайнерами, а не аналитиками или программистами. В нём предлагается спроектировать, определить и создать модели обработки данных, даже если они будут требовать переделок, и разработать документ — обзор будущей системы

Второй шаг — документирование дизайна. Ройс перечисляет важнейшие документы для процесса разработки ПО. Это требования к системе, спецификация предварительного дизайна, спецификация дизайна интерфейсов, финальные спецификации дизайна системы, план тестирования, инструкция по использованию.

Третий шаг — "do it twice". В нём Ройс предлагает провести симуляцию — тестовую разработку (параллельно основному процессу), и использовать её в качестве пилота с сокращённым временем разработки. Это позволит подтвердить или опровергнуть основные характеристики ПО.

Четвертый шаг — планирование, контроль и мониторинг тестирования. Ройс показывает, что тестирование является наиболее рискованной фазой с точки зрения денег и сроков, и является последней точкой, где может быть выбрана альтернатива. При планировании тестирования Ройс предлагает исключить дизайнера системы из процесса тестирования, провести "визуальную инспекцию" — повторный просмотр кода другим лицом, которое, не проводя глубокий анализ, отметит визуально заметные дефекты, протестировать каждый логический путь внутри программы (несмотря на то, что это труднореализуемо). После исправления большинства простых ошибок провести проверку (checkout) программы в необходимом тестовом окружении.

Шаг 5 — подключение пользователя. Ройс предлагает подключить пользователя на ранних этапах перед финальной поставкой продукта. В своей модели он представил три точки, где необходим опыт, оценка и подтверждение пользователем — предварительный, критический и финальный просмотр.



Сформулированные Ройсом требования к разработке ПО вызвали отклик в среде разработчиков. В тот момент много внимания уделялось качеству программного обеспечения, велись разработки конкретных методов и подходов к тестированию, поэтому многие модели того периода фокусируются именно на тестировании ПО.

Традиционная V-модель (или V-образная модель) была предложена Барри Боемом и Джэком Мансоном примерно в то же время, когда Ройс опубликовал свою статью. Боем упоминает, что "versions of the V-model came up as something that was “in the air” among Southern California aerospace companies at the time".

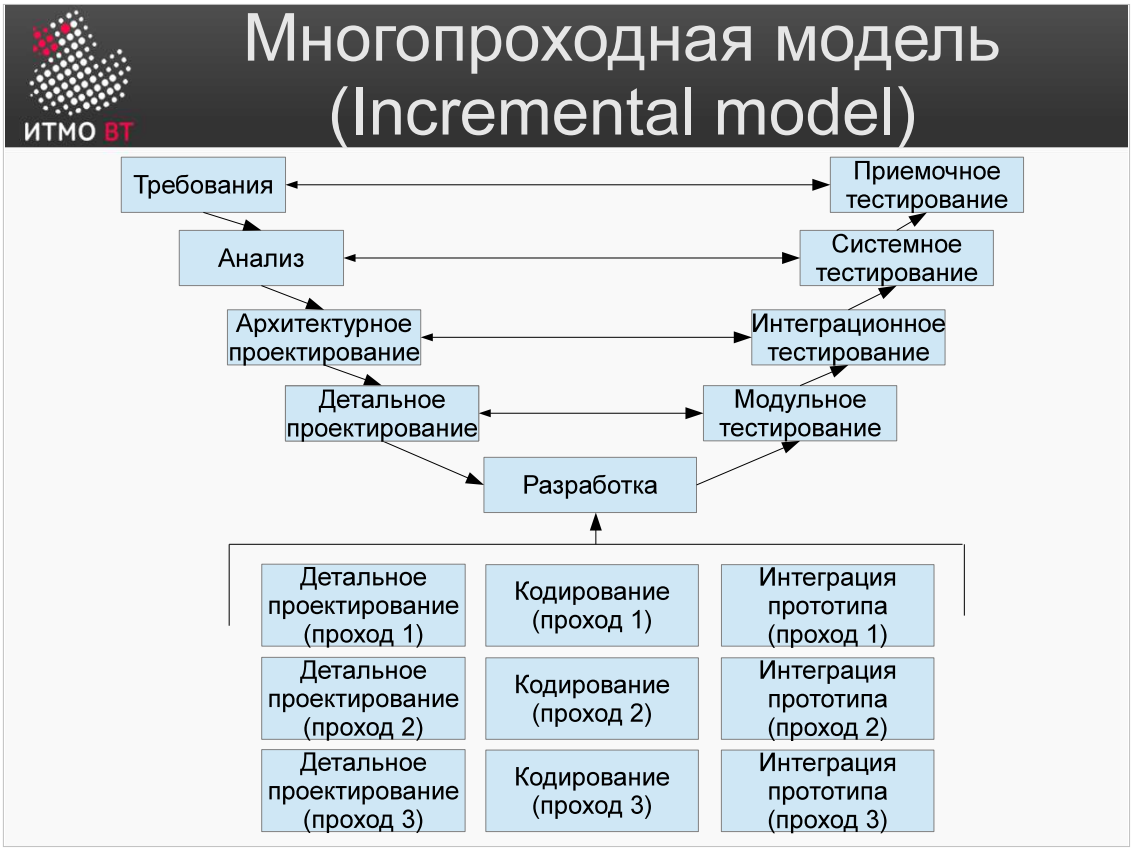
В сообществе разработчиков аппаратного обеспечения подобная модель была независимо разработана Фосбергом и Музом в 1991 году.

В основе V-модели лежит та же последовательность шагов, что и в водопадной модели, но каждому уровню разработки соответствует свой уровень тестирования. Модульное, интеграционное и системное тестирование проводятся последовательно, на основании критериев верификации, заданных соответствующими уровнями разработки. Последним этапом является приёмочное тестирование, где проверяется соответствие продукта основным функциональным требованиям.

Для проведения тестирования сначала необходимо определить корректное поведение программы, так как целью является определение соответствия поведения программы эталонному корректному поведению. Эталонное поведение бывает сложно определить, однако оно должно быть четко задано *вне кода* разработанной системы.

Динамическое тестирование (правая половина V) включает компьютерное исполнение тестов, а статическое тестирование предполагает проверку артефактов разработки без их компьютерного исполнения, например спецификации, технических решений, дизайна, и пр.

Статические тесты могут выполняться на ранней стадии проекта, что позволяет выявлять грубые ошибки в проектировании и избегать создания нефункционального или неработоспособного продукта.



Учитывая сложность изготовления продукта в один этап, было предложено разбить продукт по отдельным функциональным и техническим требованиям, а впоследствии проектировать, реализовывать и интегрировать воедино в несколько проходов разработки в виде отдельных сборок, которые стали называть инкрементами функционала или просто инкрементами.

Инкрементальный подход позволяет существенно снизить стоимость изменений требований заказчика. Разработка становится более управляемой, прогресс разработки становится видимым для заказчика, заказчик может комментировать и исправлять проектную документацию и управлять желаемой функциональностью. Более того, заказчик может использовать в своей работе частично разработанную систему. Современные гибкие (agile) методологии опираются именно на такой итеративный подход.

У инкрементного подхода есть и недостатки. Основной из них заключается в том, что архитектура системы имеет тенденцию к устареванию и деградации, и с течением времени начинает требовать рефакторинга, т.е. полной или частичной переработки, что сложно заказчику, и соответственно выполнить за счет его финансирования. Большие системы, разработка и поддержка которых осуществляются параллельно разными командами разработчиков, нуждаются в стабильной, неизменяемой архитектуре и API, которые необходимо спланировать заранее, перед разработкой основного функционала. Инкрементный подход сложен и с точки зрения управления проектом, документы по программным сборкам сложно поддерживать из-за большой скорости изменений.

Отдельной сложностью является и заключение контрактов на разработку. Традиционные подходы к бюджетированию предпочтительными полагают фиксированные суммы на разработку этапов, где сложно учесть потенциальное появление изменений, и включить почасовую оплату отработанных часов.

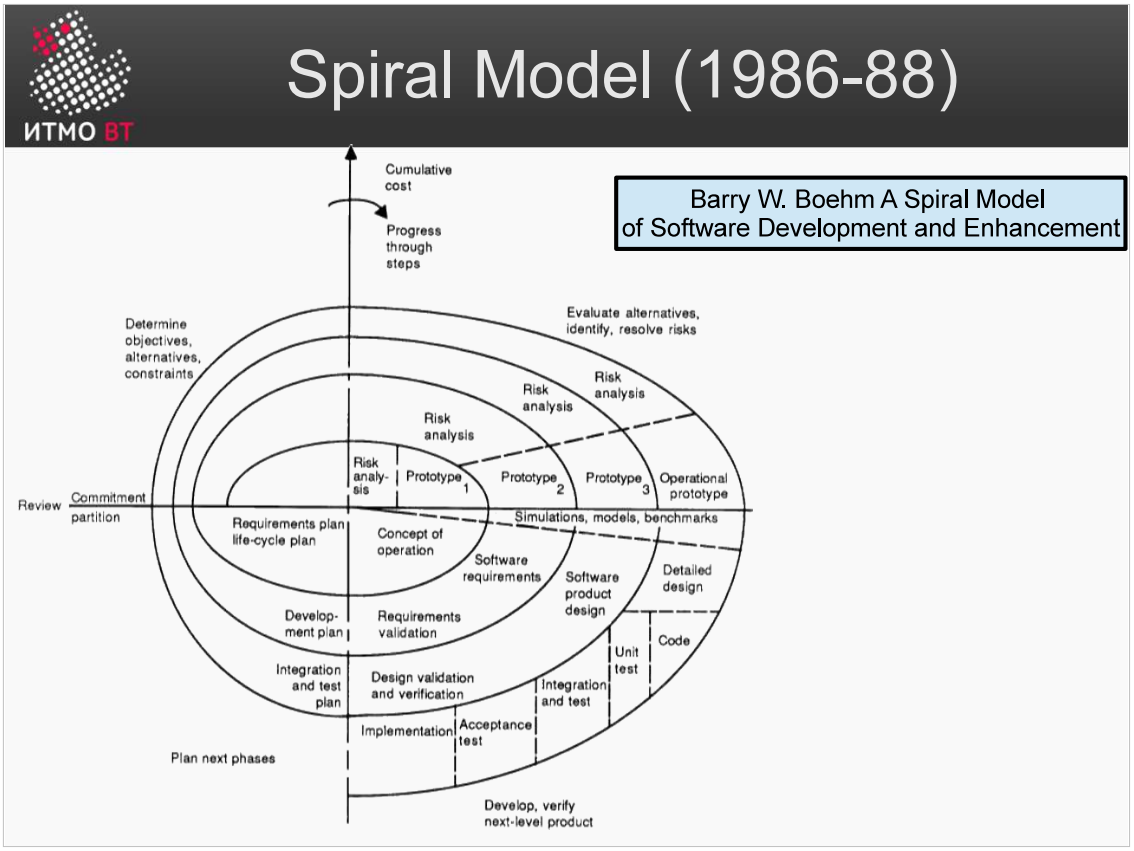
Попытка быстро оценить полезные качества разрабатываемого ПО, сократить негативное влияние архитектурных ошибок, привела к созданию подхода к разработке в виде последовательности прототипов, каждый из которых уточнял архитектуру в рамках функциональных требований. Программа в начале разработки представляет собой каркас для дальнейших наращиваний. Таким образом, ПО последовательно строится эволюционным способом.

Согласно одной из первых методик прототипирования, сначала планируется вся итерация, после этого проводится быстрый анализ текущих требований и подходов к их реализации. После этого создается база данных и интерфейс пользователя, разрабатывается функционал, и проверяется совместно с пользователями системы.

Это приводит у проверке удовлетворённости пользователей текущим прототипом и его архитектурной реализацией, тестируются основные параметры ПО. Если пользователь удовлетворён, и ему подходят выполняемые программным продуктом функции, то производится переход к разработке окончательной версии ПО.

Если нет, то создается еще один прототип, с новой БД, интерфейсом и т.д., как и предыдущий, не выполняющий всех функций, но показывающий пользователю то, как принципиально будет выглядеть работа продукта. Прототипы создаются, пока пользователь не протестирует все планируемые характеристики продукта.

Следует отметить, что в современных методах разработки применяются развитые средства макетирования, предназначеные для создания подобных прототипов. Это так называемые "мокапы" (mock-up), простые визуальные модели пользовательского интерфейса, а также более сложные UX-модели (от User Experience Design) — модели интерфейса, которые можно дать попробовать «понажимать» пользователю для оценки удобства программы.



В начале 80-х, проанализировав существующие модели, Б. Боем сформулировал метод разработки больших программных продуктов, который получил название спиральной модели.

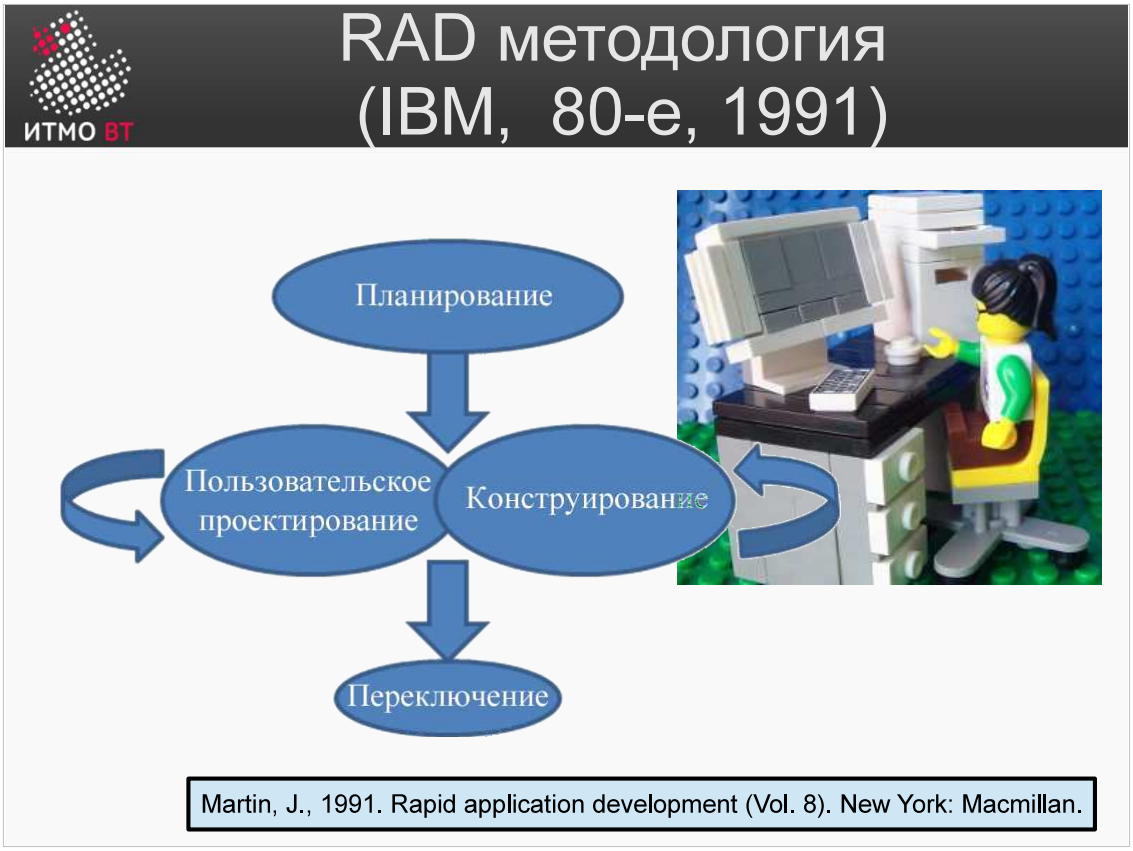
Каждый виток спирали представляет собой одну фазу разработки продукта, которая предназначена для построения очередной версии ПО или прототипа. Сначала производится постановка целей для каждой итерации, выявляются альтернативные решения и ограничения, которые необходимо учитывать в данной фазе.

Затем производится анализ рисков, за счет снижения вероятности возникновения которых минимизируется число действий, необходимых для разработки ПО. На каждой итерации спиральной модели перед созданием прототипа производится сверка с картой рисков (насколько вероятно в следующей итерации наступление того, или иного риска). При этом анализируются не только технические, но и политические и иные риски. В настоящее время анализ рисков является неотъемлемой частью любой методологии.

После анализа рисков производится разработка и валидация полученной версии ПО. На разных фазах спирали разрабатываются основные концепции ПО, функциональные требования, дизайн продукта, дизайн компонентов и их воплощение в коде программного продукта. В конце последней итерации повторяется часть V-образной модели, где производится тестирование разработанного ПО.

На последней части витка производится планирование следующей итерации. В зависимости от фазы планированию подлежат действия по разработке требований и жизненного цикла разработки, собственно сама разработка, а также действия по интеграции разработанных частей в единое целое и проведение тестирования.

Заслуга Боема состоит в том, что он включил изменения, которые вносятся в проект, как неотъемлемую часть процесса разработки, и определил, что эти изменения могут быть приняты, отклонены, или их можно игнорировать в зависимости от тех рисков, которые они несут для процесса разработки.



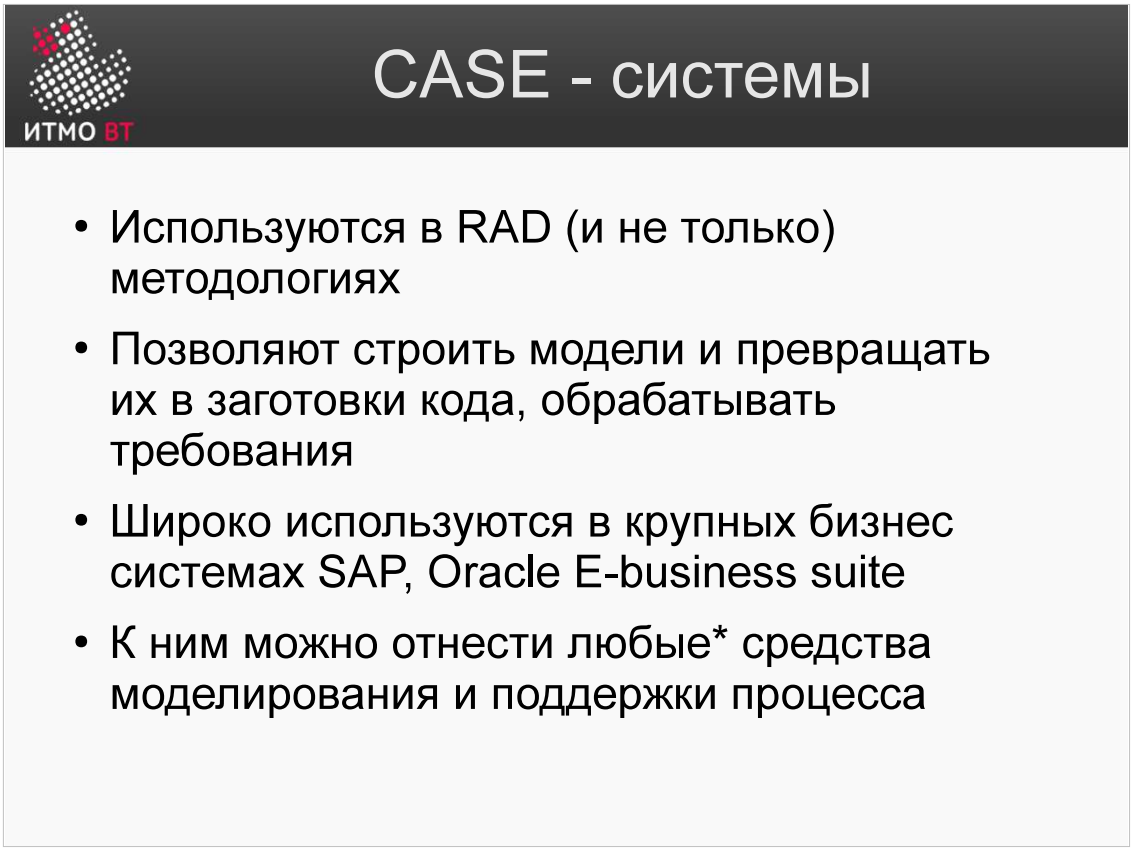
Ройс оказался абсолютно прав, указав на необходимость увеличения участия пользователя в разработке, что и показали созданные в дальнейшем методологии. Одна из широко используемых в настоящее время методологий разработки называется Rapid Application Development. Она была создана в 1980-е компанией IBM, для своих внутренних разработок, а первая подробная книга по ней вышла в 1991 году.

Предпосылками появления данной методологии (и других подобных ей) стало то, что, с одной стороны, появились средства разработки, способные автоматизировать повседневную деятельность программиста, а с другой — компании-разработчики почувствовали что созрела потребность автоматизации бизнеса крупных компаний, которые могли вкладывать существенные средства в автоматизацию своих бизнес-процессов.

До этого подобной автоматизации практически не существовало, программы писались в обычном текстовом редакторе, для подключения к базе данных требовалось формирование отдельных запросов, и невозможно было разрабатывать ПО в терминах бизнес-логики. В IBM были созданы первые прообразы современных средств автоматизации, Integrated Develpment Environment (IDE), и, как следствие, предложена методология, которая учитывает их использование и вовлечение бизнес-пользователей в разработку, благодаря чему бизнес-пользователь смог самостоятельно проектировать бизнес-функции. Методология состояла в том, что сначала проводится проектирование, а затем, при помощи средств автоматизации, как из кубиков, собирается ПО.

Пользователь при этом принимает непосредственное участие в процессе разработки, т.е. функционирование программиста и пользователя пересекаются. При помощи средств автоматизации пользователь способен создавать простейшие функции и интерфейсы. Реализация может быть не очень удобна или не отличаться продуманным интерфейсом, но реализует бизнес-функцию, которую в силу быстро меняющихся требований необходимо было разработать "уже вчера".

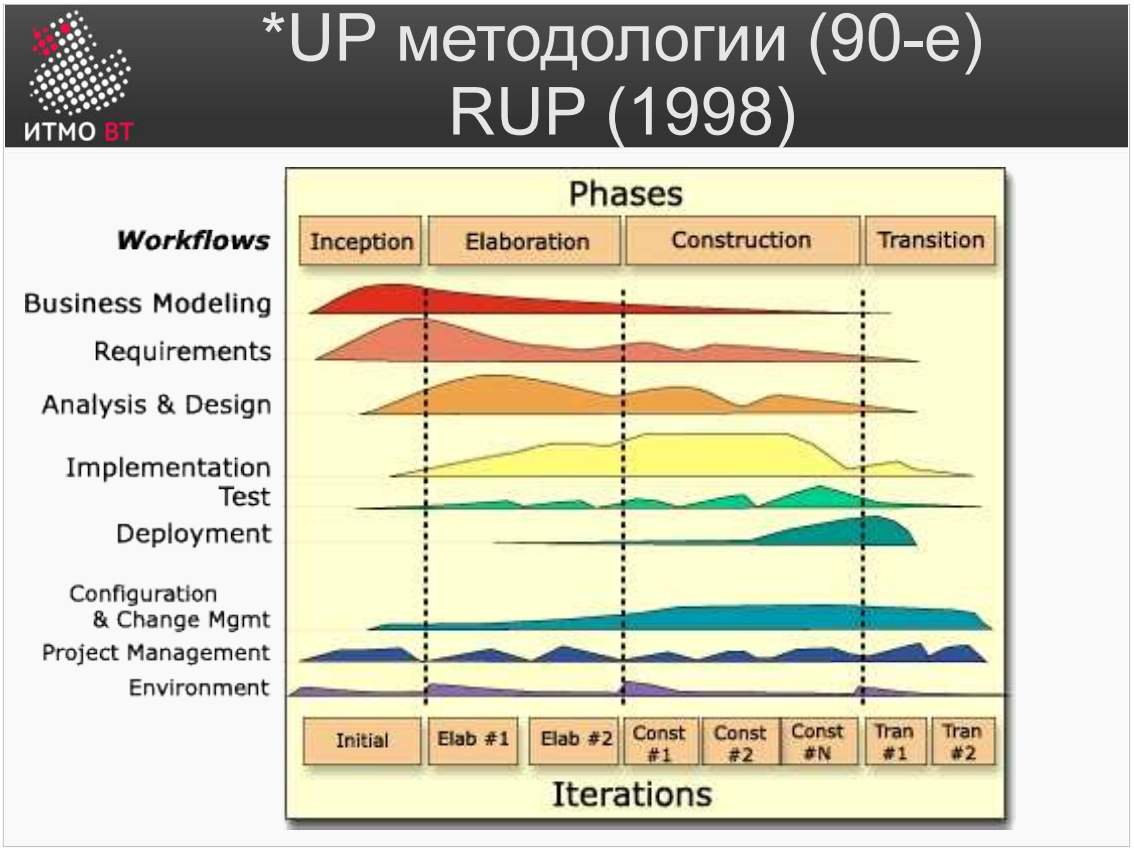
К современным средствам, предоставляющим такие возможности, относятся, например, Oracle E-Business Suite и другие крупные системы управления предприятиями. Внутри них пользователи могут, зная бизнес-логику, создавать формы данных и выполнять запросы к системе с помощью пользовательского интерфейса.



Также, в 90-е годы выяснилось, что, помимо средств разработки ПО, необходимы специализированные инструменты для работ, не связанных непосредственно с написанием кода программы, такие как построение моделей, определение требований, тестирование и т.д.). Это породило целый класс подобных систем, который принято называть CASE-системами. Это название сформировано по аналогии с CAD-системам (Computer-Aided Design), которые используются в промышленном дизайне. В программной инженерии расшифровка акронима CASE значит Computer-Aided Software Engineering или дословном переводе программная инженерия с помощью компьютера.

В большинстве случаев, такие системы полностью покрывают цикл разработки, начиная от определения требований, и заканчивая отслеживанием ошибок внутри ПО. В CASE-средствах производится автоматизированный учет и анализ требований, построение моделей и генерация первоначального кода на основании этих моделей. Кроме этого, в них интегрированы модули для взаимодействия с системами контроля версий, проведения тестирования, управления дефектами, потоком задач для разработчиков, и пр.

Одна из самых известных таких систем, существующих в настоящее время — это продукты фирмы Rational (которая была приобретена IBM). Эти продукты представляют собой полностью интегрированную инфраструктуру разработки.



В 90-е Ивар Якобсон, Гради Буч и Джеймс Рамбо объединились и создали процесс разработки Rational Unified Process (RUP). Он был построен на базе процесса OOSE компании Objectory и включил в себя элементы из процессов (OMT, OOSE и метод Буча), которыми занимались в это время его создатели.

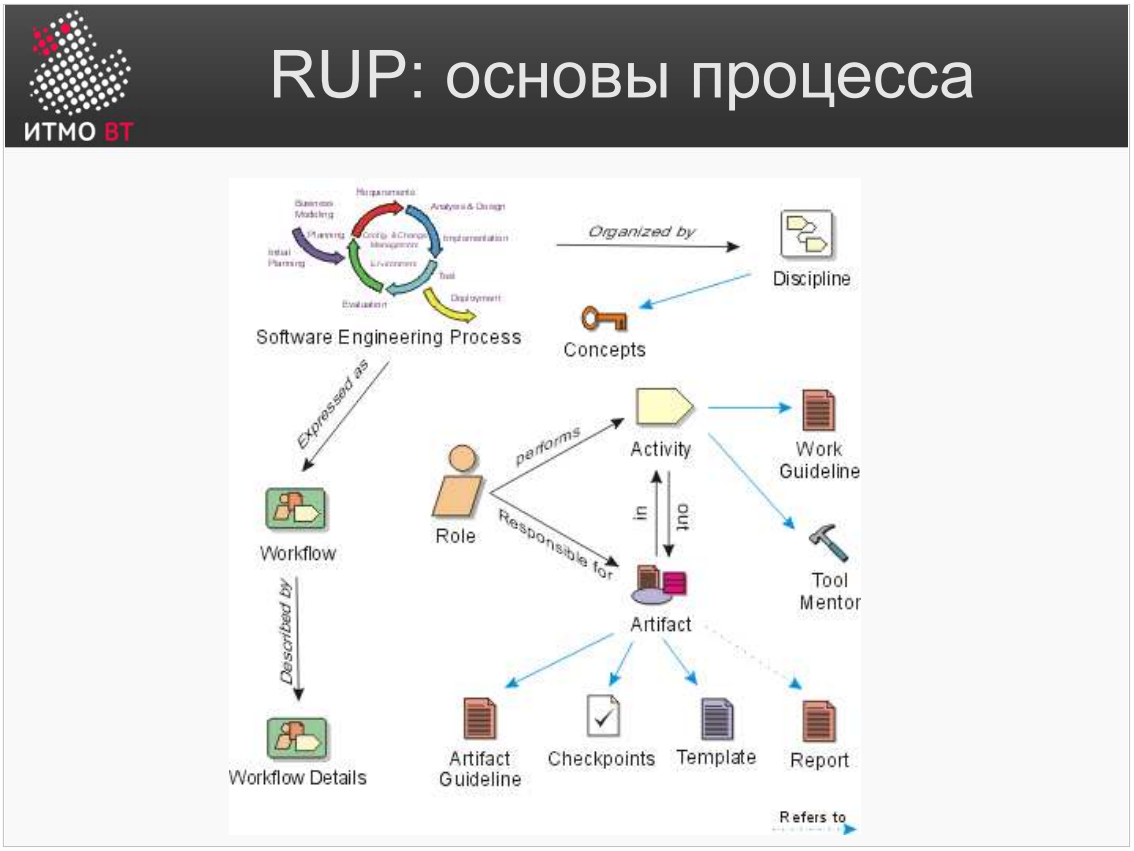
Именно в то время стала массовой и широко известной объектно-ориентированная парадигма, где картина мира представляется как совокупность взаимодействующих объектов. Также появилось много методик для создания программ с учетом объектно-ориентированного подхода. Собрав всё это воедино, «три амиго», как их называли, создали унифицированный процесс разработки, вобравший в себя все известные тогда характеристики других методов разработки.

Это процесс представлял всю разработку в виде инкрементально-эволюционного процесса. Весь процесс разработки разбит на фазы. Помимо них, в RUP также существует понятие дисциплин. Дисциплины представляют собой набор правил и указаний (как, например, арифметика, известная вам с начальной школы, формирует правила счета), которые необходимы для решения определённой задачи, например, определения требований или тестирования. Все дисциплины предназначены для того, чтобы организовать разработчиков, дать им характерные подходы, чтобы каждая определенная роль процесса разработки могла выполнять необходимые и требуемые для нее действия.

Всего фаз четыре: Inception (фаза начала), Elaboration (фаза проектирования), Construction (фаза создания продукта), и Transition (фаза внедрения продукта на стороне заказчика). График процесса показывает общий уровень активности разработчиков в каждой из фаз и в рамках каждой из дисциплин.

Например, анализируя график дятельности на слайде, можно увидеть, что работы по дисциплине Deployment активнее всего ведутся в фазах Construction и Transition, так как в это время уже существует продукт, который можно развёртывать. До этого по данной дисциплине проводилось, в основном, планирование. В дисциплине Implementation, который формирует подходы для эффективной разработки ПО, есть всплеск в фазе Transition, так как там производятся исправления продукта после размещения на стороне пользователя.

В рамках итеративного подхода фазы могут делится на итерации.



В RUP чётко определено то, как строится весь процесс. В процессе в явном виде описаны все роли, тем или иным образом связанные с созданием ПО. Всего в RUP входят около 30 различных ролей.

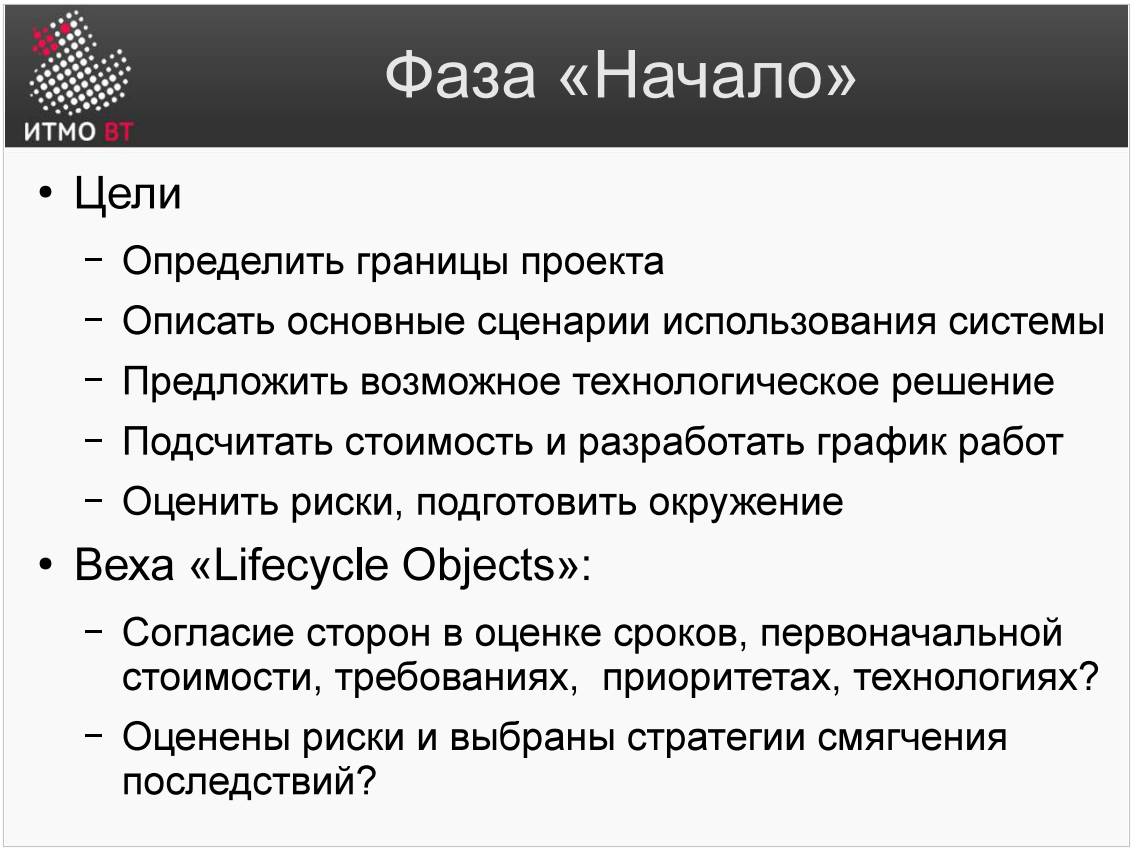
Роль — группа обязанностей, которую берет на себя участвующий в разработке для выполнения своей повседневной деятельности. Например, в роль программиста входят создание кода, фиксация изменений в системе контроля версий, разработка модульных тестов для кода, поиск справочной информации для решения возникающих проблем, получение заданий в соответствующей системе и фиксация затраченного на их выполнение времени. Таким образом, у программиста есть свой бизнес-процесс, и деятельность, входящая в этой процесс, составляет роль программиста.

У каждой роли, участвующей в разработке, есть свой набор деятельностей. На входе и выходе деятельности используются, создаются и модифицируются артефакты. Артефакт в разработке — это любой результат труда роли, например, проектный документ, графическое изображение, часть кода, исполняемый файл, и т. п. Артефакты создаются на основании инструкции (guidelines) и шаблонов, с учетом контрольных точек и других элементов процесса.

Деятельность, которая производится ролью, всегда производится согласно установленным правилам и при помощи принятых в компании средств. Компании могут изменять процессы для того, что бы они больше подходили под текущую структуру компании, RUP предлагает процессы только в качестве основы.

Итак, весь процесс описывается вокруг небольшого количества элементов, которые принимают различные значения. Эти элементы и принципы являются одинаковыми для всех ролей.

В отличие от других моделей ЖЦ и методов, в RUP каждый элемент процесса детально описан и связан с другими. Сам по себе RUP как программный продукт представляет набор связанных гиперссылками страниц HTML, каждая из которых описывает элемент процесса разработки.



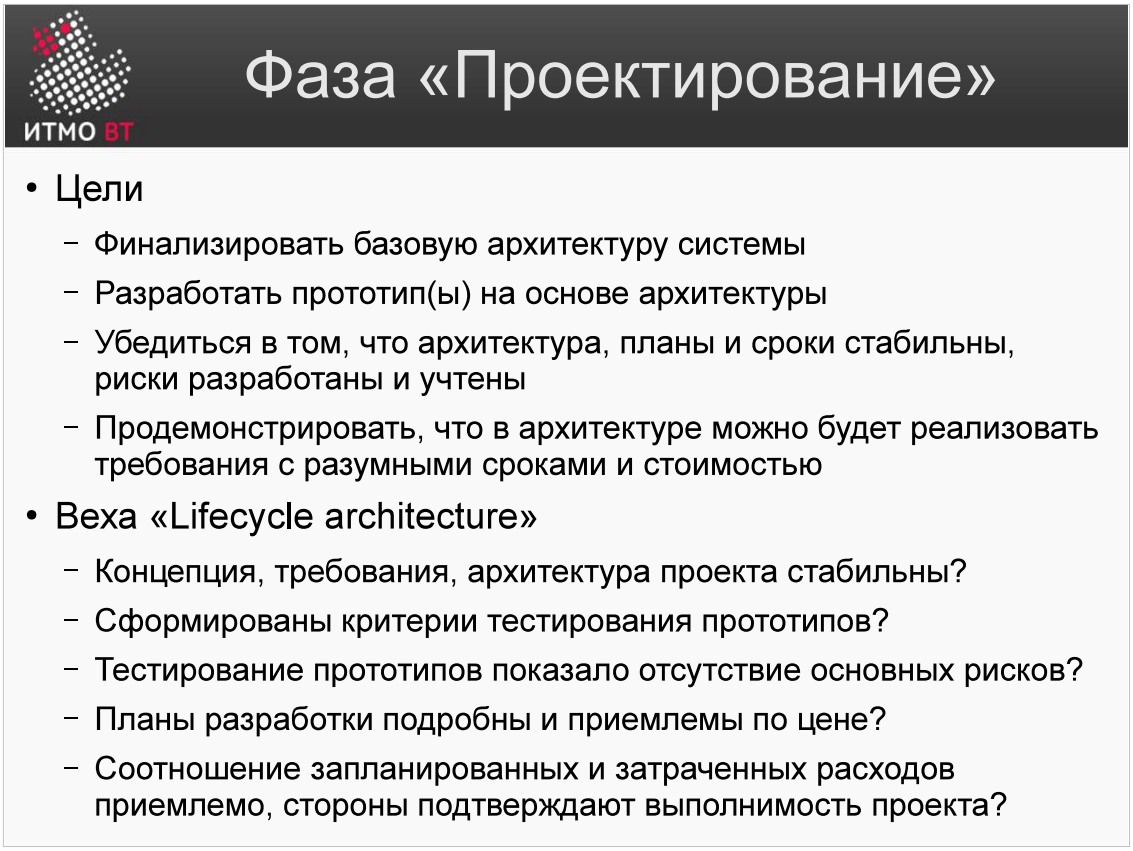
Основное направление работ в фазе Начало — оценить проект, понять, сколько на него нужно потратить ресурсов и времени, какие проблемы пользователей и при помощи какой функциональности проект он сможет решить. Формируется артефакт "Концепция" (Vision).

Цели в фазе Начало:

- Определение границ проекта, решаемых и не решаемых им задач, то есть, ограничение области применения разрабатываемого ПО.
- Разработка и описание основных сценариев использования системы. В рамках этой цели необходимо выяснить, как будут использовать систему пользователи, и определить для каждого сценария последовательность действий. Например, какова последовательность действий в сценарии отправки сообщения в социальной сети?
- Предложение возможных технических решений — на основе каких технологий, API, сторонних решений будет разработано ПО.
- Подсчёт стоимости и разработка графика работ.
- Оценка рисков и подготовка окружения, с помощью которого будет проводиться разработка.

Любая фаза в RUP заканчивается вехами. Веха — это момент времени для принятия решения о дальнейших действиях: переход на следующую фазу, либо проведение каких-либо дополнительных работ на данной фазе. Решение принимается заинтересованными сторонами (stakeholders). Стейкхолдеры или заинтересованные стороны — лица, материально заинтересованные в создании ПО, такие, например, как инвесторы, руководство высшего звена заказчика и исполнителя работ. Они хотят разработать программный продукт и получить с этого прибыль в той или иной форме. Обычные пользователи не являются заинтересованными лицами, так как для них ПО — инструмент работы, а не источник дохода.

На вехе "Lifecycle Objects" заинтересованные лица пришли к согласию в оценке сроков, первоначальной стоимости, требованиях, приоритетах и технологиях. После этого производится переход на следующую фазу разработки.



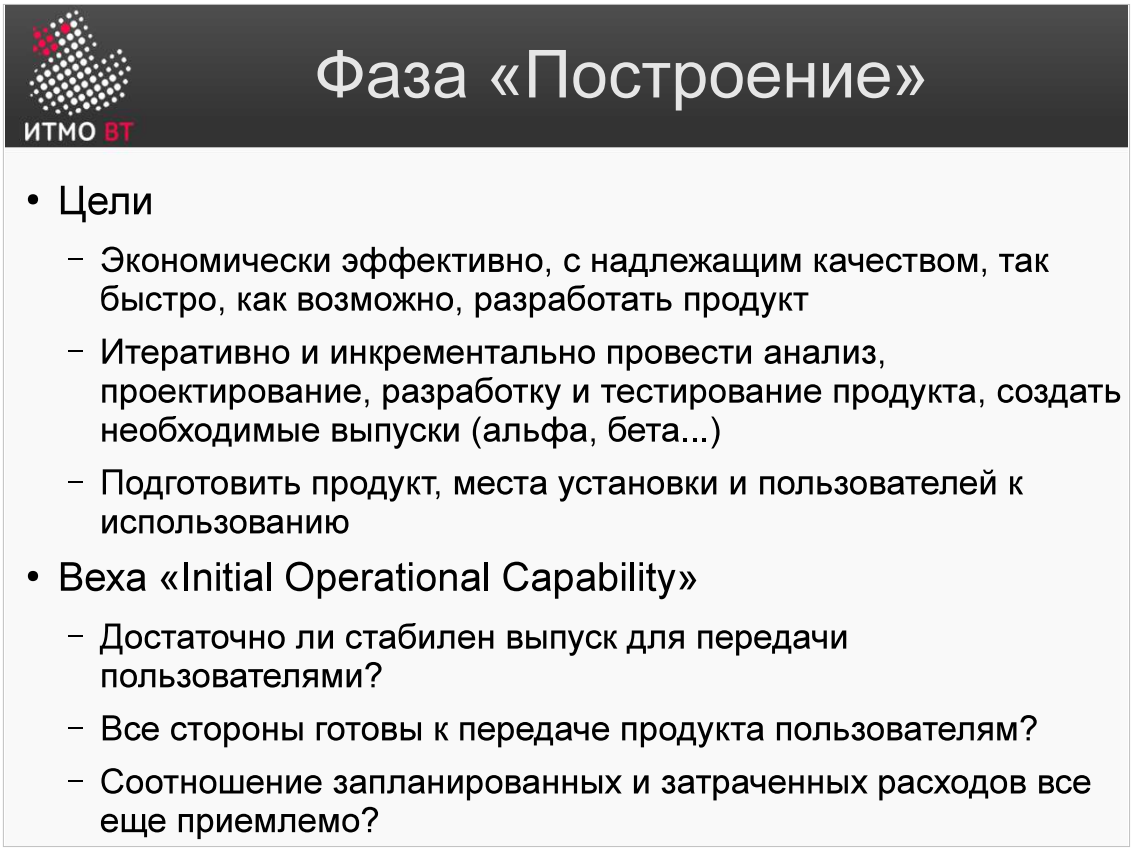
На фазе Проектирование основная задача — разработка и тестирование стабильной и неизменной архитектуры системы, создание одного или нескольких прототипов системы, которые определяют исполняемую архитектуру. Исполняемая архитектура — это полностью законченные на базе выбранных технологий несколько характерных функций разрабатываемой системы. Объём ее рекомендуемой реализации определяется на основании внесения новых архитектурных элементов каждым анализируемым требованием, и, как только реализация новых требований перестает создавать дополнительные элементы архитектуры, можно считать, что один из возможных способов построения готов.

Тестирование на этом этапе предназначено для проверки основных нефункциональных требований к системе, например, пропускной способности или времени отклика.

На основании архитектуры, прототипов и результатов тестирования уточняются планы разработки и производится переоценка и контроль рисков, уточняются сроки и стоимость системы.

Веха "Lifecycle Architecture" — веха стабильности архитектуры. Контрольные точки на данной вехе исходят из того, что стоимость и сроки разработки согласно требованиям, определенным ранее, соблюдены или могут быть приемлемы с точки зрения заказчика. Контролю подлежат потраченные ресурсы и средства, что важно с точки зрения планирования следующей фазы.

Заказчик может принять и аргументированный перерасход средств, который может возникнуть после разработки архитектуры и проверки её характеристик. Сложность такой аргументации заключается в том, что стейкхолдеры со стороны заказчика — люди с бизнес-образованием, далёкие от глубокого понимания деталей современных технологий.



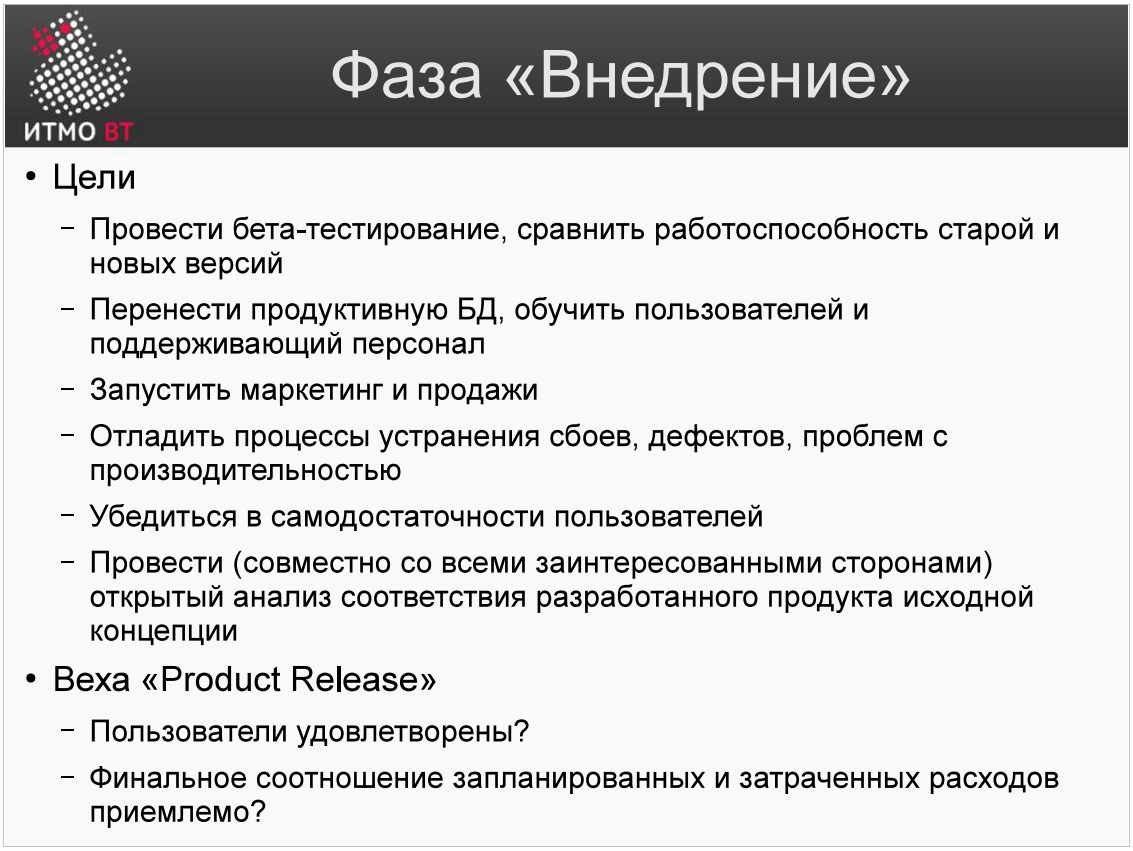
Как было отмечено ранее, до фазы Построения должна быть разработана архитектура приложения; внесение в неё кардинальных изменений к этому моменту должно быть исключено. Основная цель на данной фазе — экономически эффективно, с надлежащим качеством, так быстро, как возможно, разработать программный продукт.

Экономическая эффективность означает в том числе и то, что ресурсы команды разработки не должны тратиться на неосновные работы и переделки. Изменений и улучшений в архитектуру вноситься уже не должно, или они должны быть минимальными. Кроме того, должна проводиться постоянная работа по снижению издержек команды. Необходимо распределять работы с учетом личных особенностей и возможностей всех участников.

Во время фазы происходит создание необходимых выпусков продукта, предусмотренных планом проекта. Проводятся плановые демонстрации версий заказчику.

В конце фазы производится подготовка продукта к передаче заказчику, обследование места установки, разработка учебных материалов и обучение пользователей. Особое внимание уделяется организации и проведению тестирования согласно ранее разработанному плану тестирования, в котором описаны не только проводимые тесты на модульном, интеграционном и системном уровне, но и метрики качества ПО (например, количество вновь выявленных дефектов за неделю) и способы их получения из внутренних систем разработчиков. С точки зрения трудозатрат, на этой фазе непосредственно создание кода составляет лишь часть от полных затрат проектной команды (от 20% до 40%).

На вехе "Initial Operational Capability" принимается решение о возможности внедрения продукта на стороне заказчика. Необходимо учитывать стабильность выпуска (на основании метрик) и готовность пользователей, причём как техническую, так и с точки зрения обучения. Как и в предыдущих фазах ещё раз проверяется приемлемость соотношения реальных и запланированных затрат.



Фаза Внедрение предназначена для запуска продукта в продуктивное использование и финального подтверждения пользователями пригодности продукта к для их практических нужд.

Если система является "заказной", то есть разработанной для конкретного заказчика, то специальная группа разработчиков занимается установкой и настройкой оборудования и ПО системы. При этом может потребоваться конвертация и перенос данных из старой системы в новую, что может быть достаточно трудоёмко.

Если продукт предназначен для конечного пользователя, который сам будет устанавливать ПО у себя на компьютере, на данной фазе необходимо предусмотреть всё, что связано с будущими продажами продукта. Формирование каналов сбыта, подготовка специалистов по продажам, маркетинг и реклама, упаковка коробочных версий продукта — вот неполный список задач, с которым придётся столкнуться разработчикам, а если это стартап, то данная фаза критически важна для начала использования продукта потребителем и, соответственно, получения прибыли.

Основной вопрос на вехе "Product Release" — удовлетворены ли пользователи? Существует поверье, что если пользователи удовлетворены, то с окончательной оплатой продукта, и, соответственно, с получением премий проблем бывает существенно меньше.

Кроме того, важно провести работу над ошибками, и оценить, что в процессе разработки было хорошо, а что требует исправлений. Особому контролю подлежат затраты на разработку — необходимо оценить их отношение к спланированным в фазе Начало для формирования корректировок в будущих проектах.

Организация любой разработки требует не только соответствия каждой роли формальному рабочему процессу, но и наличия особого отношения к исполнению своих обязанностей, которое получило название "Дух RUP". Дух RUP формирует общие командные цели и основные точки фокусировки внимания каждого члена команды разработки.

Например, ранняя атака рисков означает, что чем раньше будет обнаружен и идентифицирован каждый риск разработки, тем выше будет вероятность его избежать, и тем больше времени будет на реакцию на него. Особое внимание должно быть уделено реализации требований заказчиков (а не, к примеру, удобству разработчиков), разработке кода (а не планов и отчетов), и готовности к тому, что изменения могут происходить в любой момент цикла разработки, начиная с меняющихся требований заказчиков, и заканчивая изменениями в подходах к решению задач.

Кроме того, отмечается важность компонентного подхода в разработке системы, ранней разработке архитектуры системы и обеспечение ее стабильности и неизменности на протяжении всего проекта, психологической совместимости и тесного сотрудничества каждого разработчика для достижения основных целей разработки, в том числе и качества разработанного программного обеспечения или аппаратных средств.

Дальнейшее развитие концепций, известных как Дух RUP, явилось одной из важных предпосылок к появлению Agile-манифеста.

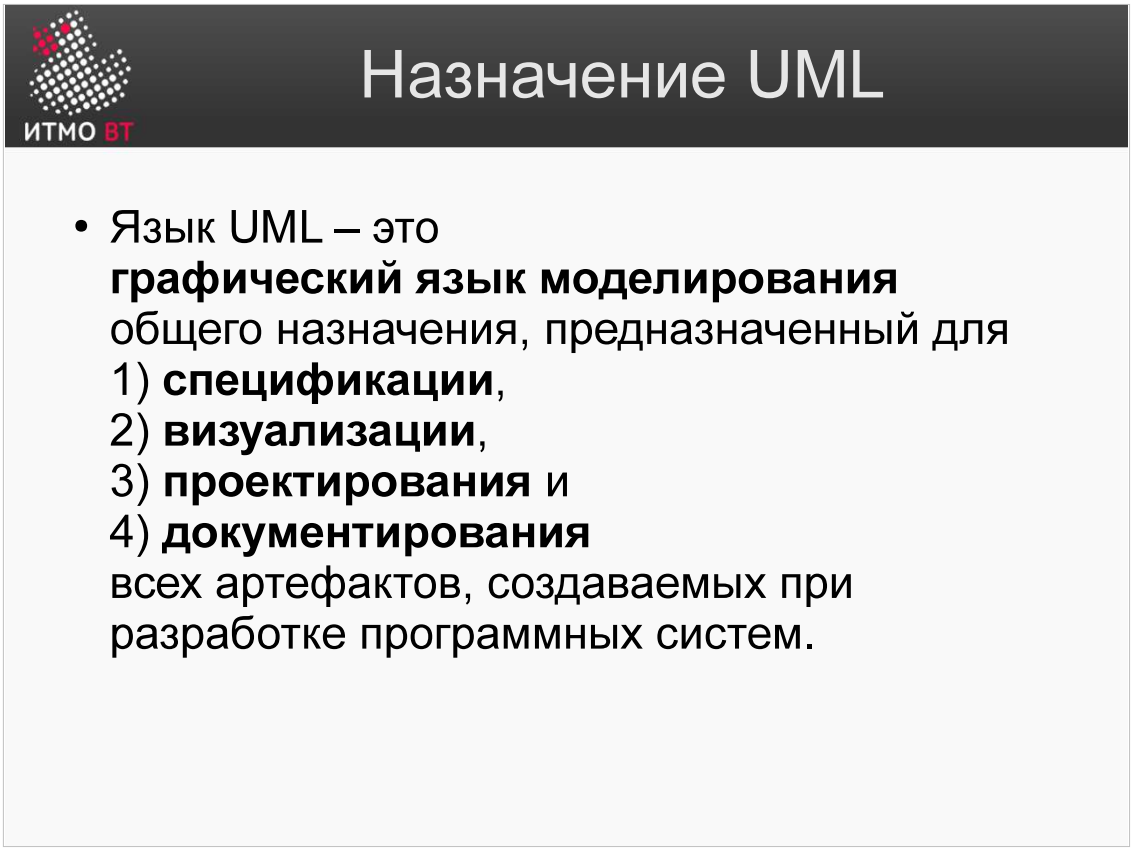
Диаграммы, как средство визуального отображения программного кода и структур данных, существовали с того момента, как люди столкнулись со сложностью передачи и восприятия информации о программных проектах. Словесное описание человеком воспринимается достаточно тяжело, поэтому подключение визуального канала восприятия информации помогало разобраться в сложных проектах. Одновременно с развитием методов разработки, совершенствовались и диаграммы.

В объектно-ориентированном подходе к разработке различные нотации диаграмм были широко известны, но использовались достаточно узко, в методах их создателей. Например, Буч рисовал изображения класса в виде характерного изображения сущности, похожего на амёбу[1], а Рамбо использовал привычные нам сейчас прямоугольники. Практически одновременно с появлением в 1998 году Rational Unified Process, язык UML (Unified Modeling Language — Унифицированный Язык Моделирования), созданный на основе диаграмм Буча, Рамбо, Якобсона, Шлаера и Меллора, Коуда и Йордона в 1997 году стал промышленным стандартом.

Основное назначение UML — графическое представление различных аспектов разработки программного обеспечения. В UML существуют структурные и поведенческие диаграммы. К структурным диаграммам относятся диаграмма классов/объектов, диаграмма компонентов, диаграмма развертывания, диаграмма пакетов и др. К поведенческим диаграммам можно отнести диаграмму деятельности, диаграмму состояний, диаграмму вариантов использования (Use-Case), диаграмму последовательности и др. Наиболее часто из UML используются:

- Диаграмма вариантов использования, при помощи которой описываются высокоуровневые требования к системе. Более подробно мы ее рассмотрим в разделе управления требованиями.
- Диаграмма классов — на уровне анализа в виде доменной модели для описания предметной области, без деталей реализации. На уровне проектирования — для иллюстрации архитектурных механизмов с деталями реализации.
- Диаграммы деятельности, последовательности и состояний — определяют логику и последовательность алгоритма, взаимодействия или внутренних состояний.
- Диаграмма размещения — описывает архитектуру системы с деталями установки компонентов, их точных версий и структуры взаимной вложенности.

1 - см. Гради Буч. Объектно-ориентированный анализ и проектирование с примерами приложений на C++. Первое издание.



Рассмотрим, что представляет собой UML более формально. Согласно представленному на слайде определению, которые дали UML его разработчики, UML - это графический язык моделирования общего назначения.

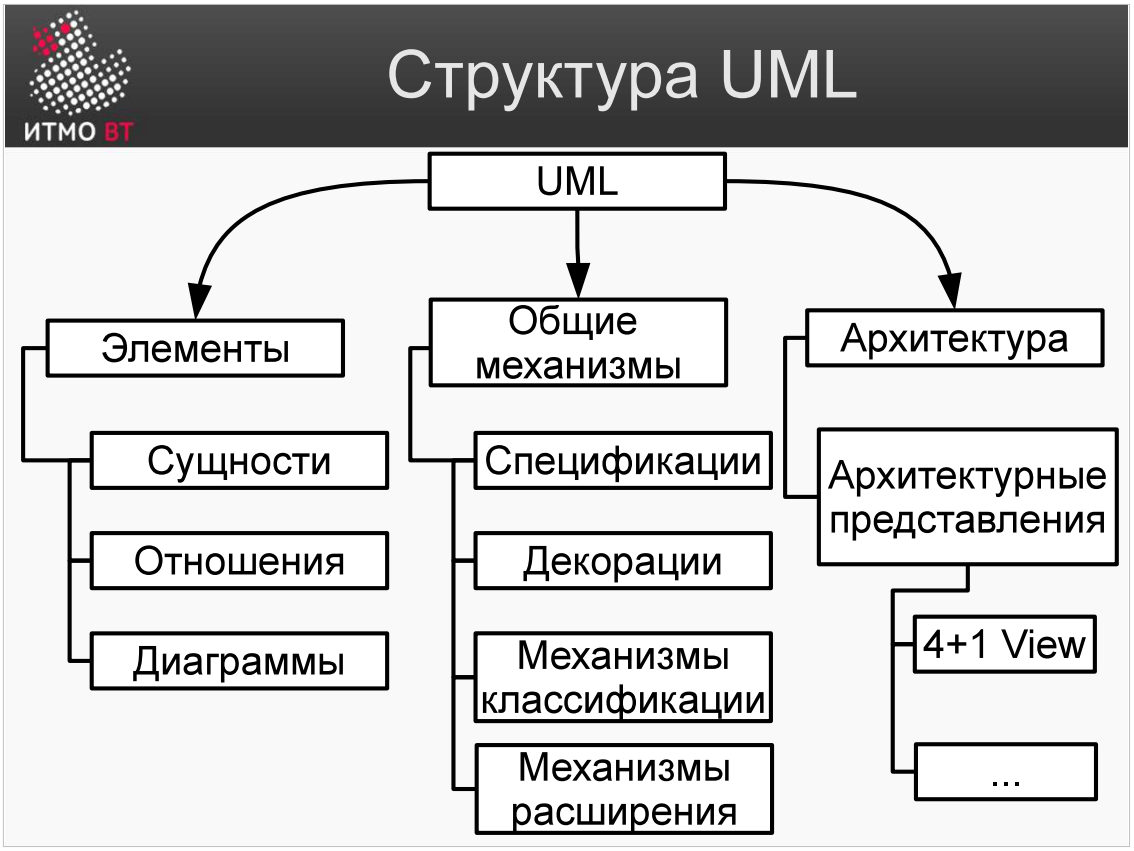
Под языком в данном случае понимают знаковую систему, которая используется для целей обмена информацией и познания. Вы конечно же знаете, чем немецкий язык отличаются от, например, китайского? Любой европейский язык всегда можно прочитать — т. е. превратить буквы в звуки и слова, если вы знаете правила. В китайском языке, если вы даже не знаете точного смысла всех иероглифов, вы можете *догадаться* о чем идет речь. Язык UML можно условно отнести к именно такой группе языков, где синтаксис и семантика выражены при помощи специальных графических конструкций, облегчающим обмен информацией о разрабатываемой программной системе между разными типами стейкхолдеров — и заказчиков и разработчиков.

В UML основным способом передачи информации являются модели. Модель какого либо объекта или системы — это упрощённый объект, который сохраняет одну или несколько функций моделируемого объекта, интересующих разработчиков. Для уменьшения сложности программных систем оказывается выгодно декомпозировать все функции системы на модели и изучить или описать их отдельно. Диаграмма в UML является *представлением* модели.

Спецификация позволяет зафиксировать между стейкхолдерами, что модель точно, полностью и в поддающейся проверке форме определяет требования, устройство, поведение или другие особенности системы, компонента или продукта, а визуализация позволяет всем заинтересованным лицам, при понимании основ языка, просто и точно определить необходимые элементы моделей.

Проектирование в UML позволяет разрабатывать детальные архитектурные шаблоны и решения, которые затем могут быть переведены в программный код разработчиком или же автоматически. Современные средства моделирования при указании целевого языка программирования могут осуществить автоматический экспорт модели в набор конструкций на языке программирования. Кроме того, модели и их части могут быть автоматически построены на основании кода приложений.

Документирование — это важная часть процесса разработки, важность которой была подмечена еще Ройсом. UML представляет для этого широкий набор возможностей.



Согласно Гради Бучу[1], UML состоит всего из трех основных элементов:

- сущности – это так называемые существительные UML модели;
- отношения связывают сущности. Отношения определяют, как семантически связаны две или более сущностей;
- диаграммы – это *представления* моделей UML. Они показывают наборы сущностей, которые раскрывают тот или иной аспект программной системы и являются способом визуализации того, *что* или *как* будет делать моделируемамя система.

В UML существует четыре общих механизма, последовательно применяемых ко всему языку моделирования. К ним относятся:

- спецификации — это детальное описание моделируемого объекта, которая существует как бы на заднем плане диаграммы, и обычно поддерживается автоматизированными средствами моделирования. Если вы создали класс «стул» то в качестве визуального представления на концептуальной диаграмме классов у вас будет прямоугольник с соответствующим заголовком, но в средстве моделирования вы можете указать, что у него есть массив из элементов типа «ножка» и атрибут цвет.
- Декорации - (в оригинале используется англ. Adornments) — набор визуальных представлений элементов модели, которые вы можете выбрать для улучшения ее читаемости.
- Механизмы классификации представлены двумя основными механизмами классификатор/экземпляр и интерфейс/реализация. В UML существует подробная таксономия стандартных классификаторов, которых более 30-ти. Для более подробного описания рекомендуется самостоятельно изучить соответствующую часть спецификации UML.
- Механизмы расширения представлены {ограничениями} — логическими условиями в фигурных скобках, которые моделируемый элемент должен поддерживать в состоянии истины, <<стереотипами>> — определяемыми пользователям новыми элементами модели на основании существующих и помеченными значениями — позволяют добавлять пользовательские свойства вида {имя=значение} к стандартным свойствам элементов модели как переднего так и заднего плана.

1 - см. Booch, G., 2005. The unified modeling language user guide. Pearson Education India.

Объект (object) – сущность, обладающая уникальностью и инкапсулирующая в себе состояние и поведение. Например объект «мой любимый пирожок» может представлять собой восхитительно пахнущий (это является в данном случае состоянием) кулинарный шедевр вашей бабушки, лежащий у вас на тарелке, и способный растаять у вас во рту (вот это можно воспринять как поведение).

Класс (class) – описание множества объектов с общими атрибутами и операциями. Понятие класса совпадает с принятым в объекто-ориентированной парадигме, и синтаксически разнообразно за счет взятых из различных языков программирования элементов их синтаксиса. Например, может существовать класс «Пирожок» и абстрактный клас «Еда».

Интерфейс (interface) – именованное множество операций, определяющее набор услуг, которые могут быть запрошены потребителем и предоставлены поставщиком услуг. В нашем примере, если пирожок остыл, мы можем использовать микроволновую печь, которая реализует интерфейс подогреватель.

Кооперация или взаимодействие (collaboration) – совокупность взаимодействующих для достижения заданной цели или заданных целей объектов. В «Карлсоне, который живет на крыше» Домомучительница готовила плюшки на кухне, а Карлсон их успешно таскал.

Действующее лицо (actor) – внешная, по отношению к системе сущность, являющаяся инициатором взаимодействия с системой. Действующее лицо может быть представлено ролью, другой системой или временем.

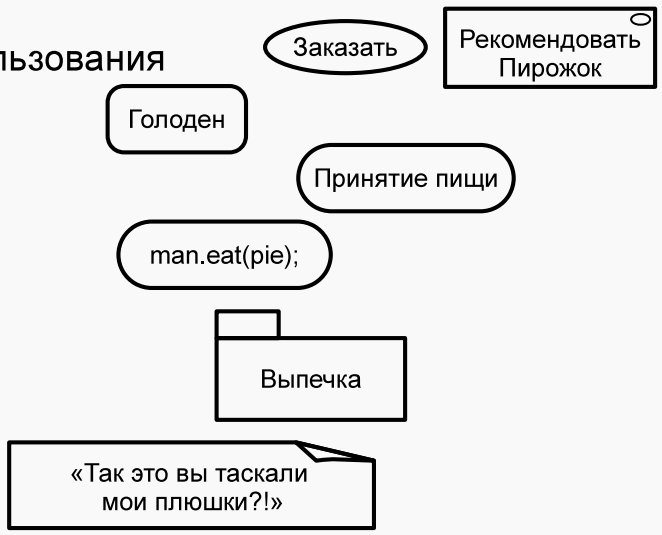
Компонент (component) – модульная часть системы с четко определенным набором требуемых и предоставляемых интерфейсов.

Артефакт (artifact) – результат работы, единица осмысленных действий в результате разработки, мы обсуждали это понятие ранее, когда говорили про RUP.

Узел (node) – вычислительный ресурс, система или комплекс, на котором размещаются и используются компоненты и артефакты.

Сущности UML: поведенческие и дополнительные

- Поведенческие
  - Прецедент использования
  - Состояние
  - Деятельность
  - Действие
- Дополнительные
  - Пакет
  - Комментарий



Поведенческие сущности представлены в UML в 4-х вариантах. Прецедент или вариант использования (use case) — сущность, которая определят акт (действие или набор действий) использования системы внешним действующим лицом, которая производит для него важный результат. На слайде приведены примеры прецедентов использования — заказа пирожков и рекомендации пирожков.

Состояние — период в общем времени жизни объекта, которых характеризуется уникальным и семантически различимым набором значимых внутренних атрибутов объекта. Отсутствие в желудке плюшек и варенья приводит Карлсона в состояние «Голоден». Состояние может быть также выражено продолжительным монотонным действием, например, «Низводить домомучительницу».

Деятельность — объединенное в логически значимое для предметной области множество действий. Действие в этом случае является простейшим вычислением.

Дополнительные сущности представлены пакетом — логической группировкой сущностей. Никакого другого смыслового наполнения кроме группировки пакет не предоставляет. Для облегчения читаемости диаграмм предназначены комментирующие сущности.

Отношение зависимости используется, когда необходимо показать, что исходный элемент зависит от целевого элемента и изменение целевого элемента может повлиять на исходный. Например класс проектной модели диаграммы классов может зависеть от класса в модели предметной области или от действующего лица в прецедентах использования. Если изменится модель действующего лица, например добавится атрибут возраст, то должен изменится и проектный класс, в него также нужно добавить этот атрибут.

Ассоциация показывает, что один элемент связан с другим. Смысловое наполнение связи может быть различным, в том числе и не специфицированным. Существуют отдельные типы ассоциаций для описания дополнительной семантики:

- агрегация, где целевой элемент является частью исходного элемента, и может существовать как отдельный элемент. Так связаны, например, автомобиль и колеса;
- композиция — строгая, форма агрегирования, при которой часть не может существовать без целого, и доступна только через точки взаимодействия, предоставленные целым. В качестве примера можно привести автомобиль и двигатель. Управляя педалью газа у автомобиля, вы на самом деле управляете инжекторами — частью системы впрыска двигателя;
- включение - исходный элемент содержит целевой элемент, и имеет доступ к его пространству имен; Обычно данная ассоциация используется в диаграмме пакетов.

Обобщение - исходный элемент является специализацией более обобщенного целевого элемента и может замещать его. Этим отношением выражается хорошо известное в объектно-ориентированных языках наследование. Стрелка направлена к классу-родителю.

Реализация - исходный элемент гарантированно выполняет контракт, определенный целевым элементом, т. е. реализует его интерфейс.

Специфика всех отношений может уточняться при помощи стереотипов. Различные диаграммы для этого определяют стандартные стереотипы. Например зависимость в диаграмме пакетов может быть уточнена пятью стереотипами: use, import, access, trace, merge.

Пример диаграммы классов уровня описания предметной области, которую еще называют доменной моделью, представлен на слайде. В ней, как можно догадаться из названий классов, речь идет о системе организации увеселительных мероприятий для детей «Малыш и Карлсон». Сразу хотелось бы предупредить — данная модель взята из студенческих работ и в ней есть некорректное использование элементов языка UML, которые вам необходимо будет найти самостоятельно.

Основной сущностью модели является заказ на увеселительные мероприятия, у заказа определены необходимые подтипы заказа, которые связаны отношением обобщения. У каждого ребенка может быть много таких заказов, каждом заказу назначаются исполнители, у которых, соответственно, может быть тоже несколько заказов в работе.

Основным преимуществом данной модели, даже несмотря на ее явные недостатки, является наглядное изображение предметной области. Данная диаграмма по мере дальнейшей работы на стадиях проектирования может быть уточнена и расширена. Например, могут быть добавлены типы данных и область их видимости, кратность и роли ассоциаций и т.д.

Диаграмма состояний, или как она называется в UML2, диаграмма конечных автоматов представлена на слайде. Как можно заметить, она выполнена с детализацией, соответствующей уровню реализации программного кода, и отражает состояние аутентификации в приложении «Малыш и Карлсон» и авторизации зарегистрированного пользователя на просмотр страницы.

Обратите внимание, что несмотря на наличие деталей уровня реализации диаграмма не является перегруженой. Даже неподготовленный представитель заказчика может разобраться в примененных логических решениях к аутентификации и авторизации (хотя, конечно, здесь тоже особая студенческая логика).

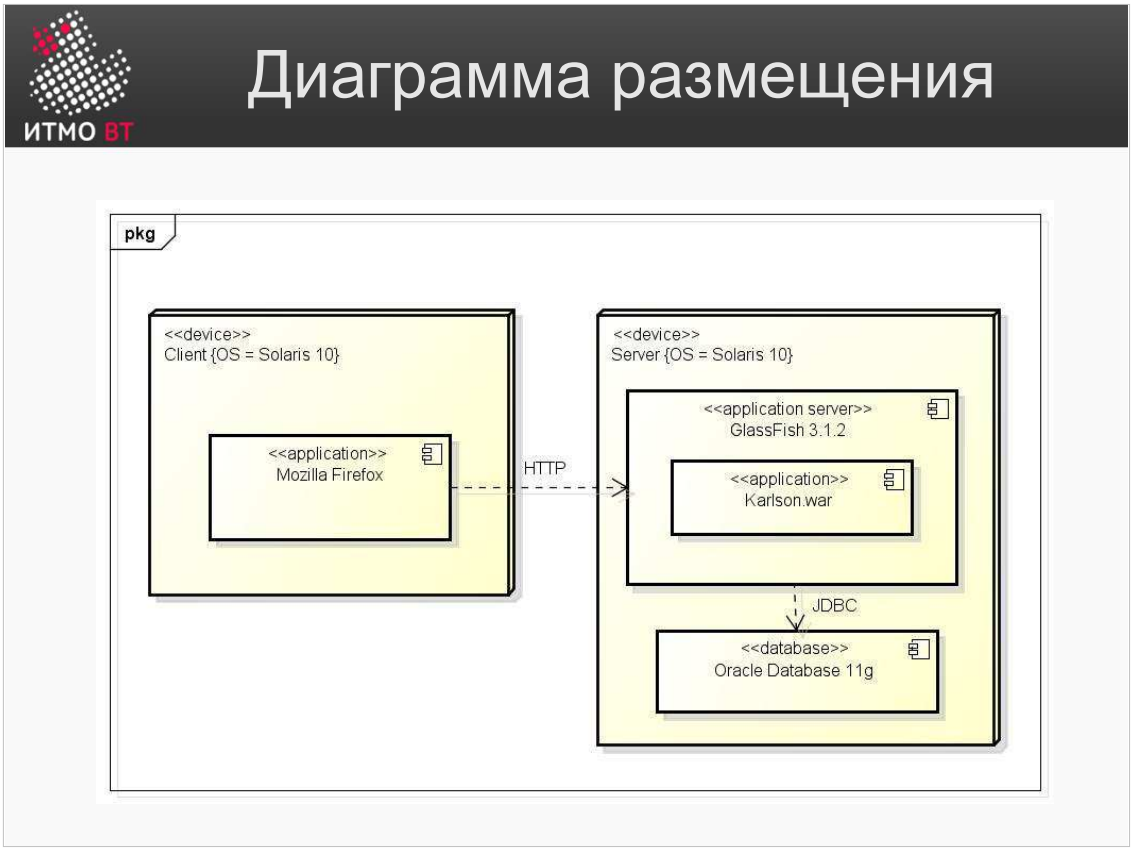


Диаграмма размещения показывает физическую архитектуру размещения частей приложения на серверах. По данной на слайде диаграмме видно что сервер представляет собой программно-аппаратный комплекс под управлением операционной системы Solaris10, на нем размещено два программных компонента — база данных и сервер приложений. Приложение представлено в виде артефакта Karlson.war.

Клиентом разработанной системы служит браузер Файрфокс, который по протоколу HTTP осуществляет запрос страниц на сервере.

Следует отметить, что рамки данного курса не позволяют полностью рассмотреть все диаграммы UML. Рекомендуем продолжить его изучение самостоятельно. Диаграмма прецедентов будет рассмотрена в разделе определения требований.

RUP — детально продуманный процесс, который предназначен для разработки от малых до больших программных продуктов. В нем, помимо собственно разработки, описывается много деятельностей, связанных с организацией процесса разработки. Однако, требования бизнеса, и продукты, пользующиеся спросом, постоянно меняются вместе с технологиями и обществом. Соответственно, требуется подход к разработке, который позволил бы гибко и быстро реагировать на такие изменения. Способность удовлетворять постоянное изменение бизнес-требований явилось важной предпосылкой появления в 2001 году Agile-манифеста (произносится как эджайл-манифест). Декларируется, что "Agile-процессы позволяют использовать изменения для обеспечения заказчику конкурентного преимущества".

Во главу угла в Agile-подходе ставятся требования заказчика, и, следовательно, реакция на них. Данный подход невозможен, или практически невозможен, если на проект выделяется фиксированный бюджет, без возможности его расширения. Действительно, если мы купили велосипед, а потом хотим поменять на нем сидение на более удобное, то обычно у нас не возникает вопросов о дополнительных расходах по его приобретению и установке. Но с точки зрения заказчика программных продуктов, а если точнее, его фиксированного бюджета, изменения в проекте не должны подлежать дополнительной оплате.

Agile-методологии хорошо работают для внутренних проектов разработки в компаниях, которые занимаются различными бизнесом, приносящим доход, или когда заказчик непосредственно покупает время (т. е. рабочие часы) разработчиков. Разработчики, со своей стороны, должны постоянно демонстрировать результат работ, этот результат оценивается, разработчики получают деньги, и немедленно начинают разрабатывать новый функционал. При этом важно открыто и искренне сотрудничать с заказчиком, принимать во внимание его замечания.

Ещё одним важным принципом является максимальное сокращение расходов на труд разработчиков, не относящийся к созданию кода. В небольших проектах с типовой архитектурой это реализуется просто, а в более сложных порождает проблемы, так как без моделей и документирования архитектуры невозможно создать проект, выходящий за рамки типового.

В итоге появился целый класс так называемых гибких методологий, наиболее известная их них называется Scrum. Существуют гибкие модификации последователей RUP — методы AUP, OpenUP.

Мы следуем таким принципам:

Наивысшим приоритетом для нас является удовлетворение потребностей заказчика, благодаря регулярной и ранней поставке ценного программного обеспечения.

Изменение требований приветствуется, даже на поздних стадиях разработки. Agile-процессы позволяют использовать изменения для обеспечения заказчику конкурентного преимущества.

Работающий продукт следует выпускать как можно чаще, с периодичностью от пары недель до пары месяцев.

На протяжении всего проекта разработчики и представители бизнеса должны ежедневно работать вместе.

Над проектом должны работать мотивированные профессионалы. Чтобы работа была сделана, создайте условия, обеспечьте поддержку и полностью доверьтесь им.

Непосредственное общение является наиболее практичным и эффективным способом обмена информацией как с самой командой, так и внутри команды.

Работающий продукт — основной показатель прогресса.

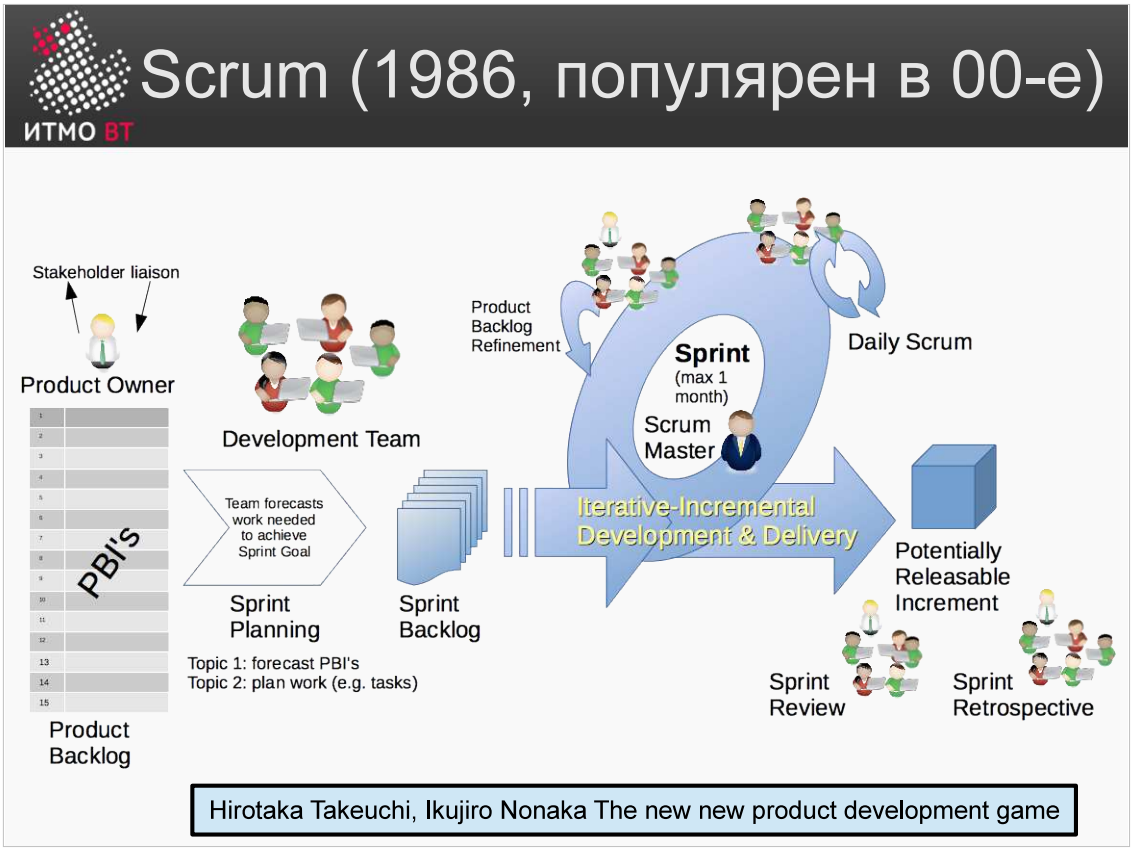
Инвесторы, разработчики и пользователи должны иметь возможность поддерживать постоянный ритм бесконечно. Agile помогает наладить такой устойчивый процесс разработки.

Постоянное внимание к техническому совершенству и качеству проектирования повышает гибкость проекта.

Простота — искусство минимизации лишней работы — крайне необходима.

Самые лучшие требования, архитектурные и технические решения рождаются у самоорганизующихся команд.

Команда должна систематически анализировать возможные способы улучшения эффективности и соответственно корректировать стиль своей работы.



Основы метода Scrum были заложены в 1986-м году в статье Хиротако Такеучи и Икуджиро Нонаки.

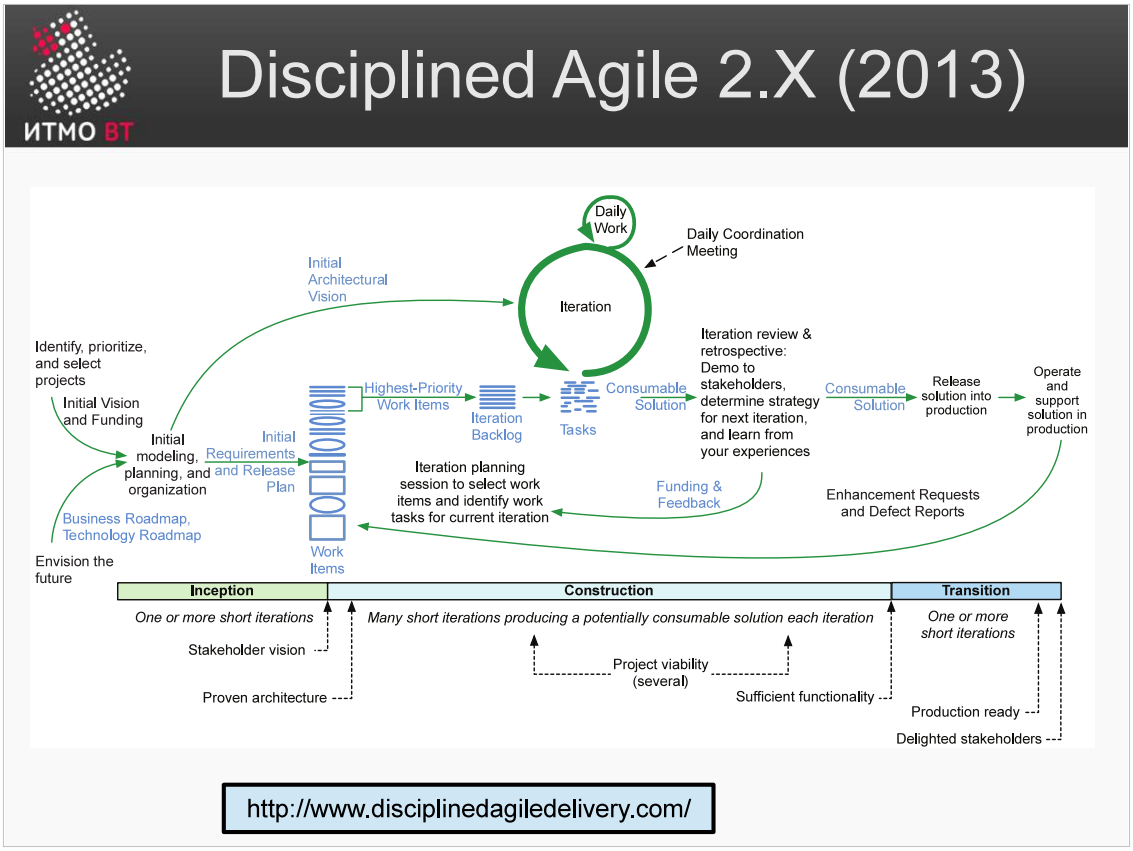
В Scrum собственно процесс разработки сильно упрощён. Основной и единственный служебный артефакт Scrum — бэклог (от англ. Backlog — долг) — упорядоченный по приоритетам список требований с оценкой трудоёмкости разработки. Существуют бэклог продукта, обычно более общий и состоящий из бизнес-требований, и бэклог спринта — более детальный, с учетом технических особенностей реализации. Другим артефактом является инкремент продукта.

Спринт — время от двух до четырёх недель, за которое разработчики реализуют выбранный набор требований из бэклога спринта. Каждый спринт заканчивается демо-версией, и проделанная работа демонстрируется заказчику. Также через несколько спринтов проводятся ретроспективы, в рамках которых может проводиться работа над ошибками и перераспределение обязанностей для более эффективного использования каждого участника команды.

Команда в Scrum небольшая, от 3 до 10 человек. Также выделяется особая роль — Владелец Продукта (Product Owner) — это человек, определяющий порядок разработки требований из бэклога (и, в случае, если он является бизнес-заказчиком, зачастую сам их формулирующий). Для каждого спринта задачи конкретизируются и передаются на разработку команде. Кроме этого, отдельно выделяется Скрам-мастер, ответственный за проведение скрам-митинга, который помогает команде планировать спринт и следит за внутренними отношениями в команде.

Ежедневно утром производится скрам-митинг, где каждый отчитывается о проделанной работе, возникших проблемах, и о том, что он собирается сделать к следующей встрече.

Достоинство Scrum — простота, минимум административной работы и документов, и максимальная концентрация на работоспособном коде. Scrum лучше всего подходит для проектов с небольшой командой разработки. Предпринимаются попытки масштабировать Scrum на большие команды, Scrum-of-Scrum, которые пока не показали существенных, зафиксированных отраслью, успехов.



Очевидная слабая способность гибких методологий масштабироваться на команды с бо́льшим числом разработчиков была замечена многими в отрасли разработки программного обеспечения. То, что хорошо работает для стартапа с высокой мотивацией всех разработчиков-стейкхолдеров, плохо подходит для ежедневно встающих на работу в 7:00 тысяч (включая вспомогательный персонал) строителей марсохода. Поэтому попытки создать процесс разработки, который несёт небольшую управленческую нагрузку, имеет низкие издержки и одновременно может включать сотни и тысячи программистов — супер-актуальная цель для рынка разработки ПО.

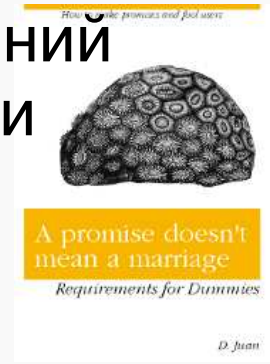
Создатель методов AUP (Agile Unified Process) и EUP (Enterprise Unified Process — метод разработки с учетом полного жизненного цикла ПО, включая фазы поддержки и вывода из эксплуатации) Скотт Амблер в 2013 году предложил новый метод — Disciplined Agile 2.X (DAD). Подход к делению на фазы и дисциплины в DAD очень похож на RUP, но основной цикл разработки построен на базе гибких методов, в том числе Scrum. Пока данная методология является относительно новой для рынка и находится в стадии развития.

Помимо деления процесса на фазы, которых предлагается три (Начало, Построение и Внедрение), и описания каждой роли в разработке, DAD рассматривает процессы, выходящие за рамки собственно процесса разработки. К ним относятся: управление архитектурой и повторным использованием кода, управление персоналом, служба поддержки и текущих операций компании-разработчика, управление портфолио компетенций, непрерывное улучшение процессов разработки и вспомогательных процессов, и многое другое. В последнее время отдельное внимание уделяется сбору и анализу большого объёма данных, накопленных в рамках разработки большого количества разнородных проектов по технологиям BigData.

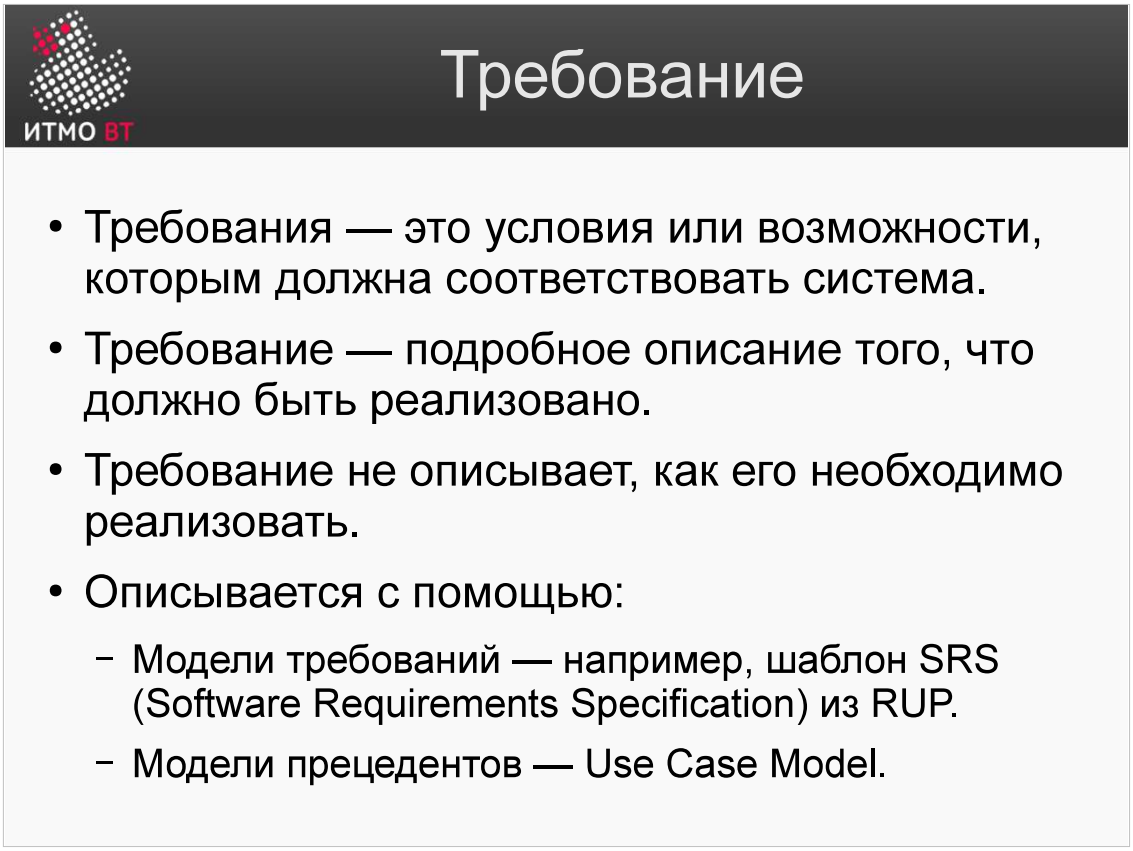
ИТМО ВТ

2

Определение требований
Управление рисками



Леффингуэлл, Дин, Уидриг, Дон. Принципы работы с требованиями к программному обеспечению. Унифицированный подход 2002, М., Вильямс ISBN: 5-845-0275-4



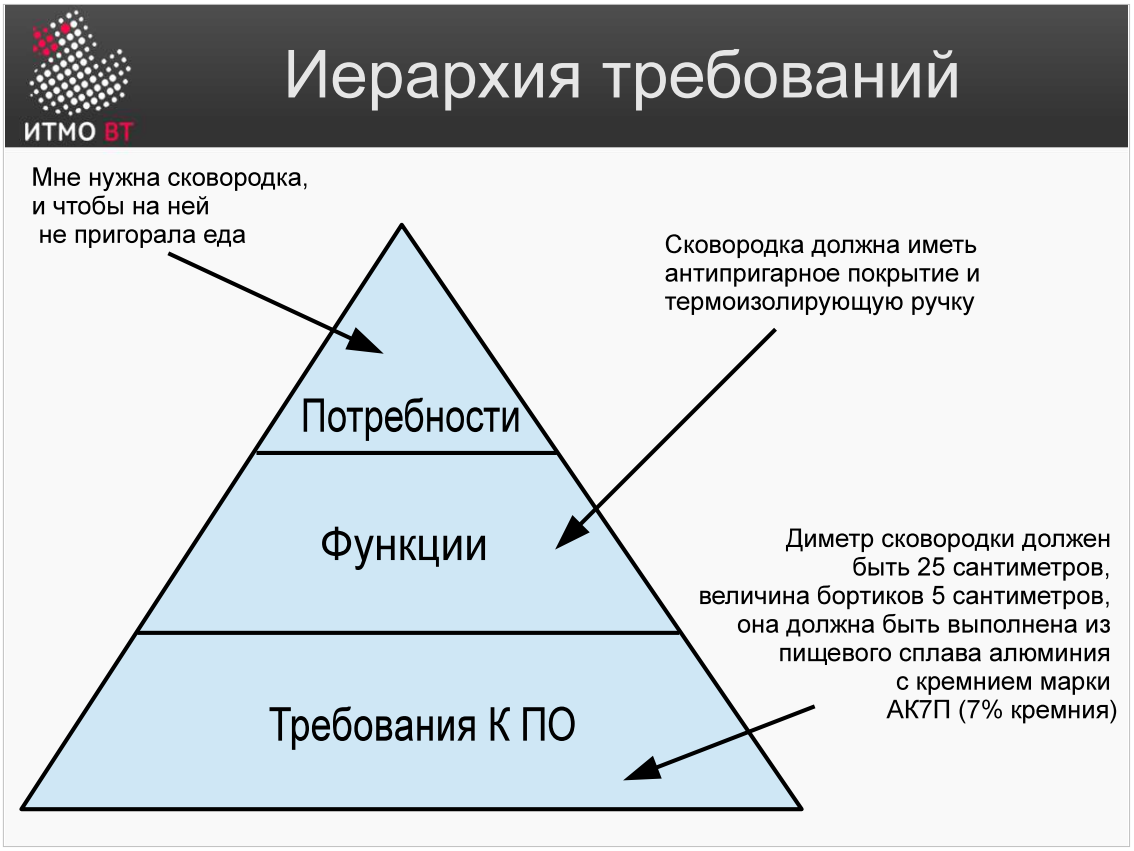
Требования — это условия или возможности, которым должна соответствовать система. Требование должно быть однозначно сформулировано, и грамотно и четко описывать то, что системе необходимо выполнить.

Требования могут общими, или подробно описывать то, что должно быть реализовано. В отличие от, например, документов об архитектуре системы, требование не описывает, **как** необходимо реализовать ту или иную часть системы. Оно лишь указывает на то, **что** нужно реализовать.

Существует много способов описания требований. В данном курсе рассматривается подход, сформулированный в Rational Unified Process.

Первый из предусмотренных способов описания требований — это модели требований. Для их описания существует шаблон — Software Requirement Specification, который обычно и является формальным техническим заданием на разработку. В дополнение к нему существует отдельный набор детализированных описаний прецедентов, который обычно включает последовательность шагов по выполнению в системе определённых действий пользователя со всеми возможными ветвлениями в сценарии работы, включая ошибочные. Также в них предусмотрены прототипы графических (или иных) интерфейсов взаимодействия с системой.

Более нагляден и лучше походит для заказчиков, не обладающих навыками разработки, второй способ — модели прецедентов (Use Case Model), описывающие требования в виде UML-диаграммы. Следует отметить, что почти все руководители бизнеса, чьи основные пожелания и реализует система, хотя и знакомы с основами вычислительной техники, не могут (да и не должны!) понимать все технические аспекты реализации вычислительных систем.

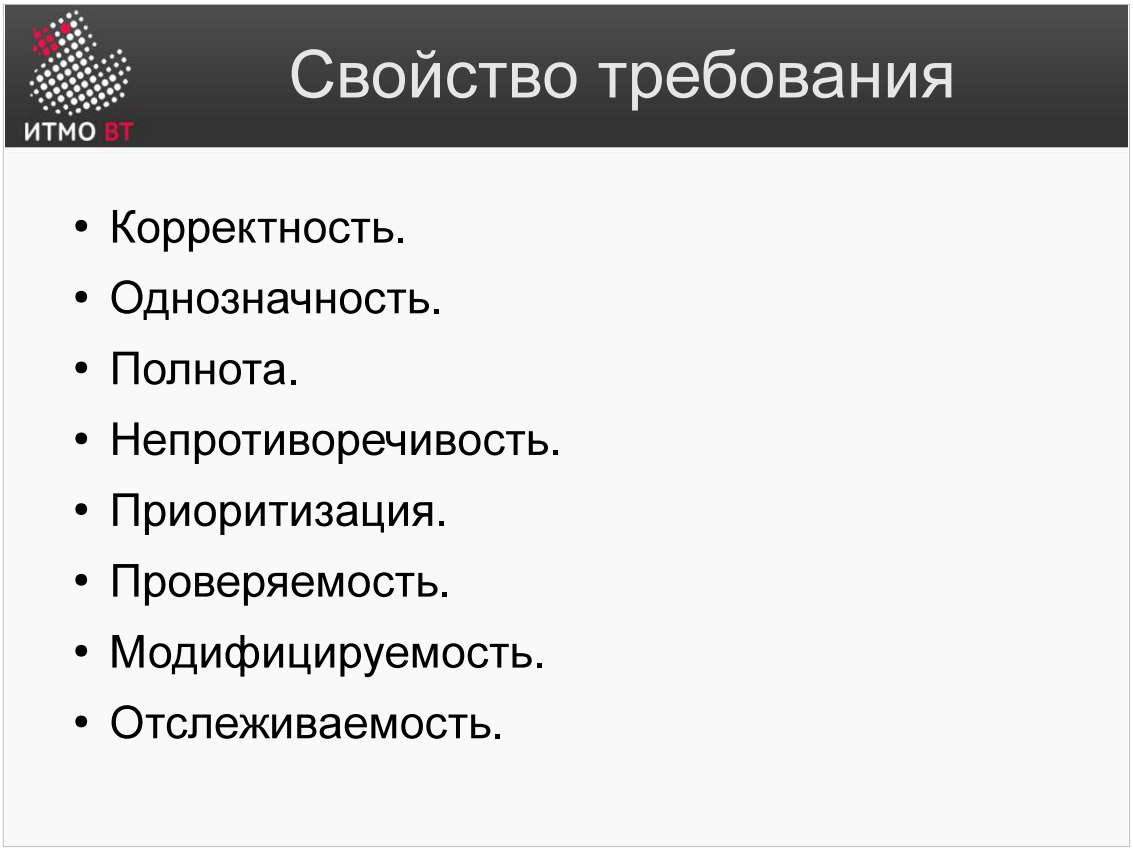


Требования по степени своей детализации можно разделить на несколько типов. Первый — потребности заинтересованных лиц. В большинстве случаев они представлены в виде пожеланий для решения конкретной задачи — "Мы хотим, чтобы нашему бизнесу было хорошо!". К сожалению, множество заказчиков представляют ПО в виде набора волшебных кнопок, которые магически выполняют поставленное желание. Например, обычно заказчик может сформулировать свои требования так: "Мне нужна сковородка, и чтобы на ней не пригорала еда".

Вместе с пожеланиями заинтересованные лица предоставляют набор артефактов, которые можно использовать для иллюстрации информации, которую будет содержать система. Этот набор обычно не очень большой, но даёт начальное представление о системе.

Задача аналитика — перейти к области решения задачи и преобразовать потребности в набор функций, которые должна реализовывать система. Здесь уже необходим больший формализм и чёткость в формулировках, однако эти требования всё ещё должны быть сформулированы языком, понятным для заказчика. Например: "Сковородка должна иметь антипригарное покрытие и термоизолирующую ручку".

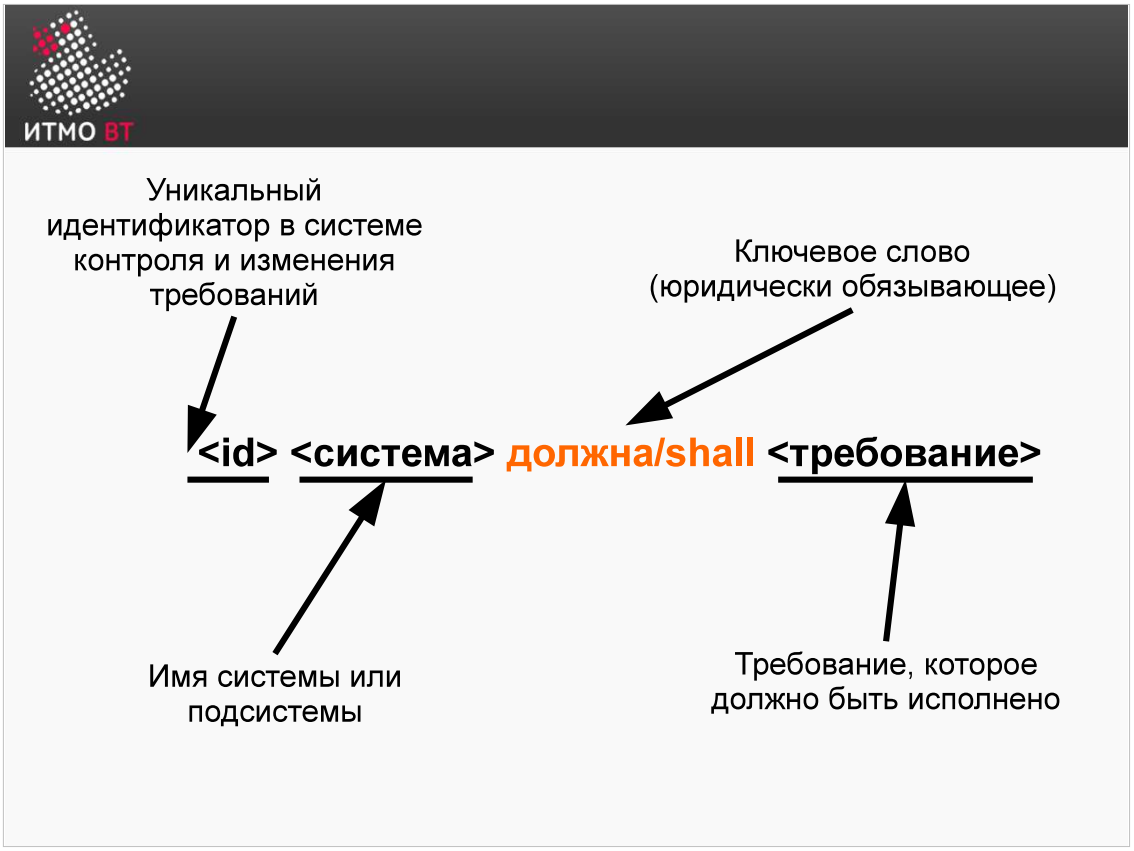
После утверждения заказчиком функций системы можно переходить к планированию реализации и формулированию конкретных, утверждающих детали реализации, требований к программному обеспечению. Например: "Диметр сковородки должен быть 25 сантиметров, величина бортиков 5 сантиметров, она должна быть выполнена из пищевого сплава алюминия с кремнием марки АК7П (7% кремния) и строго соответствовать требованиям ГОСТа 1583-93и. Для антипригарного покрытия необходимо использовать покрытие фирмы Scandia толщиной 18-25 микрон."



Свойства требований представлены на слайде. В первую очередь, требование должно быть однозначным и полным. Требования не должны противоречить друг другу. Некорректно сформулированные требования могут привести к путанице из-за того, что одним и тем же словам разные люди придают разный смысл.

Для определения последовательности реализации требования должны иметь приоритет. В современной итеративной разработке бизнес-систем приоритет отражает последовательность бизнес-функций и зависит от степени критичности для бизнеса этих функций. Заказчик обычно определяет его самостоятельно. Иногда приоритет может зависеть от технической последовательности реализации — например, сковородку надо сначала отлить, а потом уже покрыть антипригарным покрытием.

При разработке всегда необходимо понимать, на основании каких потребностей или технических особенностей требование сформулировано, т.е. требование должно ссылаться на свой источник. Модификация требований также должна быть контролируемой, в любой момент необходимо знать историю их изменений. Отслеживание требований и их модификаций производится при помощи систем контроля над изменениями, которые в современной программной индустрии достаточно развиты.



На слайде приведено используемое в курсе описание требования. Ключевым словом в описании является юридически обязывающий модальный глагол "должен" ("shall"). Обычно это слово описывает безусловную необходимость в текстах договоров на разработку данного требования для программного обеспечения.

Требование всегда нумеруется, возможны разные виды нумерации, обычно в номер стараются включить буквенный идентификатор типа требования, например FR10 — десятое функциональное требование.

Также требование должно содержать имя системы или подсистемы, к которой оно относится. Если система имеет длинное наименование (на практике встречаются реально очень длинные названия), то имя может сокращаться, записываться аббревиатурой или называться кодовым словом. Кодовые слова обычно существуют внутри компании-разработчика до выпуска продукта на рынок и могут ничего не значить для человека, не участвующего в разработке, например, кодовое имя сервера компании Sun Microsystems, который на рынке продавался как SunFire v440 — "Chalupa" — мексиканское мясное блюдо.

В RUP для описания и разработки требований используется модель FURPS+ (Functional, Usability, Reliability, Performance, Supportability).

Требования делятся на два типа:

- Функциональные, которые определяют, что система должна делать.
- Нефункциональные, описывающие характеристики и ограничения, накладываемые на систему.

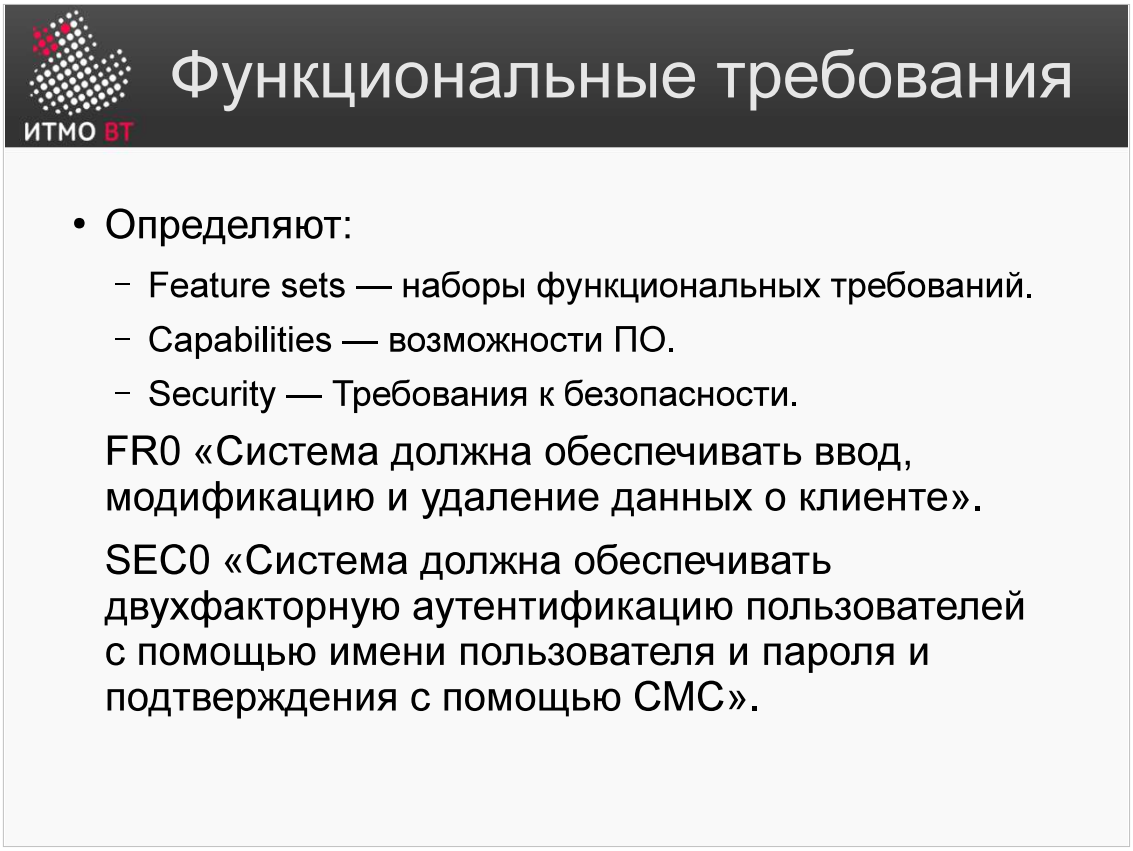
Нефункциональные требования включают в себя:

- Требования к пользовательским характеристикам ПО (на жаргоне программистов их так и называют — требования к юзабилити).
- Требования к надёжности получившийся системы и способности ее обеспечить требуемую готовность к выполнению операций.
- Требования к производительности полученного ПО (например, способность обеспечить заданное время отклика или скорость обработки задач).
- Требования к условиям поддержки (например, способность проводить обновления ПО без остановки функционала, реализуется при помощи кластерных технологий).

Более подробно об этом можно посмотреть по ссылке на слайде.

Кроме того, требования могут включать в себя ограничения на использование тех или иных прикладных программных интерфейсов и продуктов для формирования архитектуры, требования к реализации (например, математический метод решения задачи), а также физические требования (например, параметры импульсов).

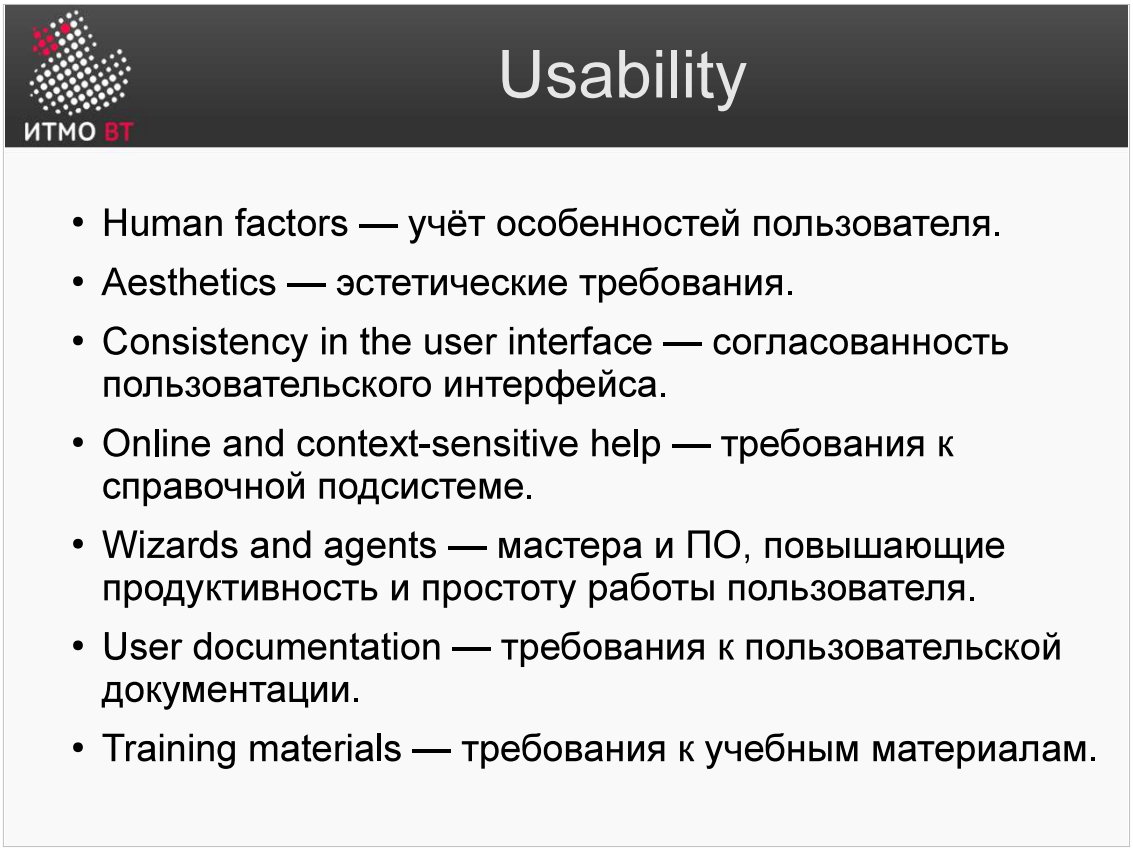
Термины, связанные с определением требований, определены в стандарте ISO/IEC/IEEE 24765:2010(E).



Рассмотрим функциональные требования. Feature Set (набор функциональных требований) — это набор свойств продукта, необходимый для выполнения конкретной деятельности. Функционал обычно идет крупными блоками, отсюда и слово "набор", то есть речь идет о наборе действий, связанных между собой по смыслу.

Требования к безопасности обычно описываются отдельно и включают в себя метод аутентификации, список ролей, существующих в системе, шифрование, хранение данных в защищенных источниках. Кроме того, данные требования могут описывать и организационные меры, исключающие доступ злоумышленников к системе.

В качестве примеров на слайде приведены функциональное требование номер FR0 «Система должна обеспечивать ввод, модификацию и удаление данных о клиенте» и требование к безопасности SEC0 «Система должна обеспечивать двухфакторную аутентификацию пользователей с помощью имени пользователя и пароля и подтверждения с помощью СМС». Повторимся, что уникальный идентификатор всегда позволяет отслеживать требование за счет уникального названия.



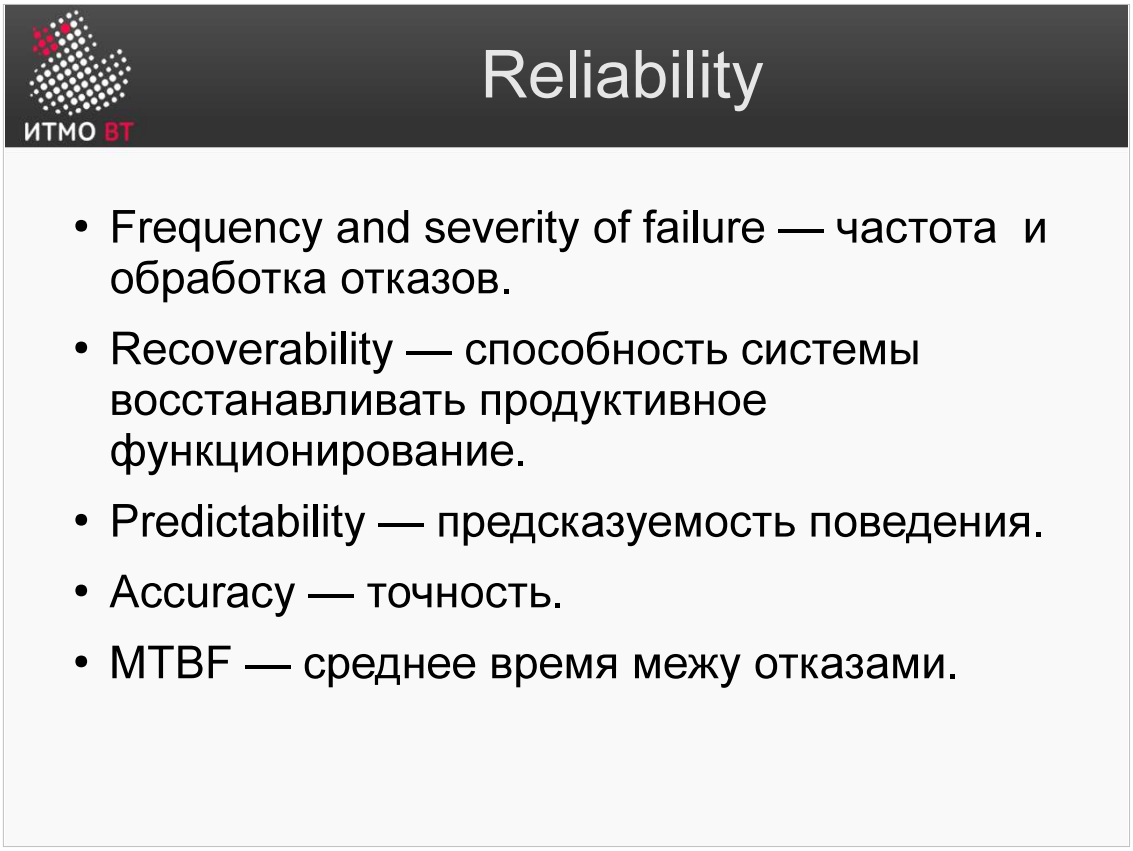
В требования по юзабилити обычно входят те особенности использования, которые надо учесть при разработке ПО.

В первую очередь это человеческий фактор, то есть учёт физических особенностей человека. Например, обычные пользователи (не связанные с компьютерами в обычной жизни) в диалоговых системах ожидают от ПО поведения человека, т.е. интервал ответа системы должен находится в диапазоне 1-5 секунд, и, если ответ приходит быстрее, человек может не понять, что что-то произошло в системе, а если медленнее — то он будет считать, что система "тупит". Кроме того, для пользователей с ограниченными возможностями обычно реализуют специальные элементы интерфейса.

Эстетические требования могут отсутствовать, но, если они определены, они могут быть достаточно сложными и подробными, с указанием цветовой палитры и точного расстояния между компонентами, которые придирчивый заказчик будет проверять с линейкой. В крупных компаниях в обязательном порядке существуют так называемые brandbooks — справочники по текущему корпоративному стилю.

Отдельным пунктом при разработке могут требоваться мастера настройки, которые упрощают последовательность действий пользователя для типовых задач. Обычно все процедуры установки программного обеспечения реализованы подобным образом. В юзабилити-требования включены требования к онлайн-подсказкам, пользовательской документации и учебным материалам, как к средству подготовки пользователей и решения возникших у них проблем.

Для реализации таких требований на рынке существует подход, который получил название User Experience Design или UX, который предназначен для создания удобных пользовательских интерфейсов.



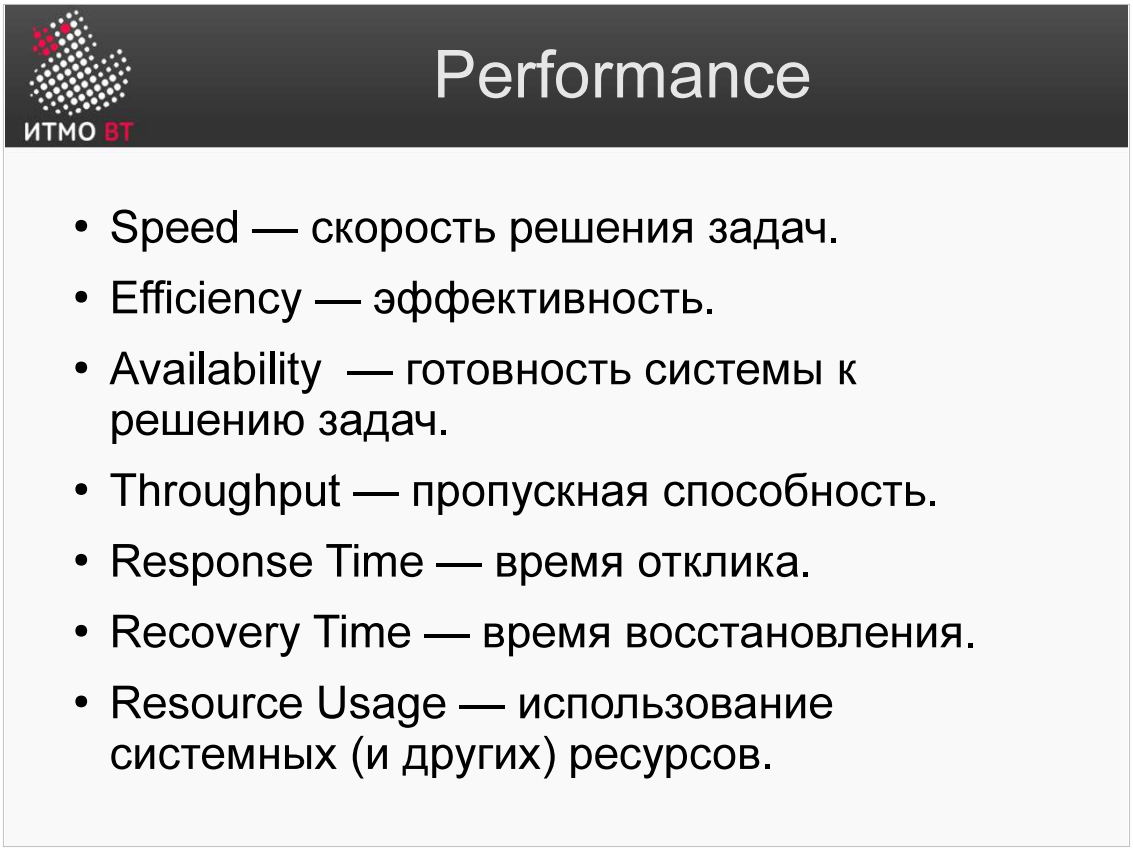
Требования к надёжности (reliability) — предназначены для фиксирования способности ПО безотказно функционировать в течение определённого периода времени. Отказ системы — это неспособность системы выполнять основную функцию. Кроме отказов, существуют еще и сбои — случаи неспособности выполнять отдельный функционал при сохранении общей работоспособности системы. Для корректной разработки требований к надёжности необходимо категоризировать сбои и отказы по уровню их критичности для заказчика, а также определить реакцию системы и обслуживающего персонала на них.

В требованиях обычно указывается допустимое число отказов и сбоев за определённый промежуток времени. При этом в договор обычно вносятся штрафы разработчиков за превышение допустимого числа отказов их продуктом. Также в требованиях указывается recoverability — способность системы восстанавливать продуктивное функционирование в течение заданного времени.

Отдельным важным требованием является accuracy — точность, например, проведения вычислений. Для математических моделей управления физическими системами точность имеет решающее значение.

Часто надёжность системы характеризуются единым интегральным параметром, который называется MTBF (Mean Time Between Failures) — среднее время между отказами. Коэффициент готовности системы — отношение времени исправной работы к сумме времён исправной работы и вынужденных простоев объекта, взятых за один и тот же календарный срок.

На рынке сейчас присутствуют программные кластерные системы, которые обеспечивают готовность прикладного ПО с коэффициентом 99,999 (жаргон — «пять девяток»). В годовом исчислении это 5 минут простоя в год. Более того, некоторые компании заявляют о готовности их систем с коэффициентом 99,99999 («семь девяток»). По нашему мнению, такой коэффициент готовности носит скорее рекламный характер.



Требования к производительности (Performance) включают в себя большой набор различных требований. В первую очередь, таким требованием является скорость решения вычислительных задач. Особое значение скорость принимает в системах реального времени, когда необходимо произвести расчеты в точно определенный интервал времени, иначе может быть нарушено функционирование всего объекта управления. Также скорость важна в длительных инженерных расчетах, когда необходимо выполнить, например, моделирование за разумное для человека время.

Требования к эффективности фиксируют процент времени, которое тратится на выполнение полезных задач, по отношению к времени на выполнение общесистемных.

Важным требованием к производительности является готовность (availability) быстро начать выполнение задачи. Иногда это ещё называют реактивностью системы. Архитектура всех современных операционных систем предполагает наличие задержек, связанные с особенностями диспетчеризации процессов и потоков на ядра процессора, а также разными алгоритмами такой диспетчеризации (пакетная, интерактивная обработка, обработка с разделением времени и пр.), что необходимо учитывать при спецификации требований к реактивности системы. Кроме того, архитектура прикладного ПО должна быть специальным образом организована с целью уменьшения времени начала обслуживания, что также может быть отражено в требованиях.

Другим важной характеристикой является пропускная способность (throughput), которая показывает, какой объём данных или запросов система может обработать за единицу времени. Эта характеристика обычно ограничивается параметрами аппаратуры. Правильно созданная система с точки зрения пропускной способности имеет линейный рост функции времени ответа на запрос от числа запросов до определённого предела, после чего наступает насыщение, и время ответа начинает расти, например, экспоненциально. Различные временные параметры связаны между собой. Например, для большой реактивности придется пожертвовать пропускной способностью. Таким образом, различные требования обычно приходится балансировать.

В сложных корпоративных системах отдельно выделяются требования поддержки (Supportability) программного обеспечения. Данные требования представлены на слайде.

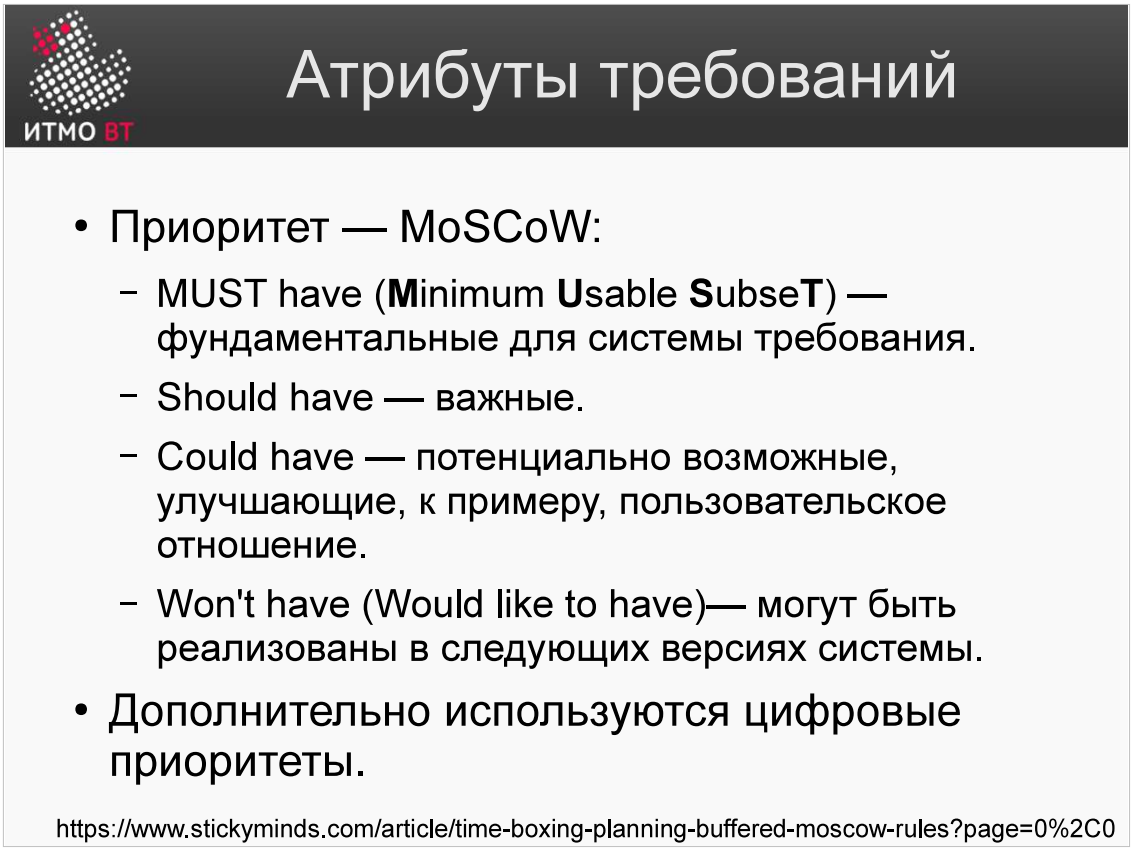
Одним из основных среди них является способность системы к расширению и масштабированию, и, соответственно, выполнению большего объема обработки данных, транзакций или пользователей. Возможность масштабирования защищает инвестиции заказчиков в разработку ПО и позволяют использовать одну и ту же систему в условиях потенциально возрастающей нагрузки.

Способность системы к поддержке группой обслуживания формирует специальные требования к возможностям остановки частей системы без прерывания обслуживания запросов пользователей. В большинстве случаев необходимы специальные архитектурные решения для обеспечения данной возможности.

Требования к совместимости позволяют использовать различные операционные системы, версии продуктов, браузеров и пр. совместно с разработанным ПО. Отдельно выделяются системные требования и минимальные требования к установке системы, например, объём ОЗУ, количество и частота процессоров и пр.

Способность системы функционировать в различных конфигурациях необходима тогда, когда одно и то же ПО может нести различные полезные функции в зависимости от потребностей и задач пользователя. Например, пользователь может использовать базы данных от разных поставщиков и разработчик может самостоятельно задавать параметры подключения к таким базам.

Возможность работать с разными языками в разных регионах земного шара формируется требованиями к локализации разрабатываемого ПО.



Требования могут быть помечены различными атрибутами, которые помогают менеджерам сортировать их. Одним из устоявшихся в разработке подходов является приоритизация требований по критерию MoSCoW. Данный критерий появился не в вычислительной технике; в соответствии с ним приоритеты требований разбиваются на несколько категорий. Следует заметить, что разработка системы ведётся в ограниченном временном промежутке, и не всегда есть возможность реализовать все требования в этом цикле разработки.

MUST have (Minimum Usable SubseT) — фундаментальные для системы требования, без которых финальная реализация системы не имеет смысла. Например современный телефон немыслим без возможности звонить другому абоненту и принимать звонки.

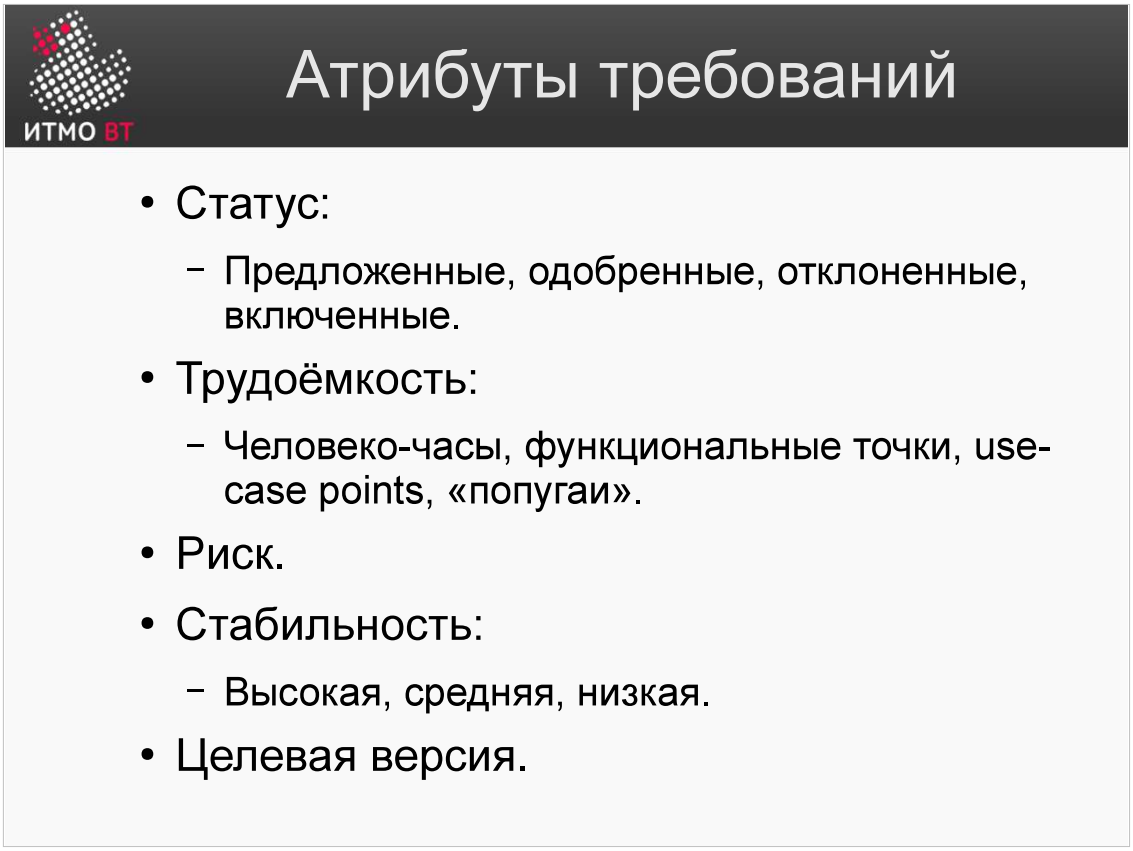
Should have — важные требования, которые следует реализовать при наличии времени в процессе разработки. Важным требованием, например, может быть возможность совершать звонки нажатием на одну кнопку телефона для вызова экстренных служб.

Could have — потенциально возможные: улучшающие, к примеру, пользовательское отношение к разрабатываемой системе, например, логичная структура меню телефона.

Won't have (Would like to have) — могут быть реализованы в следующих версиях системы, так как текущие временные рамки не позволяют провести их разработку.

На момент появления первой версии iPhone от компании Apple не существовало интерфейса взаимодействия с телефоном при помощи пальцев, которое явилось настоящим источником wow-эффекта. Очевидно, что при разработке это требование было помечено как "MUST have", и привело к удовлетворению большего количества пользователей. Поэтому такие требования, как критичные для прибыли, имеют максимальный приоритет.

Иным способом приоритизации требований является задание им цифрового показателя приоритета — например, от 1 до 10.



Требования имеют и другие атрибуты. Если требований много, и они имеют сложную организацию, то у требования должен быть статус (например, принято ли оно к исполнению или нет), и существовать отдельный рабочий процесс принятия требований в разработку.

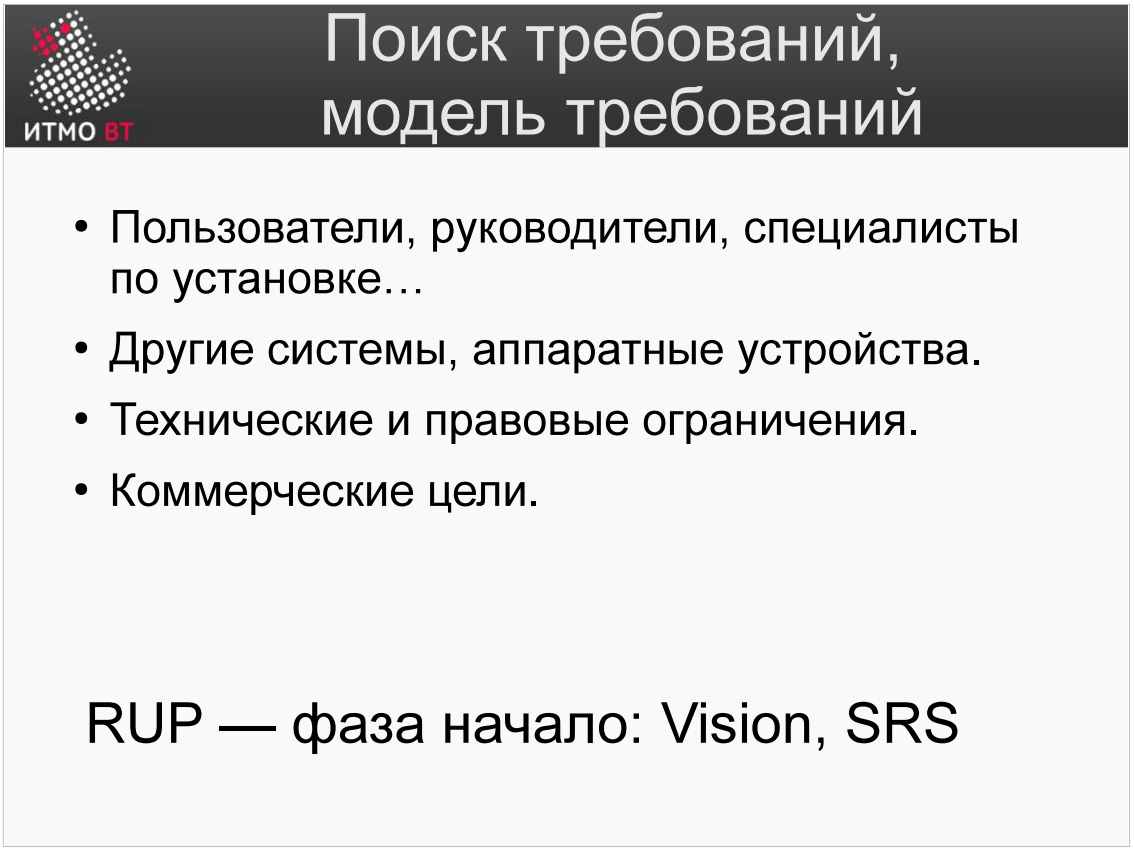
Особняком стоит трудоёмкость реализации требования, которая связана с вероятностью реализации в заданный срок. Отдельное искусство — это максимальная приближенность оценки до разработки к своему реальному значению, измеренному после реализации. На слайде указаны самые известные единицы измерения трудоёмкости. В последнее время пользуется популярностью безразмерная единица оценки работы, известная как "попугаи". Для оценки "попугая" берётся некое эталонное функциональное требование, и данная работа выполняется средним человеком в команде. Время, потраченное на данную работу, принимается за одного "попугая". Все остальные требования отсчитываются относительно этой величины. Благодаря этому, не существует жёсткой привязки к конкретным величинам. Другие единицы измерения трудоёмкости включают в себя человеко-часы, функциональные точки и use-case points.

Еще одним атрибутом требований являются риски. Риски — это факторы, которые влияют на разработку данного требования, они будут рассмотрены далее.

Стабильность характеризует частоту изменения требований.

Целевая версия — это та версия, в которую планируется включить реализацию данного требования.

С течением времени атрибуты изменяются и требуется периодическая повторная их оценка. На это очень сильно может влиять конкуренция в бизнес-среде, которая и приводит к изменению приоритетов в реализации.



В начале проекта необходимо чётко обозначить все потенциальные источники требований.

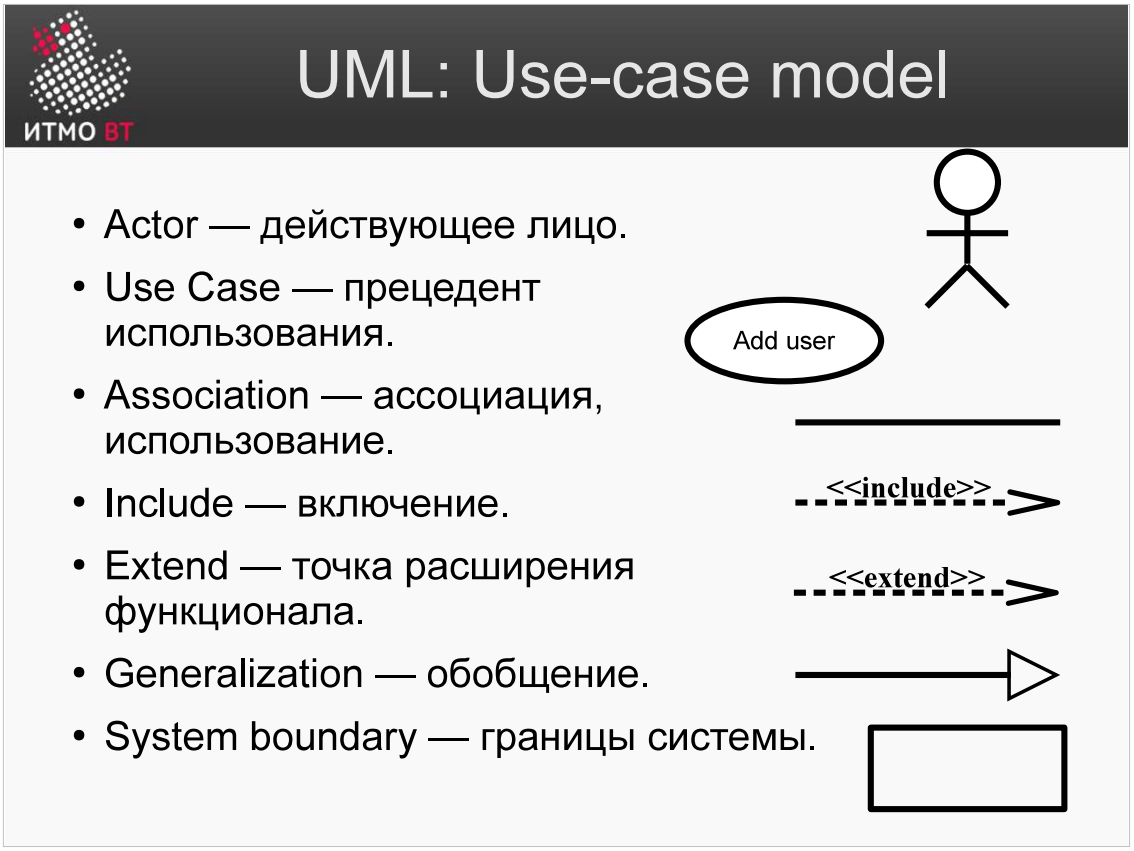
Основными такими источниками являются те, кто будут участвовать в разработке продукта, в эксплуатации и работе с ним. Например, человек, отвечающий за установку, естественным образом является ценным источником сведений о том, как производится установка системы, какие при этом могут возникать проблемы и какие типичные конфигурации оборудования используются у заказчика.

Важным источником также являются руководители проекта. Критерием эффективности для руководителя являются потраченные ресурсы и полученная прибыль. И, так как они являются источником доходов самих разработчиков, то их требования являются, возможно, наиболее значимыми.

Помимо людей, источниками являются техническая литература, другие системы, законодательство и другое.

Совокупность требований и их аттрибутов составляет модель требований, которая однозначно определяет, что необходимо разработать. Модель требований в RUP полно описывается в документе SRS (Software Requirements Specification — требования к программному продукту.

Кроме SRS, В рамках фазы "Начало" RUP создаётся документ под названием Vision (Концепция). Он содержит высокоуровневое описание требований и часть экономического обоснования. Этот документ можно предоставить потенциальным инвесторам для рассмотрения при получении финансирования, например, в рамках стартапа.



Use-case модель определена в RUP как инструмент графического отображения требований для упрощения взаимодействия заинтересованных лиц. Напомним основные элементы UML, которые могут использоваться на этой модели.

Главным элементом модели является действующее лицо или эктор (в сообществе разработчиков прижился такой вариант русского произношения) — пользователь, который непосредственно взаимодействует с самой системой. Под пользователем указывается его роль. Экторы никогда не являются частью системы. Время также может являться эктором.

Внутри прецедента использования пишется глагол или глагольная фраза, которая указывает, на то, что именно должна сделать система (пример: "Пользователь вводит логин и пароль"). Прецеденты одних модулей программы могут служить экторами для других модулей.

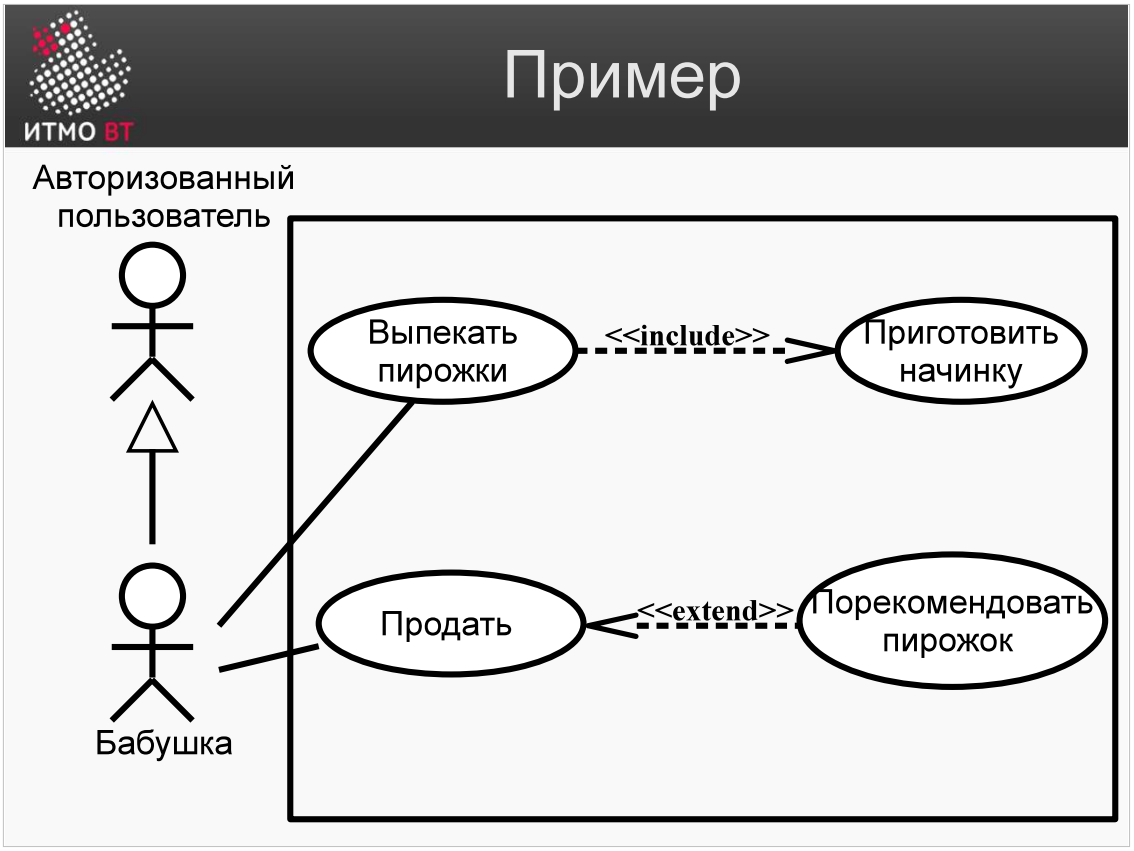
Association (ассоциация) — на диаграмме прецедентов связь, указывающая на то, что эктор использует систему и выполняет указанный Use Case.

В диаграмме прецедентов используются стереотипы связи зависимость, позволяющие организовать требования. Стереотип Include (включение) организует иерархию прецедентов использования системы и позволяет включать одну общую деятельность в несколько Use Case'ов.

Стереотип Extend (точка расширения функционала) указывает расширение от базового функционала. Например, в системе выдачи книги точкой расширения является взимание штрафа при просрочке возврата, т.е. при наличии определённых условий задействуется дополнительный функционал. Следует обратить внимание, что незнакомые с UML разработчики часто путают ее с обобщением.

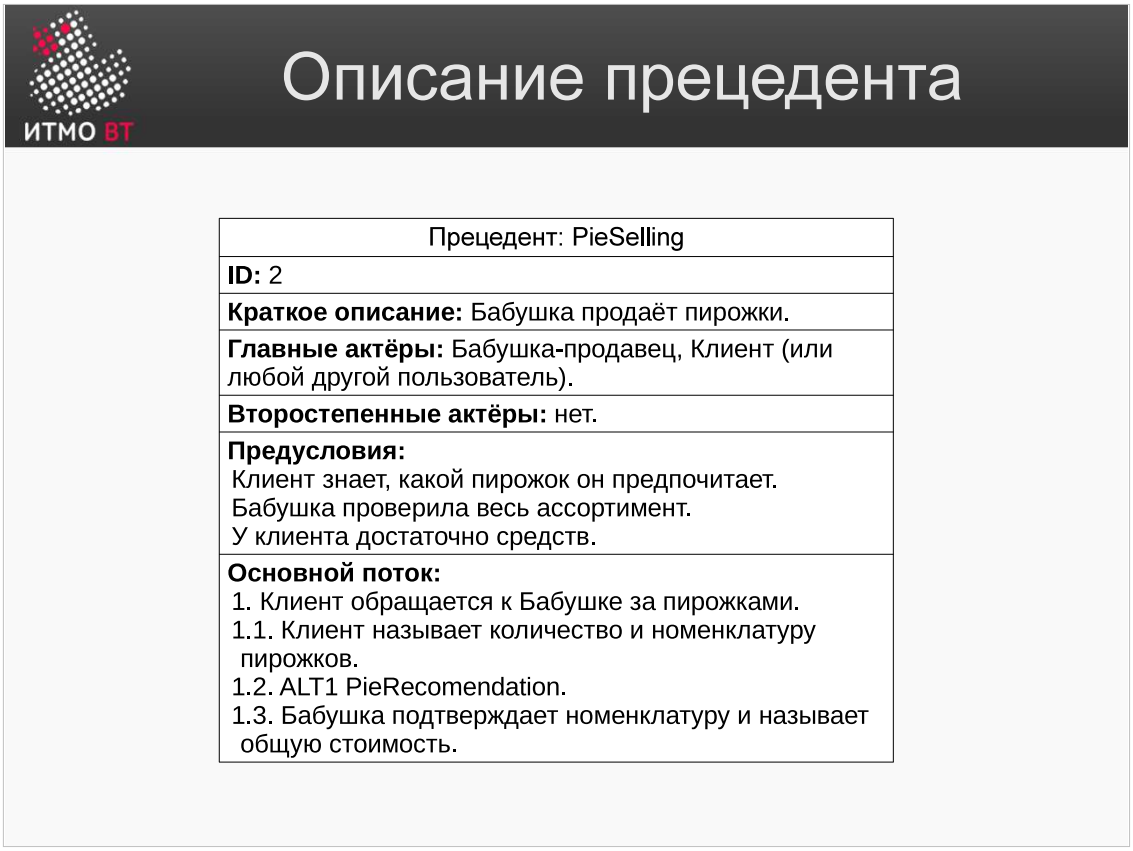
Связь Generalization (обобщение) — это так называемая связь "is a", показывающая, что данный объект является потомком более высокого класса объектов. Например, класс "утка" может являться потомком класса "птица".

Элемент System boundary (границы системы) важен тогда, когда система разбита на подсистемы, и необходимо разграничить их содержимое.



Предположим все пожилые дамы нашего дома решили объединится и начать печь пирожки на продажу. Естественно, им потребуется информационная система.

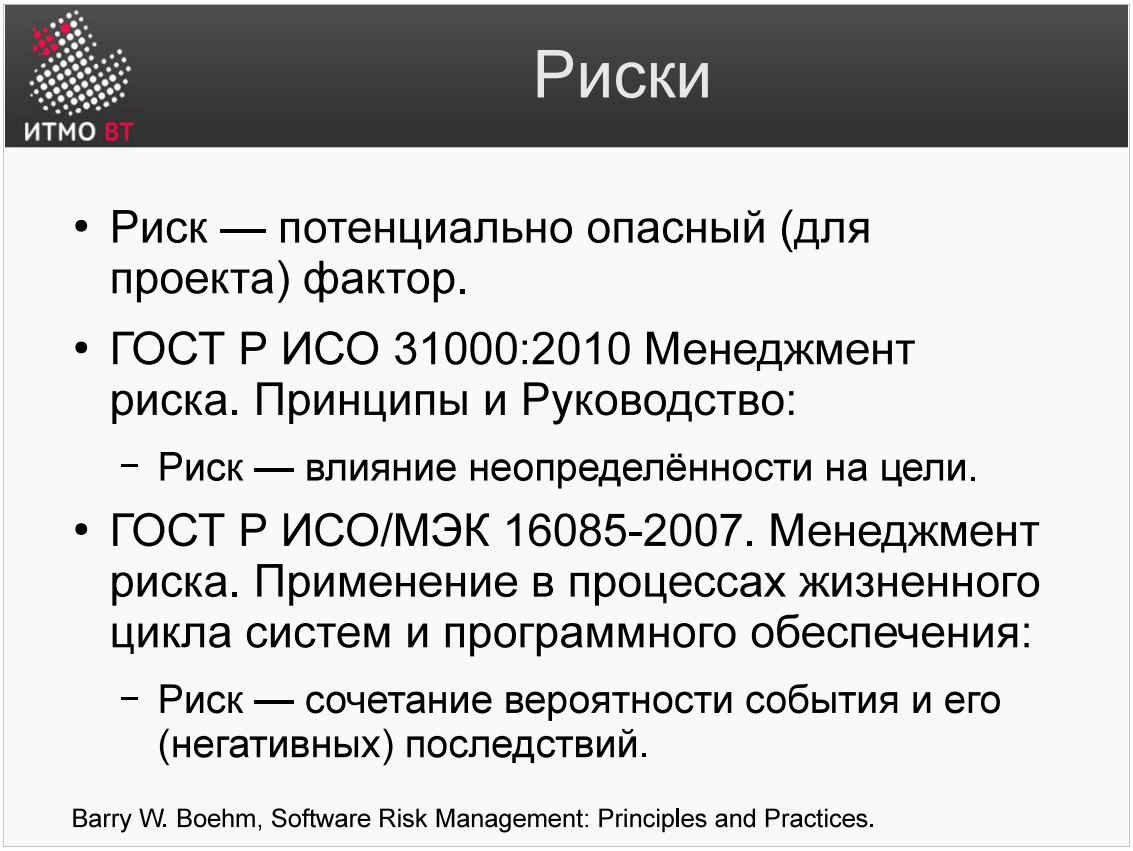
В системе «Домашние пирожки» выпуска и продажи пирожков актор «Бабушка», которая является потомком актора «Авторизованный пользователь», может «Выпекать пирожки» и «Продать» их. Use Case «Выпекать пирожки» включает в себя деятельность «Приготовить», которая также может быть включена как часть другого прецедента, например «Выпекать торты» для актора «Кондитер». Расширяющим функционалом прецедента «Продать» является прецедент «Порекомендовать пирожок», если покупатель не может определиться с выбором. Прецеденты, которые связаны с помощью стереотипа «Extend», могут описываться в модели с помощью так называемых альтернативных сценариев.



Прецедент: PieSelling
ID: 2
Краткое описание: Бабушка продаёт пирожки.
Главные актёры: Бабушка-продавец, Клиент (или любой другой пользователь).
Второстепенные актёры: нет.
Предусловия: Клиент знает, какой пирожок он предпочитает. Бабушка проверила весь ассортимент. У клиента достаточно средств.
Основной поток: 1. Клиент обращается к Бабушке за пирожками. 1.1. Клиент называет количество и номенклатуру пирожков. 1.2. ALT1 PieRecomendation. 1.3. Бабушка подтверждает номенклатуру и называет общую стоимость.

В процессе дальнейшей разработки требований диаграммы прецедентов используются только как изначальная модель. Дальнейшее развитие прецедентов связано с тем, что требования к системе формулируют в виде текстового описания, графических интерфейсов пользователя и других способов. В качестве примера приведен прецедент PieSelling — продажа пирожков.

Ключевое слово extend выражается в виде альтернативного сценария. Существует основной поток событий, где продаются именно те пирожки, и в том количестве, как это было указано клиентом. Если срабатывает условие активизации альтернативного прецедента, то в определенный момент времени вызывается сценарий, который обычно описывает дополнительные действия системы по обработке действий пользователя.



Анализ рисков пришел в разработку ПО из бизнес-практик и стал со временем основополагающим элементом цикла разработки. Риск определён как потенциально опасный (для проекта) фактор, который может привести к срыву сроков реализации проекта или потенциальной невозможности такой реализации.

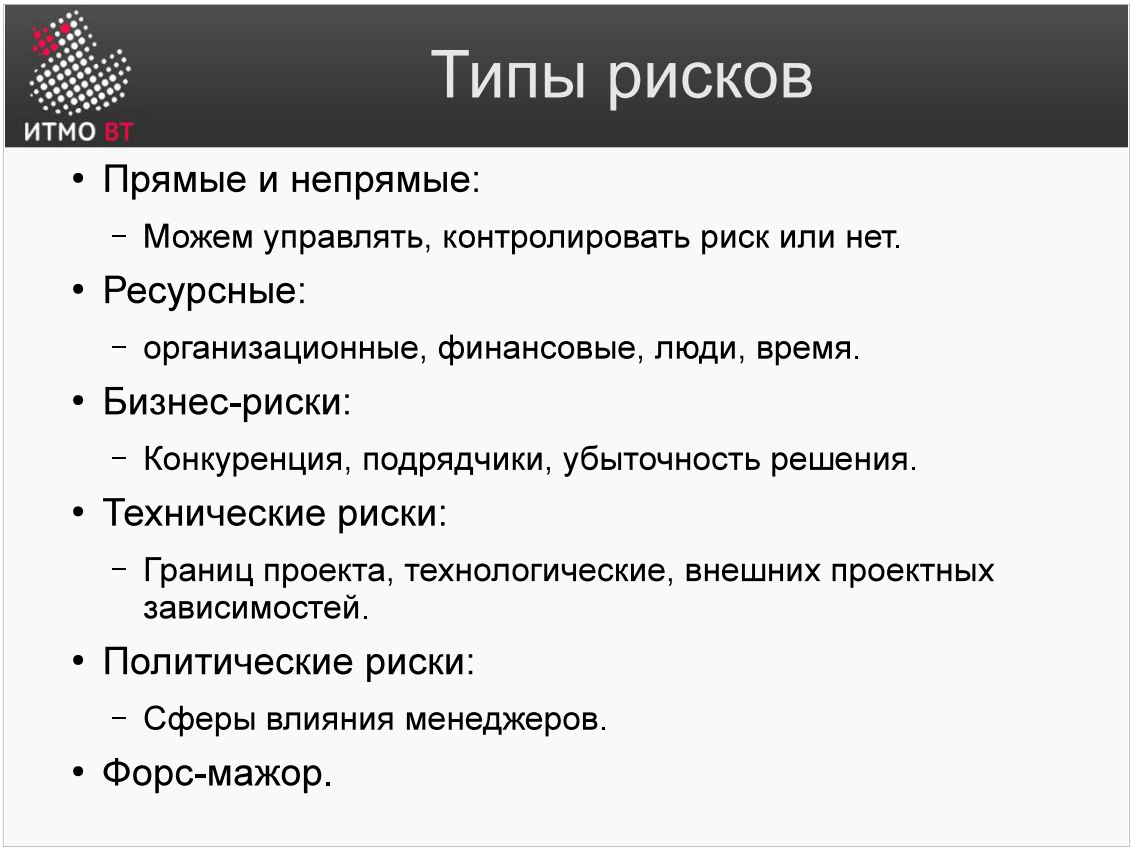
В настоящие время разработаны государственные и международные стандарты, определяющие и стандартизующие принципы управления рисками. Данные стандарты представлены на слайде.

«ГОСТ Р ИСО 31000:2010 Менеджмент риска. Принципы и Руководство» определяет общие принципы управления рисками, относящиеся к любой области деятельности.

«ГОСТ Р ИСО/МЭК 16085-2007. Менеджмент риска. Применение в процессах жизненного цикла систем и программного обеспечения» определяет принципы управления рисками в области разработки ПО.

В данных документах определение риска различается. Риск в первом из них определён как влияние неопределённости на цели, а во втором риск — это сочетание вероятности события и его (негативных) последствий на разработку системы в срок и с заданным бюджетом.

Напомним, что в области разработки программного обеспечения управление рисками впервые ввел Бари Боэм. Ссылка на его работу, посвященную рискам приведена на слайде.



Риски могут быть прямыми и непрямыми. Прямыми рисками можно в явном виде управлять: воздействовать на них, уменьшать их вероятность, реагировать на них. Непрямые риски возникают по внешним причинам и не поддаются управлению, можно только принять на себя их последствия. Отсюда существуют различные методы работы с прямыми и непрямыми рисками.

Основная задача состоит в нахождении источников рисков и минимизации последствий их наступления. Далее приведены виды рисков по их источнику и иным особенностям:

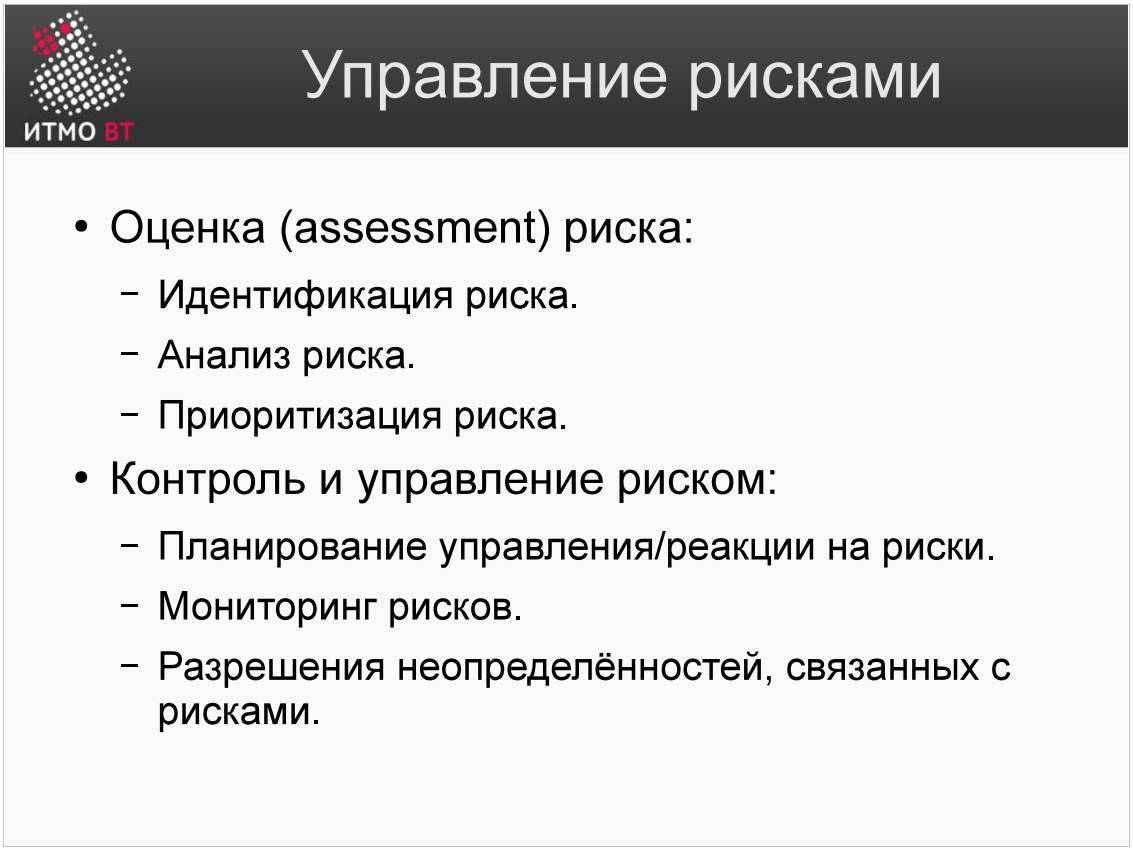
1) Ресурсные риски связаны с недостатком у компании времени, людей, денег, оборудования. Эти риски имеют отношение к особенностям и специфике конкретной организации, разрабатывающей проект. Данные риски являются управляемыми, так как возможно изменение выделяемых ресурсов.

2) Бизнес-риски появляются из-за взаимодействия с другими организациями и рынком в целом. Они определены конкуренцией, взаимодействием с подрядчиками или потенциальной убыточностью решения, т.е. отсутствием в дальнейшем прибыли от реализации и внедрения ПО. Управление бизнес-рисками сложно поддается управлению со стороны разработчиков.

3) Технические риски находятся в пределах компетенции разработчиков и поэтому являются ими управляемыми. Примеры технических рисков — отсутствие у разработчиков компетенции в применяемых технологиях.

4) Политические риски связаны с изменением сфер влияния внутри компании-заказчика. Например, с появлением нового менеджера могут измениться условия сделки. Это возможно предвидеть, но не всегда. Противодействием может быть упреждающая реакция на возможные изменения в компании (например, установление контактов с вероятным новым менеджером).

5) В отличие от политических рисков, повлиять на форс-мажор нельзя, и предугадать его тоже. К таким рискам относятся стихийные бедствия, изменение законодательства и т.п.



Управление рисками как дисциплина разбивается на две больших сферы деятельности: оценка риска и контроль и управление рисками.

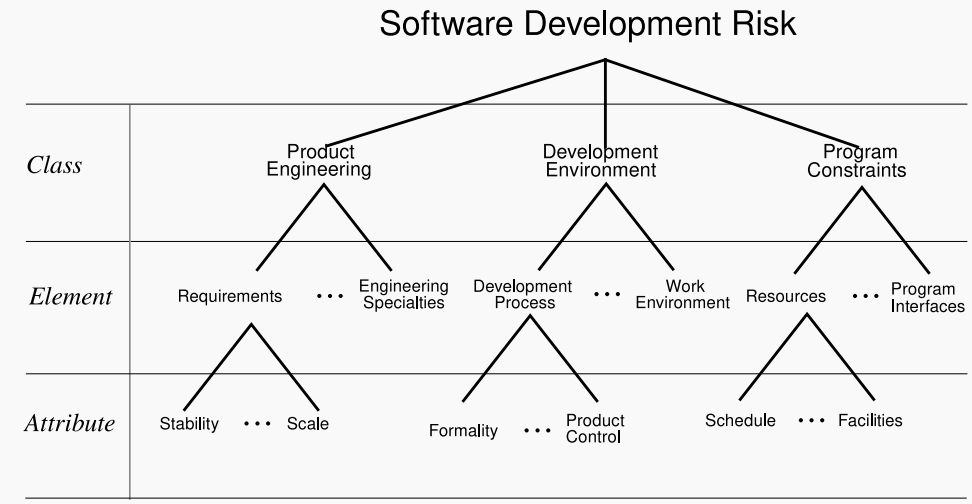
В процессе оценки (assessment) риска специалист по работе с риском проводит первичное знакомство с риском, анализирует его, определяет степень его серьёзности и продумывает план работы с ним. Оценка может производиться по разным методикам, например, по списку заранее подготовленных вопросов. Таким образом, внутри оценки риска производится его идентификация, анализ и назначение ему приоритета.

После того, как риск всесторонне оценён, его можно поместить под управление различных систем контроля риска, которые широко представлены на рынке. В случае, если система контроля определит его наступление или повышенную вероятность наступления, будет возможно предпринять корректирующие действия.

ИТМО ВТ

Идентификация рисков

- Известные, неизвестные и непознаваемые



Marvin J. Carr and other Taxonomy-Based Risk Identification

Рассмотрим подробнее идентификацию рисков. Откуда же внутри компании-разработчика могут возникать факторы неопределённости?

Марвин Дж. Карр в 1993 году предложил схему идентификации рисков на основе построения таксономии источников рисков. Все источники рисков в компании-разработчике распределяются по иерархической структуре. Первым ее уровнем являются классы рисков: риски, связанные с самой разработкой (Product Engineering); риски, связанные с окружением, где осуществляется разработка (Development Environment); и риски, связанные с программным обеспечением (Program Constraints). Каждый из классов состоит из элементов, а элемент из атрибутов, которые указывают на возможные источники возникновения рисков. В примере на слайде деятельность компании разбита на иерархию, и каждый её элемент проанализирован с точки зрения возможности возникновения ошибок.

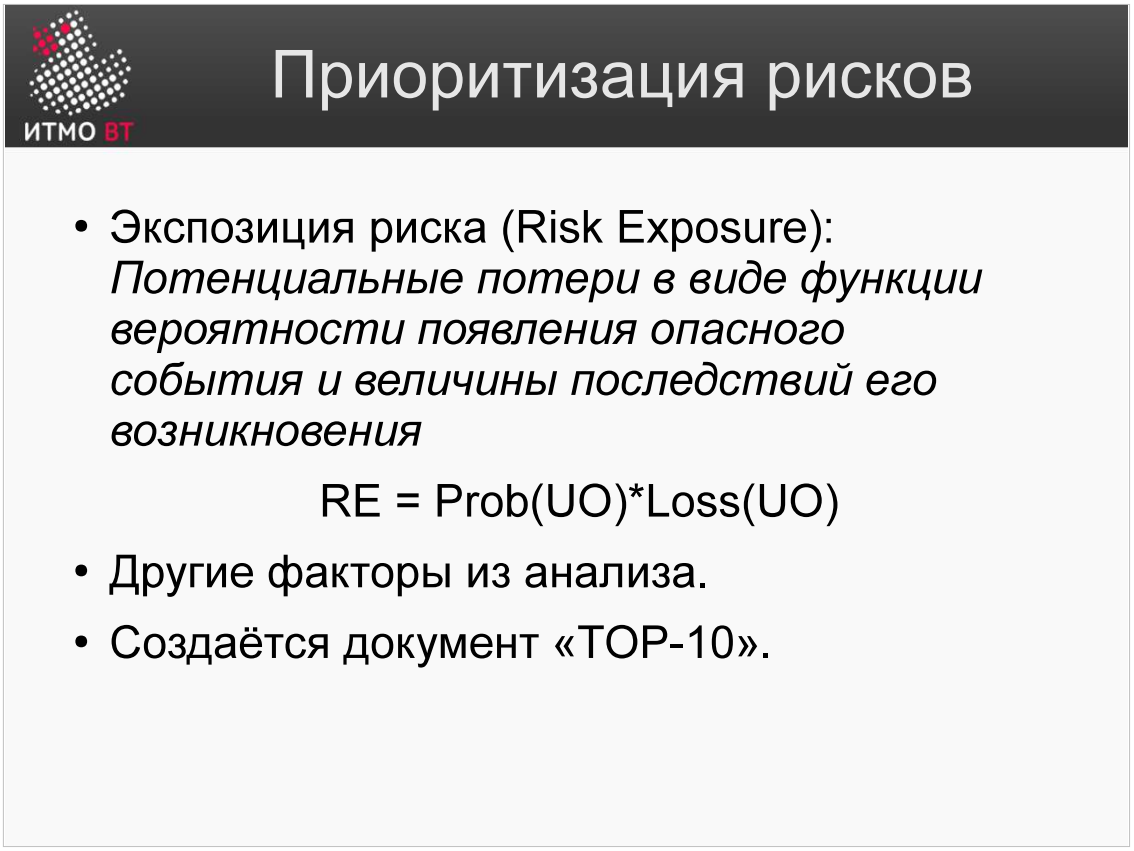
Также Карр и его коллеги разделили риски на известные, неизвестные и непознаваемые. Риск, который просто идентифицировать и определить место его возникновения в компании, называется известным. К примеру, технологическая некомпетентность части разработчиков в конкретной технологии при заключении контракта с использованием этой технологии достаточно очевидна. Место возникновения неизвестного риска можно предположить (например, при помощи детально описанной таксономии и опыта других команд разработки), но данная команда разработки ещё с ним не сталкивались. Место возникновения непознаваемых рисков невозможно предугадать, а способ борьбы разработать заранее.

После идентификации рисков необходимо провести их всесторонний анализ. Анализ рисков связан с выявлением скрытых взаимосвязей неопределённых ситуаций и источников рисков. Существует большое количество различных моделей и методов такого анализа; к ним относятся, например, стоимостной, где высчитывается в количественном выражении влияние риска при различном развитии обстоятельств.

Два основных параметра риска — вероятность его наступления и масштаб (магнитуда) возможных потерь. Так как работа осуществляется с людьми, то данные параметры обычно задаются нечётко, с учетом субъективного отношения человека. Как правило, пятибалльной шкалы оценки параметров бывает достаточно: низкие, незначительные, средние, значительные и высокие вероятность или потери.

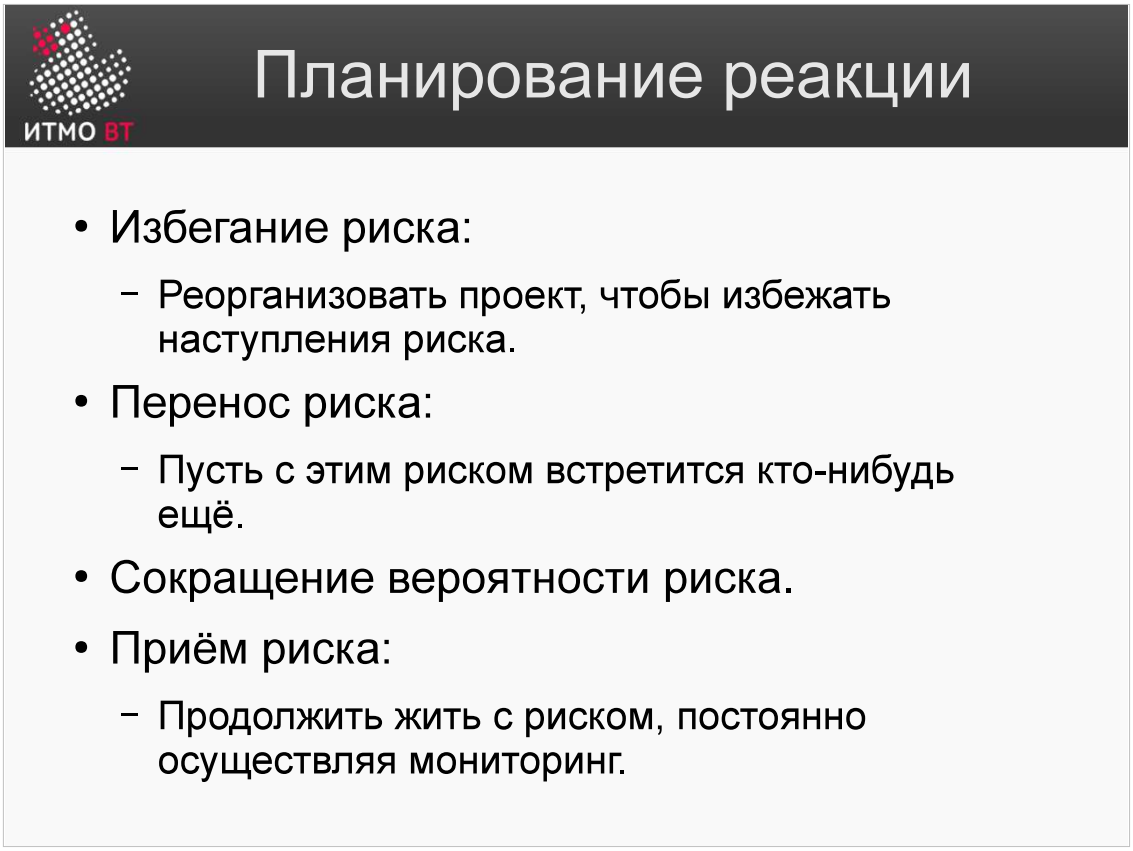
Параметры могут быть заданы математической вероятностью и объёмом денежных потерь. В этом случае влияние риска можно оценить при помощи математических моделей.

Риски также могут быть заданы шкалами. В таком случае формируются группы параметров в визуально понятной форме. В явном виде необходимо указать приближение к наступлению риска. Например, можно однозначно зафиксировать, что если заказчик задерживает оплату этапа разработки от нуля до пяти дней, скорее всего это не скажется на оперативной деятельности разработчика. Заработную плату выдают обычно два раза в месяц, и если задержка составит более 6 дней, то риск не выплатить зарплату вовремя существенно повышается, что может вызвать негативные последствия для мотивации работников. Риск оказывается в желтой зоне. При увеличение задержки финансирования более 15 дней команда разработчиков точно рискует не получить зарплату вовремя, и может потерять необходимую скорость, если разработчики не выйдут на работу.



Введём понятие экспозиции риска (Risk Exposure), которая определяется произведением вероятности наступления риска и величины денежных потерь.

После подсчета экспозиции риска создаётся список, отсортированный по величине экспозиции, и на основе этого списка создается документ «TOP-10 рисков». Данный документ необходимо постоянно обновлять по мере продвижения разработки.

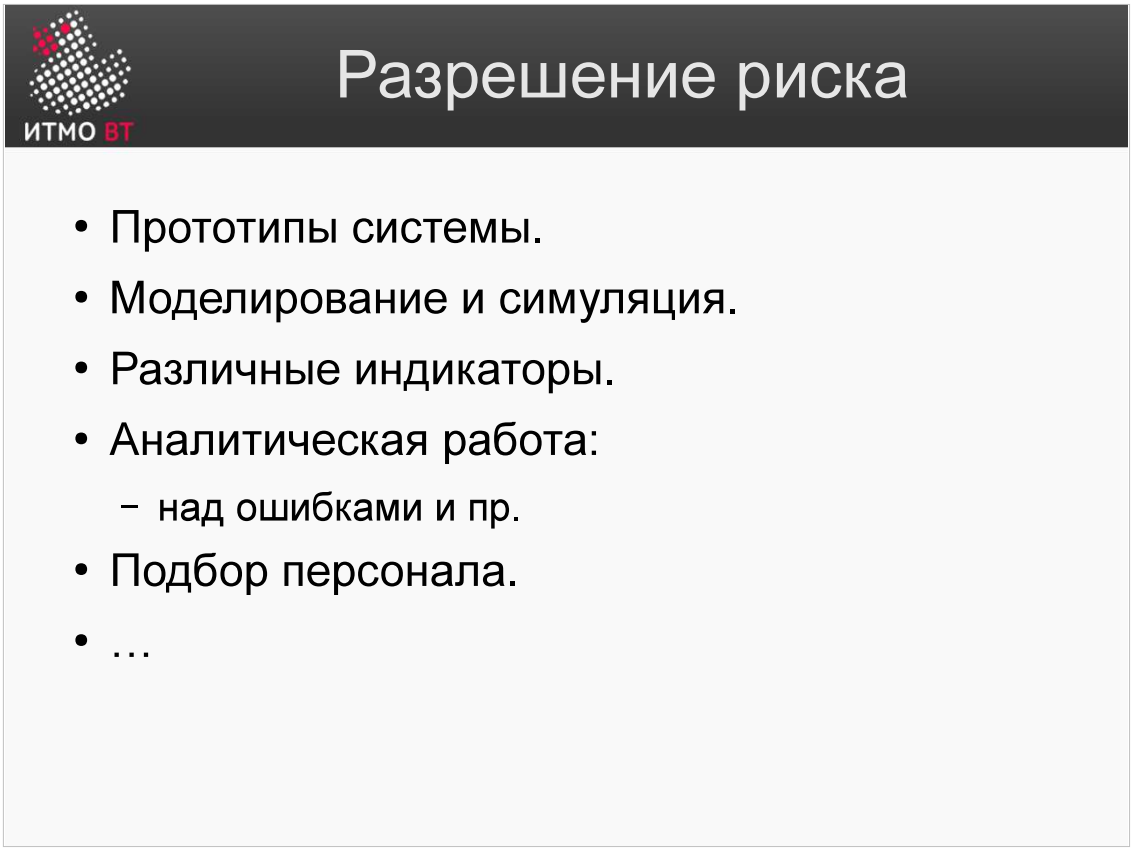


Что можно сделать с риском, который может потенциально наступить? Первый и очевидный способ реагирования на риск — попытаться его избежать. Для этого необходимо разработать комплекс мероприятий, которые помогут отсрочить или исключить вероятность его наступления. Например, в случае недостаточной квалификации, можно провести реорганизацию проекта (нанять новых людей, обучить уже существующих сотрудников и т.д.).

Второй способ реакции — перенос риска. Если известно, что риск должен наступить, можно сделать так, чтобы под него попал кто-то другой (другой человек, другая организация и так далее). Например, если для проекта нужно дополнительное финансирование, то можно в явном виде увязать вопрос проведения работ с заказчиком: нет финансирования — нет работ. Тогда риск того, что проект не будет сделан в срок, переносится на заказчика.

Возможно также пытаться сократить вероятность наступления риска. К примеру, если мы точно понимаем, что данная технология недостаточно опробована, можно попытаться обосновать использование другой, более известной, и, соответственно, менее рискованной в реализации. Правда в этом случает нужно понимать, что и программное обеспечение может получиться менее востребованным.

Также возможен приём риска. В этом варианте мы соглашаемся с тем, что риск может наступить, заранее составляются планы действий, какие действия необходимо предпринимать в случае его наступления. В примере с незнакомой или не опробованной технологией, можно сразу разработать план массового обучения сотрудников до начала разработки.



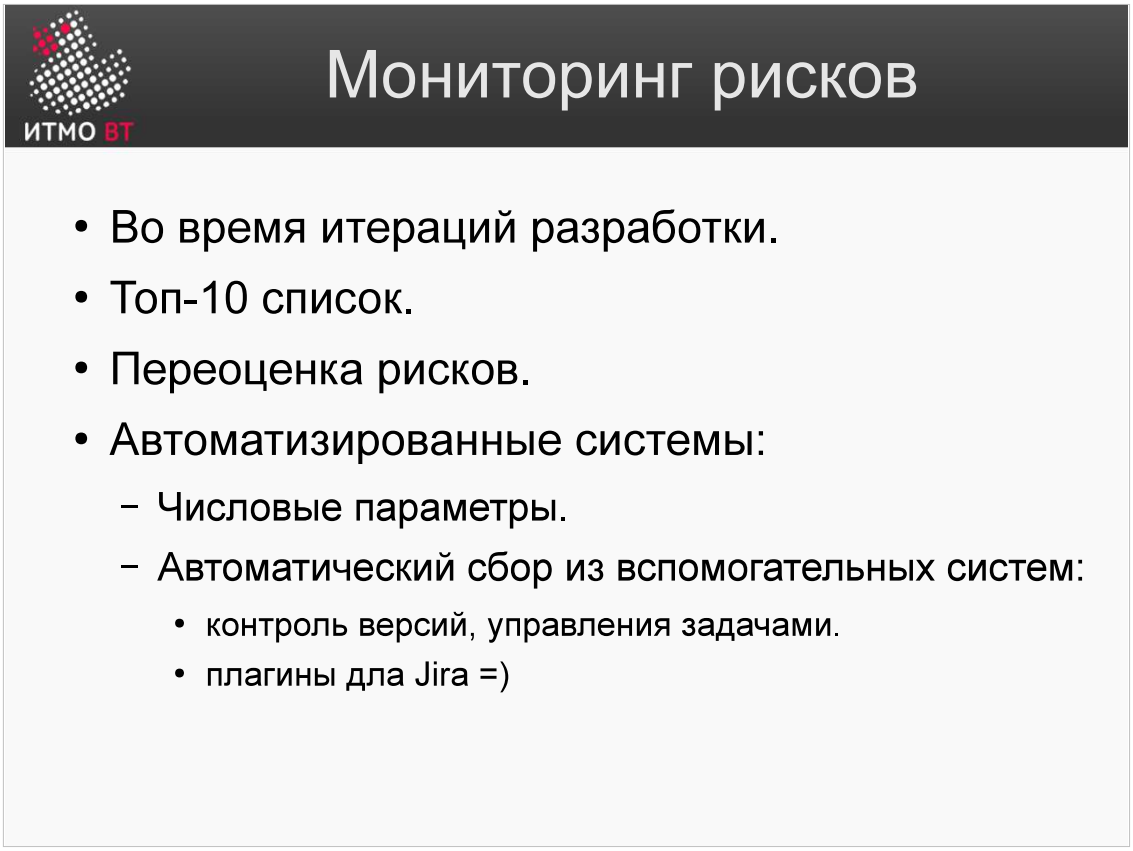
Для того, чтобы риск не наступил, возможно выполнение следующих организационных мероприятий:

1) Построение как можно большего числа прототипов системы. Каждый новый прототип приближает правильную оценку системы заказчиком (оценка scope, разрабатываемого функционала). За счет этого заказчик сможет более полно оценить то, что он получит в конце разработки. Чем больше прототипов — тем меньше вероятности всех рисков. С другой стороны, с увеличением количества прототипов увеличивается объём работ и расходуется больше средств.

2) Большую помощь может оказать построение различных моделей функционирования ПО. Построение моделей существенно дешевле, чем разработка ПО. Например, UX-модели пользовательского интерфейса позволяют пользователям оценить интерфейс в действии, «покликать» на пункты меню и сформировать пользовательское мнение. Существует большое число средств симуляции пользовательского интерфейса (в основном, платных).

3) Аналитическая работа, в том числе, над теми ошибками, которые произошли у команды разработчиков в этом или предыдущих проектах. После каждого наступления какой-либо нежелательной ситуации важно чётко отработать «работу над ошибками», и приложить все усилия, чтобы эта ситуация не повторилась.

4) Грамотная работа отдела кадров поможет подобрать квалифицированный персонал для решения конкретных задач. На реальном рынке труда не так просто найти необходимых специалистов.



Во время разработки риски необходимо непрерывно переоценивать. Менеджер проекта, к примеру, раз в неделю может анализировать существующий список рисков, формально проверять вероятность их наступления и менять список наиболее опасных.

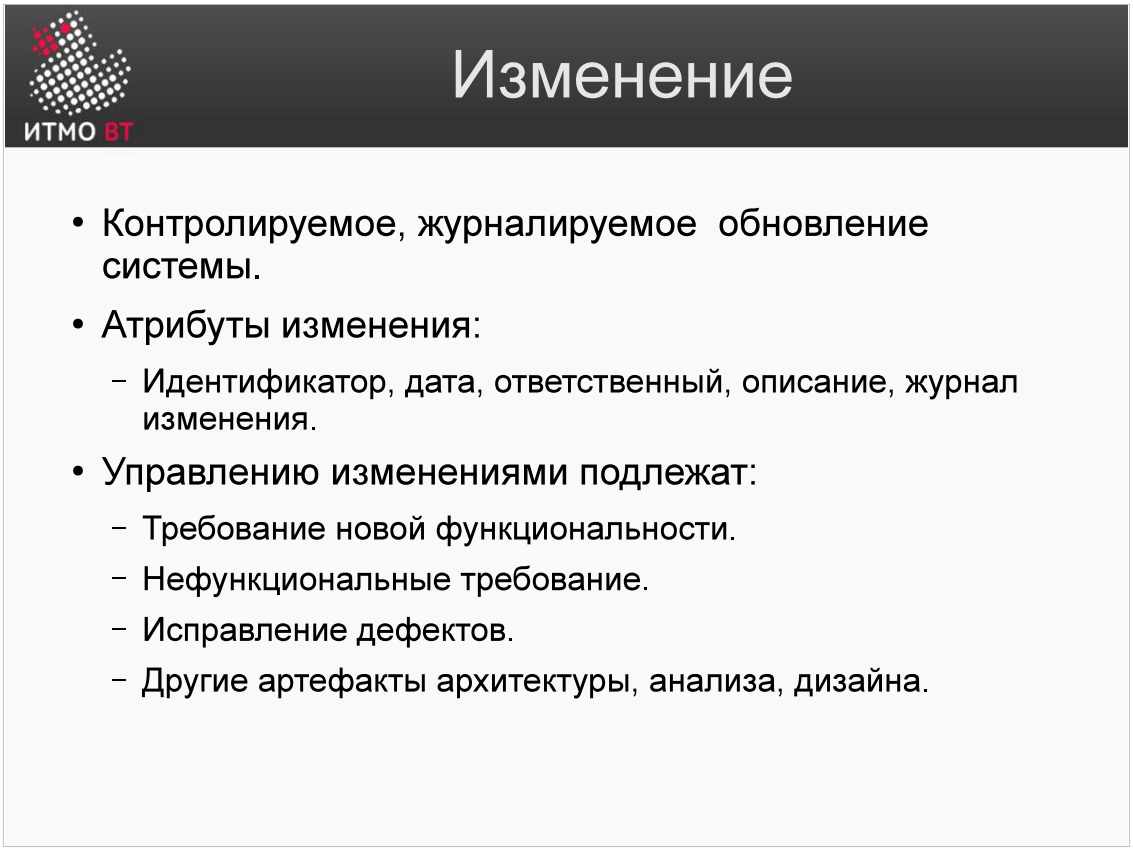
Полный список рисков может быть достаточно большим, и поэтому наиболее часто должны контролироваться 10 наиболее вероятных и деструктивных рисков. Данный список должен постоянно меняться по результатам недельного анализа всех рисков.

Большую помощь в этом могут оказать автоматизированные системы. Например, данные из систем контроля рисков и управления текущими задачами разработчиков могут обрабатываться и выводиться в виде дашбордов, которые будут показывать текущую производительность команды. Одно из наиболее известных средств автоматизации бизнес-процесса разработки, Jira, обладает большим набором различных плагинов для визуализации процессов командной разработки.

ИТМО ВТ

3

Управление изменениями программных продуктов

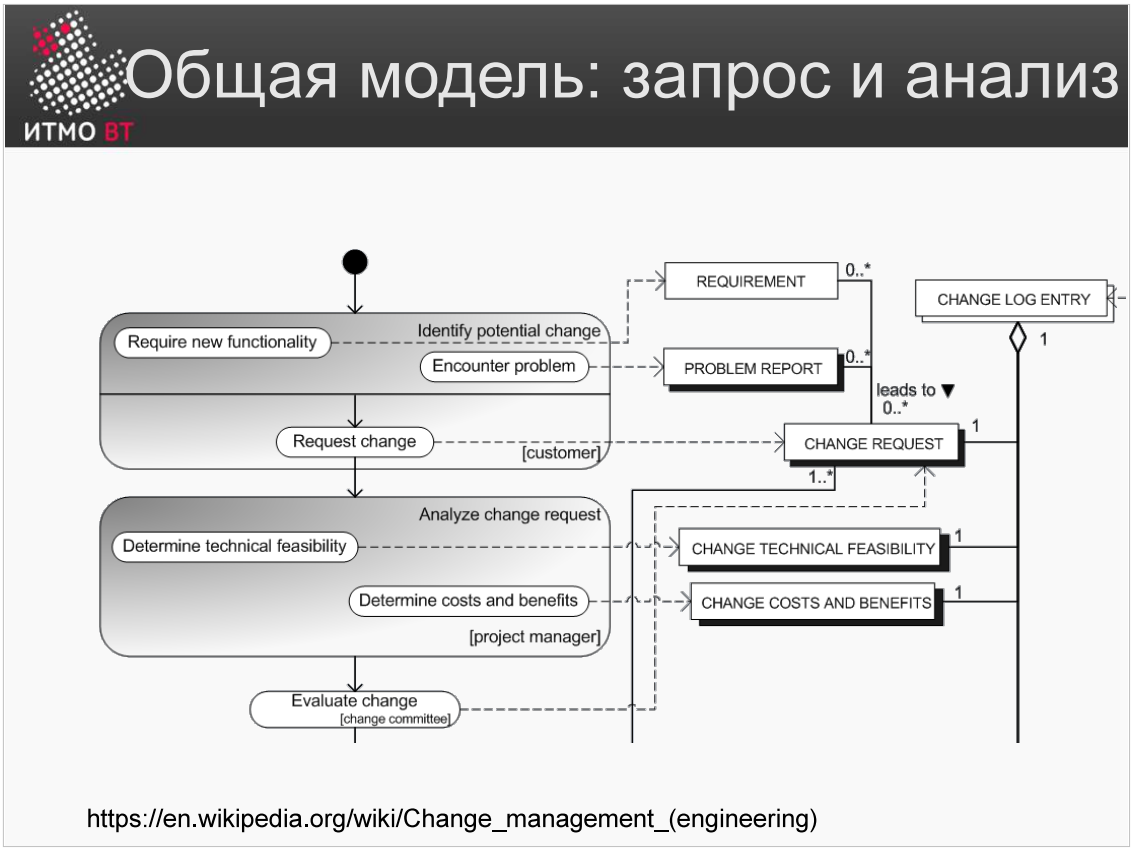


При увеличении количества людей, работающих над проектами, накапливающиеся изменения нарастают как снежный ком, и контроль и управление ими приходится организационно контролировать. Этим занимается дисциплина, называемая управление изменениями (change management). В экономике управление изменениями (например, в организации, ведущей хозяйственную деятельность), является основополагающим инструментом адаптации к меняющимся условиям бизнеса.

У сложных систем (в том числе и программных) невозможно, или очень сложно, менять систему одновременно в нескольких компонентах. Поэтому вносимые изменения обычно выстраиваются в последовательность отдельных действий, которые необходимо контролировать и фиксировать атрибуты каждого такого изменения в журнале.

Системы контроля версий являются примерами систем учета изменений и позволяют вести такие журналы. При помощи журнала можно просмотреть историю правок и оценить их влияние на поведение системы. Каждое изменение имеет атрибуты: идентификатор, дату, ответственного, описание, связанный с ним журнал изменения и т.д.

В разработке ПО не только изменения программного кода могут попадать под контроль — отдельному управлению изменениями обычно подлежат сами требования к ПО, выявленные дефекты, артефакты архитектуры, анализа и дизайна.



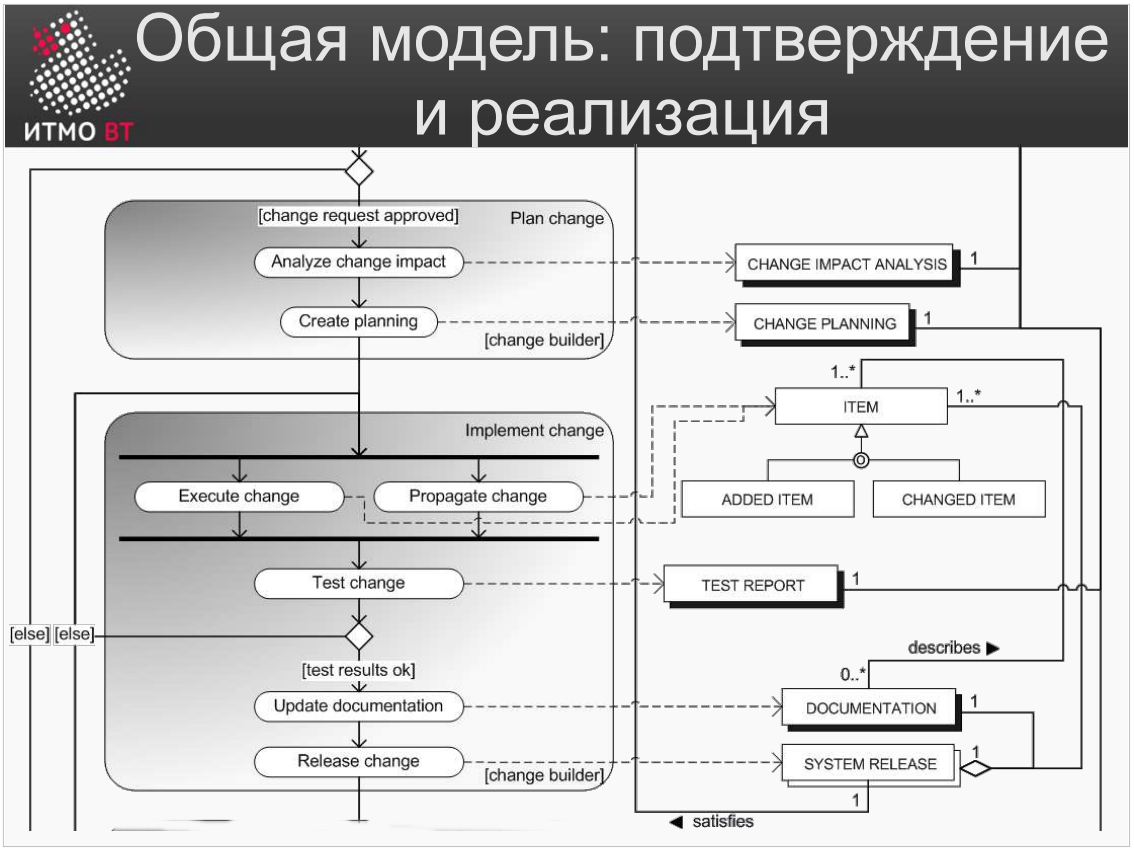
На слайде приведена модель процесса управления изменениями, который осуществляется в области разработки систем и, в том числе, в инженерии программного обеспечения. Все современные средства по управлению изменениями реализуют близкий к приведённому процесс.

В левой части указаны активности (деятельности) ролей процесса с целью формирования и изменения набора артефактов для ведения изменения. Справа показаны сами артефакты изменения, участвующие или возникающие в процессе. Следует заметить, что данная модель представляет собой гибрид двух UML-диаграмм — диаграммы деятельности и диаграммы классов верхнего уровня.

Заказчик (Customer) вносит изменения в систему ради появления новой функциональности (Require new functionality) или исправления выявленных им проблем (Encounter problem). Каждая из этих деятельностей приводит к созданию документа или материальных артефактов: Requirements (требования к системе) Problem report (отчёт о проблеме или ошибке). Оба типа запросов формируют т.н. запрос на изменение (Change Request, часто используется акроним CR), и на каждый такой запрос формируются записи в Change Log Entry.

После этого запросы на изменения поступают к менеджеру проекта (Project manager). У менеджера есть две необходимых деятельности, которые он должен осуществить в процессе анализа запроса на изменения: он определяет техническую необходимость и осуществимость изменений (technical feasibility), а также анализирует стоимость и преимущества для пользователя (costs and benefits). При этом первый фактор формирует документ или артефакт Change Technical Feasibility, а второй — Change Costs and Benefits. Всё это отражается в журнале изменений.

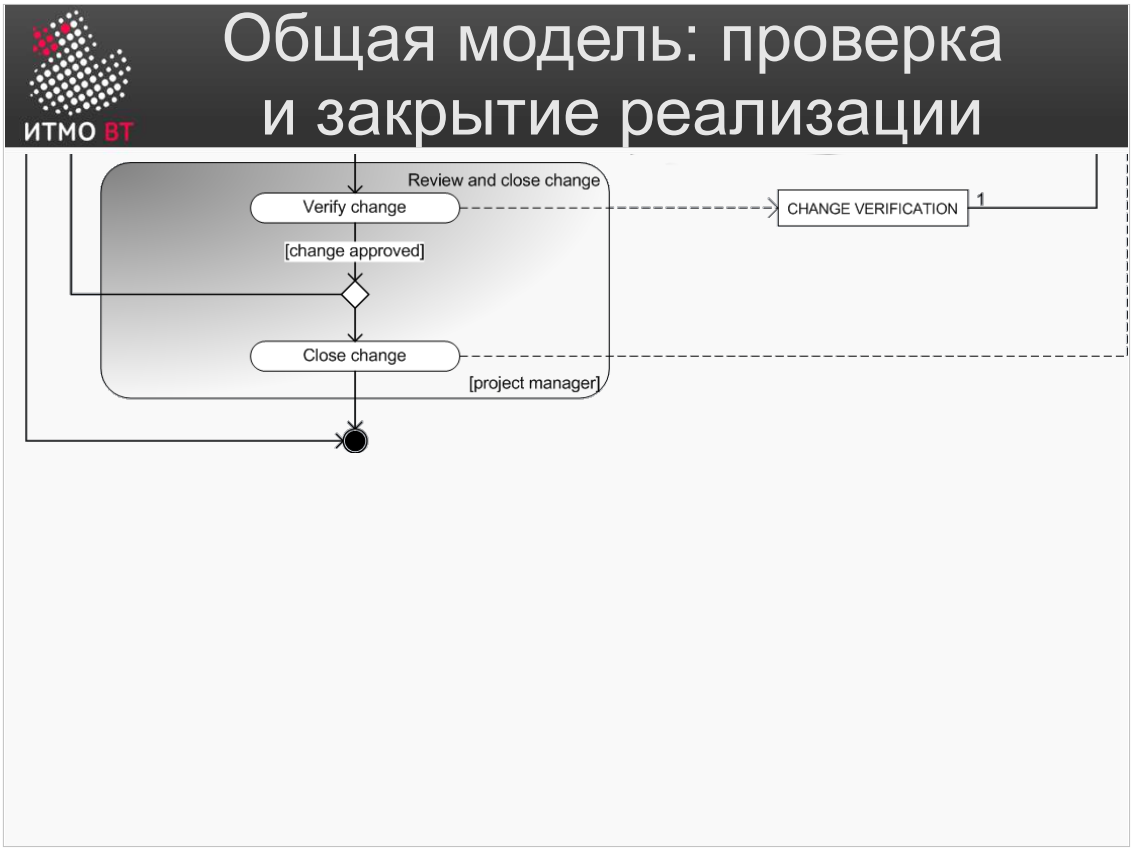
После этого запрос поступает к т.н. комитету по изменениям, который, оценив необходимость изменений и стоимость решения проблемы, меняет статус изменения в Change Request. Если запрос подтверждён, то производится анализ и реализация изменений, в противном случае запрос может быть отменён или отложен.



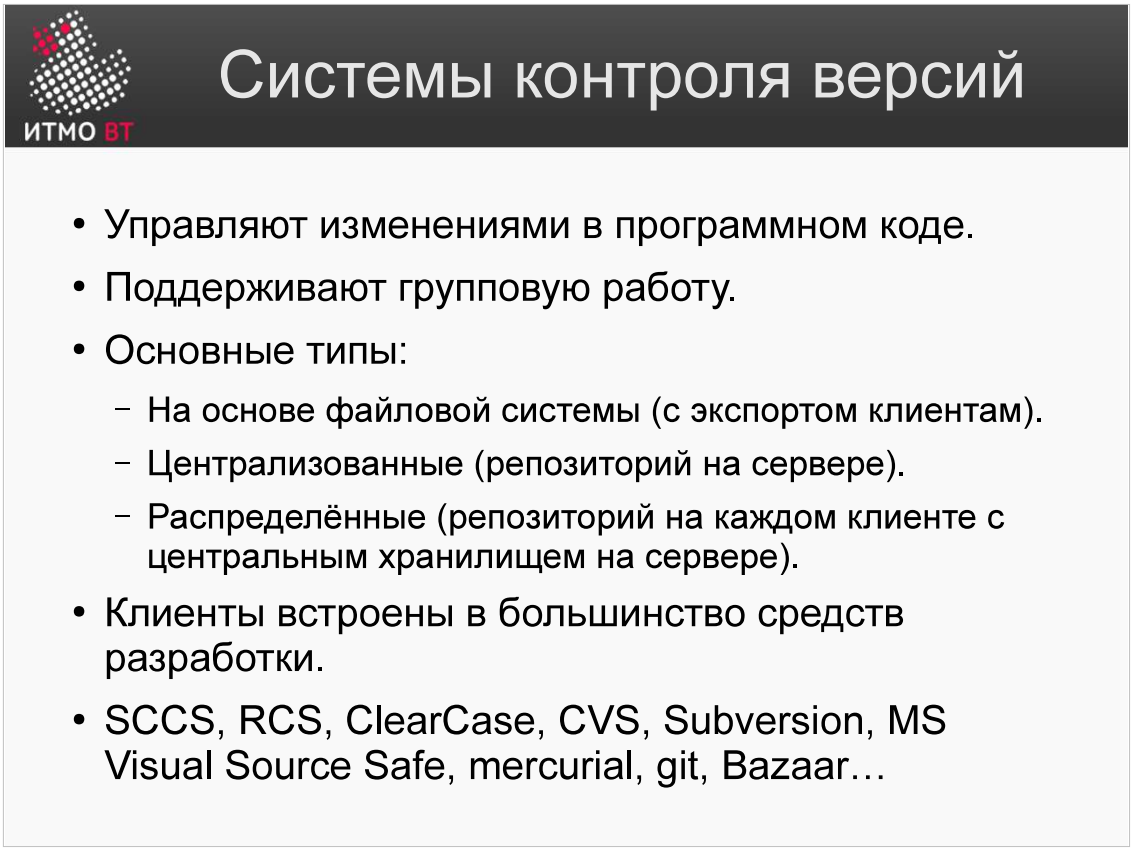
После принятия решения об изменении анализируется его возможное влияние на другие части системы и возможные последствия для пользователей (Analyze change impact). Необходимо помнить, что в сложных системах внесение изменений в одну часть системы почти всегда приводит к влиянию, и соответственно, изменениям в других частях, поэтому вопрос о влиянии изменения следует тщательно изучить.

После анализа организуется планирование проведения изменений в системе (Create Planning), в ходе которого определяются ресурсы, время и график осуществления действий, необходимых для внесения изменения. В результате этих двух действий появляются артефакты Change Impact Analysis и Change Planning.

Затем изменения передаются на реализацию. Роль, ответственная за неё (Change builder), создает объекты, совокупность которых составляет реализацию изменения (Items), например, исходный код. После этого производится тестирование, вносятся изменения в связанную документацию, и выпускается новый выпуск или версия продукта, «заплатка» и/или конфигурация. При этом создаются артефакты — отчеты о тестировании (Test Report), собственно, документация (Documentation) и выпуск системы (System Release), соответственно.



В конце процесса изменения передаются менеджеру проекта (и иногда — заказчику), проверяются (создаётся артефакт Change Verification), и формально утверждаются.



Сами по себе системы контроля версий существуют для управления изменениями в программном коде и поддерживают групповую работу нескольких человек над кодом одновременно, а также контроль над изменениями файлов, создаваемых пользователями системы контроля версий (т.е. программистами).

Существуют три основных типа систем контроля версий:

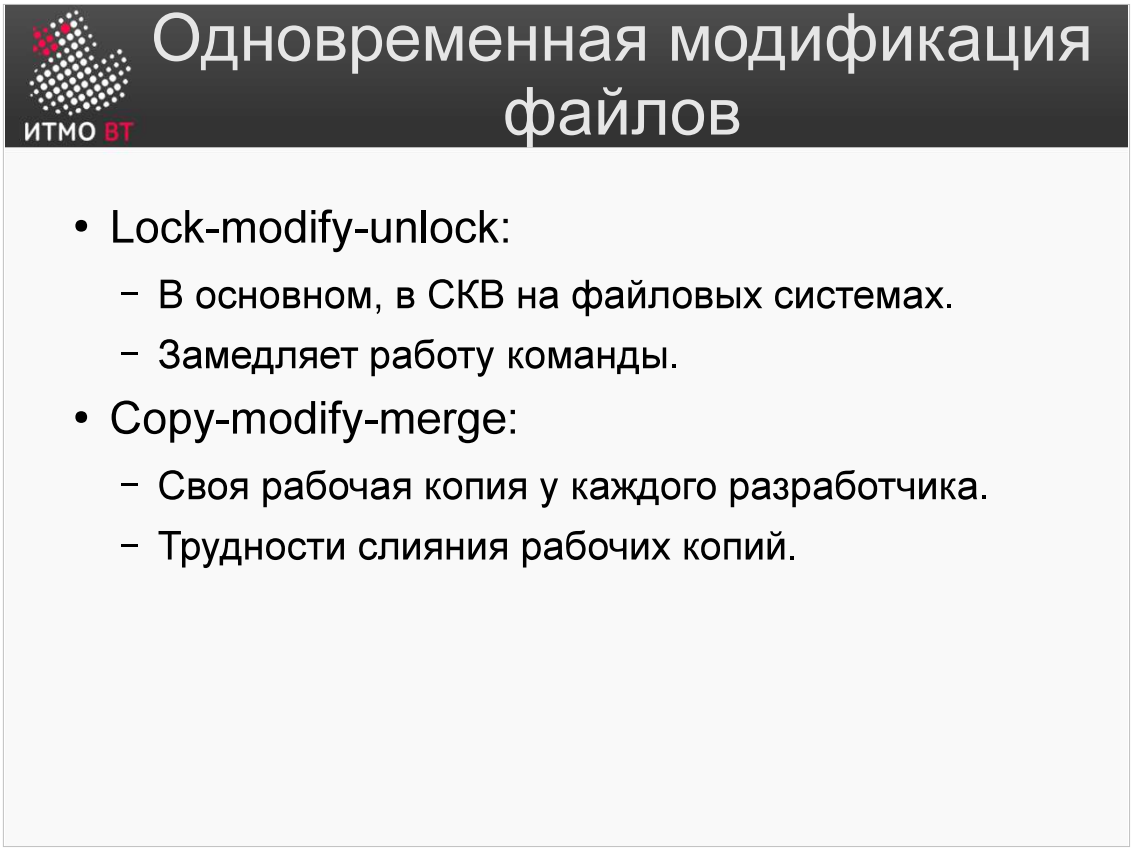
1) На основе файловой системы. Данный подход является устаревшим. Ранее разработчики пользовались центральным сервером с общим доступом к файлам. Такая система контроля создавала файлы слежения за текущей директорией и позволяла хранить и отслеживать версии только в рамках одной файловой системы. Также было возможно экспортировать файловую систему для доступа удалённых клиентов при помощи сетевых файловых систем (например, NFS).

2) Централизованная, с единым репозиторием — хранилищем исходным кодов проекта на сервере, и удалённым доступом клиентов по специальным протоколам. К таким системам относится Subversion.

3) Распределённая. В ней существует центральный репозиторий, из которого пользователи скачивают данные на свои локальные репозитории. Обратно в центральный репозиторий данные попадают после нескольких стадий локальных проверок. К распределённым системам относится Git.

В первых системах контроля версий (например, SCCS, который был интегрирован во всех потомках Unix System V) все действия производились при помощи командной строки, но в большинстве современных систем контроля версий существуют встроенные клиенты, автоматизирующие работу с версиями и организующие все необходимые действия в набор пунктов меню.

В настоящее время существует большое количество систем контроля версий, таких как ClearCase, CVS, Subversion, MS Visual Source Safe, mercurial, git, Bazaar и т.д.



Главной проблемой при работе с СКВ является необходимость одновременного редактирования одних и тех же файлов несколькими разработчиками. Одновременное редактирование было технически возможно достаточно давно, но оно приводило к войне правок, когда разработчики в рамках своих заданий вносили взаимно-конфликтующие изменения.

В мире систем контроля версий утвердились два подхода к решению данной проблемы:

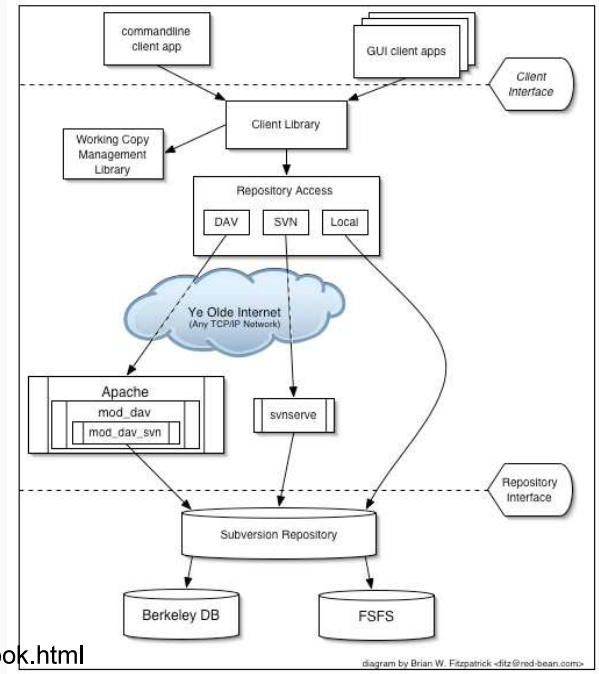
1) Подход Lock-modify-unlock, где при работе одного пользователя с файлом, он блокируется для других, и они не могут его модифицировать. Это замедляет работу команды, так как другим пользователям приходится либо ничего не делать, либо переключаться на другие задачи, не связанные с заблокированным файлом. Данный подход был характерен на системах контроля версий, связанных с общей файловой системой.

2) Copy-modify-merge, где каждый пользователь копирует себе весь репозиторий, работает с ним, и изменения всех пользователей сливаются (merge). В данном случае именно слияние рабочих копий разработчиков представляет собой основную проблему. Например, два разработчика могут одновременно изменить один и тот же код двумя различными способами. В этом случае, при попытке сохранить изменения, второй разработчик (тот, который делает коммит по времени вторым, после первого) не сможет этого сделать и будет вынужден разрешать конфликт (во многих случаях — совместно с первым разработчиком).

ИТМО ВТ

SVN

- Централизованная, репозиторий хранится на сервере.
- Доступ к репозиторию:
  - Svnserve.
  - Apache + модуль mod_dav_svn.



http://svnbook.red-bean.com/en/1.7/svn-book.html

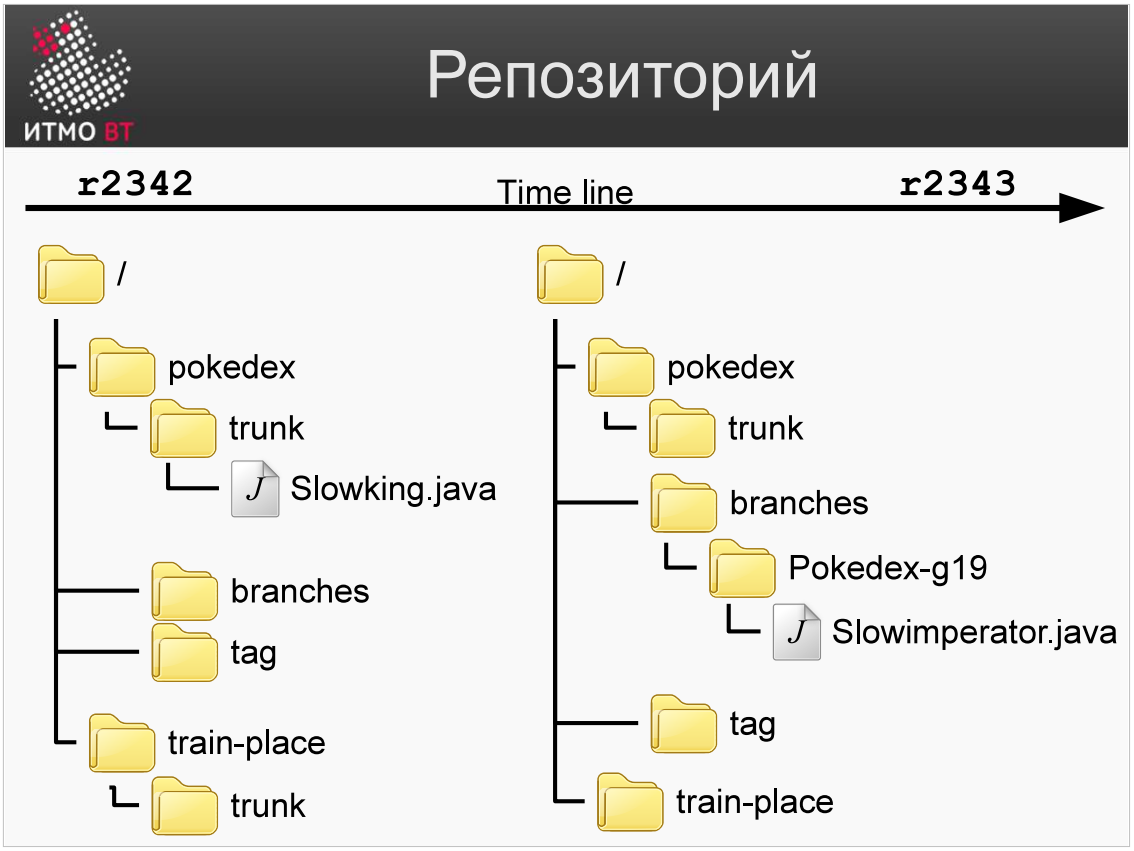
Одной из популярных систем контроля версий является Subversion (SVN).

В архитектуре, которая приведена на слайде, в существует уровень хранения репозитория (Subversion repository), который может иметь две технические реализации — размещаться в базе данных Berkeley DB, или, в простейшем случае, в файловой системе (FSFS).

Доступом к репозиторию управляет демон svnserve, который получает команды пользователя и выполняет соответствующие изменения в репозитории. Вторым вариантом доступа является использование сервера Apache и его модулей, реализующих ту же логику, что и svnserve.

Удалённый доступ к репозиторию может осуществляться по нескольким протоколам. Обычно используется протокол svn или связки svn+https и ssh+svn. Этими протоколами может пользоваться клиентская библиотека, которую, в свою очередь, используют приложения, интегрирующие в себя поддержку Subversion.

Клиент SVN не только осуществляет передачу данных с репозитория клиенту и обратно. Он ещё и осуществляет управление локальной копией файлов, которую модифицирует разработчик. Это значит, например, что когда вы скачиваете выбранную версию из репозитория, локальный набор файлов должен быть изменён в соответствии с содержимым выбранной версии.



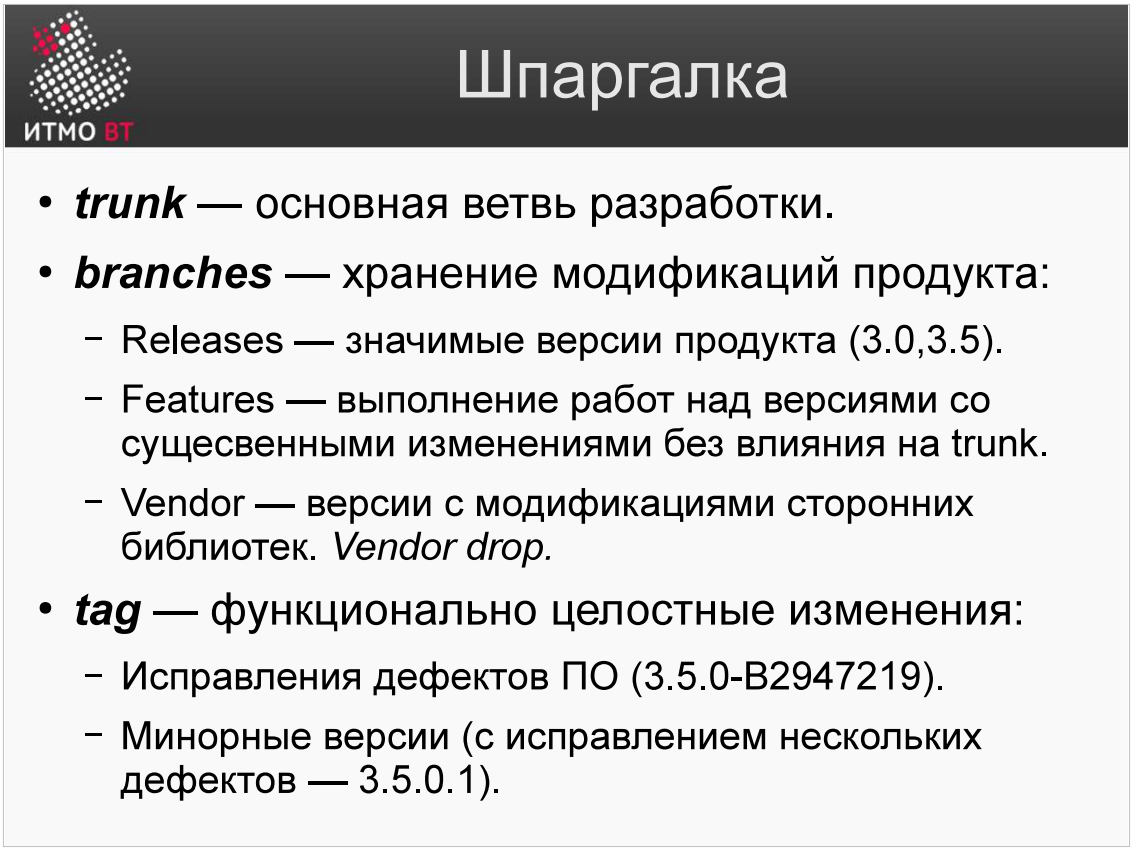
Репозиторий — это набор файлов проекта(ов), организуемый определённым иерархическим образом для удобной работы с проектом. На слайде показан пример репозитория.

В SVN каждая фиксация изменений (svn commit), пришедшая в репозиторий, повышает версию этого репозитория на 1. Ревизия репозитория — это всегда целое число, уникально характеризующее набор файлов, лежащих в репозитории на какой-то момент времени, задаваемый номером ревизии. В каждый момент времени, при условии успешной фиксации изменений, репозиторий является целостным.

Каждая фиксация изменений svn обязана включать файлы, изменения которых должны быть помещены в репозиторий. Если такое изменение производится из командной строки, то эта строка должна содержать все файлы. В одну фиксацию изменений рекомендуется помещать логически законченное обновление функционала, или, например, исправление одного дефекта. Не рекомендуется объединять разные изменения в один коммит.

Организация файлов обычно такова: существуют каталоги проектов (здесь pokedex и train-place), внутри которых размещены директории, предназначенные для организации работы с разными версиями проекта.

В каталоге trunk происходит основной процесс разработки. В этот каталог разработчики ежедневно вносят результаты своего труда, оформленные в виде целостного набора изменений (с точки зрения и набора файлов, и их логического содержимого). В случае необходимости поставки различных версий разным заказчикам, либо с целью фиксации истории релизов, стабильные на некоторый момент времени копии могут формироваться на основе содержимого каталога trunk и копироваться в дополнительные каталоги branch и tags.

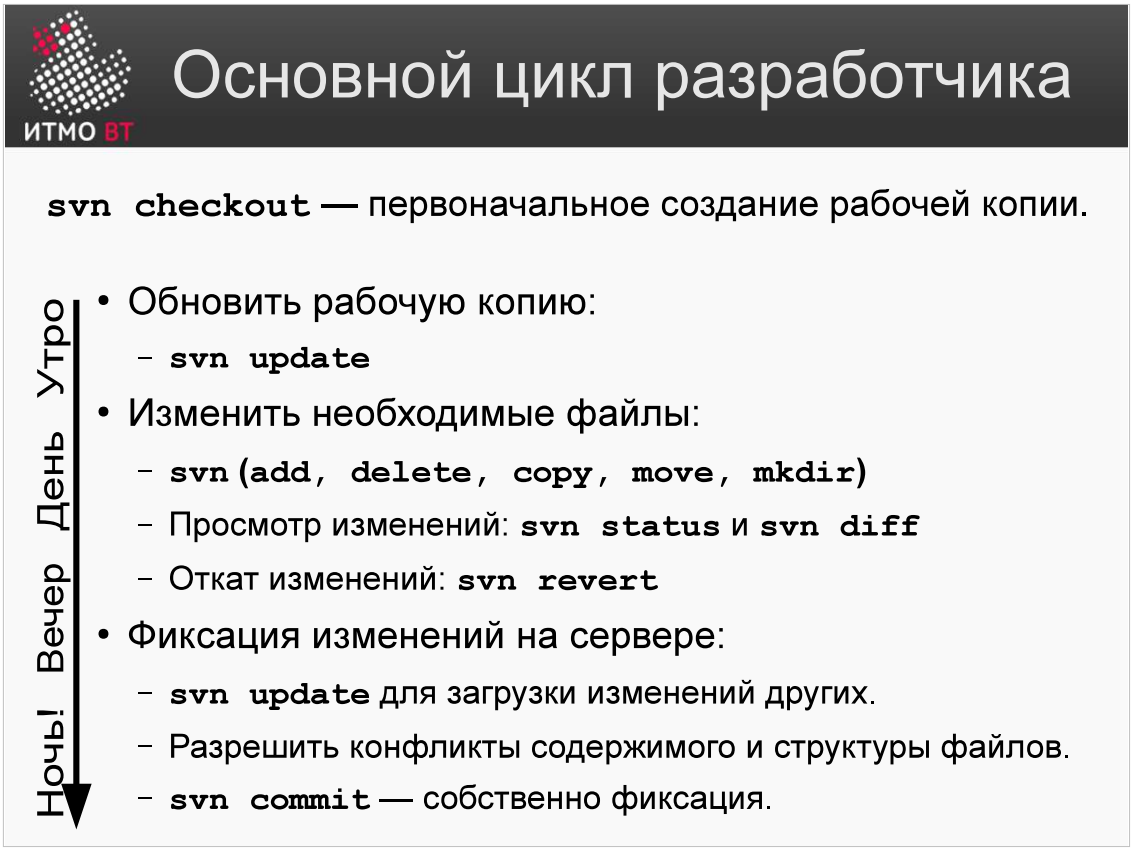


trunk — основная ветвь разработки.

В branches хранятся отдельные модификации, в частности, структурно-значимые версии продукта (например, 3.0 или 3.5). Также в branches можно вести разработку отдельных ветвей реализации, в которых внутренний функционал по сравнению с trunk серьёзно изменён, и которые не следует делать в trunk во избежание серьёзных конфликтов. После проведения всех необходимых изменений, данную функциональную реализацию сливают с основной. Существуют также Vendor branches, версии с модификациями сторонних библиотек, используемых в проекте (и, возможно, изменённых), которые необходимо держать вместе с кодом, который их использует.

В каталогах tags хранятся помеченные версии продукта. С точки зрения SVN каталоги tags используются для работы над исправлением дефектов (3.5.0-B2947219), или, например, для минорных версий с исправлением нескольких дефектов (3.5.0.1).

В SVN для создания новой ветки производится полное копирование файлов старой ветки в новый каталог. При этом точно так же всегда увеличивается версия репозитория.



Основной цикл разработки использует описанные выше свойства SVN.

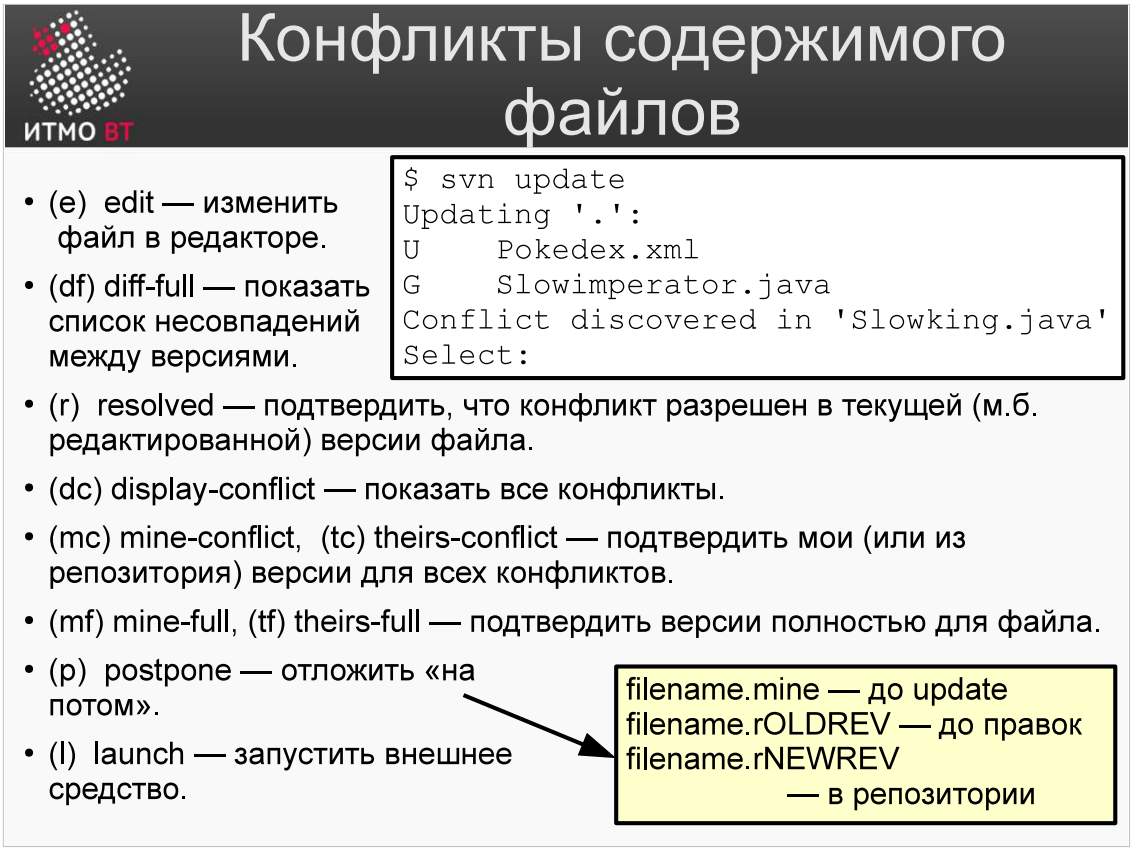
В начале работы сервер содержит в рабочей ветке все последние накопленные обновления других разработчиков, и в вашем каталоге существует локальная версия предыдущего дня. Чтобы начать ежедневный цикл работы, необходимо скачать с сервера себе в файловую систему обновление локальной копии. При этом скачиваются не только сами данные, но и метаданные. Управляющая информация нужна для обеспечения целостности и для определения того, куда будет производиться коммит. Таким образом, утром рабочего дня каждый разработчик должен выполнить svn update и привести свою рабочую копию до того состояния, которое есть на сервере.

Затем производится "великое творение кода" и меняются необходимые файлы. Используются такие команды, как svn add (для создания (добавления) файла), svn delete (для удаления файла), svn copy, и другие (то же самое касается и директорий, а также перемещения файлов внутри рабочей копии).

Если нужно провести откат изменений, используется команда svn revert, а команды svn status и svn diff помогают ориентироваться в изменениях по сравнению с сервером.

Вечером производится фиксация изменений на сервере. Так как другие разработчики тоже могли внести изменения, с сервера снова скачивается версия, там находящаяся. При этом существует вероятность возникновения конфликтов между изменениями данного разработчика и изменениями, сделанными его коллегами. Данные конфликты необходимо разрешить. После этого производится собственно фиксация с помощью команды svn commit.

При этом, тому, кто делает первый коммит, не нужно разрешать каких-либо конфликтов, так как он загружает на сервер первое изменение. Также следует отметить, что конфликты между версиями разных разработчиков могут привести и к конфликтам внутри команды. Существует также понятие ночного билда (Nightly build), который собирается именно на основе вечерних коммитов.

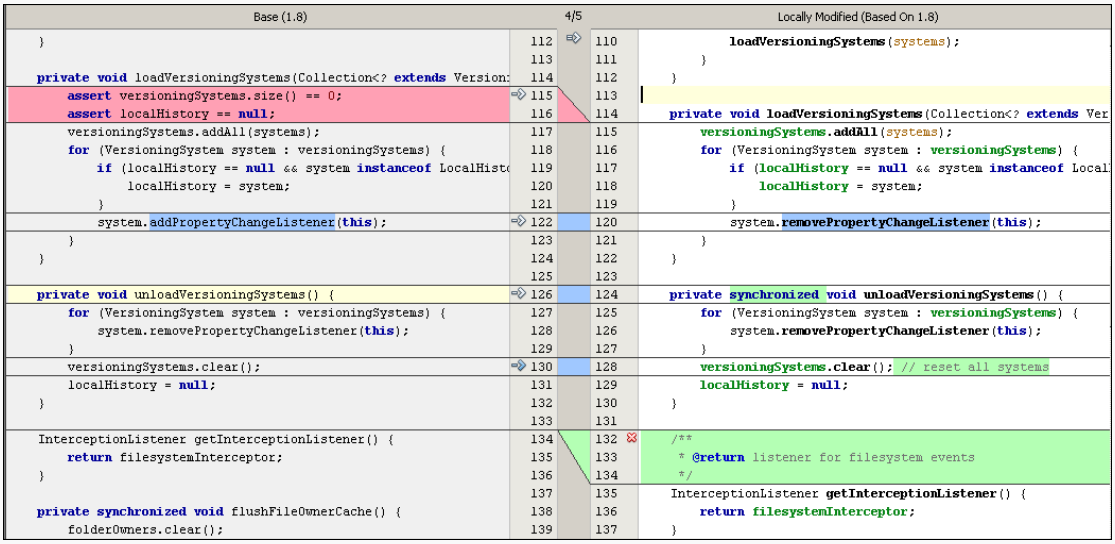


Основная проблема при фиксации изменений — конфликты содержимого файлов. На слайде приведён пример, в котором при скачивании с сервера возник конфликт между локальной версией разработчика и версией с сервера (например, по-разному поменяли одну и ту же строку).

В случае возникновения конфликта при svn update из всех возможных действий чаще всего выбирается (p) postpone — отложить на потом, так как изменения в этот момент редко возможно редактировать самостоятельно. Следует найти разработчика-автора конфликтных изменений, и согласовать с ним исправления.

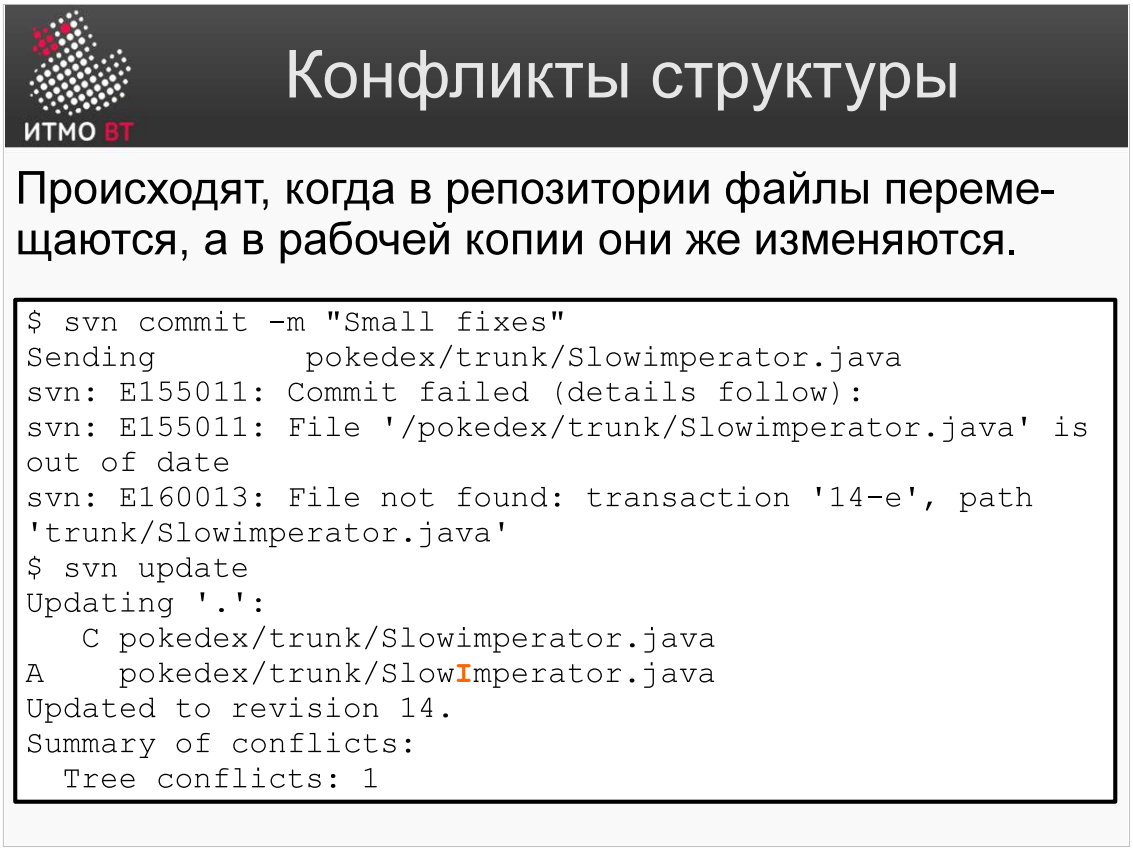
При выборе postpone также создаются три файла: filename.mine (файл разработчика до update), filename.rOLDREV (файл из репозитория до правок), и filename.rNEWREV (текущий файл в репозитории). Существуют средства (например tripIle diff), при помощи которых можно удобно сравнить одновременно все три файла.

Конфликты содержимого: Diff



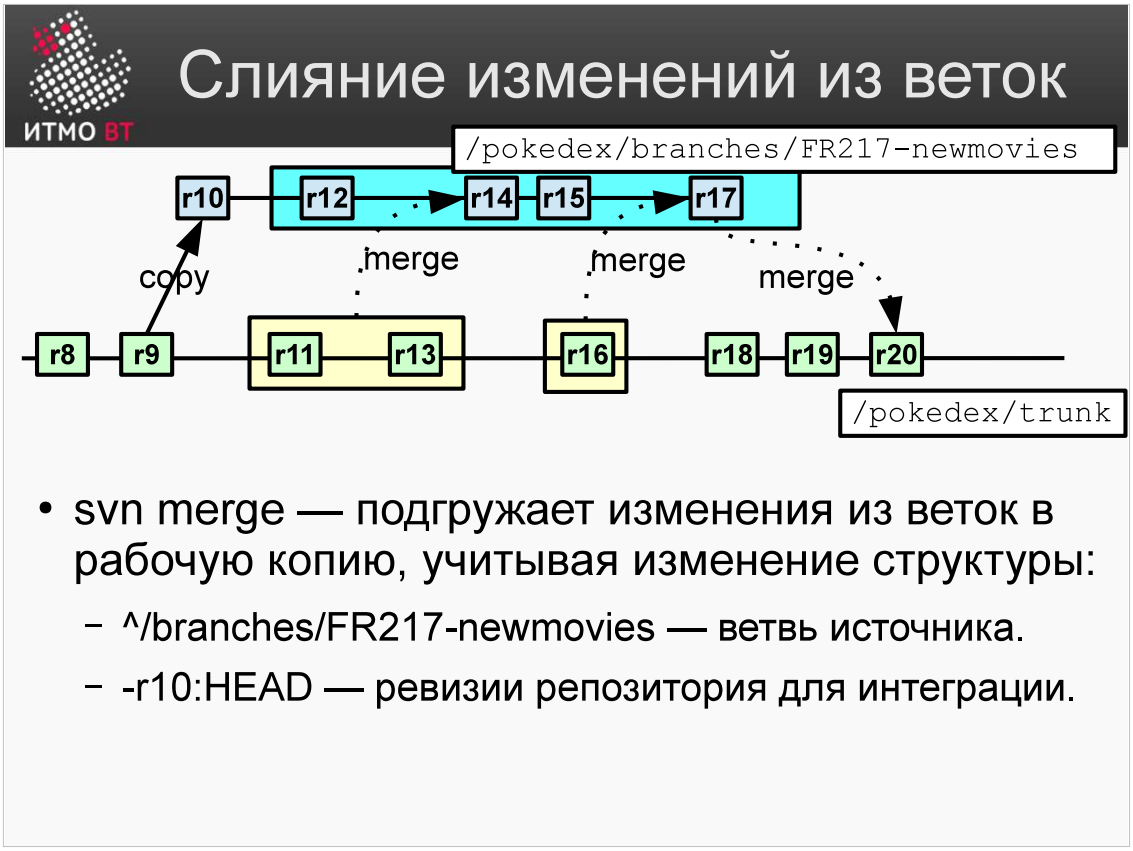
http://wiki.netbeans.org/NewAndNoteWorthyMilestone8

Обычно в средствах разработки есть собственные средства Diff для сравнения различных версий. Пример приведён на слайде. В нём различия в коде версий выделены разными цветами. Наиболее глубокого анализа требуют не добавленные или удалённые, а изменённые строки.



Конфликты такого рода происходят тогда, когда, например, в репозитории перемещаются файлы, а в локальной рабочей копии в эти же файлы вносятся изменения. Иными словами, не совпадает структура программ, и файлы, которые были просто отредактированы на локальной копии, в репозитории переименованы, перемещены, или даже удалены.

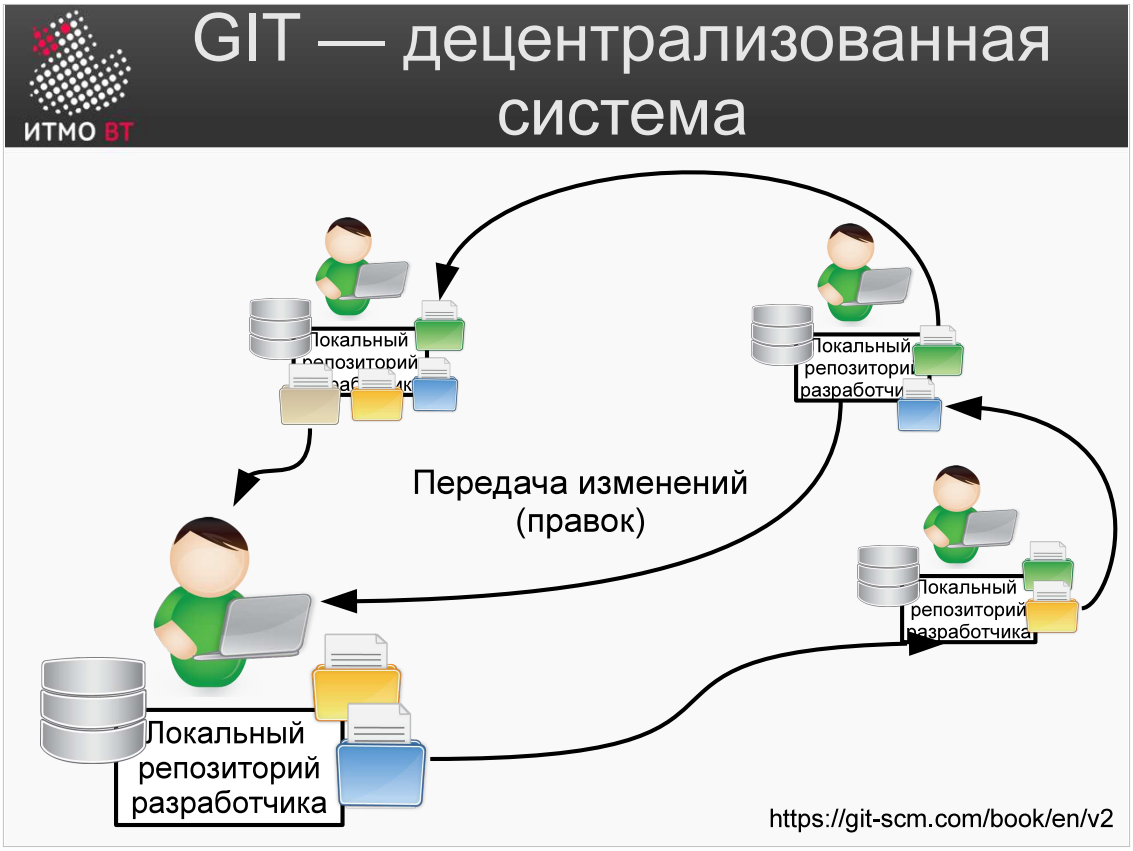
На слайде приведён пример, в котором файл в репозитории был переименован. В локальной копии его имя осталось старым. Конфликт можно разрешить как изменением структуры в локальной копии по образцу версии в репозитории, так и личным общением и согласованием исправлений с тем, кто изменил структуру в репозитории. Не всегда просто определить, в чем заключается конфликт.



При работе над параллельными ветками часто необходимо изменения из одной ветки сливать с изменениями в другой, образую одну общую ветвь. Для этого предназначена команда svn merge.

На примере представлен цикл разработки новых ходов для покемонов. Ревизия r9 была выбрана в качестве исходной, и содержимое ветви /pokedex/trunk было скопировано в ветвь /pokedex/branches/FR217-newmovies. После этого разработка производится в двух, не конфликтующих друг с другом ветвях репозитория (коммиты r11:r13, r15:r16 и r18:r19). В 14 и 17 коммитах из ветви trunk изменения добавляются в ветвь FR217-newmovies. В версии 20 изменения из 17 ветви и изменения коммитов 18 и 19 добавляются в trunk.

При операции слияния возможны конфликты, которые по своему типу не отличаются от описанных на предыдущих слайдах.



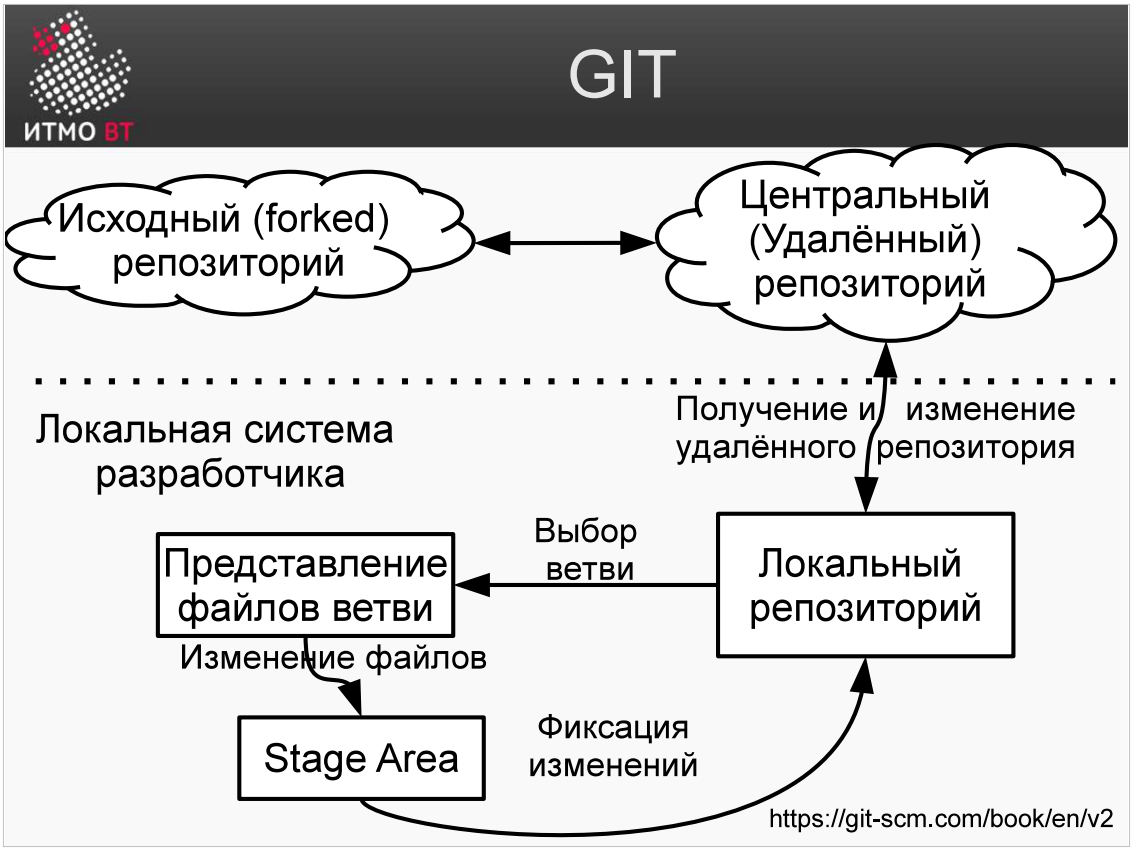
Основной философией одной из самых популярных систем контроля версий — git является ее распределенность. Репозитории существую локально у каждого разработчика, в них содержится информация о всем проекте целиком. Подчеркнем, что кодовая база, свои изменения и изменения других разработчиков, журналы применения всех доступных изменений находятся у каждого разработчика в локальной копии репозитория проекта.

Когда программист разрабатывает новый функционал (или как говорят на сленге — «фичу»), он может предложить свои правки другим разработчикам, а git предлагает удобный интерфейс по передаче этих правок и внесения их в локальные репозитории коллег-разработчиков. Таким образом может происходить, например, разработка операционных системы — пользователи сами выбирают, какие правки и фичи от других разработчиков им необходимы, чтобы работать над своим функционалом.

Каждый локальный репозиторий имеет свой URL, по которому он может быть доступен, а другие разработчики могут обращаться к нему при помощи алиасов (синонимов), что упрощает команды передачи изменений от разработчика к разработчику и от репозитория к репозиторию.

Благодаря такой архитектуре рабочий процесс совместной работы может быть адаптирован для каждой конкретной команды. В наиболее известном руководстве по Git (ссылка на которое представлена на слайде) определяются три основных рабочих процесса:

- централизованный рабочий процесс — где существует единственный централизованный репозиторий, где все разработчики синхронизируют свою работу с ним;
- рабочий процесс с менеджером по интеграции — где каждый разработчик изменяет канонический репозиторий в своей локальной копии, которую он открывает для других для чтения, а после завершения правок запрашивает интеграционного менеджера загрузить правки из своей копии для чтения в канонический репозиторий;
- рабочий процесс с диктатором и лейтенантами - в основном такой подход используется на огромных проектах, насчитывающих сотни участников; самый известный пример - ядро Linux. Лейтенанты - это интеграционные менеджеры, которые отвечают за отдельные части репозитория. У всех лейтенантов есть свой интеграционный менеджер, который называется доброжелательным диктатором. Репозиторий доброжелательного диктатора выступает как эталонный, откуда все участники процесса должны получать изменения.



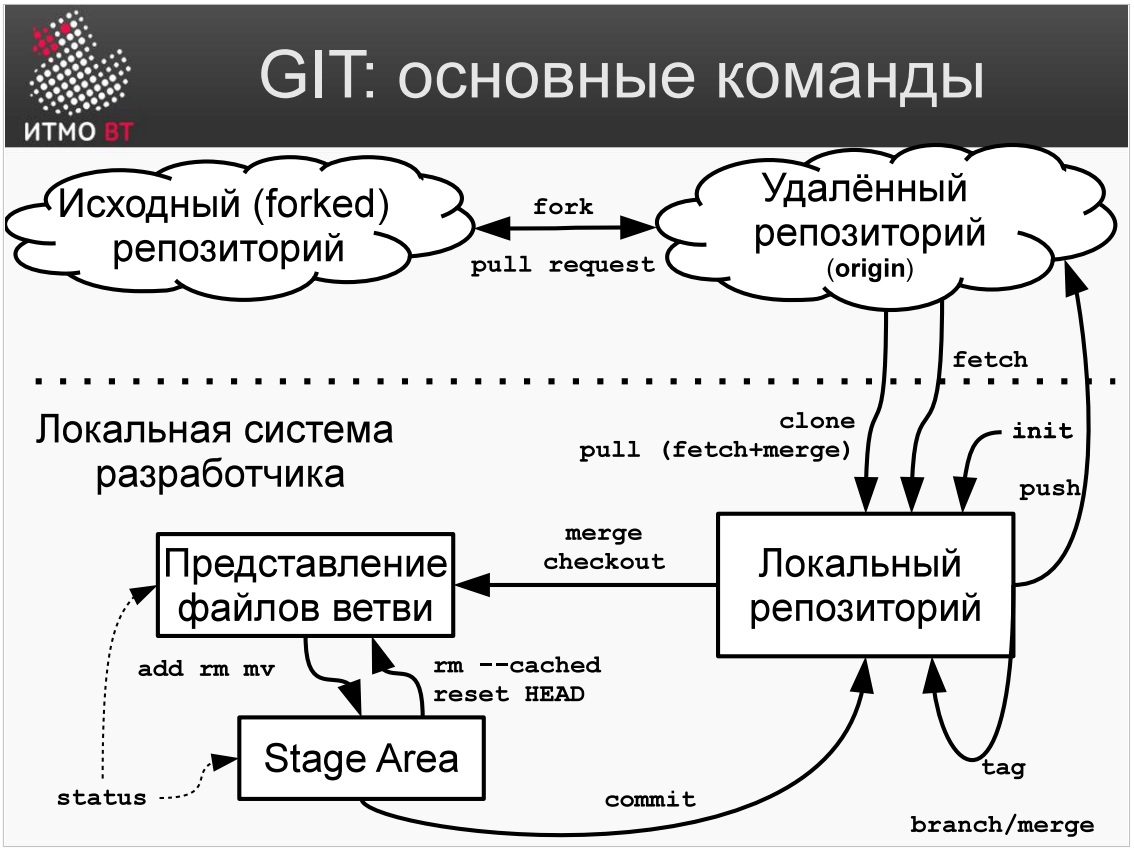
Наиболее распространенным в сообществе разработчиков систем с открытым кодом, использующим популярный ресурс GitHub, является рабочий процесс с интеграционным менеджером. В нем существует центральный удалённый репозиторий, в котором сохраняются результаты работы всей команды разработчиков. Центральный репозиторий может быть получен при помощи организации ветви (fork) от любого другого известного проекта.

Такому центральному репозиторию по умолчанию git назначает алиас origin. Центральный репозиторий для каждого из разработчиков команды является удалённым.

У каждого разработчика есть свой локальный репозиторий, который является клоном (копией) удалённого репозитория. Разработчик выбирает необходимую ветвь в локальном репозитории, получая представление на файловой системе файлов с содержимым ветви, которые он может изменять.

Во время работы с рабочей копией разработчик указывает, какие файлы включить в фиксацию изменений. Эти изменения помещаются в т.н. Stage Area — специальную область, из которой файлы будут включены в коммит. В отличии от svn, здесь нет необходимости указывать все файлы, участвующие в фиксации изменения в одной команде — в git их можно добавить за несколько операций. Затем изменения фиксируются в локальном репозитории. Для помещения информации в центральный репозиторий разработчик использует команду push, при которой, естественно, могут возникать конфликты.

Другой способ фиксации изменений в удаленном репозитории — это конфигурация запросов на фиксацию изменений (pull requests). При этом изменения, которые разработчику необходимо записать, сначала должны быть подтверждены владельцем ветви. Таким образом обычно происходит слияние функциональности в открытом программном коде, управляемым сообществом разработчиков. На слайде это показано двусторонней связью "Удалённого репозитория" и "Исходного репозитория".



На данном слайде приведены основные команды git. Каждая команда создает изменения относительно предыдущего изменения и подсчитывает хеш изменений. Именно поэтому команды git очень быстро выполняются и эффективно обрабатываются системой.

Приведём примеры команд: с помощью команды clone можно скопировать удалённый репозиторий в локальный репозиторий, командой init можно создать локальный репозиторий.

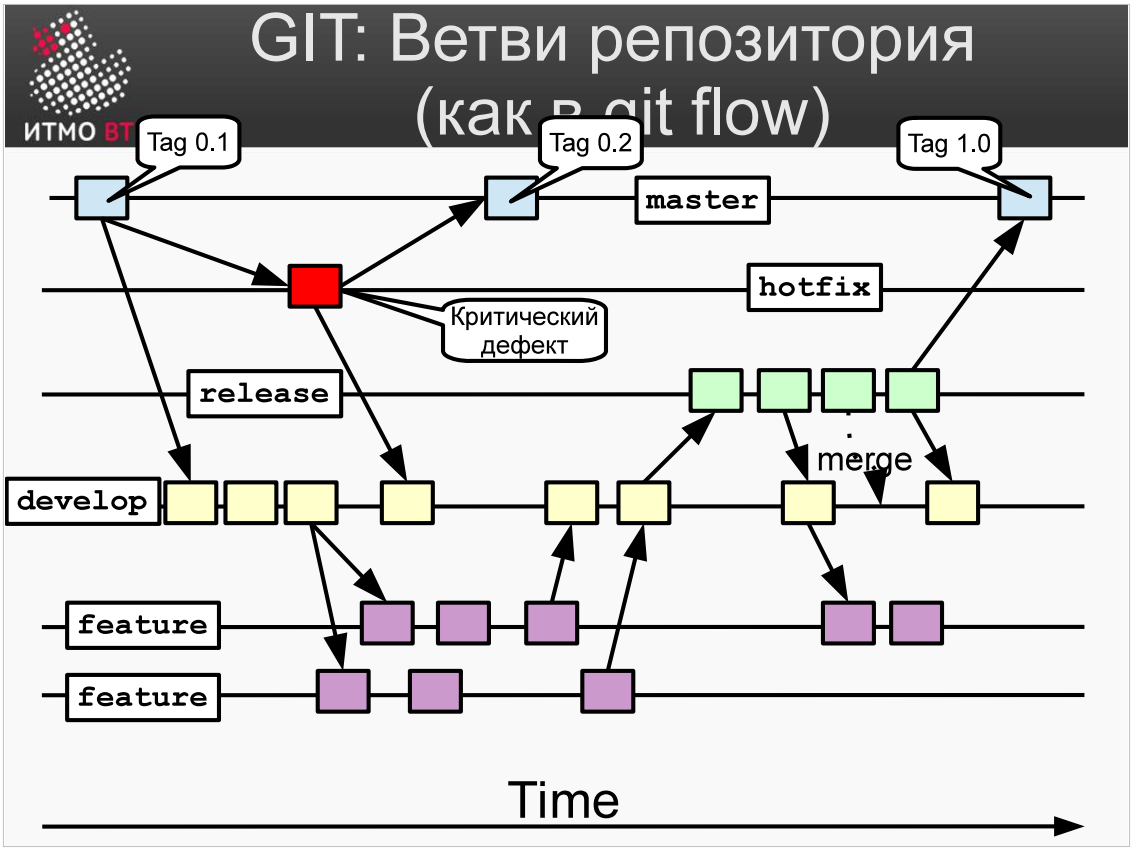
Разработчик может создать новую ветвь при помощи branch, при этом в качестве родительской ветки будет использована ветвь, с которой в настоящий момент происходит работа. Для того, чтобы установить рабочую копию в согласованное с необходимой ветвью состояние (т. е., перейти в эту ветвь), используется команда checkout. Создание ветви не подразумевает переход в нее.

После завершения работы с рабочей копией, разработчик с помощью команды add указывает, какие файлы должны попасть в Stage Area, и с помощью команды commit изменения фиксируются в локальном репозитории. С помощью команды push результаты можно отправить в удалённый репозиторий.

Изменения с удалённого репозитория скачиваются с помощью команды fetch, но рабочая копия при этом автоматически не меняется. Для её обновления нужно выполнить команду merge с желаемой ветвью (использование команды pull равносильно использованию обеих команд вместе). Не рекомендуется использовать команду merge при наличии изменений в рабочей копии.

Другим вариантом включения изменений из удалённого репозитория в рабочую копию может быть использование команд fetch+rebase, в таком случае ваши изменения рабочей копии будут логически применены к другому базовому коммиту.

Команда fork используется для создания ветвления существующего проекта в своем удалённом репозитории. Запрос на применение предлагаемых разработчиком изменений на исходный репозиторий формируется с помощью команды pull request.



Модель разработки в git основна на концепции, что каждое различимое изменение должно разрабатываться в отдельной ветви. Схематично модель ветвлений представлена на слайде.

В ветви Master находятся версии, готовые к поставке пользователю. Эти версии являются основными версиями продукта. Функционал там чётко определен, и список дефектов известен.

Основная разработка происходит в ветви Develop. Данные ветви используют Master как базовую, и в них происходит разработка новой версии продукта. После кодирования все разработчики сливают свои изменения именно в ветвь Develop. При этом изменения рекомендуется делать по функциональным требования продукта. Для каждого из требований создается своя ветвь Feature, и там аккумулируются коммиты для этого требования. По мере завершения разработок функционала коммиты из Feature-ветвей подгружаются в ветвь Develop.

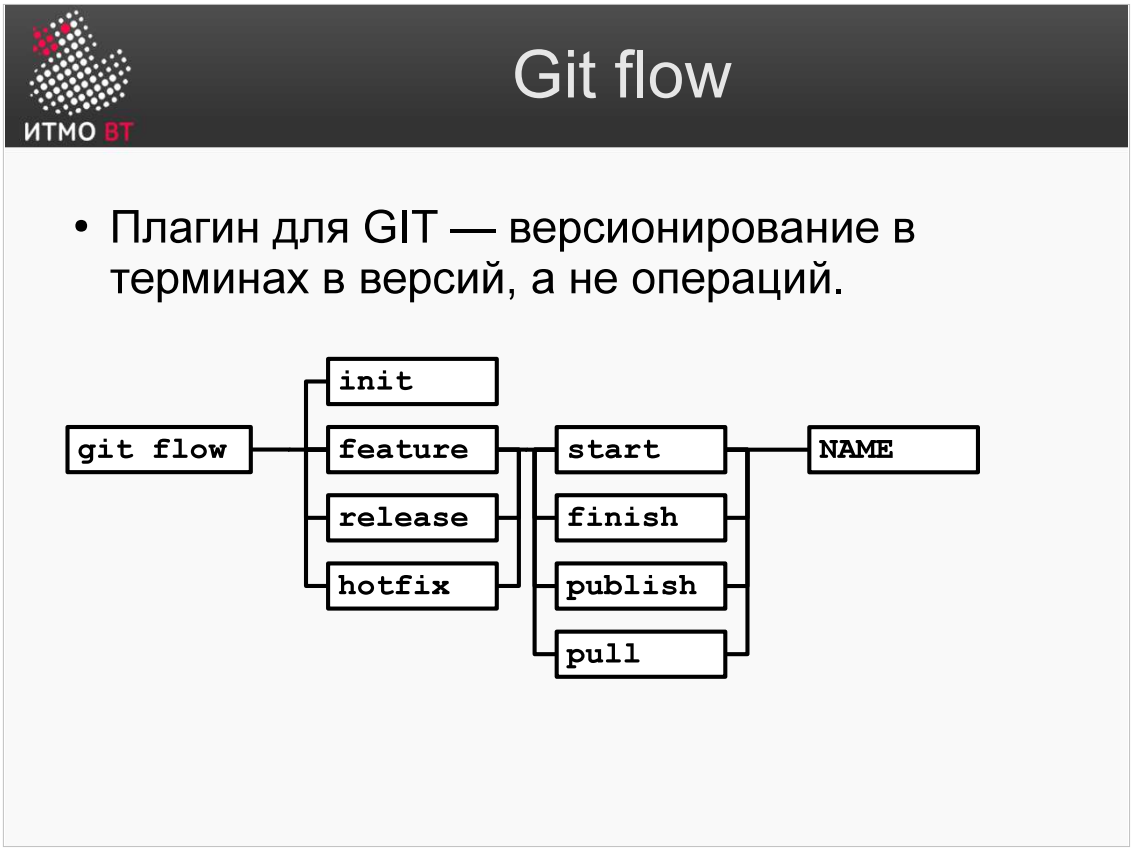
Когда накоплено достаточное количество функционала для выпуска новой, готовой к поставке версии, изменения из Develop формируют ветвь Release, а сама ветвь Develop остаётся для дальнейшей разработки новых функций. Ветвь Release используется для исправления тех дефектов, которые препятствуют выходу новой версии. После исправлений готовая версия продукта копируется в ветвь Master. При этом исправления дефектов сливаются в ветвь Develop, так как они там также будут необходимы в дальнейшем.

Ветвь Hotfix предназначена для исправления критических ошибок (т.е. тех, которые препятствуют работе) в версии пользователя. После исправления результат передаётся как в ветвь Master, так и в ветвь Develop.

Процесс повторяется для всех новых функциональных требований.

Команды, которые могут понадобится при выполнении лабораторной работы представлены на слайде. В лабораторной работе функциональность команд сильно урезана, например git branch может только показать ветви.

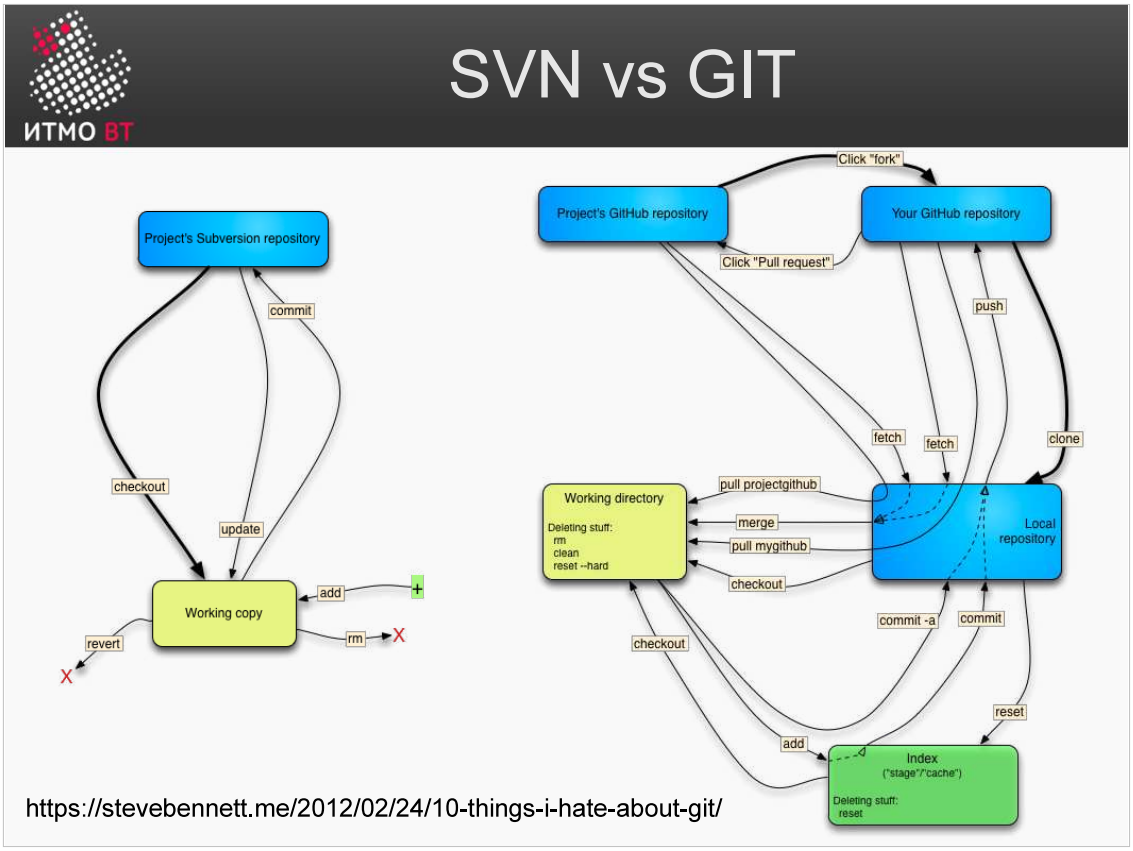
- git status — показывает статус текущих состояний файлов в файловой системе и информацию о ветви, в которой производится редактирование.
- git log показывает журнал коммитов, а опция --graph выводит в графическом виде ветви, в которых производились данные изменение.
- git diff — покажет все изменения, которые были сделаны относительно последних зафиксированных изменений.
- git reset --hard HEAD сбросит все изменения, которые были сделаны в текущем локальном репозитории.
- git branch — показывает ветви.
- git checkout — переключает разработчика между ветвями
- git merge — объединяет несколько ветвей в текущую ветвь
- git commit — фиксирует изменения в текущей ветке. Опция -m задает сообщение коммита, которое будет показываться пользователю
- git add — добавляет измененные файлы к последующему коммиту, помещая их в Stage Area.



Каждое проделываемое в Git действие над рабочей копией (создание ветвей, их слияние и т.п.) требует достаточно большого количества команд, и может вызвать у разработчиков сложности с синтаксисом команд. Для того, чтобы упростить эту работу, был создан плагин Git flow, позволяющий работать с версиями именно в терминах версий, а не операций, без точного знания команд и последовательностей действий.

Для использования Git flow необходима определённая начальная настройка репозитория. Для этого у Git flow есть функция init, которая при старте задаёт пользователю несколько вопросов и производит необходимую настройку.

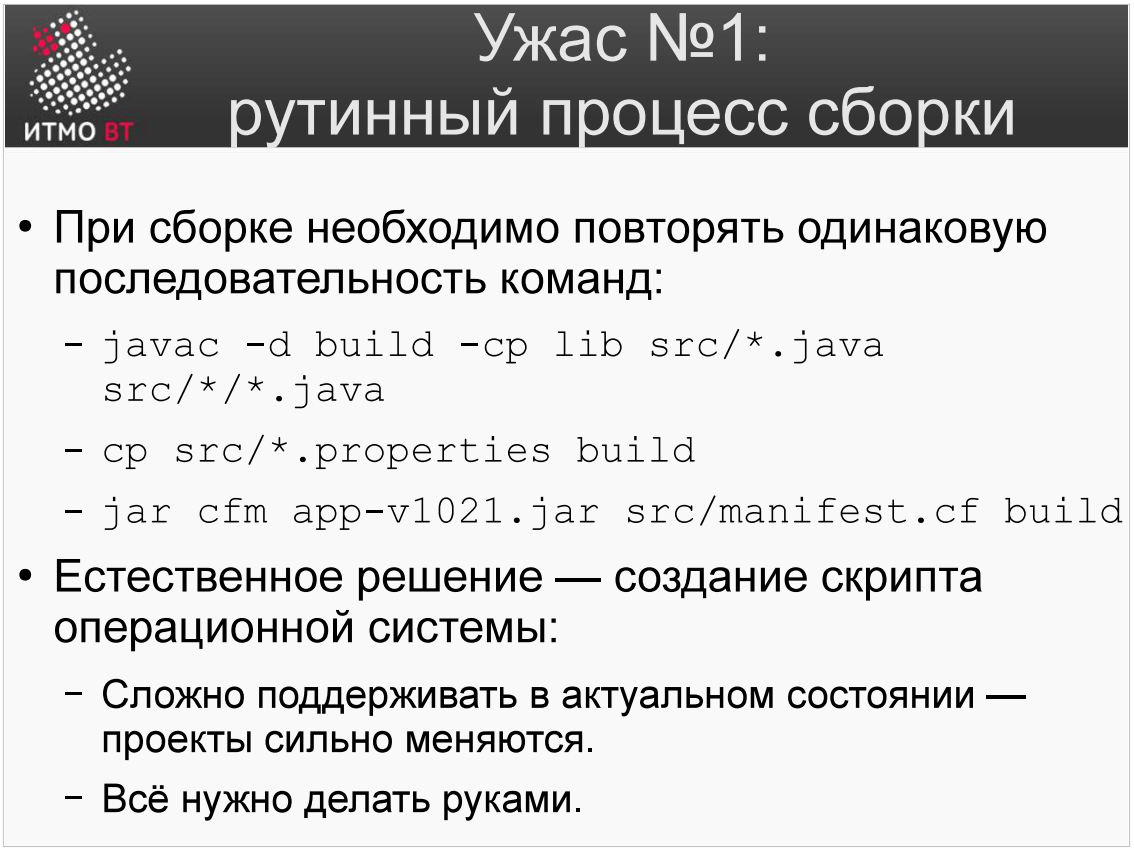
Для создания новой feature в git flow используется команда git flow feature start FEATURE_NAME. Когда надо указать на окончание изменений, используется git flow feature finish FEATURE_NAME. Остальные команды формируются аналогичным образом. Все это позволяет пользоваться репозиторием на более высоком, независимом от синтаксиса git уровне, при этом реализуется модель, указанная на предыдущих слайдах.



Для завершения обсуждения сравнительных характеристик SVN и GIT на слайде представлены два графа работы с обеими системами контроля версий. Как видно из диаграмм, GIT использует гораздо больше команд, чем SVN, и требует в повседневной работе большего количества действий и знаний от разработчика.

4

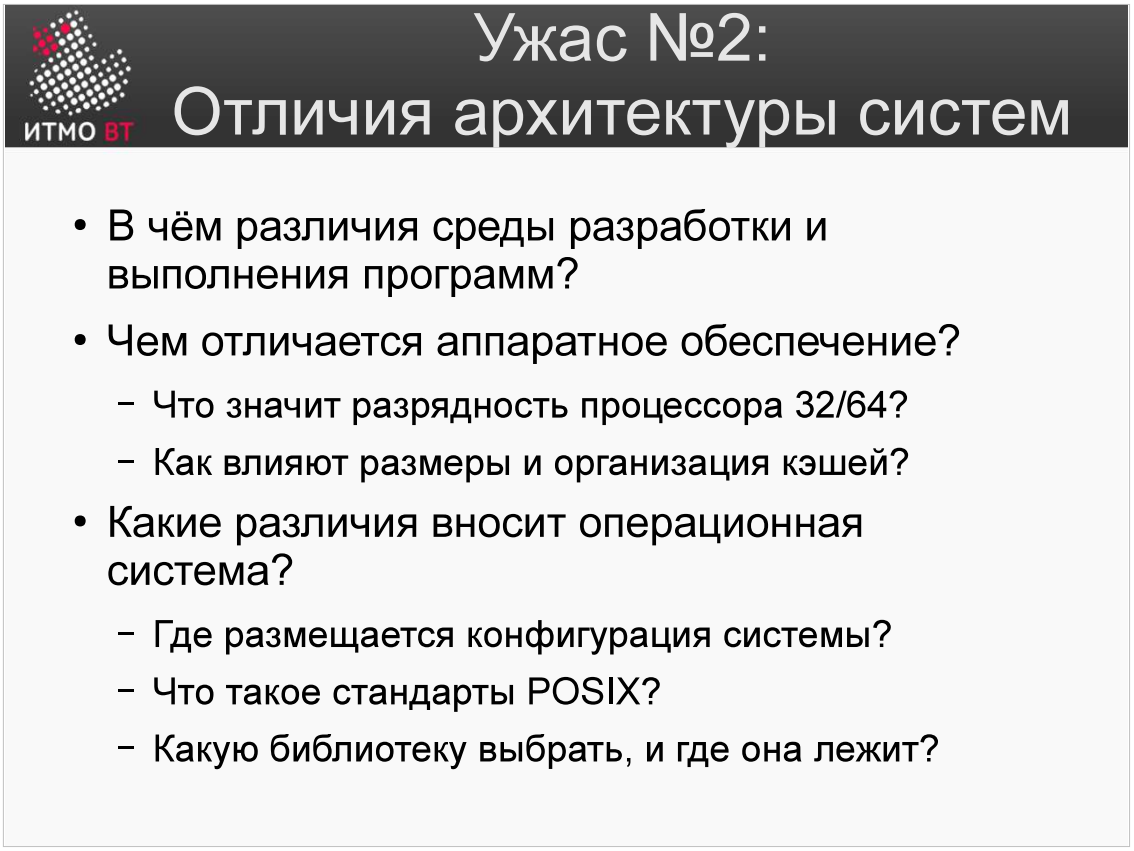
Конфигурация и сборка программных продуктов



Для чего необходимо автоматизировать процесс сборки приложений? Можно найти много причин для автоматизации, но мы выделим несколько основных.

При сборке программного продукта даже в простейшем случае приходится многократно повторять последовательно несколько команд — таких, например, как представленные на слайде. Если приложение состоит из модулей и имеет внешние зависимости, то такая последовательность становится сложной и зависимой от различных условий.

Очевидным решением данной проблемы стало создание и использование скриптов на уровне операционной системы. Их сложно поддерживать, так как любое изменение в структуре программы (например, удаление или переименование файла) нужно вручную отражать в скрипте. Современные средства разработки облегчают эту задачу, генерируя последовательность команд (или сами скрипты) автоматически. Нужно отметить, что старые средства разработки, используя описания файлов, также выступали в роли и средств сборки.



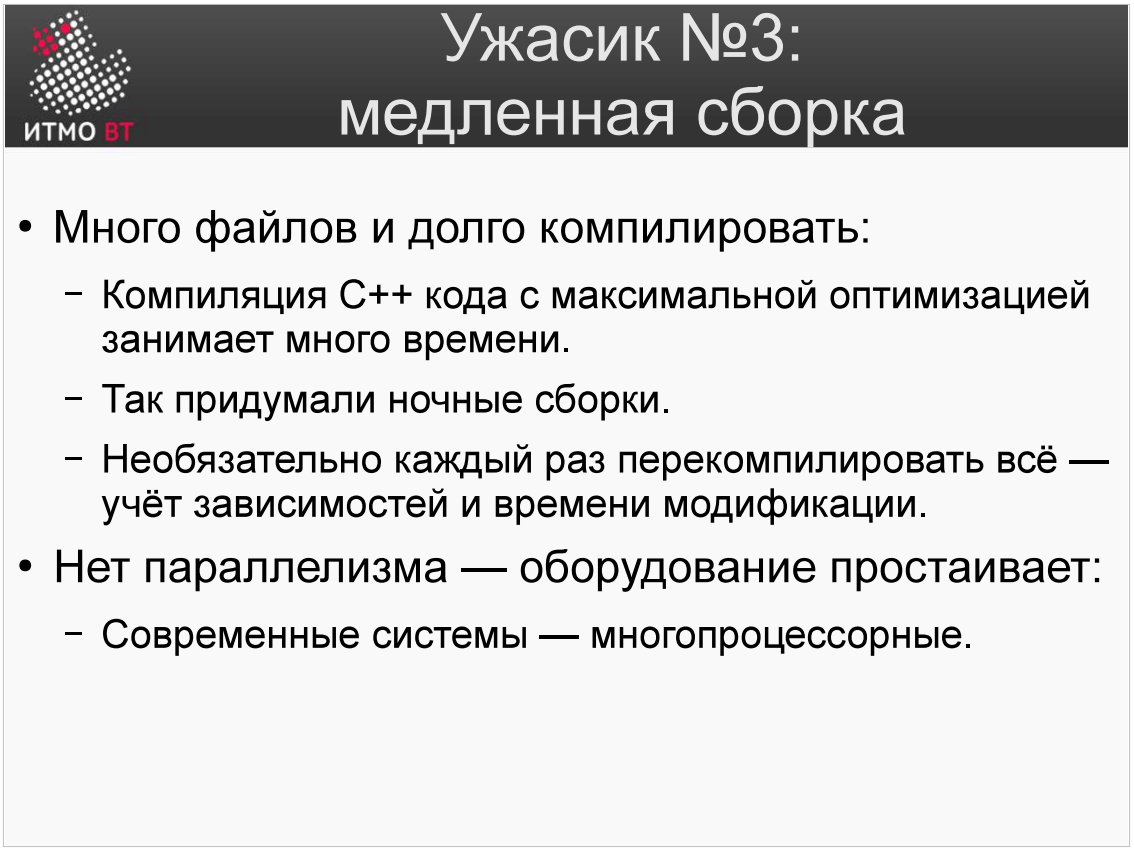
Среда разработки почти всегда отличается от той среды, где будет использоваться продукт. Эти различия могут быть качественными и накладывать значительные ограничения на проект.

Первая категория отличий связана с различиями в аппаратном обеспечении. Например, серьёзные отличия существуют в системах с разной разрядностью, что влечет за собой различия в размере минимальной единицы информации, с которой можно производить арифметические и иные действия, что влияет не только на быстродействие программы но и на необходимость генерирования другого кода для целевой системы. Отличия в разрядности системы обычно обуславливают различия в разрядности адреса, что влияет на возможности работы с памятью, что, в свою очередь, влияет на быстродействие некоторых алгоритмов (например, алгоритмов сортировки).

Большое значение имеют размеры и способ организации кэшей. При работе с переменными лучше, чтобы они находились рядом, при попадании данных в кэш резко повышается производительность, это необходимо учитывать при сборке приложения для целевой системы.

Вторая категория отличий объединяет различия, вносимые операционной системой. Стандартные библиотеки ОС могут находится в разных местах целевой системы и иметь версии, отличные от системы разработчика. Стоит отметить, что хорошо известные стандарты POSIX выросли из попытки создать библиотеки ОС, универсальные с точки зрения API для системных функций различных операционных систем.

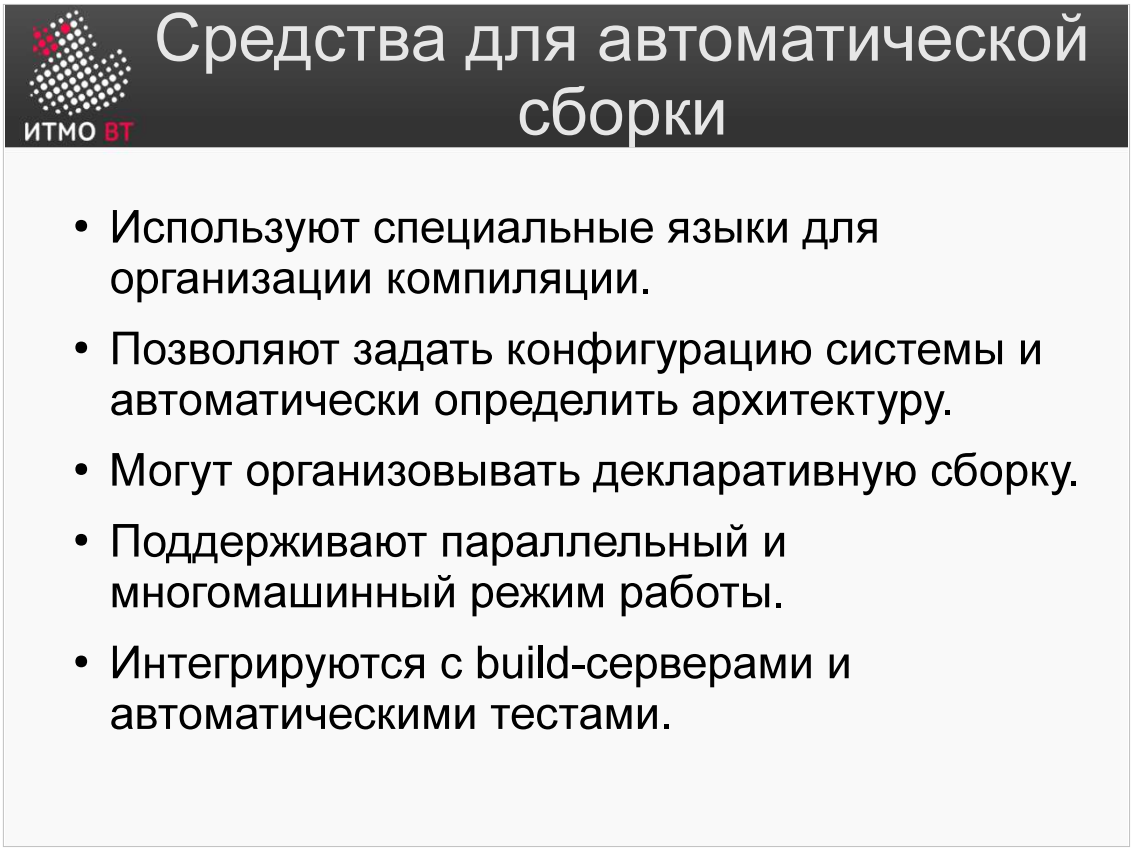
Кроме того, нужно учитывать, где размещается конфигурация системы, которая используется системными службами. Например, локальный файл соответствия IP-адресов именам машин в Ubuntu Linux находится в /etc/hosts, в Solaris в /etc/inet/hosts, а в Windows в C:\Windows\System32\drivers\etc\hosts. Эту конфигурацию необходимо учесть для обеспечения платформонезависимости ПО.



Сборка проекта может быть достаточно продолжительным процессом. В частности, много времени может уйти на оптимизацию кода компилятором. Для ускорения процесса его можно частично распараллелить. К сожалению, невозможно, например, одновременно компилировать файлы с зависимостями друг от друга.

Помимо процесса параллельной компиляции существуют также системы многомашинной сборки. Здесь сложность возникает с организацией доступа сборочных узлов к общим исходным кодам, использованием одинаковой версии компиляторов и других утилит сборки, размещением результатов построения программы в одном, общедоступном для всех узлов репозитории и т.д.

Длительность процесса сборки повлияла на расписание полной сборки "с нуля" больших программных продуктов. Логично проводить такую сборку во время, когда разработчики не сохраняют в системе контроля версий новые изменения, что и привело к созданию "ночных сборок".

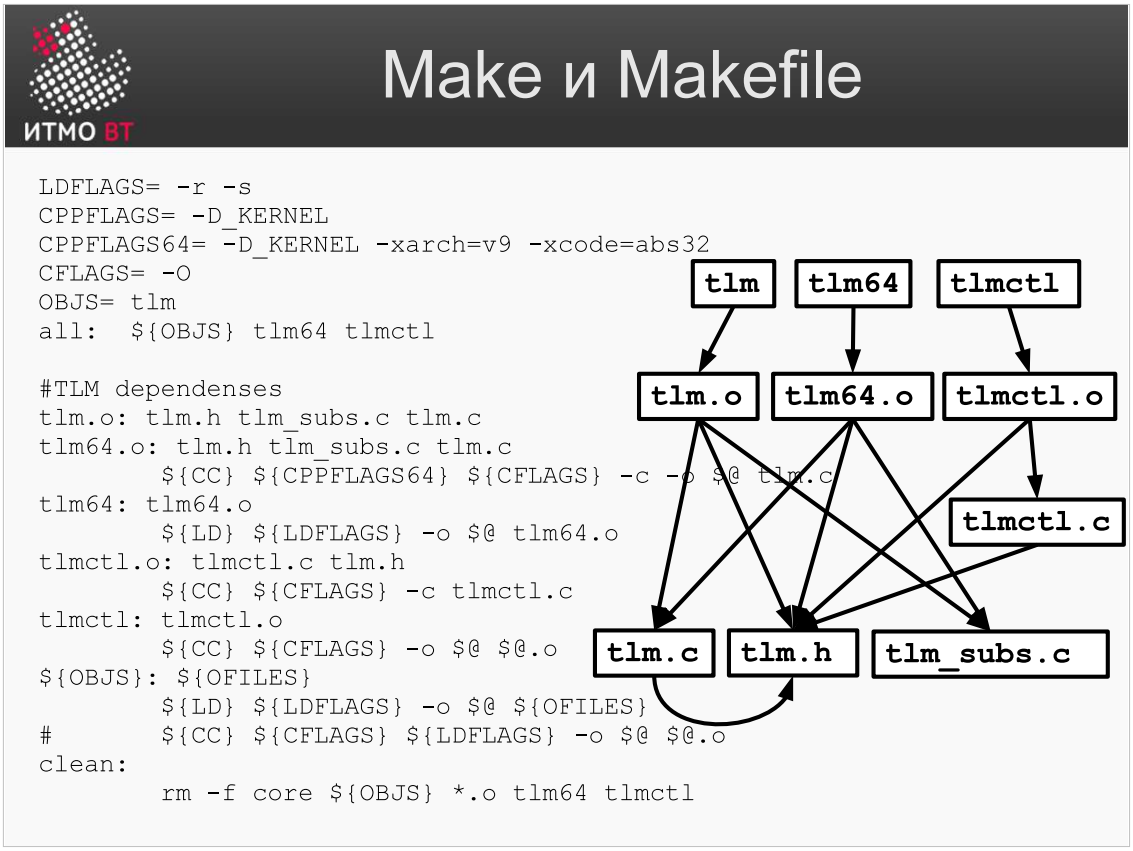


Современные системы сборки обеспечивают выполнение процесса сборки под управлением специально разработанных языков, уменьшающих трудоемкость поддержки процесса разработчиком. В идеале, скрипты сборки должны быть сгенерированы автоматизированно, с минимальными трудозатратами разработчика.

Помимо самой сборки, системы могут задавать и автоматически определять конфигурацию и архитектуру сборочной и целевой системы, определять возможности максимально эффективной утилизации ресурсов системы, на которой проводится сборка, организовывать многомашинный режим сборки.

Для возможности непрерывной интеграции разработаны специальные сборочные серверы, которые при внесении изменений в репозиторий автоматически запускают сборку и проведение тестов системы, а также хранят все сборки разработанного программного обеспечения, позволяя быстро поставить пользователям необходимую им версию с заданным набором функциональных возможностей под все поддерживаемые платформы и операционные системы. К таким сборочным серверам, например, относятся хорошо известные системы GitLab CI и Jenkins.

Все вышеперечисленные возможности позволяют сделать процесс сборки более простым, понятным и контролируемым.



Первым инструментом сборки были упомянутые ранее разработанные вручную сборочные скрипты. Со временем выяснилось, что такие скрипты поддерживать трудно, и необходимы некие промежуточные языки, на которых можно было бы запрограммировать процесс сборки.

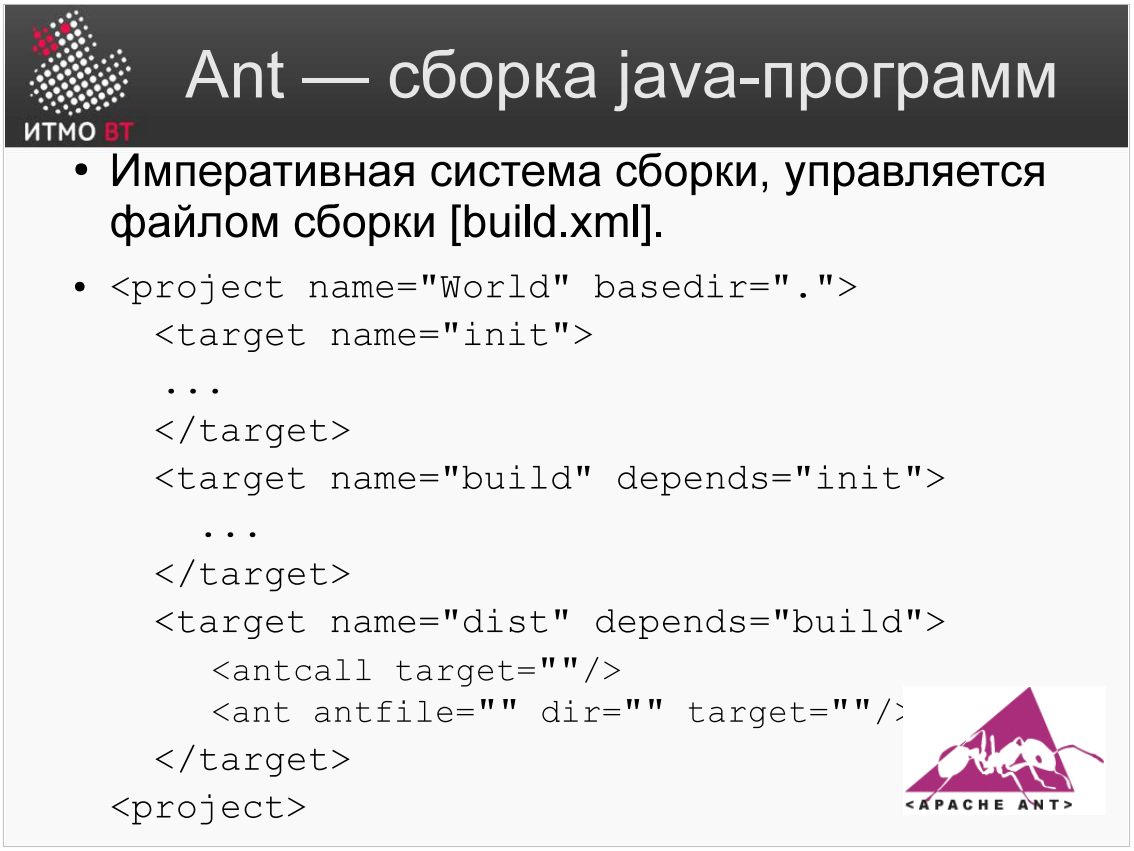
На слайде приведен пример файла, задающего конфигурацию сборки проекта при помощи утилиты make. Make впервые появилась в Unix-экосистеме и существует по сей день как средство сборки низкого уровня. В файле сборки (Makefile) указаны опции компиляции и зависимости, необходимые для работы make.

На слайде показан модуль, который может быть скомпилирован в 32-разрядном и в 64-разрядном режимах, чтобы быть загруженным в ядро ОС. Слева указано содержимое makefile, а справа в виде графа описаны зависимости между файлами. ПО данного примера состоит из трех исходных файлов с расширением .c и один заголовочный (.h). На базе этих файлов строятся три объектных модуля, которые приводят к созданию одного исполняемого файла и двух загружаемых.

Внимательно изучив пример, можно заметить, что в makefile прописаны цели, которые нужно собрать, после них через двоеточие указаны файлы, от которых зависят данные цели, а на следующей строке — последовательность действий, которые нужно выполнить для сборки каждой, формируя показанное дерево зависимостей.

При запуске make, утилита анализирует файл, начиная с цели "all" (стандартная цель по умолчанию), после чего собирает программы с учетом зависимостей. Также учитывается время последней модификации файлов. Если время последней модификации файла позже, чем время модификации любой его зависимости, то данную цель в makefile необходимо пересобрать.

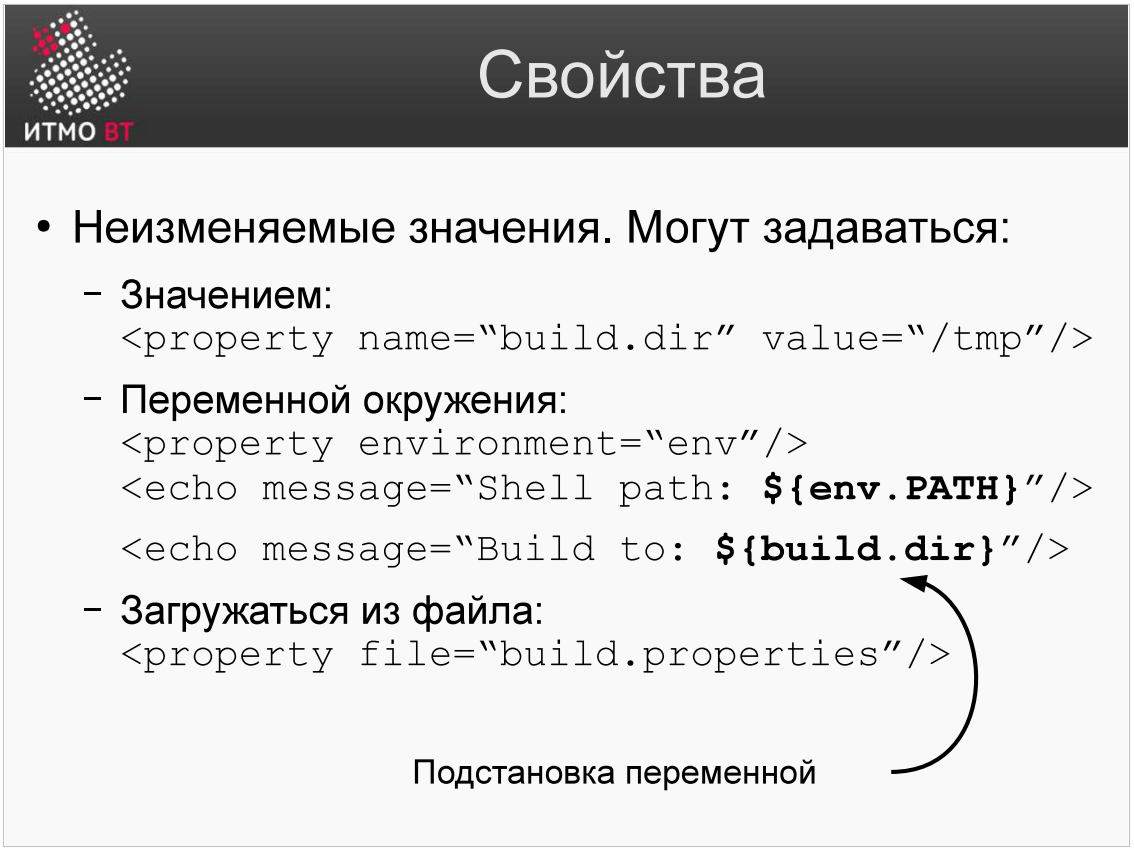
Таким образом, Make — это язык, который в неявном виде позволяет определить последовательность действий при сборке, и шаги, которые необходимо предпринять при этой сборке. К сожалению, Make нельзя назвать дружественным для разработчика, в нём необходимо следить за большим количеством макропеременных и действий по умолчанию, определённых в самой системе.



Для сборки java-программ широко используется утилита Apache Ant. По принципу действия она ничем не отличается от Make и относится к т.н. императивным системам сборки. По аналогии с makefile, Ant управляется файлом сборки [build.xml], в котором в формате XML описан проект.

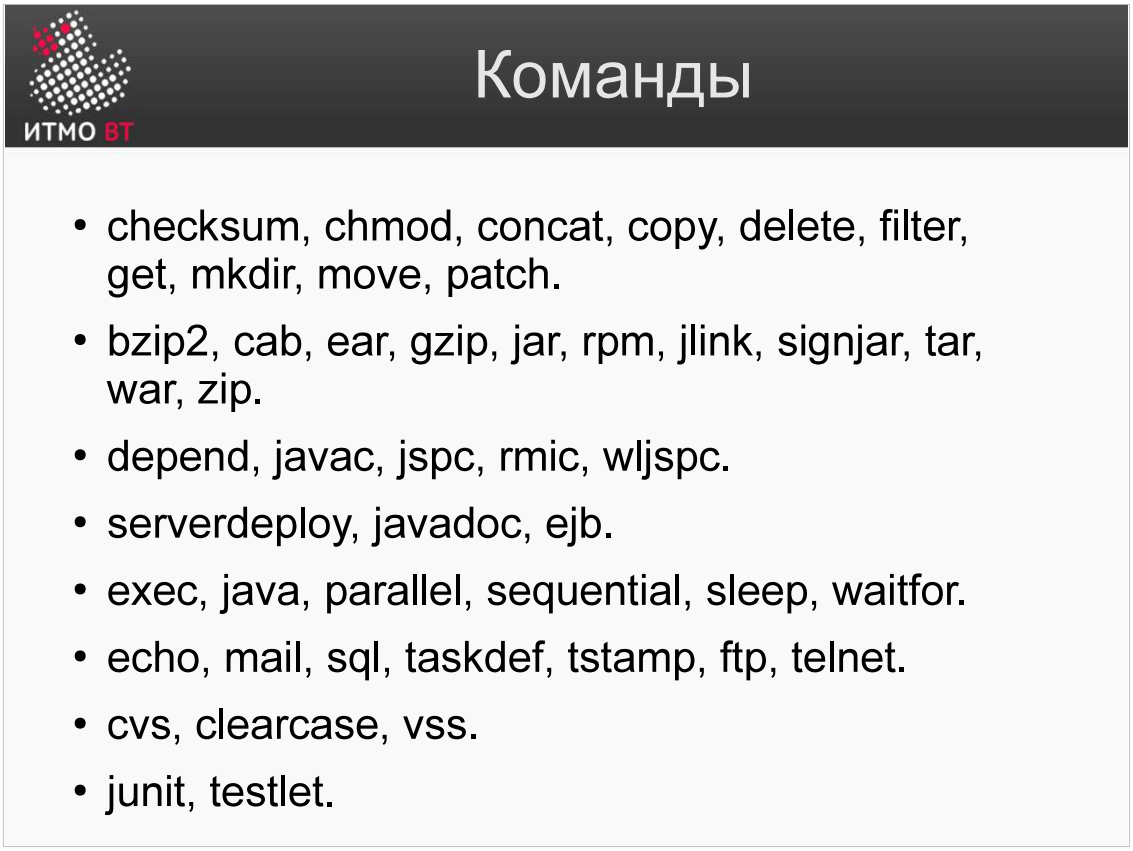
По мнению авторов, XML не является очень удобным для чтения человеком, хотя прекрасно приспособлен для машинной обработки. Как видно из примера, описание проекта состоит из именованных целей (target) с явным указанием зависимостей — других целей, что позволяет создать из зависимостей дерево. Кроме того, внутри каждой цели определены действия, которые необходимо выполнить в процессе сборки.

Цели можно вызывать (antcall) в явном виде (не через зависимости). Возможен вызов целей из другого файла, что позволяет собирать систему из модулей.



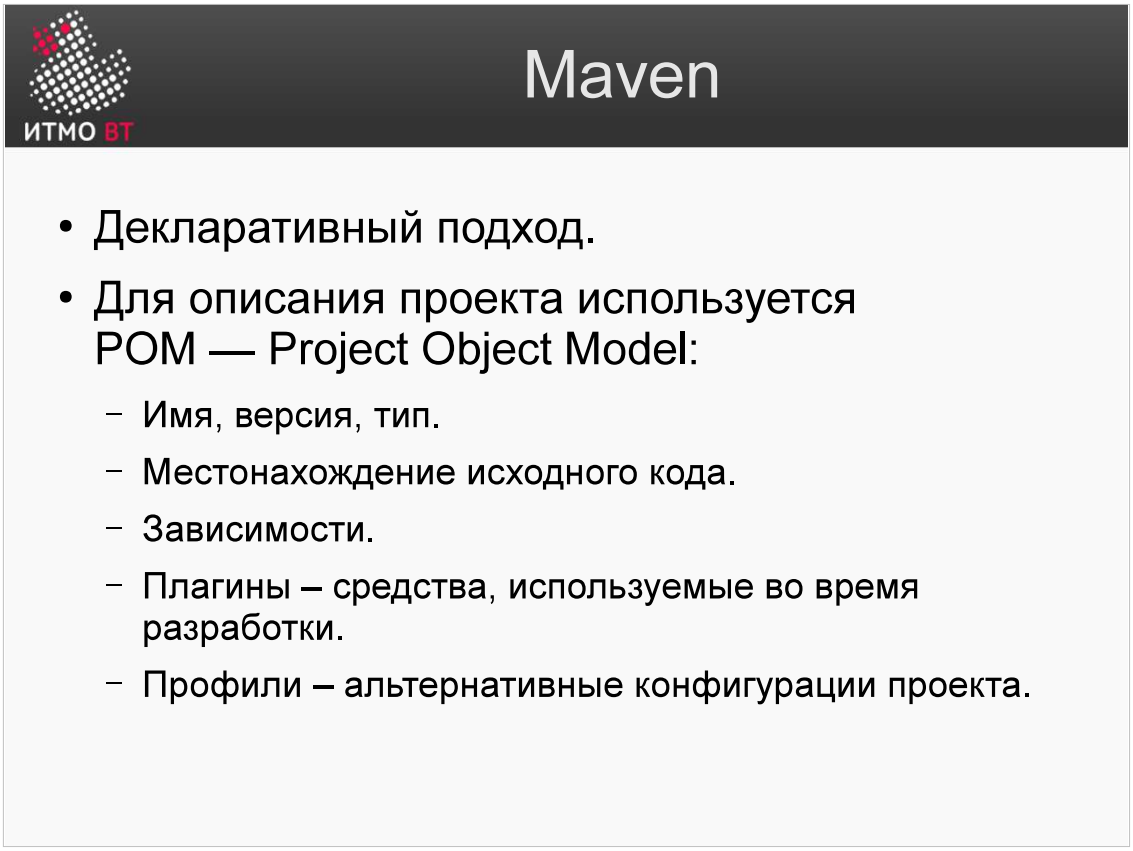
Как в makefile, так и в Ant есть переменные (свойства) с информацией о проекте, которые необходимо изменять для того, чтобы было возможно собирать проект в разных условиях.

Свойства в Ant могут задаваться прямым значением, ссылкой на перемененную окружения, или загружаться из файла (где их проще всего редактировать). После указания значения переменной, на данную переменную можно ссылаться с использованием синтаксиса, пример которого приведён на слайде.



В Ant существует большое число команд на все необходимые случаи жизни при сборке Java-приложения. То, что в платформозависимых средствах, таких как Make, выполняется при помощи команд ОС, в Ant выполняется с помощью собственных команд, задаваемых в виде XML. Это позволяет сделать сборку независимой от особенностей поведения команд в той или иной операционной системе, однако каждый раз необходимо уточнять синтаксис каждой команды.

На слайде представлены команды по манипуляции над файлами, созданию целевых архивов, размещения продуктов на серверах приложений, проведения тестов, работой с системами контроля версий, генерации документации и для осуществления других действий, необходимых при сборке java-приложений.



Императивный подход хорош до тех пор, пока удобно последовательно описывать сборку каждой из частей программы. Со временем стало очевидно, что для особенно сложных проектов императивный подход создает путаницу из-за большого количества директив по сборке файлов. При этом становится сложно отслеживать правильную последовательность действий при сборке проекта.

Как решение, появились декларативные средства разработки. Они построены на тех же принципах, что и императивные, но большинство действий по сборке выполняется прозрачно для разработчика, а сами действия вызываются через интерфейс верхнего уровня декларативно. Пользователь указывает, *что* он желает собрать, а не *как*, и средство сборки самостоятельно выполняет необходимую последовательность команд. При этом, у пользователя остаётся возможность модификации выполняемых команд сборки.

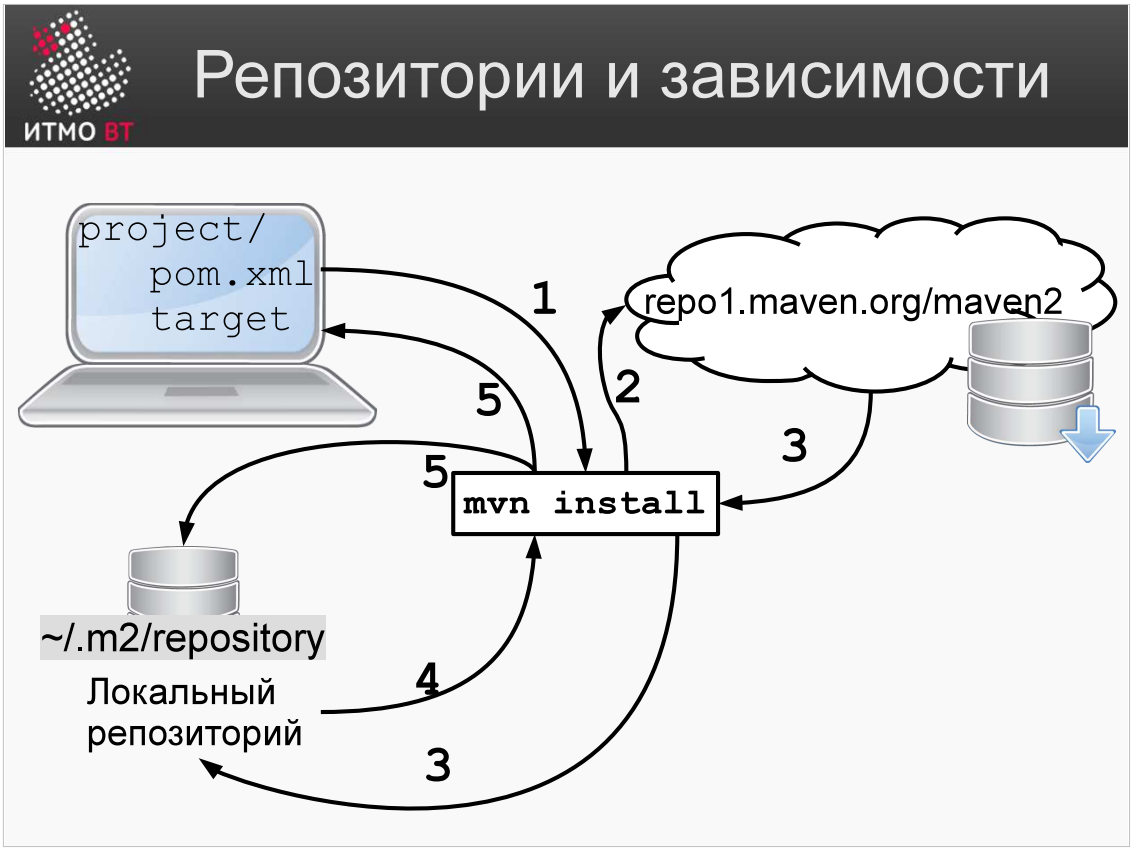
В настоящее время Maven — одно из самых популярных средств сборки для Java, подобные Maven средства сборки существуют и для других языков.

Декларативный подход появился тогда, когда появилась концепция репозиториев — мест в сети, где хранятся дистрибутивы, библиотеки, программные компоненты и т.д. Первые репозитории такого рода появились для пакетов операционных систем. Затем данный принцип перекочевал и в средства сборки.

Maven управляется на базе описания проекта, которое основано на POM — Project Object Model. POM — это XML-файл, состоящий из нескольких частей.

В POM указаны имя, версия и тип программы, местонахождение её исходного кода, существующие зависимости, используемые для сборки плагины, а также альтернативные конфигурации проекта (профили).

Примерами профилей могут быть сборка проекта с включенными возможностями отладки и с максимальной оптимизацией. Выбранный разработчиком профиль можно в явном виде указать при сборке с помощью аргументов утилиты Maven.



На слайде представлен принцип работе репозитория Maven.

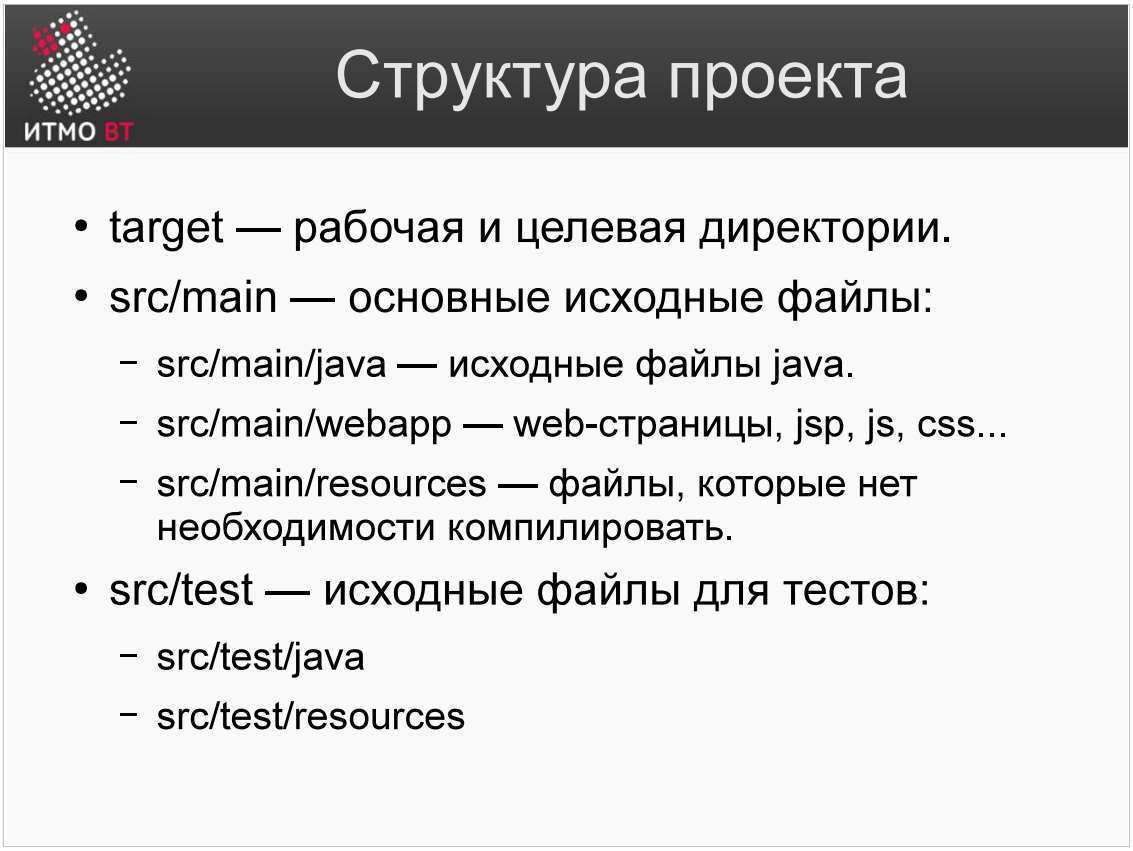
Цель по умолчанию у Maven — install. Её назначение состоит в том, чтобы в локальном репозитории пользователя была собрана версия продукта.

Продукт должен быть собран в директории target и сохранён в локальном репозитории. Сначала (1) Maven читает файл pom.xml с информацией о проекте и определяет зависимости проекта от внешних программных продуктов. Эти зависимости также декларативно описаны в pom.xml.

После этого Maven скачивает необходимые для сборки файлы проекта с удалённого репозитория (2) и сохраняет их в локальный репозиторий (3) до тех пор, пока не скачает все внешние зависимости. Затем осуществляется сама сборка (4). В итоге готовый, собранный продукт помещается в директорию target и в локальный репозиторий (5), после чего возможно помещение продукта в удалённый репозиторий.

Все цели в Maven частично скриптовые, а частично описаны внутри кода модулей, на основе своих POM.

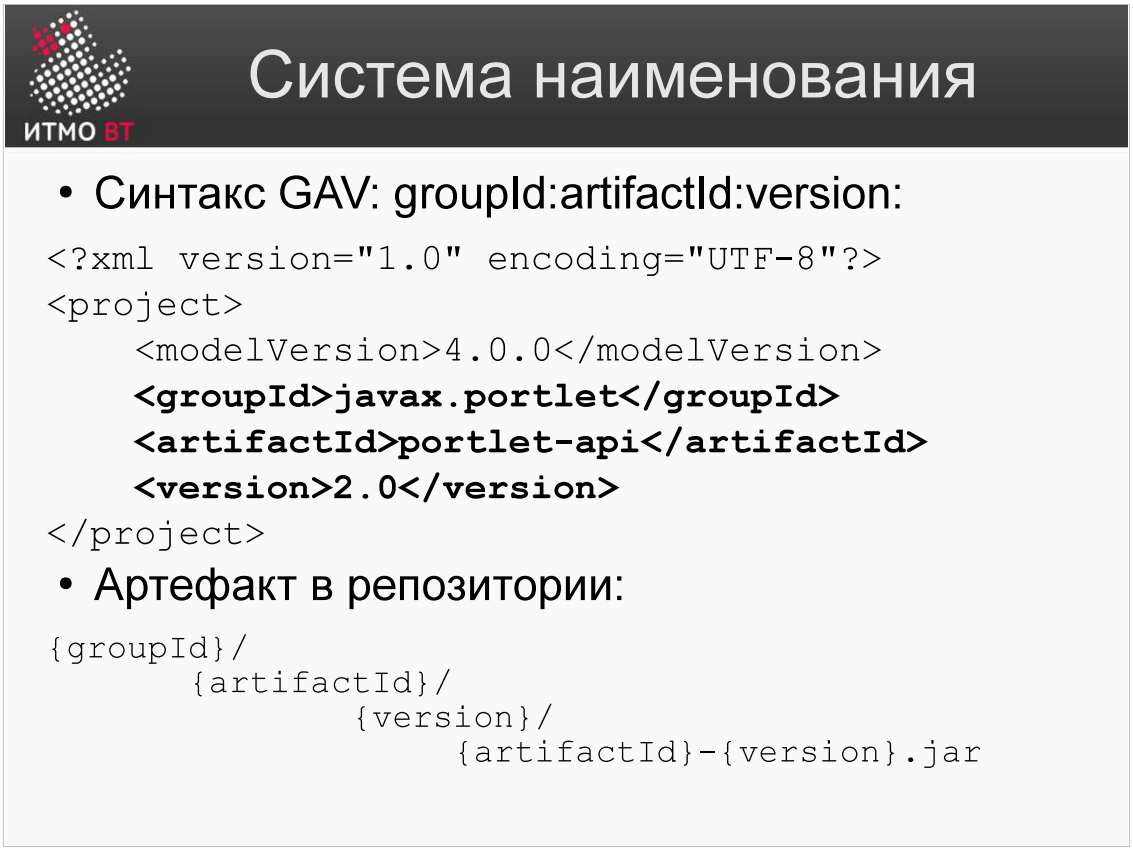
По умолчанию производится обычная последовательная сборка кода. При необходимости её можно распараллелить.



В Maven, в отличие от Ant, существует система каталогов проекта по умолчанию.

В ветке src/main расположены основные исходные файлы. Некоторые из них необходимо компилировать (src/main/java), некоторые непосредственно копируются в целевую директорию target. Отдельно выделяются зависимые от языка пользователя ресурсы в файле .propeties. С точки зрения компиляции, для директорий исходных файлов и ресурсов и выполняются разные цели.

Аналогично разделены каталоги для тестов, относящихся к исходным файлам Java и к ресурсам.

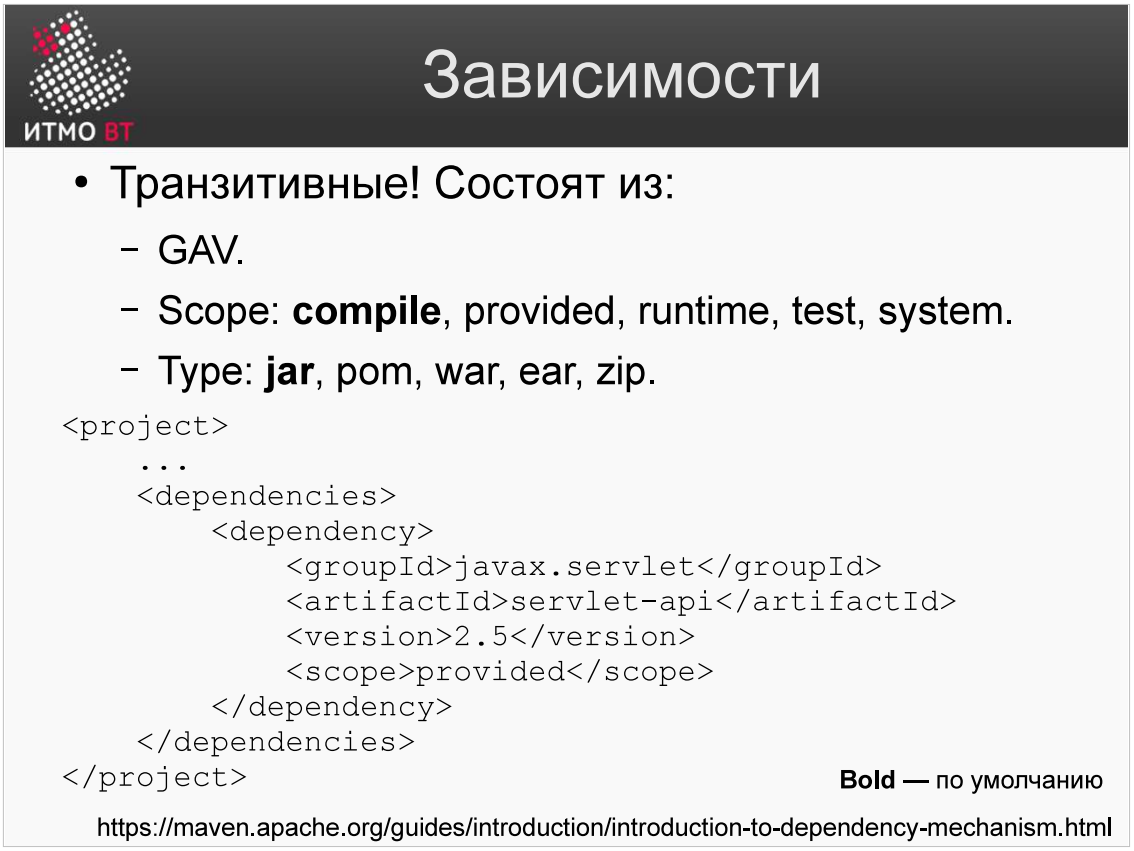


Система наименования модулей в Maven строится по принципу GAV: groupId:artifactId:version. Любая внешняя зависимость (а также продукт пользователя) описывается при помощи данной системы наименований.

На слайде приведён пример файла pom.xml с описанием проекта. Аналогичным образом описываются и внешние ресурсы (в репозиториях Maven можно изучить их файлы pom.xml).

В Maven доступна реализация почти любого программного продукта, разработанного для Java. При компиляции внешние зависимости сохраняются в локальном репозитории.

Существует некоторое неудобство для разработчика, состоящее в том, что системе сборки необходимо при периодически обращаться к ресурсам сети Интернет для получения индекса базы и обновления локального репозитория. В случае, если это мешает разработке (например, если интернет недоступен), необходимо в явном виде запретить Maven обращаться к внешним ресурсам.



Как говорилось ранее, зависимости в Maven описываются в GAV-синтаксисе и скачиваются автоматически. На слайде приведен пример описания зависимости.

У зависимости существует т.н. scope, или область действия. Данный параметр указывает, в какой момент жизненного цикла приложения применяется данная зависимость. По умолчанию используется scope=compile, указывающий, что зависимость должна учитываться при компиляции и загружаться при вызове компилятора. Зависимости со scope=test используются только во время тестирования.

Scope=provided используется в Enterprise Java. Все Enterprise-приложения работают в т.н. контейнере — элементе, управляющем жизненным циклом веб-приложений. Scope=provided говорит о том, что связывание для данных внешних связей должно происходит в момент размещения приложения в контейнере исполнения.

Зависимости в Maven являются *транзитивными* в том смысле, что если включенный подпроект зависит от определенного набора библиотек, то включающий проект так же будет от них зависеть.

В приведённом на слайде жизненном цикле Maven-приложения последовательно вызываются цели и стадии его сборки.

Фаза generate sources (генерация исходных кодов) применяется тогда, когда в приложении часть исходного кода автогенерируется (например, на базе DSL-грамматики собирается компилятор предметно-ориентированного языка). После этого исходный код компилируется (фаза compile), а затем компилируются тесты на базе приложения (фаза test-compile).

После этого приложение тестируется (фаза test), собирается в пакеты (фаза package) и производится интеграционное тестирование (фаза integration-test). Приложение устанавливается в локальном репозитории (фаза install).

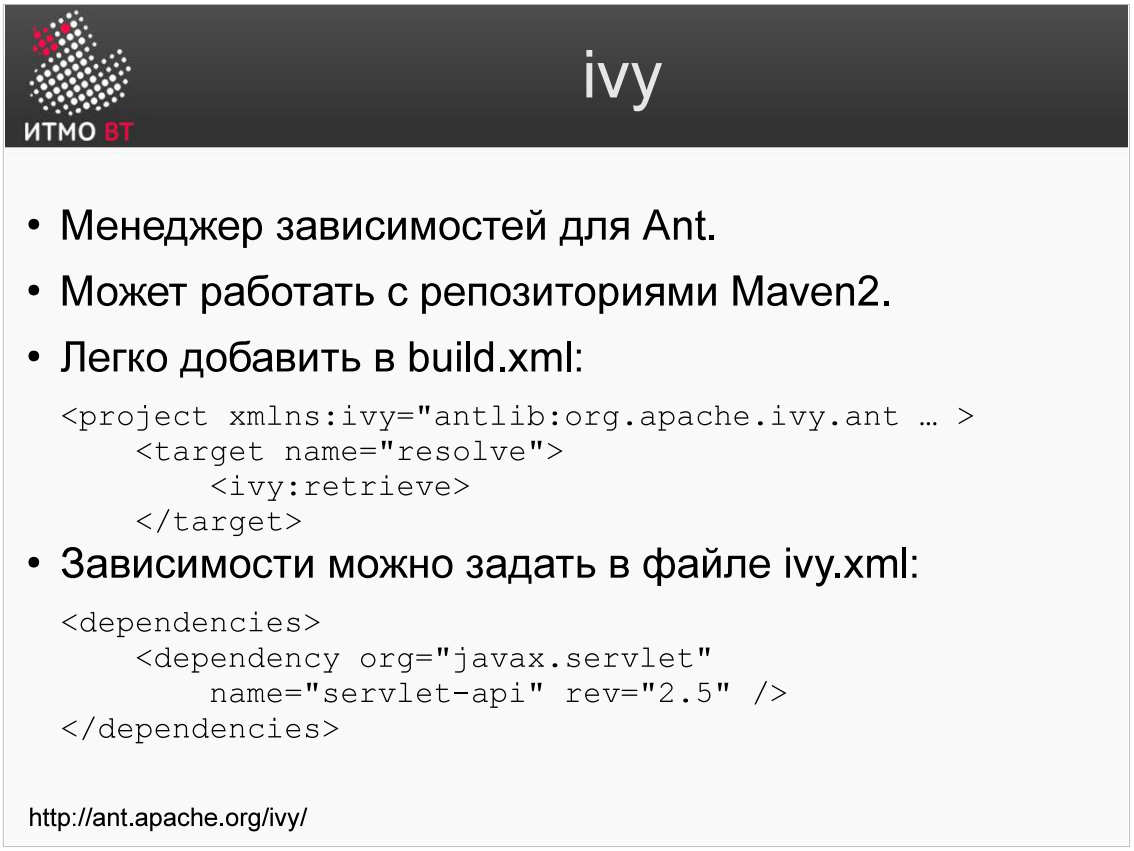
Последняя цель, deploy, вызывается, если указан сервер приложений, и заключается в установке приложения на этот сервер.

Плагины в Maven используются для управления сборкой. Их надо в явном виде включать в модель POM.

Если в Ant все действия над приложениями надо в явном виде прописывать, то в Maven достаточно декларативных высказываний.

Вызов плагинов управляется общей логикой сборки проекта. Плагины в качестве точек входа содержат цели, связанные с ЖЦ сборки.

В число целей, например, входят цели упаковки (packaging) и цели, связанные с автоматической генерацией отчетов по разработке.



Ivy — это менеджер зависимостей для Ant, который серьёзно упрощает работу с продуктом. До этого в Ant зависимости приходилось собирать вручную, этот процесс был длительным и включал в себя много процедур.

Ivy решает проблему скачивания и использования сторонних библиотек за пользователя и способен, в частности, работать с репозиториями Maven2.

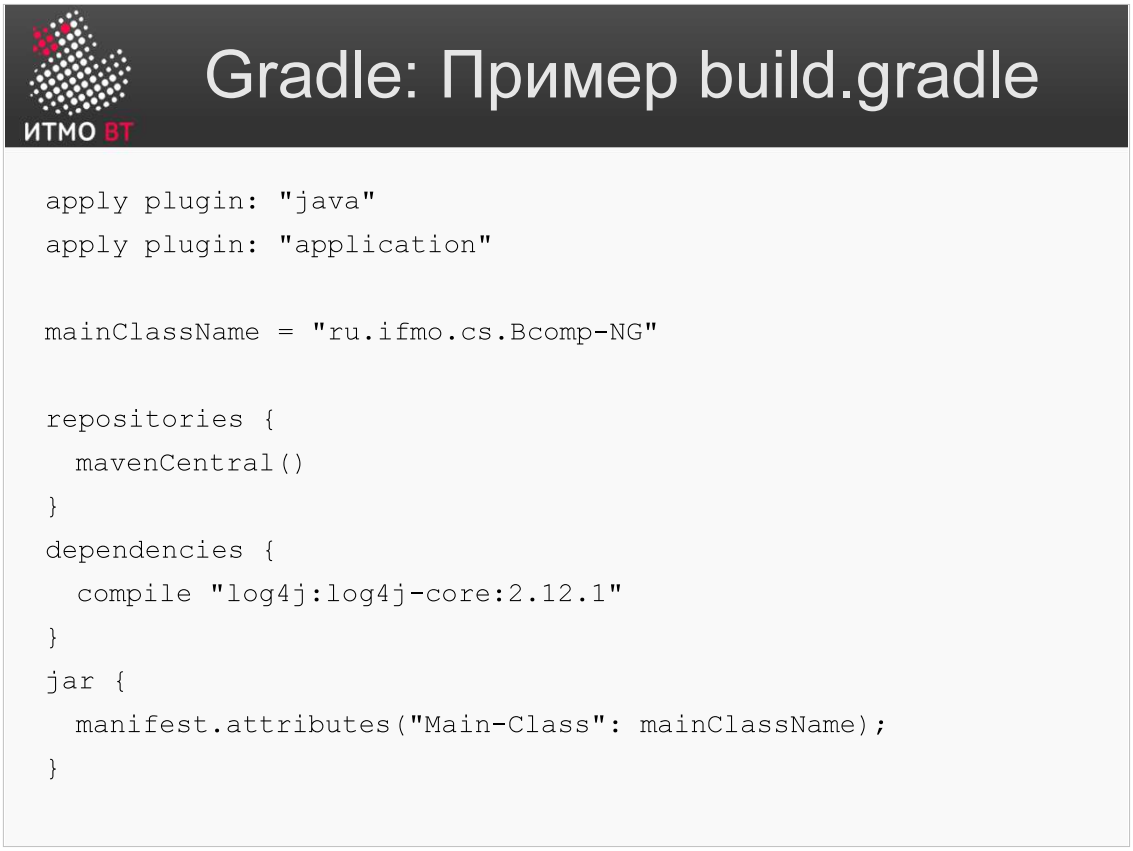
Ivy легко добавить в build.xml так, как это показано на слайде. После этого он может обращаться к удалённому репозиторию и скачивать отсутствующие зависимости.

Зависимости задаются в отдельном файле, который по умолчанию называется ivy.xml.

Проблемой декларативной сборки Maven является то, что она по сути своей является ограниченно-декларативной. Это значит, что пока ваше приложение подчиняется основной логике, которая заложена в плагины — проект конфигурируется быстро и собирается хорошо. Если по какой-либо причине вам нужно собрать что либо, не описываемое плагином — тут начинаются сложности. Скорее всего, вам придется разработать собственный плагин, что не является тривиальной задачей. Кроме того, бывает сложно разобраться в XML-файле проекта, который содержит много проектов и зависимостей. Де-факто, XML является удобным форматом для его программного анализа, а не для написания человеком.

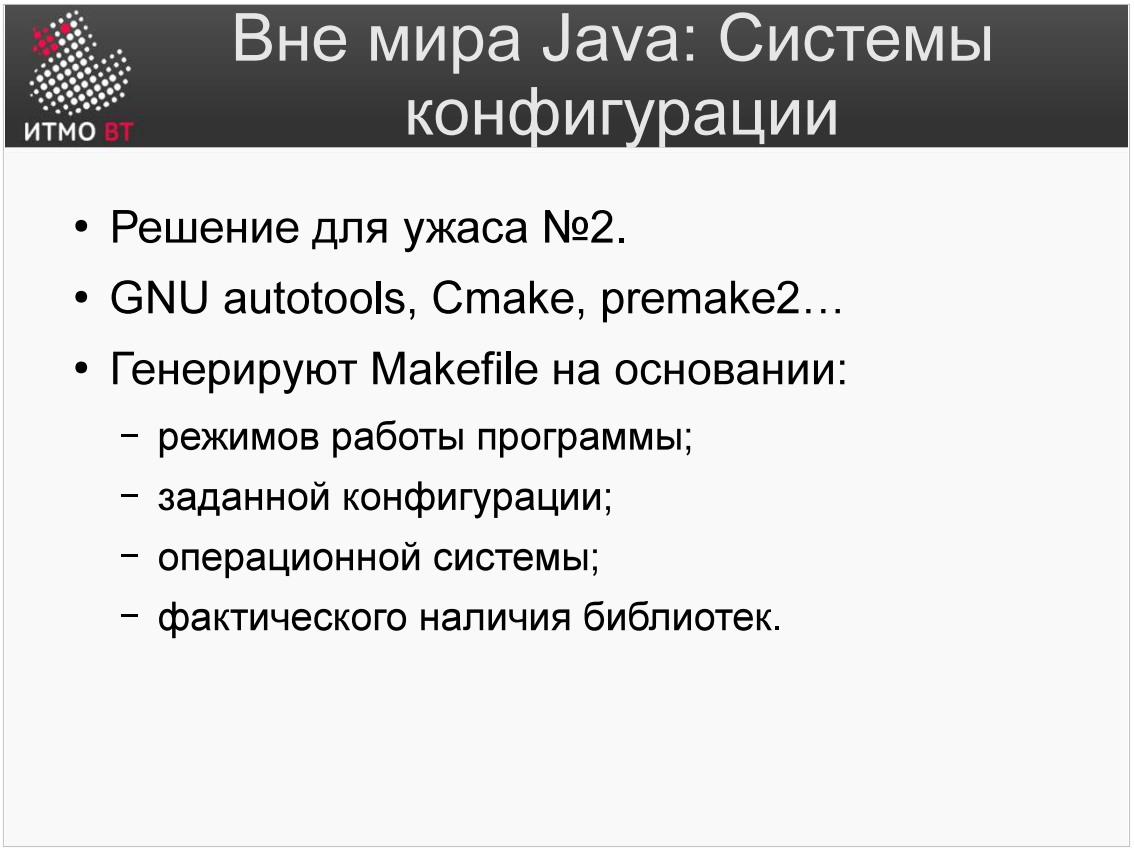
Средством, которое является удачным последователем Ant и Maven, которое получило в современном мире разработчиков большое распространение, и которое сообщество считает «стильным, модным и молодежным», является Gradle. Основными достоинствами его являются:

- Нейтральность к языкам программирования. В отличие от Maven, который предназначен, в основном, для сборки Java-приложений, gradle никак не ограничивает вас в языках программирования. Вы можете указать, как собрать приложение, которое вы можете разрабатывать на множестве различных языков.
- Преемственность описания зависимостей и возможность использования настраиваемых репозиториев зависимостей, в том числе от Maven, используя различные нотации описания.
- Использование лаконичного DSL (Domain Specific Language — проблемно-специфичного языка) для описания сборки, который сам по себе разработан и использует возможности Groovy.
- Инкрементальная и параллельная сборка — позволяет запускать плагины только при наличии действительной необходимости выполнения задач по сборке, и экономит время сборки проекта. Инкрементальная сборка существует в различных системах сборки, но, например, в Maven все равно тратится достаточное количество времени на выполнение плагина, даже если собирать нечего.



Как видно из приведенного примера, задание конфигурации сборки простое и прозрачное. Здесь декларировано применение двух плагинов, первый из них определяет набор задач для языка Java, а второй предполагает сборку jar-файла отдельно стоящего приложения. В качестве репозитория зависимостей задается использование основного сервера Maven. Зависимости для конфигурации compile указаны в соответствующей секции в виде, принятом для Maven GAV.

На простом примере сложно показать всю гибкость и простоту добавления новых задач для плагинов и задания процесса сборки, предлагаем вам самостоятельно обратиться к документации и примерам Gradle.

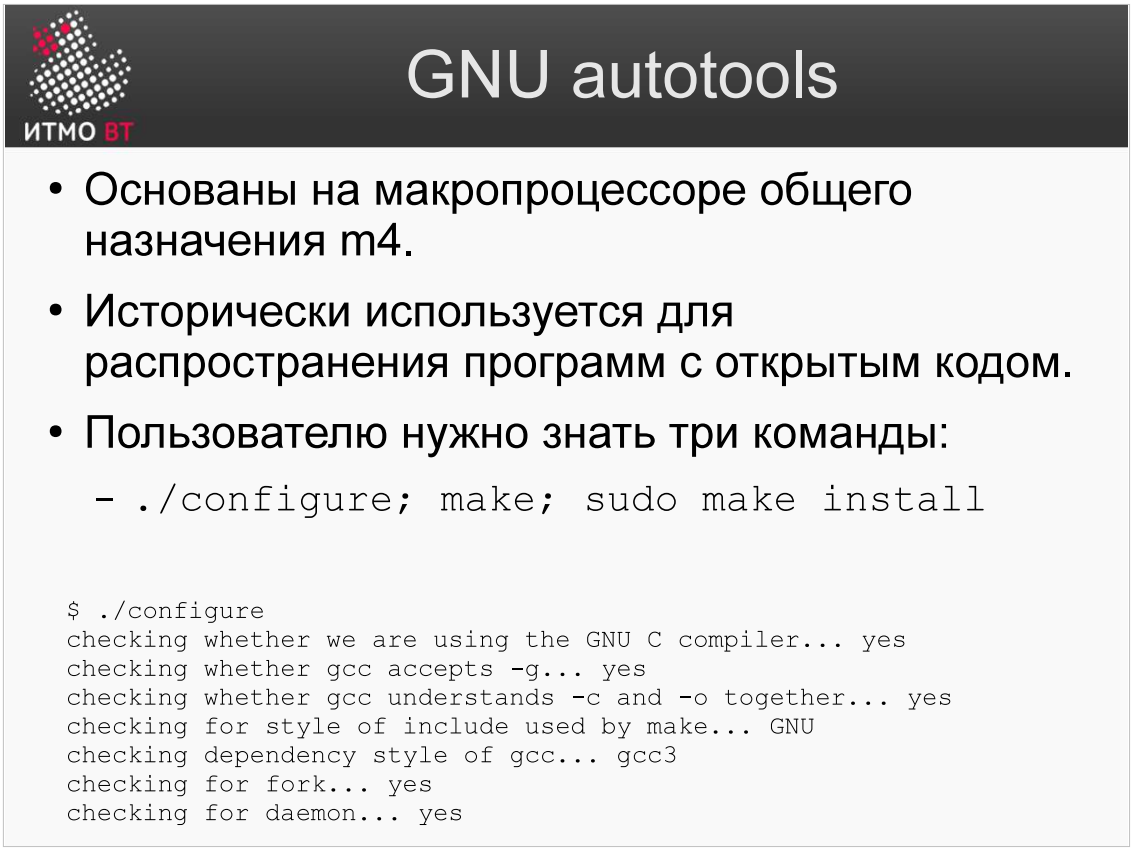


Вне мира Java тоже существуют системы, подобные менеджеру зависимостей.

Такие утилиты, как GNU autotools, Cmake, premake2, выполняют сходный набор действий, но, в основном, не скачивают удалённые зависимости, а просто автоматизируют сборку продукта, делая её более декларативной.

Данные системы используются для генерации корректного Makefile, используя в качестве системы сборки Make на нижнем уровне. На процесс генерации оказывают влияние режимы работы программы, заданная конфигурация, операционная система, и фактическое наличие библиотек.

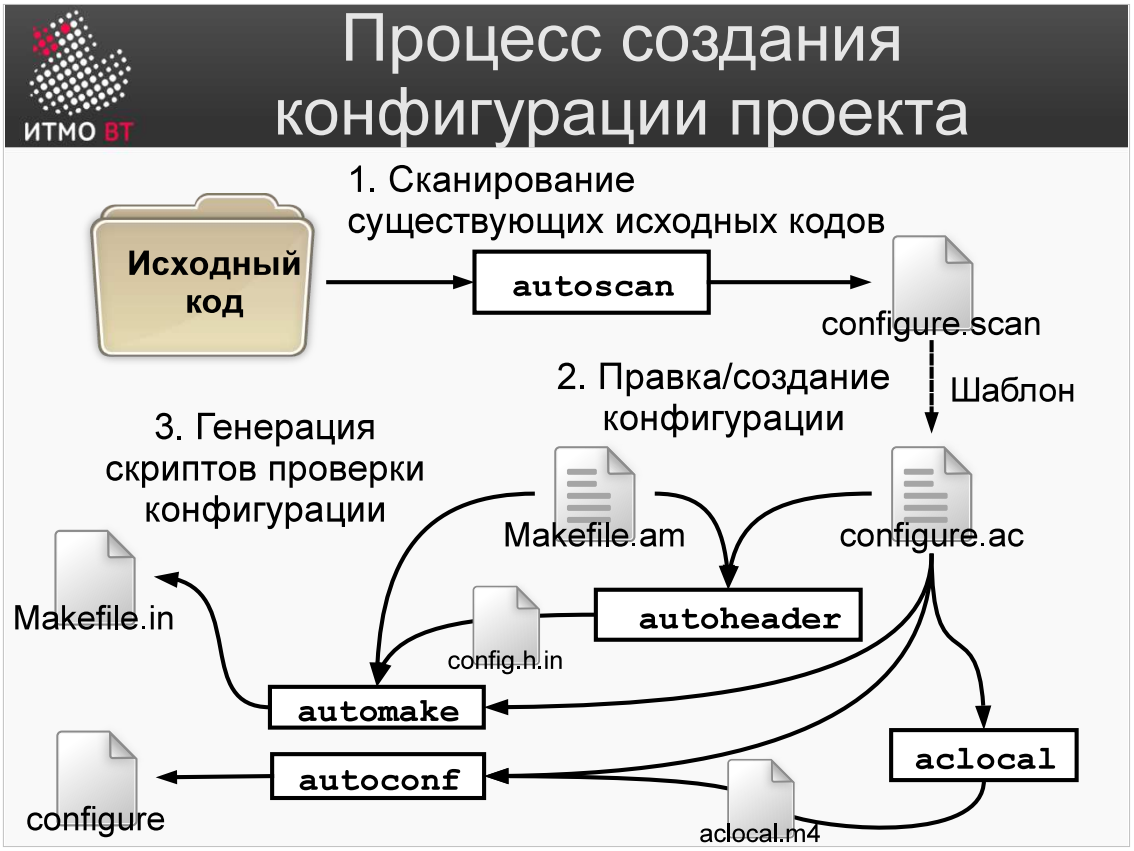
GNU autotools — самая старая и проверенная из представленных систем, хотя она и считается достаточно сложной для освоения.



GNU autotools основаны на макропроцессоре общего назначения m4, существовавшего с момента написания первых версий Unix. Макропроцессор — это программа, преобразующая входной текст в выходной, руководствуясь правилами замены последовательностей символов, называемых правилами макроподстановки. Макроподстановки по принципу работы подобны директивам препроцессора в C.

GNU autotools исторически используется для распространения программ с открытым кодом. Для сборки системы из исходных материалов пользователю нужно знать всего три команды, применяемые последовательно: ./configure, make и sudo make install. Этого достаточно для установки собранного продукта. На слайде приведен процесс конфигурации.

Стоит отметить, что GNU autotools полностью платформонезависимы.



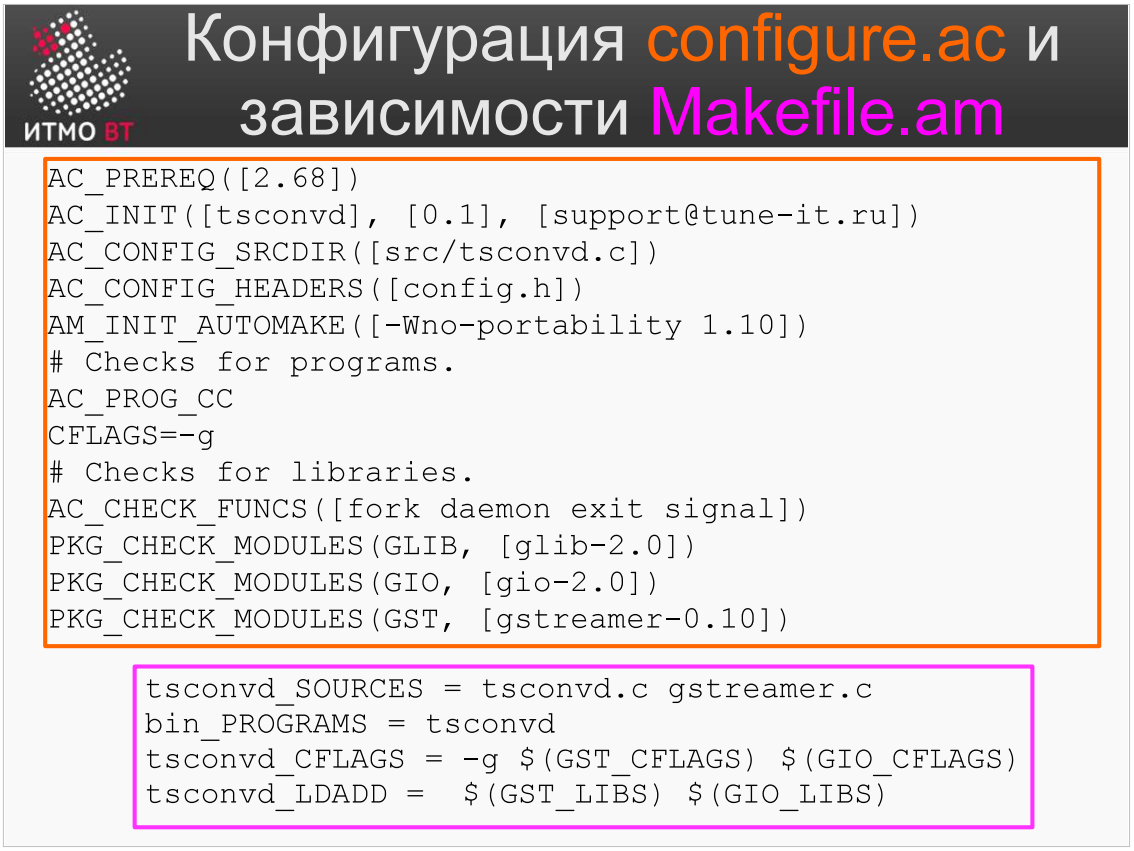
Как разработать конфигурацию программного продукта для последующего использования на разных платформах?

С помощью утилиты autoscan последовательно сканируется существующий исходный код, и на основании имеющейся базы данных выделяются участки кода, которые могут зависеть от особенностей платформы и операционной системы. По результатам сканирования формируется шаблон конфигурационного файла configure.scan, который затем редактируется вручную. В результате получается файл configure.ac.

Также вручную создается файл Makefile.am. В нём в явном виде указываются названия исполняемых программ, из каких исходных файлов они должны быть собраны, и другие зависимости. Такой файл создаётся в каждом из подкаталогов исходных файлов.

После этого вызывается последовательность команд. Первая из них, autoheader, создает шаблон config.h.in для файла config.h, а команда aclocal проверяет, что установлено на локальной системе разработчика, и должно быть использовано в проекте.

Затем запускаются утилиты automake и autoconf, в результате чего в дистрибутиве появляются файлы Makefile.in и configure, которые позже используются при определении текущей конфигурации на целевой системе, где будет собираться ПО.

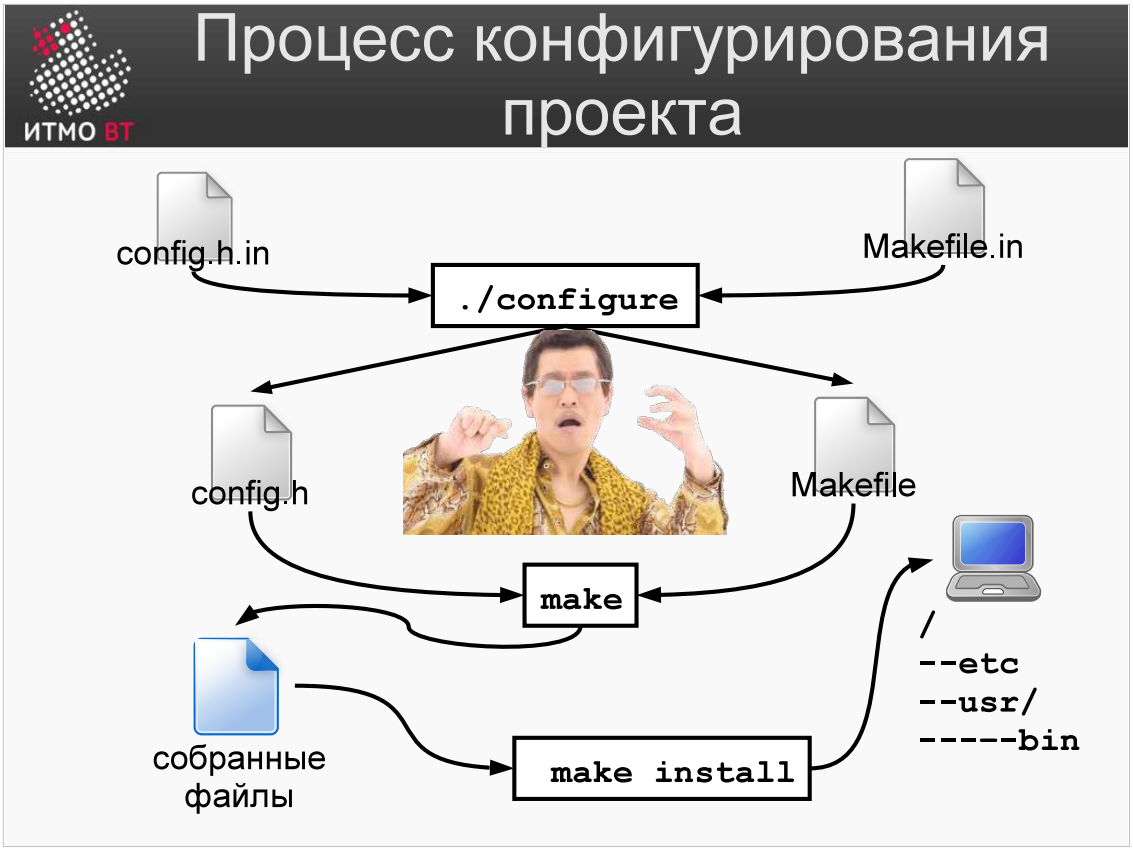


Рассмотрим подробнее, что должно содержаться в основном файле конфигурации проекта (configure.ac) и зависимостях исходных файлов Makefile.am.

В файле configure.ac сначала описывается продукт (имя проекта, версия, где находятся исходники и заголовки и т.п.). Таким образом, autoscan подготавливает шаблон для продукта. Затем проверяется наличие компиляторов и необходимых системных библиотек, а также отдельных функций в этих библиотеках.

На этапе конфигурации будут создаваться небольшие тестовые программы. Они будут тут же компилироваться выбранным компилятором, запускаться, и проверять наличие системных функций. Если эти функции есть, то в config.h будет указано их наличие. В тексте программы можно указать реакцию на наличие или отсутствие тех, или иных системных функций в соответствии с config.h.

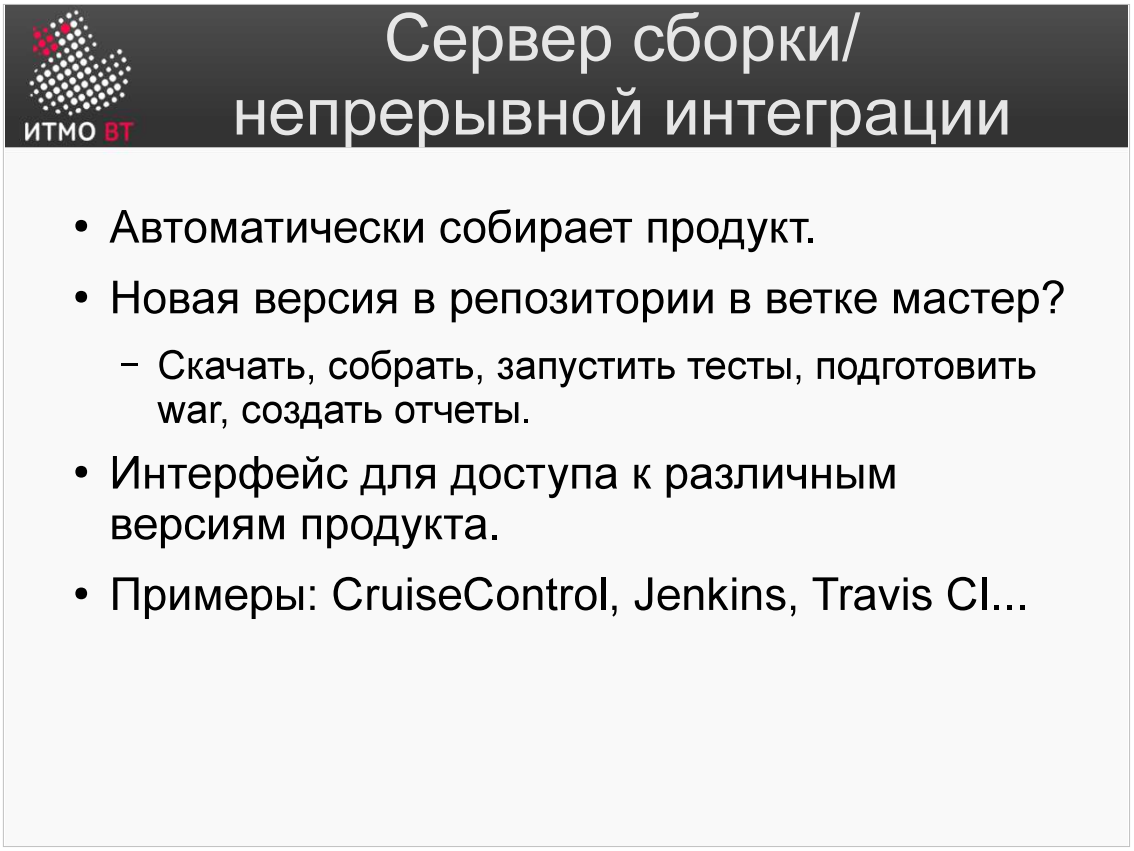
Второй файл, Makefile.am, содержит описание исходников, название модуля, и специфические ключи компиляции для каждого каталога сборки.



На слайде приведён процесс конфигурации проекта. После скачивания ПО запускается команда ./configure, и на основании текущей конфигурации создаётся файл config.h и необходимые платформозависимые файлы. После этого используется команда make, и все собранные файлы при помощи команды make install распределяются в необходимые системные каталоги.

При этом существует большое количество умолчаний, например, о местах хранения файлов, и ./configure выдает именно те директории, где должны храниться конфигурационные файлы.

Все эти умолчания можно изменить при помощи задания опций конфигурации, которые можно получить при помощи ./configure -help

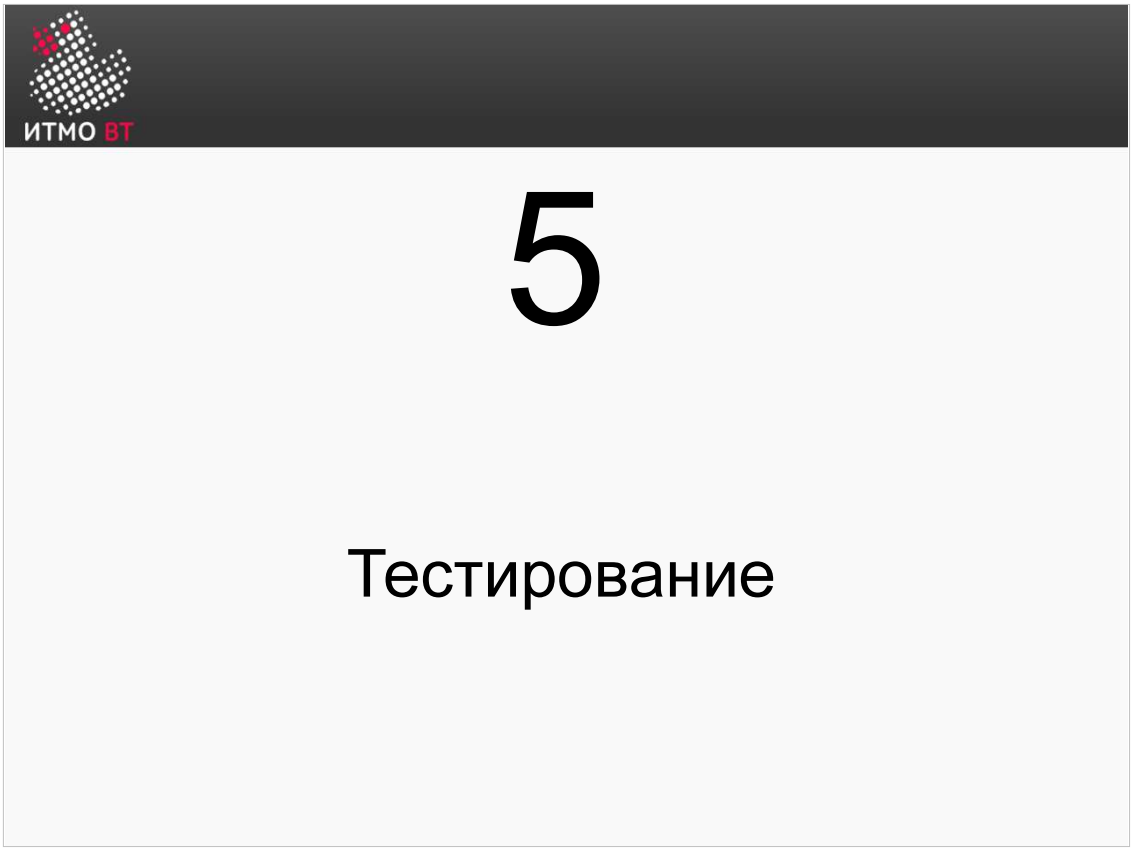


Указанные выше средства и способы удобны для однократной сборки продукта. Для выполнения этой операции в автоматическом режиме существуют отдельные средства, называемые серверами сборки или серверами непрерывной интеграции. Их основное назначение — сборка новой версии продукта при наступлении заданных администратором или разработчиком условий. Такими условиями могут быть новое обновление исходных кодов в ветке "мастер", наступление определенного момента времени (пример: ночная или еженедельная сборка) и т.д.

Помимо, собственно, сборки, билд-сервер может проводить тесты, снимать метрики с программного кода, производить автозапуск и т.д.

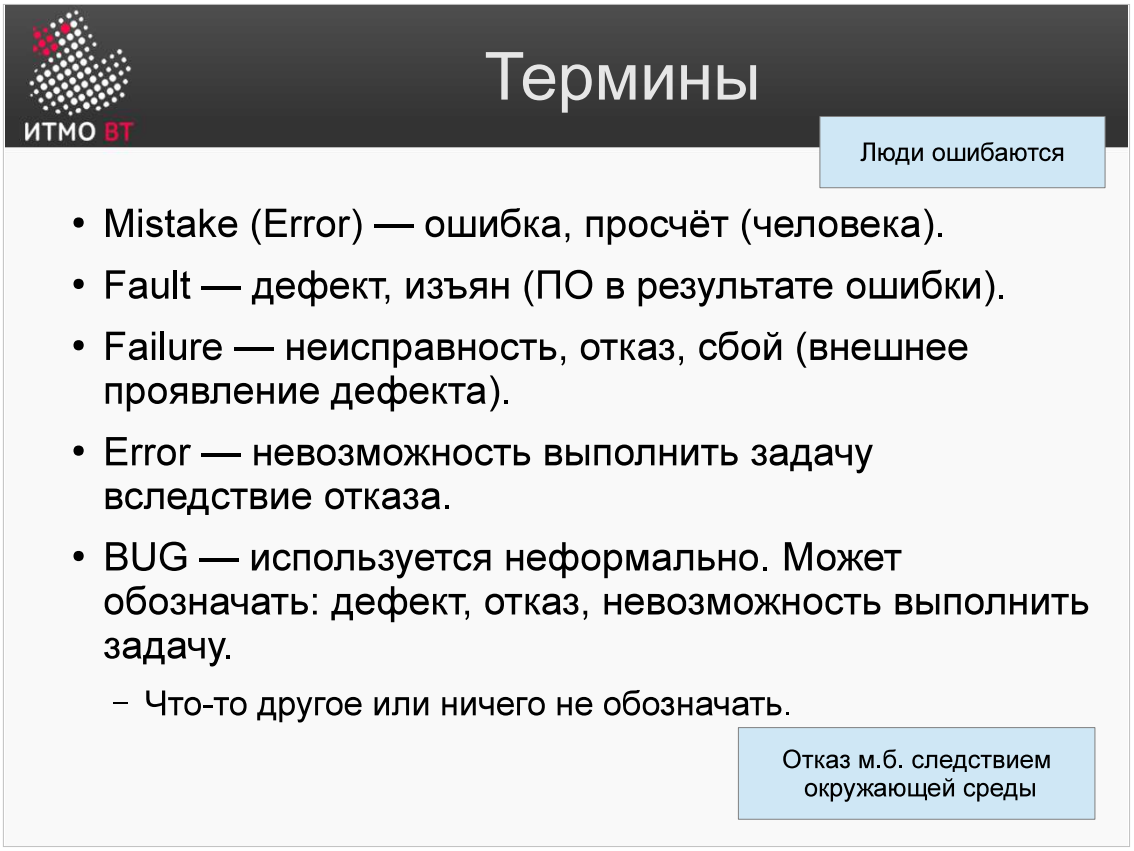
Сервер сборки также предоставляет доступ ко всем существенным собранным версиям продукта для скачивания и / или немедленной установки у пользователя.

Примерами таких систем для Java являются Jenkins и Travis CI.



Тема данной лекции - тестирование.

В ней будут рассмотрены все существующие на сей день аспекты, связанные с тестированием ПО.



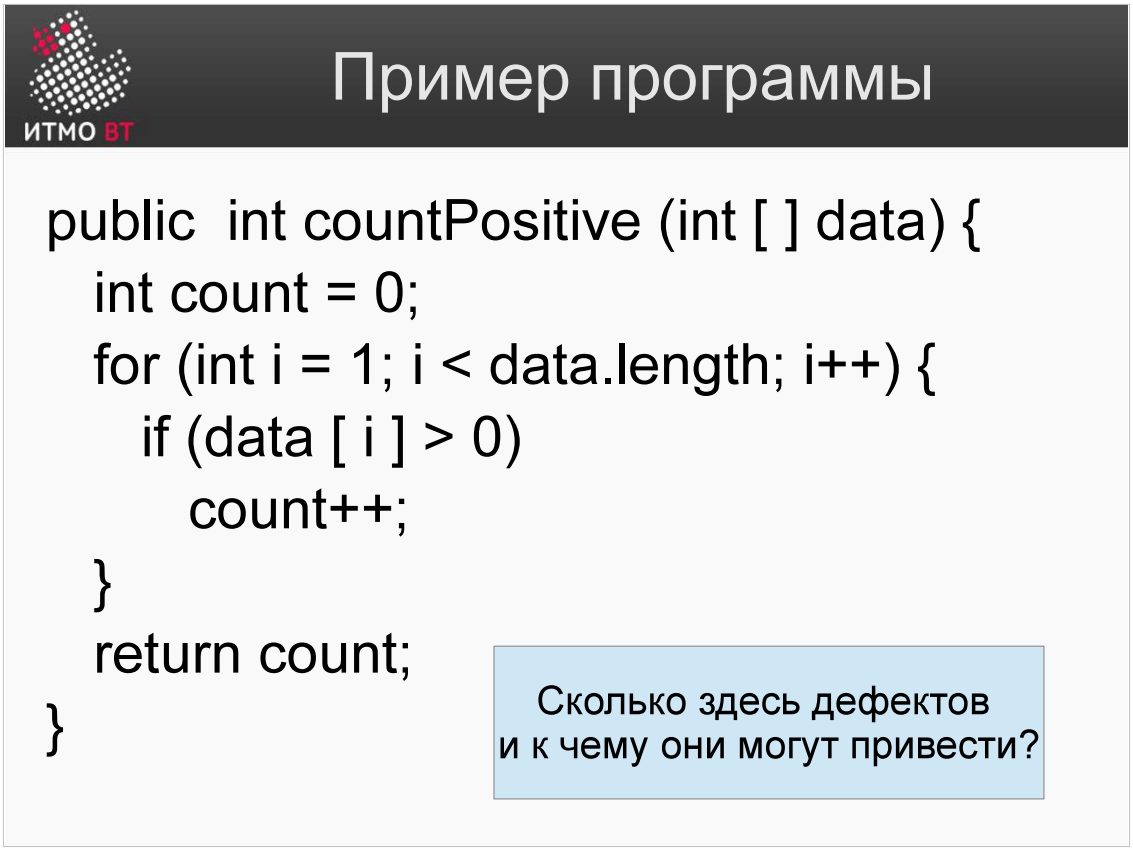
На слайде представлена общая терминология тестирования ПО, предлагаемая к использованию организацией, подтверждающей квалификацию тестировщиков (ISTQB — International Software Testing Qualification Board). На практике она, к сожалению, часто используется неформально.

- Mistake — ошибка разработчика. Человеческое деяние, которое в конечном итоге привело к получению неверного результата. В широком смысле — непреднамеренное отклонение от истины или правил.
- Fault — дефект, изъян. Неверный шаг в алгоритме (или неверное определение данных) в компьютерной программе. Следствие ошибки, потенциальная причина неисправности.
- Failure — неисправность, отказ или сбой — наблюдаемое проявление дефекта, в том числе, крах или падение программы.
- Error — невозможность выполнить с использованием программы задачу, получить верный результат.

Если вы хотите создать счастливую семью, но заблуждаетесь в качествах своего избранника, вы совершаете ошибку (mistake). Когда вы женитесь или выйдете замуж, то ваш брак будет содержать изъяны (faults). Это приведет к постоянным ссорам, "неисправностям" в отношениях (failures) и ваша несовместимость будет проявляться всё чаще и чаще. Невозможность быть счастливым, и соответственно, невозможность продолжать быть семьей (error), приведет к разводу.

Термин "BUG" является одним из основных неформальных терминов. Он обычно используется для обозначения любой некорректности в работе программы. На практике "BUG" не имеет формального определения, и обычно означает ошибку, дефект, сбой или все вместе. "Они не подходят друг другу" — приблизительно соответствует термину "BUG".

Важно отметить, что на отказы может сильно влиять окружение программы. В семейной жизни вы можете боготворить вашего избранника, но столкнуться с суровой реальностью в виде тещи или свекрови.



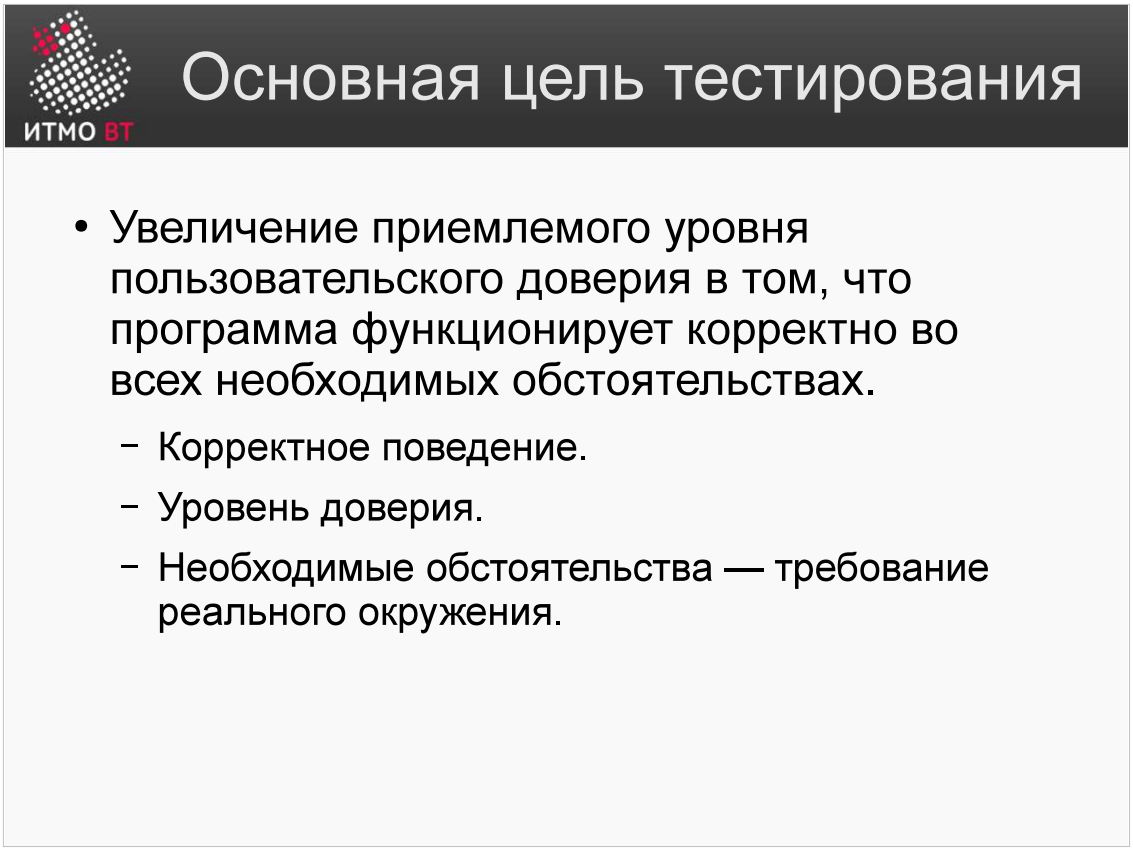
На слайде приведён пример программы подсчёта положительных чисел с дефектами.

Первый дефект: нумерация в цикле начата не с нуля. Данная ошибка зависит от данных и является условной, так как может проявиться или не проявиться. Хуже всего то, что данная ошибка может привести к сбою после нескольких лет успешного функционирования программы, когда пользователь уверен, что никаких дефектов в программе нет, и быть не может. "Я его любила, а теперь он храпит по ночам!"

Второй дефект: в качестве исходного параметра data можно передать нуль, что приведет к сбою при попытке узнать длину списка data. Данный дефект обычно обнаруживается существенно раньше, чем предыдущий. "Он не купил мне новые туфли, которые так подходят к платью, что я видела вчера!"

Организация International Software Testing Qualifications Board (ISTQB) устанавливает корпус знаний, необходимых тестировщикам в их повседневной работе, и определяет общую терминологию и подходы к разработке и проведению тестов. Этот корпус знаний размещен на сайте www.istqb.org. Те документы, которые переведены на русский язык, размещены на сайте www.rstqb.org.

В соответствии с требованиями данной организации, существует несколько целей тестирования, которые приведены на слайде. Важно понимать, что невозможно гарантировать и доказать полное отсутствие ошибок в программе даже после тестирования.

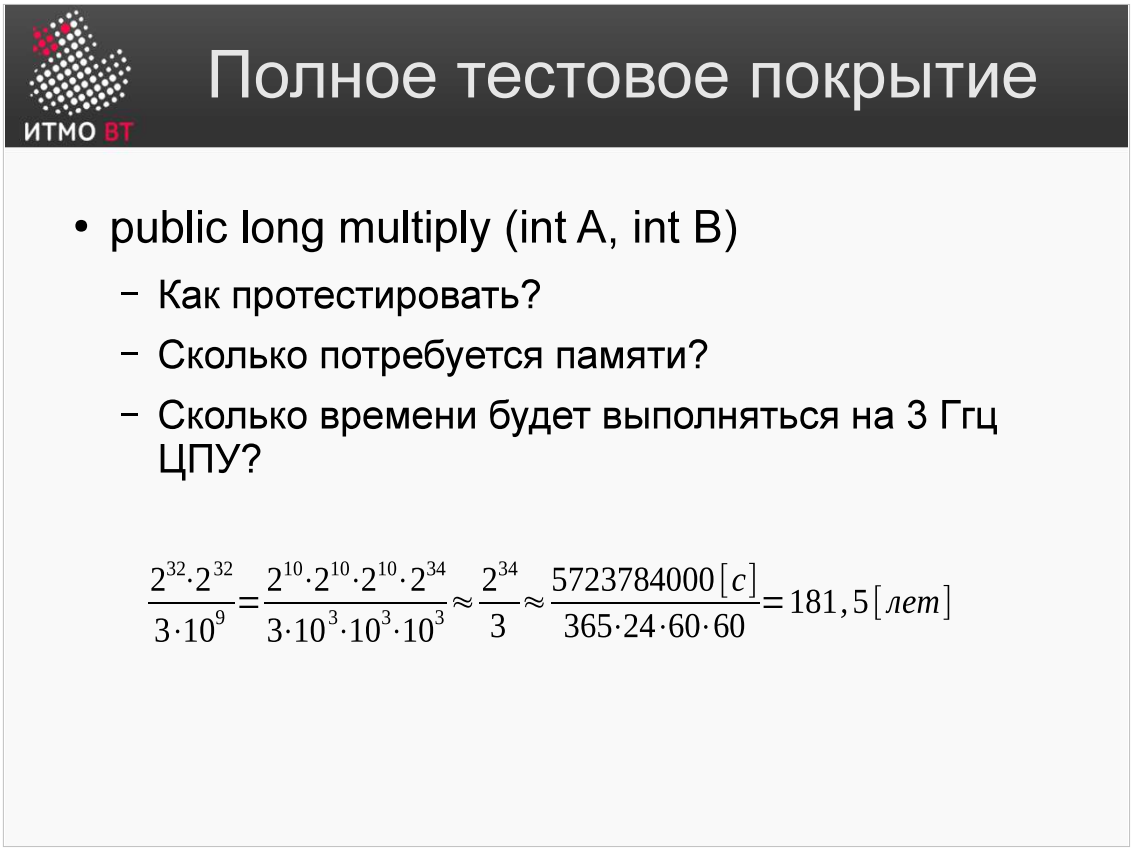


С разумной точки зрения, цель тестирования можно определить так: увеличение приемлемого уровня пользовательского доверия в том, что программа функционирует корректно во всех необходимых обстоятельствах. Это значит, что существует определенное доверие пользователя разработчику (обычно, достаточно субъективное). Выполняя тестирование, разработчик предлагает пользователю на основании объективных фактов (проведения набора тестов, выбора тестового покрытия для каждой части программы) поверить в то, что разработчик использовал все разумные средства, чтобы показать *присутствие* и исправление (а не отсутствие!) дефектов в продукте.

Корректное поведение — программа должна соответствовать предъявленным к ней требованиям. Помимо, собственно, требований, источниками информации о корректности являются, например, особенности рынка ПО и те стандарты, которые существуют в бизнес-среде. Существует много разнообразных источников информации о корректном поведении программы.

Пользовательское доверие должно быть заработано при помощи демонстрации разработчиком измеряемых показателей. Существуют метрики, позволяющие измерить уровень дефектов в коде, например, число открытых в неделю новых дефектов, которое должно уменьшаться в процессе доработки, тем самым повышая уровень пользовательского доверия.

Важно проводить тестирование в той же среде, в которой программа будет выполняться. Например система управления вузом, где учатся 5000 студентов должна быть проверена с одновременными запросами от именно 5000 или 10000 (с расчётом на рост) студентов, а не на 1000 или 100000.



Понятие "тестовое покрытие" включает в себя то, насколько код приложения покрыт тестами, которые могут находить известные и потенциальные дефекты. Полное тестовое покрытие подразумевает покрытие тестами всего кода и всех возможных вариантов развития событий, возможных в данном коде. Полное тестовое покрытие, если и достижимо, то ценой очень больших затрат.

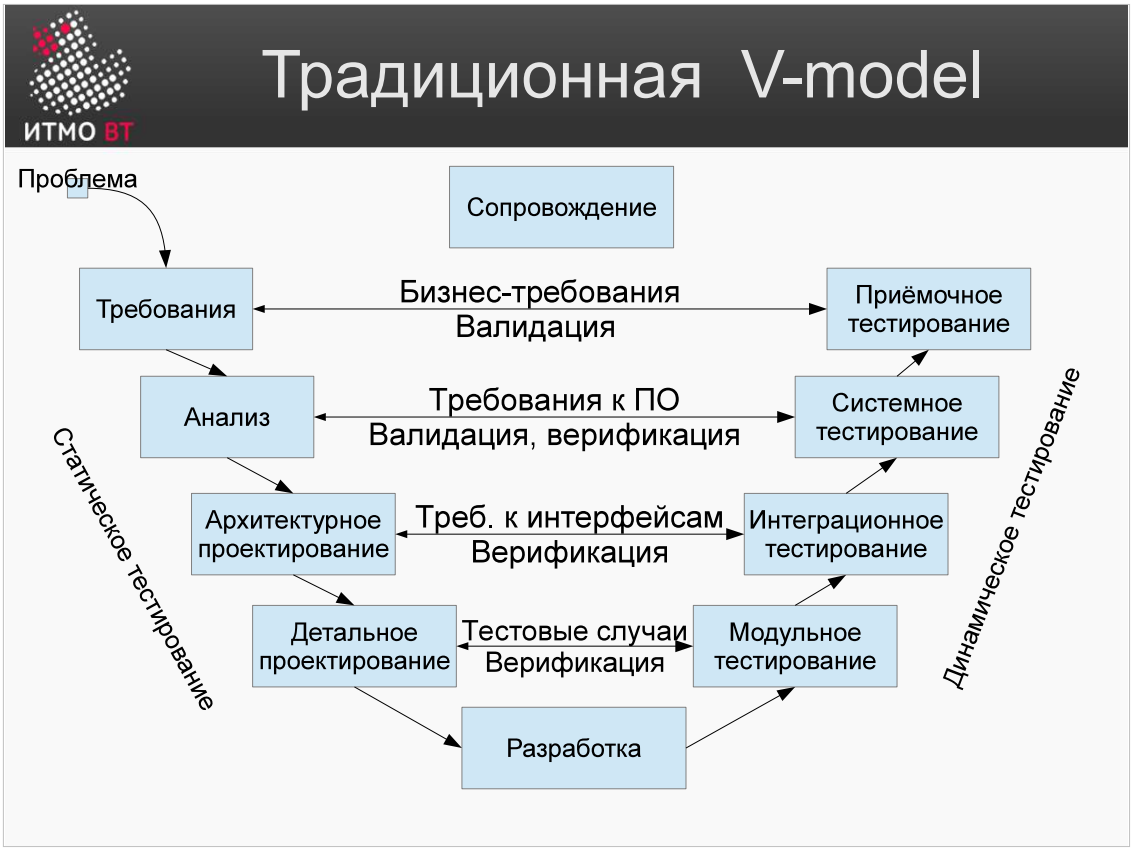
В качестве иллюстрации приведена функция умножения двух чисел. Если оба множителя имеют степени 2^32, то тогда для полного покрытия необходимо провести 2^64 операций. Предположим, что 1 операция умножения выполняется за 1 такт процессора. Тогда, учитывая что 2^10 приблизительно равно 10^3, однопоточное приложение на 3Ггц ЦПУ будет проводить тестирование функции в течение примерно 181,5 лет.

$$\frac{2^{32}\cdot 2^{32}}{3\cdot 10^9}=\frac{2^{10}\cdot 2^{10}\cdot 2^{10}\cdot 2^{34}}{3\cdot 10^3\cdot 10^3\cdot 10^3}\approx\frac{2^{34}}{3}\approx\frac{5723784000[c]}{365\cdot 24\cdot 60\cdot 60}=181,5[лет]$$

Самый простой способ организации тестирования — по таблице, где каждая строка содержит оба множителя, и результат — эталон. В приведённом примере хранение всех возможных значений и корректных результатов потребует места, превышающего возможности всех современных компьютеров.

Важно заметить, что ни в коем случае нельзя для построения подобной таблицы использовать ту же программу, что и для реального умножения. Иначе значимость эталонного результата может быть поставлена под сомнение.

Естественно, что в реальной программе вариативность данных ещё больше, чем в примере.



В V-model каждой стадии разработки ПО соответствует свой уровень тестирования. Код ПО, иерархически организованный в виде модулей, покрывается модульными тестами, а архитектура и взаимодействие слоёв архитектуры между собой — интеграционным тестированием.

Формирование и анализ требований покрываются системным тестированием. Реализующая эти требования программа принимается заказчиком в эксплуатацию во время приёмочного тестирования, где проверяются основные характеристики программы, которые были заложены в требованиях.

Валидация — мероприятия по проверке корректности требований к программному обеспечению. Верификация — проверка соответствия ПО сформулированным требованиям.

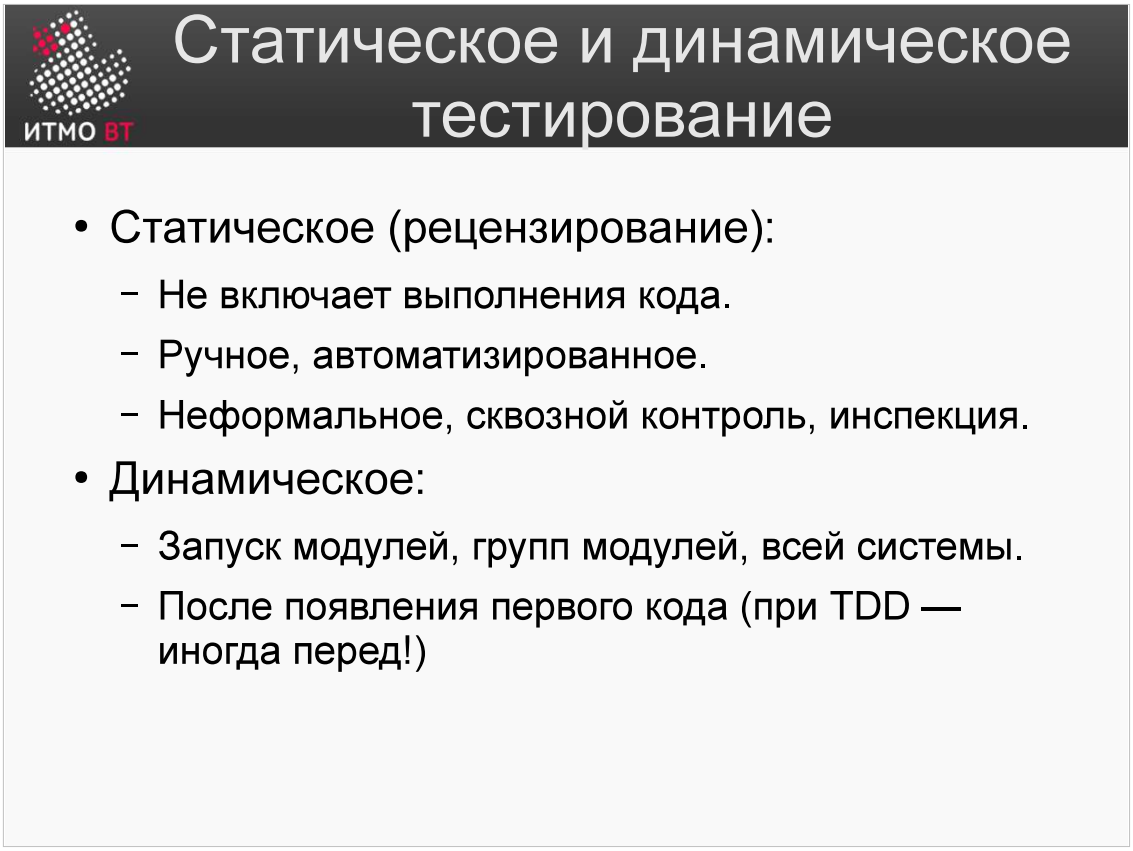
Валидация: "Have we done *the right thing*?"

"Я правильно сделал, что женился?"

Верификация: "Have we done *the thing* right?"

"Я выбрал правильную супругу?"

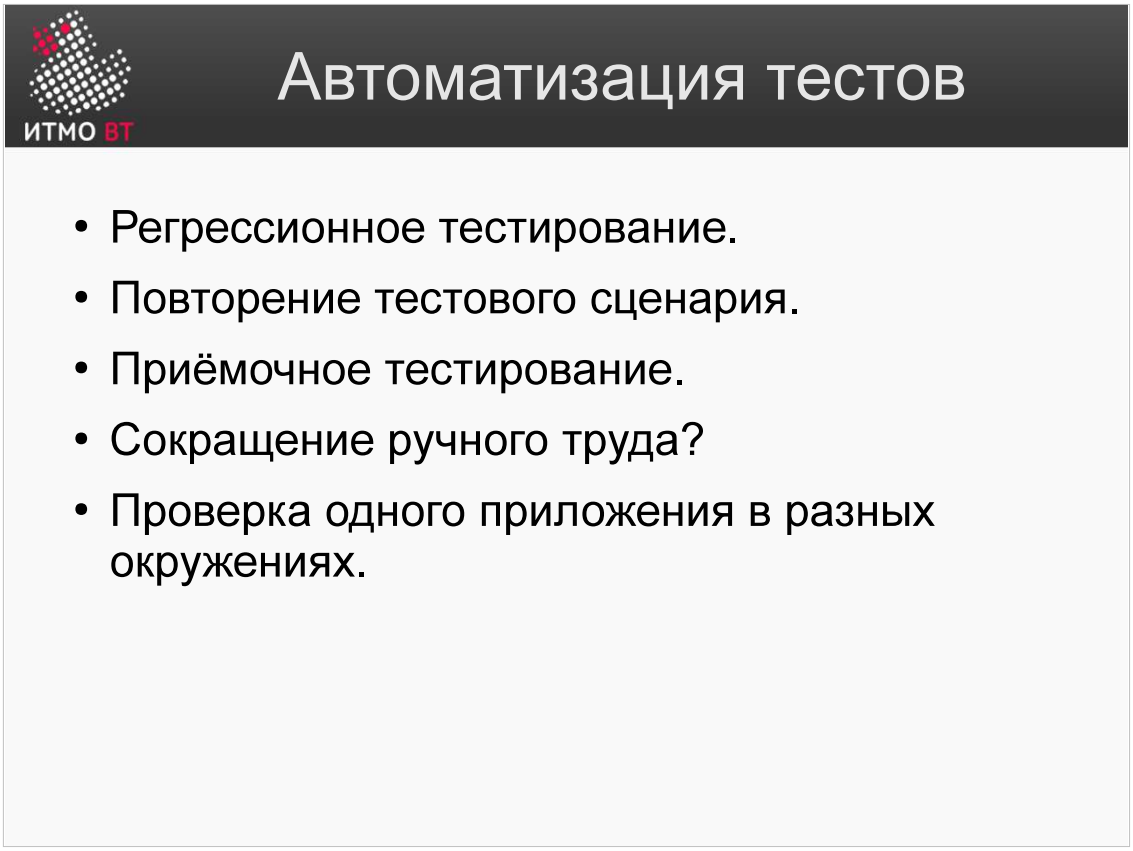
Валидация в русскоязычной литературе по тестированию обычно называется *аттестацией*.



Статическое тестирование — это тестирование, не связанное с запуском набора тестов, разработанных для ПО. Оно включает в себя методики по рецензированию и инспекциям кода. В случае, когда на ранних этапах разработки кода еще не существует, статическое тестирование подразумевает проверку спецификаций, архитектурных принципов, требований и т.п. Статическое тестирование способствует раннему нахождению дефектов, что экономит время и средства. На рынке разработки существует мнение, что, по мере перехода от одной фазы разработки к другой, стоимость исправления дефекта растет с фактором 10.

Динамическое тестирование требует уже созданной программной архитектуры (осуществляется сборка и запуск модулей, групп модулей или всей системы).

Некоторое время назад был предложен подход Test-Driven Development (TDD), в котором тесты разрабатывают перед разработкой кода. При этом на основе требований определяется тестовое покрытие, и разрабатываются тесты, которые первоначально все выполняются со сбоями. По мере разработки кода отчёт о проведении тестирования начинает "зеленеть". Разработка считается завершённой, когда все тесты выполняются успешно.

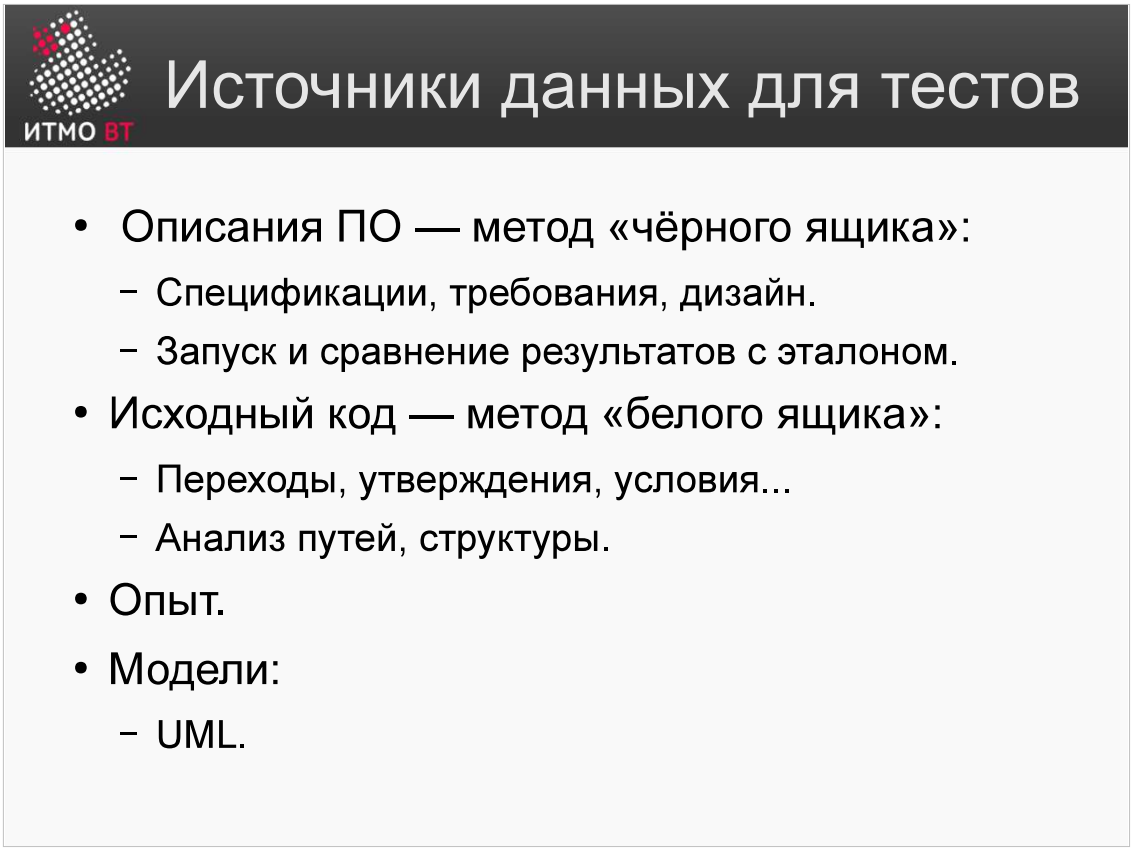


Автоматизация тестов является положительным явлением, но часто ручное тестирование оказывается дешевле и проще, особенно при простых задачах, где легче нанять низкоквалифицированный персонал, чем написать сложную программу, формирующую сценарии тестирования и меняющую эти сценарии при изменении функционала.

Отдельный вид тестирования — это т.н. регрессионное тестирование, которое заключается в том, что при изменении программы запускаются старые тесты. Это позволяет проверить, не повлияли ли внесённые изменения на работу всей программы и не появились ли нарушения в этой работе.

Чтобы автоматизировать тесты и пользоваться регрессионным тестированием, необходимо обеспечить однозначное повторение тестового сценария (внутри ПО не должно быть вариативности в поведении, одинаковые входные данные должны порождать одинаковые выходные данные).

Важной является проверка одного и того же приложения в разных окружениях (т.н. тестирование совместимости). Необходимость повторять тесты в различных окружениях порождает большую сложность всего процесса. Необходимо выполнить много тестов, поэтому чем меньше будет сформировано изначальных функциональных тестов, тем меньше общая сложность и длительность выполнения таких тестов.



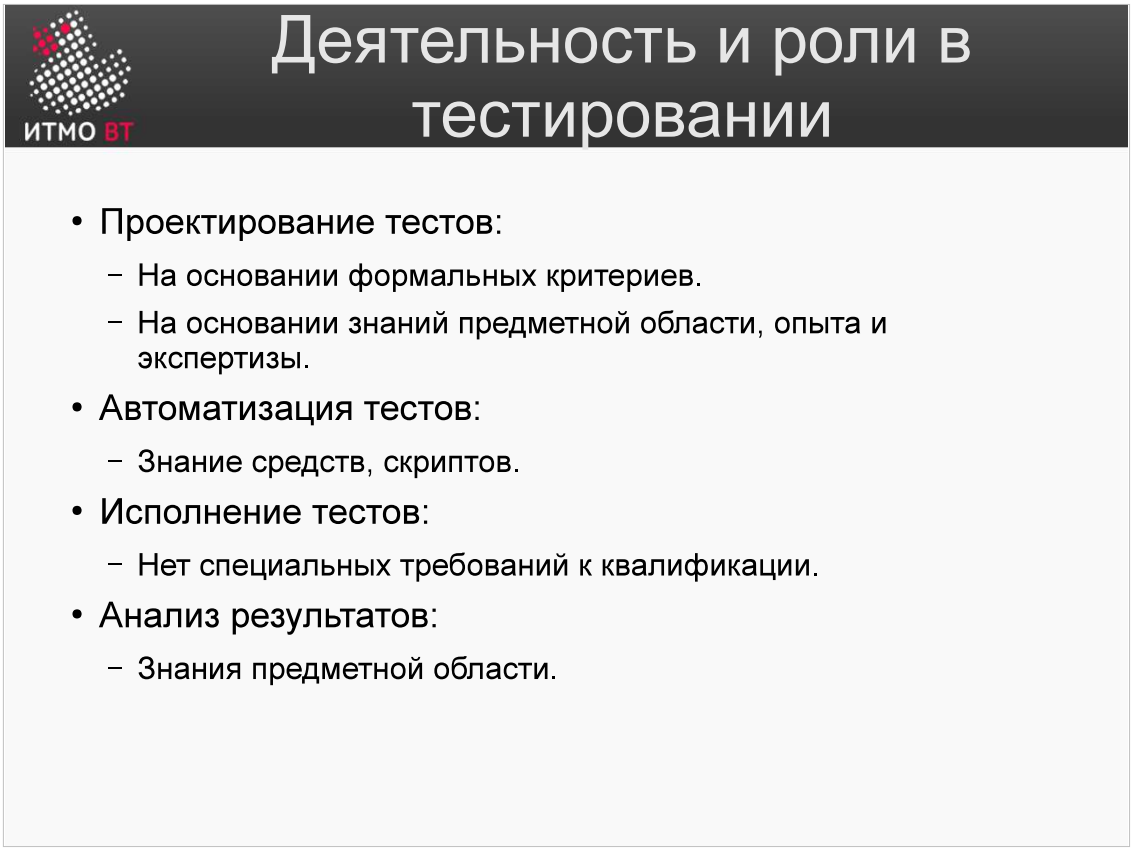
Существует два классических подхода к тестированию: метод «чёрного ящика» и метод «белого ящика». В методе «*чёрного ящика*» подразумевается, что внутреннее содержимое программы скрыто. Работа при таком тестировании идет на уровне спецификации или на основе опыта составителя тестов. Тесты подают на вход программы исходные данные и сравнивают результат с известным эталоном. Источниками данных для тестов являются спецификации, требования и дизайн.

В методе «*белого ящика*» возможно исследовать исходный код. Кроме возможности аналитически проверить корректность кода, данный метод делает возможным оценить размер тестового покрытия. Для этого из приложения строится граф, где части кода представляются как узлы графа, ветвления и циклы — как переходы. После этого определяется т.н. цикломатическая сложность программы, которая определяет число путей обхода существующего кода программы. Цикломатическая сложность указывает максимально необходимое число тестов, которые нужно выполнить для полной проверки программы. Конкретные тесты выбираются так, чтобы покрыть все пути полученного графа.

Обычно методы проведения тестов могут быть однозначно отнесены к определенной категории, некоторые же сочетают в себе сразу несколько категорий.

Дополнительным источником данных для тестов является опыт разработчика. Опыт позволяет решать стандартные ситуации типовыми методами, что минимизирует вероятность ошибки.

Источником данных для тестов могут являться модели, в том числе UML-диаграммы и их описание. Из описания UML-диаграммы выбираются основные и альтернативные пути, а затем выбираются конкретные данные для тестирования путей.



Роли и деятельности в тестировании делятся на несколько больших категорий:

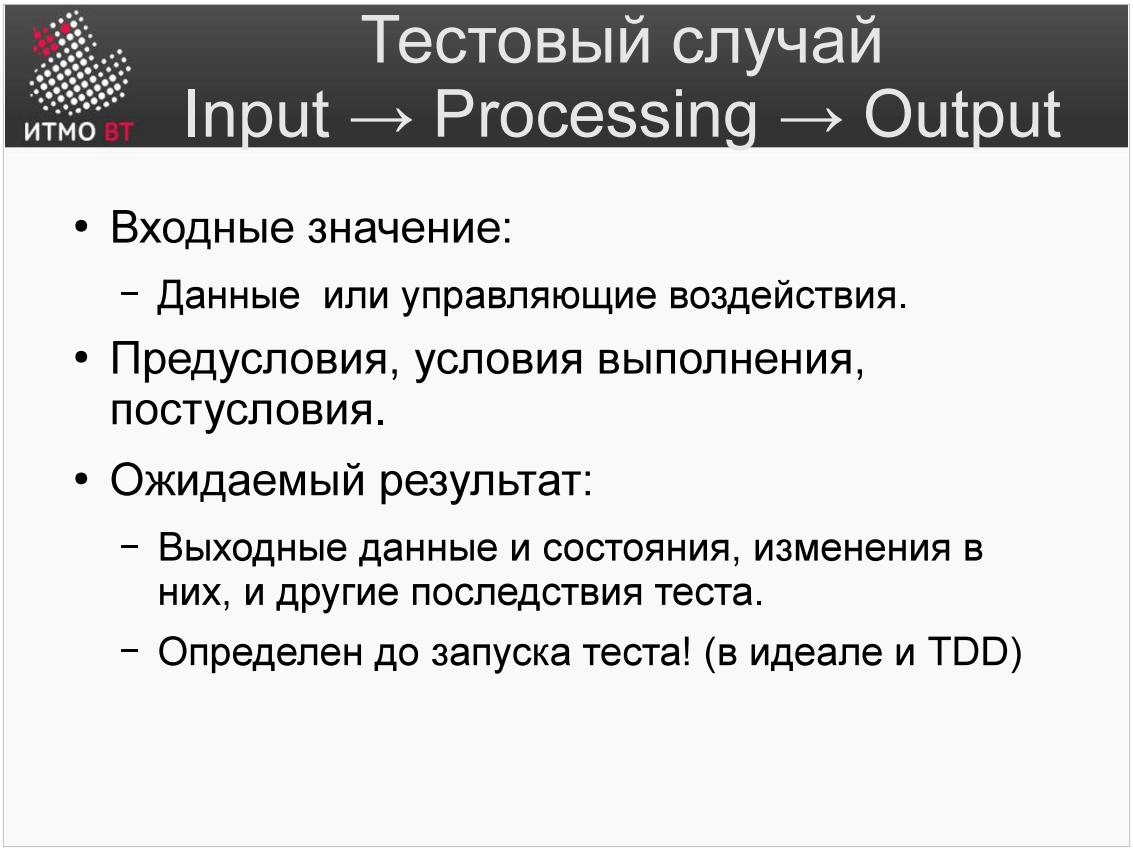
Проектирование тестов. Квалификация специалистов, проектирующих тесты, должна быть достаточно высокой и включать в себя знание предметной области, особенностей пользовательского интерфейса, а также дискретной математики, навыков программирования и тестирования.

Вторая деятельность — *автоматизация тестов.* Обычно этим занимаются программисты, специализирующиеся на разработке тестов скриптов или программ. Принято, что модульные тесты пишут сами разработчики исходного кода, а интеграционные и системные тесты создают отдельно выделенные программисты с глубокими представлениями о архитектуре и взаимодействии различных частей приложения.

Непосредственное исполнение тестов не требует высокой квалификации персонала. Тем не менее, этот персонал должен знать тестовую инфраструктуру, принципы работы с тестируемым приложением и обладать элементарными навыками работы с компьютером.

Для *анализа результатов тестов* требуются как знания в предметной области, так и технические знания. Поэтому для данной деятельности необходима достаточно высокая квалификация персонала.

Всё вышеперечисленное говорит о том, что для организации и проведения полноценного тестирования нужна квалифицированная подготовка, и ко многим участникам процесса тестирования программы предъявляются более высокие требования, чем к её рядовым разработчикам.



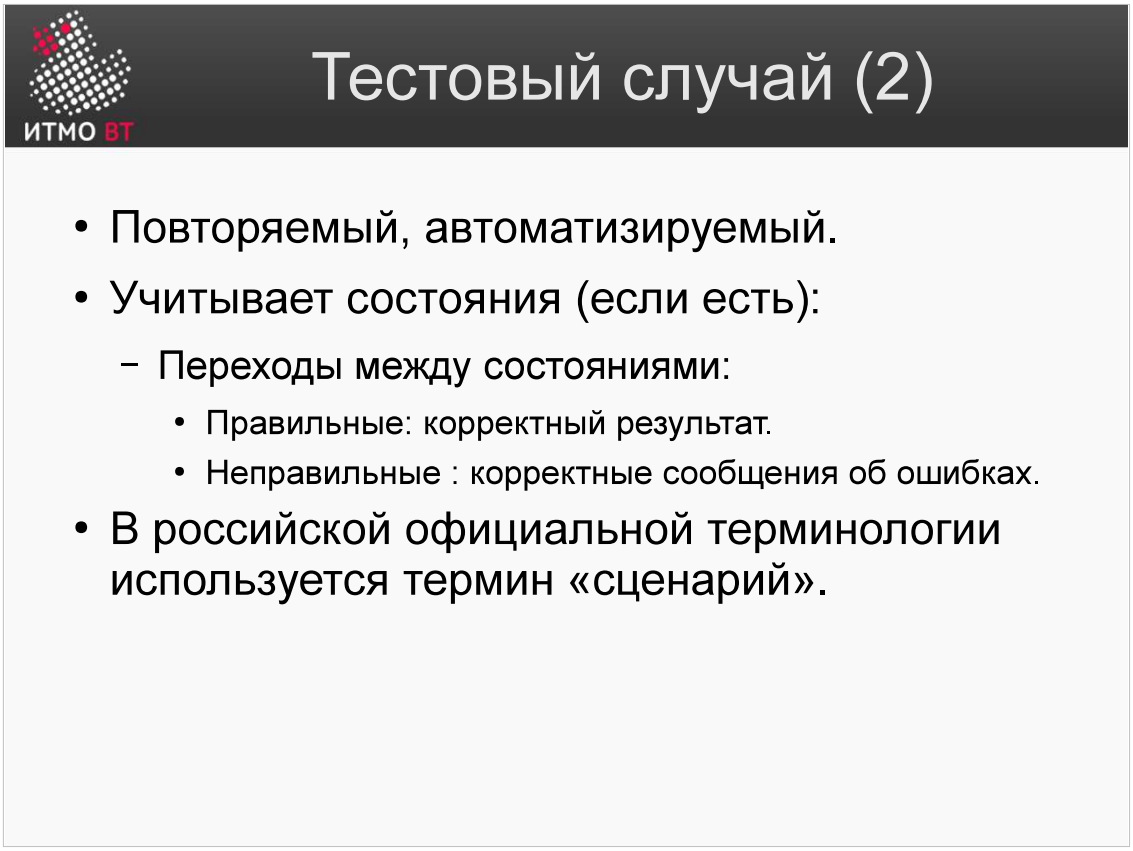
Тестовый случай состоит из набора входных значений, предусловий выполнения, ожидаемых результатов и постусловий.

Набор входных значений представляет собой набор данных, на которых разрабатываемое ПО должно вести себя определённым образом с точки зрения спецификации требований к ПО или других источников информации о поведении программы. Например, если проверяется функция sin(x) то в качестве входных значений могут быть использованы 0, 2π, 0.1, $\pi/3$ и другие, которые соответствуют определённым значениям функции. Важно отметить, что набор этих значений должен быть достаточным для того, чтобы покрыть большинство важных особенностей функции. В случае синуса, одной из таких особенностей может быть, например, периодичность.

Содержание спецификаций проектирования тестов (включая тестовые условия) и спецификаций тестовых сценариев описывается в стандарте «Документация при тестировании программ» (IEEE STD 829-1998).

Ожидаемые результаты должны создаваться как часть спецификаций тестовых сценариев и включать в себя выходные данные, изменения в данных и состояниях, и любые иные последствия теста. Если ожидаемые результаты не были определены, правдоподобные, но ошибочные результаты могут быть приняты за корректные.

В идеальных условиях ожидаемые результаты должны быть определены до момента выполнения теста. Для отражения этой концепции был разработан подход Test Driven Development — разработка через тестирование. При этом подходе разработчик сначала описывает тестовое покрытие и разрабатывает тесты для этого покрытия. После этого он начинает разрабатывать собственно программное обеспечение, и с каждым его запуском растет количество тестов, выполненных успешно. Разработка считается завершённой, когда будут успешно выполнены все тесты.



Тестовый случай должен быть повторяемым, т.е., одинаковый набор входных значений и состояний должен приводить к одинаковым выходным значениям или состояниям.

Для повторяемости теста желательно автоматизировать его проведение. Создание автоматизированных тестов особенно важно для регрессионного тестирования. Это позволит при внесении изменений проводить уже написанные тесты и проверять, не привели ли новые изменения к появлению дефектов.

В тестовом случае должны учитываться состояния внутри ПО (если они есть) и переходы между ними. Наличие состояний обычно приводит к расширению тестового покрытия.

Следует отметить, что необходимо протестировать как «нормальные» пользовательские сценарии, так и сценарии, приводящие к появлению сообщений об ошибках.

В российской практике существует определённая путаница в терминах «тестовый случай» и «тестовый сценарий». Они часто заменяются один другим. Когда разработчики говорят о сценарии, часто имеют в виду единичный тестовый случай, и наоборот.

№	Начальное состояние	Ввод	Действие системы	Вывод	Конечное состояние
1	Готов	Пользователь вставляет карточку	Успешное чтение карточки	Приглашение “введите pin”	Ожидание pin-кода
2	Ожидание pin-кода	Вводим **верный** pin-код	Проверка pin-кода	Приглашение к выбору транзакции	Ожидание выбора транзацкии
3	Ожидание выбора транзацкии	Выбор выдачи 5000 рублей	Проверка баланса, возможности выдачи	Деньги	Выдача денег
4	Выдача денег	Пользователь берет деньги и карточку	Завершение выдачи	Благодарность за использование	Готов

Тестовый сценарий — это последовательность тестовых случаев.

В примере на слайде приведён сценарий снятия денежных средств с банкомата. Данный сценарий состоит из 4 тестовых случаев. В начале сценария банкомат находится в состоянии готовности к приёму карточки. После того, как пользователь вставляет карточку в банкомат, банкомат должен проверить её, и, если карточка читается, выдать приглашение для ввода пин-кода. Состояние банкомата при этом меняется на ожидание ввода. После этого тестовый случай предполагает, что пользователь вводит верный пин-код, осуществляется проверка этого пин-кода и показывается экран, который предлагает совершить ту, или иную транзакцию.

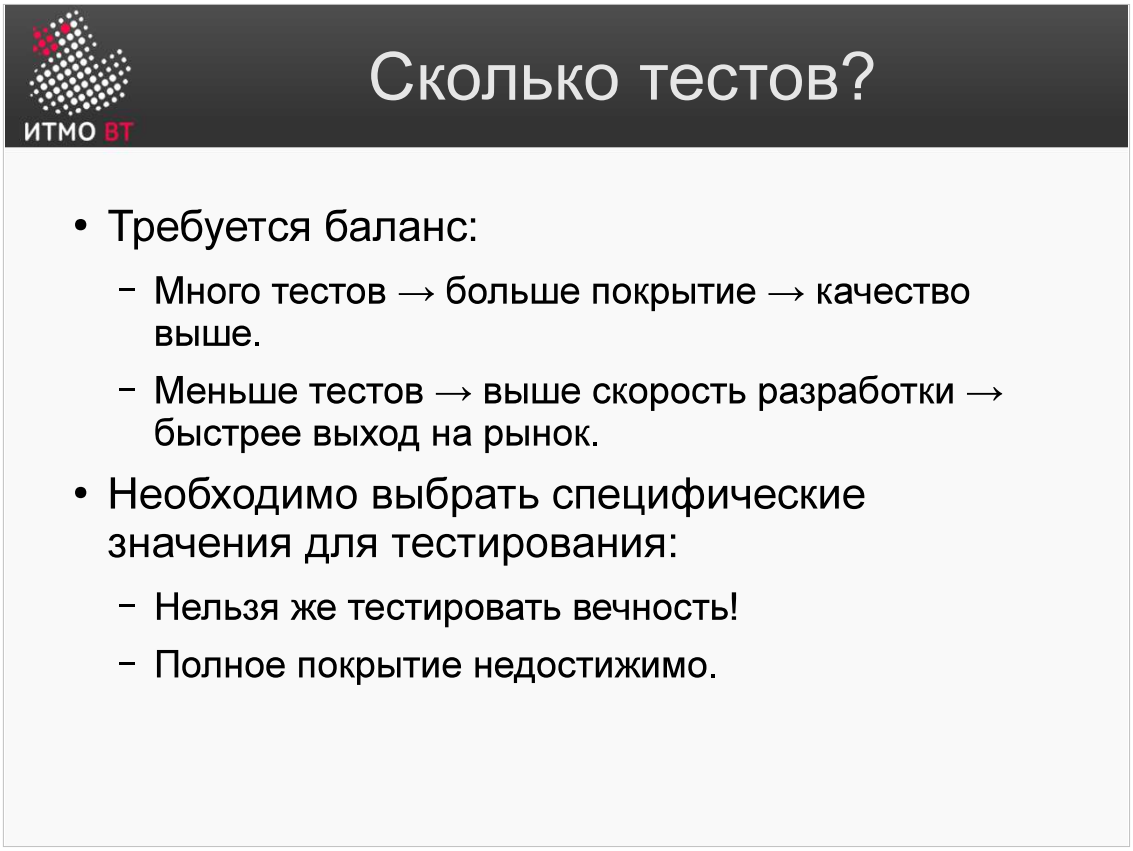
Далее пользователь выбирает необходимую операцию. В данном случае подразумевается, что эта операция — снятие 5000 руб. Правильный тестовый сценарий подразумевает, что необходимая сумма имеется на счёте пользователя. Результатом выполнения этого тестового случая будет выдача банкоматом необходимого количества денежных средств. После выдачи денег, банкомат должен вернуться в исходное состояние готовности приёма карточки.

Ещё раз подчеркнем, что каждая строка в примере на слайде соответствует отдельному тестовому случаю. Набор тестовых сценариев, в котором варьируются действия пользователя и, соответственно, создаются разные тестовые случаи на ввод различной информации, должен приводить к формированию необходимого тестового покрытия для требуемого набора функций системы.

№	Начальное состояние	Ввод	Действие системы	Вывод	Конечное состояние
1	Готов	Пользователь вставляет карточку	Успешное чтение карточки	Приглашение “введите pin”	Ожидание pin-кода
2	Ожидание pin-кода	Вводим **неверный** pin-код	Проверка pin-кода	Сообщение об ошибке	Ожидание pin-кода
3	Ожидание pin-кода	Вводим **неверный** pin-код	Проверка pin-кода	Сообщение об ошибке	Ожидание pin-кода
4	Ожидание pin-кода	Вводим **неверный!!!** pin-код	Проверка pin-кода, блокировка карточки	Сообщение об ошибке и блокировка	Готов

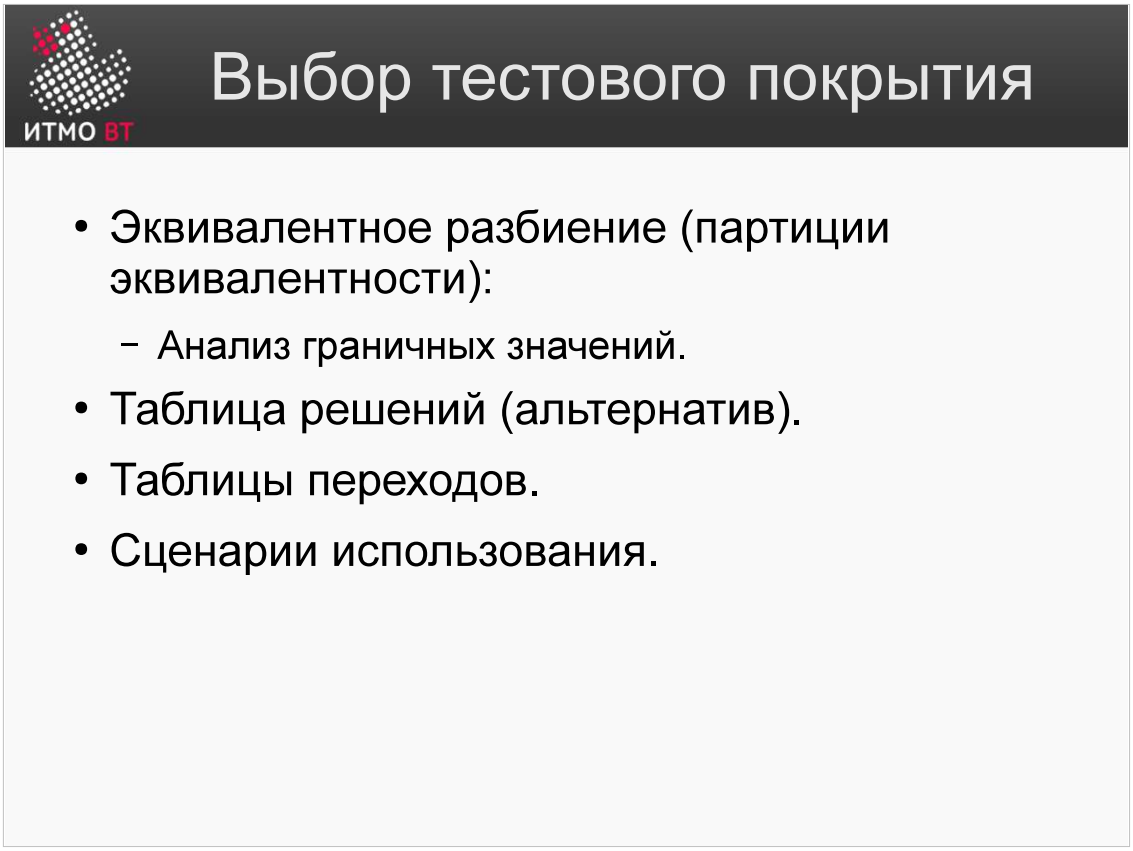
В тестовых сценариях должно быть предусмотрено появление как положительной, так и отрицательной реакции системы на действия пользователей. Для неправильных действий пользователя также создаются тестовые сценарии.

На приведённом на слайде примере тестовый сценарий проверяет неправильный ввод пин-кода пользователем три раза подряд. Должны быть проверены тестовые случаи неправильного ввода пин-кода на каждом шаге, а затем итоговая блокировка карточки в конце сценария. Банкомат после выполнения всего сценария должен оказаться в исходном состоянии.



При планировании тестирования следует соблюдать баланс между качеством и скоростью вывода продукта в эксплуатацию. Если тестов много, то покрытие растёт, и, как следствие, растёт и качество. Однако также растут затрачиваемое время и ресурсы. Может сложиться ситуация, при которой конкуренты, тестировавшие меньше и быстрее, захватят рынок сбыта, и долго тестируемая программа станет никому не нужной, несмотря на то, что она лучше продукта конкурентов.

Количество тестовых сценариев необходимо выбирать с учетом того, что полное тестовое покрытие недостижимо. Для осуществления этого существует целый набор методов, которые позволяют определить достаточное тестовое покрытие.



Методы, позволяющие определить достаточное тестовое покрытие, представлены на слайде. Наиболее известным из них является метод эквивалентного разбиения, который иногда ещё называют партициями эквивалентности. В рамках него также производится анализ граничных значений, внутри которых тестируемая функция ведет себя одинаково.

Другим методом является таблица альтернативных решений. Составляется таблица, отражающая комбинации входных данных (и/или причин) с соответствующими выходными данными (и/или действиями/следствиями), которая может быть использована для проектирования тестовых сценариев, которые необходимо провести для группы таких сочетаний.

В другом методе, основанном на таблице переходов, выделяются явные состояния внутри системы, определяются переходы между этими состояниями, которые далее покрываются тестами.

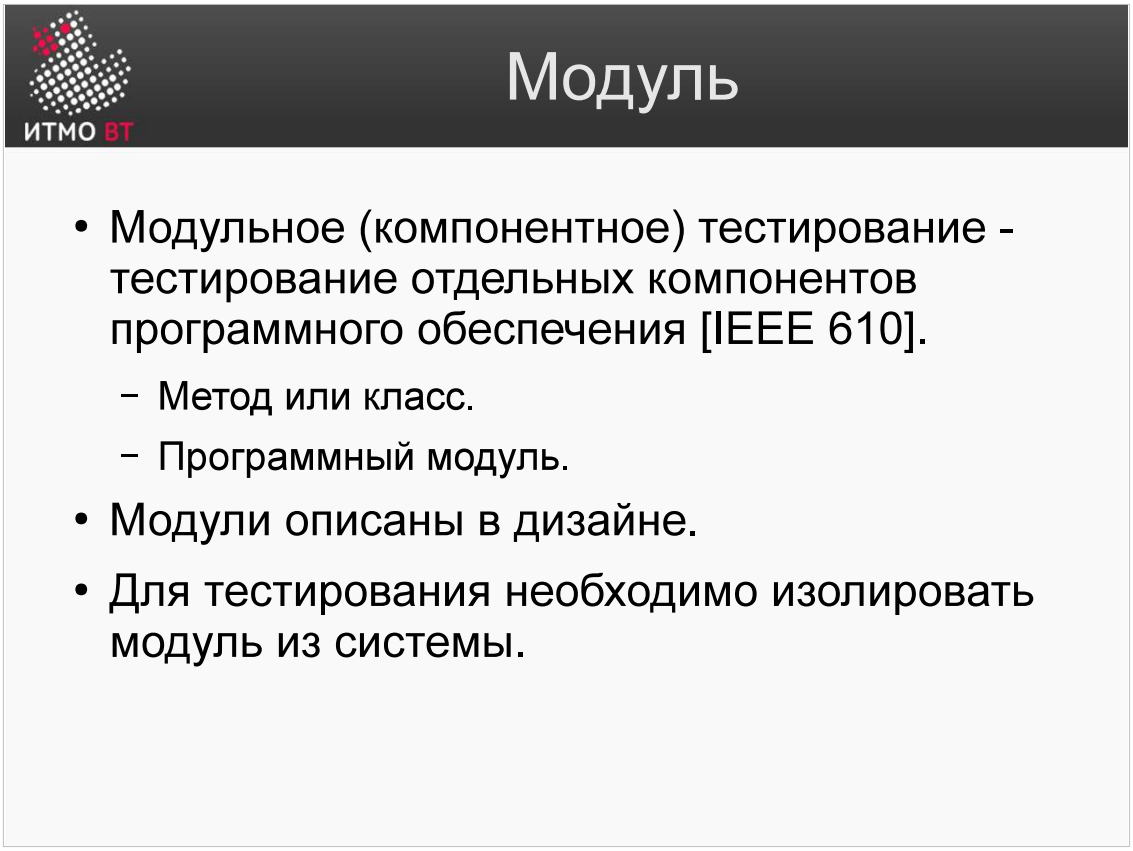
Разработанные во время определения требований сценарии использования системы могут служить источником данных для формирования тестового покрытия. При этом в каждый описанный сценарий добавляются конкретные значения, вводимые пользователем. Необходимо учитывать как основные, так и альтернативные пути сценария. Обычно каждому такому пользовательскому сценарию соответствует целая группа тестовых сценариев.

При анализе эквивалентности тестируемая функция или модуль разбивается на участки, где программа ведет себя одинаково (эквивалентно). Внутри каждого участка формируется свой набор тестовых случаев. Если таких участков относительно немного, это позволяет резко сократить количество тестовых случаев. Отдельные тесты составляются для граничных значений участков.

На слайде приведена зависимость удовлетворения человека от количества холодца, которое он съел. Если человек ещё не съел холодца, то его удовлетворение равно нулю. После того, как он начал его есть, удовлетворение линейно растет. Это продолжается до тех пор, пока не приходит насыщение. При насыщении скорость роста удовлетворения начинает падать, а после прохождения определенной точки начинает резко уменьшаться вследствие переедания.

Очевидно, что данный график функции можно разбить на несколько участков, которые представлены на графике разноцветными прямоугольниками. Внутри каждого участка функция ведет себя одинаково (точнее, мы можем считать, что функция ведет себя одинаково). Исходя из этого, мы можем выбрать всего лишь несколько значений на каждом участке для проведения тестирования.

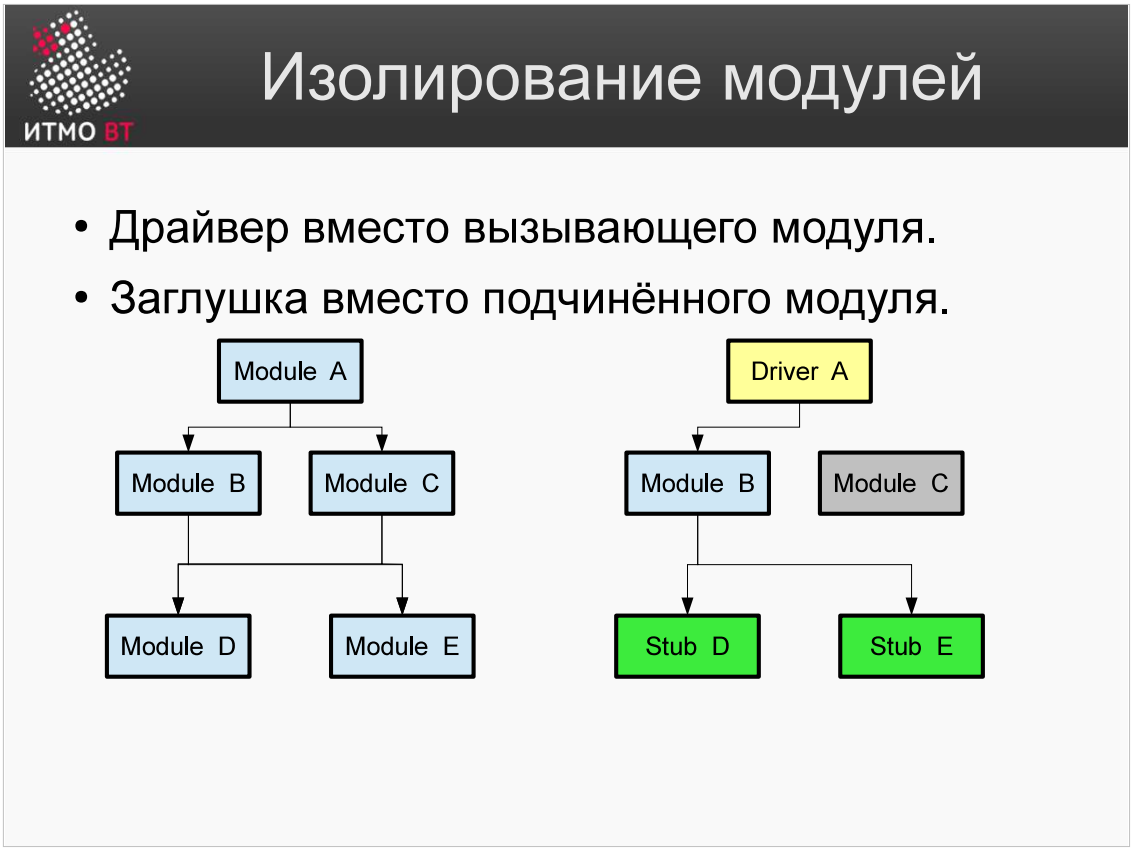
Точками на графике представлены границы каждого участка. Для формирования тестового покрытия эти точки должны быть протестированы отдельно. Количество тестов внутри каждого интервала определяется аналитиком, и обычно выбирается исходя из рациональных побуждений. В данном примере достаточно 3-5 точек для формирования тестового покрытия.



В классической V-образной модели, применяемой в тестировании (см. первую лекцию), определено несколько стадий последовательного тестирования разрабатываемого приложения. После разработки кода сперва идет модульное тестирование, затем интеграционное тестирование, а после — тестирование системы целиком, конечной фазой которого является приёмочное тестирование.

Модуль — это компонент, который необходимо протестировать отдельно от остального программного продукта. Модуль выполняет некую законченную функцию. В разных языках программирования модуль может быть методом, классом, или совокупностью методов и классов, формирующей программный модуль.

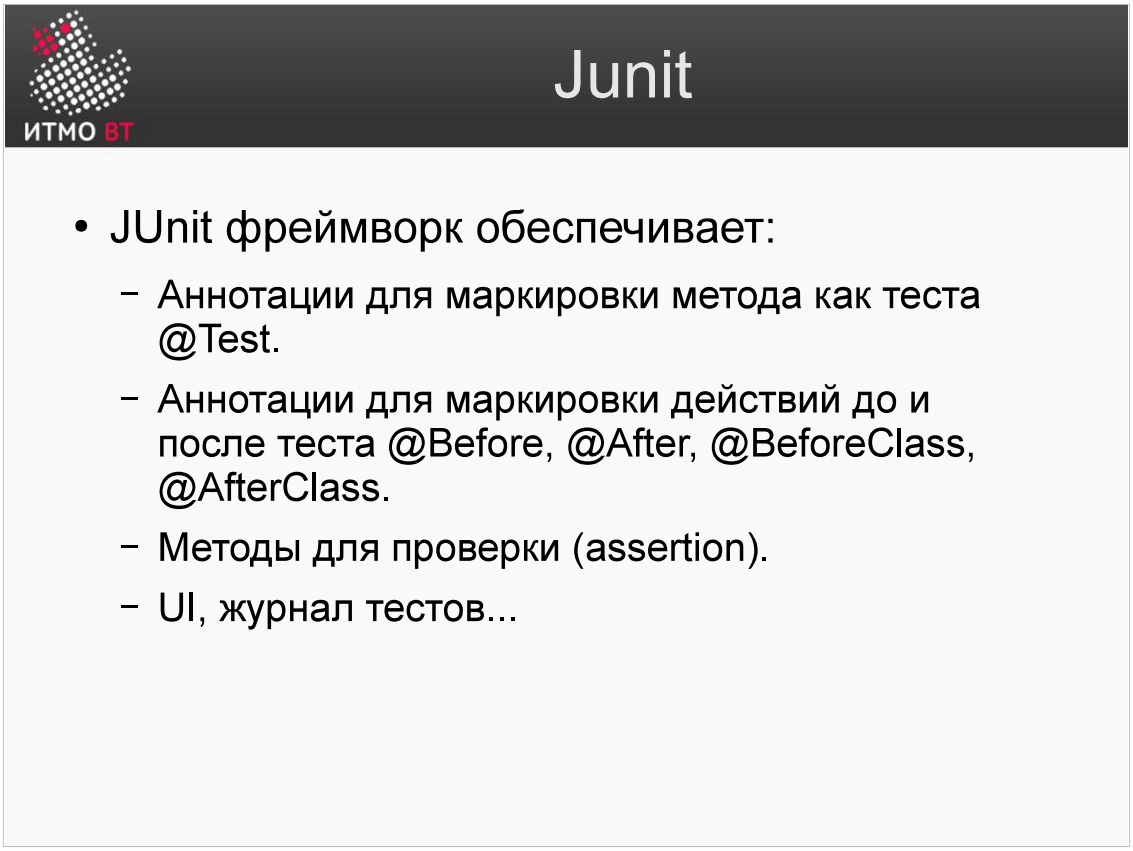
Модули определены в дизайне программы. Когда проектировщик разрабатывает архитектуру системы, он в явном виде делит её на сначала на слои, а затем на модули. Для проведения модульного тестирования, модуль необходимо изолировать из системы.



Изолирование модулей производится ради исключения сторонних воздействий. Программа, представленная на слайде, состоит из 5 модулей. При этом главным из них является модуль верхнего уровня, который осуществляет вызовы всех остальных. Модули верхнего уровня зависят от модулей, которые расположены ниже по иерархии вызовов. Изолирование модулей предполагает замену модулей, осуществляющих вызов *драйверами* (т.е. управляющими тестированием), а подчинённых модулей — *заглушками*. Иногда для этого требуется отдельная тестовая сборка приложения, в которой присутствует код только для одного модуля. При этом количество таких сборок обычно равно количество модулей в системе.

Драйвер — компонент, вызывающий модули и обеспечивающий последовательность тестирования. Драйвер должен последовательно вызывать тестируемый модуль с различными входными параметрами и условиями.

Заглушка ведет себя подобно подчинённому модулю, имеет тот же интерфейс, но гораздо более простую реализацию. При вызове заглушка возвращает заранее определённые значения. Однако при вызове заглушки не предусматривается подача входных значений, отличающихся от предусмотренных в заглушке. Часто для создания заглушек используется табличный метод с соотнесением входных параметров с возвращаемыми заглушкой. Такой подход хорош своей наглядностью.



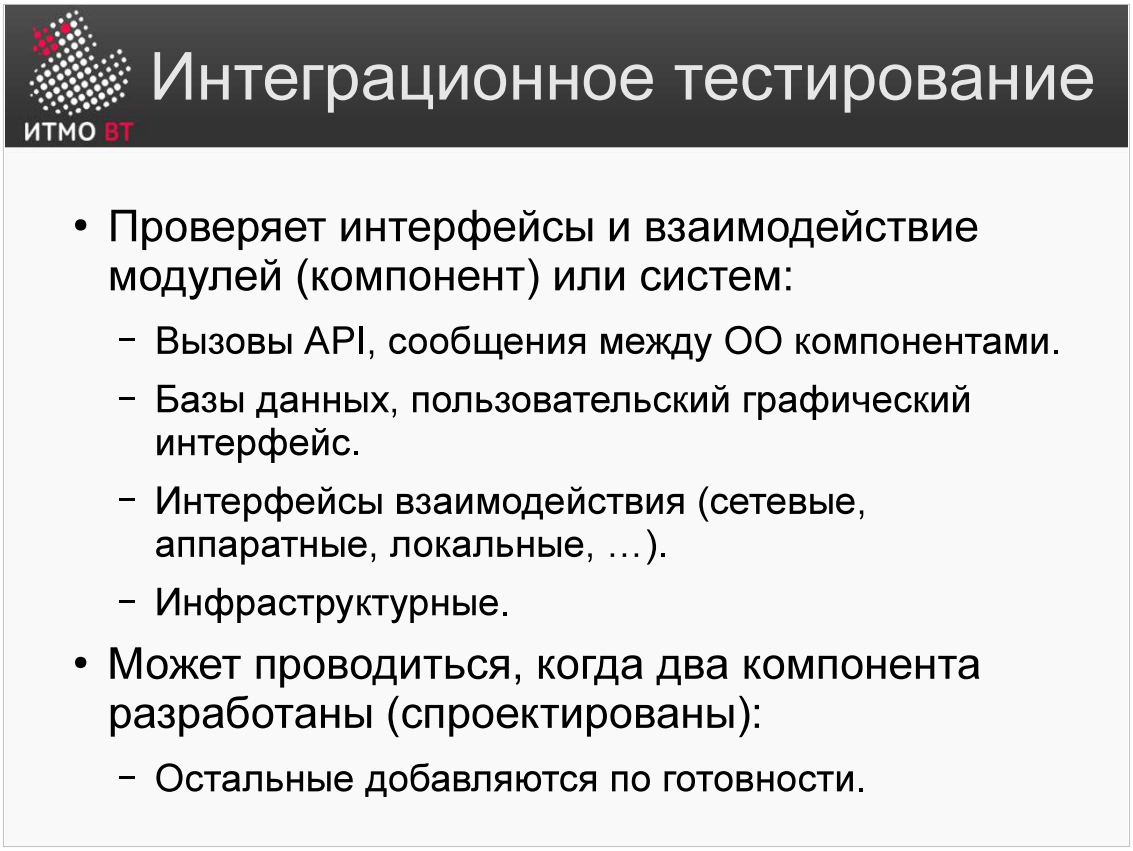
Для модульного тестирования в Java используется JUnit. JUnit — это простейший фреймворк, позволяющий создать модульные тесты и выполнить их в определённом окружении. У других языков программирования существуют средства, подобные JUnit, имеющие те же возможности формирования автоматического запуска тестов, которые будут вызывать части приложения с определёнными аргументами.

JUnit4 построен на аннотациях. Метод, который организует тестирование, помечается аннотацией @Test. При этом фреймворк тестирования последовательно просматривает загружаемые классы (при помощи reflection API) и ищет в них эту аннотацию. Все методы с найденной аннотацией запускаются (возможно, в разных потоках).

Внутри тестового метода проверяется тестовое покрытие на соответствие определённым условиям при помощи специальных функций проверки, которые называются assertion. В JUnit4 существует большой набор таких функций, которые могут проверять на равенство, совпадения, появление исключительной ситуации и так далее. Результаты тестов заносятся в специальный журнал для последующего анализа.

Для организации тестового окружения существуют аннотации, позволяющие выполнить код перед всеми тестами, после всех тестов, а также при загрузке тестового класса в память и выгрузки его из неё.

Последовательность тестов в JUnit не регламентирована (порядок можно искусственно определить, но по умолчанию считается, что все тесты выполняются параллельно и независимо). JUnit самостоятельно определяет, каким образом будет запущено тестирование.

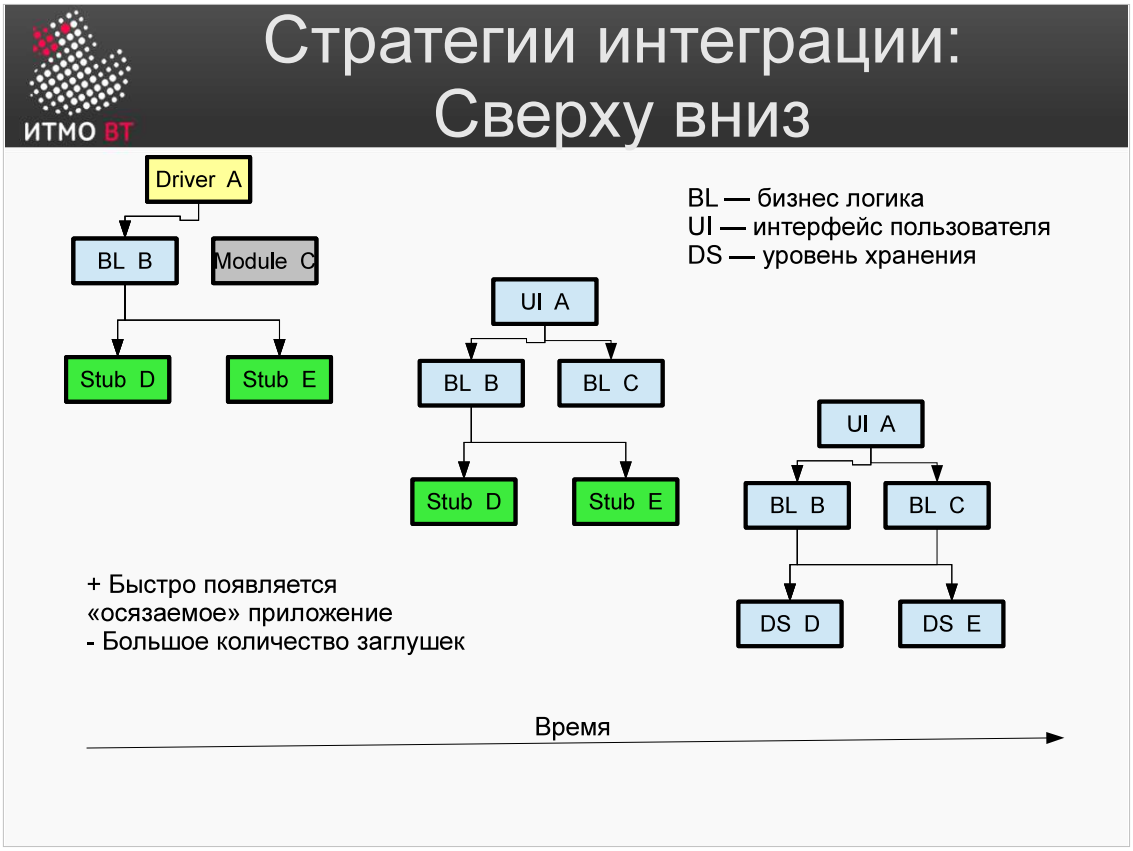


Компонентное интеграционное тестирование проверяет взаимодействие между программными компонентами и производится после модульного тестирования. По используемым средствам и подходам интеграционное тестирование очень похоже на модульное, однако проверке подлежат не функции отдельного модуля, а межмодульное взаимодействие. Смысл тестирования состоит в проверке интерфейсов взаимодействия модулей на правильную последовательность вызовов и соответствия протоколов такого взаимодействия требованиям спецификации. На слайде приведено несколько примеров интеграционного тестирования между различными частями разрабатываемого программного обеспечения.

Системное интеграционное тестирование проверяет взаимодействие между программными системами или между аппаратным обеспечением и может быть выполнено после системного тестирования. В некоторых случая, разработчики могут управлять только одной стороной интерфейса, что вносит дополнительные риски в процесс тестирования. Современные бизнес-процессы могут включать последовательность выполнения разных частей ПО на отдельных системах, в которых могут быть важны кроссплатформенные различия.

Интеграционное тестирование может проводиться, когда два компонента уже разработаны (спроектированы). Остальные компоненты добавляются по мере их реализации, при этом важна последовательность интеграции модулей, которая определяется *стратегией интеграции*. Выбор конкретной стратегии интеграции зависит от типа разрабатываемого программного обеспечения, а также общего плана разработки ПО.

При составлении плана разработки ПО необходимо учитывать будущую стратегию интеграции и подготовить, в соответствии с ней, последовательность разработки модулей, с учетом сложности, общей трудоёмкости и календарного плана.

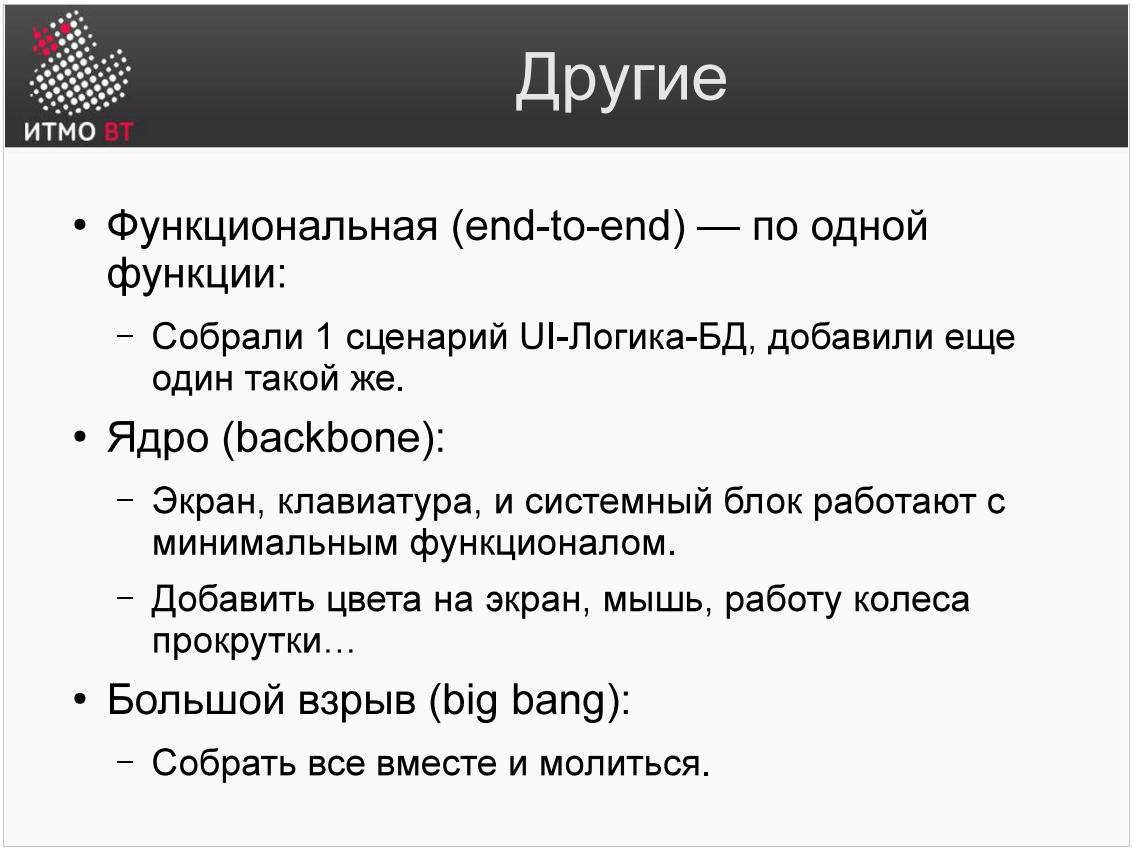


Стратегия "сверху вниз" — самая распространённая из стратегий. Она применяется, в основном, для бизнес-приложений. Сначала проверяется бизнес-логика с использованием драйвера и заглушек. После этого подключается интерфейс пользователя (UI), который выполняет реальные запросы к блоку кода бизнес-логики. Последними подключаются блоки хранения данных (блоки нижнего уровня) и проводятся тесты как модульного, так и интеграционного тестирования.

Данная стратегия позволяет быстро продемонстрировать приложение конечному заказчику, однако ведёт к разработке большого количества заглушек, следствием чего может являться разработка избыточного количества дополнительного программного обеспечения, которое участвует только в тестировании.

Стратегия "снизу вверх" чаще всего используется тогда, когда приложение сильно связано с аппаратурой. Вследствие этого интеграция начинается с блоков нижнего уровня, наиболее тесно взаимодействующих с аппаратурой.

Следует отметить, что цикл разработки аппаратуры обычно более длительный и трудоёмкий. Тестирование остальных частей приложения, использующих отдельное аппаратное обеспечение, может начаться только после того, как будут готовы первые прототипы. Компании, разрабатывающие аппаратуру, обычно подготавливают тестовые версии своих продуктов и предоставляют их смежным компаниям ещё до их широкого запуска в производство.

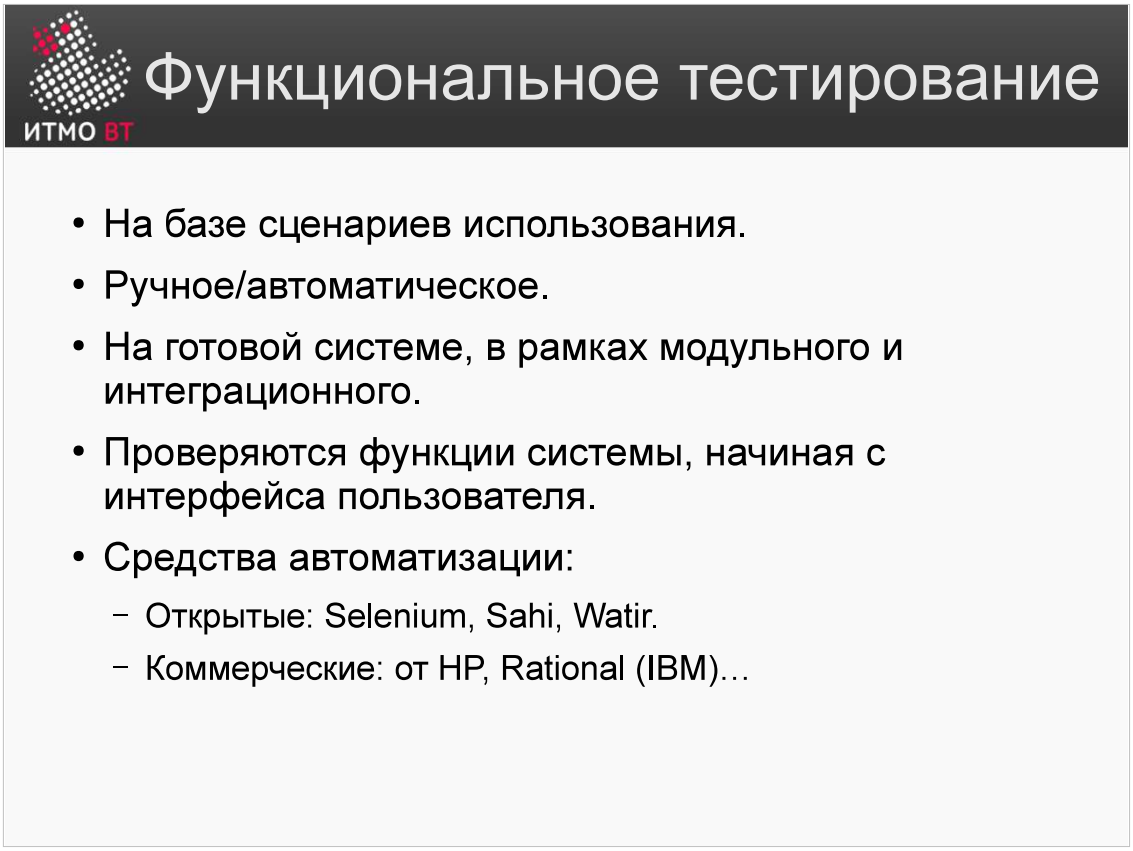


Другие возможные стратегии интеграции представлены на слайде.

Функциональная стратегия (end-to-end) подразумевает постепенное наращивание приложения по функциям. При её использовании сначала производится сборка, отладка и тестирование первого пользовательского сценария "UI-Логика-БД", затем подключаются второй, третий, и так далее, до тех пор, пока приложение не будет собрано полностью.

При стратегии сборки на основе ядра (backbone) сначала формируется минимальный работоспособный функционал, а потом добавляются остальные функции, расширяя возможности приложения.

«Большой взрыв» (big bang) — самая примитивная стратегия. В ней всё собирается одновременно, поэтому в случае возникновения ошибок неясно, где именно они возникают, и что их вызвало.



Функциональное тестирование обычно рассматривается как разновидность интеграционного.

В функциональном тестировании проверяются функционал, заложенный в программу. Такое тестирование осуществляется на базе сценариев использования, где в явном виде описаны действия пользователя в системе. Следует отметить, что обычно проверяются бизнес-процессы целиком, при этом они могут включать в себя использование различных ролей пользователей, последовательно выполняющих отдельные элементы бизнес-процесса.

Основным элементом управления при функциональном тестировании является графический интерфейс пользователя. Функциональные тесты могут выполняться как тестировщиками вручную, так и при помощи автоматических средств взаимодействия с интерфейсом. Разработка тестируемого функционала при этом должна быть полностью завершена на всех уровнях приложения.

Полностью избежать ручного тестирования при функциональном тестировании, к сожалению, невозможно. Особенно явно это проявляется при изменении функциональности уже готового приложения с разработанными тестами. Например, приложение, осуществляющие ввод и обработку персональных данных клиента, может потребовать добавление нового поля. При этом разработанные ранее тесты станут завершаться неуспешно. Тестировщик в результате тестирования вручную выявит данный дефект и сформирует запрос на изменение тестов.

Для функционального тестирования разработано большое число средств автоматизации, которые берут на себя управлением интерфейсом пользователя, включая ввод с клавиатуры, перемещение указателя, щелчки мышью и пр. Примеры данных средств приведены на слайде. Одной из наиболее распространённых утилит является дополнение к браузеру Firefox — Selenium. Оно позволяет сначала записать тестовую последовательность использования интерфейса, а затем сохранить ее в виде тестовой программы, которая может исполняться и в других браузерах.

Статическое тестирование (рецензирование) — вид тестирования ПО (включая код), который может проводиться перед динамическим тестированием. У статического анализа и динамического тестирования общая цель — обнаружение дефектов. Методы дополняют друг друга, так как с разной эффективностью находят различные типы дефектов. В отличие от динамического тестирования, статические методы находят причины сбоя (дефекты), а не сами отказы.

Рецензирование может проводиться вручную или с помощью специальных программных средств. Главной составляющей ручного процесса является исследование и комментирование программного продукта.

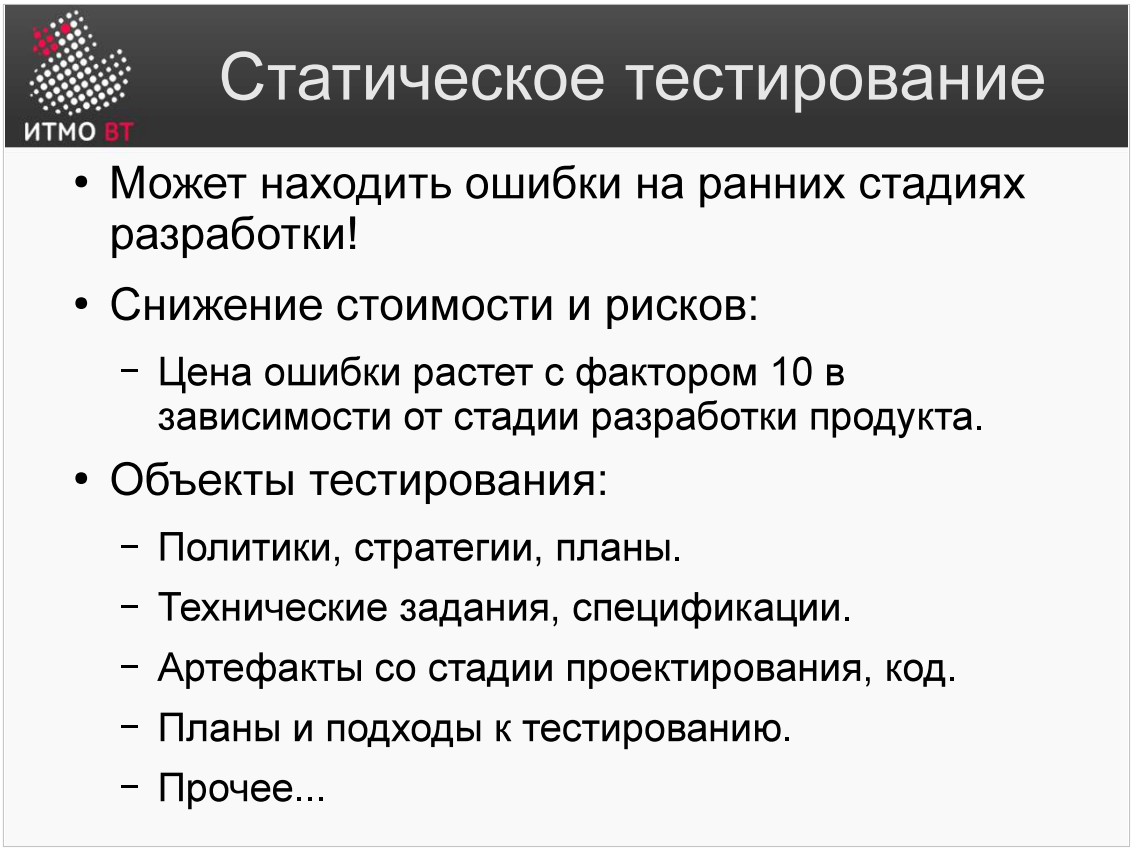
Одной из распространённых техник является *неформальное ревью* (рецензия) коллегой. Это простой способ, когда вы просите оценить коллегу принятые решения и выбранные алгоритмы в вашем коде или спецификациях. Во время работы над задачей глаза разработчика часто "замыливаются" и не замечают очевидных для других дефектов. Поэтому существует вероятность, что ваши дефекты могут быть быстро найдены. Однако, на практике, это не всегда хорошо работает, потому что коллега может ошибаться и быть субъективным (например, давать некорректную оценку из-за недостатка времени).

В стандарте IEEE 1028-2008 существуют несколько полезных формальных техник статического тестирования. Они проводятся в виде отдельно организованного собрания под управлением различных ролей.

В *техническом анализе*, который иногда называют методом повторного просмотра, собрание проводится под управлением технического лидера проекта. В *management review* — под управлением менеджера разработки.

Методика *сквозного контроля* (walkthrough) организует просмотр решений специально выделенным экспертом, который "ведёт" аудиторию через рассматриваемый артефакт, фиксируя недочеты и дефекты и контролирующий процесс их исправления.

Инспекции (придуманы в IBM Майклом Фэганом), организованы вокруг идеи, что человек может контролировать не более двух параметров одновременно. Вследствие этого каждый инспектор обладает двумя ролями и формально контролирует только их.



Рецензирование может проводиться для любого продукта, связанного с разработкой ПО, включая спецификации требований и дизайна, код, планы тестирования, спецификации тестирования, тестовые сценарии, руководства пользователя, а также веб-страницы.

Исправление дефектов, обнаруженных во время рецензирования на ранних этапах жизненного цикла ПО (например, дефектов, найденных в требованиях), часто обходится значительно дешевле по сравнению с дефектами, найденными во время выполнения тестов и исполнения кода. Во время рецензирования могут быть найдены упущения, например, в требованиях, которые маловероятно найти во время динамического тестирования.

Итак, преимущества рецензирования следующие:

- раннее обнаружение и исправление дефектов;
- улучшение продуктивности разработки;
- уменьшение времени разработки;
- уменьшение времени и стоимости тестирования;
- сокращение стоимости жизненного цикла;
- уменьшение числа дефектов;
- улучшение коммуникаций между членами команды;
- передача информации между членами команды в образовательных целях.

Типичные дефекты, которые легче найти при рецензировании, чем при динамическом тестировании: отклонения от стандартов, дефекты в требованиях или дизайне, недостаточная пригодность к сопровождению и некорректные спецификации интерфейса.

Статические методы. Обзор

	Сквозной контроль	Технический Анализ	Инспекция
Основное Предназначение	Поиск дефектов	Поиск дефектов	Поиск дефектов
Дополнительная цель	Обмен опытом	Принятие решений	Улучшение процесса
Подготовка	Обычно нет	Популяризация	Формальная подготовка
Ведущий	Автор	В зависимости от обстоятельств	Подготовленный модератор
Рекомендованный размер группы	2-7	>3	3-6
Формальная процедура	Обычно нет	Иногда	Всегда
Объем материалов	небольшой	От среднего до большого	небольшой
Сбор метрик	Обычно нет	Иногда	Всегда
Выходные данные	Неформалный отчет	Формальный отчет	Список дефектов, результаты метрик, формальный отчет

Сходства и различия различных методов статического тестирования представлены на слайде. Типичные сценарии проведения рецензий обычно похожи — например, для технического анализа они следующие:

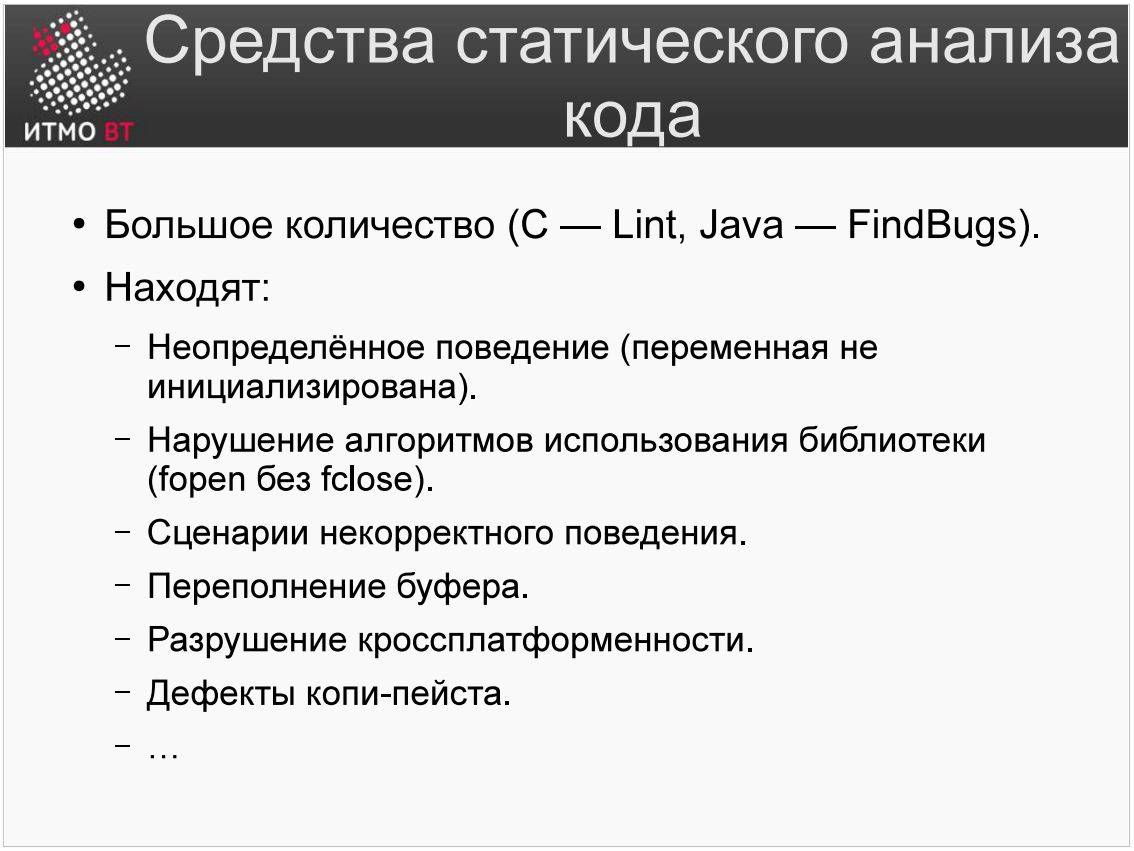
1) Организационная подготовка планов, необходимых ресурсов, обучение инспекторов, и т.д.
2) Планирование техническим лидером технического анализа артефактов.
3) Обзор процедуры проведения технического анализа для участников.
4) Обзор программного продукта.
5) Подготовка проведения встречи:

- Исследование продукта, в результате которого найденные аномалии ПО передаются техническому лидеру и авторам артефактов.
- Лидер уточняет время подготовки и осуществляет планирование встречи.

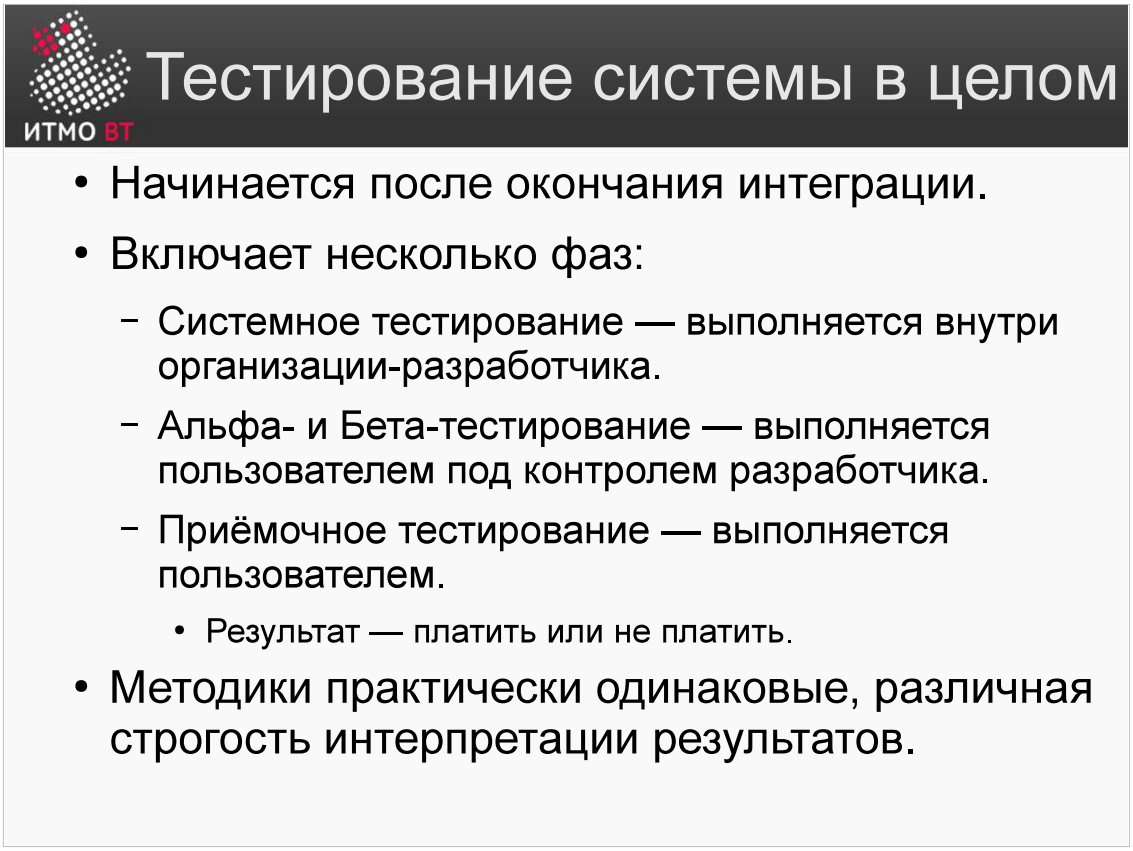
6) Проведение встречи. Включает в себя следующее:

- Определение плана тестирования. Оценка текущего состояния продукта.
- Проверка того, соответствует ли продукт необходимым критериям:
  - продукт является законченным, соответствует требованиям, надлежащим образом реализован и подходит для использования;
  - изменения в ПО соответствующим образом реализованы и подходят для указанных модулей;
  - продукт подготовлен для последующих действий над ним;
  - отсутствуют аппаратные аномалии или отклонения от спецификации.
- Идентификация аномалий в ПО.
- Составление списка действий для дальнейшей разработки.
- Составление журнала и отчёта по встрече, планирование дополнительных просмотров.

7) Принятие корректирующих действий переработка. Технический лидер должен убедиться, что все действия из списка выполнены.



Примерами средств статического анализа кода являются Lint для C и FindBugs для Java. Они строят синтаксическое дерево программы и проводят строгую формальную проверку кода на неопределённое поведение (переменная не инициализирована), нарушение алгоритмов использования библиотеки (fopen без fclose), сценарии некорректного поведения, разрушение кроссплатформенности и т.п.



Тестирование системы в целом начинается после окончания интеграции. На этом этапе нужно провести проверку заявленных характеристик.

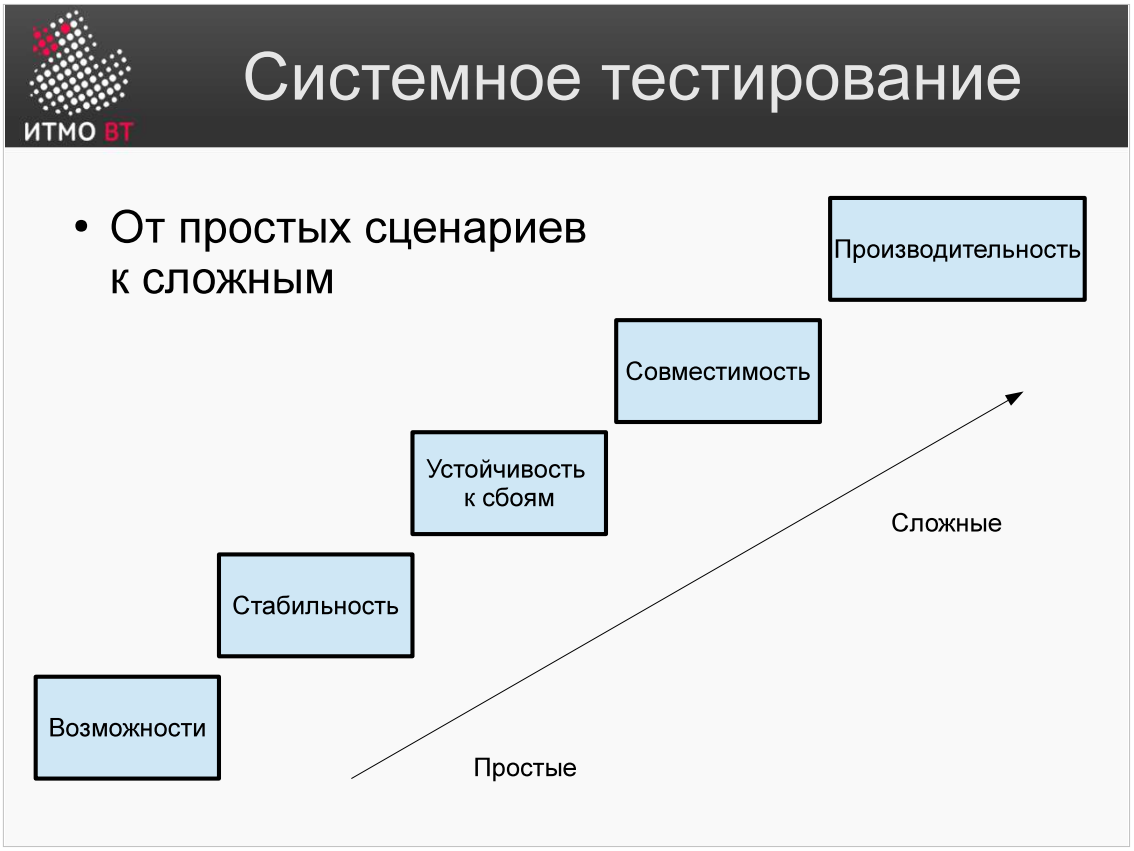
Тестирование системы в целом состоит из нескольких частей:

- *Системное тестирование*. Оно обычно выполняется внутри организации-разработчика без привлечения сторонних лиц.

- *Альфа- и Бета-тестирование* — выполняется пользователем под контролем разработчика. За счёт этого разработчики получают полезные для завершения разработки отзывы. Кроме того, так как тестирование выполняют не разработчики, то тестирующие могут пойти нестандартными путями, которые разработчики не учли. При этом альфа-тестирование производится на окружении разработчиков, а бета-тестирование — в реальном пользовательском окружении (но всё равно под контролем разработчиков).

- *Приёмочное тестирование* — выполняется пользователем в его собственном окружении без контроля разработчика. На этом этапе решается вопрос о выплате разработчикам гонорара.

Методики всех этапов тестирования системы в целом практически одинаковые, отличия есть только в строгости интерпретации результатов.



Системное тестирование обычно производится от простых сценариев к сложным.

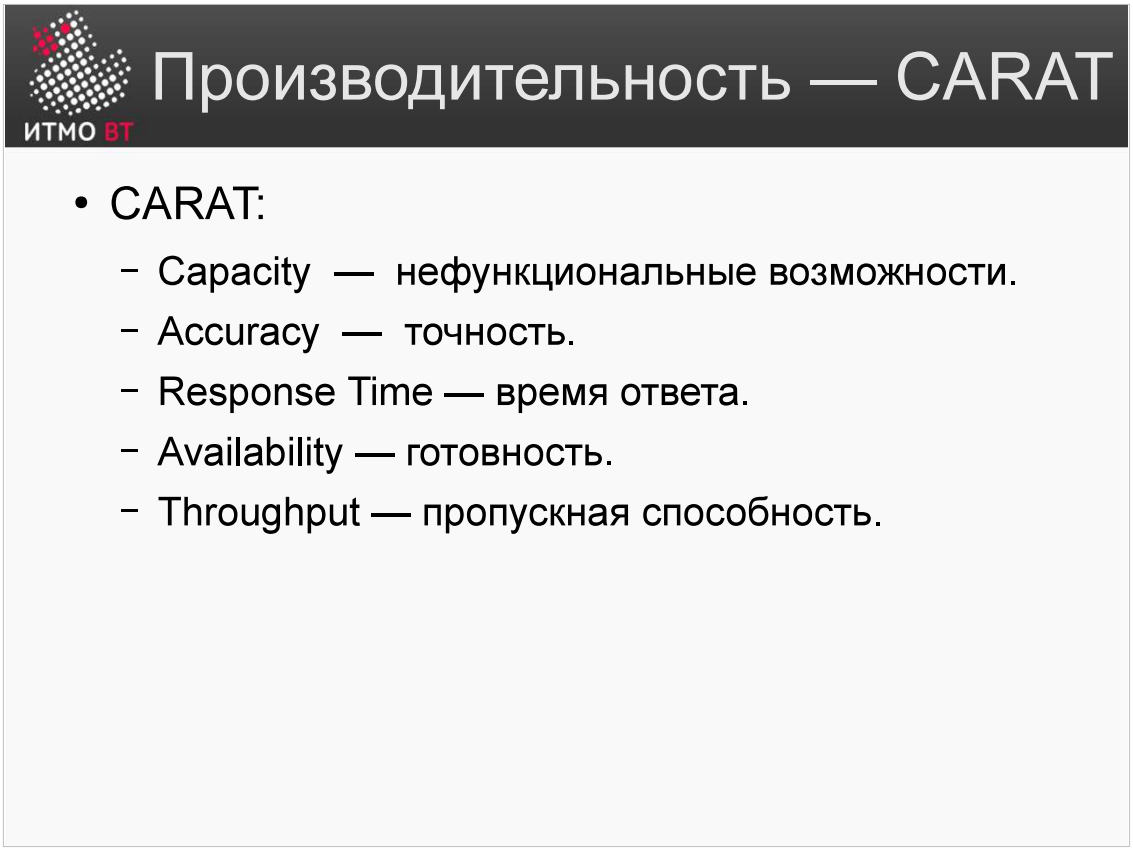
Первыми тестируются заявленные возможности ПО. Обычно на данном этапе используются те же сценарии, что и в функциональном тестировании, но на корректность реализации функций тестируется вся система целиком. Тестовые случаи, приводящие к ошибкам или перегрузке системы, не используются.

После этого сценарии постепенно усложняются. Проверяется стабильность работы системы в таких ситуациях, как одновременное обращение нескольких пользователей, или несколько запросов от одного пользователя.

Затем проверяется устойчивость системы к сбоям: вводятся заведомо неверные данные и проверяется корректность реакции системы на такие ошибки.

После этого проверяются аспекты системы, связанные с совместимостью: кросс-браузерность, возможность использования вместе с программой стороннего ПО.

Затем проводится испытание корректности работы системы в условиях высокой нагрузки и определяются пределы её производительности.



Тестирование производительности включает в себя все виды тестов, которые в англоязычной литературе получили название CARAT: Capacity, Accuracy, Response time, Availability, Throughput.

Capacity (нефункциональные возможности) — возможности, которые система должна предоставлять согласно нефункциональным требованиям. Это характеристики, связанные с заявленным объёмом обрабатываемой информации. Тестирование нефункциональных возможностей заключается в последовательном доведении до предела каждого из имеющихся параметров и наблюдении за поведением системы.

Accuracy (точность) определяет, в основном, точность математических расчётов с заданной константой погрешности. Она может быть критичной, например, для систем моделирования физических процессов. В системах реального времени разрабатываемое ПО должно обеспечить заданную точность вычислений в ограниченный промежуток времени.

Response Time (время ответа) указывает время ответа системы на запрос пользователя. Оно не должно быть слишком большим, или слишком маленьким. Время ответа в интерактивных системах должно попадать в диапазон от 1 до 5 секунд, при нормальной, определённой в требованиях, нагрузке.

Availability (готовность) обычно выражается в коэффициенте готовности.

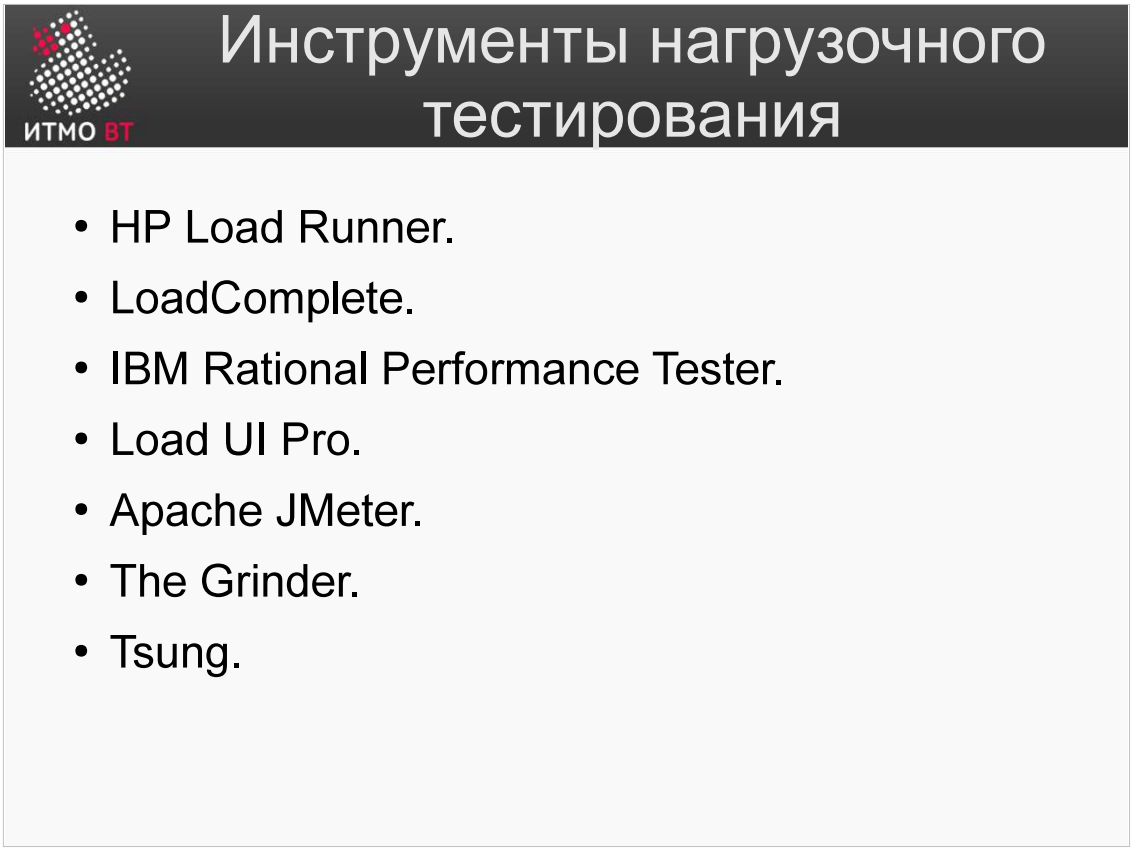
$$Availability = \frac{MTBF - MTTR}{MTBF}$$, где

MTBF — mean time before failure — среднее время до отказа,

MTTR — mean time to recover — среднее время до восстановления.

Например, коэффициент готовности 0.99999 означает, что система может простаивать неработоспособной всего 5 минут в год. Для достижения таких результатов необходимо кластерное решение.

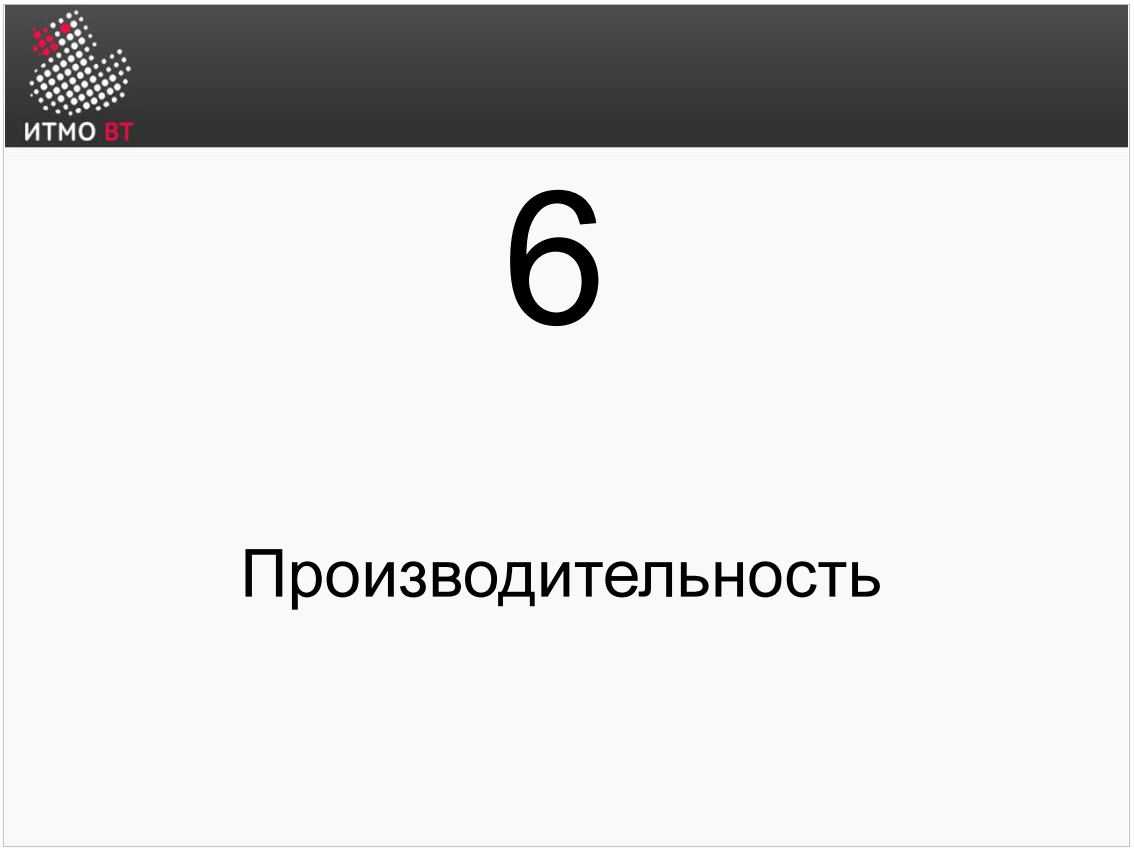
Throughput (пропускная способность) проверяет, например, сколько клиентских запросов система может обработать за единицу времени.

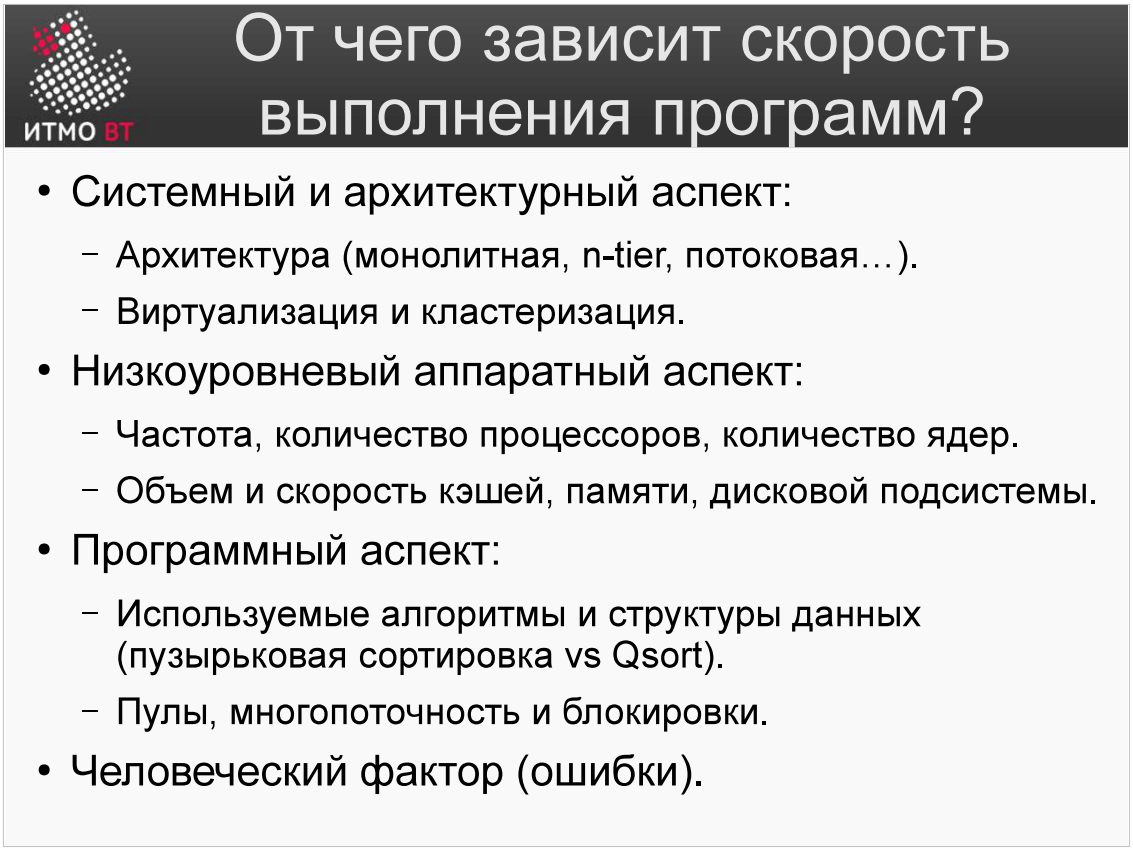


На рынке средств, которые могут организовывать нагрузочное тестирование, существует большой выбор бесплатных и коммерческих продуктов. Все они могут генерировать нагрузку с использованием большого числа протоколов, которые используются в современном программном обеспечении. Наиболее известные из этих средств представлены на слайде.

Нагрузка может быть стационарной, сформированной заданным количеством запросов в единицу времени, а также более сложной — изменяться по заданному закону, например, расти с определённым шагом до какого-то граничного значения. Все данные средства обеспечивают широкие возможности замеров времени ответа и формирование журнала тестирования.

Для программного обеспечения, разработанного на Java, наиболее популярным инструментом нагрузочного тестирования является Apache JMeter. Он позволяет удалённо формировать нагрузку на тестируемое программное обеспечение с использованием большого числа протоколов, а также организовывать распределённую нагрузку, что имеет большое значение для высокопроизводительных систем.


ИТМО ВТ
6
Производительность



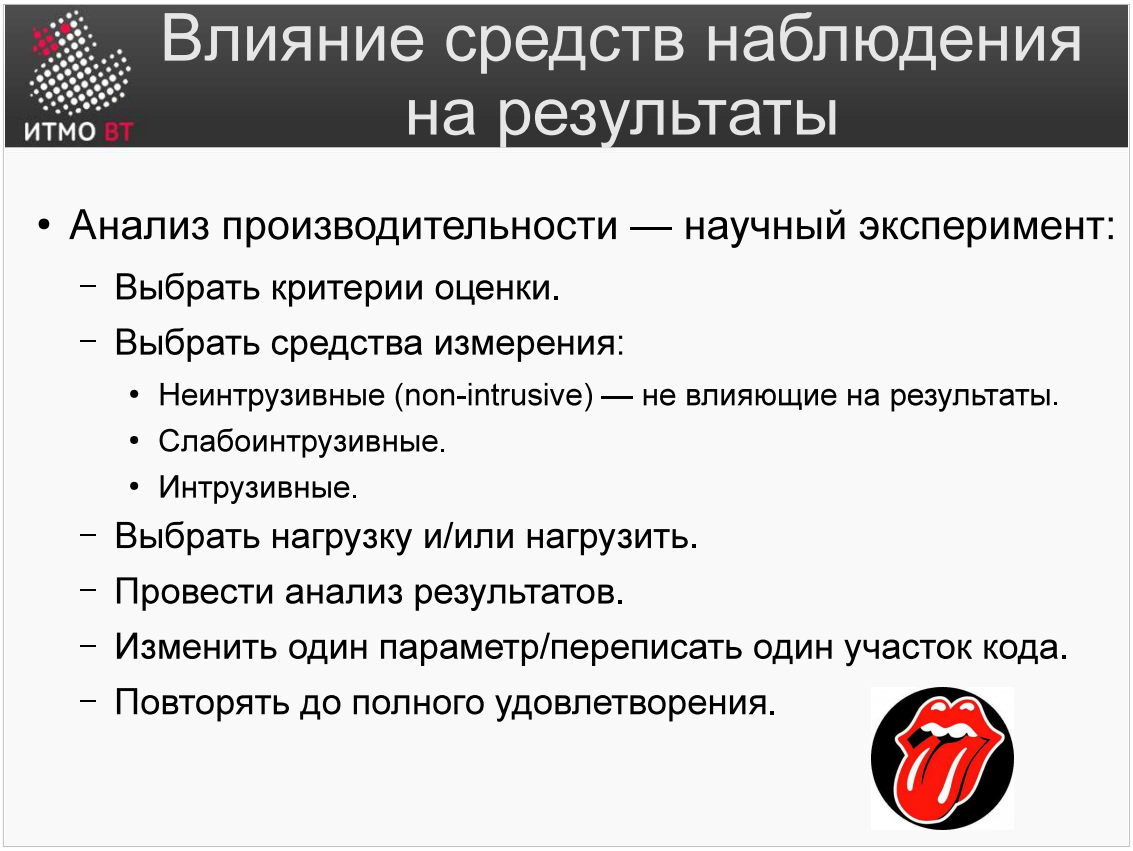
Рассмотрим аспекты современных систем, связанные с производительностью. В первую очередь это системный и архитектурный аспекты. Нужно учитывать, что при размещении программы в вычислительной среде, там уже работают другие программы. Кроме того, вычислительная среда построена согласно определённой архитектуре, которая задаёт структурные особенности выполняемых в ней программ, их параметры и характеристики. Такая архитектура может быть:

- Распределённой, где требуется баланс производительности на всех уровнях обработки информации (например, в классической трехзвенной архитектуре низкая производительность слоя БД может привести к падению производительности уровня бизнес-логики).
- Кластерной, где требуется балансировка между различными узлами кластера.
- Виртуализированной, где на одной серверной ферме работает много различных подсистем, взаимно влияя друг на друга из-за общих ограничений самой фермы.

Низкоуровневый аппаратный аспект связан с техническими характеристиками аппаратуры. Например, от тактовой частоты процессора зависит линейная скорость выполнения каждого потока программы, от размера кэшей и уровней конкуренции разных потоков зависит то, как много данных могут быть помещены в кэш второго уровня и насколько долго они могут там находиться, что также влияет на скорость обработки этих данных.

Программный аспект связан с выбором подходящих алгоритмов для разрабатываемых программ. Зачастую сроки сдачи в эксплуатацию являются более важным фактором, чем оптимальность решения поставленных задач. Примером программного аспекта является выбор между использованием метода пузырьковой сортировки и метода Qsort. Первый метод значительно менее производителен, чем второй, однако его реализация занимает меньший объем памяти. Кроме того, важными факторами могут являться пулы, реализация многопоточности, блокировки и др.

Важным фактором является человеческий — ошибки, неправильно принятые архитектурные решения, узкое знание предметной области и т.д.



Любая работа по анализу производительности похожа на научный эксперимент. Сначала выбираются критерии оценки — параметры, по которым можно оценить производительность программы. После этого выбираются средства измерения. Различаются *неинтрузивные* (non-intrusive) средства измерения, не влияющие на результаты, *слабоинтрузивные* (например, вольтметр вносит небольшие искажения в измеряемое напряжение) и *интрузивные* (например, весы у нечистого на руку продавца на рынке).

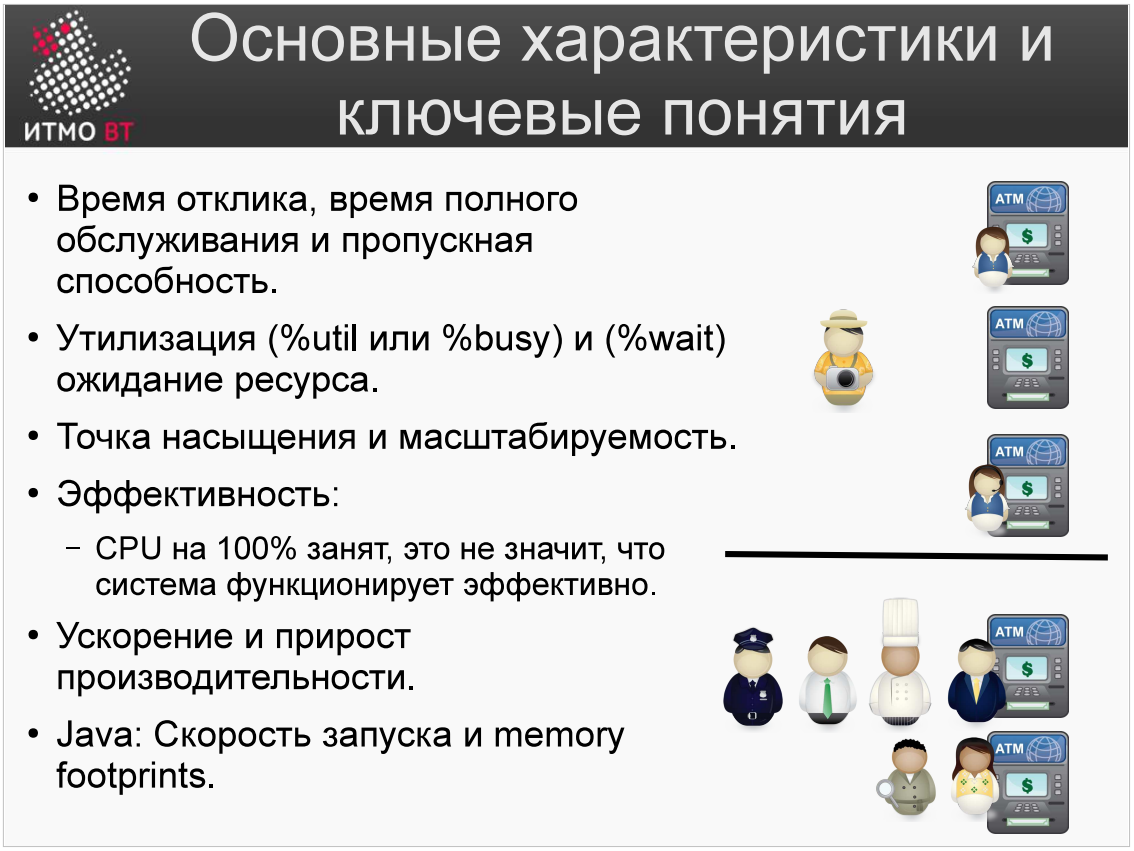
Применительно к измерению производительности вычислительных систем, практически любое измерение тем или иным образом влияет на результат самого измерения. Например, счётчики производительности операционной системы работают незаметно для пользователя (ибо они всё равно будут собираться ОС на системном уровне), но, если бы они не собирались, процессорное время на уровне системы не тратилось бы, и отдавалась пользовательской программе.

После выбора критериев и средств измерения, необходимо выбрать нагрузку и соответствующим образом нагрузить систему. Задача выбора нагрузки сложна тем, что эта нагрузка должна быть максимально приближена к реальной пользовательской нагрузке, т.е. её нужно сделать максимально *эквивалентной*. Во время работы программы под нагрузкой производится сбор результатов — например, системными утилитами производится периодический замер счётчиков ОС. Периодичность сбора и объём получаемых журналов, в свою очередь, тоже может влиять на результаты.

Затем проводится анализ результатов и исключение данных, наведённых средствами измерения. Например, если журнал измерений растёт со скоростью 1Мб/с, то это значение нужно исключить из производительности дисков.

После анализа выбирается один из параметров программы или участков кода, и варьируется/переписывается. Эксперимент повторяется. Потом аналогичным образом изменяется другой параметр и т.д.

Процедура повторяется до получения удовлетворительного результата производительности. На практике этот процесс происходит постоянно, как пели The Rolling Stones: "I can't get no satisfaction, 'Cause I try and I try and I try and I try".



Параметры производительности тесно связаны между собой. Математически точные их определения будут представлены в курсе теории массового обслуживания.

Время отклика системы (latency) измеряется от выдачи запроса до получения первых результатов. Например, время отклика банкомата может быть определено как время с момента ввода карточки до получения первой порции денег, при этом *полное время обслуживания* определено до получения полной суммы (в несколько приёмов).

Пропускная способность показывает, какое максимальное количество запросов за единицу времени может пройти через канал ввода-вывода, систему, или её конкретный узел. Очевидно, что пропускная способность на верхнем рисунке выше, чем на нижнем.

Утилизация ресурса (%util) показывает, какую долю времени ресурс занят полезной работой. Если банкомату из десяти минут времени работы девять минут приходится обслуживать клиентов, то его утилизация составит 90%. Ожидание (%wait) ресурса учитывает среднюю длину очереди. % ожидания показывает сколько времени очередь *не пуста*.

В любых системах есть *точка насыщения* — момент, когда нагрузка достигает предельного значения, которое может обработать устройство или программа. Иными словами, точка насыщения — это точка достижения максимально возможной производительности. После достижения данной точки производительность может снижаться достаточно резко (обычно — экспоненциально). Более плавное снижение говорит о том, что основные составные компоненты системы хорошо сбалансированы. *Масштабируемость* характеризует то, насколько можно количественно расширить и нагрузить систему.

Эффективность ПО показывает соотношение полезной работы системы к общему количеству работы. Например, загрузка виртуальных страниц из swap-области заставляет систему загружать процессор и диск, но общая эффективность программы падает. *Ускорение работы* и *прирост производительности* показывают, насколько больше полезной работы выполняет программа после внесения изменений.

В Java "memory footprints" называют стратегии использования памяти. Если работа с памятью ведётся некорректно, то скорость её работы резко снижается.

Классический "нисходящий" метод поиска узких мест последовательно рассматривает систему от более общих компонент к более частным.

В первую очередь администратор системы ищет ошибки аппаратуры и конфигурации системы. Проверяются системные журналы, файлы конфигурации системы и прикладного ПО. Сбойный блок на диске или дефект соединительного кабеля может привести к повторному чтению одного и того же участка информации, что снизит скорость выполнения операций ввода-вывода. В системных журналах такие ошибки видны сразу, при этом система обычно "честно" сообщает, к примеру, что из-за сбоев контрольной суммы скорость обмена по шине снижена. Администратор, например, может ошибочно выделить мало памяти для Java-машины, результатом чего будут частые сборки мусора, останавливающие систему. С другой стороны, выделение слишком большого объёма памяти приведёт к избыточному накоплению мусора и длительной его очистке в перспективе, при этом основная задача, естественно, решаться не будет.

После исключения ошибок начинается собственно наблюдение за системой. Операционная система обладает достаточно развитыми средствами наблюдения за своими подсистемами. Средства системного, межузлового мониторинга и мониторинга виртуальных машин помогают наблюдать общую картину работы приложения и определить, на что в ней тратится основное время.

Следующим этапом является наблюдение за самим приложением, т.е. фиксируются и анализируются алгоритмические проблемы, проблемы API, проблемы, связанные с многопоточностью, блокировками и синхронизацией. В качестве средств наблюдения используются средства мониторинга и профилирования приложений.

Если полученного в ходе предыдущего этапа прироста производительности оказывается недостаточно, происходит переход к мониторингу микроархитектуры: проверяются количество инструкций, обрабатываемых процессором за один такт, выравнивание данных, оптимизация кэшей, пузырьки конвейера, предсказание переходов и т.д. Для улучшения ПО на этом уровне необходимо знать ассемблерный код, архитектуру процессора, принципы компиляции и пр. Обычно в пользовательском ПО до этого уровня не доходят из-за сложности внесения изменений.

	Объем	Тд	*	Тип	Управл.
CPU	100-1000 б.	<1нс	1с	Регистр	компилятор
L1 Cache	32-128Кб	1-4нс	4с	Ассоц.	аппаратура
L2-L3 Cache	0.5-32Мб	8-20нс	30с	Ассоц.	аппаратура
Основная память	0.5Гб-4ТБ	60-200нс	3-10м	Адресная	malloc,free slab-alloc
SSD	128Гб-1Тб/drive	25-250мкс	5д	Блочн.	open read/write
Жесткие диски	0.5Тб-4Тб/drive	5-20мс	4мес	Блочн.	open read/write
Магнитные ленты	1-6Тб/к	1-240с	200л	Последов.	программно

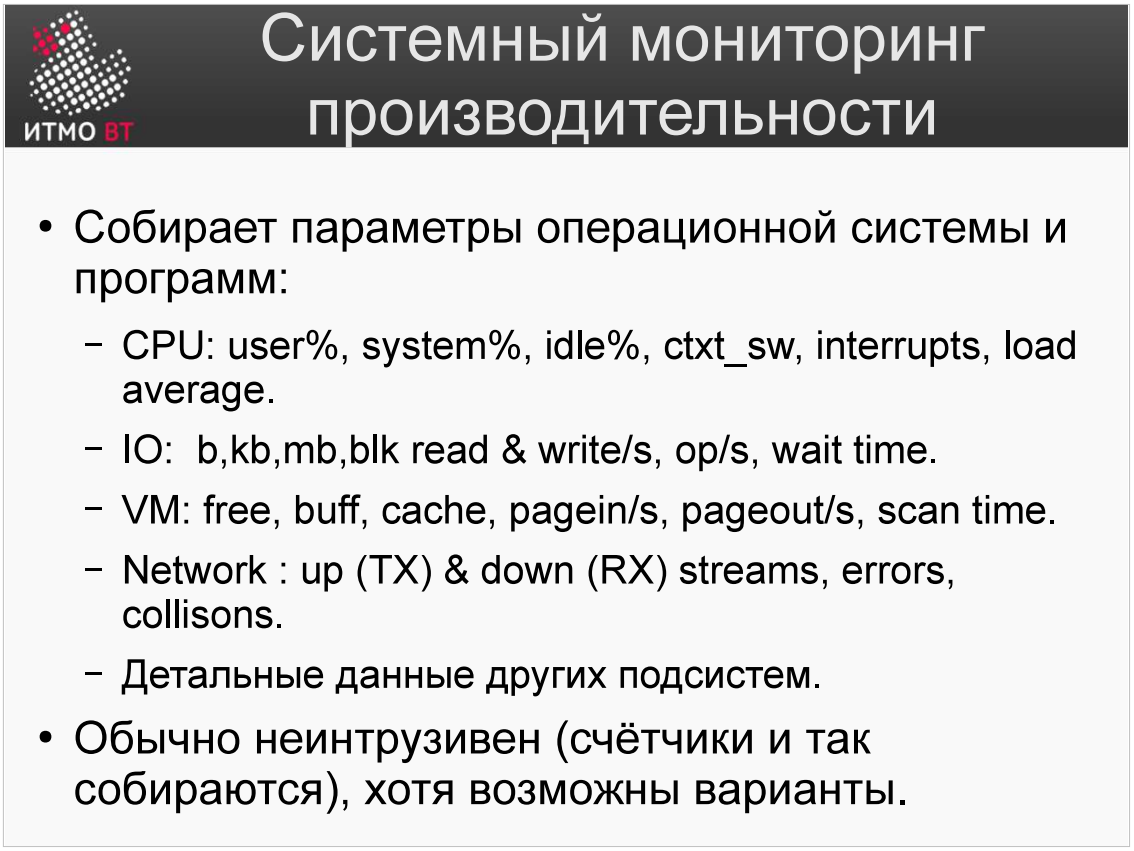
При разработке программ учёт скорости доступа к различным компонентам архитектуры вычислительной среды может существенно повлиять на итоговую производительность.

Сверху пирамиды находится самая дорогая память с наименьшим объёмом. Внизу находится самая дешевая память с максимальным объёмом. Чем ниже память в пирамиде, тем дороже в плане затрат времени обходится обращение к ней. Например, обращение к основной памяти (в середине пирамиды) может занимать в сотни раз больше времени, чем обращение к регистрам процессора или к кэшу первого уровня. Поэтому минимизация количества обращений к нижним уровням пирамиды во много раз повышает производительность — иногда в сотни или даже миллионы раз. Если исключить обращения к жёсткому диску и однократно записывать необходимые данные в память, то производительность может увеличиться в миллионы раз.

Для иллюстрации такого повышения скорости работы программы рассмотрим простой пример. Если отождествить такт процессора со скоростью диалога в процессе межличностного общения (колонка «*» на слайде), то на вопрос, заданный одним собеседником (например, "Как дела?"), ответ, данный собеседником, придёт через 1 секунду (если он отвечает со скоростью процессора). Если данные, необходимые для ответа, содержится в кэш-памяти первого уровня, то время ответа будет всё ещё приемлемым, всего 4 секунды. Если в кэше второго уровня — то уже 30 секунд, что для беседы выглядит куда как странно. Можно подумать, что человек думает о чём-то другом. Оперативная память даст задержку в минуты, что уже совсем неприемлемо для беседы. Он что, собирается соврать?!

Обращение к постоянной памяти, SSD и жестким дискам даст задержку в дни или месяцы, что в сотни тысяч и миллионы раз замедлит быстродействие процессора. Получение архивной копии данных (аналог постоянной памяти человека, заложенной в детстве) займёт и вовсе сотни лет.

Учёт архитектуры ЭВМ позволяет сделать разрабатываемые программы существенно (на несколько порядков) быстрее.



Системный мониторинг производительности осуществляется при помощи средств ОС, позволяющих получать и анализировать различные параметры операционной системы и запущенных под её управлением программ.

Для мониторинга CPU важно знать количество процессов на уровне пользователя, сколько ресурсов CPU система тратит на эти процессы, время простоя процессоров, параметры ctxt_sw (context switch — переключение контекста), interrupts (прерывания), и load average (средняя загрузка очереди исполнения).

Для мониторинга эффективности работы подсистемы ввода-вывода важно знание числа байтов, килобайтов, мегабайтов (и, иногда, блоков), читаемых и записываемых за секунду, и за какое количество операций это происходит (например, если число операций примерно равно числу переданных байтов, это может указывать на неэффективность алгоритма работы с памятью, слишком маленький размер буфера обмена данными и отсутствие кэширования). Параметр wait time применительно к задачам ввода-вывода определяет время ожидания до получения первых данных.

При мониторинге подсистемы виртуальной памяти отслеживается количество немедленно готовой к использованию свободной памяти (free), объём занятных буферов (buff) и кэшей (cache), количество страниц, читаемых (pagein/s) и записываемых (pageout/s) системой в область подкачки, и scan time — параметр, определяющий скорость сканирования изменённых страниц из этой памяти.

При решении задачи мониторинга утилизации сети отслеживаются потоки upstream (TX, передаваемые данные) и downstream (RX, получаемые данные). Также учитываются возникающие ошибки и коллизии (последние являются редкостью в современных системах, "умные" коммутаторы избавляют от коллизий, передавая данные между всеми сетевыми портами одновременно).

ОС также может предоставить детальные данные других подсистем (например, централизованную статистику ядра).

Обычно такой системный мониторинг неинтрузивен и не оказывает влияния на работу программы, так как ядро всё равно собирает указанные выше счетчики, и их нужно просто прочитать из системы.

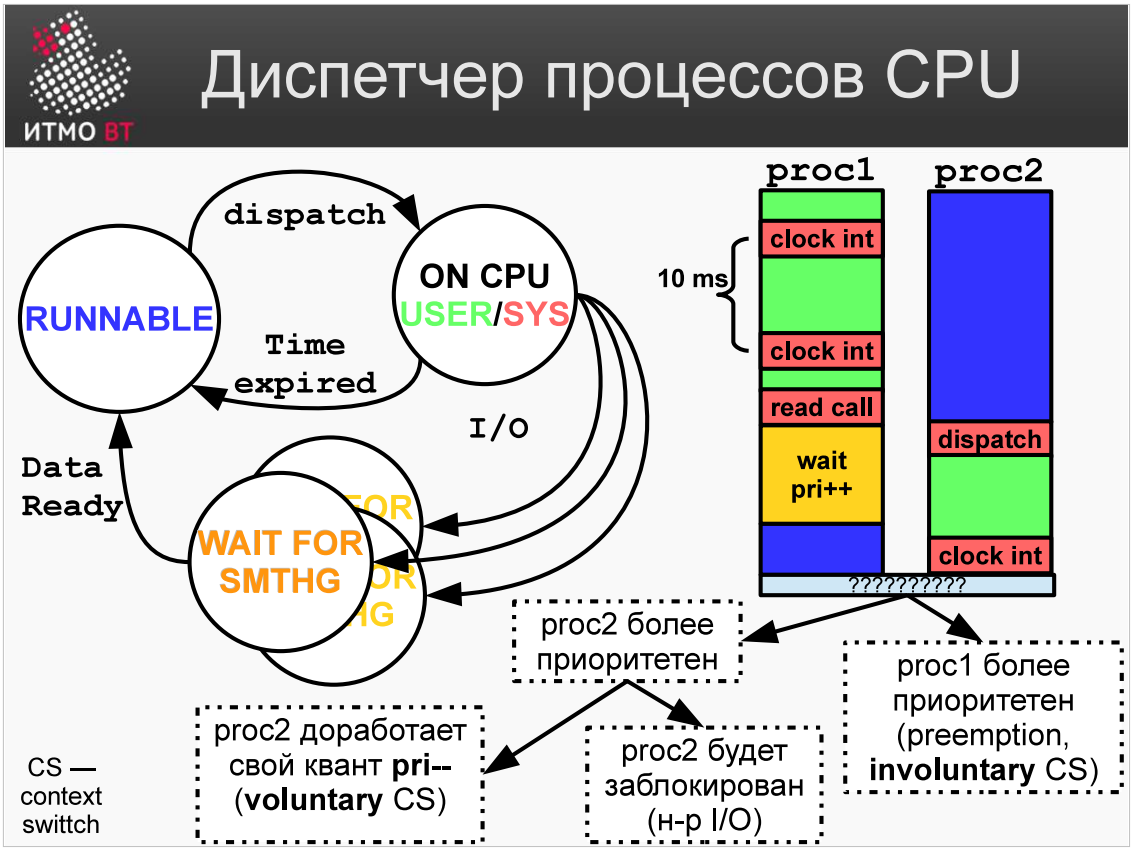


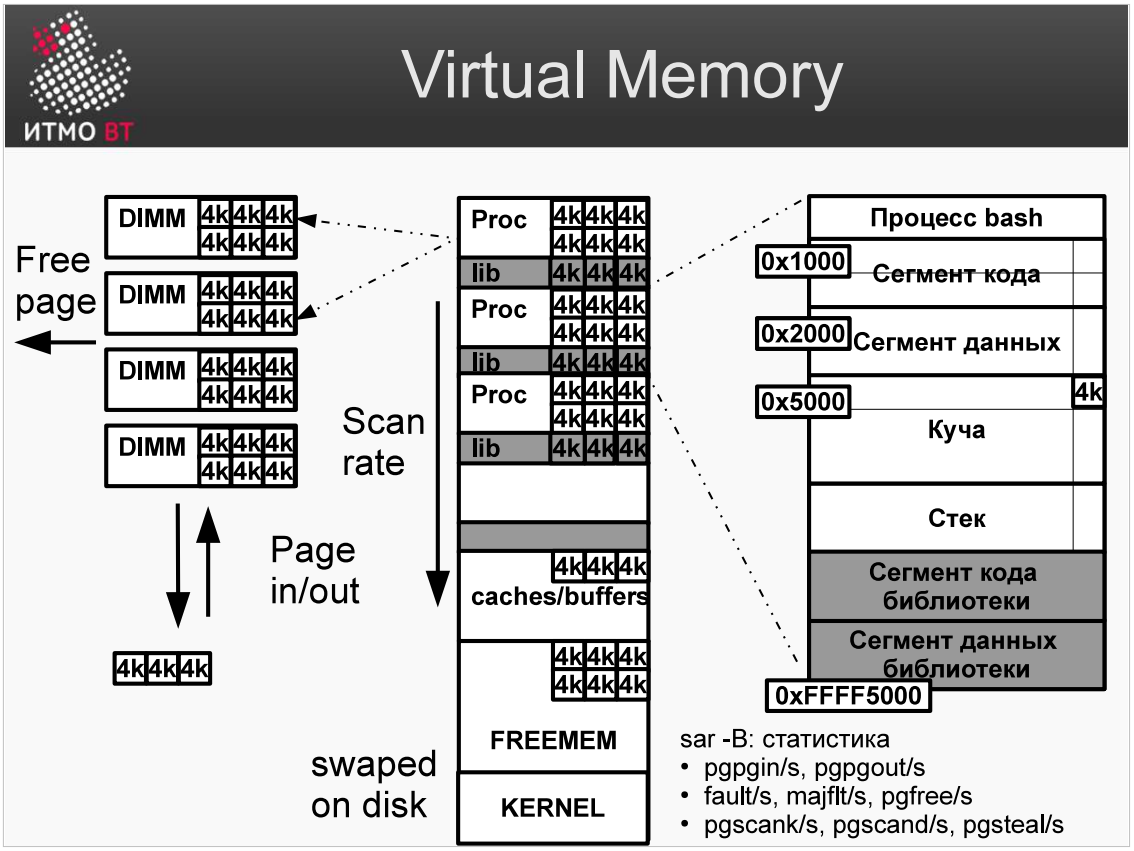
Схема диспетчеризации процессов CPU в сильно упрощенном виде показана на слайде. Показанная схема приблизительно одинакова во всех современных операционных системах. Результаты диспетчеризации можно контролировать при помощи системных утилит.

Процесс или поток внутри процесса может глобально находиться в трёх состояниях: он может быть готов к исполнению (Runnable), находиться на исполняющем устройстве (либо на уровне пользователя), либо на уровне ОС (On CPU User/Sys), или ожидать ввода/вывода, освобождения блокировки и т.п. (Wait for smthg). При этом, во время ожидания нет необходимости занимать ресурсы процессора.

Пусть процесс proc1 в настоящий момент находится на процессоре и работает программа пользователя. Раз в 10 мс (время может быть другим) происходит т.н. прерывание от часов (clock interrupt), во время которого ОС наращивает счетчики производительности, проверяет наличие в очереди на исполнение процесса с более высоким приоритетом, чем у текущего, а также проверяет, не исчерпал ли текущий процесс свой квант времени. В нашем случае, исполнение proc1 продолжается, так как квант времени не исчерпан и proc2 не является более приоритетным, чем proc1, поэтому он остается в режиме готовности в очереди на выполнение. Так продолжается до тех пор, пока не будет исчерпан квант времени, либо не будет сделан вызов из proc1 к устройству ввода-вывода.

При появлении такого вызова (на слайде – read call), proc1 переходит в режим ожидания, и происходит т.н. context switch (процессор переходит к исполнению другого процесса, сохранив окружение предыдущего). Внутри ядра происходит диспетчеризация, в ходе которой выбирается процесс с высшим приоритетом, готовый к выполнению. Процесс proc2 начинает выполняться. Процесс proc1 отправляется в состояние ожидания, но у него повышается приоритет (чем больше время ожидания, тем больше увеличивается приоритет, и наоборот, трата процессорного времени приводит к понижению приоритета, что типично для систем *разделения времени*).

В момент прерывания возможны различные варианты действий. Если приоритет proc2 выше, чем у proc1, то он либо дорабатывает свой квант времени, либо выполняет операцию, связанную с вводом-выводом. Если proc1 более приоритетен, то он снова перейдет на процессор (произойдёт вытеснение – preemption, или несвободное переключение контекста – involuntary CS).

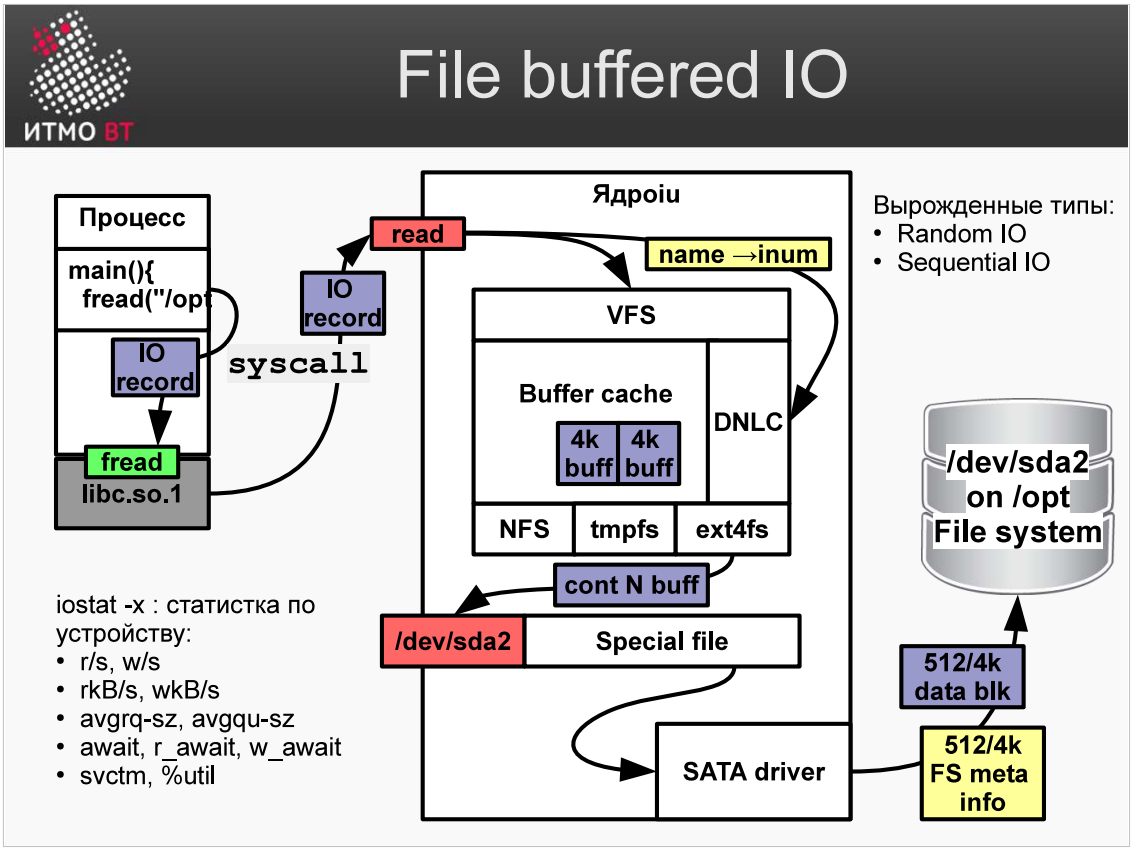


Виртуальная память — метод управления памятью компьютера, позволяющий выполнять программы, требующие больше оперативной памяти, чем имеется в компьютере, через автоматическое перемещение частей программы между основной памятью и вторичным хранилищем (например, жёстким диском).

В работе виртуальной памяти задействованы собственно физическая память, устройство подкачки на дисках (swap-устройство), и виртуальные страницы, которые могут принадлежать либо физической памяти, либо виртуальной памяти. Сегментно-страничная структура процессов типична для современных ОС, где каждая страница может находиться в нескольких состояниях: быть в памяти, находиться на swap-устройстве, или быть зарезервированной (необходимой для выполнения, но физически еще нигде не расположенной). Вся память процесса поделена на страницы (которые могут быть размером 4Кб и больше, например, 2Мб). Размер страницы влияет на количество *мэпингов* (записей отображений структур виртуальных адресов ОС на страницы внутри банка памяти). Следует отметить, что используемые для чтения страницы (такие, как код программы или статически скомпилированные данные) необязательно перемещать на swap-устройство, их можно просто удалить из памяти, потому, что они могут быть загружены из исполняемого файла программы. Страницы, связанные с файлами, обычно называют *именованной* памятью, а те, которые созданы динамически в куче (*heap*), получили название *анонимные*.

С виртуальной памятью связано несколько параметров. Scan rate — число просканированных страниц за единицу времени. Если значение skan rate высоко, это означает недостаток оперативной памяти или постоянную работу ОС, связанную с манипуляцией страницами. ОС при недостатке памяти производит последовательное сканирование редко используемых, неизменных страниц, и принимает решения (в зависимости от их типа) о записи их на диск или удалении. Если процессор обратился к отсутствующей в памяти странице, происходит *страничный сбой* (page fault). Данное событие может быть минорным (minor), когда для страницы нужно просто создать запись (мэпинг) в таблицах адресации, или *существенным* (major), когда страница должна быть загружена из диска подкачки.

Сбор статистики виртуальной памяти в ОС unix/linux осуществляется при помощи команды sar -B. Аргументы этой команды и собираемые данные представлены на слайде.



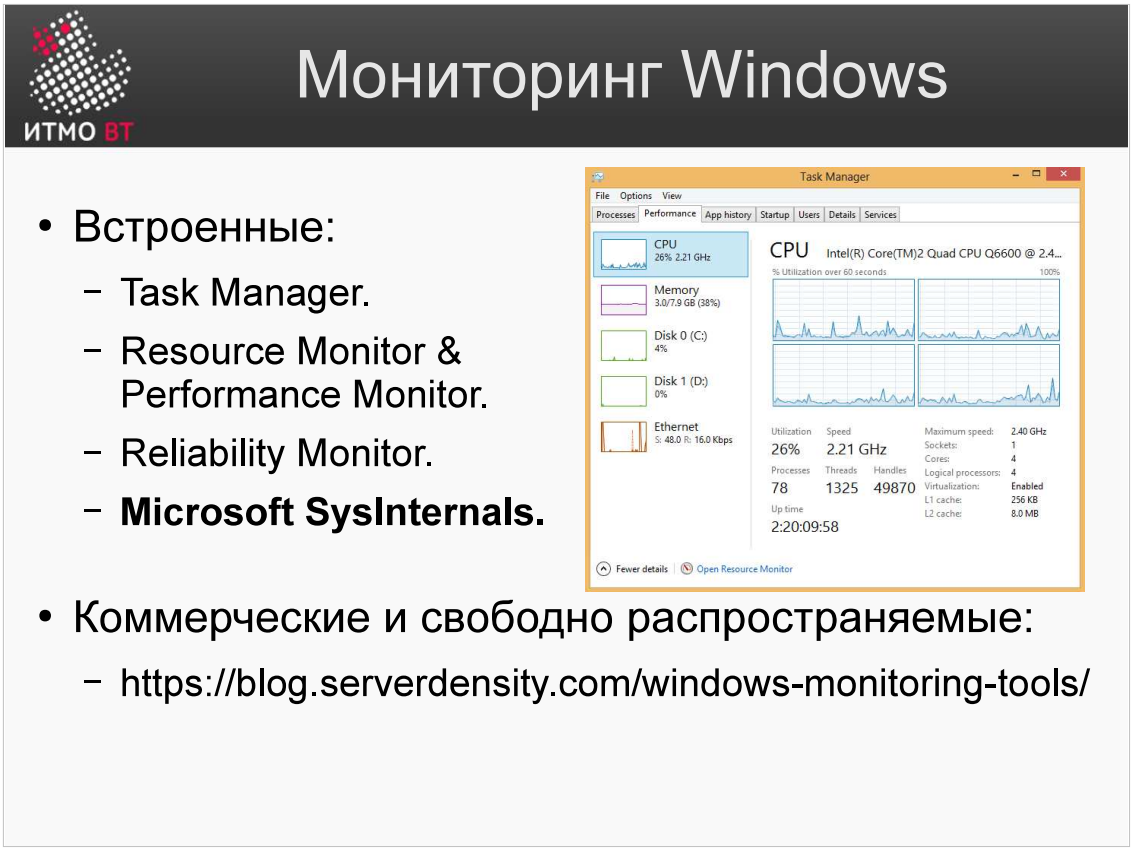
На слайде приведена организация буферизованного ввода-вывода. При чтении данных вначале указывается количество байт, которые нужно прочитать (IO record size). Интенсивность чтения данных определяется требованиями программы.

Предположим, внутри программы производится обращение к функции fread, которая находится в системной библиотеке. Функция fread формирует запрос к ядру и вызывает соответствующую функцию ядра read. Ввод-вывод в ядре попадает в подсистему VFS (virtual file system, виртуальная файловая система), и запрос попадает на уровень, не связанный с конкретной файловой системой. Имя файла преобразуется в DNLC (directory name lookup cache — кэш, ускоряющий обработку имен файлов) в номер файла inode в необходимой файловой системе.

ОС экономит ресурсы при обращении к устройствам ввода-вывода, и операция read будет работать в первую очередь с буферным кэшем (Buffer cache). В нём данные содержатся в виде блоков данных файла в ОЗУ, формируя промежуточное хранилище. Каждые 30 секунд (время зависит от ОС) данные, которые были помечены как изменённые, сохраняются на диск. Ниже, под виртуальной файловой системой существуют реализации модулей ядра физических файловых систем. К точке монтирования подключается устройство с использованием его специального файла, после этого работу ведёт драйвер, и на уровне аппаратного интерфейса данные перемещаются в дисковое устройство.

Для каждой файловой системы существуют наблюдаемые параметры: количество чтений (r/s), количество записей (w/s), объем читаемых и записываемых данных (rkB/s и wkB/s), средние времена запросов (avgrq-sz и avgqu-sz), время ожидания (await, r_await, w_await), время обслуживания (svctm), и процент занятости устройства (%util).

Применительно к вырожденным типам обмена с дисковой подсистемой различают *случайный доступ* (random IO, каждый запрос обращается к новому месту диска) и последовательный доступ (sequential IO, данные записываются или читаются большими группами последовательно), при этом скорость чтения/записи во втором случае во много раз выше, чем в первом. Для случайного доступа существенно повысить скорость обмена данными может использование твердотельных накопителей (solid-state drive, SSD) из-за отсутствия в них (в отличие от дисков) движущихся частей. Команда iostat показывает рассмотренные выше характеристики обмена.

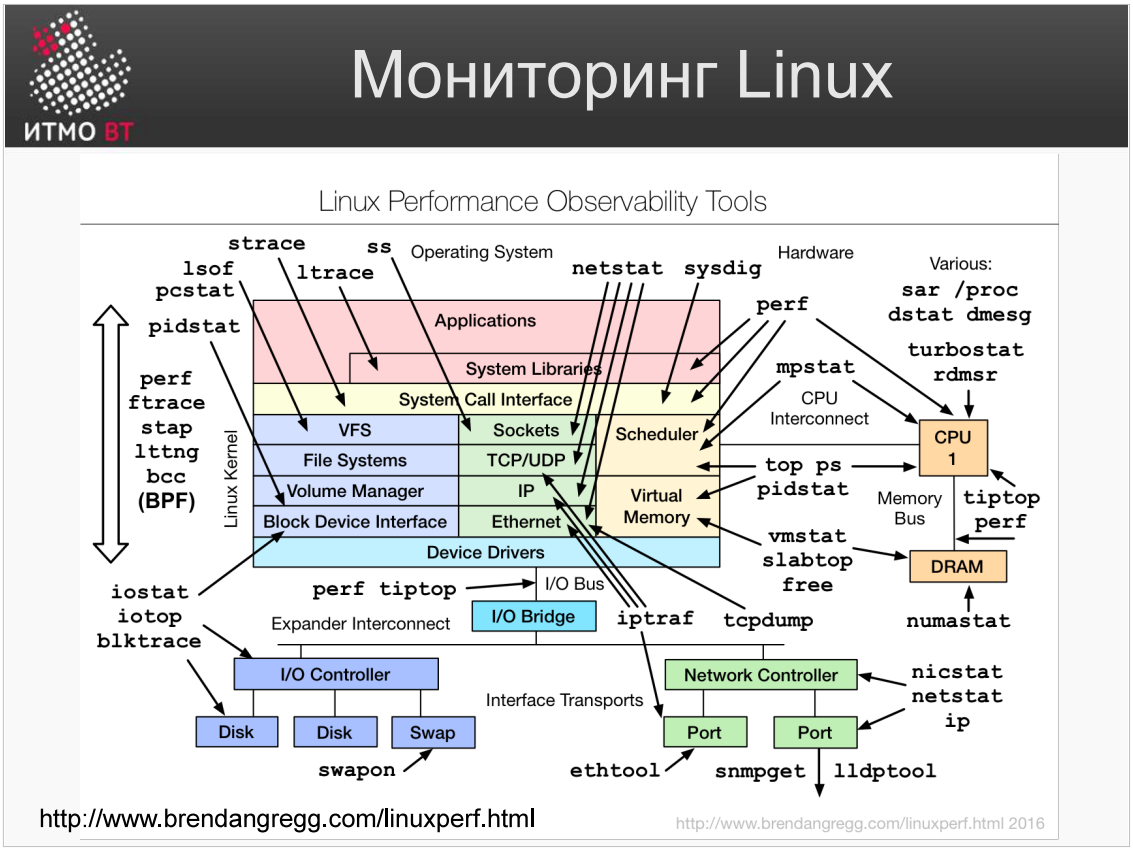


В различных операционных системах существуют разные средства мониторинга.

В Windows самым распространённым является стандартное средство системного мониторинга Task Manager, где, помимо всего прочего, выводится статистика использования процессора, памяти, дисковой подсистемы и др.

Для более детального исследования поставляются также специальные системные оснастки, такие, как Resource Monitor & Performance Monitor, Reliability Monitor и т.д.

Наиболее детальную информацию можно получить из средства Microsoft SysInternals (независимая разработка, впоследствии купленная компанией), которое базируется на внутрисистемных счетчиках и информации, вследствие чего показывает наиболее точные результаты по сравнению с другими средствами.



В ОС семейства Linux ситуация на первый взгляд выглядит сложнее, чем с Windows. В Linux существует большое число средств мониторинга, наблюдающих за определёнными подсистемами ОС и учитывающих особенности архитектуры этих подсистем. В общем случае они являются неинтрузивными.

На слайде видно, из счетчиков каких подсистем такие стандартные для всего семейства Unix средства, как vmstat, mpstat, iostat и netstat берут информацию о производительности.

Утилита top позволяет динамически наблюдать характеристики запущенных процессов, такие, как текущий приоритет (PR), занимаемая память (VIRT — общий размер адресного пространства процесса, RES — размер в физической памяти, SHR — в совместно используемой с другими процессами) и др. В статусной строке можно вывести большое число общесистемных характеристик.

Утилита sar имеет множество опций для вывода той, или иной системной информации ядра. Это одна из первых утилит мониторинга производительности во всём семействе ОС UNIX. В современной ситуации следует ее использовать осторожно, поскольку используемые в этой утилите подходы ко сбору некоторых метрик устарели.

Отдельно следует выделить средство perf, которое может собирать и показывать большое число характеристик не только ядра, но и запущенной под управлением perf программы. Perf умеет работать со счётчиками производительности процессора для сбора таких событий, как промахи мимо кэша.

Для детального наблюдения за процессом можно использовать strace. Эта утилита позволяет проводить трассировку системных вызовов, которые процесс выполняет к ядру ОС и системным библиотекам (таким, как libc.so). Strace в общем случае достаточно интрузивен.

System tap (stap) позволяет установить точки сбора информации в ядре, собрать и агрегировать информацию о подсистемах ядра. Это средство использует свой собственный язык программирования. В Solaris существует аналог stap, который называется dtrace, который впервые продемонстрировал подобный (слабоинтрузивый) подход к мониторингу.

На слайде приведены примеры наблюдения за производительностью с помощью Linux-утилиты vmstat. Формат команды определяет, что нужно произвести 5 последовательных измерений с интервалом в 1 секунду.

Первая строчка под заголовком таблицы vmstat показывает усреднённые системные значения с момента запуска ОС, а вторая и последующие — мгновенные значения с последнего измерения.

В первом примере процессор загружен на 100%, и большая часть этой загрузки приходится на работу с приложением пользователя (столбец cpu us).

Во втором примере основное время процессор проводит на уровне ядра операционной системы (столбец cpu sy). Такие высокие значения нетипичны для нормальной работы системы. Можно предположить, что в каком-то модуле ядра дефект. Искусственно получить такую ситуацию можно чтением из датчика псевдослучайных чисел (который реализован в драйвере ядра) большими блоками, например при помощи команды: dd if=/dev/urandom of=/dev/null bs=1024k

В третьем примере значительная часть времени процессора приходится на ожидание на блокировках (столбец cpu sy). Наличие значительного дискового ввода-вывода (bi, bo) и высокая выгрузка на swap-устройство страниц памяти (so) указывают на ситуацию, когда недостаточно оперативной памяти. Этот недостаток можно также наблюдать в столбце free. В этом случае понятно, на каких блокировках ждёт процессор — это очередь записи на диск подкачки.

Примеры: что тут происходит? (2)

```
$ vmstat 1 5
procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu------
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 3  1 465872  36132  82588 1018364    7   17    70   127  214  838 12  3 82  3  0
 0  1 465872  33796  82620 1021820    0    0 34592     0  357  781  6 10  0 84  0
 0  1 465872  36100  82656 1019660    0    0 34340     0  358  723  5  9  0 86  0
 0  1 465872  35744  82688 1020416    0    0 33312     0  345  892  8 11  0 81  0
 0  1 465872  35716  82572 1020948    0    0 34592     0  358  738  7  8  0 85  0

$ vmstat 1 5
procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu------
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 1  1      0  35628  14700 1239164    0    0  1740   652  117  601 11  4 66 20  0
 0  1      0  34852  14896 1239788    0    0     0 23096  300  573  3 16  0 81  0
 0  1      0  32780  15080 1241304    0    0     4 21000  344  526  1 13  0 86  0
 0  1      0  36512  15244 1237256    0    0     0 19952  276  394  1 12  0 87  0
 0  1      0  35688  15412 1237180    0    0     0 18904  285  465  1 13  0 86  0

$ vmstat 1 5
procs -----------memory---------- ---swap-- -----io---- --system-- -----cpu------
 r  b   swpd   free   buff  cache   si   so    bi    bo   in   cs us sy id wa st
 2  1 403256 602848  17836 400356    5   15    50    50  207  861 13  3 83  1  0
 1  0 403256 601568  18892 400496    0    0  1048   364  337 1903  5  7  0 88  0
 0  1 403256 600816  19640 400568    0    0   978     0  259 1142  6  4  0 90  0
 0  1 403256 600300  20116 400800    0    0   946     0  196  630  8  5  0 87  0
 0  1 403256 599328  20792 400792    0    0  1076     0  278 1401  7  5  0 88  0
```

Примеры данного слайда посвящены вырожденным случаям обмена с обычными (не-SSD) дисками. В таких дисках существуют несколько носителей информации ("блинов"), сам диск при этом разбит на дорожки, а те, в свою очередь, на сектора. Размер такого сектора составляет 512 байт или 4 Кб. Размер блока (используемого в многих системных утилитах, в т.ч. vmstat) в ОС Linux составляет 1024 байта, т.е. можно считать, что данные ввода-вывода представлены в килобайтах в секунду.

Чтобы произвести обмен с диском, необходимо установить читающую головку на нужную дорожку, дождаться поворота диска к нужному сектору и передать (прочитать или записать) информацию. Ядро при этом ожидает на блокировке, что видно во всех примерах на слайде — высокое время ожидания процессора (cpu wa).

В первом примере происходит активное чтение с диска (bi). Скорость этого чтения составляет 34 Мб/с., и является типичным для *последовательного* (sequential) чтения большого объёма данных с дисков из недалекого прошлого. К сожалению, на слайде нет информации по количеству операций ввода-вывода (информацию по этим операциям можно посмотреть в vmstat -d или iostat). Если провести анализ количества операций, то выяснится, что данные читались большими порциями (размерами блока ввода-вывода). Данные при таком способе выбираются с диска последовательно, количество лишних движений головки диска и его поворотов сведено к минимуму.

Характерное поведение для последовательной записи представлено во втором примере. При этом скорость записи (bo около 20 Мб/с) меньше скорости чтения. Это тоже типичное значение для скорости записи таких дисков.

В третьем примере показана картина *случайного* (random) чтения. При этом данные сильно разбросаны по диску и ввод-вывод происходит небольшими блоками. Это приводит к большим передвижениям головки по диску, скорость чтения падает до 1 Мб/с. Такую деградацию кардинально исправляют SSD. Третий пример также может быть примером работы с медленным устройством, таким, как cdrom.

Скорость чтения-записи жёсткого диска в его начале и конце может отличаться до 40% (в начале быстрее). Это связано с разницей линейной скорости движения головки над поверхностью в зависимости от угловой скорости вращения. Именно поэтому swap-область размещают в начале диска.

Системный анализ "за 60 секунд"

- uptime — load average за 1, 5, 15 минут.
- dmesg | tail — последние ошибки.
- vmstat 1 — есть ли свободная память, paging, распределение CPU.
- mpstat -P ALL 1 — распределение по CPU.
- pidstat 1 — статистика по процессам, горячие процессы .
- iostat -xz 1 — параметры ввода-вывода.
- free -m — проверка исчерпания кэшей/буферов.
- sar -n DEV 1 — сетевая статистика по интерфейсам.
- sar -n TCP,ETCP 1 — сетевая статистика по соединениям.
- top — онлайн-мониторинг параметров.

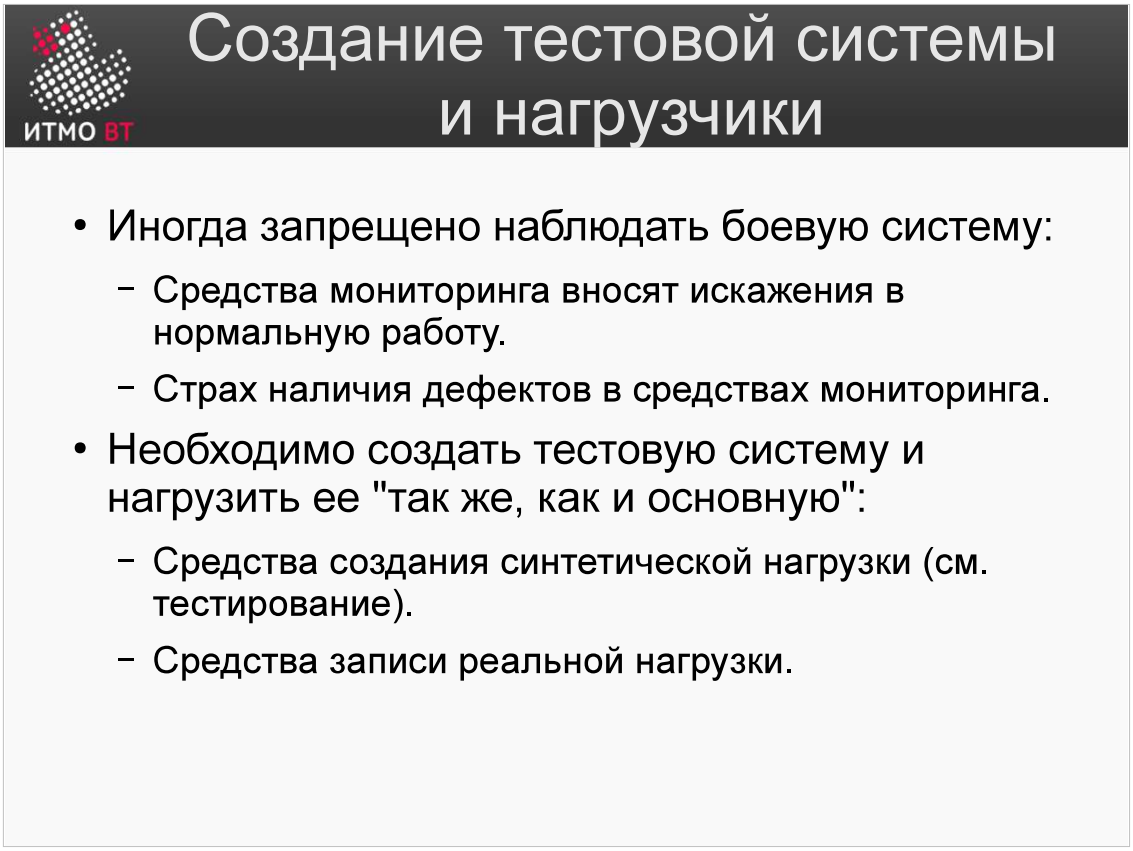
http://www.brendangregg.com/Articles/Netflix_Linux_Perf_Analysis_60s.pdf

На слайде приведён практический подход, позволяющий провести системный анализ «за 60 секунд». Это последовательность команд, позволяющих быстро оценить ситуацию и составить первое впечатление о возможном источнике ошибок.

В первую очередь, команда uptime позволяет показать load average — среднюю загрузку процессорной подсистемы. Значения load average показывают количество готовых к выполнению процессов в очереди на диспетчеризацию за 1, 5 и 15 минут. Если таких процессов много, то система не справляется с нагрузкой.

После этого просматриваются последние ошибки с помощью dmesg | tail. Затем проверяется виртуальная память (с помощью vmstat 1) и распределение процессов по CPU (с помощью mpstat -P ALL 1), так как возможна ситуация, когда один процессор занят, а остальные простаивают. Проверются наиболее «горячие» процессы (с помощью pidstat 1), характеристики ввода-вывода (с помощью iostat -xz 1), память (free -m — проверка исчерпания кэшей/буферов) и сетевая статистика по интерфейсам и по соединениям (sar -n DEV 1 и sar -n TCP,ETCP 1).

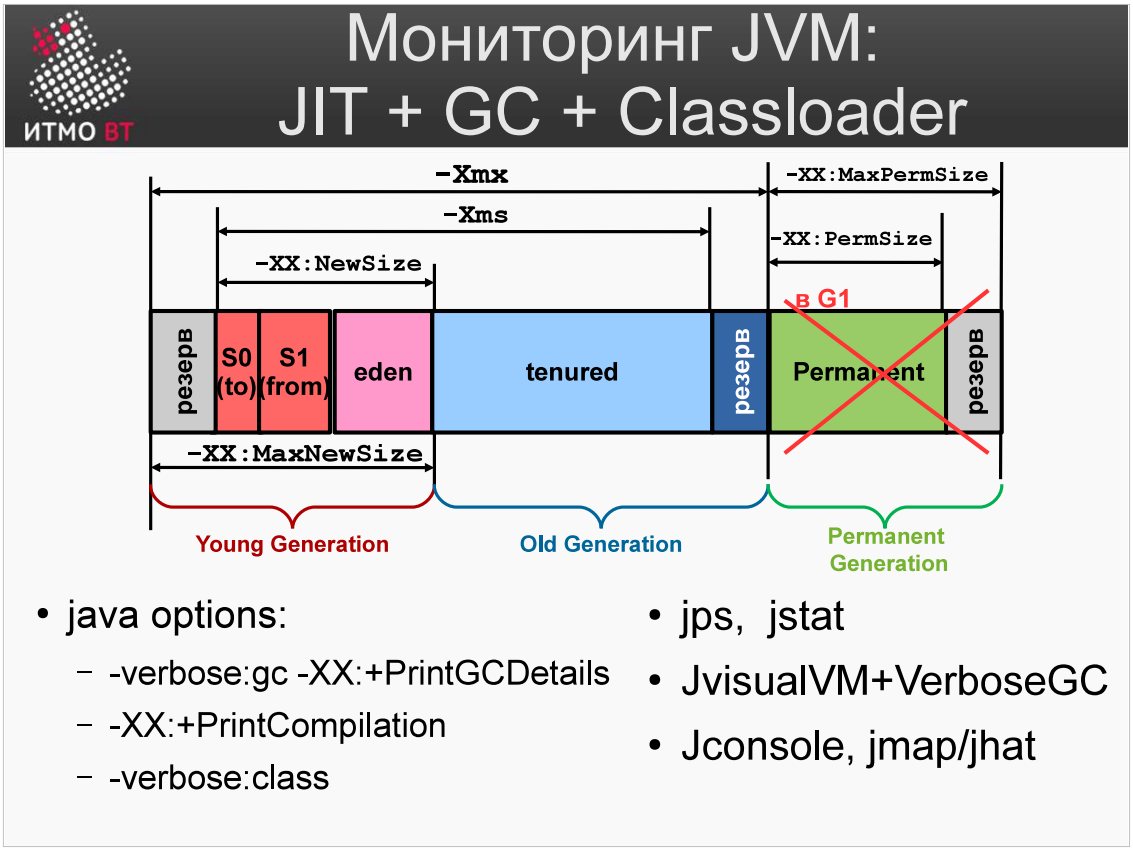
Практически все эти характеристики вместе можно проверить с помощью команды top (онлайн-мониторинг параметров), которая представляет собой «швейцарский нож» для администратора.



В корпоративных системах наблюдение реальных, критически важных для бизнеса систем часто запрещено из-за страха перед внесением искажений в нормальную работу системы средствами мониторинга, или перед наличием дефектов в этих средствах. В таких случаях обычно создают отдельную *тестовую* систему, являющуюся полной копией реальной, и проводят измерения на ней.

Для тестовой системы нужно иметь возможность создавать нагрузку, близкую по характеристикам к реальной пользовательской нагрузке. В тестовой системе также возможно использование интрузивных средств мониторинга.

Нагрузку, аналогичную реальной, можно создать с помощью средства создания синтетической нагрузки или средства записи реальной нагрузки, позволяющего запомнить нагрузку на реальном устройстве и использовать эти данные для тестовой системы. При этом синтетическая нагрузка всегда будет отличаться от реальной, и в средствах создания такой нагрузки используется большое число параметров, позволяющих гибко её настроить.



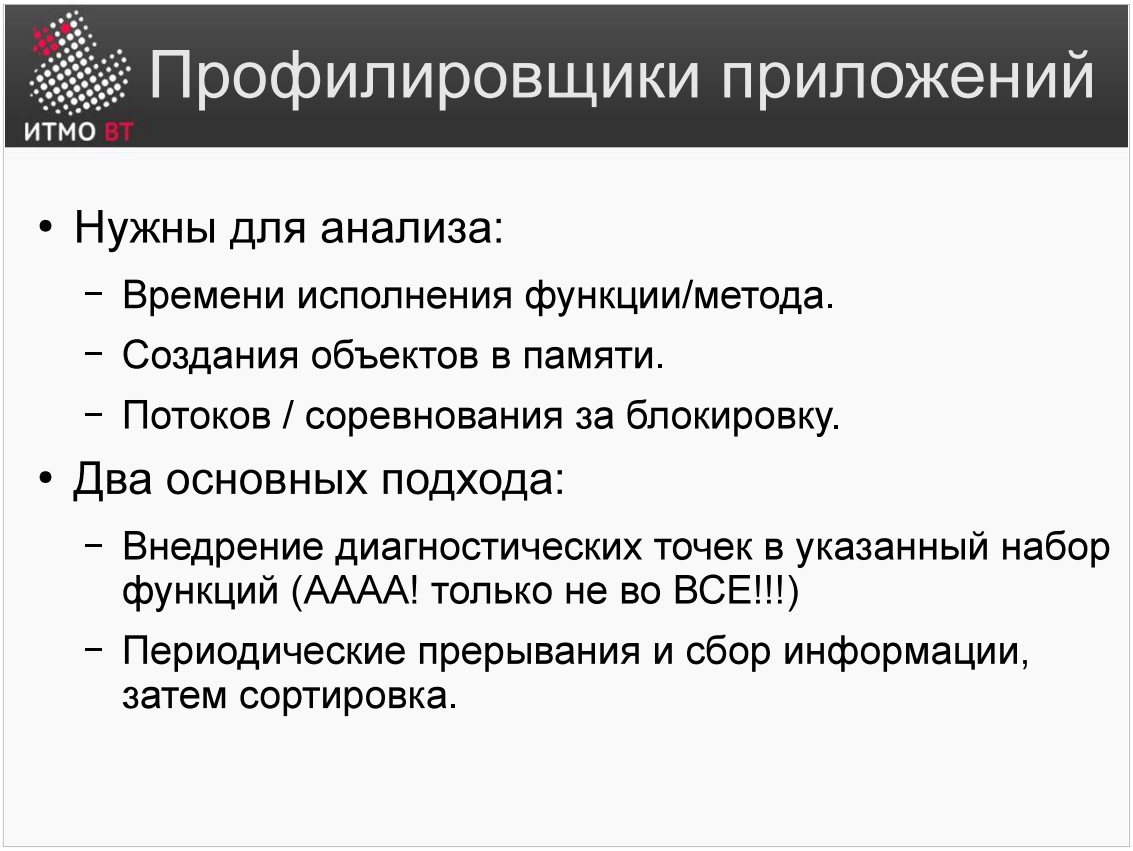
Память, используемая JVM (виртуальная машина Java), состоит из областей памяти для молодого поколения (Young Generation), выживших объектов (Old Generation) и области постоянного хранения (Permanent Generation). В современных JDK Permanent Generation отсутствует.

Создаваемые объекты помещаются в eden. Через определённое время запускается *минорная* сборка мусора (Minor Garbage Collection), и объекты, на которые до сих пор имеются ссылки, копируются в S0 или S1 — области в молодом поколении для выживших (перенесших сборку мусора) объектов. Использование S0 и S1 происходит по очереди. При следующей сборке мусора на наличие ссылок проверяются объекты и в eden, и, предположим, в S0. Если на объект в S0 имеются ссылки, то он перемещается в S1. Так продолжается некоторое количество сборок мусора, и, если объект пережил их все (число может динамически варьироваться от 8 до 32), то он помещается в область выживших объектов. В перманентом поколении обычно находятся объекты, которые специально туда помещены (загруженные классы, пулы строк и т.п.).

Параметрами JVM можно задать размеры памяти для всех поколений. Ключи для задания параметров представлены на слайде. Зарезервированные области выделены операционной системой согласно ключам запуска, но ещё не используются JVM, так как в этом нет пока необходимости.

С точки зрения Java-машины возможно наблюдать всего за тремя группами характеристик: JIT (just-in-time) компиляцией, сборкой мусора, и загрузкой классов Classloader'ом. Для Java существуют развитые средства мониторинга, такие, как jps, jstat, jVisualVM+VerboseGC (удобное средство для мониторинга сборщика мусора), JConsole, jmap/jhat и т.д.

Загрузчик классов сильно влияет на время запуска приложения при большом количестве подгружаемых библиотек. Можно уменьшить время, затрачиваемое на запуск путём предварительной компиляции необходимых классов и помещения их в отдельный файл для быстрой загрузки.



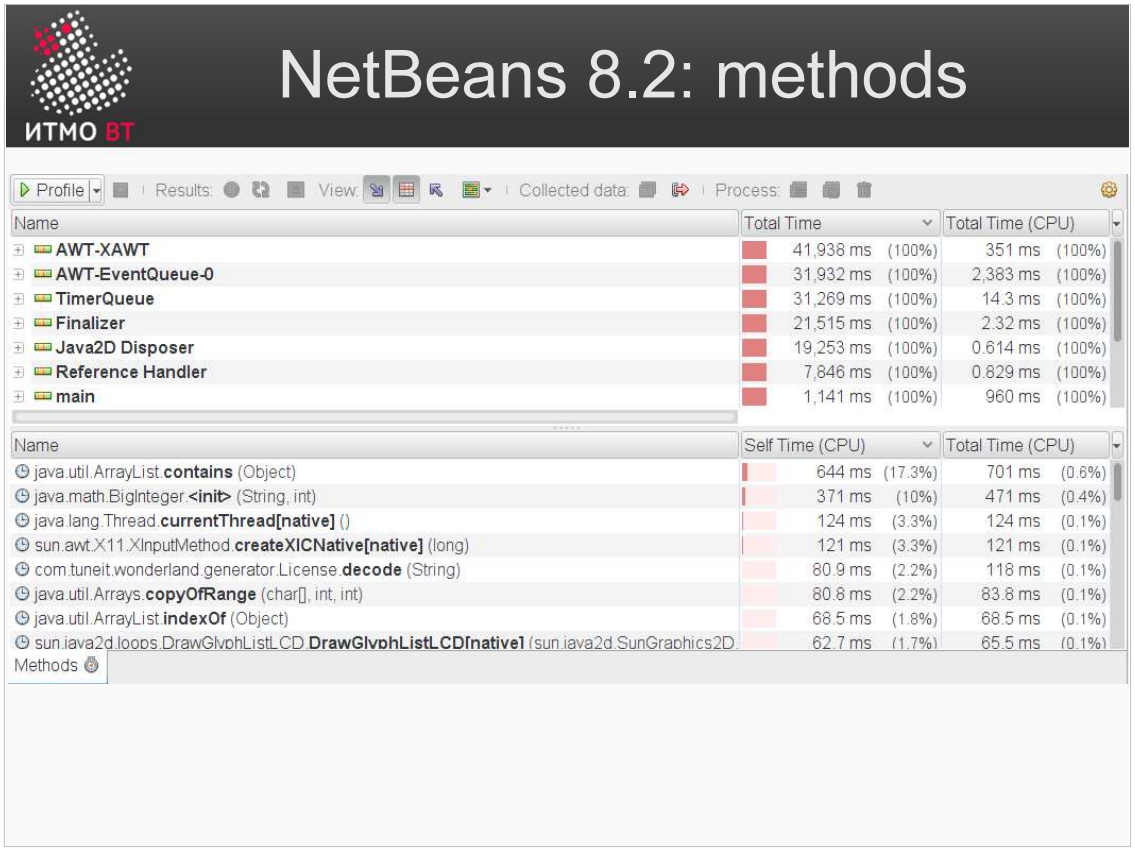
Согласно нисходящей модели поиска узких мест, после общесистемного мониторинга выявляется одно или несколько приложений, которые функционируют неудовлетворительно. Если мы являемся разработчиками этих приложений, и у нас есть исходные коды, то мы можем перейти к исследованию наших приложений для поиска алгоритмических дефектов.

Для этого существуют т.н. профилировщики приложений. С их помощью можно узнать время исполнения функции/метода, объём созданных объектов в памяти, проследить за потоками приложений и борьбой потоков за захват блокировки, временем ожидания ими освобождения блокировок (lock contentions) и др.

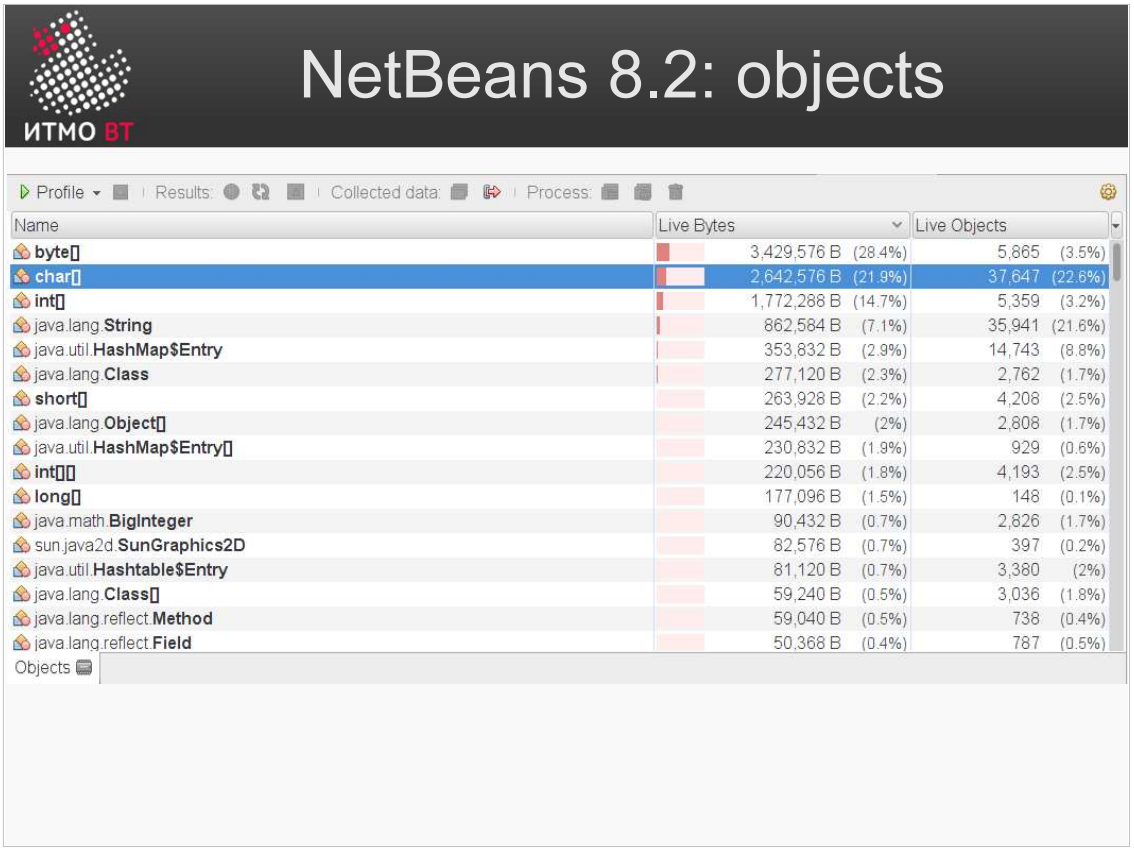
Существуют два основных подхода к профилировке. Согласно первому, диагностические точки можно внедрять в сами функции из указанного набора (например, в начале и в конце функции). Данный способ мониторинга является интрузивным, поэтому следует сначала определить проблемные места, и уже в них ставить диагностические точки.

Второй подход предполагает использование прерываний программы с заданной периодичностью. Профилировщик прерывает работающую программу и собирает интересующую его информацию. Для определения времени исполнения функций используется состояние стека программы в момент прерывания. Для определения характеристик использования памяти собирается информация о куче (heap).

Интервал прерываний выбирается так, чтобы не слишком сильно снижать производительность, и в то же время включить в выбранные данные даже методы с малой продолжительностью работы. Так работают коммерческие профилировщики, например, Performance Analyzer из Oracle Solaris Studio.



На слайде показан снимок экрана профилировщика из NetBeans 8.2. Отображены потоки в составе приложения и методы, занимающие значительное время в CPU. Можно отсортировать методы по времени исполнения, найти "горячий" метод и попытаться провести его оптимизацию.



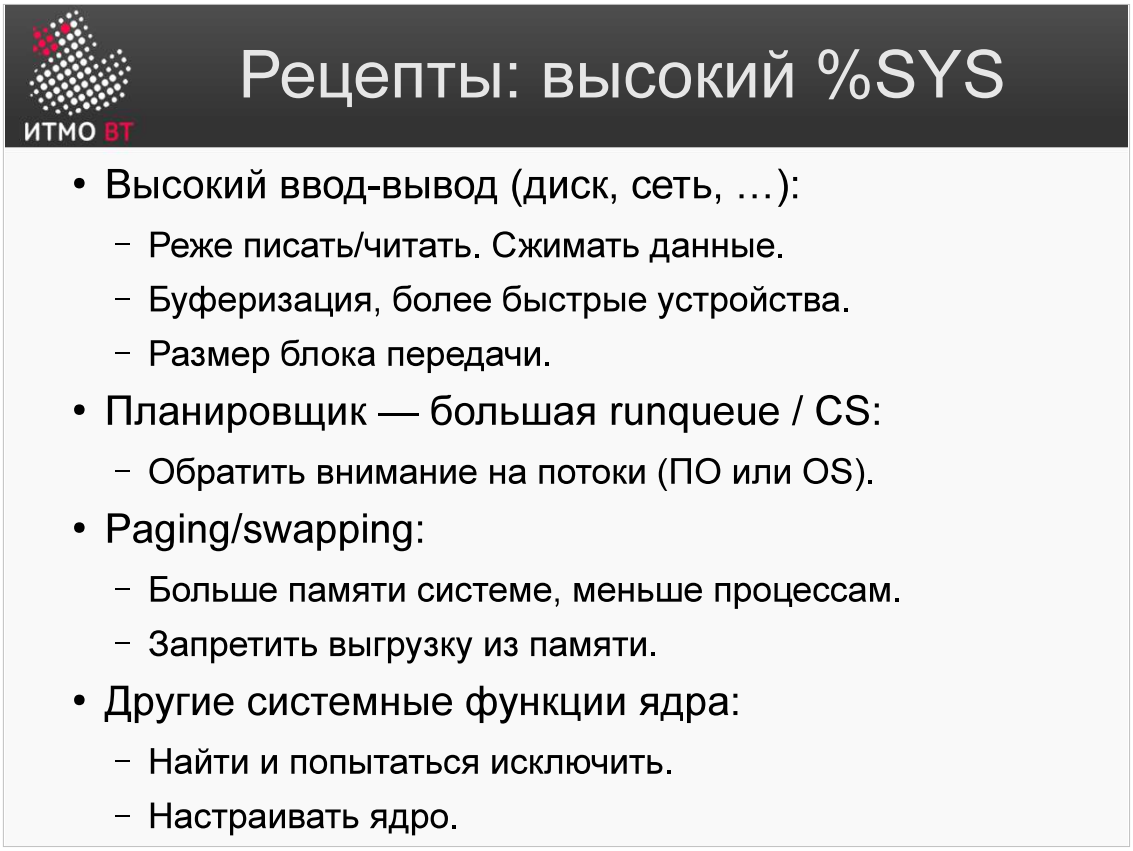
Действия, аналогичные описанным выше, можно сделать и применительно к сбору информации о созданных в программе объектах. В данном примере больше всего места занимают байтовые массивы (byte[]). Если занимаемое место во время наблюдения растет, то имеет место так называемая *утечка памяти* — занятие и неосвобождение ресурсов. В java это означает, что где-то в программе сохраняются ссылки на созданный объект и сборщик мусора не может его уничтожить.

Борьба за производительность сопряжена с компромиссами (trade-offs). Изменения в одном месте (компоненте, методе, алгоритме и т.п.) неизбежно ведут к изменениям в других местах. Например, чем быстрее мы хотим получить доступ к данным, тем больше нам потребуется на это памяти: последовательный поиск в массиве (linear seach) всегда вступает в противоречие в плане скорости/занимаемой памяти с индексированием (indexing), так как для индекса требуется дополнительная память, но его присутствие намного ускоряет поиск.

Таким образом, время, потраченное CPU, связано с тем количеством памяти, которое требует программа: если сделать требования программы скромнее, то она будет требовать больших ресурсов CPU, а программа с большими требованиями к памяти будет работать быстрее за счёт используемых алгоритмов. Однако память тоже является ограниченными ресурсом.

Другим примером компромисса является блочный доступ к диску с кэшированием блоков данных в ядре. Больше кэш — в общем случае, быстрее чтение-запись, но меньше памяти останется для других задач.

Повторимся, грамотный выбор алгоритма и учёт требований архитектуры может ускорить приложение в миллионы раз!



Рассмотрим, что можно изменить в ваших программах, если вы сталкиваетесь с конкретными симптомами проблем производительности вычислительной системы.

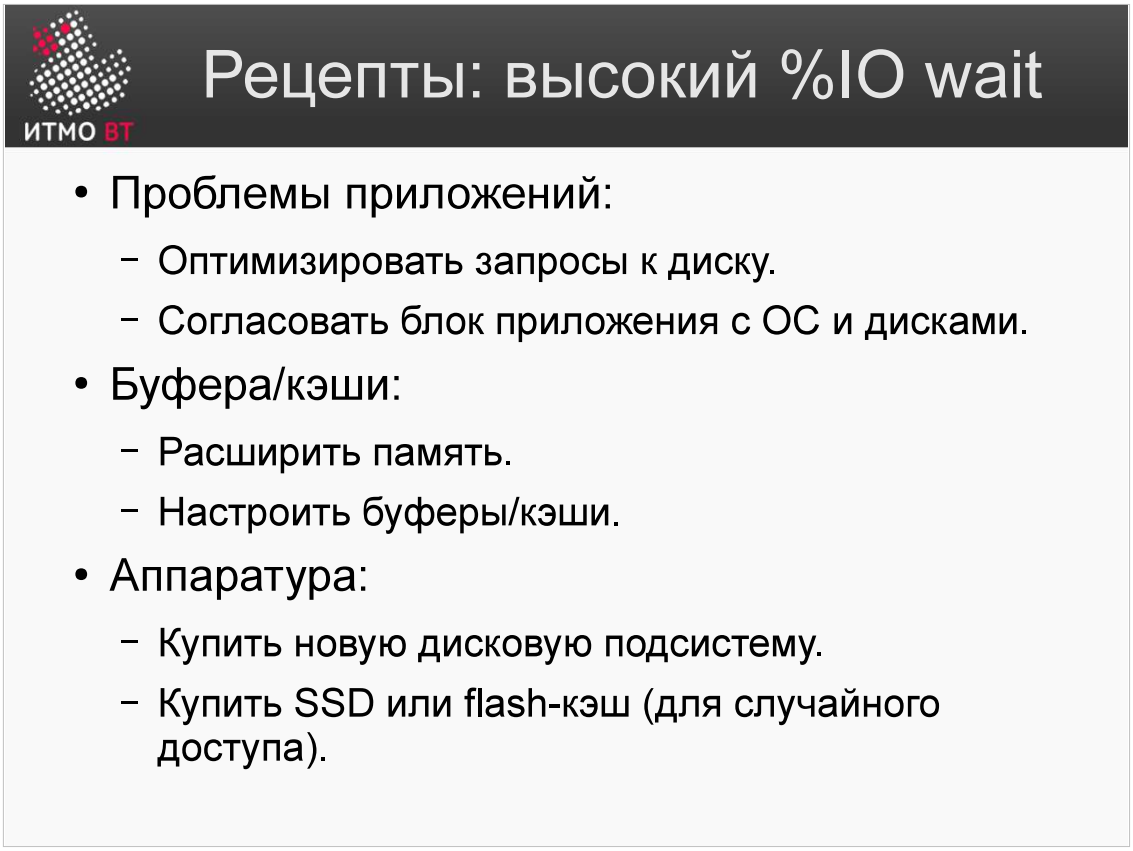
Причин, которые порождают высокую загруженность CPU задачами уровня ядра (высокий %SYS) может быть много. Одни из самых распространённых:

1) Высокая или непродуманная нагрузка на подсистему ввода-вывода. В таком случае нужно реже писать/читать, а также стараться сжимать данные при помощи алгоритмов сжатия, или использовать бинарное представление данных числа вместо строкового. Помогают также буферизация и замена устройств на более быстрые. Можно попытаться синхронизировать размер блока передачи с ОС или устройствами хранения.

2) Недостатки в работе планировщика: чрезмерно частая диспетчеризация, приводящая к избыточному переключению контекстов процессов и т.п. В этом случае нужно проверить, не слишком ли много у вас используется потоков, и что делает в это время ОС.

3) Избыточная подкачка страниц. Если существуют проблемы с виртуальной памятью, то можно выдать больше памяти системе за счёт процессов. Другим решением может быть запрет выгрузки некоторых критически важных процессов из памяти, вследствие чего они не будут подвержены swapping'у.

4) Трата времени процессора в других системных функциях и процессах ядра. В этом случае можно попытаться найти и исключить лишние системные процессы или настроить параметры ядра. Тонкая настройка ядра связана с глубоким пониманием его структуры и анализом большого объёма документации.

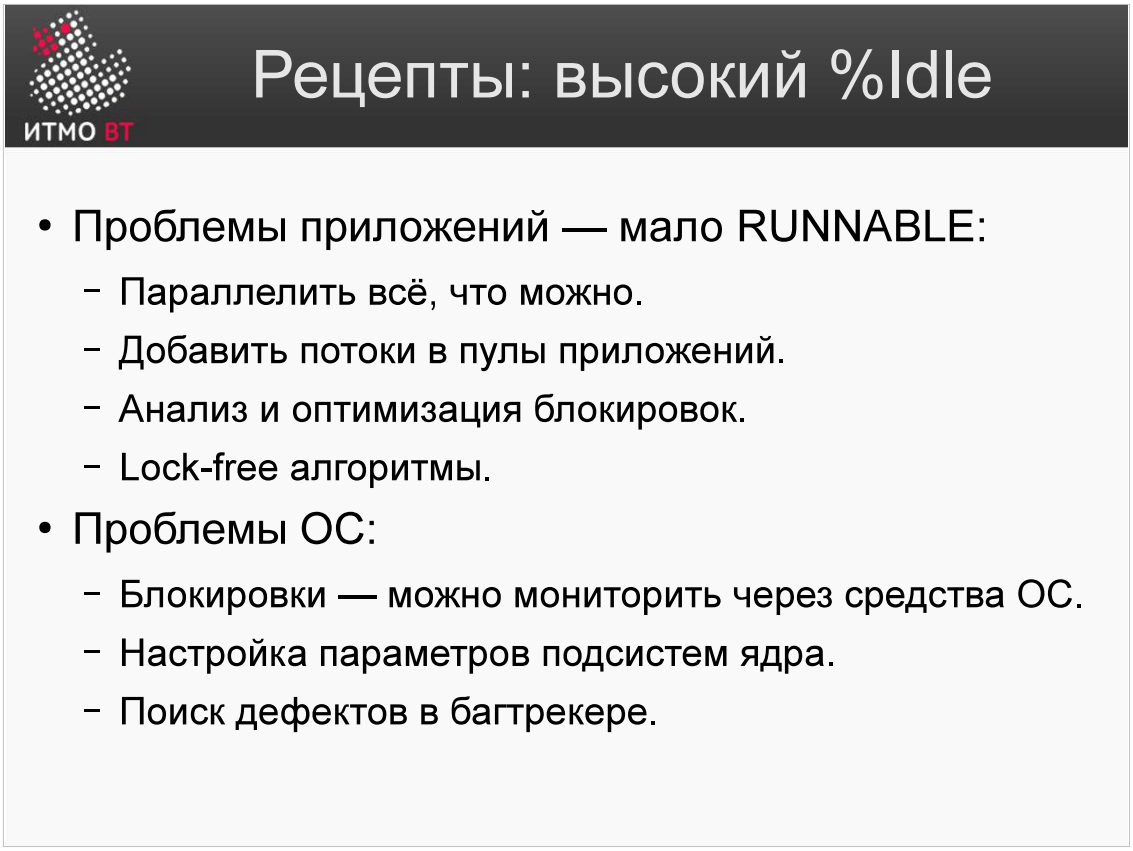


Указанные ниже проблемы порождают высокое время ожидания CPU (высокий %IO wait):

1) Проблемы, связанные с самими приложениями. Как и в первом решении при высоком %SYS, следует оптимизировать запросы к диску: меньше и реже читать/ писать. Если предполагается активный обмен данными, нужно вести обмен большими порциями за одну операцию. Нужно также проверить, согласован ли блок приложения с ОС и дисками: например, если при размере stripe-size в 16 Кб, читать данные блоками по 8 Кб, то происходит, фактически, двойное прочтение, и преимущества RAID-массива сходят на нет.

2) Буфера/кэши, т. е., в системе выделено мало памяти под промежуточное хранение данных. В этом случае можно расширить память, либо настроить использование системных кэшей.

3) Проблемы с аппаратурой. Можно купить новую, более совершенную дисковую подсистему. Для ускорения ввода-выводы очень эффективны SSD или отдельные карты flash памяти, непосредственно устанавливаемые в шину (например PCIe) вычислительной системы. Также повторимся, что такие устройства хорошо подходят для ПО, осуществляющего случайный доступ на диск.



Перечисленные на слайде проблемы проблемы порождают высокое время простоя CPU (высокое %Idle):

1) В системе мало процессов в стадии выполнения.

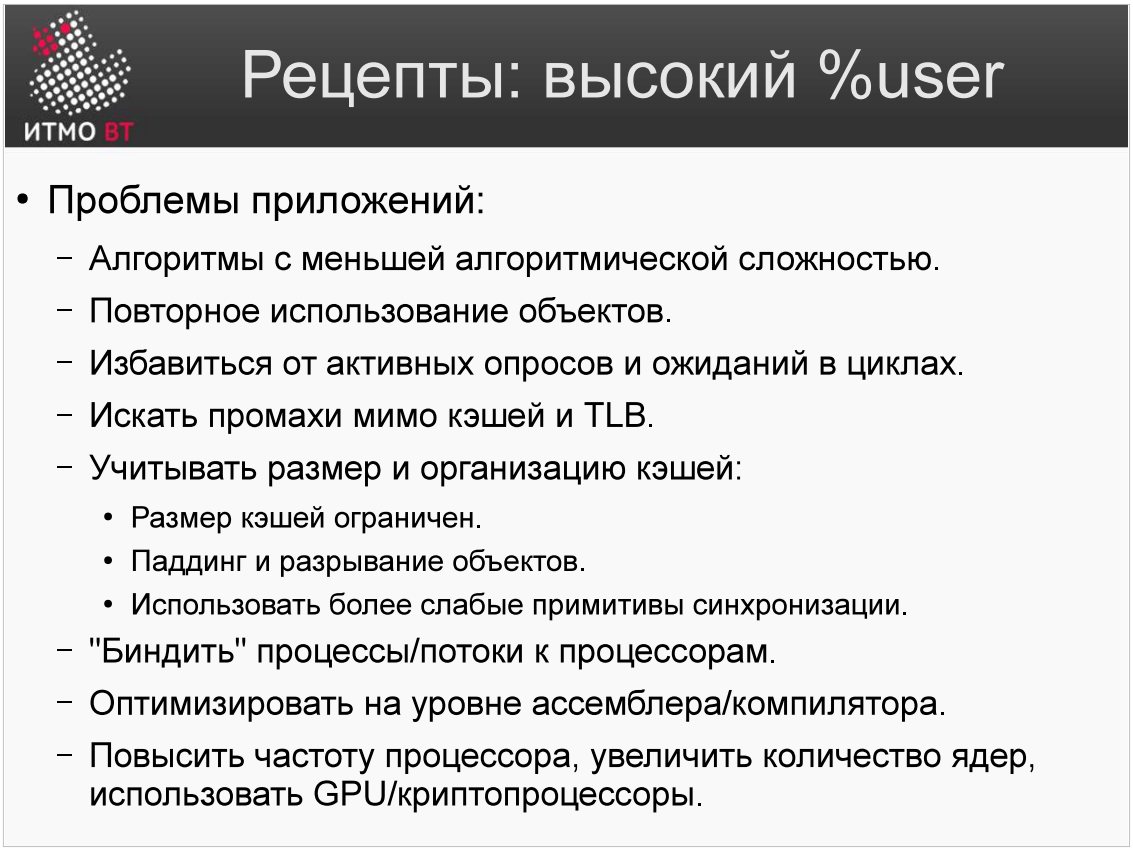
В этом случае может помочь распараллеливание алгоритмов в приложении, особенно, если система многопроцессорная. Если существуют пулы потоков-worker'ов, то в них можно добавить дополнительные потоки для равномерной загрузки всех ядер ЦПУ.

Вторым источником проблем в данном случае могут являться внутренние блокировки в приложении, например, когда много потоков пытаются завладеть одним и тем же участком кода. В этом случае блокировки необходимо оптимизировать. Основной рецепт — держать блокировку как можно меньше времени, а также по возможности использовать более «лёгкие» типы блокировок.

Кроме этого, возможно использование алгоритмов без блокировок (Lock-free).

2) Проблемы могут также корениться в самой ОС.

В большинстве случаев это связано с дефектами в ОС на системных блокировках. Следует посмотреть багтрекер ОС и сопоставить написанное там с симптомами конкретной проблемы. Может помочь также настройка параметров подсистем ядра.



Часто можно встретить ошибочное мнение, что если процессор на 100% загружен, то он эффективно работает, и сделать ничего нельзя. Как в таком случае еще сильнее ускорить работу? Эта задача обычно сложнее (и интереснее) предыдущих.

Для определения узкого места в программе необходимо воспользоваться средствами профилирования. Они помогут найди наиболее затратные функции, объекты с частым созданием/удалением в памяти и другие аномалии вашего ПО. Наиболее действенные рецепты следующие:

1) Использование алгоритмов с меньшей алгоритмической сложностью (помним про баланс и компромиссы — если где-то что-то убыло, значит где-то что-то прибыло).

2) Принцип повторного использования объектов. Крупные объекты и структуры нет необходимости удалять и создавать повторно. Можно взять старый объект и установить в него новые данные.

Этот принцип, например, применяется в UNX для структур процессов. Когда процесс "умирает" (выходит по функции exit() или kill()), он сначала попадает в состояние "зомби". Когда код возврата прочитан родительским процессом, он должен окончательно "умереть" и исчезнуть из структур ядра. Однако, ядро складывает часть "зомби" на "кладбище" (death row), и когда необходимо создать новый процесс, он "восстаёт из мертвых" — его структура повторно используется.

3) На уровне микроархитектуры важно избавляться от кэш-промахов и промахов мимо TLB (translation lookaside buffer, буфер ассоциативной трансляции). Кэш-промахи можно исправить работой над структурами данных (группировка данных для попадания в одну строку кэша, паддинг и др.) Промахи мимо TLB можно исправить использованием больших страниц памяти.

Для исключения "остывания кэшей" можно закреплять потоки за процессорами. ("биндинг" или установка аффинити). Кроме того, биндинг позволит обращаться к локальной памяти процессора при NUMA архитектуре, минимизируя запросы через шину к другой процессорной плате.

Для истинных гиков можно попробовать переписывать части кода на ассемблере и использовать специальные средства аппаратуры — GPU и криптопроцессоры.

Счётчики производительности CPU

- Аппаратные регистры внутри процессора:
  - Позволяют накапливать произошедшие события.
  - Количество регистров и набор событий сильно зависит от процессора:
    - Обычно всегда присутствуют CPU+MMU+TLB.
- Поддерживаются специальными средствами.
  - Linux perf:
    - perf list | grep -i hardware
  - Коммерческие средства разработки.

Для анализа микроархитектуры внутри процессора существуют т.н. счётчики производительности ядра. Это регистры, собирающие данные тех параметров, которые им назначаются (не определенные заранее). Данные регистры являются счётчиками, которые наращиваются самим процессором. Число таких счётчиков ограничено (обычно не более 2-4), поэтому одновременно можно наблюдать только за несколькими параметрами.

В Linux счётчиками управляет команда perf. Используются также коммерческие средства разработки.

Для примера приведём список событий, по которым perf может собирать информацию:

branch-instructions branch-misses cache-misses cache-references cpu-cycles instructions ref-cycles stalled-cycles-backend stalled-cycles-frontend	L1-dcache-load-misses L1-dcache-loads L1-dcache-prefetch-misses L1-dcache-prefetches L1-dcache-store-misses L1-dcache-stores L1-icache-load-misses L1-icache-loads LLC-load-misses LLC-loads LLC-prefetch-misses LLC-prefetches LLC-store-misses LLC-stores	branch-load-misses branch-loads dTLB-load-misses dTLB-loadsdTLB-store-misses dTLB-stores iTLB-load-misses iTLB-loads node-load-misses node-loads node-prefetch-misses node-prefetches node-store-misses node-stores