Содержание

Стр.

Введение

Для того чтобы понять, насколько актуальна тема управления проектами необходимо дать определение термина «проект».

Проект (от лат. *projectus* — брошенный вперед, выступающий, выдающийся вперед) — замысел, идея, образ, воплощенные в форму описания, обоснования, расчетов, чертежей, раскрывающих сущность замысла и возможность его практической реализации.

Проект — это работы, планы, мероприятия и другие задачи, направленные на создание уникального продукта (устройства, работы, услуги).

Таким образом под определение проекта попадает практически любая деятельность человека, характеризующаяся такими признаками как:

1) ограниченность во времени;
2) конкретность, достижимость результата;
3) наличие плана действий;

Естественно желать, чтобы деятельность человека была успешной, то есть чтобы проекты завершались в срок и без превышения требований к ресурсам. Однако даже в современном мире далеко не все проекты заканчиваются успешно. Многие даже не доходят до своего логического завершения, а некоторые вообще тянутся годами, потребляя как временные так и человеческие ресурсы.

Особенно остро данная проблема чувствуется в области разработки программного обеспечения. Так, в исследовании *Standish Group* под названием «*Chaos Report*» от 1995 года говорится, что в США в год тратится более 250 миллиардов долларов на реализацию примерно 175 000 программных проектов. Средняя стоимость разработки проекта для крупной кампании составляет более 2 000 000 долларов, для компании средних размеров более 1 000 000 долларов, для небольшой компании — порядка полумиллиона долларов. При этом большинство проектов проваливаются. *Standish Group* выяснила, что работа над 31,1% проектов прекратится еще до их логического завершения, 52,7% проектов будут стоить на

89% дороже изначальной стоимости. Так в 1995 году частные и государственные компании США потратили 81 миллиард долларов на программные проекты, разработка которых не будет завершена, 59 миллиардов долларов на программные проекты, завершенные с сильным опозданием. При этом речь не идет о сверхсложных системах. Провальные проекты в большинстве своем также просты как рядовая система учета заказов.

Несмотря на то, что данному исследованию уже 20 лет, опыт автора подсказывает, что ситуация в сфере разработка программных продуктов за прошедшее время не сильно изменилась. Многие проекты все также затягиваются или проваливаются. Следовательно обозначенная тема будет актуальна до тех пор, покуда будут существовать проекты.

Целями данной работы является исследование различных методологий управления разработкой программных продуктов, нахождение оптимальной методологии для управления разработкой программных продуктов в сфере решений для управления и поддержки бизнеса, разработка автоматизированной информационной системы на основе выбранной методологии для упрощения, рационализации и стандартизации взаимодействия между заказчиком разработки и командой, реализующей программный продукт, оптимизации работы команды.

В данной работе решаются вопросы взаимодействия между разработчиками, конечными пользователями, заказчиками и спонсорами разработки программного продукта, вопросы эффективной организации команды разработки, рассматриваются полезные практики, увеличивающие шансы на успех при разработке программного продукта. Также анализируются факторы, наличие которых приводят к задержке выпуска программного продукта, перерасходам, низкой эффективности команды разработки.

1. Исследование методологии *Scrum*

В данном разделе охватываются вопросы о том что такое проект, какова специфика программных проектов. Также освещаются факторы успеха программных проектов и рассматриваются подходы к автоматизации управления разработкой программного обеспечения.

1.1 Понятие управления проектами

Управление проектом — это управление процессом его реализации. В свою очередь реализация проекта — это комплекс мер, дел и действий, направленных на достижение целей проекта. Таком образом, управление проектом - это управление комплексом мер, дел и действий, направленное на достижение целей проекта.[3]

Довольно любопытен тот факт, что при всей понятности приведенного выше определения, не у всех организаций получается производить грамотное управление проектом. Кажущаяся простота данного процесса является обманчивой. Это также верно и для программных проектов. Даже при условии наличия грамотного менеджера проекта, достаточного количества ресурсов, адекватной оценке сроков возможен срыв проекта из-за неповоротливости команды, из-за проблем с коммуникациями между заказчиком проекта и командой разработки. Так же может произойти следующее — проект разработан, протестирован и сдан вовремя без перерасходов по финансам, однако им никто не пользуется. Очевидно, что в данной ситуации обратная связь между командой разработки и конечным пользователем была очень слабой или ее вообще не было. Такой проект также нельзя считать успешным по причине его убыточности для заказчика. Следовательно в контексте управления разработкой программных продуктов данное определение необходимо дополнить. Автор в своем исследовании сформулировал следующее определение.

Управление проектами — это комплекс мер, дел и действий направленный на достижение целей проекта путем создания комфортных условий работы для исполнителей проекта, установление и поддержка качественной и своевременной связи между исполнителями, заказчиком и конечным пользователем.

Как видно, в данном определении фигурируют заказчик, исполнитель проекта и конечный пользователь. По мнению автора эти действующие лица можно считать тремя китами в управлении разработкой программного продукта. Без заказчика не возникнет идеи проекта, без исполнителя его некому будет реализовывать, а если не спрашивать мнения конечного пользователя о реализуемом проекте, то есть риск сделать проект, который будет нравится только заказчику или только исполнителю.

1.2 Факторы, ведущие к успеху

Как у любого другого процесса у разработки программного обеспечения имеются факторы как ведущие к успеху, так и влекущие к провалу. Рассмотрим, какие причины приводят к успеху разработку программного продукта.

Для того чтобы понять причины следствия необходимо определить, что именно мы считаем успехом разработки программного продукта?

Автор выделяет следующие критерии, при наличии которых можно считать разработку программного продукта успешной:

1. Проект уложился в бюджет;
2. В проекте реализована вся заявленная функциональность;
3. Проект выполнен в срок;
4. Все функции программного продукта выполняются по указанным в документации сценариям при любых действиях пользователя;

Рассмотрим каждый пункт подробнее и выясним, как можно добиться такого результат при работе над проектом.

Проект способен уложиться в бюджет при условии, что оценка его стоимости (учитывая погрешности) произведена адекватно, в процессе работы над проектом не добавляется новых "быстрых" задач, которые накапливаются как снежный ком и отвлекают исполнителя от основной работы. Адекватная оценку стоимости проекта может произвести только эксперт с опытом реализации подобных проектов. Однако в некоторых случаях оценка не устраивает заказчика и он пытается занизить цену,

что приводит либо к урезанию возможностей, либо к увеличению сроков.

Перейдем к рассмотрению реализации заявленной функциональности. Самое главное в этом пункте - должен быть составлен список конкретных задач по проекту. При этом задачи должны быть поставлены правильно. Они должны быть количественно измеримы, достижимы, актуальны.

В любом проекте во время реализации список задач имеет свойство раздуваться, дополняться "быстрыми" задачами. Так же старые задачи изменяются и дополняются новыми требованиями. Чтобы функциональность реализовывалась в полном объеме следует либо составлять конечный список задач в начале проекта и ни в коем случае ничего не менять до окончания проекта, либо работать над проектом итеративно, реализуя небольшую часть задач за короткий промежуток времени и производя переоценку приоритетов в конце каждой итерации. Первый способ весьма неэффективен при разработке средних и больших проектов. Невозможно учесть все требования в самом начале проекта. Если руководствоваться таки принципом, то заказчик постарается собрать в одном техническом задании все возможные требования, несмотря на то, что эти требования могут противоречить друг другу или быть просто ненужными для конкретно этого проекта. Очевидно, что такой проект затянется.

Таким образом второй способ ведения проекта - итеративный - наиболее эффективен в плане реализации всей необходимой функциональности. Данный метод также хорошо поддерживает изменения в требованиях.

Четвертый пункт подразумевает наличие достаточного опыта в проектировании программ у разработчиков и четкость, конкретность поставленных заказчиком задач. Если будут соблюдены два этих условия, то реализуемая функция будет выполнятся только по тем сценариям, которые будут указаны в документации.

1.3 Обзор методологии *Scrum*

Приведенные выше требования к работе над программным проектом реализуются в семействе гибких методик разработки среди которых особенно выделяется методология *Scrum*.

Scrum (от англ. *scrum* «толкучка») — методология управления проектами, активно применяющаяся при разработке информационных систем для гибкой разработки программного обеспечения. *Scrum* чётко делает акцент на качественном контроле процесса разработки. Кроме управления проектами по разработке ПО, *Scrum* может также использоваться в работе команд поддержки программного обеспечения (*software support teams*), или как подход управления разработкой и сопровождением программ.

Скрам (*Scrum*) это набор принципов, на которых строится процесс разработки, позволяющий в жёстко фиксированные и небольшие по времени итерации, называемые спринтами (*sprints*), предоставлять конечному пользователю работающее ПО с новыми возможностями, для которых определён наибольший приоритет. Возможности ПО к реализации в очередном спринте определяются в начале спринта на этапе планирования и не могут изменяться на всём его протяжении. При этом строго фиксированная небольшая длительность спринта придаёт процессу разработки предсказуемость и гибкость.

Спринт это итерация в скраме, в ходе которой создаётся функциональный рост программного обеспечения. Жёстко фиксирован по времени. Длительность одного спринта от 2 до 4 недель. В отдельных случаях, к примеру согласно *Scrum* стандарту *Nokia*, длительность спринта должна быть не более 6 недель. Тем не менее, считается, что чем короче спринт, тем более гибким является процесс разработки, релизы выходят чаще, быстрее поступают отзывы от потребителя, меньше времени тратится на работу в неправильном направлении. С другой стороны, при более длительных спринтах команда имеет больше времени на решение возникших в процессе проблем, а владелец проекта уменьшает издержки на совещания, демонстрации продукта и т. п. Разные команды подбирают длину

спринта согласно специфике своей работы, составу команд и требований, часто методом проб и ошибок. Для оценки объема работ в спринте можно использовать предварительную оценку, измеряемую в очках истории. Предварительная оценка фиксируется в бэклоге проекта. На протяжении спринта никто не имеет права менять список требований к работе, внесённых в бэклог спринта.

Бэклог проекта это список требований к функциональности, упорядоченный по их степени важности, подлежащих реализации. Элементы этого списка называются «пожеланиями пользователя» (*user story*) или элементами бэклога (*backlog items*). Бэклог проекта открыт для редактирования для всех участников скрам процесса.

Бэклог спринта содержит функциональность, выбранную владельцем проекта из бэклога проекта. Все функции разбиты по задачам, каждая из которых оценивается скрам-командой. Каждый день команда оценивает объем работы, который нужно проделать для завершения спринта.

Диаграмма, показывающая количество сделанной и оставшейся работы. Обновляется ежедневно с тем, чтобы в простой форме показать подвижки в работе над спринтом. График должен быть общедоступен.

Существуют разные виды диаграммы:

- диаграмма сгорания работ для спринта показывает, сколько уже задач сделано и сколько ещё остаётся сделать в текущем спринте;

- диаграмма сгорания работ для выпуска проекта показывает, сколько уже задач сделано и сколько ещё остаётся сделать до выпуска продукта (обычно строится на базе нескольких спринтов);

Методология скрам необходима но не достаточна для успешного управления разработкой ПО. Еще одним немаловажным фактором будет наличие автоматизированной информационной системы управления проектами, основанной на данной методологии.

Рассмотрим, почему возникает необходимость в такой системе. Если ее нет, то вся работа по записи, оценке и распределению пользовательских историй будет вестись вручную в электронных документах, в тетрадях или на досках. Ко всему

прочему часто требуется наглядная отчетность по выполненным пользовательским историям, по выполненным задачам. Заказчик желает видеть прогресс не только в виде прироста возможностей его продукта, но и в виде бумажных отчетов. А рисовать диаграммы сгорания задач от руки или в каком-либо графическом редакторе также не очень удобно. Поэтому оптимальным решением будет переложить всю рутинную работу по хранению, отслеживанию пользовательских историй, формированию отчетов на автоматизированную информационную систему.

1.4 Достоинства применения системы управления проектами на основе методологии *Scrum*

Система управления проектами на основе методологии скрам позволит поддержать внедрение самой методологии на каждом уровне работы в организации. Так как сама методология концентрируется на взаимодействии между людьми, то и система будет делать упор на взаимодействие, на более тесное общение, не упуская однако и значимости управленческой составляющей.

Современные технологии позволяют создавать возможности для интерактивного общения внутри команды, такие как корпоративный чат или видео конференции. Данная возможность будет весьма полезна, если все члены команды не могут разместиться в одном месте.

Система учета задач позволит формировать стандартизированные отчеты по текущему проекту, оформлять графики и представлять другую необходимую информацию. А подсистема шаблонов позволит гибко настраивать нужные возможности.

Система отслеживания задач позволит определить на какой стадии находится каждая конкретная задача. Также система управления проектами сможет самостоятельно рассчитывать коэффициент полезного действия команды, помогая таким образом объективно оценивать количество задач, которое возможно реализовать за одну итерацию.

Таким образом можно сделать вывод, что необходимо либо найти систему управления программными проектами, удовлетворяющую потсавленным условиям, либо разработать ее самостоятельно.

1.5 Обоснование необходимости разработки

На данный момент существует достаточно много систем управления программными проектами. Данные системы используются как для управления разработкой ПО на заказ, так и для внутренних разработок компаний. Вследствие ограниченности времени автором рассматривались далеко не все программные решения, призванные стандартизировать и автоматизировать разработку программных проектов. Список продуктов-аналогов проекта, разрабатываемого в данном дипломе приведен в таблице 1.1.

Т а б л и ц а 1.1 - Системы управления проектам

Система управления проектами	Поддерживаемые модели жизненного цикла проекта	Основные возможности
Atlassian Jira	Водопад, *SCRUM*,	Отслеживание ошибок, моделирование и управление процессами, планирование задач, совместная работа разработчиков,
MS Project	Водопад, метод кратчайшего пути	Диаграмма Гантта, планирование задач, управление ресурса
Мегаплан	Водопад, водопад с обратными связями	Планирование задач, ведение клиентской базы и базы сотрудников, управленческий и финансовый учет
Basecamp	Водопад с обратной связью.	Планирование задач, разграничение доступа к задачам и проектам,, визуализация прогресса в виде истории
Wrike	Водопад с обратной связью	Планирование, приоритезация задач, совместная работа, визуализация загрузки работников и выполнения задач с помощью диаграммы Гантта
Spider Projects	Водопад с обратной связью	Планирование и приоритезация задач, управление ресурсами, визуализация загрузки на диаграмме Гантта, широкие возможности по работе с ресурсами.
Asana	Водопад, возможна итеративная разработка	Планирование задач, подключение тэгов, назначение задачам сроков

Продолжение таблицы 1.1

Система управления проектами	Поддерживаемые модели жизненного цикла проекта	Основные возможности
Redmine	Водопад, возможно итеративная разработка	Планирование задач, создание wiki страниц, формирование отчетов.

Как видно, асе приведенные системы реализуют одни и те же функции. Многие имеют свои специфические возможности, однако при ближайшем рассмотрении автор пришел к следующим выводам. Большинство приведенных систем не имеют русскоязычного интерфейса. Исключения - *«Wrike»*, *Redmine»*, *«Spider Project»*. Подавляющее большинство не поддерживает методологию *SCRUM (*за исключением *«Atlassian Jira*»), по которой ведется разработка программных продуктов в компании автора. Ни у одной системы нет возможности взаимодействия между членами группы разработки или между разработчиками и заказчиками в реальном времени в виде чата или видеоконференции. Данный аспект немаловажен в том случае, если члены группы разработки находятся далеко друг от друга и не могут ежедневно собираться на *SCRUM*-митинги в одном месте. Конечно, для устранения данного недостатка можно использовать стороннее ПО, однако намного удобнее было бы иметь встроенную в систему возможность связи в реального времени с функцией создания и изменения расписаний конференций.

Далее, в компании, для которой автор разрабатывает данный проект использую методику *Kanban*, для отслеживания стадий проекта и реализации принципа «точно в срок». На данный момент указанная методология реализована только в *Atlassian Jira,* в остальных же системах данный функционал реализован либо в виде расчета кратчейшего пути и построения диаграммы Гантта, что не совсем подходит под концепцию *Kanban*, либо не реализован вообще.

Наконец вопрос цены. *Asana, Wrike, Basecamp* предоставляют возможность бесплатного использования данных систем для ограниченного количества пользователе. Количество пользователей варьируется от пяти до двадцати. Если системами пользуются большее количество человек, то приходится платить ежемесячную абонентскую плату. Другие системы могут предоставляться в демо-

версиях с урезанными возможностями либо на пробный период в течении тридцати дней.

«*Spider Project*» является лишь настольной системой и не имеет функций совместной работы над проектом. «*Atlassian Jira*» нужно разворачивать на собственном сервере.

Ввиду перечисленных выше недостатков существующих систем автором предлагается разработать собственную систему управления проектами, которая реализовывала бы все необходимые возможности планирования задач, отслеживания стадий выполнения проектов, совместной работы над проектами,, формирования отчетов, декларируемые методологиями *SCRUM* и *Kanban.* Разработка собственной системы также позволит сэкономить средства на оплачивание лицензий сторонних систем, а при условии реализации продаж может приносить прибыль.

Далее рассматривается способ реализации данной системы, описываются ее возможности и выбираются технологии, требуемые для ее реализации.

1.6 Требования к разрабатываемому программному продукту

На основании изложенного в разделе «Обоснование актуальности разработки» автор составил список требований, предъявляемых к разрабатываемой системе управления проектами по методологии *SCRUM.*

1. Возможность создания, редактирования и удаления задач (пользовательских историй);
2. Возможность создания, редактирования и удаления проектов;
3. Возможность создания, редактирования и удаления спринтов. Спринты должны быть ограничены по времени;
4. Возможность перемещения задач из списка проекта в список задач спринта;
5. Возможность формирования отчета в виде диаграммы сгорания задач;
6. Система должна автоматически рассчитывать КПД команды;

7. Возможность регистрации пользователей;

8. Разграничение возможностей пользователей по ролям;

9. Возможность прикреплять к проекту пользователей;

10. Возможность объединять пользователей с ролью рзарботчика в команды;

11. В системе должна быть возможность определения стадий процесса реализации задачи и реализована возможность отслеживания перемещения задачи по стадиям. Данный функционал должен быть выполнен в стиле методики K*anban*;

12. В системе должна быть возможность комменитрования задач;

13. В системе должна быть возможность прикреплений графических файлов к задачам, спринтам, проектам;

14. При закрытии задачи система должна требовать от пользователя-разработчика краткого отчета по выполненной работе. При формировании общего отчета по спринту система должна компилировать отчеты каждого, кто работал над задачей, сроки работы в единый удобочитаемый отчет;

15. При закрытии проекта система должна требовать обязательного отчета по проекту;

16. В системе должна быть возможность интерактивного общения между участниками проекта. В качестве реализации данной возможности предполагается чат;

2. Моделирование проектируемой системы

В данном разделе приводятся диаграммы, описывающие систему моделей программного продукта, разрабатываемого в данном дипломном проекте. Диаграммы разработаны на языке *UML*.

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

2.1 Диаграммы прецедентов.

Как было описано выше, среди пользователей разрабатываемого программного продукта выделяются две ключевые роли: p*roduct owner* (владелец продукта) который отвечает за постановку задач и их приоритезацию, а также за приемку готового продукта в конце каждого спринта и, собственно, команда разработки, которая отвечает за разработку программного продукта и тестирование. Следовательно моделирование будет производиться с точек зрения этих двух ролей, с учетом специфичных требования от каждого действующего лица. Ниже приведены диаграммы прецедентов, и их описания моделирующие взаимодействия между разрабатываемой системой и каждой из указанных ролей.

Диаграмма прецедентов (диаграмма вариантов использования) в *UML* - диаграмма, отражающая отношения между актерами и прецедентами и являющаяся составной частью модели прецедентов, позволяющей описать систему на концептуальном уровне.

Product Owner (владелец продукта) является представителем заказчика или самим заказчиком в случае разработки ПО на заказ. В случае проекта, разрабатываемого внутри компании, как-то коммерческие продукты или программные комплексы для использования внутри компании в качестве владельца продукта может выступать менеджер проекта. Диаграмма прецедентов с точки зрения владельца продукта приведена на рисунке 2.1.

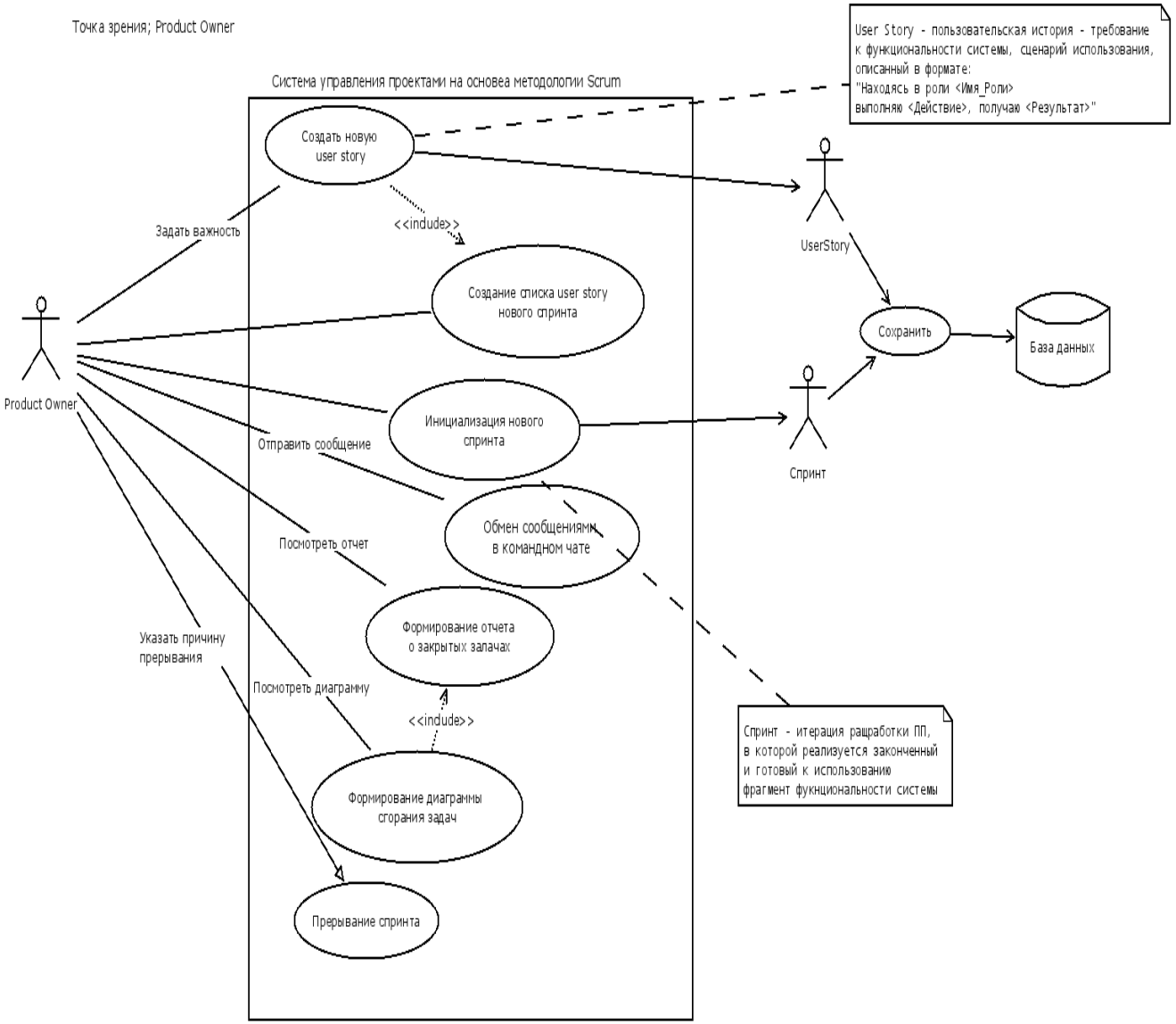

Рисунок 2.1 - Диаграмма прецедентов, точка зрения владельца продукта

В приведенной выше диаграмме выделены следующие прецеденты:

1. Создать новую u*ser story.*

Действующие лица: *product owner.*

Предусловия: пользователь находится на странице создания *user story.*

Основной сценарий:

а) пользователь нажимает кнопку «Создать новую историю»

б) Система предоставляет форму ввода данных

в) Пользователь вводит необходимые данные и отпарвляет форму

г) Система создает новую *user story*;

2. Создание списка u*ser story* для нового спринта.

Действующие лица: *product owner.*

Минимальное условие: емкость спринта определена разработчиками. (Прим. Емкость спринта — количество задач, которое команда разработки способна выполнить за один спринт. Емкость спринта определяется в очках историй. Очки историй — относительная величина применяемая для оценки сложности одной задачи относительно другой. Может изменяться в пределах от нуля до бесконечности.)

Предусловия: пользователь создал новый спринт и перешел на страницу его редактирования;

Основной сценарий:

а) Система отобразила бэклог (список user story) проекта и и текущего спринта;

б) Пользователь переносит наиболее важные задачи из бэклога проекта в бэклог спринта;

в) Пользователь вводит необходимые данные и отпарвляет форму;

г) Система запрещает перенос, когда емкость спринта заполнена;

3. Инициализация нового спринта

Действующие лица: *product owner.*

Минимальное условие: пользователь имеет права на создание спринта.

Предусловия: Предыдущий спринт окончен.

Основной сценарий:

а) Пользователь нажимает кнопку «Создать новый спринт»

б) Система создает черновик спринта и предоставляет форму для ввода цели спринта, даты начала, набора бэклога спринта;

в) Пользователь вводит необходимые данные и отправляет форму

г) Система создает новый спринт;

Альтернативные сценарии:

в.1) Не введена цель спринта

в.1.1) Система выдает сообщение о том что необходимо ввести цель;

в.1.2) Переход к шагу (б)

в.2) Не введена дата начала спринта

в.2.1 Система устанавливает текущую дату как дату начала спринта;

в.2.2 Переход к шагу (г)

4. Обмен сообщениями в командном чате.

Действующие лица: *product owner.*

Предусловия: пользователь находится на странице чата.

Основной сценарий:

а) Пользователь вводит сообщение и отправляет форму;

б) Система отправляет сообщение в чат;

Альтернативные сценарии:

а.1) Пользователь не ввел сообщение

а.2) Система бездействует;

5. Формирование диаграммы сгорания задач.

Действующие лица: *product owner.*

Предусловия: пользователь находится на странице спринта.

Основной сценарий:

а) Пользователь нажимает кнопку «Показать отчет»;

б) Система выводит диаграмму сгорания задач в составе общего отчета;

6. Формирование отчета по выполненным задачам.

Действующие лица: *product owner.*

Предусловия: Пользователь находится на странице спринта.

Основной сценарий:

а) Пользователь нажимает кнопку «Показать отчет»;

б) Система выводит отчет по выполненным задачам;

7. Прерывание спринта.

Действующие лица: *product owner.*

Минимальное условие: у пользователя должны быть права на прерывание спринта.

Предусловия: Пользователь находится на странице спринта.

Основной сценарий:

а) Пользователь нажимает кнопку «Прервать спринт»;

б) Система выводит предупреждение;

в) Пользователь продолжает прерывание;

г) Система выводит форму ввода причины прерывания

д) Пользователь вводит причину прерывания и отправляет форму;

е) Система прерываем спринт и сохраняет отчет о прерывании;

Альтернативные сценарии:

г.1) Пользователь не вводит причину

г.1.1) Система выдает сообщение об ошибке

г.1.2) Переход к шагу (г)

Разработчик - член команды разработки, также является основным действующим лицом в контексте процесса разработки программного обеспечения. Поэтому критично важно выделить и описать все требуемые для его нормальной работы функции системы. Диаграмма прецедентов, описывающая взаимодействие разработчика с проектируемой системой приведена на рисунке 2.2.

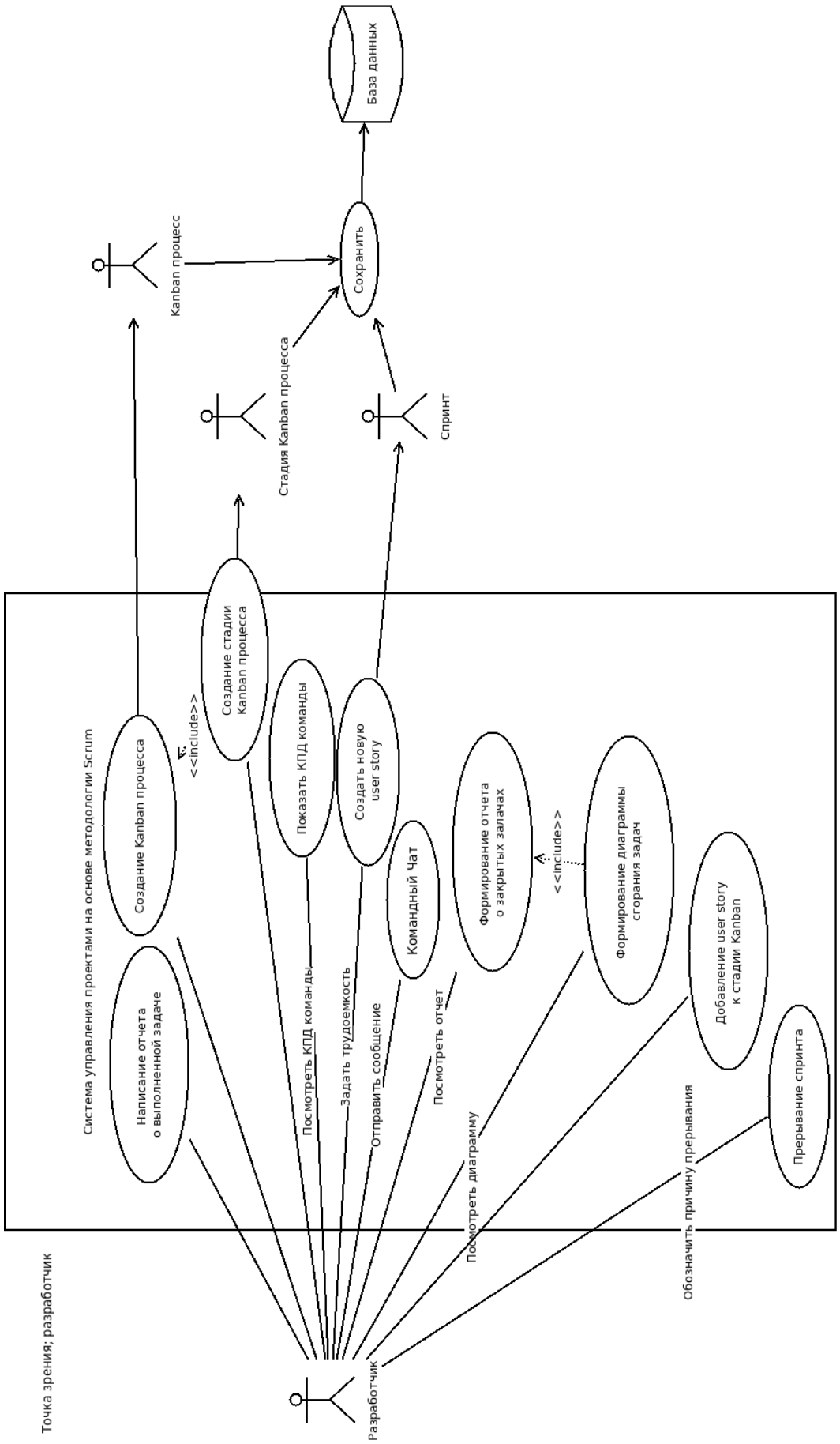

Рисунок 2.2 - Диаграмма прецедентов, точка зрения разработчика

Очевидно, что большая часть функций аналогична функциям владельца продукта, поэтому далее описываются только те прецеденты, которые не были указаны в предыдущей диаграмме.

1. Написать отчет о выполненной задаче.

Действующие лица: разработчик.

Предусловия: пользователь находится на странице просмотра задачи.

Основной сценарий:

а) Пользователь нажимает кнопку "Завершить задачу";

б) Система выводит форму для ввода отчета;

в) Пользователь вводит отчет и отправляет форму;

г) Система сохраняет отчет и выводит сообщение об успешном сохранении;

Альтернативные сценарии:

в.1) Пользователь не ввел отчет

в.2) Система выдает сообщение о необходимости отчета;

а.3) Переход к шагу б;

2. Показать КПД (коэффицент полезного действия - отношение количества реально выполненных задач к заявленному) команды.

Действующие лица: разработчик.

Предусловия: пользователь вошел в систему.

Основной сценарий:

а) Система выводит в информационном окне КПД команды в очках историй.

3. Создать *Kanban* процесс.

Действующие лица: разработчик.

Предусловия: пользователь находится на странице создания *Kanban* процесса.

Основной сценарий:

а) Система отображает форму создания Kanban процесса;

б) Пользователь вводит название и описание процесса;

в) Пользователь создает стадии процесса;

г) Пользователь нажимает кнопку "Сохранить";

д) Система сохраняет процесс и его стадии и выводит сообщение об успешном сохранении;

Альтернативные сценарии:

б.1) Пользователь не ввел названия процесса

б.2) Система выдает сообщение о необходимости названия;

б.3) Переход к шагу а;

г.1) Пользователь не создал ни одной стадии для данного процесса;

г.2) Системы выводит сообщение о необходимости создания хотя бы одной стадии;

г.3) Переход к шагу в;

4. Создание стадии *Kanban* процесса.

Действующие лица: разработчик.

Предусловия: пользователь находится на странице создания или редактирования *Kanban* процесса.

Основной сценарий:

а) Система отображает список стадий процесса на форме редактирования *Kanban* процесса;

б) Пользователь нажимает кнопку "Добавить стадию";

в) Система создает новую стадию и отображает форму ее редактирования;

г) Пользователь вводит название и описание стадии и нажимает кнопку "Сохранить";

д) Система сохраняет стадию процесса и выводит сообщение об успешном сохранении;

Альтернативные сценарии:

г.1) Пользователь не ввел названия стадии процесса

г.2) Система выдает сообщение о необходимости названия;

г.3) Переход к шагу в;

5. Добавление *user story* к стадии *Kanban* процесса.

Действующие лица: разработчик.

Предусловия: пользователь находится на странице создания или

редактирования *user story*;

Основной сценарий:

а) Система отображает список стадий процесса на странице редактирования user story;

б) Пользователь выбирает нужную стадию и нажимает кнопку "Прикрепить";

в) Система прикрепляет user story к выбранной стадии и выполняет сохранение;

Альтернативные сценарии:

б.1) Пользователь не выбрал стадию и нажал кнопку "Прикрепить"

б.2) Система выдает сообщение о необходимости выбора стадии;

б.3) Переход к шагу а;

На основе приведенных диаграмм прецедентов можно строить диаграммы взаимодействия по каждой отдельной функции. Диаграммы взаимодействия - это диаграммы, на которых показано взаимодействие объектов (обмен между ними сигналами и сообщениями), упорядоченное во времени, с отражением продолжительности обработки и последовательности их проявления. Диаграммы взаимодействия отражают логику процесса взаимодействия между действующими лицами, в частности между разработчиком и системой.

2.2 Диаграммы взаимодействия.

1. Создание пользовательской истории. Данная диаграмма описывает взаимодействие между разработчиком и системой управления проектами во время создания пользовательской истории. Диаграмма приводится на рисунке 2.3.

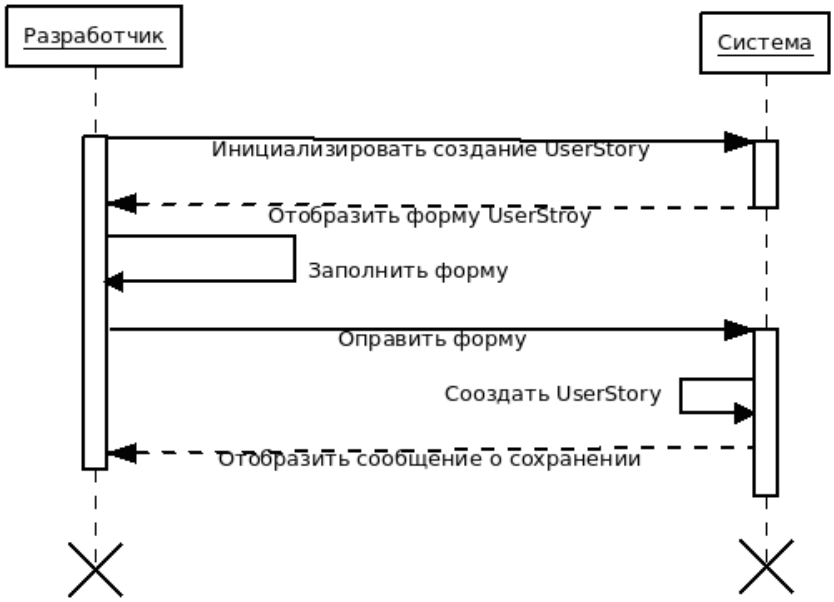

Рисунок 2.3 - Диаграмма взаимодействия функции "создание пользовательской истории"

2. Инициализация нового спринта. Данная диаграмма отражает логику взаимодействия между пользователем и системой при создании нового спринта и набора списка пользовательских историй для создаваемого спринта. Диаграмма приведена на рисунке 2.4.

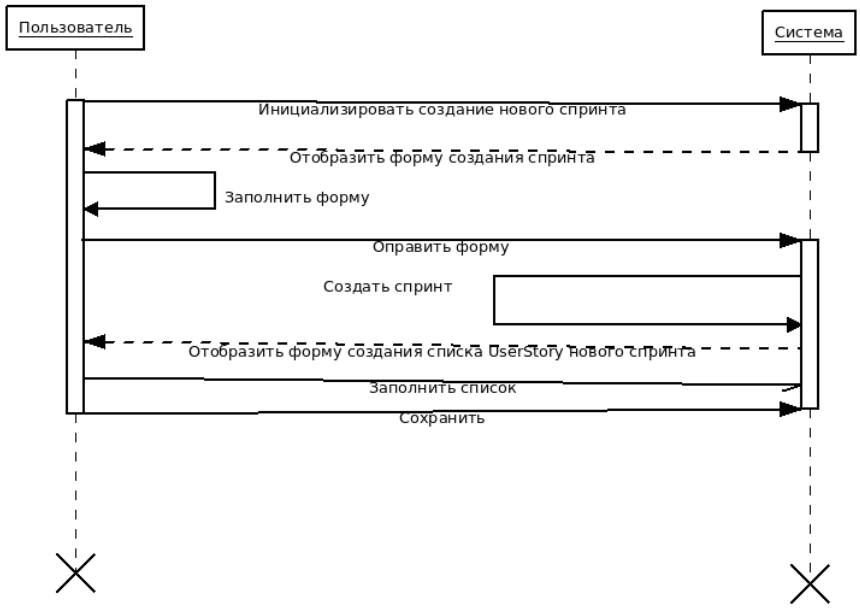

Рисунок 2.4 - диаграмма взаимодействия функции "Инициализация спринта"

3. Создание списка пользовательский историй нового спринта. Данная

диаграмма описывает логику взаимодействия между пользователем и системой во время создания списка пользовательских историй нового спринта. Диаграмма приведена на рисунке 2.5.

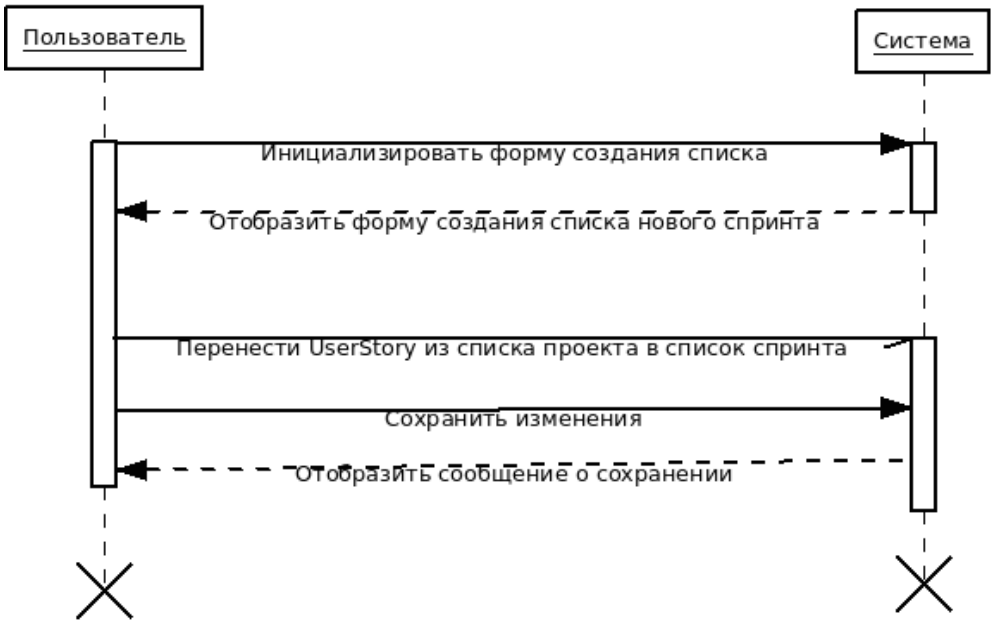

Рисунок 2.5 - Диаграмма взаимодействия функции "Создание списка пользовательских историй нового спринта"

4. Прерывание спринта. Данная диаграмма описывает логику взаимодействия между пользователем и системой во время внезапного прерывания спринта. Диаграмма приведена на рисунке 2.6.

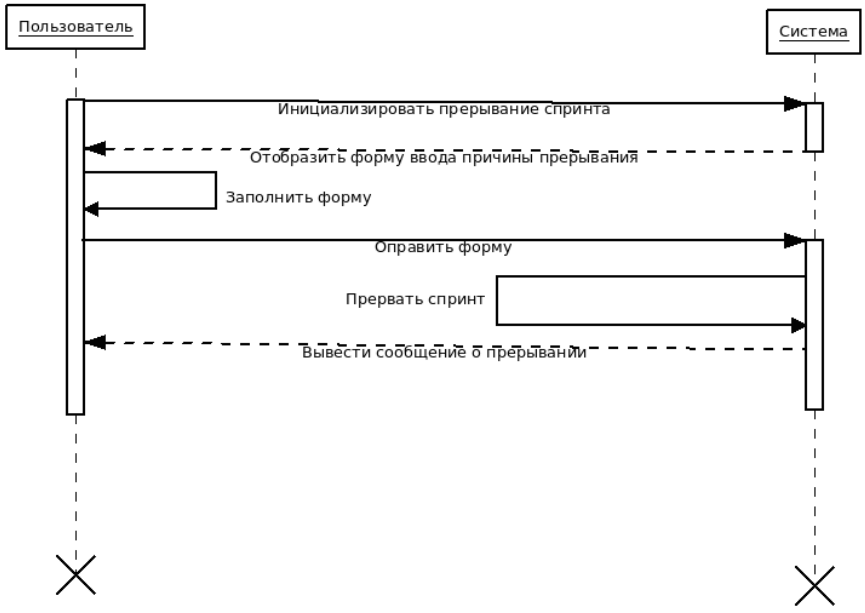

Рисунок 2.6 - диаграмма взаимодействия функции "прерывание спринта"

5. Формирование отчета о закрытых задачах. Данная диаграмма описывает логику взаимодействия между пользователем и системой во время формирования отчета о выполненных задачах. Диаграмма приведена на рисунке 2.7.

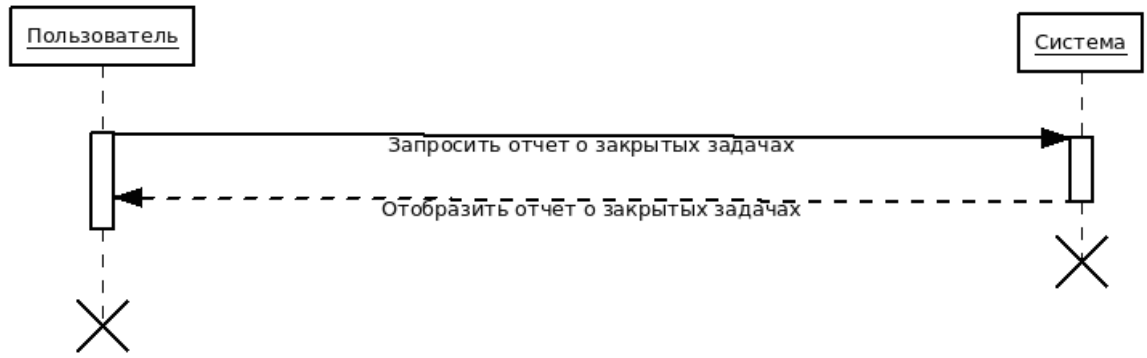

Рисунок 2.7 - диаграмма взаимодействия функции "формирование отчета о закрытых задачах"

6. Написание отчета о закрытой задаче. Данная диаграмма описывает логику взаимодействия между пользователем и системой во время написания отчета о закрытых задачах. Диаграмма приведена на рисунке 2.8.

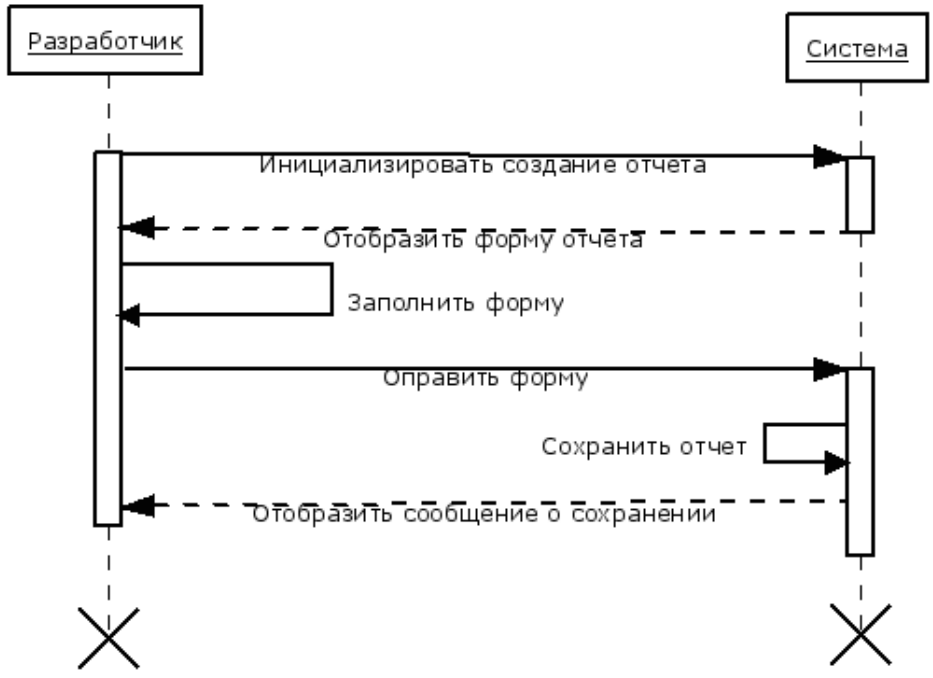

Рисунок 2.8 - диаграмма взаимодействия функции "Написание отчета о закрытой задаче"

7. Обмен сообщениями в командном чате. Данная диаграмма описывает логику взаимодействия между пользователем и системой во время обмена сообщения меджду пользователями в командном чате. Несмотря на то что в контексте данного процесса взаимодействие происходит между несколькими пользователями системы, для упрощения разработки данной функции лучше будет представить ее как взаимодействие между пользователем и системой, которая предоставляет ему доступ в реальном времени к базе данных сообщений, отправленных другими пользователями и интерфейс для отправки собственных сообщений. Диаграмма приведена на рисунке 2.9.

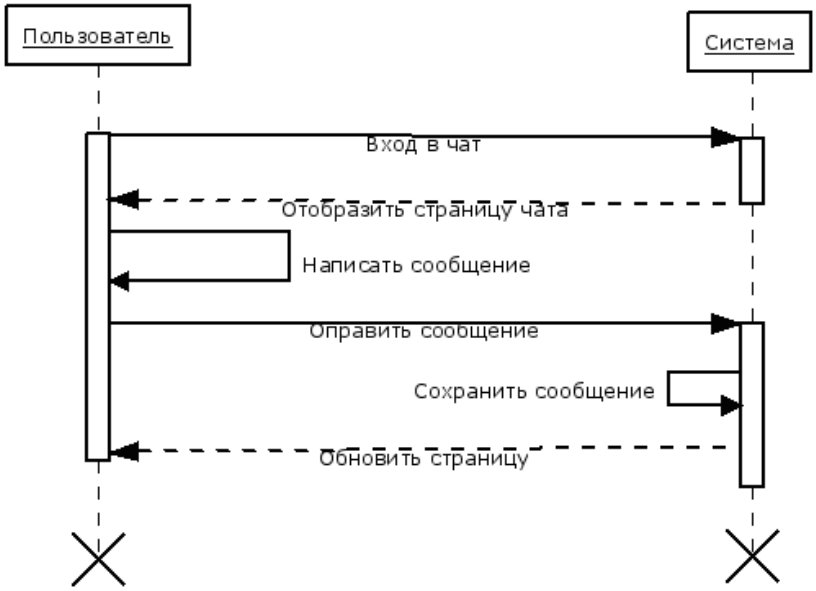

Рисунок 2.9 - диаграмма взаимодействий функции "обмен сообщениями в командном чате"

8. Формирование диаграммы сгорания задач. Данная диаграмма описывает логику взаимодействия между пользователем и системой во формирования диаграммы сгорания задач. Диаграмма приведена на рисунке 2.10.

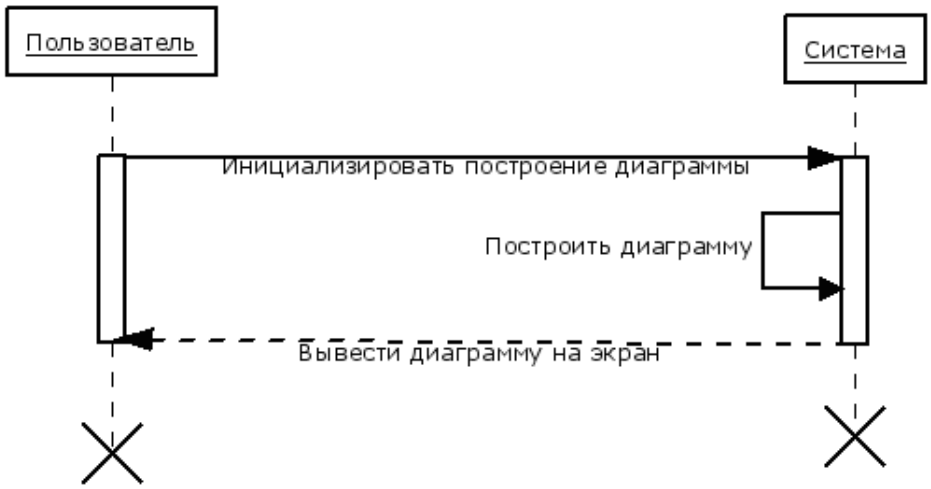

Рисунок 2.10 - диаграмма взаимодействий функции "формирование диаграммы сгорания задач"

9. Создание *Kanban* процесса. Данная диаграмма описывает логику взаимодействия между пользователем и системой во время определения стадий работа над задачами. Определение стадий процесса работы над реализацией функций разрабатываемого ПО является одним из важнейших аспектов разработки. Это позволяет отслеживать готовность той или иной задачи, и доставлять задачи к нужным специалистам. Диаграмма приведена на рисунке 2.11.

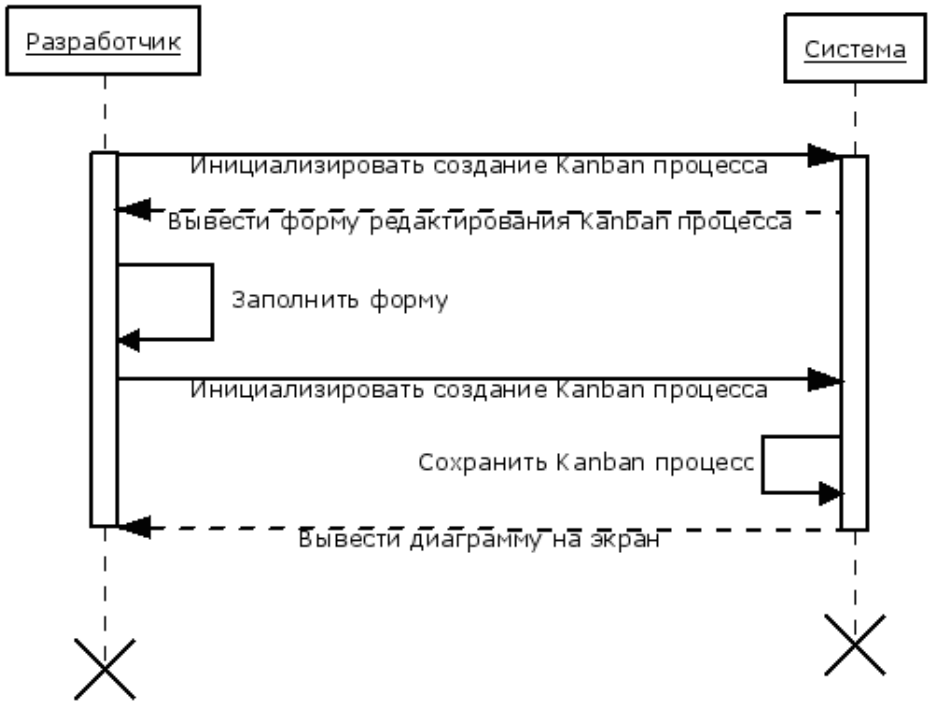

Рисунок 2.11 - диаграмма взаимодействия функции "создание kanban процесса"

10. Создание стадии *Kanban* процесса. Данная диаграмма описывает логику взаимодействия между пользователем и системой во время создания стадий работы над задачами проекта. Диаграмма приведена на рисунке 2.12.

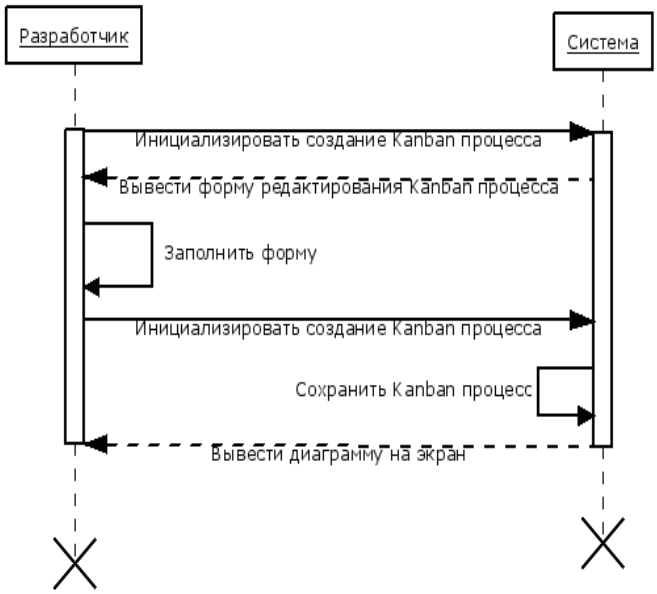

Рисунок 2.12 - диаграмма взаимодействия функции "создание стадии *Kanban* процесса"

11. Прикрепление пользовательской истории к стадии *Kanban* процесса. Данная диаграмма описывает логику взаимодействия между пользователем и системой во время прикрепления пользовательской истории к стадии *Kanban* процесса. Диаграмма приведена на рисунке 2.13.

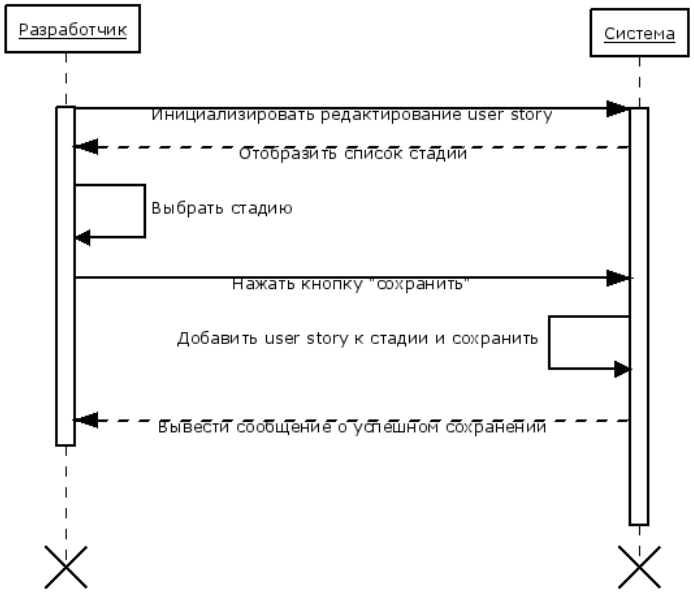

Рисунок 2.13 - диаграмма взаимодействия функции "прикрепление пользовательской истории к стадии *Kanban* процесса"

Приведенные выше диаграммы взаимодействия позволяют выделить сущности, характеризующие предметную область и смоделировать их взаимоотношения на диаграмме классов - диаграмме, демонстрирующей классы системы, их атрибуты, методы и взаимосвязи между ними.

2.3 Диаграмма классов.

Диаграмма классов представляет собой модель объектов и взаимоотношений между ними, составляющих проектируемую систему. Диаграмма приведена на рисунке 2.14.

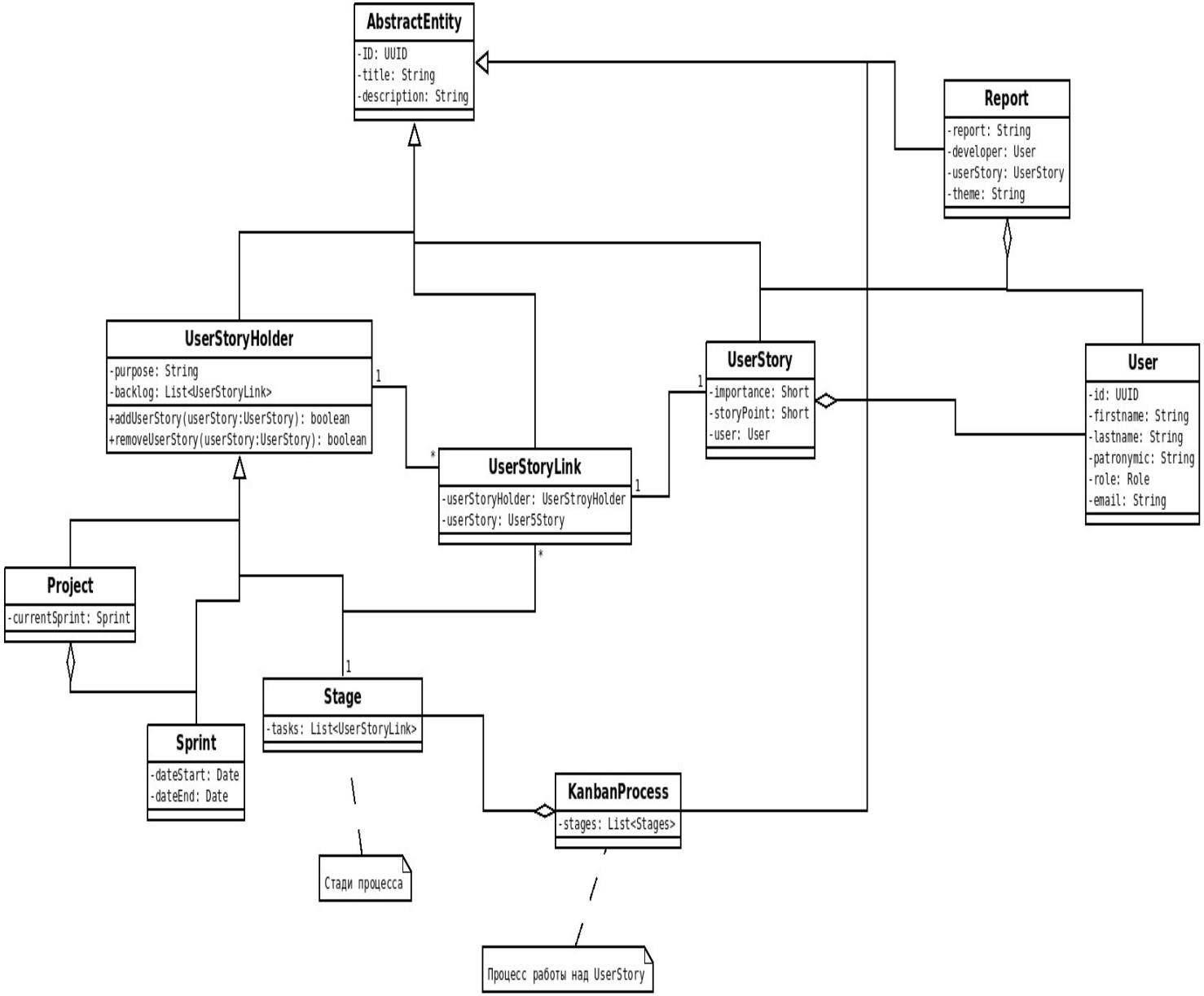

Рисунок 2.14 - диаграмма классов

Ниже приводится описание диаграммы классов, в которой указывается для каких целей нужна каждая из приведенных сущностей.

Класс *AbstractEntity***.**

Назначение: представляет собой абстрактную сущность, инкапсулирует общие для всех классов системы поля.

Класс *UserStory***.**

Наследует *AbstractEntity*.

Назначение: представляет собой сущность пользовательской истории, инкапсулирует ее поля. Является объектом предметной области.

Класс *UserStoryHolder***.**

Наследует *AbstractEntity*.

Назначение: представляет собой абстрактный класс-контейнер для *UserStory*. Инкапсулирует общую для всех контейнеров *UserStory* логику.

Класс *UserStoryLink***.**

Наследует *AbstractEntity*.

Назначение: связывает объекты классов *UserStory* и *UserStoryHolder*.

Класс *Report***.**

Наследует *AbstractEntity*.

Назначение: представляет собой класс отчета разработчика о выполнении требований UserStory.

Класс *User***.**

Назначение: класс описывает пользователя в контексте системы.

Класс *Project***.**

Наследует *UserStoryHolder*.

Назначение: класс описывает сущность проекта в контексте системы. Является объектом предметной области.

Класс *Sprint***.**

Наследует *UserStoryHolder*.

Назначение: класс описывает сущность спринта в контексте системы. Является объектом предметной области.

Класс *Stage***.**

Наследует *UserStoryHolder*.

Назначение: ккласс описывает стадию работы над *UserStory* (разработка/тестировани/приемка и др.) в контексте системы. Является объектом предметной области.

Класс *KanbanProcess***.**

Наследует: *AbstractEntity*.

Назначение: описывает процесс работы над *UserStory*, в контексте системы. Является контейнером для объектов класса *Stage*.

Далее идет технологический раздел, в котором приводится описание и обоснование выбора используемых в проекте технологий, описывается архитектура проектируемого приложения, и принятые в рамках разработки проектные решения

3. Реализация проектируемой системы.

В данном разделе рассматривается архитектура разрабатываемого программного продукта, а также обосновывается выбор технологий, используемых для реализации программных компонентов приложения.

3.1 Архитектура.

Невозможно переоценить, насколько важен выбор правильной архитектуры при проектировании программного приложения. Архитектура должна отвечать таким требованиям как простота, гибкость и расширямость. Иными словами приложения должно быть устроено достаточно просто, при этом необходимо, чтобы оно имело достаточно широкий диапазон настроек, а механизм расширения позволял наращивать функционал не затрагивая уже существующие модули. Системы.

Так как. подразумевается, что пользователь получает доступ к приложению через сеть Интернет необходимо использовать клиент-серверную архитектуру. Однако также нужно решить, на какие логические части необходимо разбить систему для тогго чтобы реализовать гибкость настройки и взаимозаменяемость компонентов. Для выполнения поставленной задачи было принято решение использовать шаблон модель-вид-контроллер. Данный шаблон разделяет приложение на три части, одна из которых — модель — инкапсулирует модель предметной области и сервисы для взаимодействия с базой данных. Вторая часть шаблона — вид — отвечает исключительно за представление данных пользователю, а также за отслеживание событий, таких как нажатие на кнопку и информирование о произошедшем событии контроллера. Контроллер в свою очередь инкапсулирует поведение системы, ее бизнес логику. Он является связующим звеном между видом и моделью. Контроллер обращается в модель для того чтобы получить данные, подготавливает их и формирует представление (вид), заполняя его данными, полученными из модели. В обратную сторону, контроллер

получает сигнал от представления, о том что произошло некоторое событие, производит необходимые действия, заданные разработчиком и записывает и передает данные в модель для записи их в базу данных. Схема работы приложения представлена на рисунке 3.1.

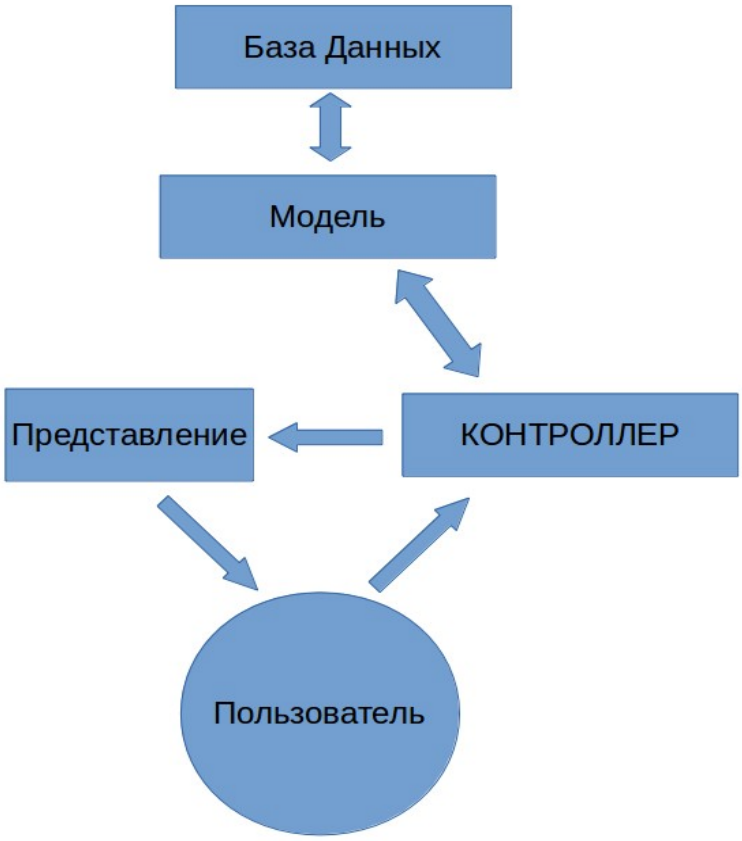

Рисунок 3.1 - архитектура программного продукта.

Если с контроллером и представлением все понятно, то модель стоит рассмотреть поближе. Модель представляет из себя цельный модуль, содержащий в себе несколько слоев абстракции.

Первый уровень — уровень предметной области. На данном уровне хранятся все сущности описываемой предметной области, а также связи между ними. Иными словами это уровень модели предметной области.

Далее идет инфраструктурный уровень — репозиторий, - который отвечает за взаимодействие с базой данных — за сохранение, извлечение, создание и удаление экземпляров сущностей из базы. Данный уровень собственно и является

промежуточным слоем абстракции между базой данных и контроллером. На рисунке 3.2 показана детализированная архитектура приложения.

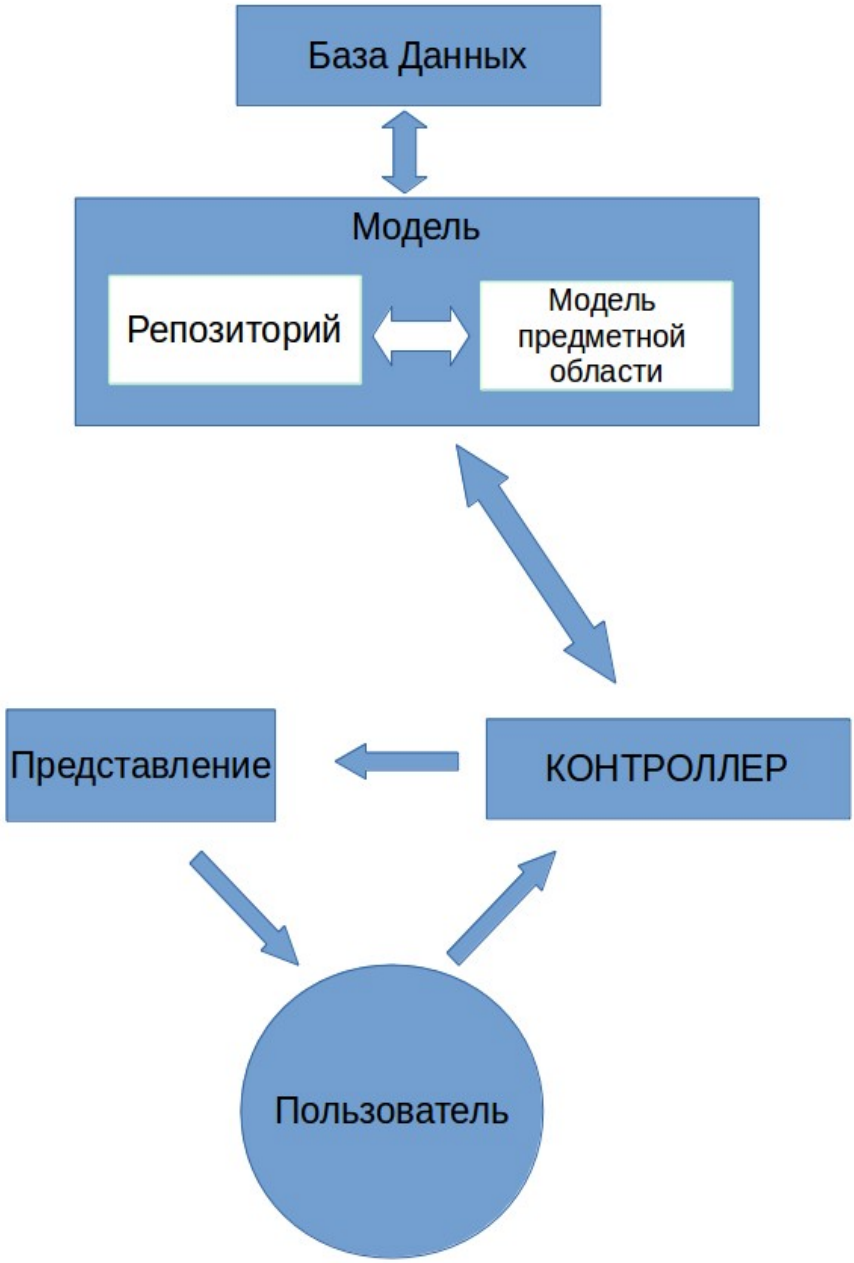

Рисунок 3.2 - детализированная архитектура.

После того, как спроектирована архитектура разрабатываемого приложения, необходимо выбрать технологии реализации приложения.

3.2 Технологии, используемые в проекте.

Многие языки программирования поддерживают описанную выше архитектуру приложений модель-вид-контроллер. Однако в данном проекте подразумевается разработка веб-приложения, поэтому круг средств разработки несколько снижается.

Рассмотрим три самых популярных на данный момент языка программирования для разработки веб-приложений; *Java*, *C#*, *php*.

Сравнение будет производится по следующим критериям:

1) Скорость работы и производительность;

2) Кроссплатформенность;

3) Поддерживаемые возможности;

Скорость работы и производительность.

С точки зрения конечного пользователя одно и тоже приложение на разных языках программирования на одном и том же оборудовании будет работать с одной и той скоростью. Однако при повышении нагрузки с 10 одновременно работающих пользователей хотя бы до 100 уже можно будет увидеть разницу в производительности. Так при большой нагрузке *php* перестает справляться с требуемыми функциями, замедление работы приложения уже видно невооруженным глазом. К сожалению, за свою простоту и нестрогую типизацию *php* платит низкой производительностью. В этом отношении компилируемые и строго типизированные *Java* и *C#* намного превосходят конкурента. Данные языки ориентированы на разработку корпоративных приложений, то есть приложений, которые априори будут подвергаться высокой нагрузке, поэтому их производительность на порядок превосходит *php*, который создавался для быстрой разработки простых веб сайтов. Результаты относительного сравнения по данной характеристике представлены в таблице 3.1. Сравнение производилось по десятибальной шкале.

Т а б л и ц а 3.1 - Сравнение языков программирования по производительности

	C#	*Java*	*php*
Производительность	10	10	5

Кроссплатформенность.

Технологии разработки веб-приложений развиваются с колоссальной скоростью и сейчас рассматриваемые языки программирования поддерживаются на всех популярных операционных системах. Однако есть некоторые различия. *Php* изначально распространяемый под лицензией *GPL* имеет реализации от производителя как для *Windows*, *Linux*, *Mac OS X*. *Java* в свою очередь также представляет собой открытый стандарт, но распространяется в двух реализациях — свободная от *open source* сообщества и официальная от компании-производителя *Oracle*. Обе реализации распространяются бесплатно.

C# работает на платформе *.NET* — альтернативе платформы *Java*, разработанной компанией *Microsoft*. Официальных реализаций платформы *.NET* для *Linux Microsoft* не выпускает, а среды разработки являются платными. Тем не менее существует свободная реализация платформы *.NET* и языка *C#* для *Linux*. Эта платформа называется Mono. На данный момент платформа разрабатывается компанией *Xamarin*. Поставляемая среда разработки — *Xamarin Studio* - реализована лишь для ОС *Windows* и *Mac OS X*.

Результаты относительного сравнения по данной характеристике представлены в таблице 3.2. Сравнение производилось по трехбальной шкале (учитывались три наиболее популярные операционные системы - *MS Windows, Linux, Mac OS X*).

Т а б л и ц а 3.2 - Сравнение языков программирования по кроссплатформенности

	C#	*Java*	*php*
Кроссплатформенность	2	3	3

Поддерживаемые возможности.

C# относительно молодой язык, поэтому для него написано еще не так много библиотек и каркасов приложений, как для *Java* или для *php*. *Java* поддерживает несколько технологий разработки пользовательского интерфейса для веб приложения — стандарты *JSP* (*JavaServer Pages*) — компилируемые на стороне сервера страницы, *JSF* (*JavaServer Faces*) — технология создания пользовательского интерфейса, реализующая компонентную модель, библиотека *Google Web Toolkit* и основанный на ней каркас приложений *Vaadin* позволяют создавать пользовательские интерфейсы веб-приложений исключтильно на языке *Java* — иными словами все веб приложение пишется на *Java*. C# на данный момент предоставляет только технологию *ActiveServer Pages*, сходную с *JSP*. *Php* в свою очередь позволяет смешивать *html* разметку с собственно *php*-кодом, получая таким образом весьма запутанные страницы, отвечающие за представление. Такой подход считается весьма ущербным, т. к. предполагает возможность реализации всей бизнес-логики приложения в компонентах, отвечающих за представление, что является неправильным и даже вредным, хотя и позволяет разработать приложение быстро. Также *php* предоставляет шаблонизатор *twig* для создания страниц представления. Данный шаблонизатор сходен по работе с *JSP* и предоставляет более идеологически верный способ представления данных.

Еще один недостаток *php* по сравнению с *C#* и *Java* — отсутствие возможности хранить состояние. В то время как *C#* и *Java* способны в течениии сессии хранить рабочую информацию в оперативной памяти, *php*, являясь по сути языком сценариев, способен работать только по модели запрос-ответ. То есть информация хранится только в базе данных, а в оперативную память она попадает только на время выполнения сценария. Конечно есть возможность сохранять данные в объектах сессии веб-браузера или в куки-файлах, однако данные способы весьма сомнительны с точки зрения безопасности. Таким образом невозможность хранить состояние является очень существенным недостатком в случае, когда нужно реализовывать многошаговые процедуры в веб-приложении. Результаты относительного сравнения по данной характеристике представлены в таблице 3.3.

Сравнение производилось по пятибальной шкале - за каждую предоставляемую возможность добавлялся один балл.

Т а б л и ц а 3.3 - Сравнение языков программирования по поддерживаемым возможностям

	C#	*Java*	*php*
Возможности	4	5	3

Из приведенного выше сравнения можно сделать вывод, что язык программирования *Java* является наилучшим выбором для реализации данного программного проекта, т.к. обладает широким набором поддерживаемых возможностей, работает на всех популярных платформах, средства разработки бесплатны, *Java* де факто является стандартом корпоративной разработки приложений.

В качестве сервера приложений выбирается *Apache Tomcat.* Выбор мотивирован тем, что данный сервер является самым легковесным, простым в использовании и надежным. Он бесплатен и прост в настройке. Также хорошо справляется с высокой нагрузкой и обладает встроенным балансировщиком.

В качестве системы управления базой данных выбирается СУБД *MySQL*. Данная СУБД является одной из самых популярных в силу производительности, стабильности работы и простоты использования. Все функции и возможности *MySQL* полностью задокументированы, по данной СУБД выпущено большое количество литературы, в том числе русскоязычной. *MySQL* поддерживает большое количество движков таблиц, в том чисел *MyISAM, InnoDB* и т.д. Репликация *MySQL* отлично протестирована и используется на таких сайтах как *Yahoo Finance, Mobile.de, Slashdot.* Создано большое количество программного обеспечения для управления и разработки баз данных с помощью *MySQL*. Это такие решения как *phpMyAdmin* и *MySQL Workbench*.

3.3 Проектирование модулей системы

В соответствии с принципом модульности необходимо разбить систему на несколько модулей, отвечающих за различные функции. Данные модули должны быть слабо связаны между собой. Следовательно наиболее рациональным способом разделения было бы деление по сущностям предметной области.

На диаграмме классов были выделены следующие сущности предметной области:

1) *AbstractEntity*
2) *UserStory*
3) *UserStoryHolder*
4) *UserStoryLink*
5) *User*
6) *Report*
7) *Project*
8) *Sprint*
9) *Stage*
10) *KanbanProcess*

Распределим эти сущности по логическим модулям системы. *AbstractEntity* является базовым абстрактным классом и определяет сущность вообще, поэтому ее стоит хранить в отдельном модуле, чтобы не было циклических зависимостей. Данный модуль будет называться модулем модели.

Далее, рассмотрим классы *UserStory*, *UserStoryLink*, *UserStoryHolder*. *UserStory* — это класс, определяющий пользовательскую историю, UserStoryHolder — абстрактный класс-контейнер, наследуя который можно создать некоторую сущность, способную ссылаться, то есть хранить в себе, список пользовательских историй. Для того чтобы данный класс-контейнер имел возможность ссылать на пользовательские истории он имеет в качестве закрытого поля список обхектов класса UserStoryLink, реализуя таким образом отношение «один ко многим».

В свою очередь класс UserStoryLink является ссылкой на определенный

объект класса UserStory и может выступать в качестве связи между объектов класса UserStory и контейнером. Таким образом эти три сущности выделяются в отдбьелный модуль пользовательских историй.

Сущность User является описанием пользователя в системе. С этой сущностью необходимо производить такие операции, как аутентификация, авторизация, проверка прав, не выполнение того или иного действия или на переход на тот или иной адрес внутри системы. Иными словаами необходимо выполнять процедуры связанные с безопасностью. Следовательно сущност выделяется в модуль безопасности.

Сущность Report представляет собой отчет о выполненной задаче. Также он может представлять собой отчет по нескольким выполненным задачам, или по всему спринту или проекту. Таким образом стоит пометить данную сущность в отдельный модуль отчетов.

Сущности Project и Sprint являются наследниками класса UserStoryHolder и представляют собой описание проекта и спринта внутри системы. Эти сущности хранят в себе списки пользовательских историй а также установленнеые на решение тех или иных задач сроки. Данные сущности выделяются в отдельный модуль проектов.

Сущности Stage и KanbanProcess являются представлениями стадии разработки функциональности и общим процессом работы над функциональностью соответственно. Сущность Stage является наследником класс UserStoryHolder, так как она должна содержать в себе список пользовательских историй. Каждая стадия указывает, что сейчас происходит с конкретной задачей проекта, разрабаотывается ли функция, затребованная в задаче, проходит ли она тестирование или же готова к использованию. Список стадий неограничен и не фиксирован. Каждая группа разработки может создавать собственный процесс разработки программного обеспечения и, соответственно, создавать свои стадии.

KanbanProcess в свою очередь, является контейнером для объектов класса Stage. Он хранит набор определенных стадий и контролирует перемещение задач

между этими стадиями. В системе может быть определено несколько процессов. Тем не менее для одного проекта может существовать только один процесс. Таким образом сущности Stage и KanbanProcess выделяются в отдельный модуль процессов.

Также необходимо выделить несколько служебных модулей, с помощью которых будут производиться операции и процедуры над сущностями. В частности необходим модуль, который будет содержать все *html*-страницы представления — модуль представления, а также модуль, отвечающий за обработку запросов к системе и переадресацию пользователя на соответствующую страницу — модуль переадресации. Так как эти две задачи довольно тесно связаны между собой было принято решение объеденить их в один общий модуль переадресации и представлений. Также, необходим модуль, содержащий настройки подключения к базе данных и объектно-реляционного сопоставления. Структура системы с учетом включенных модулей представлена на рисунке 3.3.

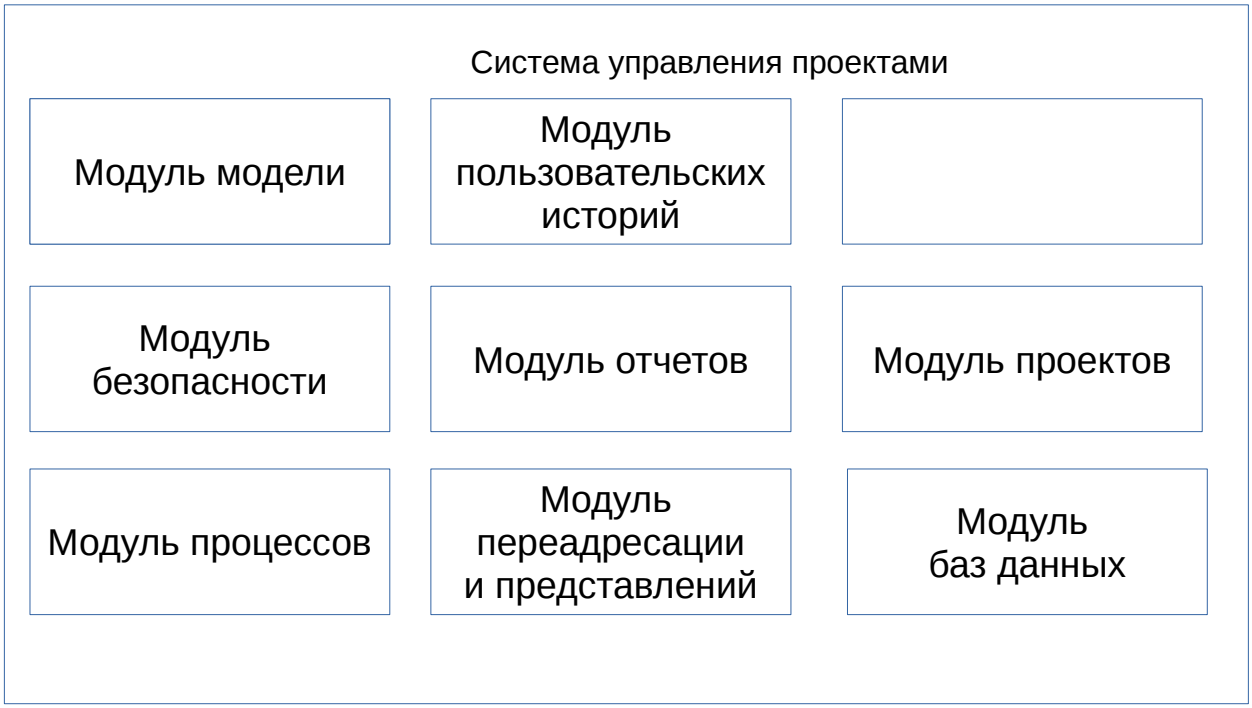

Рисунок 3.3 — Структура системы с учетом необходимых модулей

Общая структура системы спроектирована, теперь нужно спроектировать, внутреннюю архитектуру каждого модуля.

Все модули данной системы, за исключением модуля переадресации и представления и модуля базы данных инкапсулируют в себе некоторую часть предметной области. Данные модули строятся вокруг сущностей предметной области должны включать в себя сервисы и классы, необходимые для выполнения операций над своими сущностями. Оптимальной архитектурой модуля здесь выбрана многослойная архитектура, описанная Эриком Эвансом в [10]. Нижним слоем является слой предметной области. Он включает в себя сущности предметной области, с которой работает данный модуль. Далее идет слой взаимодействия с базой данных посредством библиотеки объектно-реляционного сопоставления. Этот слой должен иметь интерфейс и реализацию. Над слоем базы данных строится слой сервисов, которые собственно отвечает за извлечение данных из базы посредством интерфейса предыдущего слоя, подготовку выборки данных для визуализации (фильтрация, сортировка и т. п.) и передачу данных на уровень выше — в слой контроллера. Контроллер в свою очередь отвечает за визуализацию представлений, передачу данных из слоя сервисов в представления, обработку событий, произошедших в представлении и передачу данных из представлений в слой ниже — на уровень сервисов.

Такая архитектура позволяет модулю быть гибким и легко настраиваемым. Взаимодействие слоев через интерфейсы снижает зависимости между уровнями — каждый нижний уровень не знает о существовании верхнего, а каждый верхний уровень не зависит от реализации операции в нижележащем уровне. Таким образом, если меняется алгоритм работы одной операции, то он меняется только внутри конкретного уровня, не затрагивая остальные. Схема архитектуры модулей показана на рисунке 3.4.

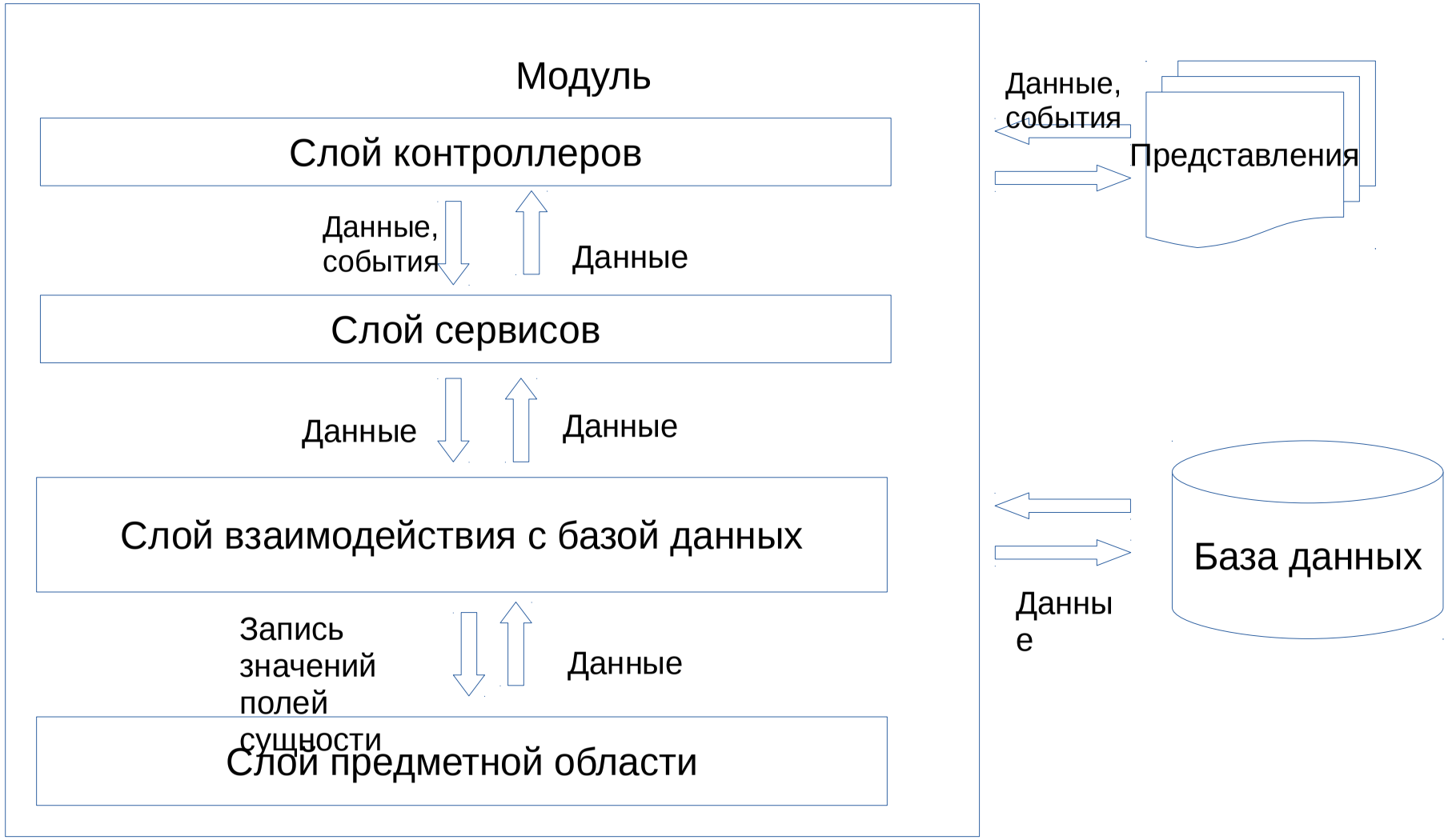

Рисунок 3.4 — Архитектура модуля

В системе выделены все необходимые для работы модули. Спроектирована архитектура самой системы, а также общая архитектура модулей. Исходный код приложения можно увидеть в приложении 1.

Заключение

В данном дипломном проекте были рассмотрены факторы, влияющие на успешность управления разработкой программного обеспечения, были рассмотрены возможности увеличения эффективности управления разработкой программного обеспечения, в связи с чем получили освещение различные методологии управления разработкой программных продуктов. На основе проведенного анализа автор пришел к выводу, что оптимальной методологией является *Scrum*, относящаяся к семейству гибких методологий разработки.

Помимо того, что в процессе анализа проблемы неэффективности управления разработкой программного обеспечения была определена оптимальная методология, также была выявлена необходимость разработки программного продукта, позволяющего производить управление разработкой в рамках указанной методологии.

Далее был разработан проект программного продукта, включающий в себя *UML* схемы и сценарии взаимодействия пользователя с системой. Были выбраны и обоснованы технологии, используемые для реализации программного продукта.

Список использованной литературы

1. Кент Бек. Экстремальное программирование: разработка через тестирование. БИБЛИОТЕКА ПРОГРАММИСТА, 2003

2. Заренков В.А. Управление проектами. Изд-во АСВ, СПб. 2006

3. Джефф Сазерленд, Кен Швабер. Скрам Гид. http://www.scrumguides.org/docs/scrumguide/v1/Scrum-Guide-RUS.pdf;

4. Стелтинг Стивен, Маасен Олав. Применение шаблонов *Java*. Издательский дом "Вильямс". 2002.

5. Хенрик Книберг, Маттиас Скаррин. *Scrum* и *Kanban*. Выжимаем максимум.: *C4Media Inc.* 2010

6. Хорстман Кей, Корнелл Гарри. "*Java* 2. Библиотека профессионала, том 1 - Основы". Издательский дом "Вильямс", 2007

7. *Katy Sierra, Bert Bates. Head First Java. O'Raily, 2005*

8. *PMBOK Guide, Project Management Institute Inc*. http://www.cs.bilkent.edu.tr/~cagatay/cs413/PMBOK.pdf

9. *The Standish Group*: "*Chaos Report"*, 1995;

10. Эванс Эрик. "Предметно ориентированное проектирование. Структуризация сложных программных систем". Издательский дом "Вильямс". Москва, 2011

11. Буч, Рамбо, Джекобсон. *UML*. Руководство пользователя, Издательский дом "Вильямс". Москва. 2001

12. Брукс Фредерик. "Мифический человеко-месяц". Издательский дом "Вильямс". Москва. 2001

13. Макконнелл Стив. "Совершенный код". Издательский дом "Вильямс". Москва. 2005

14. Энди Хант, Дэвид Томас. "Программист-прагматик. Путь от подмастерья к мастеру". Издательский дом "Вильямс". Москва 2004

15. Джералд Джей Сассман. "Структура и интерпретация компьютерных программ". Издательский дом "Вильямс". Москва 2006

16. Мэттью Флэтт, Роберт Брюс Финдлер, Шрайрам Кришнамерти. "Как

проектировать программы". Издательский дом "Вильямс". Москва. 2001

17. Джон Влиссайдс, Ральф Джонсон, Ричард Халм, Эрих Гамма. "Шаблоны проектирования". Издательский дом "Вильямс". Москва. 2004.

18. Грегор Хохпи, Бобби Вульф. "Шаблоны интеграции корпоративных приложений". Издательский дом "Вильямс". Москва. 2001.

19. Гвидо Шмутц, Даниэль Лейбхарт, Питер Велькебанч. "Сервисно-ориентированная архитектура". Издательский дом "Вильямс". Москва. 2005.

20. Майк Кон. "*Scrum*: гибкая разработка ПО". Издательский дом "Вильямс". Москва. 2003.

Приложение 1

Класс *AbstractEntity*

```
package ru.imilienko.dipmon.userstory;

import org.hibernate.annotations.Type;
import org.hibernate.validator.constraints.NotBlank;
import javax.persistence.*;
import javax.validation.constraints.Size;
import javax.xml.bind.annotation.XmlRootElement;
import java.util.UUID;

@Entity
@XmlRootElement
public class AbstractEntity {

    @Id
   @Column(name = "id")
   private UUID id;

   @Column(name = "title")
   private String title;


   @Column(name = "description")
   private String description;

/*...*/
}
```

Класс *UserStory.*

```
package ru.imilienko.dipmon.userstory;

import org.hibernate.annotations.Type;
import org.hibernate.validator.constraints.NotBlank;
import javax.persistence.*;
import javax.validation.constraints.Size;
import javax.xml.bind.annotation.XmlRootElement;
import java.util.UUID;

@Entity
@Cacheable
@Table(name = "userStory", schema = "userStory")
@XmlRootElement
@CopyDescriptor(filter = DefaultCopyFilters.class)
public class UserStory extends AbstractEntity {

    @Column(name = "importance")
    private Integer importance;

  @Column(name = "user")
    private User importance;

    @Column(name = "storypoints")
    private Integer storyPoints;

    @Column(name = "howToDemo")
    private String howToDemo;

    /*...*/
```

```
}
```

Класс *UserStoryLink*

```
package ru.imilienko.dipmon.userstory;

import org.hibernate.annotations.Type;
import org.hibernate.validator.constraints.NotBlank;
import javax.persistence.*;
import javax.validation.constraints.Size;
import javax.xml.bind.annotation.XmlRootElement;
import java.util.UUID;

@Entity
@Table(name = "userStoryLink", schema = "userStoryLink")
@XmlRootElement
@CopyDescriptor(filter = DefaultCopyFilters.class)
public class UserStoryLink extends AbstractEntity {

    @Column(name = "userStory")
    private UserStory userStory;

        @Column(name = "userStoryHolder")
    private UserStoryHolder userStoryHolder;

    /*...*/
}
```

Класс *UserStoryHolder*

```
package ru.imilienko.dipmon.userstory;
import org.hibernate.annotations.Type;
import org.hibernate.validator.constraints.NotBlank;
import javax.persistence.*;
import javax.validation.constraints.Size;
import javax.xml.bind.annotation.XmlRootElement;
import java.util.UUID;
@Entity
@Cacheable
@Table(name = "UserStoryHolder", schema = "UserStoryHolder")
@XmlRootElement
@CopyDescriptor(filter = DefaultCopyFilters.class)
public class UserStoryHolder extends AbstractEntity {

    @Column(name = "image")
    private String purpose;

    @Column(name = "backlog")
    private List<UserStoryLink> backlog;

        public addUserStory(UserStory userStory){
                /*...*/
        }

        public removeUserStory(UserStory userStory){
                /*...*/
        }
}
```

Класс *Report*

```
package ru.imilienko.dipmon.userstory;

import org.hibernate.annotations.Type;
import org.hibernate.validator.constraints.NotBlank;
import javax.persistence.*;
import javax.validation.constraints.Size;
import javax.xml.bind.annotation.XmlRootElement;
import java.util.UUID;

@Entity
@Cacheable
@Table(name = "report", schema = "report")
@XmlRootElement
@CopyDescriptor(filter = DefaultCopyFilters.class)
public class Report extends AbstractEntity {

    @Column(name = "report")
    private String report;

    @Column(name = "developer")
    private User developer;

     @Column(name = "userStory")
    private UserStory userStory;
    @Column(name = "theme")
    private String theme;

        /*...*/
}
```

Класс *User*

```
package ru.imilienko.dipmon.userstory;

import org.hibernate.annotations.Type;
import org.hibernate.validator.constraints.NotBlank;
import javax.persistence.*;
import javax.validation.constraints.Size;
import javax.xml.bind.annotation.XmlRootElement;
import java.util.UUID;

@Entity
@Cacheable
@Table(name = "report", schema = "report")
@XmlRootElement
@CopyDescriptor(filter = DefaultCopyFilters.class)
public class User extends AbstractEntity {

    @Column(name = "firstname")
    private String firstname;

    @Column(name = "lastname")
    private String lastname;

    @Column(name = "patronymic")
    private String patronymic;

    @Column(name = "email")
    private String email;

    @Column(name = "role")
```

```
    private Role role;

        /*...*/
}
```

Класс *Project*

```
package ru.imilienko.dipmon.userstory;

import org.hibernate.annotations.Type;
import org.hibernate.validator.constraints.NotBlank;
import javax.persistence.*;
import javax.validation.constraints.Size;
import javax.xml.bind.annotation.XmlRootElement;
import java.util.UUID;

@Entity
@Cacheable
@Table(name = "project", schema = "project")
@XmlRootElement
@CopyDescriptor(filter = DefaultCopyFilters.class)
public class Project extends UserStoryHolder {

    @Column(name = "firstname")
    private Sprint currentSprint;


        /*...*/
}
```

Класс *Sprint*

```
package ru.imilienko.dipmon.userstory;

import org.hibernate.annotations.Type;
import org.hibernate.validator.constraints.NotBlank;
import javax.persistence.*;
import javax.validation.constraints.Size;
import javax.xml.bind.annotation.XmlRootElement;
import java.util.UUID;

@Entity
@Cacheable
@Table(name = "sprint", schema = "sprint")
@XmlRootElement
@CopyDescriptor(filter = DefaultCopyFilters.class)
public class Sprint extends UserStoryHolder {

    @Column(name = "dateStart")
    private Date dateStart;

    @Column(name = "dateEnd")
    private Date dateEnd;

        /*...*/
}
```

Класс *Stage*

```
package ru.imilienko.dipmon.userstory;
import org.hibernate.annotations.Type;
import org.hibernate.validator.constraints.NotBlank;
```

```
import javax.persistence.*;
import javax.validation.constraints.Size;
import javax.xml.bind.annotation.XmlRootElement;
import java.util.UUID;

@Entity
@Cacheable
@Table(name = "stage", schema = "stage")
@XmlRootElement
@CopyDescriptor(filter = DefaultCopyFilters.class)
public class Stage extends UserStoryHolder {

    @Column(name = "tasks")
    private List<UserStoryLink> tasks;



        /*...*/
}
```

Класс *KanbanProcess*

```
package ru.imilienko.dipmon.userstory;

import org.hibernate.annotations.Type;
import org.hibernate.validator.constraints.NotBlank;
import javax.persistence.*;
import javax.validation.constraints.Size;
import javax.xml.bind.annotation.XmlRootElement;
import java.util.UUID;

@Entity
```

```
@Cacheable
@Table(name = "kanbanprocess", schema = "kanbanprocess")
@XmlRootElement
@CopyDescriptor(filter = DefaultCopyFilters.class)
public class KanbanProcess extends AbstractEntity {

    @Column(name = "stage")
    private List<Stage> stage;


        /*...*/
}
```