Содержание

1 Введение.

Данная дипломная работа написана в рамках промышленного проекта по разработке информационной системы SRP.

В связи с чрезвычайной сложностью бизнес-логики в проекте SRP повышенное внимание уделяется вопросам обеспечения и контроля качества. Для достижения этой цели применяется многоступенчатая система тестирования разрабатываемой функциональности.

Одной из важнейших ступеней этого процесса является модульное тестирование - покрытие функциональности тестами, проверяющими работоспособность отдельных участков (модулей) кода. Разработкой этих тестов занимаются сами программисты, создававшие тестируемую функциональность.

В связи с тем, что во многих ситуациях разработчик должен дожидаться (удачного) выполнения набора модульных тестов, важной является задача минимизации времени их выполнения.

Большая часть времени при работе модульных тестов уходит на подготовку необходимых тестовых данных. В связи с этим, была поставлена задача разделить создание этих данных и собственно тестирование.

В главе "Описание проекта SRP" представлен краткий обзор структуры архитектуры и бизнес-логики проекта. Эта информация позволяет оценить важность и сложность процесса контроля качества в SRP. В завершение главы приводится описание методов и средств, используемых для достижения этой цели.

В главе "Кэширование" данных при тестировании" более подробно обозначается поставленная задача и предлагается метод ее решения.

Описывается логика, дизайн и сигнатуры классов предложенного решения.

В "Заключении" представлены полученные результаты.

2 Описание проекта SRP.

Проект Shared Royalty Platform (SRP) посвящен разработке программного комплекса, предназначенного для расчета авторских и исполнительских отчислений с продаж музыкальной продукции. Размер отчислений регулируется контрактами, каждый из которых может содержать массу условий, индивидуальных для каждого исполнителя; процент отчислений для каждой отдельной продажи также различен.

Система представляет пользователю возможность редактировать все бизнес-объекты (контракты, состав музыкальных продуктов и т.п.), а также взаимодействует со многими внешними системами (системы учета продаж, бухгалтерские системы и т.д.).

Заказчиком продукта является компания Royalty Services - совместное предприятие Exigen Services и двух крупнейших музыкальных компаний - UMG (Universal Music Group) и WMG (Warner Music Group).

2.1 Архитектура SRP.

2.1.1 Требования, предъявляемые к архитектуре.

Стандартные требования, предъявляемые к промышленным системам:

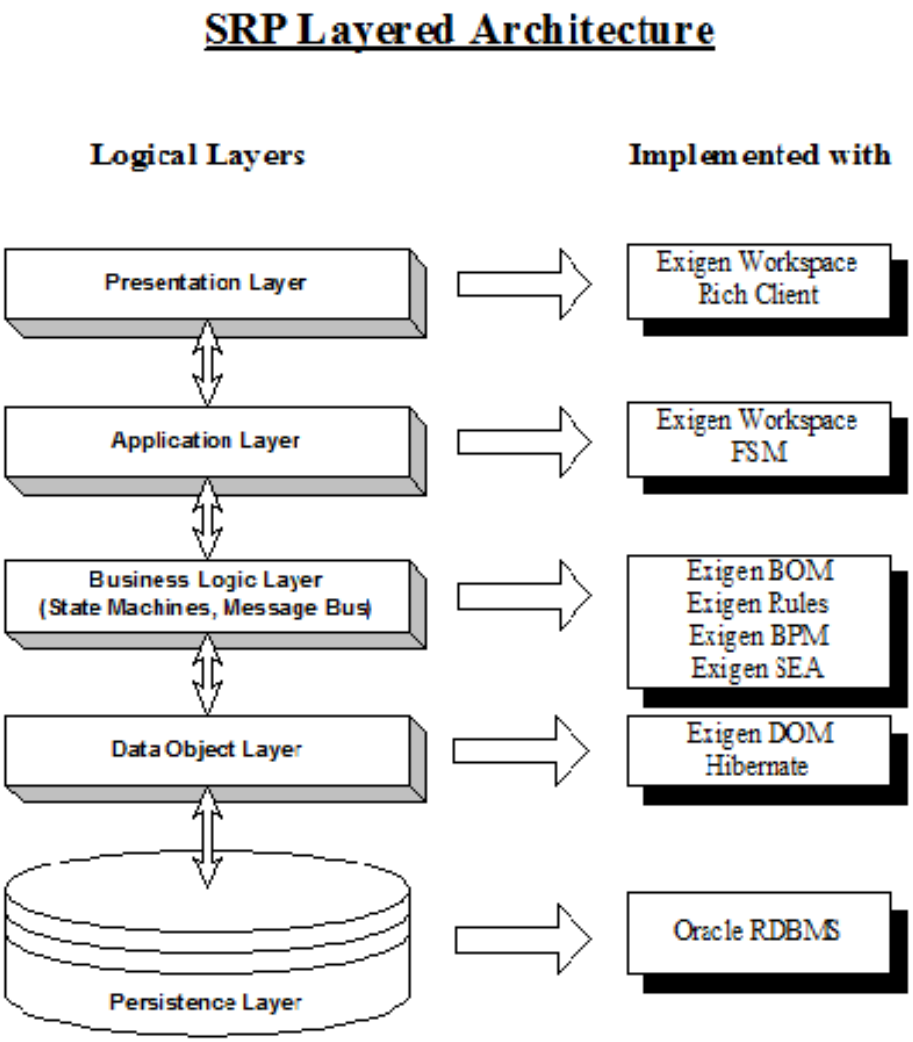
- Безопасность / контроль доступа.

- Надежность / отказоустойчивость.
- Масштабируемость.
- Производительность.

Особенности SRP, накладывающие дополнительные требования:

- Чрезвычайно сложная бизнес-логика.
- Сложные и многочисленные взаимодействия между сущностями.
- Необходимость сохранения истории изменений у некоторых сущностей.

2.1.2 Логические уровни и их реализация



Как видно из схемы SRP является многоуровневым приложением.

На нижнем уровне сохранение данных (persistence) обеспечивается с помощью базы данных Oracle.

Следующий уровень - это уровень объектных данных (Data Objects),

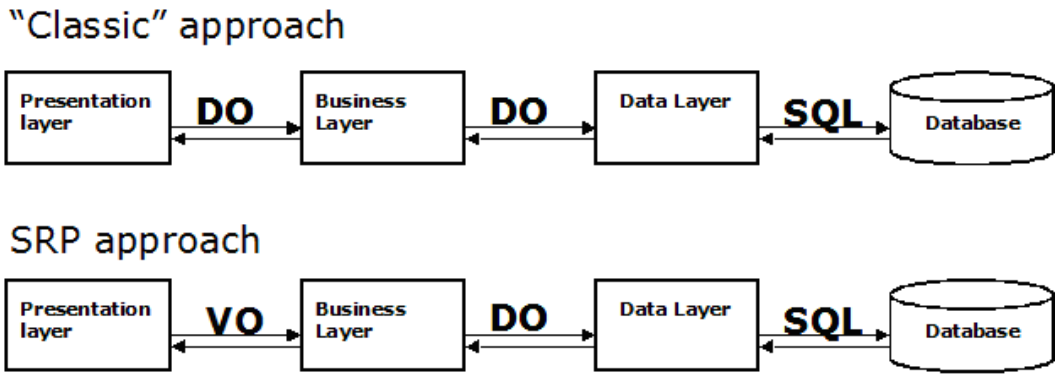
который реализован через систему объектно-реляционной связи.

Уровень бизнес-логики (Business Logic Layer) содержит в себе основной код приложения и реализуется с помощью различных библиотек и технологий (BOM, BPM, SEA).

Уровень приложения - это дополнительный уровень в SRP, который обычно в других приложениях объединен с презентационным. Он является связующим между уровнями бизнес-логики и пользовательского интерфейса. Получая запросы с презентационного уровня, он вызывает нужную функциональность уровня бизнес-логики, обрабатывает результат и передает его на пользовательский интерфейс.

Таким образом презентационный уровень полностью освобожден от какой-либо логики. Единственной его задачей является посылка запросов на уровень приложения и отображение полученных результатов.

2.1.3 Хранение и передача данных.



Классический подход: одни и те же объекты (Объекты Данных - Data Objects) используются на всех уровнях, включая презентационный.

Подход SRP: на уровне бизнес логики и пользовательского интерфейса используют специальные, созданные вручную объекты (Объекты Значений - Value Objects), моделирующие только ту ограниченную область данных, которая необходима для выполнения конкретной задачи. Например, для Объекта Данных Publisher:

```
public interface Publisher {
   public void setPublisherId(Long publisherId);
   public Long getPublisherId();
   public void setRefPublisher(String refPublisher);
   public String getRefPublisher();
   public void setName(String name);
   public String getName();
   public void setActiveBl(Boolean activeBl);
   public Boolean getActiveBl();
   public void setAffiliation(String affiliation);
   public String getAffiliation();
   public void setComments(String comments);
   public String getComments();
   public Set getPmuNewPublishers();
   public Set getPublisherMassWorklists();
   public Set getSaSongOwners();
}
```

можно создать Объект Значений, содержащий только нужные в данном случае поля:

```
public class PublisherEditVo extends PublisherReferenceVo {
   public String refPublisher;
   public String name;

   public String getRefPublisher() {
      return refPublisher;
   }

   public void setRefPublisher(String refPublisher) {
      this.refPublisher = refPublisher;
   }

   public String getName() {
      return name;
   }

   public void setName(String name) {
      this.name = name;
   }
}
```

Преимущества использования Объектов Значений:

- Реализуется парадигма программирования, позволяющая явно и детально описать используемые данные и работу с ними
- Строгое описание интерфейсов бизнес методов
- Объекты значений сериализуемы (удобно для презентационного уровня и использования во внешних интерфейсах)

Недостатки использования Объектов Значений:

- Требуется написание кода большого количества объектов для каждой сущности. Однако, данный недостаток в значительной степени нивелируется тем, что большая часть этого процесса автоматизирована.

2.1.4 Версионирование.

Требования:

- Все, в том числе и проведенные в достаточно отдаленном прошлом, рассчеты должны иметь в системе подтвержденное обоснование.
- Все «функционально значимые» изменения требуют утверждения ответственным пользователем.
- История утвержденных изменений должна сохраняться для возможности внутреннего аудита.

- При проведениях окончательных рассчетов система должна поддерживать фиксированное состояние.

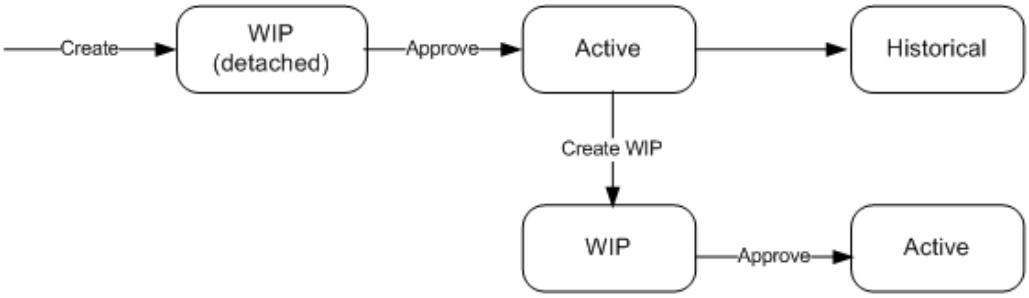
Реализация:

- Каждый версионируемый объект имеет несколько версий, как-то: активная (Active), черновая (WIP – Work In Progress), исторические.
- Все версии одного объекта объединены в версионное дерево специальным объектом-маркером - головной версионной меткой (Versioned Header).
- Связи между объектами могут быть двух типов: ссылка на конкретную версию либо ссылка на головную версионную метку.

Жизненный цикл версионируемого объекта выглядит следующим образом:

- Создается первоначальная черновая версия (detached WIP).
- Происходит утверждение (Approve) первоначальной черновой версии и она становится активной (Active). На данный момент черновой версии данного объекта в системе нет.
- Происходит создание новой черновой версии (WIP). В системе одновременно сосуществуют активная и черновая версии объекта. При этом в функциональном коде для рассчетов используется исключительно активная.

- Происходит утверждение (Approve) черновой версии. Активная версия становится исторической. Черновая версия становится активной.



2.1.5 Внешний интерфейс.

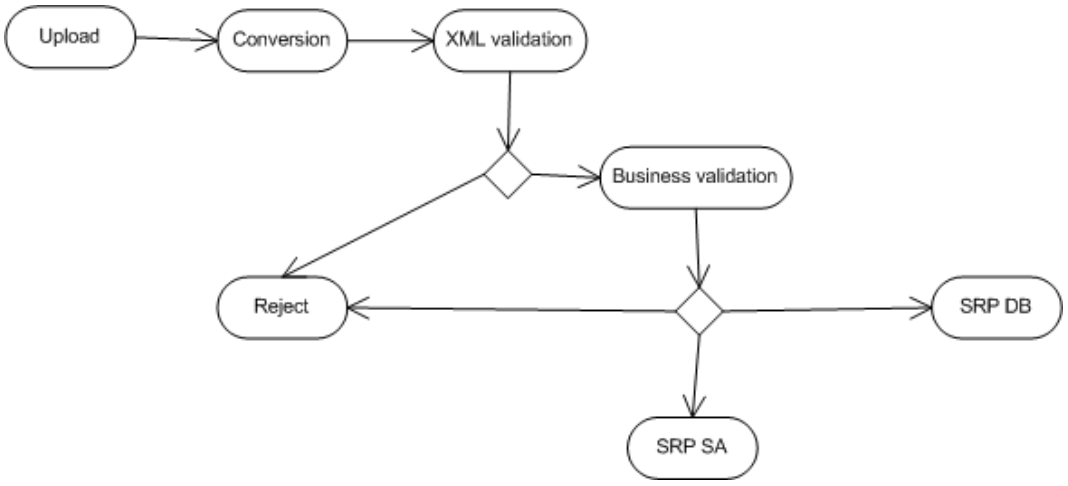
SRP не является независимой отдельностоящей системой. В процессе работы ей приходится обмениваться данными с различными внешними системами заказчиков. На самом деле, в общем смысле жизненный цикл SRP представляет собой следующую последовательность: получение исходных данных из внешних систем; обработка этих данных; отправка результатов в другие внешние системы.

Задача передачи данных реализуется с помощью механизма Поставки пакетов (Feeds), который разделяется на Входящие поставки пакетов (Inbound Feeds) и Исходящие поставки пакетов (Outbound feeds) - соответственно, для информация поступающей в SRP и исходящей из SRP.

Входящие пакеты:

- Заказчики загружают файл с необходимой информацией в заранее оговоренную директорию на ftp-сервере.

- SRP по имени файла определяет тип входящего пакета и вызывает нужный конвертор для преобразования пришедшей информации из формата заказчиков в специальный SRP XML-формат.
- Полученный в результате конвертации XML файл валидируется по схеме. В случае неправильной структуры файл отвергается.
- Прошедший валидацию по схеме файл передается на бизнес-валидацию, специфическую для каждого типа входящих пакетов. В зависимости от результатов, пакет либо отвергается (в случае неустранимых ошибок или несоответствий), либо помещается в специальное место – Подготовительную зону (Staging Area) (в случае незначительных ошибок), либо обрабатывается и информация сохранятся в базу данных SRP.
- При любом исходе обработки входящего пакета в систему заказчика отправляется уведомление, содержащее информацию о наличии и типе ошибок, либо о успешной загрузке.



Исходящие пакеты:

- В рамках регулярного процесса либо по явной команде пользователя системы (в зависимости от типа исходящих пакетов) SRP собирает и систематизирует информацию, которая будет отправлена в данном конкретном пакете.
- Полученная информация проходит через конвертор, преобразующий ее в формат, распознаваемый внешней системой заказчика.
- Полученный файл в формате заказчика загружается на ftp-сервер заказчика в заранее оговоренную директорию.
- Уведомления, приходящие от заказчика, обрабатываются как специальный тип входящих пакетов.

2.1.6 Обеспечение безопасности.

Цели обеспечения безопасности и контроля доступа к информации в SRP осуществляются посредством следующих парадигм:

Привилегии:

- Представляют собой разрешения на выполнение определенной операции.

- Набор привилегий задается при создании системы, в дальнейшем не редактируется.
- Пример: ”просмотр личных адресных данных музыкального исполнителя”.

Роли:

- Роль представляет собой набор привилегий, необходимых для выполнения задач, поставленных перед определенным пользователем системы.
- Набор ролей задается в XML-файле и может быть отредактирован.
- Пример: ”Аналитик, управляющий контрактами на сумму менее $ 100’000”.

Группы:

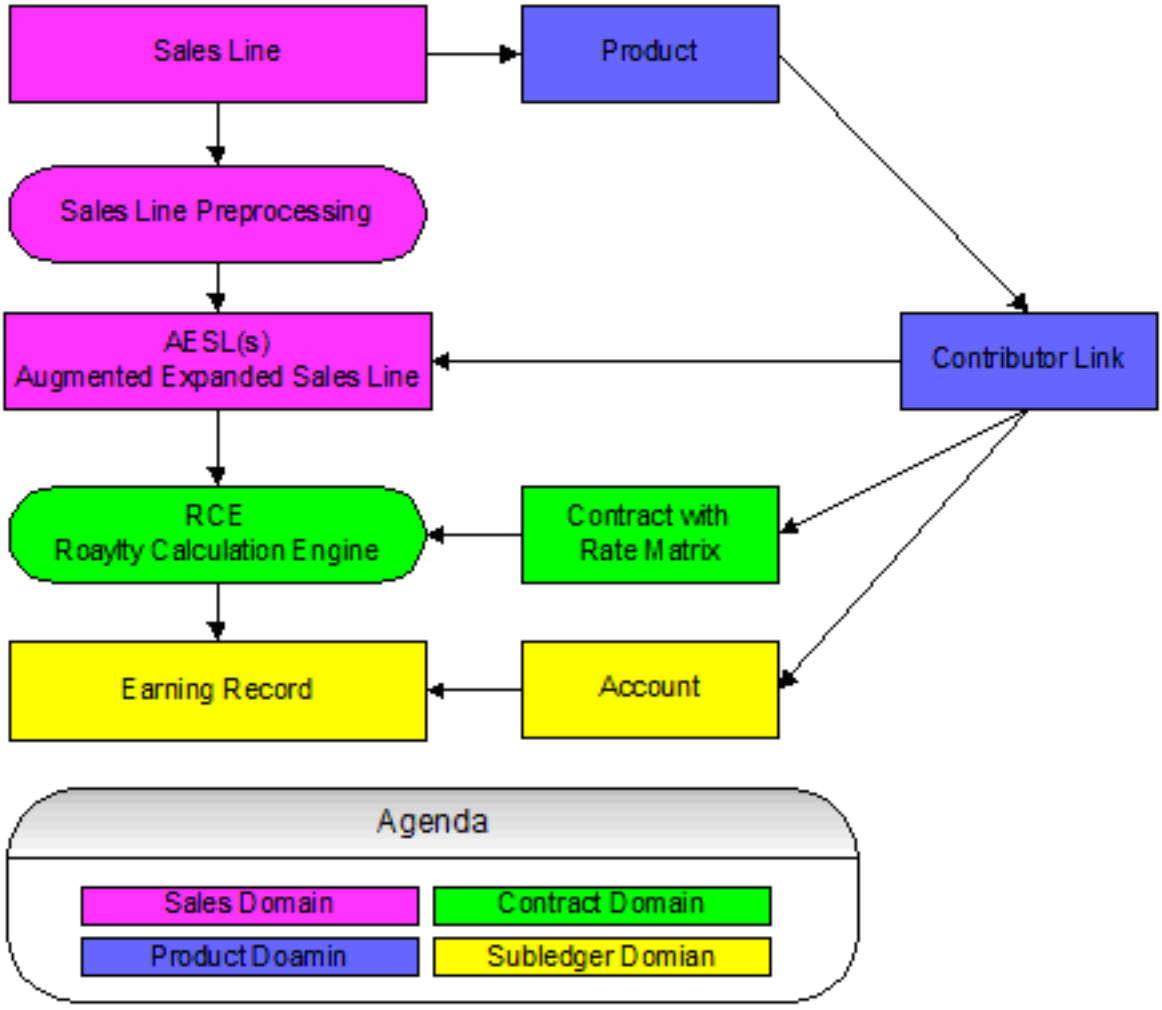
- Группа представляет собой набор ролей.
- Редактируется администратором через пользовательский интерфейс.
- Пример: ”Группа ”Аналитики” включает в себя роли ”Аналитик контрактов” и ”Аналитик продуктов”.

2.2 Бизнес - логика SRP.

2.2.1 Общая схема логики системы.

SRP представляет из себя сложную распределённую систему с большим количеством транзакций.

Схему основных взаимосвязей и процессов взаимодействия различных модулей данной системы схематично можно представить следующим образом:



Каждый модуль SRP, называемый **доменом** (domain) представляет собой набор модулей, реализующих функциональность, связанную с определенным участком бизнес-логики системы.

2.2.2 Краткое описание логики доменов.

Sales Line

- Sales Line содержит данные, источниками которых являются сообщения из многих коммерческих источников из разных стран.
- Каждая Sales Line Entry может содержать коммерческие данные только из единственного коммерческого источника из одной страны.
- SRP Sales Line складывается из данных, полученных во время обработки информации о продаже продукта.
- SRP Sales Line содержит данные для каждого отдельного продукта, который используется для подсчета отчислений по авторским правам.

Product

- Product является элементом, который предлагается для продажи звукозаписывающей компанией.
- Таким продуктом может быть компакт-диск, кассета, отдельная песня (сингл) и т.д.

Augmented Explored Sales Line

- Augmented Exploded Sales Line – данные, созданные на основе информации Sales Line и информации, предоставляемой продуктом.

- Содержит данные из Контракта, Аккаунта, Продукта, связанные с данной продажей.
- Содержит информацию для дальнейшего подсчета размера отчислений правообладателю.

Contributor Link

- Связь между Product и объединением Contract + Subcontract + Account.
- Определяет алгоритм, по которому должен быть вычислен лицензионный платеж.

Contract

- Соглашение между двумя или более объектами, которое создает обязательство по предоставлению определенных услуг

Contract with Rate Matrix

- Rate Matrix - содержит одну или более Rate Table, которые вместе определяют правила для того, чтобы вычислить лицензионный платеж.
- Subcontract - часть контракта, описывающая его более подробно – раскрывающая дополнительные детали соглашения, определяющая дополнительные условия и т.п.

- Rate Period - более детальное рассмотрение Subcontract за определённый период времени.
- Каждому Rate Period соответствует одна Rate Matrix

Earning

- По результатам пришедших SL для Аккаунта создабтся Earnings, содержащие информацию о заработках артиста с каждой продажи его продуктов.

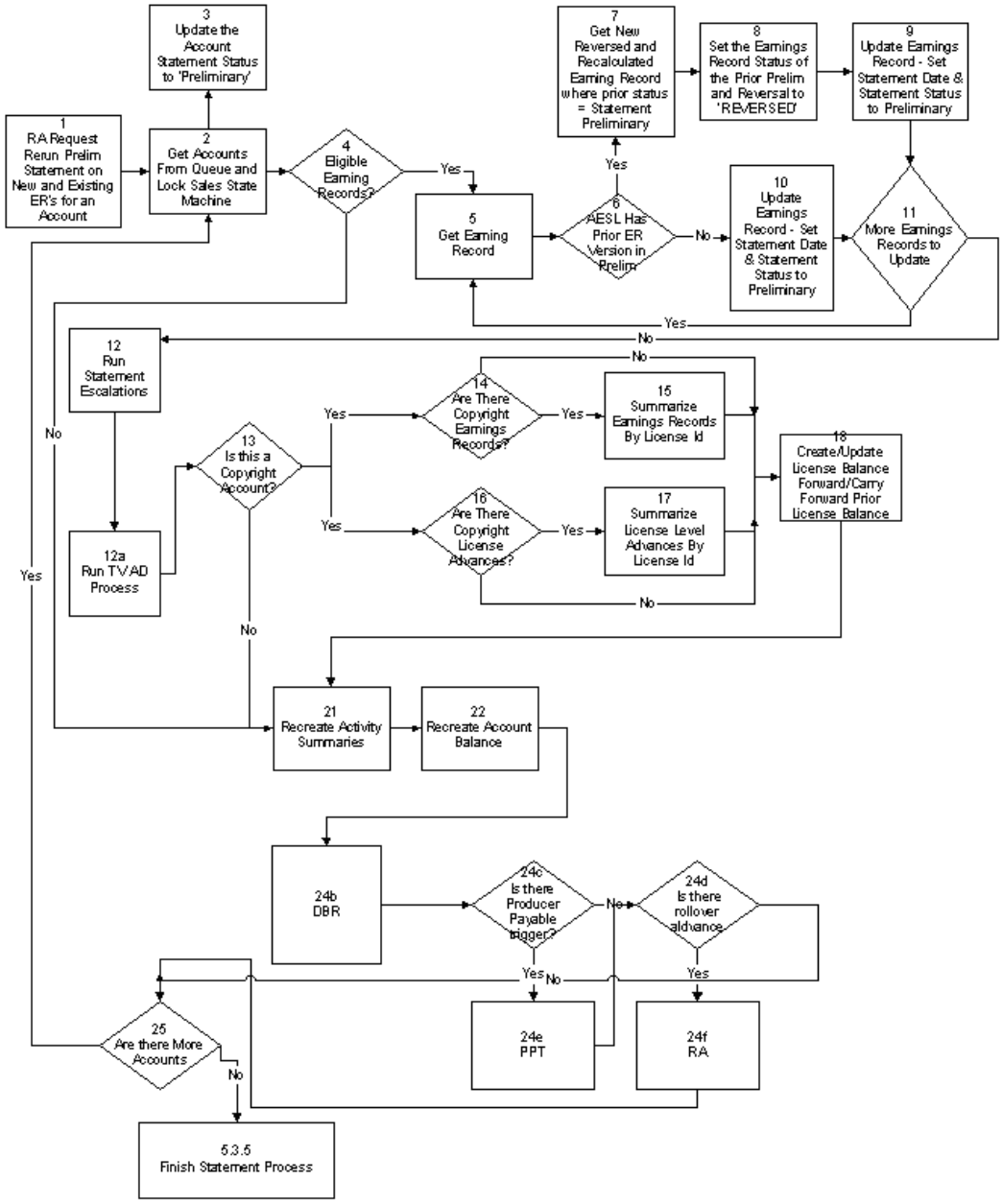
Account

- Представляет собой логическое отображение банковского счета получателя лицензионных отчислений.
- Накапливает информацию о заработках конкретного получателя за финансовый период.
- Содержит историческую информацию о заработках за предыдущие периоды.
- Позволяет в конце финансового периода собрать и передать во внешние системы заказчика информацию о количестве необходимых отчислений.

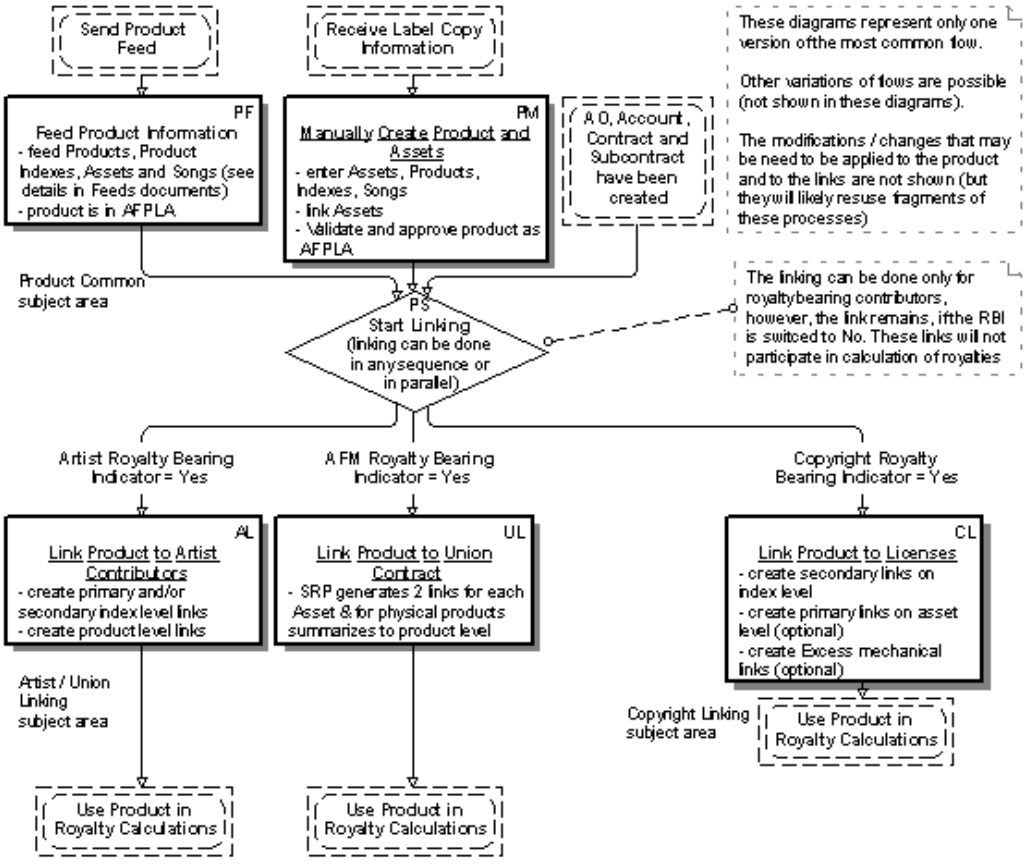
2.2.3 Внутридоменная логика.

Следует отметить, что каждый домен в свою очередь является композицией более мелких модулей, являющихся реализациями различных бизнес-процессов, происходящих в данном домене.

Типичным примером такого подмодуля может служить, например, финальная часть процесса пересчета текущего баланса на счете при поступлении в систему сведений о новых earnings (заработках) артиста:



Или, например, обзорная диграмма продукт-процесса:



Работа системы SRP - это результат взаимодействия множества сложных процессов. В силу этой сложности очень велика вероятность возникновения различных ошибок. Поэтому, при разработке системы SRP одной из важнейших поставленных задач было обеспечения нахождения и распознования проблем и ошибок на максимально возможно ранней стадии.

2.3 Обеспечение качества в проекте SRP.

2.3.1 Общая схема.

В целях обеспечения необходимого качества разрабатываемой системы в проекте SRP существует многоступенчатая система тестирования существующей функциональности.

Функциональность тестируется на следующих уровнях:

- Модульное тестирование (Unit-testing)
- Функциональное тестирование (Functional testing)
- Всевозможное тестирование в отделе контроля качества (QA - Quality Assurance department)

Необходимо отметить, что выполнение всех существующих модульных и функицональных тестов осуществляется для каждой успешной сборки проекта.

QA, разумеется, проводит тестирование регулярно (в соответствии с заранее оговоренным планом) передаваемых в этот отдел релизов (release) системы.

2.3.2 Модульное тестирование.

Модульные тесты создаются разработчиками параллельно с созданием любой системной функциональности. По принятому в рамках проекта соглашению, разработчик обязан написать тест для каждого метода реализуемого им интерфейса, проверяющий правильную работу метода на данных из допустимого набора и корректное сообщение об ошибке на недопустимых данных.

Преимущества модульного тестирования:

- обеспечивает первичное тестирование разработанной функциональности самим разработчиком.
- позволяет оценить работоспособность отдельных мелких модулей системы на этапе разработки.
- упрощает интеграцию модулей системы.
- на определенном уровне обеспечивает регрессионное тестирование, позволяя в будущем без опаски менять ранее разработанный код.

Недостатки модульного тестирования:

- не в состоянии проверить ошибки интеграции, проблемы производительности и другие недостатки системы.
- создание тестов отнимает у разработчиков достаточно много времени, которое могло бы пойти на разработку новой функциональности.

2.3.3 Функциональное тестирование.

В отличие от модульных, функциональные тесты создаются совместно разработчиками и специалистами отдела контроля качества(QA).

Процесс выглядит следующим образом:

- тестировщики создают Тестовый план (Test plan), состоящий из последовательности Тестовых ситуаций (Test cases). Каждый из них содержит описание последовательности шагов, которые надо выполнить, и определяет ожидаемые результаты.

- разработчики производят автоматизацию выполнения данного тестового плана и сравнения ожидаемых и реальных результатов после каждого шага.

- конфигурационные менеджеры (configuration managers) обеспечивают интеграцию нового функционального теста в тестовую среду, выполняющуюся на каждой успешной сборке системы.

Преимущества функционального тестирования:

- позволяет тестировать работоспособность системы в целом (или, как минимум, крупного процесса, объединящего несколько модулей системы).

- как правило, является достаточно сложным и детализированным способом проверки соответствия работы системы спецификации, поскольку план разрабатывается специалистами по тестированию.

- будучи исполняемым на регулярной основе, является очень хорошим способом оценки наличия серьезных проблем и ошибок в функциональности в процессе разработки.

Недостатки функционального тестирования:

- специфика функционального теста, опирающегося на взаимодействия нескольких модулей системы, приводит к тому, что небольшая

ошибка на ранней стадии (например, техническая, вроде на время отказавшей локальной сети) приводит к полностью неверным результатам, которые нет смысла анализировать.

- моделируя работу системы в целом (или крупного процесса в ее рамках) функциональные тесты зачастую требуют на выполнение весьма продолжительного промежутка времени.

Как легко видеть из приведенных списков преимуществ и недостатков, модульное и функциональное тестирование наиболее эффективны при работе в паре. При этом обеспечивается серьезная проверка качества системы, позволяющая в случае обнаружения проблемы с достаточной точностью локализовать место ее возникновения.

2.3.4 Тестирование в отделе QA.

Описание методологий тестирования системы в отделе QA выходит за рамки данной работы, поскольку она, прежде всего, пишется с позиции разработчика и о том, каким образом разработчик может первично проверить качество системы, тем самым заранее исправив незначительные и \ или легкообнаружимые проблемы.

В связи с этим нужно сказать лишь о том, как интегрированы процессы тестирования функциональности разработчиками и отделом обеспечения качества QA.

В проекте SRP команда разработчиков имеет право выпустить очередную версию системы в отдел QA в том и только том случае, если все модульные и функциональные тесты прошли успешно (либо есть

лишь незначительные проблемы, которые не помешают всестороннему тестированию системы и которые не могут быть на данный момент устранены разработчиками в силу непреодолимых обстоятельств).

После выпуска новой версии отдел QA запускает на переданной ему сборке свой набор специальных функциональных тестов (созданных без участия разработчиков) и называемых Smoke тестами, которые быстро и неглубоко тестируют базовые процессы системы.

В случае прохождения Smoke тестов релиз принимается и QA переходит в стадию всестороннего тестирования. В обратном случае релиз отвергается и возвращается к разработчикам для исправления обнаруженных критических проблем.

Описанная схема тестирования позволяет обеспечить многоступенчатое и тщательное проверку качества системы. Важное для любой системы, в SRP это приобретает особое значение в силу чрезвычайно сложной бизнес-логики.

3 ”Кэширование” данных при тестировании.

В предыдущей главе мы рассмотрели процесс модульного тестирования и подчеркнули его важность как одного из шагов многоступенчатой системы контроля и обеспечения качества в проекте SRP.

Одним из упомянутых недостатков было заявлено достаточно большое количество времени, которое требуется от разработчика для создания качественных модульных тестов. При этом стоит сказать, что помимо времени, уходящего на создание модульных тестов, чрезвычайно важное значение имеет и время их работы.

В самом деле, в ряде случаев разработчику необходимо дожидаться окончания выполнения модульных тестов:

- конкретного теста при отладке разрабатываемой функциональности
- набора тестов данного модуля перед коммитом своих изменений в систему контроля версий (VCS)
- всех наборов тестов перед выпуском очередной версии системы в QA

В связи с необходимостью соблюдения сроков разработки системы и общего снижения стоимости ее создания весьма важной задачей представляется минимизация времени работы модульных тестов. Тем самым разработчик сможет больше времени уделять созданию системной функциональности.

В рамках данного дипломного проекта был предложен, разработан и внедрен метод решения данной задачи.

3.1 Задача.

Модульные тесты в проекте SRP базируются на специальных тестовых сценариях (Unit-test scenario), которые создают новую сущность в базе данных (БД) каждый раз, когда это требуется. Такой подход к созданию тестовых данных имеет как преимущества, так и недостатки.

К преимуществам стоит отнести то, что набор модульных тестов может использоваться для стрессового тестирования, поскольку тесты могут выполняться в нескольких отдельных нитях (threads), не мешая друг другу - каждая нить будет использовать свои собственные данные, созданные "на лету".

Недостатком является то, что время работы модульных тестов при таком подходе достаточно велико. Дело в том, что система SRP имеет сложнейшую систему сущностей, имеющих многочисленные связи между собой. Эти сущности через систему реляционно-объектного отображения (ORM - Object-Relational Mapping) отображаются более чем в 800 таблиц в БД. Таким образом, создание объекта практически любой бизнес-сущности выливается в достаточно большой набор вставок и обновлений в БД. Разумеется, это приводит к значительным временным затратам.

В итоге была поставлена задача обеспечить возможность разделить подготовку тестовых данных и собственно тестирование. Такой подход позволит одновременно сохранить возможность использования модульных тестов для стрессового тестирования и предположительно ускорит выполнение модульных тестов в несколько раз.

3.2 Варианты работы модульных тестов.

В зависимости от выполняемой задачи, модульные тесты будут работать в одном из описанных ниже режимов.

3.2.1 Разработка функциональности и \ или новых тестов.

В данной фазе подготовка тестовых данных не требуется. В связи с этим, она будет отключена путем установки специального значения в глобальную переменную тестовой среды.

3.2.2 Подготовка тестовых данных.

Возможны два случая подготовки данных для модульных тестов:

- первоначальная подготовка данных после релиза нового дампа информации для модульных тестов.
- подготовка данных после создания нового или изменения существующего модульного теста.

На самом деле, разница состоит лишь в том, что в последнем случае необходимо подготовить данные лишь для новых (или измененных) тестов.

В общем случае необходимо выполнить следующие шаги:

1. убедиться, что целевой модульный тест работает так, как ожидается.

2. получить из системы контроля версий(VCS) и импортировать на БД-схему последнюю версию тестового дампа.

3. установить значение глобальной переменной, определяющей режим работы модульных тестов, равным "подготовка тестовых данных".

4. запустить выполнение модульных тестов. При этом фактически будет выполнена только та часть каждого теста, которая касается подготовки тестовых данных.

5. экспортировать измененный дамп в файл.

6. перевести приложение в режим "использования готовых данных"

7. запустить выполнение модульных тестов и убедиться, что не возникает никаких неожиданных проблем.

8. положить файл с измененным дампом и другие необходимые файлы в VCS, подвинуть метку (tag) на новую версию дампа.

Таким образом, мы получаем дамп с подготовленными тестовыми данными, не подвергшимися изменениям во время работы тестов просто потому, что собственно тесты на этом дампе не работали.

3.2.3 Использование подготовленных данных.

Во данном случае предполагается, что первоначальная подготовка данных уже была проведена.

Для выполнения модульных тестов необходимо сделать следующие шаги:

1. получить из VCS и импортировать на БД-схему последнюю версию тестового дампа. Он уже содержит заранее подготовленную информацию.

2. перевести приложение в режим "использования подготовленных данных".

3. запустить выполнение модульных тестов.

3.3 Дизайн предложенного решения.

3.3.1 Базовая часть решения.

Для достижения цели разделения подготовки тестовых данных и собственно выполнения теста, тестовые данные должны быть подготовлены на момент начала выполнения теста. В соответствии с перечисленными ранее возможными вариантами, приложение может быть запущено в трех режимах.

Данная таблица описывает эти режимы и соответствующие им переменные среды:

Режим	**Переменная среды**
разработки	UT_CACHING_MODE = "use_generated" or "g"
подготовки данных	UT_CACHING_MODE = "data_preparation" or "p"
использования данных	UT_CACHING_MODE = "use_cached" or "c"

Перед началом выполнения теста для каждого из неоходимых объектов будет выполнена следующася логика:

1. построено уникальное ссылочное имя, которое связано с ID объекта в БД через файл мэппинга. Это имя будет построено по

правилу: <class name>.<method name>.<variable name> конструктором класс TestObjectRefId. При этом имена класса и метода будут получены из стэк-трэйса, а имя переменной передано как параметр для коструктора.

2. если приложение работает в режиме "использования готовых данных", то объект будет получен по имеющемуся ID и начнется выполнение собственно теста.

3. в противном случае, необходимый для теста объект будет создан вручную, его БД ID будет сохранен в файле мэппинга. При этом собственно тест выполняться не будет.

В процессе исполнения пункта 2. может возникнуть ситуация, когда файл мэппинга не будет содержать информации о БД ID объекта, используемого в тесте (например, если тест был только что добавлен и дамп не содержит подготовленных для него данных).

В таком случае выполнится следующая логика:

1. необходимый объект будет создан "на лету".

2. тест будет выполнен (ведь приложение находиться в режиме "использования подготовленных данных", что подразумевает выполнение собственно теста).

3. по завершению выполнения всех запущенных тестов будет заново запущена процедура подготовки тестовых данных, при том только для тех тестов, которые в ней нуждаются (для всех остальных данные были созданы при первичной подготовке).

3.3.2 Необходимые изменения в модульных тестах и тестовых сценариях.

Поскольку тестовые сценарии широко используются в модульных тестах, имеет смысл сосредоточить основную логику подготовки тестовых данных именно там, тем самым минимизировав ее количество в коде самих тестов.

Для того, чтобы поддерживать использование заранее подготовленных данных, тестовые сценарии и модульные тесты должны быть модифицированы следующим образом:

(1) для каждого public createXXXXX() метода создания тестового объекта необходимо добавить соответствующй геттер. Например, для метода

```
public static ProductDataSliceKey createCopyrightProduct(int indexQuantity)
                                                                    throws BusinessException;
```

надо добавить следующий геттер:

```
public static ProductDataSliceKey getCopyrightProduct(int indexQuantity, TestObjectRefId refId)
throws BusinessException {
      // проверяем, работаем ли мы в режиме испjльзования подготовленных данных
      if (getDataCache().isDataPrepared()) {
          // получаем ID из файла мэппинга и загружаем объект из БД
         return getDataCache().getProductSliceKey(refId, ProductArea.COPYRIGHT);
      }
      ProductDataSliceKey productKey = createCopyrightProduct(indexQuantity);
      // сохраняем ID в файле мэппинга (работаем в режиме подготовки данных)
      getDataCache().storeTestObjectId(refId, productKey);
      return productKey;
   }
```

(2) соответствующий модульный тест изменить следующим образом:

```
public void testSecondaryStubGeneration() throws Exception {
        ProductDataSliceKey key = CopyrightLinkingScenario.getCopyrightProduct(3,
                                                    new TestObjectRefId("productKey"));
      // закончить тест, если работаем в режиме подготовки данных
      getDataCache().onDataPreparationFinished();
      //далее - код собственно теста
```

Важно отметить, что параметр **refID** должен быть проинициализирован созданием **new TestObjectRefId($variable name$)** в каждом тесте, поскольку его конструктор использует стэктрэйс для определения имени класса и метода, которые его вызывают.

При желании, логику подготовки данных можно включить и прямо в код теста:

```
public void testIndexSaveWithDisabledPrimaryLinksAssetWip() throws Exception {
      ProductDataSliceKey firstProductKey;
      ProductDataSliceKey secondProductKey;
      TestObjectRefId firstRefId = new TestObjectRefId("firstProduct");
      TestObjectRefId secondRefId = new TestObjectRefId("secondProduct");

      if (getDataCache().isDataPrepared()) {
         //Получить оба объекта
         firstProductKey= getDataCache().getProductSliceKey(firstRefId , ProductArea.COPYRIGHT);
         secondProductKey = getDataCache().getProductSliceKey(secondRefId ,
                                                              ProductArea.COPYRIGHT);
      } else {
         //создать оба объекта
         firstProductKey = ...
         secondProductKey = ...
         //сохранить ID обоих объектов
         getDataCache().storeTestObjectId(firstRefId, firstProductKey);
         getDataCache().storeTestObjectId(secondRefId, secondProductKey );
      }
      getDataCache().onDataPreparationFinished();
      // далее - код собственно теста
```

3.4 Используемые классы.

Здесь мы кратко опишем классы данного решения.

3.4.1 Class TestObjectRefId.

Данный класс описывает уникальное ссылочное имя тестового объекта.

Метод	Описание
TestObjectRefId(String variableName)	Конструктор для создания уникального ссылочного имени.
String getRefId()	Получение ссылочного имени в необходимом формате.

3.4.2 Class TestDataCache.

Данный класс обеспечивает функциональность получения тестовых объектов по уникальным ссылочным именам.

Метод	Описание
getInstance()	Возвращает глобальный статический экземпляр класса.
isDataPreparingMode()	Возвращает true если PREPARING_DATA = 1
isDataPrepared()	Возвращает true если DATA_PREPARED= 1
storeTestObjectId(TestObjectRefId refId, SrpObjectId<BType> id)	Сохраняет уникальное сслычное имя и БД ID объекта в файле мэппинга.
getVersionableObjectId(TestObjectRefId refId)	Загружает имеющийся версионируемый тестовый объект по его уникальному ссылочному имени
getNonVersionableObjectId(TestObjectRefId refId)	Загружает имеющийся неверсионируемый тестовый объект по его уникальному ссылочному имени
getTestObjectVo(TestObjectRefId refId, Class<AVoClass> targetVoClass)	Загружает тестовый объект из БД по его уникальному ссылочному имени и названию VO класса.
onDataPreparationFinished()	Завершает выполнение теста в случае, если приложение работает в режиме подготовки данных.

3.4.3 Class ProductTestDataCache.

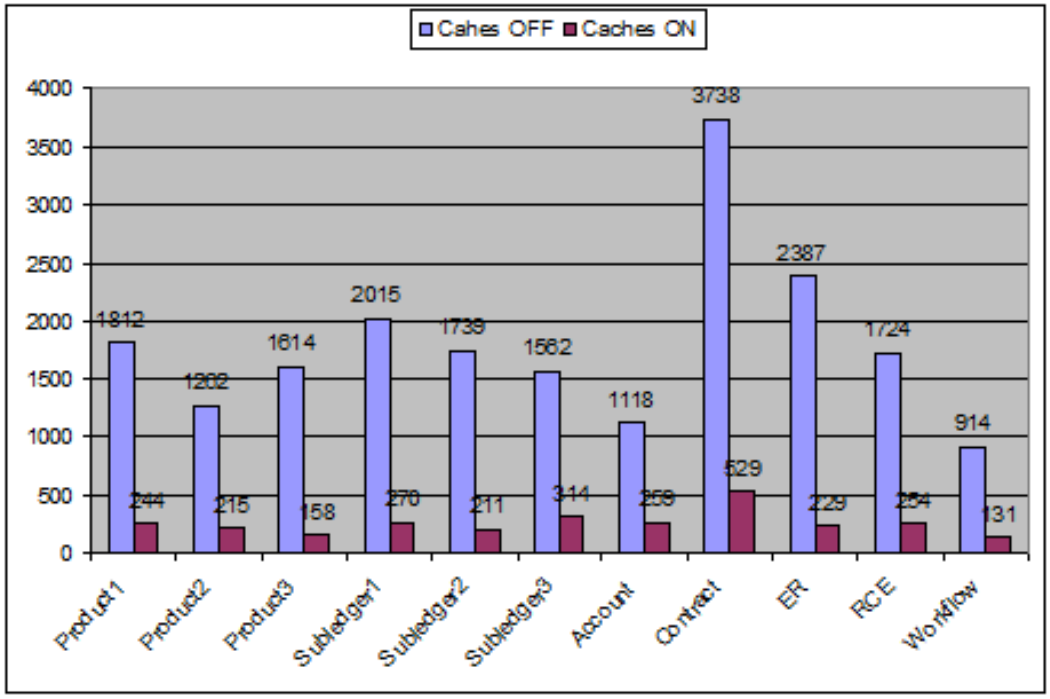
Данный класс добавляет к функциональности класса **TestDataCache** методы, специфичные для Продукт-домена.

Метод	Описание
getInstance()	Возвращает глобальный статический экземпляр класса.
storeTestObjectId(TestObjectRefId refId, ProductDataSliceKey productKey)	Сохраняет тестовый ProductDataSliceKey
storeTestObjectId(TestObjectRefId refId, ProductHeaderUnitedInfo productHeader)	Сохраняет тестовый ProductHeaderUnitedInfo
storeTestObjectId(TestObjectRefId refId, AssetDataSliceKey assetKey)	Сохраняет тестовый AssetDataSliceKey
storeTestObjectId(TestObjectRefId refId, AssetHeaderUnitedInfo assetHeader)	Сохраняет тестовый AssetHeaderUnitedInfo
getProductSliceKey(TestObjectRefId refId, ProductArea area)	Загружает тестовый ProductDataSliceKey
getProductHeaderUnitedInfo(TestObjectRefId refId)	Загружает тестовый ProductHeaderUnitedInfo
getAssetSliceKey(TestObjectRefId refId, AssetArea area)	Загружает тестовый AssetDataSliceKey
getAssetHeaderUnitedInfo(TestObjectRefId refId)	Загружает тестовый AssetHeaderUnitedInfo

4 Заключение.

Разработанное решение было реализовано, успешно протестировано и внедрено в проекте SRP.

Использование данного решение позволило значительно сократить время работы модульных тестов. Полученные результаты проиллюстрированны на следующем графике:



Использование стратегии предварительной подготовки данных позволило в среднем сократить время выполнения модульных тестов в проекте SRP в **8 (восемь)** раз, что, безусловно, является отличным результатом.

5 Список литературы.

1. Фаулер М. Архитектура корпоративных программных приложений / Вильямс пабликейшнс, 2006

2. Bauer C., King G. Hibernate in action / Manning Publications, 2005

3. Gamma E., Helm R., Johnson R., Vlissides J. Design Patterns / Addison-Wesley, 2001

4. Internal SRP Business process and Domian analysis specification documents.