Выпускная квалификационная работа на тему «Разработка модуля системы CSIA для мониторинга описания работ по разработке ПО в системах контроля версий и управления проектами»

Исходные данные к работе

1. ГОСТ 34.602.89 «Техническое задание на создание автоматизированной системы»
2. СМК-О-СМГТУ-36-20 «Выпускная квалификационная работа: структура, содержание, общие правила выполнения и оформления»
3. Документация к системе RadixWare компании ООО «Компас Плюс»
4. Структура системы контроля версий «Subversion»
5. Структура системы контроля версий «GitTea»
6. Структура системы управления проектами «Jira»

Перечень вопросов, подлежащих разработке в выпускной квалификационной работе:

1. Анализ требований, предъявляемых к разрабатываемой системе
2. Изучение систем контроля версий и систем управления проектами
3. Разработка проектного решения модуля на базе системы «CSIA»
4. Разработка модуля на базе системы «CSIA»

Графическая часть: подготовка электронной презентации выпускной квалификационной работы средствами ...

Слайд 1 - ...

...

**Реферат**

Тема выпускной квалификационной работы: «Разработка модуля системы CSIA для мониторинга описания работ по разработке ПО в системах контроля версий и управления проектами».

Ключевые слова: СИСТЕМА КОНТОРЯ ВЕРСИЙ, СИСТЕМА УПРАВЛЕНИЯ ПРОЕКТАМИ, ...

Пояснительная записка содержит n страниц без приложения, n иллюстраций, n информационных источников.

Объектом исследования в данной выпускной квалификационной работе является процесс разработки программного обеспечения. Предмет исследования – описание работы в системах контроля версий и управления проектами.

Цель данной работы – разработка модуля системы CSIA для мониторинга описания работ по разработке ПО в системах контроля версий и управления проектами.

В пояснительной записке изучены особенности работы систем контроля версий и управления проектами. Рассмотрены проектные решения по разработке модуля на базе системы CSIA. ...

Отзыв

Содержание

[Введение 8](#_Toc1)

[1 Особенности разработки программных продуктов в ООО «Компас плюс» 9](#_Toc2)

[1.1 Общие характеристики коммерческих продуктов компании 9](#_Toc3)

[1.1.1 Общая архитектура продуктов 9](#_Toc4)

[1.1.2 Хостовые приложения 12](#_Toc5)

[1.1.3 Front-end приложения 13](#_Toc6)

[1.2 Системы используемые при разработке ПО 14](#_Toc7)

[1.2.1 Система контроля версий Subversion 14](#_Toc8)

[1.2.2 Система контроля версий Git 17](#_Toc9)

[1.2.3 Система управления проектами Jira 19](#_Toc10)

[1.3 Нормативы описания разработок 21](#_Toc11)

[1.3.1 Формат описания разработок в системах контроля версий 21](#_Toc12)

[1.3.2 Регламенты описания задач в системах управления проектами 25](#_Toc13)

[1.4 Контроль качества продукта 30](#_Toc14)

[1.4 Выводы к первой главе 31](#_Toc15)

[2 Проектные решения разработки модуля системы csia для мониторинга описания работ по разработке по в системах контроля версий и управления проектами 33](#_Toc16)

[2.1 Общие сведения о системе CSIA 33](#_Toc17)

[2.1.1 Система CSIA 33](#_Toc18)

[2.1.2 Платформа RadixWare 34](#_Toc19)

[2.2 Требования к разрабатываемому модулю 36](#_Toc20)

[2.3 Организация взаимодействия с системами контроля версий 37](#_Toc21)

[2.3.1 Общий механизм работы с системами контроля версий 37](#_Toc22)

[2.3.2 Взаимодействие с Subversion 43](#_Toc23)

[2.3.3 Взаимодействие с Git 44](#_Toc24)

[2.4 Взаимодействие с Jira 48](#_Toc25)

[2.5 Мониторинг нарушений в проекте 49](#_Toc26)

[2.5.1 Структурная модель 49](#_Toc27)

[2.5.2 Алгоритм поиска нарушений 52](#_Toc28)

[2.6 Генерация отчета 52](#_Toc29)

[2.6.1 Работа со статистикой нарушений 52](#_Toc30)

[2.6.2 Алгоритм генерации отчета 53](#_Toc31)

[2.7 Выводы ко второй главе 53](#_Toc32)

[3 Результаты опытной эксплуатации модуля системы csia для мониторинга описания работ по разработке по в системах контроля версий и управления проектами 54](#_Toc33)

[3.1 Работа с разработанными сущностями 54](#_Toc34)

[3.1.1 Сущность проекта 54](#_Toc35)

[3.1.2 Сущность репозитория 54](#_Toc36)

[3.1.3 Сущности системы управления проектами 54](#_Toc37)

[3.2 Создание и настройка задачи для мониторинга 54](#_Toc38)

[3.2.1 Задача Task.CheckProjectViolations 54](#_Toc39)

[3.2.2 Настройка расписания 54](#_Toc40)

[3.3 Пример сгенерированного отчёта 54](#_Toc41)

[3.4 Выводы к третьей главе 54](#_Toc42)

[Заключение 55](#_Toc43)

[Список используемых источников 56](#_Toc44)

[Приложение А 57](#_Toc45)

# Введение

В настоящее время программное обеспечение ...

# **1 Особенности разработки программных продуктов в ООО «Компас плюс»**

## **1.1 Общие характеристики коммерческих продуктов компании**

### 1.1.1 Общая архитектура продуктов

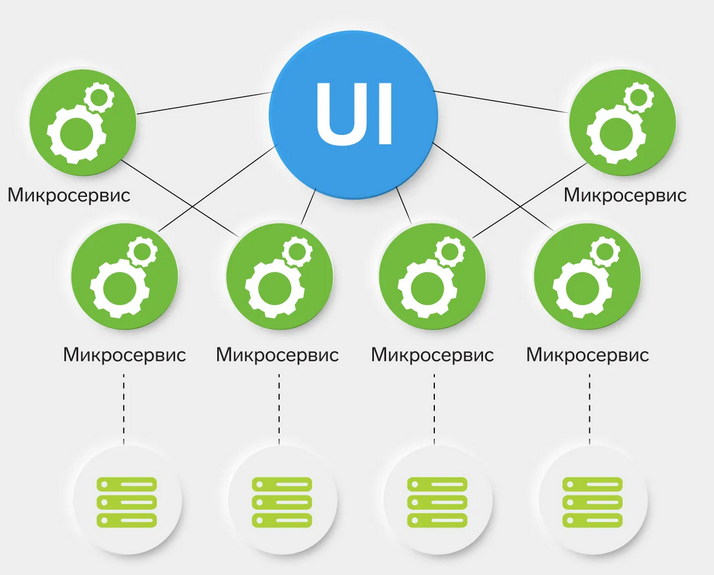
Специфика коммерческих продуктов компании предполагает большой набор критериев, которым должны соответствовать разрабатываемые информационные системы. Из основных выделяются следующие критерии:

1. Отказоустойчивость – системы, работающие в реальном времени и выполняющие услуги по денежному переводу, что является одной из основных задач продуктов компании, должны работать с минимальным количеством сбоев. Клиент должен быть уверен, что его перевод совершится, либо все связанные с переводом сущности вернутся в исходное состояние, иными словами должна обеспечиваться транзакционная целостность. Помимо этого, выход из строя одного из компонентов системы не должен приводить к неработоспособности всей системы в целом;
2. Масштабируемость – информационная система должная обладать свойствами горизонтального масштабирования. Это необходимо для распределения высокой нагрузки на систему. Сервисы системы, физически, могут находиться на различных вычислительных устройствах и использовать максимальное количество доступных им ресурсов, что увеличивает скорость выполнения запроса;
3. Эффективное выполнение задач – скорость выполнения запроса очень важна. Это улучшает пользовательский опыт взаимодействия с системой и позволяет обрабатывать большее количество запросов в единицу времени. Очевидно, что любая задача может быть выполнена на любом современном языке программирования. Однако это вовсе не означает что такая задача будет выполнена эффективно. Поэтому существует необходимость создания продукта, состоящего из ряда систем, написанных на языках программирования, которые наиболее эффективно выполняют ту или иную задачу;
4. Поддержка кодовой базы – современные программные продукты имеют весьма большую кодовую базу, которая легко может содержать несколько сотен тысяч строк кода. Поддержка таких продуктов достаточно сложная задача. Для упрощения процесса поддержки необходимо разделять части продукта друг от друга для того, чтобы иметь четкое представление о том, что выполняет каждый элемент разрабатываемой системы.

Одной из архитектур, хорошо подходящих под данные критерии является микросервисная архитектура.

Микросервисная архитектура – архитектура программного продукта, основанная на создании небольших, автономных, совместно работающих сервисах [1].

На рисунке 1 представлен пример микросервисной архитектуры. На нем видно, что существует система, предоставляющая графический пользовательский интерфейс (UI). Выполняя какой-либо запрос пользователя обращение не поступает только к одной системе, а в зависимости от поставленной задачи используется определенный микросервис. В зависимости от необходимости, микросервисы могут обращаться друг к другу, либо, например, иметь подключение к базе данных.



*Рисунок 1 – Пример микросервисной архитектуры*

Выход из строя какого-либо элемента в этой сети не приводит к прекращению работы всего продукта, так как каждый микросервис является независимой системой, работоспособность которой не влияет на все остальные элементы. Выход из строя микросервиса может означать лишь прекращение работы определенного функционала, но никак не всего продукта.

Масштабируемость может достигается с помощью клонирования микросервиса, что приводит к распределению нагрузки и повышению скорости работы всего продукта.

Каждый микросервис может быть написан на любом языке программирования с использованием любой технологии, подходящей для конкретной задачи которая ставится перед микросервисом. Это, как уже было сказано, также благоприятно сказывается на скорости работы продукта. Помимо этого, такой подход позволяет поделить разработку продукта между несколькими, независимыми друг от друга командами, что в свою очередь повышает скорость разработки продукта, а также его качество, так как решаемые микросервисом задачи ограничены и относятся к одной области, что позволяет сфокусироваться только на их решении.

При таком подходе, деление продукта на компоненты, имеющие ограниченную область применения, происходит само собой. Каждый микросервис проектируется и разрабатывается с ограниченной функциональностью и не может заменять другой микросервис. Это позволяет разработчику достаточно быстро разобраться в структуре работы данного сервиса и внести необходимые изменения.

Таким образом, микросервисная архитектура является наиболее подходящим архитектурным паттерном для разработки высоконагруженых программных продуктов и успешно используется на протяжении нескольких лет разработки в компании «Компас плюс».

### ***1.1.2 Хостовые приложения***

Основой любого коммерческого продукта, разрабатываемого в компании «Компас плюс», является хостовое приложение.

Хостовое приложение – программное обеспечение, выполняющее основные методы разрабатываемого продукта.

В зависимости от проекта, хостовое приложение может иметь различную функциональность, архитектуру и окружение. В данном случае, под окружением понимается ряд дополнительных систем, разработанных, как в компании, так и вне её, которые в совокупности с хостовым приложением и формируют готовый программный продукт.

В качестве основы хостового приложения используется одна из двух систем. FloraWare и RadixWare.

FloraWare представляет собой технологическую платформу, предназначенную для разработки и исполнения программных приложений и систем.

Технология FloraWare используется при проектировании различных прикладных систем. При этом системы могут разрабатываться как на основе самой технологии FloraWare, так и на базе ее приложений - в частности, Data Base Project (DBP).

Среда исполнения приложений FloraWare позволяет выполнять приложения и программные модули, реализованные на базе FloraWare.

FloraWare является кросс-платформенной средой, которая может работать в ОС семейств Microsoft Windows и UNIX (Linux, AIX).

Со временем, на смену FloraWare пришла система RadixWare. Данная технологическая платформа предназначена для разработки прикладных систем корпоративного уровня. Применение RadixWare наиболее эффективно при разработке следующих разновидностей приложений:

1. Транзакционные системы;
2. Системы массового обслуживания;
3. Системы высокого уровня доступности и устойчивости с боям;
4. Системы с трехзвенной архитектурой: SQL БД, сервер приложения, клиент приложения;
5. Системы на базе сервис-ориентированной архитектуры (SOA);
6. Системы управления бизнес процессами [5].

### 1.1.3 Front-end приложения

Помимо хостового приложения, проект может состоять из ряда дополнительных систем, называемыми Front-end приложениями.

Front-end приложения – программные продукты, отвечающие за взаимодействие пользователя с системой. Данные продукты являются прослойкой между пользователем и хостовой системой. Они, с помощью пользовательского интерфейса, позволяют пользователю передавать информацию к хосту и выполнять различные операции.

В качестве основы взаимодействия с пользователем может выступать как веб-страница (в качестве примера таких сервисов могут выступать Merchat portal, HPP, Internet bank), так и мобильное приложение (Mobicash, Pocket bank).

Однако нередки случаи, когда системы, которые непосредственно предоставляют взаимодействие с пользователем являются лишь частью от общего числа Front-end приложений. К Front-end приложениям можно также отнести и вспомогательные системы для кэширования данных, системы хранения статических ресурсов, системы маршрутизации запросов с возможностью конвертации протокола взаимодействия. Все эти системы могут быть написаны с помощью различных технологий и языков программирования, выделим основные из них:

1. NodeJS – асинхронная среда выполнения JavaScript кода, предназначенная для создания масштабируемых сетевых приложений;
2. React – декларативная JavaScript библиотека для создания пользовательских интерфейсов;
3. Android – программный стек на базе Linux с открытым исходным кодом, созданный для широкого спектра устройств и форм-факторов;
4. Swift – язык программирования, предназначенный для создания приложений, работающих на операционной система iOS;
5. Spring – фреймворк для языка программирования Java, предназначенный для разработки веб-сервисов.

## 1.2 Системы используемые при разработке ПО

### 1.2.1 Система контроля версий Subversion

В настоящее время, разрабатываемые системы имеют огромную сложность и объемы разработки. Проектирование и реализация такой системы одним человеком просто невозможна, так как это займет большое количество времени и сильно снизит качество конечного продукта. Поэтому над разработкой систем большой сложности, зачастую, трудится команда из нескольких разработчиков.

В свою очередь, разработку в команде трудно представить без какой-либо системы контроля версий. Это обуславливается тем, что все разработчики должны иметь доступ к исходному коду. Должно осуществляться логирование, версионирование и хранение всех внесенных изменений. Одной из таких систем контроля версий является Subversion.

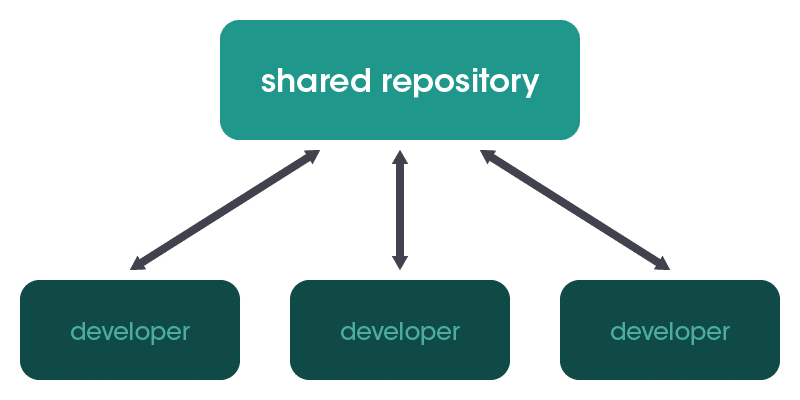
Subverison (SVN) – система контроля версий с открытым исходным кодом. Она позволяет организовать управление каталогами и файлами, а также изменениями, внесенными с течением времени. Это даёт возможность восстановить ранние версии данных и изучить историю всех изменений [2].

В 2004 году, данная система пришла как альтернатива существующей на тот момент CVS. Изначально, SVN создавался для того, чтобы исправить основные недостатки CVS, вместе с тем, сохранив высокую совместимость между этими системами.

К основным отличиям SVN от CSV можно отнести следующее:

1. Контроль изменений каталогов. CSV имеет возможность отслеживания только файлов, в то время как SVN имеет свою собственную «виртуальную» файловую систему, что позволяет этой системе отслеживать изменения каталогов;
2. Настоящая история версий. CVS позволяет контролировать только изменения в файлах. Изменений названий и копирование файлов не поддерживается. SVN же дает возможность копирования, удаления и переименование как каталога, так и файла с возможностью сохранения истории;
3. Атомарная фиксация изменений. Ряд изменений имеет возможность попасть в хранилище либо целиком, либо не попасть туда вовсе. Это предотвращает наличие ошибок вида частичного внесения изменений в хранилище;
4. Выбор средства доступа к хранилищу по сети. SVN имеет абстракцию над доступом к хранилищу. Такой подход дает возможность реализации с помощью различных сетевых механизмов доступа. Можно организовать доступ как по протоколу HTTP, так и туннелировать передаваемые данные через SSH;
5. Единый способ работы с данными. Поиск различий между файлами выполняется с помощью специального бинарного алгоритма. Данный алгоритм работает одинаково как с бинарными, так и с текстовыми данными;
6. Эффективная работа с ветками и метками. Создание ветки (копии проекта с дополнительными изменениями) не должно сопровождаться полным копированием проекта и пропорциональному увеличению занимаемого на диске пространства. Subversion создает ветки с помощью механизма, похожего на механизм создания жестких ссылок в файловых системах. Благодаря этому сокращается используемое дисковое пространство и увеличивается скорость создания такой ветки [2].

В основе данной системы контроля версий лежит принцип централизованной модели (рисунок 2). Этот принцип означает наличие единственной «главной копии». Локально хранятся только те файлы, которые были изменены разработчиком. Когда разработчик вносит изменения в репозиторий это приводит к ограничению доступа для других членов команды. Этот контролируемый подход к рабочему процессу разработки снижает риск возникновения конфликтов и привлекает из-за его безопасного и централизованного характера и четкого процесса.



*Рисунок 2 – структура организации Subversion*

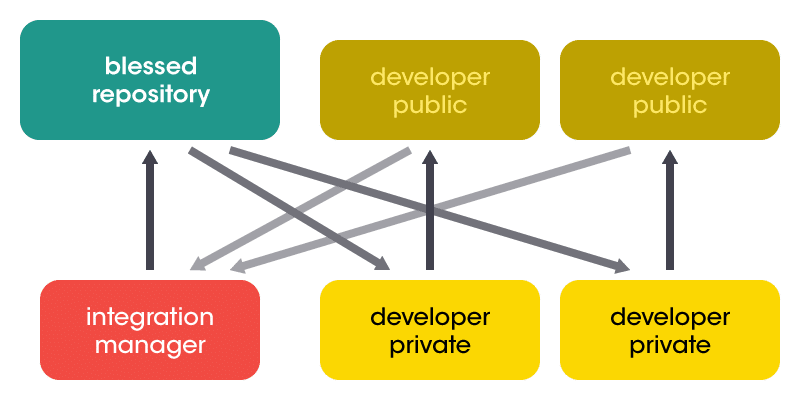
Все вышеперечисленные параметры привели к тому, что все хостовые приложения, разрабатываемые в компании «Компас плюс» используют именно эту систему контроля версий. Более того, технологические платформы, с помощью которых и ведется разработка хостовых систем имеют полную поддержку данной системы контроля версий.

### 1.2.2 Система контроля версий Git

Помимо Subversion и CSV существуют и другие системы контроля версий. Одна из таких систем является Git.

Git – распределенная система контроля версий. В таких системах каждый разработчик проекта имеет возможность создания локального репозитория, который является полной копией центрального хранилища. Благодаря такому подходу каждый разработчик может работать над проектом автономно на своем компьютере. В тот момент, когда изменения будут готовы, разработчик может отправить их в общее хранилище и объединить.

В основе системы Git, как уже было сказано лежит распределенный подход (рисунок 3). Это означает, что все разработчики имеют одинаковый доступ ко всем файлам в хранилище и каждый из них может склонировать хранимый проект с исходным кодом и всеми метаданными. Работа с локальным репозиторием приводит к тому, что, за исключением внесения изменений, все остальные действия могут быть выполнены быстро, поскольку они затрагивают только файлы на локальном диске разработчика, а не на удаленном сервере, как это происходит в SVN. Это позволяет нескольким членам команды получать доступ к одному и тому же файлу и вносить изменения по мере необходимости, но это удобство сопряжено с риском конфликтов слияния Git. Несмотря на это, разработчики по всему миру понимают ценность распределенной модели, поскольку она упрощает совместное использование кода и облегчает удобную совместную работу с большими группами.



*Рисунок 3 – структура работы Git*

К основным отличиям Git от Subversion можно отнести следующее:

1. Автономная работа. SVN требует, чтобы участники были подключены к главному серверу хранилища, что существенно исключает работу в автономном режиме. Git же, наоборот, не требует постоянного подключения к серверу. Единожды склонировав хранилище можно полноценно работать, иногда, синхронизируясь с основным сервером, загружая в него свои изменения и сохраняя к себе изменения других разработчиков.
2. Разрешение конфликтов. Git также превосходит SVN, когда дело доходит до объединения и разрешения конфликтов. Git был разработан для работы с открытым исходным кодом, где многочисленные участники могут работать над одними и теми же частями кодовой базы. Для обеспечения такого типа совместной работы Git создала надежную систему разрешения конфликтов слияния, которая делает процесс более плавным и управляемым
3. Отказоустойчивость. Распределенная модель контроля версий Git помогает снизить вероятность потери основного хранилища. Поскольку участники клонируют основной репозиторий, риск полной потери основного репозитория значительно снижается. С другой стороны, централизованная модель управления версиями SVN создает потенциал для единой точки отказа, если что-то случится с основным репозиторием.

Такая структура работы отлично подходит для Front-end разработки. Поэтому многие приложения, относящиеся к данной категории используют именно Git из-за его гибкости, работы с ветками и возможностями слияния изменений.

### 1.2.3 Система управления проектами Jira

Для эффективной работы в команде требуется придерживаться четких правил и методологий разработки. Это важно, так как подход к разработке программного обеспечения напрямую связан с успешностью разрабатываемого продукта, ведь без правильного подобранной методологии достижение стабильности в работе продукта, безопасности и устойчивости функциональных особенностей практически невозможно.

Методология разработки ПО – это система, определяющая порядок выполнения задач, методы оценки и контроля. Модели разработки ПО выбирают, исходя из направления проекта, его бюджета, сроков реализации конечного продукта. Подходы разработки ПО отличаются друг от друга тем, как этапы жизненного цикла программного обеспечения взаимосвязаны между собой внутри цикла разработки.

Зачастую, разработка программного обеспечения строится из следующих этапов:

1. Определение стратегии. Необходимо выяснить требования заказчика, оценить затраты на реализацию, подсчитать бюджет, определить возможности выполнения работ на взаимных для клиента условиях.
2. Анализ. На данном этапе происходит исследование функций, атрибутов и связей, определенных на стадии определения стратегии.
3. Проектирование. Выполняется построение модели разрабатываемой системы, определяются основные сущности и архитектура проекта.
4. Реализация. Этап непосредственной разработки продукта на основе спроектированной модели.
5. Тестирование. Процесс проверки разработанной системы на наличие ошибок.
6. Внедрение. Процедура передачи готовой системы заказчику. На данном этапе могут устраняться раннее необнаруженные ошибки.
7. Использование и техническая поддержка. После завершения разработки клиент нуждается в технической поддержке. Разработчик, по договоренности с клиентом, предоставляет поддержку на временной или постоянной основе.

Одной из самых распространенных и гибких методологий разработки ПО является Agile.

Agile – гибкая система разработки, в которой большие задачи разбиваются на небольшие этапы. После каждого этапа команда выдает готовую часть продукта на тестирование и оценку.

К основным принципам данной методологии относят:

1. Наивысшим приоритетом является удовлетворение потребностей клиентов за счет своевременной и непрерывной поставки ценного программного обеспечения;
2. Меняющиеся требования не является критичным фактором даже на поздних стадиях разработки. Гибкие процессы используются для конкурентного преимущества;
3. Частая доставка работающего программного обеспечения. От пары недель до пары месяцев, отдавая предпочтение более коротким временным рамкам;
4. Самый эффективный и действенный метод передачи информации команде разработчиков и внутри нее – беседа лицом к лицу;
5. Менеджеры и разработчики должны ежедневно работать вместе на протяжении всего проекта;
6. Работающее программное обеспечение является основным мерилом прогресса [4].

Принимая на вооружение данную методологию необходим инструмент, который будет соответствовать данным принципам и использоваться для организации работ. Одним из таких инструментов является система управления проектами Jira. Эта система является agile-инструментом управления проектами, который позволяет командам планировать, отслеживать, выпускать и поддерживать программное обеспечение. Это единый достоверный источник информации для всего жизненного цикла разработки, благодаря которому автономные команды получают доступ к контексту и могут быстро двигаться вперед, не теряя из виду более масштабную бизнес-цель. ...

## 1.3 Нормативы описания разработок

## 1.3.1 Формат описания разработок в системах контроля версий

Процесс внесения изменений в системы контроля версий сопровождается сообщением, в котором разработчик оставляет небольшой комментарий, описывающий внесенные изменения. По-умолчанию формат таких сообщений не регламентируется системами контроля версий, что позволяет разработчику оставить любой текст в качестве сообщения. Однако во время разработки может возникнуть проблема отслеживания изменений. Взглянув на текст сообщения не всегда возможно понять какие именно изменения были внесены. Разработчик попросту может написать неинформативный текст или вовсе оставить сообщение пустым. Во избежание данной ситуации в компании «Компас плюс» существует регламент описания изменений в системах контроля версий.

Рассматриваемый регламент описывает следующий формат сообщения:

[‘Тип изменения’] (‘Номер задачи’) ‘Название компонентов’. ‘Краткое описание’ <’Номер версии’> (‘Метка подтверждения’)

‘Дополнительное описание’

Приведем пример правильно оформленного текста описания изменения:

[#] (TXPG-2154) Order,Token. Исправлена ошибка оплаты с помощью карты <4.0.23> (QAD)

Была изменена логика работы метода execute класса Order.Payment. Добавлено вспомогательное поле tempSurcharge в классе Token.

Как можно наблюдать, данный формат обладает рядом параметров. Описание каждого из параметров представлено ниже:

1. Тип изменения. Описывает то, какой характер носит изменение. Может содержать одно из определенных в регламенте значений. Описание приведено в таблице 1.
2. Номер задачи. Номер определенной задачи, в рамках которой велась работа по внесению описываемых изменений. Сам номер генерируется автоматически системой управления проектами Jira во время создания задачи;
3. Название компонентов. Имена компонентов, в которых были внесены изменения. Особенно актуально данное поле во время описания изменений в хостовых приложениях, разрабатываемых на платформе RadixWare, так как структура проекта обязательно содержит модули, определяемые во время проектировки приложения;
4. Краткое описание. Небольшой текст, вкратце описывающий внесенное изменение;
5. Номер версии. Необязательный параметр, которое заполняется при внесении изменений более чем в одну версию. Хранит номер самой старой версии, в котору вошло изменение (например, изменение вошло в версии 4.0.23, 4.0.24 и 4.0.25, тогда данный параметр будет иметь значение 4.0.23 т.к. это самая старая версия);
6. Метка подтверждения. Необязательный параметр. Имеет лишь одно значение ‘QAD’. Наличие данной метки означает, что внесение изменений согласовано с отделом тестирования;
7. Дополнительное описание. Необязательный параметр. Произвольная дополнительная информация, предназначенная для других разработчиков, может включать технические детали или историческую справку. Имеет произвольный формат. Может быть многострочным. Помещается через пустую строку после краткого описания [6].

*Таблица 1. Типы изменения в системах контроля версий*

|  |  |
| --- | --- |
| Тип изменения | Описание |
| [++] – Major improvement | Значительная доработка продукта, крупная по объёму либо имеющая коммерческое значение Такие изменения в обязательном порядке проходят тестирование в QAD и описываются в описании релиза для клиентов |
| [+] – Minor improvement | Рядовая доработка, которая может быть интересна клиентам. Такие изменения в обязательном порядке проходят тестирование в QAD и описываются в описании релиза для клиентов |
| [#] – Bugfix | Исправление ошибки, найденной клиентами, либо исправление серьёзной ошибки, обнаруженной внутри Компании, о которой обязательно нужно уведомить клиентов. Такие изменения в обязательном порядке проходят тестирование в QAD и описываются в описании релиза для клиентов |
| [\*] – Change | Изменение существующей функциональности. Например, изменение способа заполнения поля в протоколе, изменение алгоритма взаимодействия с другой системой, и т.д. Такие изменения в обязательном порядке проходят тестирование в QAD и описываются в описании релиза для клиентов |
| [-] – Deletion | Удаление функциональности. Такие изменения должны планироваться  заранее и проходить через стадию deprecation, когда клиенты уведомляются о том, что данная функциональность больше не будет поддерживаться, начиная с определённого релиза. Такие изменения описываются в описании релиза для клиентов |
| [!] – Critical bugfix | Исправление критической ошибки, о которой обязательно нужно уведомить клиентов |
| [.ib] – Internal bugfix | Исправление ошибки, найденной QAD либо другими сотрудниками  Компании. Такие изменения проходят тестирование в QAD, но не попадают в описание релиза для клиентов |
| [.ii] – Internal improvement | Внутренние доработки продукта. Такие изменения проходят  тестирование в QAD, но не попадают в описание релиза для клиентов |
| [.i] – Incomplete | Внутренние изменения кода, не требующие тестирования или описания для клиентов |
| [.r] – Refactoring | Внутренние изменения кода, не требующие тестирования или описания для клиентов |

Как можно наблюдать, описанный регламент дает исчерпывающее представление о вносимом изменении. Данный формат является обязательным, каждое сообщение к изменению в системах контроля версий должно был описано именно в нём.

## 1.3.2 Регламенты описания задач в системах управления проектами

Помимо сообщений к изменениям в системах контроля версий, каждая доработка в продукте сопровождается задачей в системе управления проектами Jira. Процесс ведения задачи также имеет определенный регламент. Рассмотрим основные из них.

Задача в системе управления проектами имеет свой жизненный цикл, состоящий из 3 элементов:

1. Создание задачи;
2. Процесс разработки;
3. Закрытие задачи.

Вовремя создания задачи сотрудник должен заполнить следующие атрибуты задачи:

1. Указать номер версии программного продукта, в котором требуется доработка;
2. Указать компонент, подходящий к теме создаваемой задачи, если это возможно;
3. Необходимо указать клиента, если создаваемая задача связана с запросом от клиента. В противном случае – указать технологическое подразделение в качестве инициатора задачи;
4. Выставить тип задачи. Все типы описаны в таблице 2;
5. Выставить приоритет задачи. Все приоритеты описаны в таблице 3;
6. Дать подробное описание требуемой доработки, либо, в случае ошибки, дать список шагов для воспроизведения найденной ошибки;
7. Приложить к задаче все необходимые логи/трассы/дампы в виде файлов, если это требуется.

В дальнейшем, задача назначается на указанного исполнителя, либо, на лидера компонента, который был назначен создателем задачи. В случае, если исполнитель, как и компонент, не были указаны, задача автоматически назначается на лидера проекта.

*Таблица 2. Типы задачи в системе управления проектами*

|  |  |
| --- | --- |
| Тип задачи | Описание |
| ‘Question’ | Запрос консультации по программному продукту. Создается по вопросам, относящимся к функционированию ПП, ответы на которые не описаны в его документации и спецификациях протоколов или описаны там недостаточно подробно и понятно. К этой категории также относятся вопросы по разработке на  средствах кастомизации ПП |
| ‘Bug’ | Запрос по разрешению подтвержденного дефекта в ПП |
| ‘Task’ | Запрос на аналитику/разработку, связанную с изменениями или доработками в ПП в рамках его существующих функций, а также задач по сопровождению  (тестирование, документирование, инспекция кода) ПП и т.д. |
| ‘New Feature’ | Запрос на аналитику или разработку новой функциональной возможности  в ПП |
| ‘Review’ | Запрос на ревью спецификации и/или кода |
| ‘Epic’ | Организационный Запрос, предназначенный для координации работ по крупным проектам |

*Таблица 3. Приоритеты задачи в системе управления проектами*

|  |  |
| --- | --- |
| Приоритет задачи | Описание |
| ‘Info Request / Minor’ | Минимальный приоритет. Запрос исполнения задачи Разработчиком в периоды времени, свободные от другой работы |
| ‘Ordinary’ | Обычный приоритет. Запрос на исполнение задачи в порядке общей  очереди задач |
| ‘Moderate / Major’ | Повышенный приоритет задачи.  Заказчик запрашивает возможность рассмотрения задачи Разработчиком, как только это будет возможно |
| ‘Blocker’ | Критический приоритет. Выставляется при Отказе ПП |

Процесс работы по задаче сопровождается следующими регламентами:

1. Исполнитель имеет право не начинать работы над задачей, если в нем нет всей необходимой информации. В этом случае он должен указать заказчику комментарий о том, какая дополнительная информация ему нужна;
2. Если задача относится с системной платформе или другому программному продукту, то она перемещается исполнителей в соответствующий проект системы управления проектами;
3. Приступив к работе над задачей, исполнитель обязан изменить её статус на ‘In progress’;
4. В случае приостановки работы над задачей на период более двух рабочих дней её статус должен быть изменен исполнителем на ‘Open’;
5. Если в ходе решения задачи, исполнитель считает нужным создать дополнительные задачи на других разработчиков, то он уведомляет об этом заказчика в своей задаче отдельным комментарием или привязывает создаваемые задачи к основной;
6. Для задач с типами ‘Bug’, ‘New Feature’ и ‘Task’ исполнителем в обязательном порядке заполняется поле ‘Need review’, означающее необходимость создания задачи с типом ‘Review’ для проверки внесенных изменений;
7. Для задач с типами ‘Bug’, ‘New Feature’ и ‘Task’ исполнителем в обязательном порядке заполняется поле ‘Test Report’, информирующее о том, что и как было протестировано в рамках выполнения задачи.

По завершении работы над задачей исполнителю требуется:

1. Закрытие задачи (изменение её статуса на ‘Closed’) осуществляется заказчиком после подтверждения клиентом того, что исходный клиентский запрос может быть закрыт, или после выполнения задачи в случае, если клиент отсутствует;
2. Задача переоткрывается (изменение её статуса на ‘Reopened’) заказчиком только если клиент в течение 14 дней со дня закрытия задачи отзывает своё подтверждение, что исходная задача может быть закрыта;
3. При закрытии задачи исполнитель должен убедиться в отсутствии чувствительных данных в задаче;
4. При закрытии задачи исполнитель должен выставить определенный статус закрытия (табл. 4) [7].

*Таблица 4. Статусы закрытых задач с системе управления проектами*

|  |  |
| --- | --- |
| Статус | Описание |
| ‘Fixed’ | Выполнение всех запланированных работ по задаче,  предоставление всей необходимой по задаче информации |
| ‘Version waiting’ | Опционально. Выполнение всех запланированных работ по Задаче, за исключением сборки релиз-кандидатов версий ПП, содержащих соответствующие изменения. После выхода соответствующих версий ПП необходимо изменение значения на ‘Fixed’ |
| ‘Review’ | Опционально. Передача результата выполненных работ по задаче на процедуру рецензирования. Перед выставлением подобного значения необходимо связать задачу с задачами на рецензирование, а также сопроводить комментарием с указанием перечня  задач на рецензирование. По завершении работ по рецензированию необходимо изменение значения |
| ‘Won’t Fix’ | Отказ от выполнения запрошенных работ по Задаче в том числе при отсутствии в течение более 14 дней ожидаемой активности Заказчика. Выставление  подобного значения необходимо сопровождать  комментарием-обоснованием |
| ‘Duplicate’ | Созданная Задача повторяет другую ранее созданную открытую или успешно завершённую Задачу в том же проекте. Перед выставлением подобного значения необходимо связать эти задачи между собой, а также сопроводить комментарием-указанием какая именно задача была повторена |
| ‘Incomplete’ | В течение более 14 дней отсутствуют необходимые для продолжения работы над задачей данные от заказчика. Выставление подобного значения необходимо сопровождать комментарием-обоснованием |
| ‘Cannot Reproduce’ | Отсутствие возможности повторного воспроизведения ошибки, заявленной заказчиком, приводящее к невозможности локализации и исправления ошибки. Применяется только к задачам типа ‘Bug’ при невозможности установления причин ошибки и подтверждения заказчика об отсутствии  повторяемости ошибки |
| ‘Postponed’ | Задача помещена в перечень потенциальных (перспективных) будущих работ. Применяется только к Задачам типа ‘Task’ или ‘New Feature’,  реализация которых возможна только при соблюдении особых условий, – с указанием таких необходимых условий и возможных сроков реализации при их выполнении |

## 1.4 Контроль качества продукта

Как было указано раннее, процесс разработки программного обеспечения сопровождается такими этапами как тестирование и поддержка. В компании «Компас плюс» для этих процессов существуют специальные бюро продуктового тестирования и технической документации. Работа данных отделов строится, в том числе, на анализе изменений, которые были внесены разработчиком. Сотрудник каждого из отделов имеет доступ к задаче, в рамках которой были внесены изменения. На основе этого и строится его работа.

Получив полную информацию об изменении сотрудник отдела продуктового тестирования может приступить к его проверке. Для него важно то как внесенное изменение влияет на поведение какого-либо функционала системы и какие версии продукта это затрагивает. Из задачи сотрудник отдела продуктового тестирования, может определить, например, правильные параметры запроса к системе, понять как она должна отреагировать на то или иное действие после внесения изменения разработчиком.

В случае несоответствия ожидаемого результата с полученным, тестировщик вправе создать задачу об ошибке, однако не всегда неожиданное поведение является ошибкой. В такой ситуации неожиданное поведение является лишь результатом неполноценного описания изменения разработчиком, поэтому очень важно грамотно описывать изменения как в системе контроля версий, так и в соответствующей задаче.

Работа сотрудника бюро технической документации схожа с работой, описанной выше. Очень важную информацию, в данном случае, несет тип внесенного изменения, так как не все то, что было внесено в разрабатываемый продукт подлежит документации. Это относится к внутренним изменениям в системе, которые не влияют на функциональность, а лишь оптимизируют работу системы. Для тестировщика всегда возникает необходимость проверки функционала на корректность работы, однако для документатора это означает что данное изменение не подлежит описанию.

Также, во время передачи новой версии программного продукта, клиенту передается и список изменений, который строится на основе сообщений в системах контроля версий. В данном случае, как и в случае с документацией, важную роль играет тип внесенного изменения, так как не все типы изменений передаются конечному клиенту (см. таблицу 1).

## 1.4 Выводы к первой главе

Таким образом, в первой главе были рассмотрены основные положения при разработке программного продукта в компании «Компас плюс».

Было выяснено, что многие продукты компании имеют микросервисную архитектуру. При таком подходе вся система разбивается на некоторое количество микросервисов, каждый из которых выполняет определенную задачу, начиная от проксирования запросов, заканчивая выполнением основных функций продукта. Микросервисы создаются с использованием наиболее подходящих языков программирования и технологий, что делает каждый сервис наиболее эффективным и быстродействующим.

В основе каждого проекта находится хостовое приложение. Данная система выполняет основные операции разрабатываемого продукта. Помимо этого, продукт может состоять из ряда дополнительных систем, называемых Front-end приложения. Данные системы предоставляют конечному пользователю способ взаимодействия с продуктом, а также, некоторые из этих систем, используются в качестве вспомогательных или обслуживающих.

При разработке продукта могут быть использованы разные системы контроля версий. К ним относятся Subversion и Git. Subversion используется при разработке хостовых приложей, Git же, наоборот, при разработке Front-end приложений.

Для ведения разработки используется система управления проектами Jira. Данная система разработана с учётом методологии Agile, по которой происходит разработка в компании.

Для контроля качества разрабатываемого продукта в компании существуют регламенты оформления вносимых разработчиком изменений. На основе этих данных свою работу осуществляют бюро технической документации и продуктового тестирования.

# 2 Проектные решения разработки модуля системы csia для мониторинга описания работ по разработке по в системах контроля версий и управления проектами

## 2.1 Общие сведения о системе CSIA

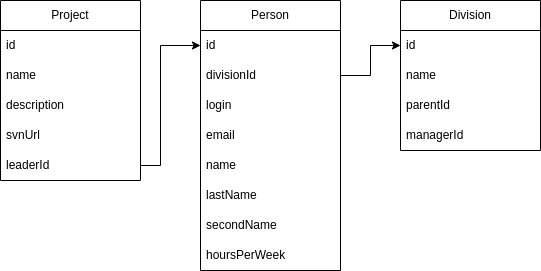
### 2.1.1 Система CSIA

CSIA (Core System for Internal Automation) – продукт на базе платформы RadixWare, предназначенный для автоматизации внутренних бизнес-процессов. В частности, продукт призван решать следующие задачи:

1. Автоматический контроль соблюдения правил, закрепленных внутренними регламентами;
2. Формирование различных видов отчетности о деятельности подразделений для линейных руководителей.

Одновременно с этим CSIA выступает в роли учебной площадки для новых сотрудников, позволяя на примере реальных задач знакомиться с принципами разработки и сопровождения продуктов на базе платформы RadixWare.

Система CSIA обладает моделью, описывающую внутреннюю структуру в компании. Диаграмма модели представлена на рисунке 3.



*Рисунок 3 – Базовая структура сущностей в системе CSIA*

На данном рисунке можно видеть 3 сущности:

1. Project – Представляет из себя проект. Содержит название, описание, путь до репозитория, идентификатор лидера проекта;
2. Person – Сущность, описывающая разработчика проекта. Имеет информацию об отделе, в котором работает разработчик, его имя, почту, доменное имя (уникальное значение, представляющее имя и фамилию сотрудника компании), количество часов, которое работник должен отработать в неделю;
3. Division – Отдел внутри компании. Содержит имя, идентификатор родительского отдела, идентификатор начальника отдела.

На основе данных сущностей работают все модули, разработанные внутри системы.

Как было указано выше, основная задача данной системы – генерация отчетов для внутренних бизнес процессов. На момент начала работы над разрабатываемым модулем для для мониторинга описания работ пUntitled Diagramо разработке ПО в системах контроля версий и управления проектами, CSIA содержала 2 модуля, выполняющих следующие функции:

1. Контроль количества отработанных сотрудником часов;
2. Мониторинг включения задачи в системе управления проектами в релиз программного продукта.

### 2.1.2 Платформа RadixWare

Система CSIA реализована на базе платформы RadixWare, которая, в свою очередь, написана на языке Java. Данная платформа предназначается для разработки приложений, построенных на основе трехзвенной архитектуры, включающей в себя:

1. Сервер системы управления базой данных;
2. Сервер приложения;
3. Клиентское приложение.

В описанной структуре, программный продукт, основанный на данной платформе выступает в роли серверного приложения, что является связующим звеном в данной архитектуре. Поэтому в платформе RadixWare используется технология объектно-реляционного отображения (ORM). Данная технология решает следующие задачи:

1. Синхронизация и контроль согласованности структуры БД (таблиц, связей и т.д.) и кода сервера приложений, использующего эти структуры;
2. Инкапсуляция логики хранения данных: код сервера приложений оперирует не сущностями БД, а объектами предметной области в рамках языка программирования, что уменьшает сложность разработки;
3. Разработка всех трёх звеньев в одной среде;
4. Минимизация трафика между сервером приложений и клиентом.

Платформа основана на декларативном принципе разработки на основе модели (MDA). Разработка сводится к созданию и редактированию модели создаваемого продукта, которая включает в себя:

1. Общие типы данных (типы данных на уровне БД и специфические словарные перечисления);
2. Структуру БД;
3. Структуру и код прикладных классов;
4. Алгоритмы Workflow;
5. Структуры XML-документов и протокольных сообщений;
6. Структуру JSON-схем;
7. Роли пользователей и правила разграничения доступа;

Программный код при разработке используется только в составе модели, для определения специфических алгоритмов поведения прикладной системы, т.е. её бизнес-логики, презентационной логики и логики хранения данных. Среда разработки генерирует Java, SQL и PL/SQL-код из модели продукта.

Помимо средств разработки и поддержки жизненного цикла продуктов, платформа включает повторно используемые компоненты, которые могут использоваться для построения корпоративных систем. RadixWare включает, в частности:

1. Компоненты сервис-ориентированной архитектуры (SOA): системные сервисы,
2. Каталог сервисов, маршрутизация, безопасность и т.д.;
3. Систему разграничения доступа;
4. Средства аудита действий пользователей;
5. Workflow Engine для оркестровки бизнес-процессов, в том числе с человеческим участием;
6. Поддержку различных каналов персональных коммуникаций (Email, SMS и т.д.);
7. Планировщик заданий;
8. Дизайнер и генератор отчетов;
9. Средства описания и поддержки протоколов (как основанных на XML, так и прочих).

Набор готовых компонентов RadixWare достаточен для решения многих задач, возникающих при построении корпоративных систем. При необходимости он может быть расширен [5].

## 2.2 Требования к разрабатываемому модулю

Исходя из процессов и регламентов разработки в компании «Компас плюс», необходимо разработать систему которая бы позволила отслеживать несоответствия регламентам оформления сообщений к изменениям в системах контроля версий и заполнения задач в системе управления проектами.

Ошибки в оформлениях сообщений к изменениям должны отслеживаться как в Subversion репозиториях, для хостовых приложений, так и в Git репозиториях для Front-end приложений. Необходимо учитывать различия в подходах к работе с ветками в данных системах контроля версий. Требуется производить анализ сообщения к изменению для определения корректности заполнения каждого из параметров, описанных в регламенте. Стоит учитывать наличие специальных технических сообщений, характеризующих, например, выпуск определенной версии продукта. Такие сообщение не считаются ошибочными и должны быть проигнорированы.

Проверяемый проект может состоять из нескольких программных продуктов, расположенных в разных репозиториях и разных системах контроля версий.

Каждая задача в системах контроля версий, должна проверяться на соответствие регламенту, описанному выше. Задачи, относящиеся к непосредственной разработке продукта должны иметь хотя бы одно изменение в системах контроля версий.

Результатом работы системы должен быть отчёт, высылаемый как линейному руководителю проекта, так и разработчику, допустившему ошибки при оформлении. Отчёт должен содержать список нарушений, допущенных разработчиком. Нарушения, которые уже были исправлены должны отображаться некоторый период времени для понимания разработчиком, что системой было обнаружено исправление. Также, необходимо привести статистику нарушений разработчика за после несколько дней. Сгенерированный отчёт высылается на рабочую почту сотрудника.

Каждый параметр системы (периодичность запуска проверки, количество дней, которые показываются исправленные нарушения, количество дней для хранения статистики и т.д.) должен быть настраиваемым администратором системы.

## 2.3 Организация взаимодействия с системами контроля версий

### 2.3.1 Общий механизм работы с системами контроля версий

Работа разрабатываемого модуля предполагает тесное взаимодействие с системами контроля версий. Необходимо получать список изменений, а также список веток, каждая из которых представляет определенную версию продукта. Данные процедуры весьма типичны для систем контроля версий, которые используются при разработке в компании. Однако существуют и весомые различия при работе с Subversion и Git. К ним можно отнести:

1. Протоколы и способы взаимодействия;
2. Получение списка веток и изменений;
3. Данные, получаемые от систем контроля версий.

При работе с различными системами контроля версий присутствует необходимость работы с ними, как с некой сущностью, которая имеет определенный общий функционал. Такой подход удобен тем, что не придется делить основную логику работы модуля на две части. А также, снизит трудоёмкость при необходимости поддержки новой системы контроля версий.

Платформа RadixWare не имеет поддержки Git и не обладает возможностями работы с различными системами контроля версий как с одной, обобщенной сущностью. Уже разработанные модули системы CSIA, которые также имеют взаимодействие с системами контроля версий, имеют сугубо внутреннюю реализацию такого взаимодействия, переиспользование которой не представляется возможным.

Основываясь на вышеизложенном, появляется необходимость реализации сущности, которая предоставляет возможность общего подхода при работе с различными системами контроля версий. Для реализации такой сущности отлично подходит парадигма объектно-ориентированного программирования, называемая полиморфизмом.

Полиморфизм – Парадигма ООП, означающая вызов методов в зависимости от типа объекта.

Для реализации такого механизма требуется создать общую, абстрактную сущность, которая имеет в себе необходимые нам методы. А также, требуется создать две сущности, которые имею конкретную реализацию данных методов в зависимости от того, с какой системой контроля версий происходит взаимодействие.

Язык программирования Java предоставляет возможность работы с полиморфизмом, так как является объектно-ориентированным языком. Однако помимо этого, платформа RadixWare позволяет реализовать полиморфизм и при работе с базой данных. Достигается это с помощью механизма Master-detail. Этот особый тип связи между таблицами, позволяющий расширять базовую таблицу дополнительными полями, аналогично тому, как расширяется класс при использовании механизма наследования в языке Java (полиморфизм на уровне персистентной части модели). Radix берет на себя подавляющую часть работы по синхронизации записей из этих двух таблиц, что позволяет при разработке диаграммы классов широко пользоваться механизмом наследования, не прибегая при проектировании диаграммы классов к неудобоваримым конструкциям и программным ухищрениям.

Реализуемые методы должны возвращать определенный тип данных, который содержит в себе всю полученную информацию. Однако при работе с различными системами контроля версий можно заметить, что они хранят различную информацию как о ветках, так и об изменениях. Несмотря на это, можно выделить те данные, которые являются общими для обеих систем, а также, те данные, которые нам необходимы.

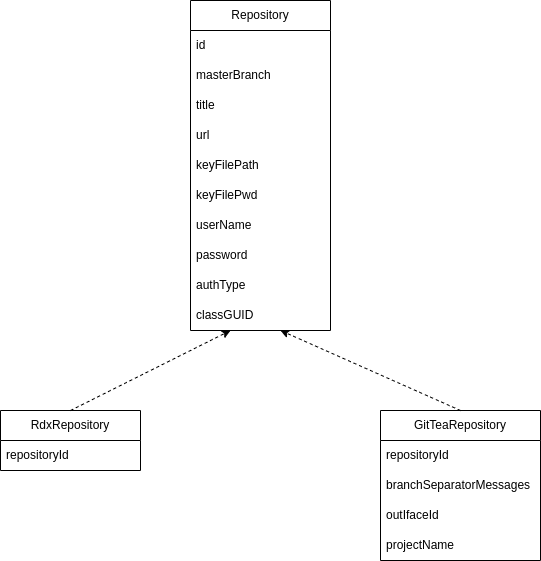
В случае с ветками к таким данным относятся:

1. Название ветки;
2. Дата создания ветки;

Для изменений такими данными будут:

1. Дата внесения изменения;
2. Имя пользователя, внёсшего изменения;
3. Сообщение к изменению;
4. Название ветки, в которую изменение вошло;
5. Номер ревизии изменения.

Итоговую диаграмму сущностей можно увидеть на рисунке .

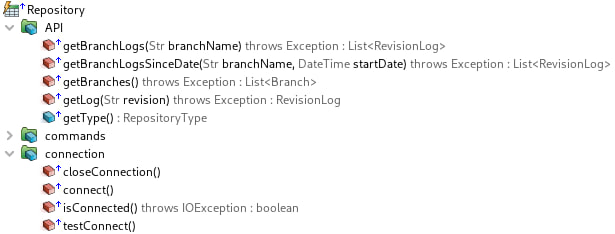


*Рисунок – Диаграмма структуры таблиц для репозиториев*

Таблицы, представленные на диаграмме имеют следующую структуру. Таблица Repository является общей. Она содержит поля, необходимые как для Git, так и для Subversion репозитория. К ним относятся:

1. id – уникальный идентификатор записи в базе данных;
2. masterBranch – название ветки, в которой ведется основная разработка продукта. От данной ветки, впоследствии, будут отрезаны ветки, представляющие определенную версию;
3. title – название репозитория;
4. url – путь до репозитория;
5. authType – тип аутентификации для доступа к репозиторию. Поддерживается аутентификация по логину и по публичному ключу;
6. keyFilePath/keyFilePwd – путь до публичного ключа и пароль от него. Используются при аутентификации по публичному ключу;
7. userName/password – логин и пароль пользователя. Используются при аутентификации по логину;
8. classGUID – служебное поле, используемое платформой RadixWare для реализации механизма Master-detail.

Класс Repository также обладает рядом абстрактных методов, представленных на рисунке



*Рисунок – список методов абстрактной сущности Repository*

Реализуемые методы поделены на две категории. К первой относится непосредственная работа с репозиториям (API). Такие методы возвращают определенные типы данных, хранящие в себе общую информацию об изменениях и ветках для обеих систем контроля версий. В эту категорию входят следующие методы:

1. getBranchLogs – получение списка всех изменений в ветке;
2. getBranchLogsSinceDate – получение списка изменений в ветке, начиная с определенной даты;
3. getBranches – получение списка веток;
4. getLog – получение изменения;
5. getType – получение типа репозитория.

Вторая категория (connection), описывает набор методов для работы с подключением к репозиторию:

1. closeConnection – закрыть соединение с репозиторием;
2. сonnect – подключиться к репозиторию;
3. isConnected – проверка работоспособности текущего подключения к репозиторию;
4. testConnect – проверка возможности подключения к репозиторию.

Таблица RdxRepository представляет из себя сущность, хранящую в себе данные для работы с репозиториями продуктов, основанных на RadixWare. Можно заметить, что данная таблица хранит в себе всего одно поле – repositoryId. Данное поле используется для связи с таблицей Repository, в которой хранятся все остальные данные, необходимые для работы с репозиторием. При использовании сущности, связанной с данной таблицей в программном коде, она будет иметь все поля, описанные в этих двух таблицах.

Таблица GitTeaRepository хранит информацию, необходимую для работы с Git репозиторием. К полям, хранимым в таблице относятся:

1. repositoryId – уникальный идентификатор записи в таблице Repository, необходимый для связи этих таблиц;
2. branchSeparatorMessages – поле, хранящее массив строк, которые используются при разделении веток в Git репозиториях;
3. outIfaceId – уникальный идентификатор исходящего HTTP интерфейса, необходимого для подключения к репозиторию;
4. projectName – имя проекта, с которым будет вестись работа.

Таким образом, создание необходимых таблиц в базе данных с использованием механизма Master-detail и описание классов, основанных на данной модели, позволяет нам работать с репозиторием, как с абстрактной сущностью, что в свою очередь, позволяет не делить логику модуля на две части, для работы с Subversion и Git репозиториями.

### 2.3.2 Взаимодействие с Subversion

Для работы classGuidс Subversion репозиторием был создан класс Repository.RdxSvn, который основывается на таблице RdxRepository и является наследником абстрактного класса Repository.

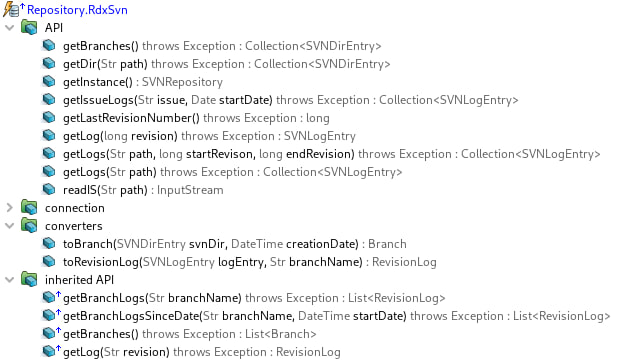
Для обращений к Subversion серверу используется библиотека SvnKit. Данная библиотека предоставляет широкий спектр возможностей для взаимодействия с используемой системой контроля версий на языке Java.

При вызове метода для подключения к репозиторию, в динамическое свойство instance записывается инстанция класса SVNRepository, который является частью библиоткеки SvnKit и предоставляет набор методов, для взаимодействия с репозиторием.

Любой репозиторий продукта на базе платформы RadixWare имеет определенную структуру директорий:

1. Archive – директория для хранения устаревших версий продукта;
2. Config – директория для хранения конфигурации продукта;
3. Dev – директория для хранения веток продукта;
4. Doc – директория для хранения документации по продукту;
5. Etc – директория для хранения дополнительных файлов, необходымых при разработке продукта;
6. releases – директория для хранения выпущенных версий продукта.

Таким образом, несмотря на то, что система контроля версий Subversion хоть и не описывает четкую структуру хранения, мы всё таки можем реализовать некоторые методы для работы с данным типом репозиториев благодаря чёткой структуре репозиториев проектов, использующих RadixWare. Полный список методов представлен на рисунке .



*Рисунок – список методов класса Reposiotry.RdxSvn*

Категория API описывает методы, которые необходимы для непосредственного взаимодействия с сервером Subversion. Возвращаемым типом данных методов являются объекты библиотеки SvnKit, содержащие информацию об изменениях и ветках.

В категории inherited API находятся методы, унаследованные от класса Repository. Их реализация строится на вызове методов из категории API с последующей конвертацией в нужный тип данных с помощью методов из категории converters.Категория API описывает методы, которые необходимы для непосредственного взаимодействия с REST API сервиса GitTea. Возвращаемым типом данных методов являются объекты, классы которых были сгенерированы на основе JSON схем.

### 2.3.3 Взаимодействие с Git

Реализация взаимодействия с Git репозиториями имеет ряд особенностей, так как в компании «Компас плюс» используется нестандартный Git сервер, создаваемый с помощью команды «git init». Вместо этого, в качестве основы выступает GitTea.

GitTea – это автономный сервис разработки программного обеспечения «все в одном», он включает в себя Git-хостинг, проверку кода, совместную работу в команде, реестр пакетов и CI/CD (continuous integration, continuous delivery). Данный сервис аналогичен GitHub, Bitbucket и GitLab, однако отличается тем, что предоставляет возможность создания собственного сервера для хранения репозиториев.

GitTea имеет следующие преимущества над классическим Git сервером:

1. Пользовательский интерфейс;
2. Панель мониторинга проблем (аналог Jira);
3. Поддерживается реестр пакетов;
4. Поддерживается Markdown разметка;
5. Веб-редактор кода;
6. Поддерживается управление системой посредствам REST API.

В качестве сущности для взаимодействия с репозиториями выступает класс Repository.GitTea, который базируется на таблице GitTeaRepository и наследуется от абстрактного класса Repository.

Особенностью обращений к Git репозиториям является то, что для этого не используется библиотека, реализующая стандартные способы взаимодействия с репозиториями данного типа. Вместо этого, используется REST API сервиса GitTea, который также позволяет получить всю необходимую для нас информацию о репозитории.

REST – стиль построения архитектуры распределённого клиент-серверного приложения.

Интерфейс взаимодействия с данным сервисом позволяет получать следующие данные:

1. Список репозиториев, хранящихся на данном сервисе;
2. Список веток репозитория;
3. Список изменений в репозитории и ветке.

REST API GitTea работает по протоколу HTTP. Для работы с сервисом по данному протоколу, в платформе RadixWare существует специальный класс, называемый исходящим HTTP интерфейсом. Данный класс позволяет генерировать и отправлять любой запрос по протоколу HTTP. Он поддерживает все HTTP методы (например, GET, POST, PUT), позволяет устанавливать любые заголовки запроса, предоставляет возможность заполнения тела запроса. Для работы с исходящим интерфейсом, его необходимо настроить, указав адрес сервера, к которому будет производиться обращение. Также присутствует и ряд других настроек, таких как время ожидания ответа от сервера, стандартный HTTP метод, клиентский сертификат для аутентификации и т.д. Для связи с конкретным исходящим интерфейсом в таблице GitTeaRepository присутствует поле outIfaceId.

В качестве формата обмена данными, который используется в REST API, выступает JSON (JavaScript Notation Object). Для описания данных в этом формате используется JSON Schema – специальная структура, позволяющая описывать любой JSON документ. Radix поддерживает работу с JSON схемами. Выглядит это следующим образом: разработчик описывает JSON схему, после чего, на основе данной схемы генерируется Java класс со всеми описанными полями, другими словами, происходит кодогенерация. Сгенерированный класс обладает специальными методами для сереализации и десериализации JSON документа, то есть преобразования текста в формате JSON в Java объект и наоборот.

Пример JSON схемы можно увидеть на рисунке .

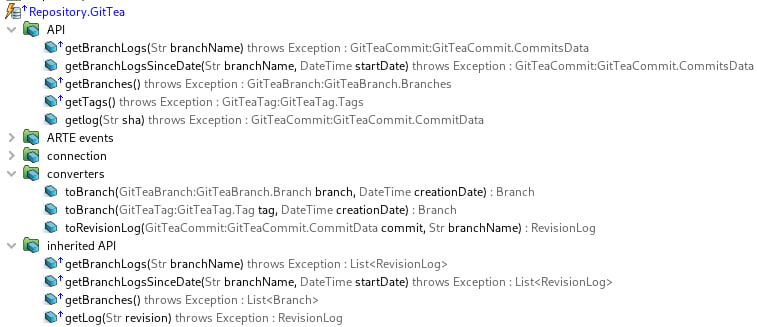


*Рисунок – пример JSON схемы описывающей ответ GitTea со списком изменений*

В совокупности, исходящий HTTP интерфейс и поддержка JSON схем позволяет легко создать собственный клиент для обращения к сервису GitTea. Для этого необходимо реализовать методы, которые будут возвращать требуемые для работы модуля данные и описать схемы запросов и ответов от GitTea.

Список реализуемых методов представлен на рисунке .

*Рисунок – Список методов класса Repository.GitTea*



Данная структура полностью аналогична структуре методов в классе Reposioty.RdxSvn, с тем отличием, что методы категории API возвращают объекты, классы которых были сгенерированы на основе JSON схем.

## 2.4 Взаимодействие с Jira

Классом, реализующим взаимодействие с системой управления проектами Jira является OutInterface.Jira. Данный класс расширяет стандартный Radix класс OutInterface.Http с помощью механизма Master detail.

Организация взаимодействия с Jira аналогична организации взаимодействия с GitTea. Система управления проектами Jira также предоставляет REST API, основанное на протоколе HTTP c использованием формата описания данных JSON.

Класс описывает методы, представленные на рисунке .

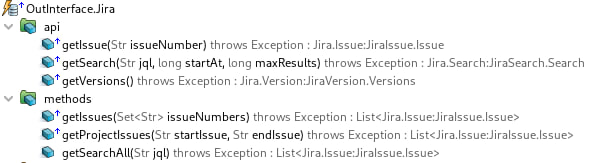


Рисунок – Методы класса OutInterface.Jira

1. getIssue – получение информации о задаче;
2. getSearch – получение списка задач, отфильтрованных с помощью JQL запроса;
3. getVersion – получение списка версий проекта;
4. getIssues – получение списка задач;
5. getProjectIssues – получение списка задач, в диапазоне;
6. getSearchAll – метод выполнения запроса поиска с поддержкой пагинации.

## 2.5 Мониторинг нарушений в проекте

### 2.5.1 Структурная модель

Платформа RadixWare имеет специальный механизм, который позволяет производить периодичный запуск задач. Для этого существует специальный класс Task, представляющий задачу для выполнения, а также класс Scheduler, который позволяет настроить периодичность запуска задачи. Для реализации сущности, которая будет производить требуемые проверки это отлично подходящий инструментарий.

На основе класса Task, расширив его с помощью механизма Master detail, создана сущность Task.checkProjectViolations. Данная сущность имеет следующие поля:

1. taskId – уникальный номер задачи;
2. notificationSenderId – уникальный номер сущности для отправки сообщений, содержащих отчёт;
3. projectId – уникальный номер проверяемого проекта;
4. statRecordCount – максимальной количество записей в диаграмме, отображающей нарушения;
5. sendRepost – флаг, отвечающий за отправку сообщений;
6. resolvedViolationShowDays – количество дней, которые будут показывать исправленные нарушения в отчёте;
7. masterVersion – номер следующей релизной версии продукта;
8. lastCheckedIssue – номер последней проверенной задачи в системе управления проектами.

Для поддержи возможности проверки нескольких репозиториев создана специальная сущность ProjectRepositoryItem. Она необходима лишь для хранения следующих параметров:

1. Id – уникальный идентификатор;
2. projectRuleViolationTaskId – уникальный идентификатор задачи к которой относится данная настройка репозитория;
3. repository – уникальный идентификатор репозитория;
4. lastCheckedRevision – номер последнего проверенного изменения;
5. ignoreCommitRegexes – список регулярных выражений, необходимых для фильтрации сообщений;
6. checkBranchesRegexes – список регулярный выражений для названий проверяемых веток.

Для работы с нарушениями используется подход, похожий на тот, что использовался при работе с репозиториями. То есть, есть общий класс RuleViolation, представляющий информацию о нарушении. Он расширяется наследниками CommitRuleViolation, хранящий в себе информацию по нарушениям в изменении в системе контроля версий, и IssueRuleViolation, в котором хранится информация о нарушении в системе управления проектами.

Класс RuleViolation содержит следующие поля:

1. Id – уникальный идентификатор нарушения;
2. classGuid – служебное поле, используемое платформой RadixWare для реализации механизма Master-detail.
3. projectId – уникальный идентификатор проекта, в котором было допущено нарушение;
4. personId – уникальный идентификатор человека, допустившего нарушение;
5. message – сообщение, описывающее нарушение;
6. type - тип нарушения;
7. resolutionType – тип исправления нарушения;
8. createdAt – дата обнаружения нарушения;
9. resolvedAt – дата исправления нарушения.

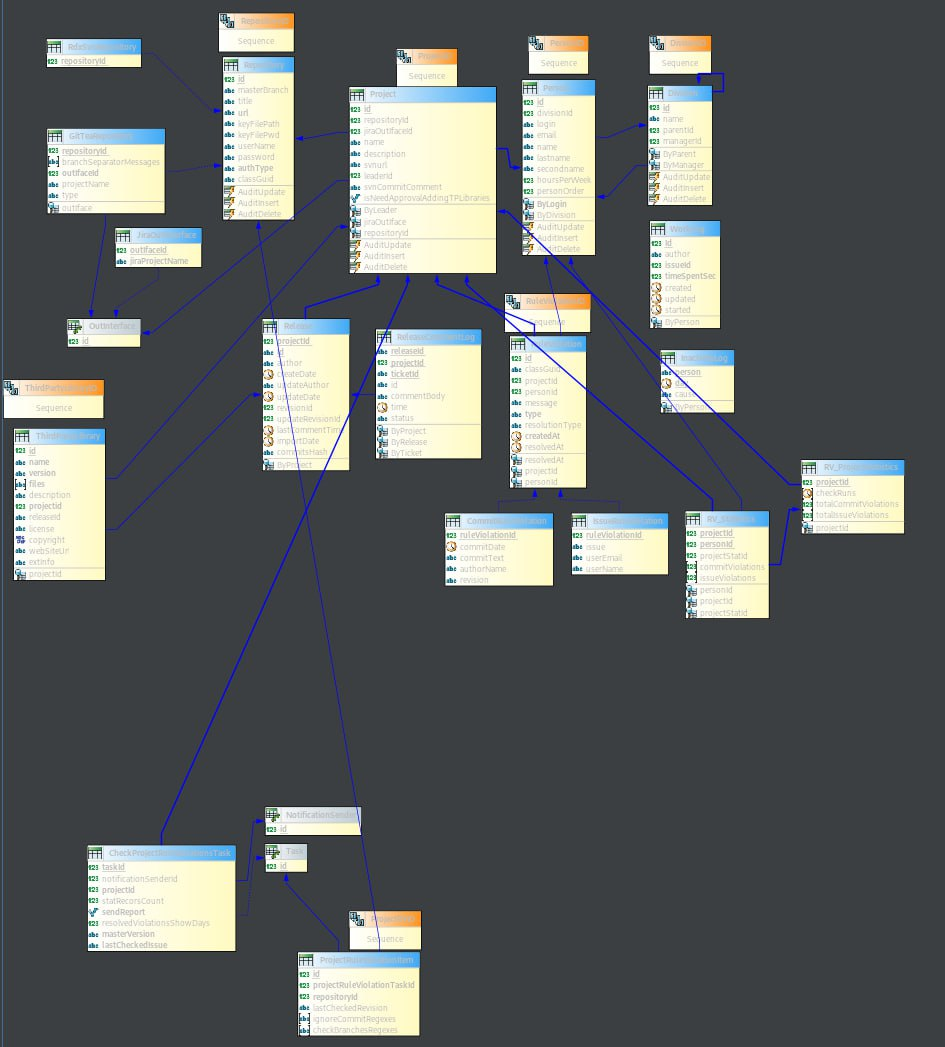
Класс CommitRuleViolation расширяет сущность RuleViolation следующими полями, необходимыми для описания нарушения в сообщении к изменению в системе контроля версий:

1. ruleViolationId – уникальный идентификатор нарушения;
2. commitDate – время внесения изменения в систему контроля версий;
3. commitText – текст сообщения к изменению;
4. authorName – имя автора изменения (используется при отсутствии пользователя в системе, т.е. поле personId пустое);
5. revision – уникальный идентификатор изменения в системе контроля версий;

Для описания нарушения в системе управления проектами используется сущность IssueRuleViolation, расширяющая класс RuleViolation:

1. ruleViolationId – уникальный идентификатор нарушения;
2. issue – номер задачи, в которой обнаружено нарушение;
3. userEmail – электронная почта пользователя, допустившего нарушение;
4. userName – имя пользователя, допустившего нарушение (используется по аналогии с authorName).

Полная модель сущностей, необходимых используемых для работы модуля представлена на рисунке .



*Рисунок – Модель сущностей используемых в работе модуля.*

### 2.5.2 Алгоритм поиска нарушений

Процесс поиска нарушений в проекте описан алгоритмом на рисунке .

*Рисунок – Алгоритм поиска нарушений в проекте.*

## 2.6 Генерация отчета

### 2.6.1 Работа со статистикой нарушений

Текст

### 2.6.2 Алгоритм генерации отчета

Текст

## 2.7 Выводы ко второй главе

Текст

# 3 Результаты опытной эксплуатации модуля системы csia для мониторинга описания работ по разработке по в системах контроля версий и управления проектами

## 3.1 Работа с разработанными сущностями

### 3.1.1 Сущность проекта

Текст

### 3.1.2 Сущность репозитория

Текст

### 3.1.3 Сущности системы управления проектами

Текст

## 3.2 Создание и настройка задачи для мониторинга

### 3.2.1 Задача Task.CheckProjectViolations

Текст

### 3.2.2 Настройка расписания

## 3.3 Пример сгенерированного отчёта

Текст

## 3.4 Выводы к третьей главе

Текст

# Заключение

Текст

# Список используемых источников

1 - <https://vk.com/doc7608079_449808456?hash=TrY4ho4utnq5LpCaxFpuamI4W8paaZPTR1sjBm5taBz&dl=49oI2tsG9r5KEMwqABnsociXSnqZQzVK3UstJ5VR2U4>

# Приложение А