Лекция 7а. Виды требований к ПО (software requirements)

Требования к ПО

Функциональные и нефункциональные.

Функциональные требования отвечают на вопрос «Что система должна делать»; нефункциональные – «Как она должна это делать».

Если функциональные требования будут соблюдены, то система будет работоспособной, но недружественной к пользователю.

Если функциональные требования не будут соблюдены, то система работать не будет.

Примеры функциональных требований

Способ обработки входных данных

Шаги для аутентификации

События, которые система должна отслеживать

Отчеты, которые система должна предоставлять

Соблюдение авторских прав на код и ресурсы

Описание внешних интерфейсов системы и зависимостей

Примеры нефункциональных требований

Производительность (время отклика, время обработки данных)

Доступность (сколько времени система доступна для пользователя)

Пропускная способность (какое количество информации система способна обработать в единицу времени)

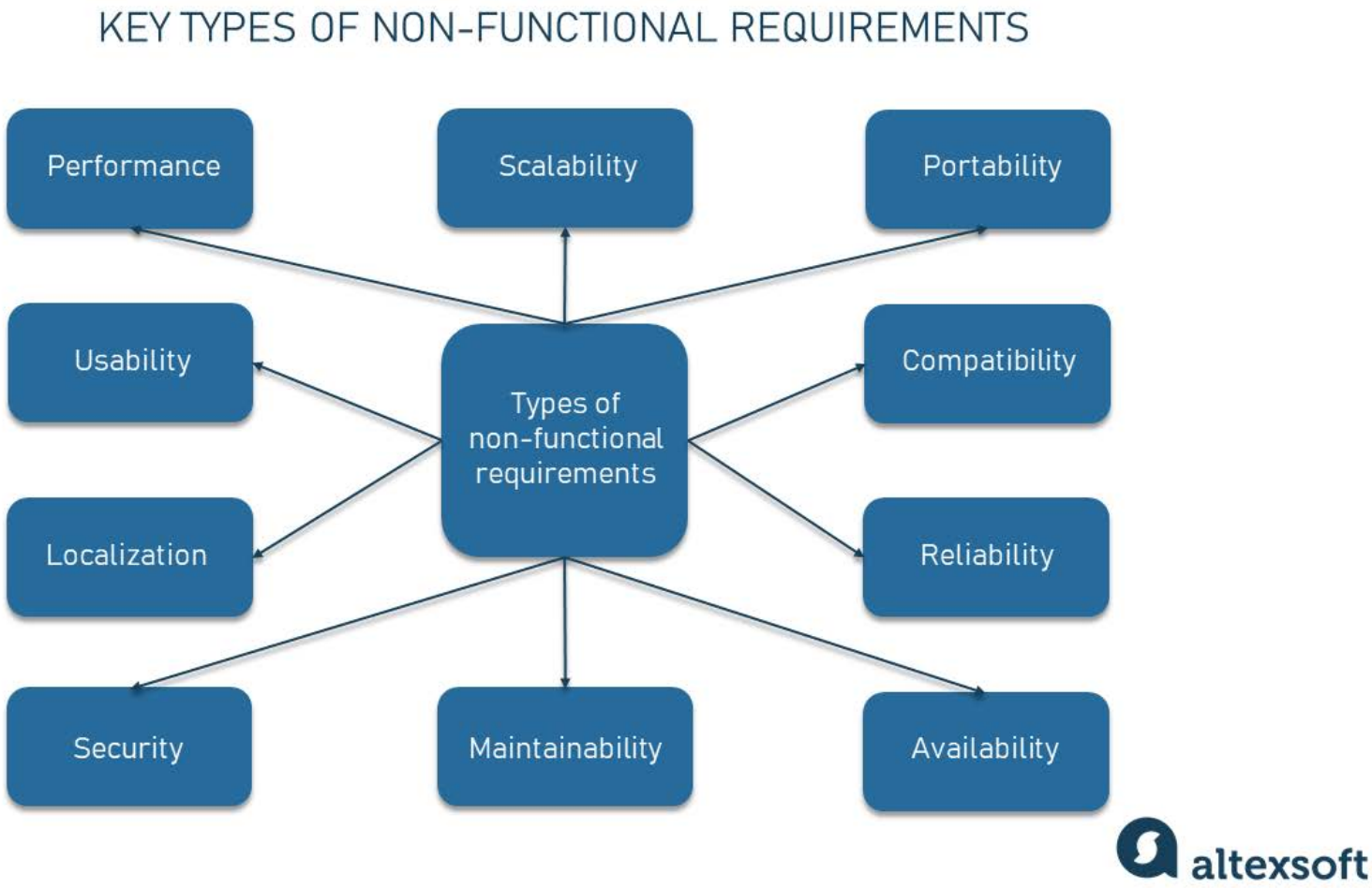
Надежность (количество отказов в единицу времени)

Удобство использования (usability) – к нему могут быть сведены все остальные нефункциональные требования.

Безопасность

Инсталляция

Типы нефункциональных требований



Где описываются функциональные требования

Спецификация (requirements, specification)

Вариант использования (use case), который включает акторов, систему и цели

Прототип (визуальная презентация работы части программы, интерфейса)

Модели и диаграммы (например, для описания взаимодействия функциональных частей программы)

Пользовательская история (user story) – «Как потребитель, я хочу ознакомиться с характеристиками продукта, чтобы решить, стоит ли его покупать»

Иерархическая структура работ (Work Breakdown Structure) – каким образом сложные функции могут быть разбиты на более простые компоненты.

Где описываются нефункциональные требования

Эпик (epic) – большая задача, на которую нужно несколько спринтов

Возможность, функция программы (feature)

Пользовательская история (user story)

Критерий приемки (acceptance criteria)

Определение критериев окончания работ (Definition of Done)

Требования к продукту (product backlog)

Лекция 7б. Критерии оценки требований ПО (software requirements)

Критерии оценки требований

корректность;

недвусмысленность;

полнота;

непротиворечивость;

упорядоченность по важности и стабильности;

проверяемость;

модифицируемость;

трассируемость

Критерии оценки требований

Корректные требования. Набор требований к программному обеспечению является корректным тогда и только тогда, когда каждое требование, сформулированное в нем, представляет нечто, требуемое от создаваемой системы.

Недвусмысленные требования. Требование является недвусмысленным тогда и только тогда, когда его можно однозначно интерпретировать. Хотя главным свойством любого требования по праву считается корректность, неоднозначность зачастую представляет собой более сложную проблему. Если формулировка требований может по-разному интерпретироваться разработчиками, пользователями и другими участниками проекта, вполне может оказаться, что построенная система будет полностью отличаться от того, что представлял себе пользователь.

Критерии оценки требований

Полнота набора требований. Набор требований является полным тогда и только тогда, когда он описывает все важные требования, интересующие пользователя, в том числе требования, связанные с функциональными возможностями, производительностью, ограничениями проектирования, атрибутами или внешними интерфейсами. Полный набор требований должен также задавать требуемый ответ программы на всевозможные классы ввода — как правильные, так и неправильные — во всевозможных ситуациях. Помимо этого, он должен содержать полные ссылки и подписи всех рисунков, таблиц и диаграмм набора требований, а также определения всех терминов и единиц измерения.

Критерии оценки требований

Непротиворечивость набора требований. Множество требований является внутренне непротиворечивым, когда ни одно его подмножество, состоящее из отдельных требований, не противоречит другим подмножествам. Конфликты могут иметь различную форму и проявляться на различных уровнях детализации; если набор требований был написан достаточно формально и поддерживается соответствующими автоматическими средствами, конфликт иногда удается обнаружить посредством механического анализа. Но, скорее всего, разработчикам вместе с другими участниками проекта придется провести проверку множества требований вручную, чтобы удалить все потенциальные конфликты.

Критерии оценки требований

Упорядочение требований по их важности и стабильности. В высококачественном наборе требований разработчики, клиенты и другие заинтересованные лица упорядочивают отдельные требования по их важности для клиента и стабильности. Этот процесс упорядочения особенно важен для управления масштабом. Если ресурсы недостаточны, чтобы в пределах выделенного времени и бюджета реализовать все требования, очень полезно знать, какие требования являются не столь уж обязательными, а какие пользователь считает критическими.

Проверяемые требования. Требование в целом является проверяемым, когда каждое из составляющих его элементарных требований является проверяемы, т.е. когда можно протестировать каждое из них и выяснить, действительно ли они выполняются.

Критерии оценки требований

Модифицируемый набор требований. Множество требований является модифицируемым, когда его структура и стиль таковы, что любое изменение требований можно произвести просто, полно и согласованно, не нарушая существующей структуры и стиля всего множества. Для этого требуется, чтобы пакет требований имел минимальную избыточность и был хорошо организован, с соответствующим содержанием, индексом и возможностью перекрестных ссылок.

Критерии оценки требований

Трассируемые требования. Требование в целом является трассируемым, когда ясно происхождение каждого из составляющих его элементарных требований и существует механизм, который делает возможным обращение к этому требованию при дальнейших действиях по разработке. На практике это обычно означает, что каждое требование имеет уникальный номер или идентификатор. Возможность трассировки имеет огромное значение. Разработчики могут использовать ее как для достижения лучшего понимания проекта, так и для обеспечения более высокой степени уверенности, что все требования выполняются данной реализацией.

Существуют различные методы тестирования требований:
1. **Метод просмотра** (универсальный метод, выполняется бизнес-аналитиком или тестировщиком): - Ознакомление с требованиями. - Проверка требований по критериям качества. -Оформление дефектов. - Оформление отчета.

2. **Метод экспертизы** (выполняется при участии команды из бизнес-аналитиков, представителей заказчика, разработчиков, лояльных пользователей, тестировщиков): - Планирование. - Обзорная встреча. - Подготовка. - Совещание. - Переработка. - Завершающий этап.

3. **Метод составления вариантов тестирования** (выполняется тестировщиком). Варианты тестирования занимают промежуточную позицию между Use Case и Test Case, помимо использования для тестирования требований в дальнейшем легко расширяются до Test Cases и составляют основу тестовой документации.

Лекция 7в. Баг трекинговые системы

Система отслеживания ошибок (англ. bug tracking system) — **прикладная программа, разработанная с целью помочь разработчикам программного обеспечения** (программистам, тестировщикам и др.) учитывать и контролировать ошибки и неполадки, найденные в программах, пожелания пользователей, а также следить за процессом устранения этих ошибок и выполнения или невыполнения пожеланий.

Среднестатистическая баг-трекинговая система имеет следующий функционал:

- создание тикетов с подробным описанием багов
- классификация и расстановка приоритетов в тикетах
- назначение тикетов конкретным специалистам
- отслеживание статуса бага на разных этапах
- удобный поиск, сортировка и составление баг-репортов
- аналитика и автоматическое формирование репортов.

- **Jira**

Изначально Jira предназначалась только для отслеживания ошибок. Однако сейчас она также используется для планирования agile-проектов.

Jira представляет собой интерактивную доску (Дашборд), с помощью которой можно следить за выполнением поставленных задач. Все задачи классифицируются различными видами функций, подзадач, багов и т.д. Они могут редактироваться, назначаться на различных исполнителей или просто изменять статус с «открыт» на «закрыт». Все изменения по задаче записываются в журнал.

- **Redmine**

Это бесплатное веб-приложение. И это больше, чем просто трекер ошибок. Redmine — решение для управления проектами с открытым исходным кодом. Написан на Ruby и совместим с MySQL, PostgreSQL, Microsoft SQL и SQLite.

Баг-репорт может отслеживаться любым сотрудником, который добавлен в проект и отмечен как наблюдатель.

- **Mantis**

Бесплатный инструмент. По сравнению с другими баг-трекинговыми системами, это довольно простой инструмент. Он доступен как в виде web-приложения, так и в мобильной версии. Баг-репорт можно назначить на любого пользователя, который работает в проекте.

Инструмент построен на PHP и совместим с базами данных MySQL и PostgreSQL. Его также можно настроить для управления проектами.

- **Яндекс.Трекер**

Сервис для управления проектами по методологии Agile. Платный, но предоставляется бесплатный доступ на 30 дней.

Пользователи могут создавать задачи, описывать их, назначать исполнителей и наблюдателей, а также комментировать ход решения вопроса в карточке задачи. Гибкие настройки прав доступа. Высокая производительность.

- **BUGZILLA**

Bugzilla — одна из самых известных программ для отслеживания ошибок с открытым исходным кодом. Эта программа была представлена Mozilla еще в 1998 году.

Bugzilla ориентирована исключительно на отслеживание ошибок и предлагает все инструменты и функции, необходимые для этого процесса. Программа широко используется небольшими компаниями и корпорациями.

Youtrack

Это багтрекинговая система от Jetbrains, которая делает акцент на разработке программного обеспечения.

У YouTrack очень простой интерфейс — все задачи по проектам размещаются на Agile-досках (как в Trello). Также YouTrack интегрируется с другими продуктами JetBrains (Space, вся линейка IDE). Для отслеживания задач есть отдельное мобильное приложение для iIOS и Android.

Стоимость: до 10 пользователей — бесплатно

Лекция 7г. Система контроля версий (Git, Github)

Git. Что такое система контроля версий?

Система контроля версий (англ. Version Control System) - программное обеспечение хранящееся все версии возможного файла, и дающее возможность получить к ним доступ. Существуют разные системы: git, mercurial, subversion(svn), Team Foundation server (TFS), однако наибольшую популярность завоевала система Git из-за простоты использования и внедрения в другие системы.

- **Что было до внедрения систем контроля версий и как они используются сейчас.**

До распространения VCS параллельная разработка велась при помощи простых архивов кодов, которые разработчики передавали между собой, а если требовалось версионирование - оно создавалось при помощи различных наименований файлов. При этом итоговая сборка проекта была сильно затруднена, поскольку никто из разработчиков по сути не знал над чем и как работает его коллега, и не был уверен в совместимости кода. Поэтому на итоговую компиляцию и исправление ошибок уходило значительное время.

В текущих реалиях системы контроля версий являются важнейшей частью процесса Continuous integration, поскольку позволяют производить одновременную разработку, через доступное для всех разработчиков версионирование и через удалённый сервер со всеми версиями. Таким образом любой из разработчиков может получить последнюю версию рабочего кода, соединить её со своей и проверить работоспособность. Также в данный процесс тесно внедрены системы тестирования. Так, в некоторых случаях буквально на каждый коммит могут запускаться юнит-тесты и интеграционное тестирование на CI сервере, которое в автоматическом режим проверит работоспособность и совместимость кода.

- Основные понятия **Git**
- Репозиторий - обособленное хранилище кода внутри которого будут отслеживаться изменения файлов
- Ветвь (branch) - отдельная цепочка (ветка) отслеживаемых изменений
- Мастер branch - главная ветка в репозитории
- Коммит - зафиксированная точка изменений
- Удаленный сервер (remote server) - сервер на котором хранится репозиторий
- Merge (слияние) - процесс слияния двух ветвей
- tag - ссылка на определенный коммит

- **Git** - это утилита для работы в командной строке (хотя есть и опции для работы через графический интерфейс)
- Основные команды:
- **git pull** - забрать изменения с удалённого сервера
- **git push** - отправить локальные изменения на удалённый сервер
- **git branch** - показать текущую ветку/список веток
- **git merge** <branch_name> - влить в текущую ветку указанную
- **git add** - добавить в текущее состояние изменённые файлы
- **git commit** - зафиксировать изменения и сделать коммит
- **git stash** - сохранить локальные изменения не создавая коммит и откатиться до состояния удалённого сервера
- **git checkout** - перемещать указатель HEAD, т.е. то куда смотрит ваша локальная копия. Вы можете переместить его на вершину ветки: git checkout <branch> или на отдельный коммит: git checkout <sha>. Вспомогательные это создание веток:git checkout -b, отмена изменений в файле:git checkout -- <file>

- **GitFlow** - это методология работы с **Git** или, проще говоря, модель ветвления.

В разработке существуют различные стратегии ветвления для того чтобы все осуществлялось параллельно и без конфликтов.

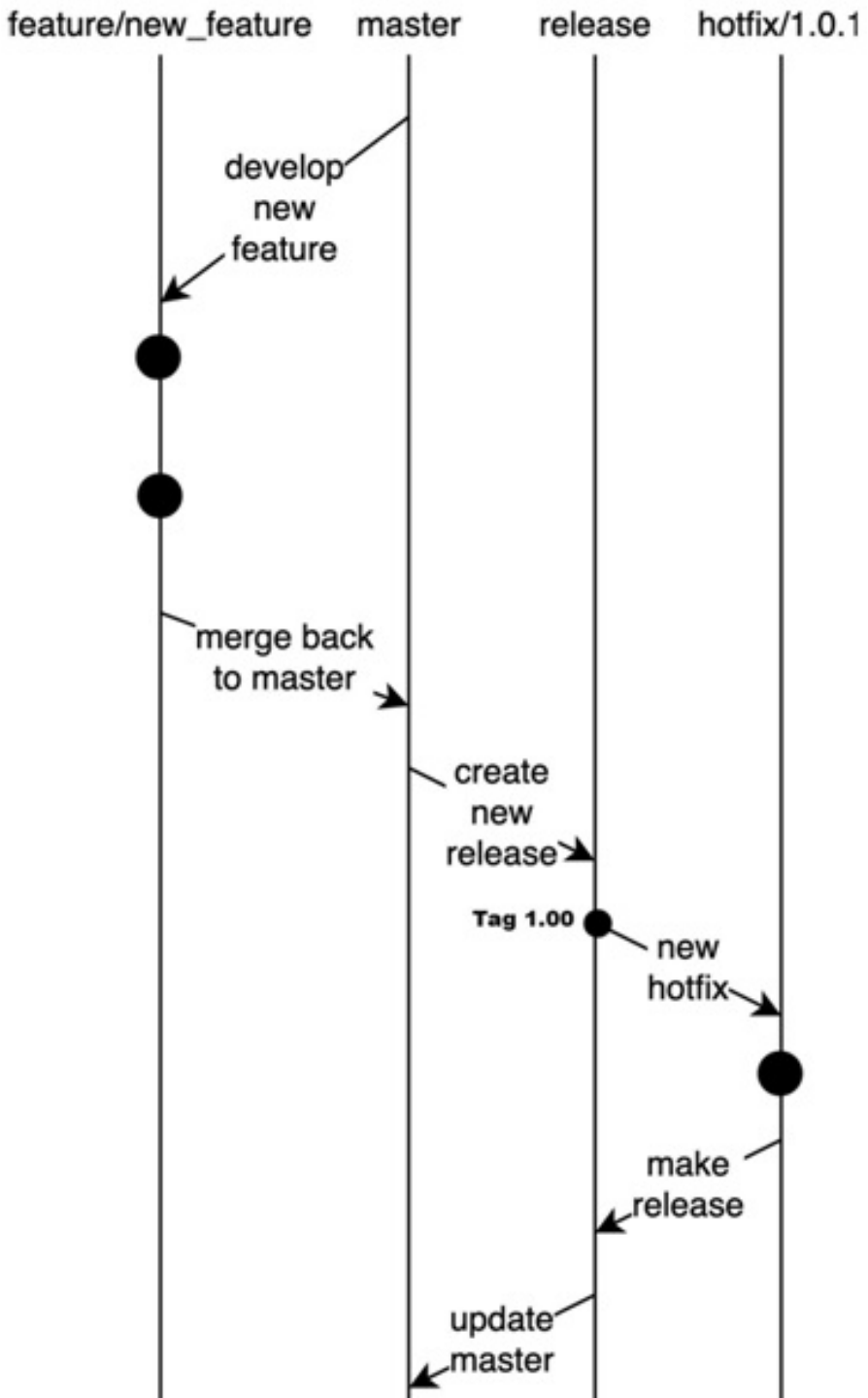
Одна из самых популярных стратегий представляет собой достаточно простую концепцию:

Существует master ветка в которой всегда находится рабочий код

Из нее в нужные моменты создаются релиз ветки, из которых собирается результирующий артефакт.

Для разработки нового компонента используется feature-ветки, которые создаются из мастера, проходят стадии разработки и затем вливаются обратно в мастер. (Опционально master ветка может быть заменена на dev ветку в которой собирается весь разрабатываемый код, и которая в свою очередь перед релизом вливается в master ветку)

Для создания срочных патчей создаются Ehotfix-ветки из релиза, разрабатываются и вливаются обратно в релиз, после чего релиз ветка вливается в мастер ветку.


feature/new_feature
master
release
hotfix/1.0.1
develop
new
feature
merge back
to master
create
new
release
Tag 1.00
new
hotfix
make
release
update
master