Содержание

[Перечень условных обозначений 7](#_Toc357720774)

[Введение 8](#_Toc357720775)

[1. Исследование предметной области 9](#_Toc357720776)

[1.1 Актуальность проблемы 9](#_Toc357720777)

[1.2 Обзор существующих методологий 11](#_Toc357720778)

[1.3 Оценка задач проекта 15](#_Toc357720779)

[2. Обзор технологий 17](#_Toc357720780)

[2.1 Платформа .NET 17](#_Toc357720781)

[2.2 ADO.net 22](#_Toc357720782)

[2.3 MVC.net 24](#_Toc357720783)

[3. Архитектура приложения и особенности реализации 28](#_Toc357720784)

[3.1 Описание системы 28](#_Toc357720785)

[3.1.1 Состав системы 28](#_Toc357720786)

[3.1.2 Функциональное назначение элементов системы 29](#_Toc357720787)

[3.1.3 Основные функции элементов системы 33](#_Toc357720788)

[3.2 Особенности реализации 37](#_Toc357720789)

[3.2.1 Импорт и экспорт данных 37](#_Toc357720790)

[3.2.2 Система интерфейсов ядра 38](#_Toc357720791)

[3.2.3 Основные методы ядра 43](#_Toc357720792)

[3.2.4 Взаимодействие ядра с базой данных 47](#_Toc357720793)

[4. Обеспечение безопасных условий труда на предприятии «Скэнд». 49](#_Toc357720794)

[5. Технико-экономическое обоснование инвестиционного проекта по разработке программного обеспечения “Система учета метрик и распределения ресурсов при Agile разработке” 55](#_Toc357720795)

[5.1 Расчет затрат при создании ПО. 55](#_Toc357720796)

[5.2 Расчет экономической эффективности от разработки ПО 65](#_Toc357720797)

[6. Заключение 71](#_Toc357720798)

[7. Список источников 72](#_Toc357720799)

[Приложение А: реализация соединения с базой данных 73](#_Toc357720800)

[Приложение Б: подключение ядра приложения 77](#_Toc357720801)

Перечень условных обозначений

ПО – программное обеспечение.

ПП – программный продукт.

ПС – программное средство.

БД – база данных.

СУБД – система управления базами данных.

UML - Unified Modeling Language – язык графического описания для объектного моделирования в области разработки программного обеспечения.

XML – eXtensible Markup Language (расширяемый язык разметки).

SQL – Structured Query Language (структурированный язык запросов).

Loop – система учета метрик и распределения ресурсов (название проекта)

Введение

Когда речь идет о разработке программного обеспечения, все менеджеры рано или поздно сталкиваются с пресловутым «железным треугольником».



В начале разработки всегда даются какие-то начальные оценки временных и материальных затрат на разработку проекта, однако мало кто умудряется в них вписываться. И в итоге всегда приходится чем-то жертвовать: будь то деньги за работу, дополнительное время или функционал проекта. Почему? Неужели даже менеджеры зря занимают свою должность и патологически не способны выполнить свою работу хорошо? Вовсе нет. Дело в том, что оценки работ зачастую меряются в идеальных часах. То есть в единицах, которые являлись бы корректными, если бы всякий разработчик действительно мог работать, не отрываясь от компьютера, 8 часов в день, если бы он не болел или у него не было личных проблем. Дело не только в разработчиках, но всех людях, участвующих в процессе разработки: спецификация составлена некорректно, требования изменились, тестировщик пропустил большую ошибку на раннем этапе разработки и из-за нее нужно перекраивать большую часть архитектуры проекта. Все эти и многие другие факторы представляют собой риски при разработке ПО, и с ними нельзя не считаться.

Многие менеджеры также считают, что могут выставить оценки для проекта на глаз, просто прочитав спецификацию. Такого рода оценки берутся с потолка и, даже если оказываются верными, то случайно.

Целью дипломного проекта является, разработка для системы расчета рисков (под названием Loop), возникающих при разработке программного продукта, и реализация основных требований к ней.

1. Исследование предметной области
   1. Актуальность проблемы

Проблема выставления актуальных оценок для программного продукта является ключевой при разработке любого ПО. Не зависимо от того, выполняется проект по предоплате или создатели будут пытаться продать его по окончании разработки, вписаться с некий план является целью любого менеджера.

В последнее время очень популярна «гибкая разработка». Многие оффшорные предприниматели отдают предпочтение компаниями, работающим по методологиям Agile (“гибкая разработка”). Почему? Во-первых, разделение рисков. Сегодня самые распространенные виды взаимодействия с заказчиком — это fixed-price контракты или контракты time & material. В первом случае все риски по проекту ложатся на подрядчиков со всеми вытекающими из этого последствиями. Согласно второй схеме заказчик оплачивает фактическую работу, что мало работает на конечный результат.

Во-вторых, предсказуемость. «Гибкая» методология разработки подразумевает отказ от долгосрочного планирования, а также отсутствие цифр по срокам и стоимости продукта целиком. Agile предлагает воспринимать команду как некий черный ящик, имеющий определенный объем входных данных и отведенное количество времени, для предоставления результата. В начале итерации команда оценивает задачу и заранее обязуется предоставить результат.

В-третьих, обратная связь. Потерять свою актуальность может даже очень точно документированный проект после полугода разработки. Желание же перестроить исходную конструкцию может привести к длительной переработке продукта. Кроме того, постоянная динамика бизнеса не предоставляет лишнего времени на детальное документирование по причине различности первоисточника анонса, а также изменчивости запросов пользователей.

В-четвертых, быстрая и легкая реакция на изменения. Предложенная система подразумевает, что уже после первой итерации функциональность будет не полностью завершенной, и заказчик сможет вносить комментарии и поправки практически с самого старта проекта. Спустя две итерации для получения обратной связи с пользователями уже можно будет запускать бета версию продукта.

И, наконец, самоорганизация. «Гибкая» методология разработки позволяет уйти от излишней структуры менеджмента, потому что существует только представитель заказчика, представляющий интересы бизнеса. Кроме того, нет необходимости в проверке каждого участника: команда сама распределит задачи и внутреннюю ответственность внутри, а также определенно будет гарантировать производительность труда и качество продукта. Самоорганизация здесь выступает в качестве мотивации команды.

Все вышесказанное делает Agile популярным как с точки зрения бизнеса, так и с точки зрения разработки. Поэтому актуальным вопросом является обеспечение удобного способа расчета всех рисков и предоставления отчетности заказчику. Система должна быть достаточно простой и точной, чтобы с ней мог справиться и новичок в сфере менеджмента.

* 1. Обзор существующих методологий

Основное отличие методологий разработки заключается в принципах, которых следует придерживаться при производстве программного продукта. По количеству этих самых принципов, пожалуй, XP можно отнести к самой строгой из методологий разработки, потому как она содержит аж 12 основных приемов.

1. Короткий цикл обратной связи (Fine scale feedback)
   1. Разработка через тестирование (Test driven development)
   2. Игра в планирование (Planning game)
   3. Заказчик всегда рядом (Whole team, Onsite customer)
   4. Парное программирование (Pair programming)
2. Непрерывный, а не пакетный процесс
   1. Непрерывная интеграция (Continuous Integration)
   2. Рефакторинг (Design Improvement, Refactor)
   3. Частые небольшие релизы (Small Releases)
3. Понимание, разделяемое всеми
   1. Простота (Simple design)
   2. Метафора системы (System metaphor)
   3. Коллективное владение кодом (Collective code ownership) или выбранными шаблонами проектирования (Collective patterns ownership)
   4. Стандарт кодирования (Coding standard or Coding conventions)
4. Социальная защищенность программиста (Programmer welfare):
   1. 40-часовая рабочая неделя (Sustainable pace, Forty hour week)

Что все это значит? Сами по себе разделы не несут никакой конкретной информации, поэтому рассмотрю сразу подпункты.

*Разработка через тестирование.* Принципиально важным моментом при гибкой разработке является покрытие кода автоматическими тестами. Тесты помогают удостовериться, что новая функциональность надежна, а старый функционал остался неизменным. Конечно, тут нельзя не учитывать и качество тестов. Тесты должны покрывать все строчки, написанного кода, а так же покрывать все возможные сценарии использования кода (use-case). Этот пункт так же утверждает, что написание тестов предшествует написанию самого функционала. Причина проста: если в спецификации однозначно указана сигнатура необходимых методов, то написанные заранее тесты помогут проверить, что новая функция отвечает всем прописанным в спецификации требованиям. Программист зачастую автоматически избегает при тестировании use-case(ы), которые не будут поддерживаться его кодом. Написанные заранее, тесты предотвратят подобных упущений. Если же сигнатура не указана, написание тестов поможет разработать набор необходимых методов для нового функционала и их интеграцию, прочувствовать требования и вникнуть в тему до написания самих функций.

*Игра в планирование.* Существует масса методик, применяемых при гибкой разработке, для оценивания, распределения и планирования задач. Подробнее их опишу в следующем разделе.

*Заказчик всегда рядом.* В проектах, где гибкая разработка не используется, требования по функционалу программного продукта очень тщательно оговариваются заранее. Продукты, где точный функционал заранее определить невозможно, к примеру, из-за скорости изменения рынка или технологий, зачастую разрабатываются при помощи гибкой разработки. Но тут важно, чтобы заказчик был легко достижим, и скорость выяснения неточных вопросов была наиболее быстрой. Простой стоит времени, а значит и денег. Заказчик должен быть членом XP-команды (on-site customer). Он пишет пользовательские истории, выбирает истории, которые будут реализованы в конкретной итерации, и отвечает на вопросы, касающиеся бизнеса. Заказчик должен быть экспертом в автоматизируемой предметной области. Необходимо постоянное наличие обратной связи с заказчиком (feed-back).

*Парное программирование.* Очень редко разработка все время ведется в слаженной, дружной команде. Чаще постоянно идет какое-то движение кадров. Новые люди, мало знающие о проекте, могут какое-то время (к примеру, 2-3 дня в неделю по 3 часа) заниматься разработкой программы с людьми, которые уже хорошо знают продукт. Это помогает улучшить отношения в коллективе, быстро обучить новичков с минимальных отрывом от работы старших разработчиков. Этот прием также удобен, если решение какой-то задачи одному разработчику дается нелегко, а другой хорошо разбирается в сути задачи и знает ее решение.

*Непрерывная интеграция.* Разработку программного продукта ведут обычно сразу несколько разработчиков, которые постоянно коммитят в репозиторий какие-то новые разработки. Важно при этом контролировать, что новый код не сломал старый функционал, поэтому хорошей практикой считается построение билдов и прохождение автоматизированных тестов по расписанию. Это помогает быстро найти виновника нежелательных изменений и поправить ошибки. Основное правило интеграции следующее: интеграцию можно производить, если все тесты проходят успешно. Если тесты не проходят, то программист должен либо внести исправления и тогда интегрировать составные части системы, либо вообще не интегрировать их. Правило это жесткое и однозначное — если в созданной части системы имеется хотя бы одна ошибка, то интеграцию производить нельзя.

*Рефакторинг.* К сожалению, даже если собирается команда опытных программистов, со временем их общий код превращается в кашу и его следует приводить в порядок. Для этого и выполняется рефакторинг: он помогает структурировать код во избежание дальнейшей путаницы и дублирования функционала.

*Частые небольшие релизы.* Основной принцип любого направления гибкой разработки. Суть его в готовности выпустить проект в любое время: итеративная разработка.

*Простота.* Любое свойство системы должно быть реализовано как можно проще. Программисты в XP-команде работают под девизом: «Ничего лишнего!». Принимается первое наипростейшее работающее решение, реализуется необходимый уровень функциональности на данный момент. Тем самым экономится время программиста.

*Метафора системы*. Хорошая метафора системы означает простоту именования классов и переменных. В реальной жизни поиск метафоры — крайне сложное занятие; найти хорошую метафору непросто. В любом случае команда должна иметь единые правила именования.

*Коллективное владение кодом.* Каждый программист в коллективе должен иметь доступ к коду любой части системы и вносить изменения в любой код. Обязательное правило: если программист внес изменения и система после этого работает некорректно, то именно этот программист должен исправить ошибки. В противном случае работа системы уподобится тотальному хаосу.

*Стандарт кодирования.* Стандарты кодирования нужны для обеспечения других практик: коллективного владения кодом, парного программирования и рефакторинга. Без единого стандарта выполнять эти практики как минимум сложнее, а в реальности вообще невозможно: группа будет работать в режиме постоянной нехватки времени. Детальные стандарты не требуются, необходимо стандартизировать только важные вещи.

*40-часовая неделя.* Объем сверхурочных работ не может превышать по длительности одну рабочую неделю. Даже отдельные случаи сверхурочных работ, повторяющиеся слишком часто, служат признаком серьезных проблем, которые требуют безотлагательного решения. Как показывает практика применения экстремального программирования (несмотря на целый ряд положительных примеров, приводимых сторонниками данного метода), сверхурочная работа при таком подходе — это правило, а не исключение, и борьба с проблемами в данном случае — явление постоянное. Усиливается она в период замены текущей сырой версии продукта очередной — менее сырой. Если заказчик не получает постоянных доказательств улучшения системы, значит, возникли серьезные проблемы.

Хоть все вышеуказанные приемы и относятся к экстремальному программированию, Scrum и Kanban являются формами этой базовой методологии. Всякий менеджер может выбрать, какой набор практик он хочет использовать на своем проекте, что ему подходит и в чем реально есть необходимость. Однако, конечно, и scrum и kanban имеют свои особенности, которые нельзя не учитывать.

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. То есть этот набор принципов однозначно делает временное ограничение на выполнение итерации.

Kanban – набор принципов разработки проекта, в основе которого лежит визуализация потока разработки (доска со стикерами, к примеру) и строгое ограничение на количество выполненной работы за спринт.

* 1. Оценка задач проекта

Как известно, люди плохие оценщики в абсолютных величинах. Мы легко сможем сказать, что в среднем яблоко имеет размер теннисного мяча, но едва ли сможем сказать диаметр одного из них в сантиметрах. Предположим, нужно оценить новый проект. Разобьем его на небольшие модули, которые будут выполняться последовательно. Эти модули разобьем еще на атомарные подзадачи, срок выполнения которых не превышает 10 дней, не давая более точную оценку, сколько времени на их выполнение потребуется.

Рассортируем задачи в первом модуле по сложности. Теперь попытаемся оценить, сколько займет у нас выполнение этого модуля, оценив все его подзадачи. Для оценки часто прибегают к Planning poker(у).

*Planning poker.* Каждый член команды разработчиков (а иногда и тестировщиков) выбирает для оценки одну выданных ему/ей заранее карт. Карты представляют собой значения из чисел Фибоначчи (1, 2, 3, 5, 8 и т.д.). Эти цифры представляют собой число дней, которые предположительно придется затратить на выполнение задачи. Карты всех членов команды должны совпадать или отличаться незначительно. Если разбежка в значениях оказывается большой, те, кто выбрал наибольшее и наименьшее значения объясняют, почему они полагают, что они верные. Потом команда снова выбирает карты. И так, пока все не приходят к единому мнению.

Таким образом мы оценили примерные временные затраты на первый спринт. Далее приступаем к его выполнению. По мере выполнения, становится очевидна погрешность оценки. К примеру, первый спринт был при разбиении на модули самым маленьким и занял 10 рабочих дней. Второй модуль больше первого и трудозатраты на него, по нашим прикидкам, вдвое больше. Следовательно, второй спринт займет 4 недели. Что приводит нас к оцениванию модулей проекта в сравнении с первым спринтом. А это уже не так сложно.

Подводя итог этой методики, можно заключить, что изначально проект разбивается на подчасти, которые измеряются в некой абстрактной величине (попугаях). Первичной целью является понять, сколько же в нашем попугае дней.

Отмечу еще один интересный прием – калибровка. Со временем, после уже нескольких спринтов, может оказаться, что оценка временных затрат сильно отличается от реальной. В этом случае, следует снова рассортировать модули и попытаться заново оценить их величину относительно уже реализованных.

Конечно, planning poker не является единственным верным способом выставления оценок проекта. Хороший менеджер с большим опытом работы может выставлять оценки самостоятельно практически безошибочно. Особенно если он хорошо знает своих сотрудников, а те хорошо знают предметную область.

В заключение скажу, что гибкие методологии это сборники общепринятых практик, которые широко применяются менеджерами при разработке программного продукта. Мало кто использует методологии ровно так, как описано в книжках о них. В большинстве случаев, собирается приемлемый и удобный набор практик, которые реально помогают ускорить разработку и улучшить ее качество.

1. Обзор технологий
   1. Платформа .NET

При проектировании платформы Net Framework, компания Microsoft учла недостатки существующихWindows-платформ.NET Framework состоит из двух частей: общеязыковой исполняющей среды (commonlanguageruntime, CLR) и библиотеки классов (Framework Class Library, FCL). CLR предоставляет модель программирования, используемую во всех типах приложений. У CLR собственный загрузчик файлов, диспетчер памяти (сборщик мусора), система безопасности (безопасность доступа к коду), пул потоков и многое другое. Кроме того, CLR предоставляет объектно-ориентированную модель программирования, определяющую, как выглядят и ведут себя типы и объекты. FCL предоставляет объектно-ориентированный API-интерфейс, используемый всеми моделями приложений. В ней содержатся определения типов, которые позволяют разработчикам выполнять ввод/вывод, планирование задач в других потоках, создавать графические образы, сравнивать строки и т. п. Естественно, что все эти определения типов соответствуют существующей модели программирования в CLR. Ниже представлен список возможностей и преимуществ платформы NET:

- Полное и абсолютное межъязыковое взаимодействие. В .NET Framework. Поддерживаются межъязыковое наследование, межъязыковая обработка исключений и межъязыковая отладка.

- Общая среда выполнения для любых приложений .NET, вне зависимости от того, на каких языках они были созданы. Один из важных моментов при этом то, что для всех языков используется один и тот же набор встроенных типов данных.

- Единая программная модель. В отличие от существующего подхода, когда одни функции операционной системы доступны через процедуры динамически подключаемых библиотек (DLL), а другие - через СОМ-объекты, весь прикладной сервис представлен общей объектно-ориентированной программной моделью.

- Упрощенная модель программирования. Избавляет от работы с разными структурами, как это было с Win32 и СОМ. Так, разработчику не нужно разбираться с реестром, глобальными уникальными идентификаторами (GUID), IUnknown, AddRef, Release, HRESULT и т. д.

- Отсутствие проблем с версиями. Все Windows-разработчики знают о проблемах совместимости версий, известных под названием «DLL hell». Эта проблема возникает, когда компоненты, устанавливаемые для нового приложения, заменяют компоненты старого приложения, и в итоге последнее начинает вести себя странно или перестает работать. Архитектура .NET Framework позволяет изолировать прикладные компоненты, так что приложение всегда загружает компоненты, с которыми оно строилось и тестировалось. Если приложение работает после начальной установки, оно будет работать всегда.

- Упрощенное развертывание. Ранее Windows-приложения было очень трудно устанавливать и разворачивать: обычно нужно было создать массу файлов, параметров реестра и ярлыков. К тому же полностью удалить приложение практически невозможно. С приходом NET Framework все эти проблемы остаются в прошлом. Компоненты NET Framework не связаны с реестром. Установка приложений NET Framework сводится лишь к копированию файлов в нужные каталоги и созданию ярлыков. Удаление же приложений сводится к удалению файлов.

- Работа на многих платформах. При компиляции кода для .NET Framework компилятор генерирует код на общем промежуточном языке (CommonItermediateLanguage, CIL), а не традиционный код, состоящий из процессорных команд. При исполнении CIL транслируется в команды процессора. Поскольку трансляция выполняется в период выполнения, генерируются команды конкретного процессора. Это значит, что можно развертывать свое приложение NET Framework на любой машине, где работает версия .NET Framework соответствующая стандарту ЕСМА: с архитектурой х86, х64, IA64 и т. д.

- Интеграция языков программирования. Технология СОМ поддерживает взаимодействие разных языков – .NET Framework обеспечивает интеграцию разных языков, то есть один язык может использовать типы, созданные на других языках. Например, .NET Framework позволяет создать на C++ класс, производный от класса, реализованного на VisualBasic. В CLR это возможно из-за наличия общей системы типов (Common Type System, CTS), которую должны использовать все языки, ориентированные на CLR. Общеязыковая спецификация (Common Language Specification, CLS) определяет правила, которым должны следовать разработчики компиляторов, чтобы их языки интегрировались с другими. Сама Microsoft предлагает несколько таких языков: C++/CLI (C++ с управляемыми расширениями), С#, VisualBasic NET. Кроме того, другие компании и учебные заведения создают компиляторы других языков, совместимых с CLR.

- Упрощенное повторное использование кода. Все описанные выше механизмы позволяют создавать собственные классы, предоставляющие сервис сторонним приложениям. Теперь многократное использование кода становится исключительно простым и создается большой рынок готовых компонентов (типов).

- Автоматическое управление памятью (сбор мусора). Программирование требует большого мастерства и дисциплины, особенно когда речь идет об управлении использованием ресурсов (файлов, памяти, пространства экрана, сетевых соединений, ресурсов баз данных и прочих). Одна из самых распространенных ошибок - небрежное отношение к освобождению этих ресурсов, что может привести к некорректному выполнению программы в непредсказуемый момент. CLR автоматически отслеживает использование ресурсов, гарантируя, что не произойдет их утечки.

- Проверка безопасности типов CLR может проверять безопасность использования типов в коде, что гарантирует корректное обращение к существующим типам. Если входной параметр метода объявлен как 4-байтное значение, CLR обнаружит и предотвратит передачу 8-байтного значения в качестве значения этого параметра. Безопасность типов также означает, что управление может передаваться только в определенные точки (точки входа методов). Невозможно указать произвольный адрес и заставить программу исполняться, начиная с этого адреса. Совокупность всех этих защитных мер избавляет от многих распространенных программных ошибок (например, от возможности использования переполнения буфера для «взлома» программы).

- Развитая поддержка отладки. Поскольку CLR используется для многих языков, можно написать отдельный фрагмент программы на языке, наиболее подходящем для конкретной задачи, – CLR полностью поддерживает отладку многоязыковых приложений.

- Единый принцип обработки сбоев. Один из самых неприятных моментов Windows-программирования – несогласованный стиль сообщений о сбоях. Одни функции возвращают коды состояний Win32, другие – HRESULT, третьи генерируют исключения. В CLR обо всех сбоях сообщается через исключения, которые позволяют отделить код, необходимый для восстановления после сбоя, от основного алгоритма. Такое разделение облегчает написание, чтение и сопровождение программ. Кроме того, исключения работают в многомодульных и многоязыковых приложениях. И в отличие от кодов состояний и HRESULT исключения нельзя проигнорировать. CLR также предоставляет встроенные средства анализа стека, заметно упрощающие поиск фрагментов, вызывающих сбои.

- Безопасность. Традиционные системы безопасности обеспечивают управление доступом на основе учетных записей пользователей. Это проверенная модель, но она подразумевает, что любому коду можно доверять в одинаковой степени. Такое допущение оправданно, когда весь код устанавливается с физических носителей (например, с компакт-диска) или с доверенных корпоративных серверов. Но по мере увеличения объема мобильного кода, например Web-сценариев, приложений, загружаемых из Интернета, и вложений, содержащихся в электронной почте, нужен ориентированный на код способ контроля за поведением приложений. Такой подход реализован в модели безопасности доступа к коду.

- Взаимодействие с существующим кодом. В Microsoft понимают, что разработчики накопили огромный объем кода и компонентов. Переписывание всего этого кода, так чтобы он задействовал все достоинства NET Framework, значительно замедлило бы переход к этой платформе. Поэтому в .NET Framework реализована полная поддержка доступа к СОМ-компонентам и Win32-функциям в существующих динамических библиотеках DLL.

При работе с платформой .NET можно создавать файлы с исходным кодом на любом языке, поддерживающем CLR. Затем соответствующий компилятор проверяет и анализирует исходный код. Независимо от компилятора результатом его работы является управляемый модуль (managedmodule) – стандартный переносимый исполняемый (portableexecutable, РЕ) файл 32-разрядной (РЕ32) или 64-разрядной Windows (PE32+), который требует для своего выполнения исполняемую среду CLR.

В прошлом почти все компиляторы генерировали код для конкретных процессорных архитектур, таких как x86, IA64, Alpha или PowerPC. Все CLR-совместимые компиляторы вместо этого генерируют IL-код. IL-код иногда называют управляемым (managedcode), потому что CLR управляет его жизненным циклом и выполнением.

Каждый компилятор, предназначенный для CLR, кроме генерации IL-кода, также должен создавать полные метаданные (metadata) для каждого управляемого модуля. Коротко говоря, метаданные – это просто набор таблиц данных, описывающих то, что определено в модуле, например типы и их члены. Метаданные служат многим целям:

- Они устраняют необходимость в заголовочных и библиотечных файлах прикомпиляции, так как все сведения о типах/членах, на которые есть ссылки, содержатся в файле с IL-кодом, в котором они реализованы. Компиляторы могут читать метаданные прямо из управляемых модулей.

- В процессе верификации кода CLR использует метаданные, чтобы убедиться, что код совершает только «безопасные» операции.

- Метаданные позволяют сериализовать поля объекта в блок памяти на удаленной машине и затем десериализовать, восстановив объект и его состояние на этой машине.

- Метаданные позволяют сборщику мусора отслеживать жизненный цикл объектов. Используя метаданные, сборщик мусора определяет тип объектов и узнает, какие поля в них ссылаются на другие объекты.

Метаданные расширяют возможности таких старых технологий, как библиотеки типов и файлы языка описания интерфейсов (Interface Definition Language, IDL). Важно заметить, что метаданные CLR гораздо полнее. И в отличие от библиотек типов и IDL они всегда связаны с файлом, содержащим IL-код. Фактически метаданные всегда встроены в тот же ЕХЕ/DLL, что и код, так что их нельзя разделить. Так как компилятор генерирует метаданные и код одновременно и привязывает их к конечному управляемому модулю, рассинхронизация метаданных и описываемого ими IL-кода исключена.

После компиляции набор управляемых модулей объединяется в сборку. Сборка – это логическая группировка одного или нескольких управляемых модулей или файлов ресурсов. Это самая маленькая единица с точки зрения повторного использования, безопасности и управления версиями. Сборка может состоять из одного или нескольких файлов – все зависит от выбранных средств и компиляторов.

Можно сделать вывод, что платформа Microsoft NET Framework в свое время явилась большим достижением в области разработки программного обеспечения, предоставляя уникальные инновационные возможности.

Также необходимо отметить, что с момента выпуска первой версии платформы NET Framework 1.0 она претерпела некоторые изменения и много дополнений, которые также призваны повысить эффективность разработки. Компания Microsoft продолжит развитие своей платформы и в будущем.

Таким образом, платформа Microsoft NET Framework является прекрасной универсальной платформой для разработки многочисленных типов программных средств, начиная от простых настольных программ, заканчивая сложными корпоративными системами и серверами.

* 1. ADO.net

ADO .NET (ActiveX Data Objects .NET) является набором классов, реализующих программные интерфейсы для облегчения подключения к базам данных из приложения независимо от особенностей реализации конкретной системы управления базами данных и от структуры самой базы данных, а также независимо от места расположения этой самой базы — в частности, в распределенной среде (клиент-серверное приложение) на стороне сервера.

ADO .NET широко используется совместно с технологией web-программирования с использованием объектов ASP .NET для доступа к расположенным на сервере базам данных со стороны клиента.

Решение даже самой простой задачи, связанной с данными, предполагает использование множества разнообразных объектов – представителей классов ADO .NET, которые находятся между собой в достаточно сложных взаимоотношениях. Из-за этих отношений строго последовательное описание элементов ADO .NET представляется весьма проблематичным. С какого бы элемента ни начиналось описание, всегда предполагается предварительное представление о множестве других элементов.

По этой причине часто упоминание и даже примеры использования некоторых классов предшествуют их подробному описанию.

ADO .NET используется, чтобы максимально скрыть детали реализации конкретной базы и предоставить программисту набор стандартных классов, интерфейсов, программных средств, которые превращают процесс создания приложения в стандартизированный технологический процесс.

В приложениях, работающих с базами данных, до недавних пор применялся доступ к данным через постоянное соединение с источником данных. Приложение открывало соединение с базой данных и не закрывало его по крайней мере до завершения работы с источником данных. В это время соединение с источником поддерживалось постоянно.

Недостатки такого подхода стали выявляться после появления приложений, ориентированных на Интернет.

Соединения с базой данных требуют выделения системных ресурсов, и если база данных располагается на сервере, то при большом количестве клиентов это может быть критично для сервера. Хотя постоянное соединение и позволяет немного ускорить работу приложения, общий убыток от растраты системных ресурсов преимущество в скорости выполнения приложения сводит на нет.

Факт плохого масштабирования приложений с постоянным соединением известен давно. Соединение с парой клиентов обслуживается приложением хорошо, 10 клиентов обслуживаются хуже, 100 – много хуже...

В ADO .NET используется другая модель доступа – доступ к отсоединенным данным. При этом соединение устанавливается лишь на то время, которое необходимо для проведения определенной операции над базой данных.

Модель доступа – модель компромиссная. В ряде случаев она проигрывает по производительности традиционной модели, и для этих случаев рекомендуется вместо ADO .NET использовать ADO.

* 1. MVC.net

Шаблон архитектуры Model-View-Controller (MVC) разделяет приложение на три основных компонента: модель, представление и контроллер.Платформа ASP.NET MVC представляет собой альтернативу схеме веб-форм ASP.NET при создании веб-приложений. Платформа ASP.NET MVC является легковесной платформой отображения с широкими возможностями тестирования и, подобно приложениям на основе веб-форм, интегрирована с существующими функциями ASP.NET, например с главными страницами и проверкой подлинности на основе членства. Платформа MVC определяется в сборке System.Web.Mvc.

MVC представляет собой стандартный шаблон разработки, знакомый многим специалистам. Некоторые типы веб-приложений имеют преимущества при создании на платформе MVC. Для других может быть целесообразно использование традиционной схемы приложения ASP.NET, основанной на веб-формах и обратной передаче. В некоторых случаях возможно сочетание двух подходов: применение одной схемы не исключает использования другой.

В состав платформы MVC входят следующие компоненты.

* *Модели.* Объекты моделей являются частями приложения, реализующими логику для домена данных приложения. Объекты моделей часто получают и сохраняют состояние модели в базе данных. Например, объект Product может получать информацию из базы данных, работать с ней, а затем записывать обновленные данные в таблицу Products базы данных SQL Server.

В небольших приложениях эта модель подразумевает концептуальное, а не физическое разделение. Например, если приложение только считывает набор данных и отправляет его в представление, то физический слой модели и связанных классов отсутствует. В этом случае набор данных принимает роль объекта модели.

* *Представления.* Представления служат для отображения пользовательского интерфейса приложения. Пользовательский интерфейс обычно создается на основе данных модели. Примером может служить представление для редактирования таблицы Products, которое содержит текстовые поля, раскрывающиеся списки и флажки, значения которых основаны на текущем состоянии объекта Product.
* *Контроллеры.* Контроллеры осуществляют взаимодействие с пользователем, работу с моделью, а также выбор представления, отображающего пользовательский интерфейс. В приложении MVC представления только отображают данные, а контроллер обрабатывает вводимые данные и отвечает на действия пользователя. Например, контроллер может обрабатывать строковые значения запроса и передавать их в модель, которая может использовать эти значения для отправки запроса в базу данных.

Шаблон MVC позволяет создавать приложения, различные аспекты которых (логика ввода, бизнес-логика и логика интерфейса) разделены, но достаточно тесно взаимодействуют друг с другом. Эта схема указывает расположение каждого вида логики в приложении. Пользовательский интерфейс располагается в представлении. Логика ввода располагается в контроллере. Бизнес-логика находится в модели. Это разделение позволяет работать со сложными структурами при создании приложения, так как обеспечивает одновременную реализацию только одного аспекта. Например, разработчик может сконцентрироваться на создании представления отдельно от бизнес-логики.

Связь между основными компонентами приложения MVC также облегчает параллельную разработку. Например, один разработчик может создавать представление, другой — логику контроллера, а третий — бизнес-логику модели.

В дополнение к упрощению сложных структур схема MVC также облегчает тестирование приложений по сравнению с веб-приложениями ASP.NET на основе веб-форм. Например, в веб-приложении ASP.NET на основе веб-форм один класс используется для отображения вывода и для ответа на ввод пользователя. Создание автоматических тестов для приложений ASP.NET на основе веб-форм может представлять сложности, так как для тестирования отдельной страницы следует создать экземпляр класса страницы, всех дочерних элементов управления и других зависимых классов приложения. Большое число экземпляров классов, необходимое для запуска страницы, усложняет создание тестов для отдельных частей приложения. Из-за этого тестирование приложений ASP.NET на основе веб-форм может быть сложнее тестирования приложения MVC. Более того, для тестирования приложения ASP.NET необходим веб-сервер. Платформа MVC разделяет компоненты и активно использует интерфейсы, что позволяет тестировать отдельные элементы вне остальной структуры.

Преимущества веб-приложения на основе MVC:

* Она облегчает управление сложными структурами путем разделения приложения на модель, представление и контроллер.
* Она не использует состояние просмотра и серверные формы. Это делает платформу MVC идеальной для разработчиков, которым необходим полный контроль над поведением приложения.
* Она использует схему основного контроллера, при которой запросы веб-приложения обрабатываются через один контроллер. Это позволяет создавать приложения, поддерживающие расширенную инфраструктуру маршрутизации.
* Она обеспечивает расширенную поддержку разработки на основе тестирования.
* Она хорошо подходит для веб-приложений, поддерживаемых крупными коллективами разработчиков, а также веб-разработчикам, которым необходим высокий уровень контроля над поведением приложения.

Платформа ASP.NET MVC предоставляет следующие возможности.

* 1. Разделение задач приложения (логика ввода, бизнес-логика и логика пользовательского интерфейса), широкое возможности тестирования и разработки на основе тестирования. Все основные контракты платформы MVC основаны на интерфейсе и подлежат тестированию с помощью макетов объекта, которые имитируют поведение реальных объектов приложения. Приложение можно подвергать модульному тестированию без запуска контроллеров в процессе ASP.NET, что ускоряет тестирование и делает его более гибким. Для тестирования возможно использование любой платформы модульного тестирования, совместимой с .NET Framework.
  2. Расширяемая и дополняемая платформа. Компоненты платформы ASP.NET MVC можно легко заменить или настроить. Разработчик может подключать собственный механизм представлений, политику маршрутизации URL-адресов, сериализацию параметров методов действий и другие компоненты. Платформа ASP.NET MVC также поддерживает использование моделей контейнера внедрения зависимости (DI) и инверсии элемента управления (IOC). Модель внедрения зависимости позволяет внедрять объекты в класс, а не ожидать создания объекта самим классом. Модель инверсии элемента управления указывает на то, что если один объект требует другой объект, то первые объекты должны получить второй объект из внешнего источника (например, из файла конфигурации). Это облегчает тестирование.
  3. Расширенная поддержка маршрутизации ASP.NET. Этот мощный компонент сопоставления URL-адресов позволяет создавать приложения с понятными URL-адресами, которые можно использовать в поиске. URL-адреса не должны содержать расширения имен файлов и предназначены для поддержки шаблонов именования URL-адресов, обеспечивающих адресацию, оптимизированную для поисковых систем (SEO) и для передачи репрезентативного состояния (REST).
  4. Поддержка использования разметки в существующих файлах страниц ASP.NET (ASPX), элементов управления (ASCX) и главных страниц (MASTER) как шаблонов представлений. Вместе с платформой ASP.NET MVC можно использовать существующие функции ASP.NET, например вложенные главные страницы, встроенные выражения (<%= %>), декларативные серверные элементы управления, шаблоны, привязку данных, локализацию и т. д.
  5. Поддержка существующих функций ASP.NET. ASP.NET MVC позволяет использовать такие функции, как проверка подлинности с помощью форм и Windows, проверка подлинности по URL-адресу, членство и роли, кэширование вывода и данных, управление состоянием сеанса и профиля, наблюдение за работоспособностью, система конфигурации и архитектура поставщика.

1. Архитектура приложения и особенности реализации
   1. Описание системы
      1. Состав системы

Структурная схема системы представлена на рисунке 3-1.

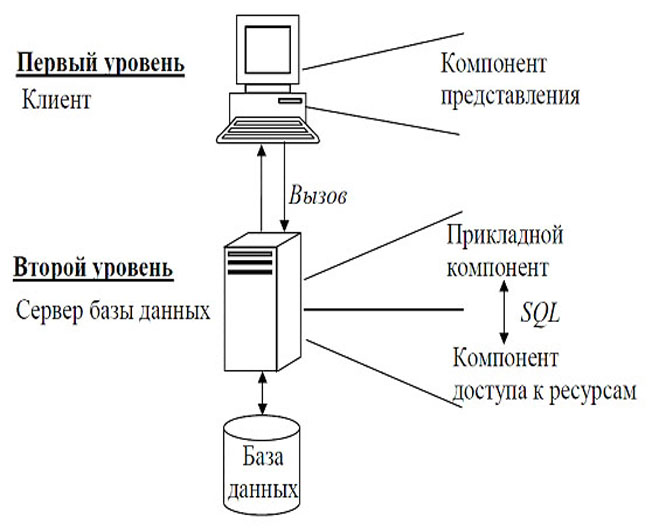


Рисунок 3-1: структурная схема системы

В системе распределения ресурсов реализована трехуровневая архитектура. Первый уровень, клиент, отвечает за представление данных пользователю приложения и ничем более. Связь клиента со вторым уровнем (бизнес логика – business logic) обеспечивается только посредством интерфейсов. На втором уровне реализована вся логика приложения. Через него же происходит доступ к базам данных – третьему уровню.

Третий уровень (доступа к данным – data access layer) только считывает из базы данных (также XML и т.п.) данные и выдает их в BL или записывает в файл.

Нельзя допустить какие-либо связи первого и третьего уровней.

Как видим, функции каждого слоя строго разграничены, каждый компонент приложения занимается своим делом.

* + 1. Функциональное назначение элементов системы

Потому как система оптимизированного распределения ресурсов Loop является клиент-серверным приложением, первый слой (пользовательский интерфейс – user interface) содержит разметку страниц, стили, а так же немногочисленные методы, написанные на javascript, отвечающие за обработку действий пользователя при работе на страницах.

Второй уровень – бизнес логика – содержит все методы, отвечающие за работу приложения. В том числе обработка запросов клиента, составление запросов в базу данных и формирования ответа клиенту. Здесь же расположен весь функционал: расчет статистики, составления списка задач на спринт, добавление, удаление и редактирование элементов системы.

Третий слой отвечает за взаимодействие с хранилищами данных. Он предназначен для сохранения в базе данных информации пользователях системы, их правах, элементах в системе и их взаимодействии.

Благодаря трехуровневой структуре приложения обеспечивается максимальная независимость бизнес логики от пользовательского интерфейса и хранилища данных. Так, ядро системы является полностью отделяемой частью системы, а приложение Loop является всего лишь частным случаем его использования.

Рассмотрим подробнее систему взаимодействия частей системы.

Для обеспечения независимости ядра был реализован шаблон проектирования «Мост» (рисунок 3-2).

Bridge.emf

Рисунок 3-2: диаграмма классов шаблона “Мост”

Шаблон Мост используется в тех случаях, когда имеется отдельная иерархия абстрактных классов и интерфейсов и соответствующая иерархия реализаций. Мост соединяет абстракции и реализации в виде независимых динамических классов.

Модуль ядра поставляется с набором интерфейсов, которые необходимо реализовать для успешной работы ядра. В качестве параметров конструктора ядра, должны быть переданы объекты реализаций этих интерфейсов. Так пользователь может определить своего поставщика данных и свой клиент приложения, не меняя методы ядра.

Взаимодействие с базой данных происходит через интерфейс. Что позволяет создать абстракцию над хранилищем данных. Это обеспечивает дополнительную расширяемость слоя доступа данных, который в свою очередь является частным случаем реализации доступа данных для ядра.

В перспективе пользователь может определить несколько источников данных в своем приложении, при помощи фабричного метода возвращать различную реализацию поставщика данных, который будет передан в ядро для обработки запросов с клиента.

Рассмотрим пример взаимодействия компонентов системы.

Клиент посылает запросы на сервер. Запросы содержат критерии, согласно которым сервер должен вернуть информацию. Критерии, в случае POST запросов, отправляются в формате json и в этом же формате сервер возвращает ответ. В случае Get запроса записываются дополнительными параметрами в строке адреса запроса.

Рассмотрим пример GET запроса.

Клиент делает запрос на сервер, адрес имеет формат:

http://{servername}/Admin/Task/GetTasks?\_search=false&nd=1369232411321&rows=10&page=1&sidx=Title&sord=asc

Где Admin – роль пользователя, залогиненного в системе.

Task – тип данных системы, с которым идет взаимодействие.

GetTasks – действие, которое должно выполниться на сервере.

Далее идут дополнительные параметры запроса. Потому как Get запрос не может содержать никаких данных в теле, эти данные записываются параметрами адреса.

Сервер, получив этот запрос, анализирует данные, которые ему пришли. В данном случае это запрос на список всех задач. Параметры запроса используются для формирования запроса в хранилище данных.

Когда данные из хранилища получены, в слое бизнес логики формируется ответ клиенту в формате json. В данном случае это коллекция json объектов, описывающих задачи.

Ответом сервера будет следующий набор данных:

{

"total":1,

"page":1,

"records":3,

"rows":[

{

"id":1,

"cell":["IR-0001111","Test title456","1","1","True"]

},

{

"id":7,

"cell":["IR-00022211","TestTitle","1","1","False"]

},

{

"id":10,

"cell":["IR-12121214","Sandra14","2","2","True"]

}]

}

Эта структура содержить описание, как должны располагаться данные в тарблице, где они будут отображаться (общее число страниц, номер текущей страницы, число записей на странице), а так же данные самих записи.

Получив эти данные, клиент запускает скрипт, который обеспечивает их отображение пользователю.

Рассмотрим пример POST запроса.

Клиент делает запрос на сервер, адрес имеет формат:

http://{servername}/Admin/Task/StartTask

Как видим, никаких дополнительных параметров адрес не содержит. Все данные для обработки запроса описаны в его теле.

{

"number":"IR-0001111",

"title":"Test title456",

"priority":"1",

"assignment":"1",

"isactive":"True",

"actions":"",

"id":"1"

}

Эта структура содержит данные о задаче, которую нужно стартовать. Обработав эти параметры сервер вернет результат.

В данном случае результатом будет та же самая запись, но с обновленными, во соответствии с новым статусом данными.

{

"number":"IR-0001111",

"title":"Test title456",

"priority":"1",

"assignment":"1",

"isactive":"True",

"actions":"",

"id":"1"

}

* + 1. Основные функции элементов системы

Система подразумевает работу пользователей с различными ролями, поэтому в приложении предусмотрены 3 уровня доступа к функционалу.

Администратор – имеет полный доступ ко всем функциям системы. Администратор управляет данными в системе. Он может управлять пользователями: добавлять, удалять, редактировать, назначать задачи на определенных пользователей.

Пользователь – права пользователей системы имеют люди, непосредственно занятые в разработке программного продукта, такие как разработчики, тестировщики и менеджеры. Они могут просматривать список задач, переводить задачи из состояния в состояние.

Клиент – самый низкий уровень доступа. Права клиента имеют заказчики программного продукта. Подразумевается, что для них нет необходимости вникать в технические детали разработки проекта. Для них предусмотрен только просмотр динамики проекта, а так же список целевых задач в режиме «только для чтения».

Основные функции администратора показаны на рисунке 3-3.

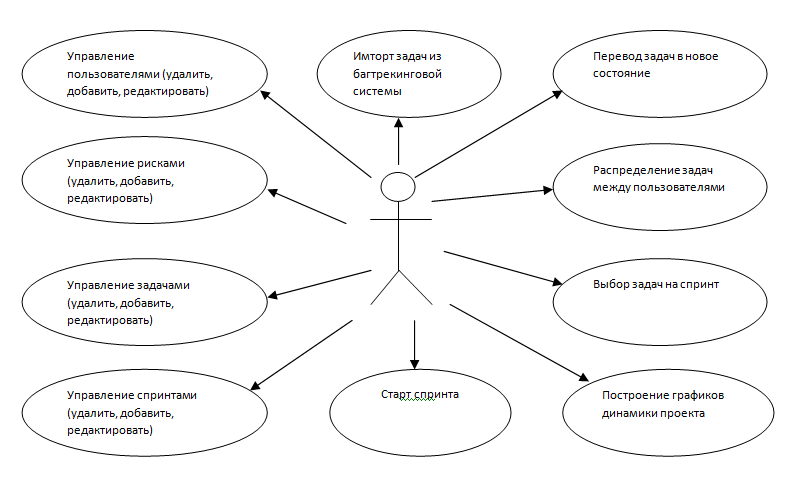


Рисунок 3-3: упрощенная use-case диаграмма для администратора

Опишем подробнее основные функции системы.

На диаграмме коротко описаны все основные функции доступные администратору.

*Управление пользователями* – функционал позволяющие добавлять, удалять и редактировать пользователей системы. Этот функционал является настолько стандартным и примитивным, что он был вынесен в один отдельный блок.

*Управление задачами*. Задачи в системе распределения ресурсов - это работа, которую нужно выполнить в определенные срок. Это могут быть ошибки в программе, которые необходимо исправить или новый функционал программы, который нужно реализовать. Под управлением ими, так же, как и в управлении пользователями, подразумевается удаление, добавление и редактирование (рисунок 3-4).

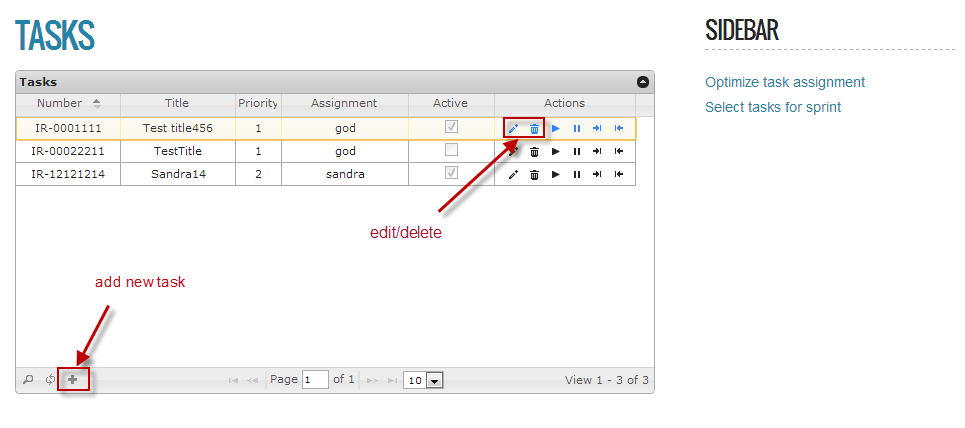


Рисунок 3-4: управление данными в Loop

*Управление рисками*. Риски эта та самая неуловимая штука, которая обычно портит менеджерам статистику. Добавляя элементы этого типа, администратор укажет, что при разработке проекта, нужно делать поправки на некие неточности, которые впоследствии будут учитываться. В проекте Loop риски могут быть различных типов: per Task, per User, per Sprint. В случае если сотрудник болеет или у него возникают личные проблемы это per User риск. Если задача описана недостаточно детально и ее реализация требует дополнительных трудозатрат это per Task риск. Если же заказчик говорит, что в рамках спринта должен быть разработан дополнительный функционал это per Sprint риск. Болезнь сотрудника, недетальное описание, увеличение числа задач в данном случае – риски приложения Loop, имеющие различные типы. В зависимости от типа система пытается учитывать эти риски в различных местах.

Управлять рисками тут – добавлять новые риски, удалять или редактировать существующие.

*Управление спринтами* – риски учитываются в течение определенного промежутка времени, называемого спринтом. В рамках спринта активные пользователи решают назначенные на них задачи. По итогам спринта учитываются риски, которые реализовались во время спринта. Управление ими опять-таки это их удаление, добавление или редактирование.

Импорт задач из багтрекинговой системы – позволяет подключаться к багтрекинговой системе проекта и импортировать задачи оттуда по определенным критериям, копируя их в базу данных приложения.

*Перевод задач в новое состояние* – задачи Loop всегда имеют какое-то состояние и статус. О статусах и состояниях задач будет описано подробнее в описании основных методов ядра, потому как переход между состояниями происходит благодаря реализации конечного автомата в ядре системы.

*Выбор задач на спринт* – приложение предоставляет возможность по приоритетам задач и времени, необходимом для их реализации, выбрать задачи, которые могут быть реализованы активными пользователями в двухнедельный срок. Проще говоря, это решение задачи о рюкзаке, где приоритет задачи – важность вещи, а временная оценка – ее вес.

*Распределение задач между пользователями* – если предположить, что на каждого активного пользователя системы у нас собран некий набор статистических данных, то этот функционал позволяет, с учетом статистических данных, делая поправки на особенности пользователей, распределить задачи между разработчиками так, чтобы с максимальной вероятностью списаться в отведенные на разработку сроки.

*Построение графиков динамики спринта* - в agile существует понятие графика сгорания, который отражает разницу между реальным положением дел при разработке проекта и прогнозами, которые были на него сделаны (рисунок 3-5).

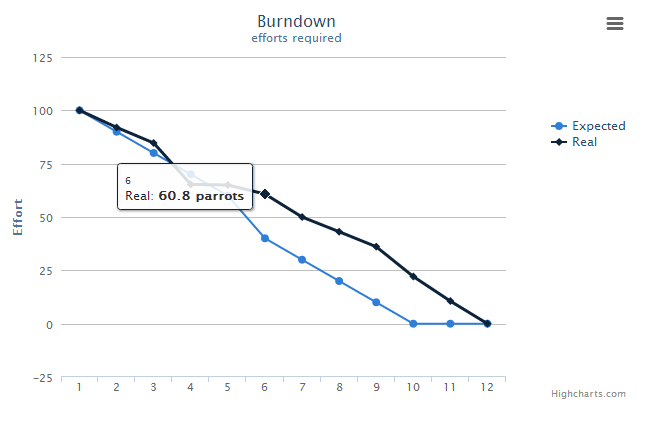


Рисунок 3-5: график сгорания в система Loop

На рисунке 3-6 пример такого графика. Голубая линия показывает, какой объем работ должен был быть выполнен в определенный день, а синяя – какой объем работ был выполнен реально. Так на 3 дне разработки наметилось некоторое отставание от графика, но на 4 дне разработчикам удалось догнать намеченный объем работ.

*Старт спринта* – разработчик, который имеет более низкий уровень доступа, чем администратор, может видеть только задачи, назначенные на него в активном спринте. Это делает поиск работы, которую необходимо выполнить, максимально простой для разработчика.

* 1. Особенности реализации
     1. Импорт и экспорт данных

При разработке любого проекта существует система отслеживания ошибок в программе – багтрекинговая система. Система содержит не только список ошибок, но и список задач, которые необходимо реализовать. Записи туда ведутся на протяжении разработки проекта, а так же во время его сопровождения. Если проект оказывается большим и существует на рынке какое-то время, количество задач в багтрекинговой системе может оказаться огромным и навигация по ним может оказаться проблематичной.

Так, опция импорта небольшого набора данных, которые будут использоваться именно во время спринта, может оказаться полезной.

Для этого в системе присутствует интерфейс IDataExporter.

Реализовав этот интерфейс и передав его в качестве параметра в соответствующий метод ядра, пользователь получает данные указанным им способом из любого поставщика данных, будь то API багтрекинговой системы или простой xml файл на его машине.

Также любой менеджер сталкивается во время разработки с отчетностью. Более высокое начальство и клиенты зачастую любят просматривать отчеты в MS Projectе, в которым отражены объемы работ, которые нужно выполнить и их статус готовности.

Loop для этого предусматривает интерфейс IDataExporter, реализовав который пользователь может определить, какие данные и как он хочет реализовать.

Так, реализовав способ, к примеру, экспорта данных в Excel и передав реализацию этого интерфейса в метод ядра, пользователь может автоматически генерировать отчеты по задачам.

Подробнее методы ядра будут рассмотрены далее.

* + 1. Система интерфейсов ядра

Для предоставления пользователю работы с ядром, разработана система интерфейсов, реализовав которые можно использовать методы ядра.

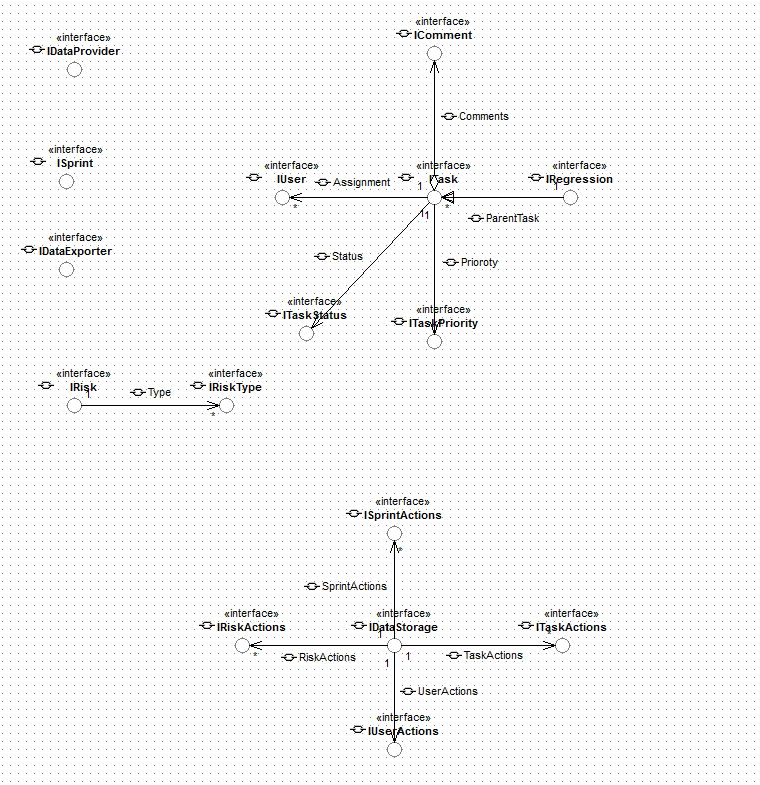
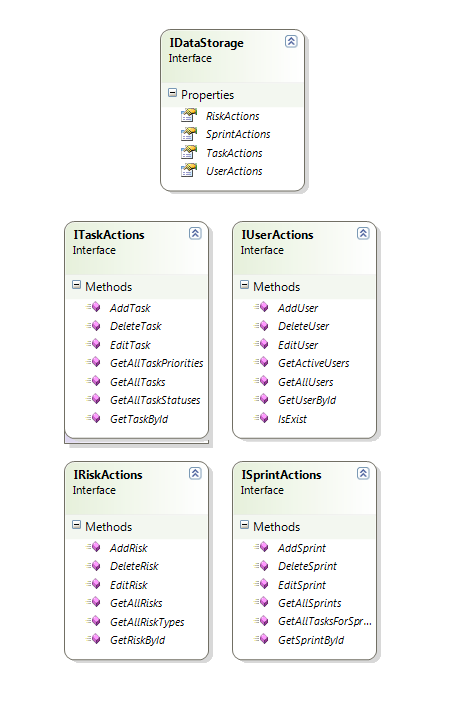
Полная система интерфейсов представлена на рисунке 3-6.

Рисунок 3-6: система интерфейсов ядра Loop

Для работы ядра необходимо реализовать 1 интерфейс: интерфейс для работы с хранилищем данных IDataStorage.

Этот интерфейс агрегирует 4 интерфейса, имеющие схожее строение, но отвечающие за работу с различными сущностями проекта (Task, Risk, User, Sprint).

Строение интерфейсов ITaskActions, IUserActions, ISprintActions, IRiscActions и IDataStorage приведены ниже (рисунок 3-7).

Рассмотрим подробнее строение интерфейсов.

**Рисунок 3-7: интерфейсы для работы с базой данных**

ITaskActions – интерфейс для работы с задачами. Предоставляет методы для добавления, удаления и редактирования задач в базе даннх. Помимо этого, должны быть реализованы функции, возвращающие приоритеты задач, статусы задач и возможность получение задачи по ее Id.

IUserActions – интерфейс для работы с пользователями. Предостовляет методы для добавления, удаления и редактирования пользователей в базе данных. Также есть метод для получения всех пользователей, и пользователя по его Id. Метод GetActiveUsers возвращает список активных пользователей. На любом проекте существует некоторая «текучка» кадров: люди уходят в отпуск, на больничный, в декрет или вовсе увольняются. Активным считатется пользователь, который может выполнять назначенную на него работу, то есть, к примеру, появляется в офисе каждый рабочий день с 9 до 17. Метод IsExist проверяет, существует ли указанный в параметре метода пользователь в базе данных.

IRiskActions – интрефейс для работы с рисками. Предоставляет методы для добавление, удаления и редактирования рисков в базе данных, а так же возможность получения всех рисков, всех типов рисков или определенного риска по его Id.

ISprintActions – интерфейс для работы со спринтами. Предоставляет методы для удаления, добавления и редактирования спринтов в системе. Так же есть методы для получения всех задач по Id спринта, всех спринтов или определенного спринта по его Id.

Теперь подробнее рассмотрим интерфейсы сущностей.

На рисунке 3-8 представлены все интерфейсы сущностей и их связи.

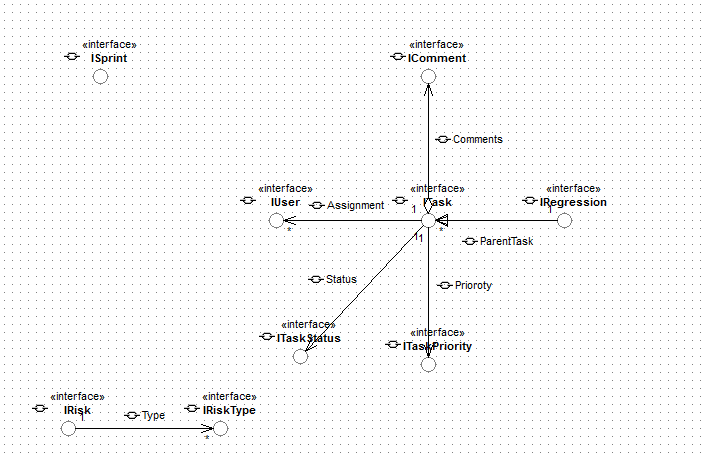


Рисунок 3-8: интерфейсы сущностей ядра

На рисунке 3-9 показаны все свойства и методы интерфейсов сущностей.

Помимо интерфейсов на рисунке 3-9 описаны 2 перечисления StatusStae и UserType.

StatusState – описывает возможные состояния задач в любом из статусе. Так задача в статусе Development может иметь состояние Pause, к примеру, в случае, если сотрудник его выполняющий заболел.

UserType – как уже был упомянуто, пользователи могут иметь различные права, но так же они могут выполнять различные функции. В зависимости от этих функций, им присваивается UserType. Впоследствии это помогает более эффективно фильтровать задачи для пользователей различных типов.

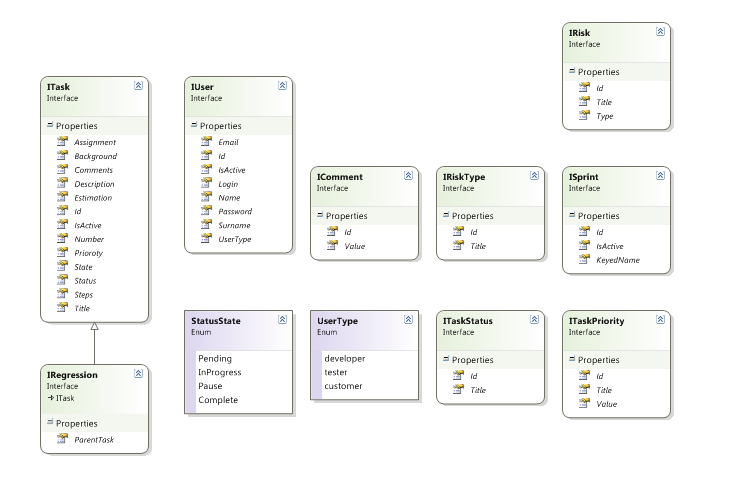


Рисунок 3-9: свойства и методы интерфейсов сущностей

IUser – интерфейс описывает пользователя в системе. Содержит стандартные для описания пользователя поля: Login, Password, Id и т.д.

IRisk – интерфейс описывающий риск. Риск, как и любая другая сущность системы, имеет Id, а так же название риска и его тип. Тип риска описывается другим интерфейсом IRiskType. Пользователь может определять свои типы риска и, подписавшись на событие подсчета статистики в системе, дописывать свою логику для этих типов. Это позволяет системе не замыкаться на существующих типах рисков, для которых предусмотрен специальный механизм рассчета статистики.

ISprint – интерфейс описывающий спринт. Спринт и задача связаны отношением многие ко многим, поэтому между ними не отображено связи на диаграмме классов. Для получения задач, назначенных на спринт, реализуется метод в поставщике данных GetAllTasksForSprint, кторый делает выборку данных непосредственно из таблицы и возвращает набор задач. Свойство IsActive этого интерфейса, указывает, что спринт в процессе и задачи на него назначенные закрыты не были. Это позволяет упростить фильтрацию данных для пользователей системы.

ITask – ключевой интерфейс системы, описывающий задачи, которые необходимо выполнить. Свойство Assignment имеет тип IUser и указывает на пользователя, который должен выполнить эту задачу. Свойства Background, Description, Number, Steps и Title – описывают стандартные поля задачи в багтрекинговой системе. Estimation – выставляется вручную администратором или менеджером и представляет временные оценки задачи. IsActive, как уже упоминалось ранее, указывает на то, что задача активна в рамках спринта. Это значит, что активный спринт не может оказаться в неактивном состоянии, пока эта (и другие активные) задача не будет завершена (то есть перейдет в статус Complete). Priority свойство имеет тип ITaskPriority, описываюего приотитет задачи. Спецификой ITaskPriority является то, что у его есть 2 основных свойства (помимо Id), одно из которых служит название приоритета (видно пользователю), а второе – его значимость (вес при решении задачи о рюкзаке). Далее идут свойства State и Status описывающие состояние задачи в рамках рассмотренного ранее конечного автомата. Status свойтсво имеет тип ITaskStatus, который и описывает статус задачи. И, наконец, свойство Comments это коллекция элементов типа IComment, содержащих дополнительную информацию по задаче от различных пользователей.

От интерфейса ITask наследуется интерфейс IRegression, который описывает задачи, которые появились из-за каких-то изменений в существующей и ранее работавшей функциональности. Он имеет одно дополнительное свойство ParentTask которое указывает на задачу, в результате решения которой появился регрешн.

* + 1. Основные методы ядра

Рассмотрим основные функциональные возможности самого ядра системы. На рисунке 3-10 показан класс Core и набор его public методов и свойств.

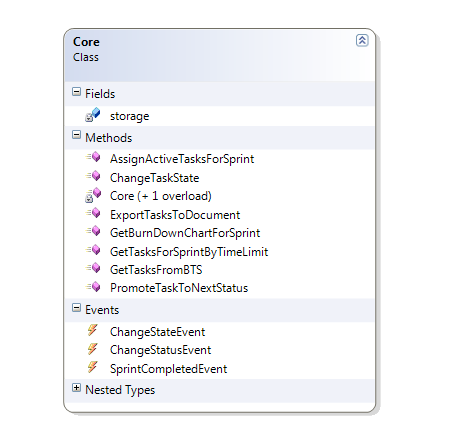


Рисунок 3-10: класс ядра, его public методы и свойства

Как уже упоминалось ранее, ядро зависит от реализации интерфейса IDataStorage, который является типом единственного свойства класса ядра – storage. Это свойство выставляется при инициализации ядра и используется всеми его методами.

GetTasksFromBTS – метод возвращает набор задач, полученных из багтрекинговой системы. Интерфейс IDataProvider, принимаемый этим методом в качестве параметра, обеспечивает кастомизируемость поставщика данных для задач.

ExportTasksToDocument – метод отвечающий за экспорт задач в документ. Документ и способ экспорта задается пользователем, когда тот реализует интерфейс IDataExporter и передает эту реализацию в качестве параметра метода.

GetBurndownChartForSprint – метод возвращает список из классов BurndownPoint (см. рисунок 3-11). Этот класс описывает точку на графике сгорания, состоящую из 2 частей: ожидаемая и реальная. Каждая из этих точек представлена классом ChartPoint.

Когда пользователь начинает выполнение своей задачи (нажимает кнопку “старт” на пользовательском интерфейсе) стартует счетчик, который должен отслеживать, сколько ведется разработка задачи. График должен отражать, существует ли разница между ожидаемыми результатами и реальными.

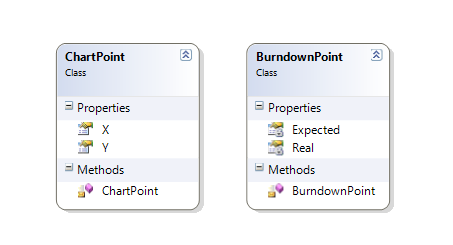


Рисунок 3-11: классы для описания точек на графиках

GetTasksForSprintByTimeLimit – метод должен решить задачу о рюкзаке для задач, назначенных на спринт, переданный в качестве параметра метода. Так, здесь выбираются задачи на для спринта и по их оценке подсчитывается какие задачи максимального приоритета, могут быть решены в заданный временной промежуток. Список задач возвращается пользователю. Далее он может выдать этот список для подтверждения пользователю или автоматически сделать эти задачи активными для спринта.

AssignActiveTasksForSprint – высчитав, какие задачи мы теоретически можем решить в заданный временной промежуток, нужно назначить их на активных пользователей системы. Что и делает этот метод. Формально он решает задачу об оптимальном распределении ресурсов, учитывает статистические данные пользователей (per user риски). Так, если производительность пользователя, к примеру, 0.7, то задача estimation которой 3 дня будет решена этим пользователем за 4 дня. Учитывая все это задачи распределяются между пользователями и на выходе мы опять-таки получаем список задач с “черновыми” назначениями, которые пользователь может выдать для подтверждения или автоматически сохранить.

Всякий раз, когда создается новая задача, она оказывается в статусе “New” состояние Pending (диаграмма переходов на рисунке 3-12). Этот статус подразумевает проработку спецификации и ожидаемого результате после завершения задачи. Основываясь на этих данных, будет происходить разработка и тестирование. Когда этот этап завершен, происходит переход в разработку (Development). Здесь программисты решают поставленную задачу и пишут автоматические тесты. По окончанию это работы задача переходит в тестирование (Testing), где решение, представленное разработчиками, тестируется в соответствии с требованиями, описанными на этапе “New”. Если тестирование прошло успешно, задача закрывается.

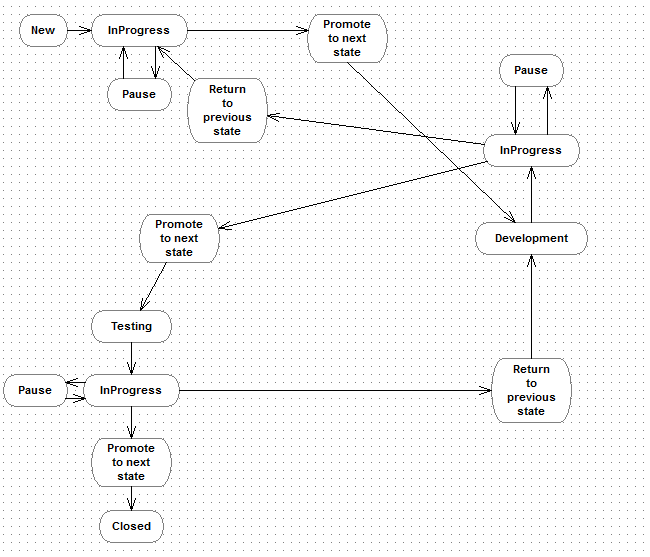


Рисунок 3-12: диаграмма переходов для задач проекта

На каждом описанном выше этапе у пользователя есть возможность приостановить выполнение задачи или вернуть ее на предыдущий этап.

Все эти действия обеспечиваются методами ядра PromoteTaskToNextStatus и ChangeTaskState. Так, при переходе задачи в новое состояние (например пауза) метод ChangeTaskState прекращает подсчет времени, затраченного на реализацию задачи. Это сделано, чтобы более точно учитывать время, затраченное на ее решение.

В ядре так же существует несколько событий, подписавшись на которые пользователь может переопределять подсчет рисков в своей системе.

ChangeStateEvent – событие возникает, когда пользователь меняет состояние задачи. Аргументы обработчика содержат информацию о новом состоянии задачи и о самой задаче (рисунок 3-13).

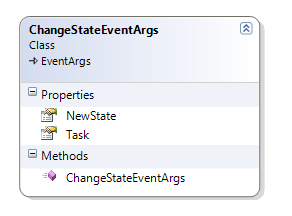


Рисунок 3-13: аргументы обработчика ChangeStateEventHandler

ChangeStatusEvent – событие возникает, когда пользователь меняет статус задачи. Аргументы обработчика содержат информацию о новом статусе задачи и о самой задаче (рисунок 3-14).

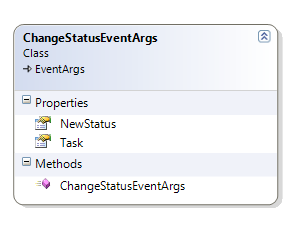


Рисунок 3-14: аргументы обработчика ChangeStatusEventHandler

SprintCompletedEvent – событие возникает, когда все задачи активного спринта оказываются закрытыми. Аргументы обработчика содержат информацию о спринте (рисунок 3-15).

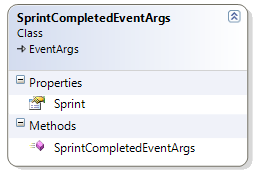


Рисунок 3-15: аргументы обработчика SprintCompletedEventHandler

* + 1. Взаимодействие ядра с базой данных

Рассмотрим пример частной реализации IDataProvider, который используется в Loop (рисунок 3-16).

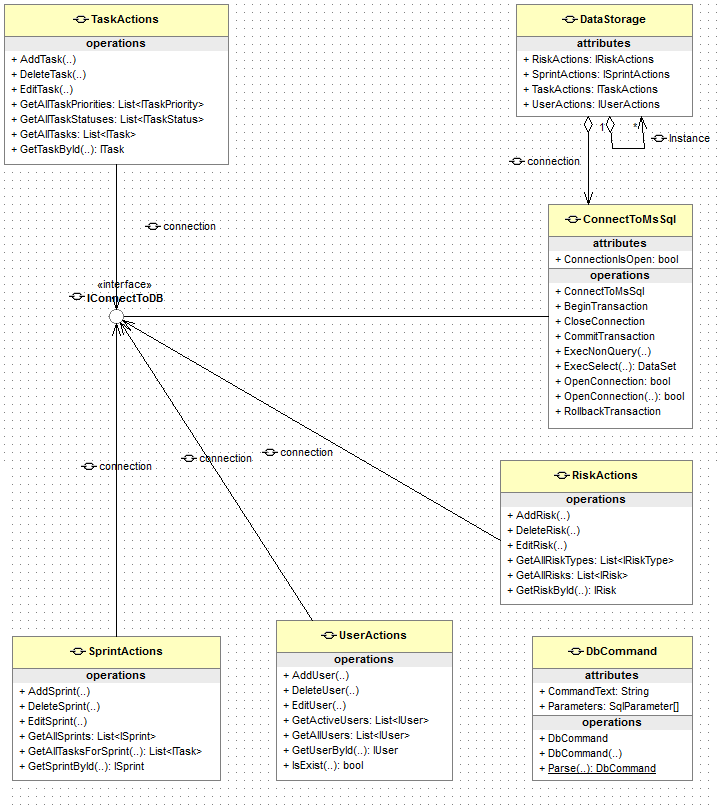


Рисунок 3-16: реализация интерфейсов для взаимодействия с базой данных

Здесь существует интерфейс, представляющий соединение с базой данных - IConnectToDB. Этот интерфейс – абстракция надо поставщиком данных. В данном случае существует одна реализация этого интерфейса – ConnectToMsSql. Эта реализация предоставляет методы для быстрого и удобного выполнения обращений в базу данных, а так же выполнения операций с ней. Рассмотрим их подробнее.

OpenConnection – открывает соединение с базой данных, прописанное в конфигурационном файле приложения (по названия строки соединения “Loop”). Существует также перегруженная вервия этого метода, которая принимает строку соединения в качестве параметра.

CloseConnection – закрывает соединение, в случае, если оно на данный момент открыто. Открыто ли соединение можно определить по флагу ConnectionIsOpen.

BeginTransaction – открывает новую транзакцию в базе данных, в рамках которой будут выполняться запросы к базе. Механизм транзакций необходим в случае, если нужно выполнить несколько взаимосвязанных запросов, так, что в случае неудачного выполнения одного из них, нужно откатить все.

CommitTransaction – закрывает предварительно открытую транзакцию.

RollbackTransaction – откатывает последнюю открытую транзакцию и делает ее неактивной.

ExecNonQuery – выполняет запрос к базе данных в соответствии с инструкциями, переданными в качестве параметра метода. Инструкции содержатся в класса DbCommand, который служит для их описания.

ExecSelect – функция, как и ExecNonQuery, выполняет запрос к базе данных, но в данном случае предполагается возвращаемое значение – DataSet.

Теперь, когда у нас есть реализация подключения к базе данных, рассмотрим, как она используется в реализации IDataStorage.

Как уже было сказано выше, в IDataStorage агрегирует еще 4 интерфейса ITaskActions, ISprintActions, IUserActions и IRiskActions. Они реализованы классами TaskActions, SprintActions, UserActions и RiskActions соответсвенно, в них нет ничего примечательного. А вот IDataStorage рассмотрим внимательнее.

При инициализации DataStorage (класса, реализующего IDataStorage), инициализируется соединение с базой данных (ConnectionToMsSql), которое передается в конструкторов всех агрегируемых DataStorage классов. Этот класс также реализует шаблон проектирования Singleton так, что экземпляр этого класса создается единожды и используется везде в приложении.

1. Обеспечение безопасных условий труда на предприятии «Скэнд».

Целью дипломного проекта является разработать инструмент (проект под названием «Loop»), который поможет менеджерам на предприятиях в распределении ресурсов и выставлении оценок трудозатрат при разработке программного продукта. Этот инструмент предоставит менеджерам простой и удобный способ распределить задачи между разработчиками, учитывая индивидуальные особенности каждого. Это обеспечится путем отслеживания статистических данных о рисках, реализующихся на каждой небольшой итерации разработки, и «стиле» написания кода разработчиками. Так же «Loop» строит графики, отображающие прогнозы по звершению итераций и динамику разработки на каждой итерации. В настоящем разделе рассмотрим вопрос связанный с обеспечением безопасных условий труда при создании вышеуказанной системы учета метрик.

При производстве программного обеспечения на ООО «СКЭНД» инженер-программист проводит большую часть рабочего времени находясь на своем рабочем месте за столом, на котором расположен ПК. Для успешного функционирования системы «человек-машина» при работе обеспечивается пять следующих видов совместимостей: информационная, биофизическая, энергетическая, пространственно-антропометрическая и технико-эстетическая.

В повседневной работе инженер-программист постоянно изучает огромное количество информации необходимой для осуществления текущих задач. Информационная совместимость обеспечена созданной информационной моделью устройства (ПК) – средств отображения информации (СОИ) и сенсомоторных устройств (органы управления – клавиатуры, мышки, кнопки, выключатели и др.), которая отражает все нужные характеристики машин в данный момент и позволяет разработчику безошибочно принимать и перерабатывать информацию в соответствии с его психофизиологическими возможностями (особенности внимания и памяти).

На предприятии ООО «СКЭНД» созданы комфортные условия труда: здание расположено вдали от улиц с оживленным движением, рядом нет дополнительных источников шума и вибрации, рабочее место разработчика расположено в просторном помещении площадью 30 м2 с тремя большими окнами, обеспечивающими хорошее естественное освещение, кроме того в помещении достаточный уровень искусственного освещения, есть система вентиляции и кондиционер. Данные условия труда соответствуют требованиям нормативных документов и обеспечивают приемлемую работоспособность и нормальное физиологическое состояние инженера-программиста.

Энергетическая совместимость обеспечена согласованием прилагаемых усилий, затрачиваемой мощности, скорости и точности движений органов управления оборудованием (ПК) с оптимальными возможностями инженера-программиста.

При проектировании рабочего места учтены размеры тела человека, его возможности обзора внешнего пространства, рабочего положения (позы) (определены зоны досягаемости для конечностей инженера-программиста, выбраны габариты рабочего стола (160x90 см.), подобран удобный эргономичный стул с регулируемой высотой, а также углом наклона спинки). На предприятии ООО «СКЭНД» предпринят широкий ряд мер по улучшения качества рабочего места инженера-программиста.

В работе инженера-программиста на ООО «СКЭНД» используются современные ПК с высокими показателями производительности (процессор Intel Core i5-2400 CPU 3.10GHz, оперативная память 8 Gb), широкоформатные мониторы (диагональ 24'', разрешение 1920x1080), а также только новейшее программное обеспечение (Windows 7 SP1, Visual Studio 2010, MS SQL Server R2), что повышает производительность труда.

Условия труда инженеров-программистов на ООО «СКЭНД» имеют частные и общие характеристики, обусловленные спецификой деятельности. Для выявления и оценки этих факторов проводится регулярное обследование рабочих мест, условий труда и профессиографический анализ деятельности инженера-программиста. На основе этих данных разработаны организационные, технические, оздоровительные мероприятия по обеспечению благоприятных эргономико-гигиенических условий труда и повышению его эффективности, по решению проблем надёжности работы инженера-программиста и сохранению его профессионального долголетия.

В соответствии с ГОСТ 12.1.003-83 «Шум. Общие требования к безопасности» уровень шума на рабочих местах операторов ВТ рекомендуют не более 50 дБА. Нормируемые уровни шума обеспечиваются путем использования малошумящего оборудования, применением звукопоглощающих материалов для облицовки помещений.

Нормальные микроклиматические условия в помещении достигнуты уменьшением тепловыделений оборудования, обеспечением соответствующей площади и объема помещения, устройством эффективных систем вентиляции и кондиционирования.

Рабочее место инженера-программиста занимает достаточную площадь – 6 м2. Между отдельными рабочими местами есть проход шириной 1,2 метра (рисунок 4-1). Данные параметры рабочего места обеспечивают безопасные условия труда в соответствии с гигиеническими требованиями к ВТ и ПК.

Рисунок 4-1: рабочее место инженера-программиста

В связи с повышенной нагрузкой на орган зрения инженера-программиста важное место среди мероприятий по гигиене их труда занимает работа, направленная на изучение оптимальной световой среды, то есть организация комфортного и гигиенического естественного и искусственного освещения рабочих мест и помещений.

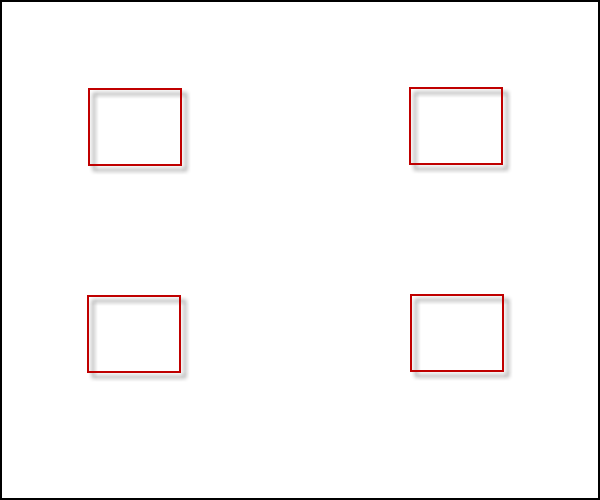
Для освещения помещения имеются светильники с энергосберегающими лампами до 100 Вт (рисунок 4-2). Общее количество светильников составляет 4 штуки. Окна помещения ориентированы на юго-запад. Освещенность используемого помещения общего назначения с использованием компьютеров составляет 250-300 лк. Кабинет оснащен системой кондиционирования и очистки воздуха, также имеется возможность открытия оконных рам для проветривания кабинетов. Отопление помещений осуществляется централизованно с использованием металлических радиаторов «Лидея». В отапливаемый период температура воздуха поддерживается не ниже 18оС. В помещениях используется напольное покрытие ламинат. Влажная уборка помещений производится ежедневно. Кабинет оборудован системой пожарного оповещения. Электрические розетки, используемые в помещении, имеют заземление. В кабинете висит инструкция по работе с компьютерами. Стены помещения окрашены в светлые зелено-голубые тона малой насыщенности. Потолки и самый верх стен выкрашены в белый цвет.

Рисунок 4-2: схема освещения офисного помещения

При организации рабочего места учитывались антропометрические данные инженера-программиста, а также размещение элементов оборудования соответственно характеру и последовательности выполняемой работы. Рабочий стол имеет стабильную конструкцию. Тип рабочего кресла был выбран с учетом предпочтений инженера-программиста и продолжительности работы: кресло с регулируемой высотой и уровнем наклона спинки, которое свободно отодвигается. Подножка кресла имеет пять опор, что исключает опрокидывание.

В соответствии с характером работы и суточным расходом энергии выработан и режим питания людей. Учитывая, что график работы инженера-программиста гибкий, перерыв на обед может быть в любое время во временном промежутке с 12.00 до 15.00 в течение 1 часа, для этого можно использовать кухню, расположенную в офисе, где есть все необходимое (холодильники, микроволновые печи, чайник, кофеварка, места для приема пищи и т.д. см. рисунки. 4-3 и 4-4) или посещать места общественного питания (столовые, кафе и т.д.). Отсутствие необходимой физической нагрузки в сочетании даже с умеренным питанием ведёт к полноте, а в последующем и к ожирению людей. Многие сотрудники в летний период выбирают велосипед, как средство передвижения, для них у входа в офис предусмотрена стоянка для велосипедов. Для всех сотрудников ООО «СКЭНД» предоставляются абонементы на посещения тренажерного зала и бассейна, расположенного в здании офиса.

Использование на практике психологических особенностей и закономерностей способствует значительному повышению безопасности трудовой деятельности.

Эффективность деятельности или работоспособность базируется на уровне психического напряжения (стресса). Длительные психические напряжения и особенно их запредельные формы ведут к выраженным состояниям утомления.

Прием легких стимуляторов (чай, кофе) способствует повышению работоспособности, для этого на предприятии ООО «СКЭНД» оборудовано отдельное помещение (кухня).

Рисунок 4-3: кухня ООО «СКЭНД»

**Рисунок 4-4: кухня ООО «СКЭНД»**

А также, есть комнаты психологической разгрузки, помещения для игр, спортивные площадки на открытом воздухе, где можно проводить перерывы для отдыха (рисунок 4-5). У каждого сотрудника есть памятка с рекомендациями по гимнастике для глаз. Все это эффективно влияет на снятие психического и физического напряжения.

Контроль психического состояния персонала ответственных работ и принятие административных мер на предприятии ООО «СКЭНД» положительно влияет на сокращение травматизма и повышение надежности работ сложных систем. Выполняются требования к производственной среде, рабочей мебели и организации труда для профилактики психофизиологических перегрузок предусмотрена комната отдыха для персонала, а также кухня, где можно полноценно принять пищу и посмотреть телевизор.

Рисунок 4-5: помещение для отдыха ООО «СКЭНД»

Таким образом, изложенные выше условия труда обеспечивают эффективную и устойчивую работу разработчиков. Стимулируют создание инструментов, рассматривающих роль человеческого фактора в современных производственных процессах, системах и структурах.

1. Технико-экономическое обоснование инвестиционного проекта по разработке программного обеспечения “Система учета метрик и распределения ресурсов при Agile разработке”

Разработка программного продукта (ПП) связана с различными затратами ресурсов (трудовых, материальных и финансовых). В связи с этим необходимо привести как техническое, так и экономическое обоснование экономических параметров будущего проекта. Целью данной работы является разработка ПП, упрощающего работу с задачами во время разработки программного обеспечения и позволяющего отслеживать статистику осуществления рисков. Функциональные возможности позволят учитывать собранную статистику при последующем распределении задач между разработчиками.

* 1. Расчет затрат при создании ПО.

Расчет основной заработной платы исполнителей, занятых разработкой, проводятся на основе исходных данных, представленных в таблице 5-1.

|  |  |  |  |
| --- | --- | --- | --- |
| Наименование статей | Условные обозначения | Единицы | Норматив |
| 1 | 2 | 3 | 4 |
| Коэффициент новизны | Кн | ед. | 1,0 |
| Категория сложности |  | ед. | 1 |
| Дополнительный коэффициент сложности | Кс | ед. | 1,12 |
| Коэффициент, учитывающий степень использования при разработке ПО стандартных модулей | Кт | ед. | 0,7 |
| Установленная плановая продолжительность разработки | Тр | Лет | 0,2 |
| Годовой эффективный фонд времени | Фэф | Дней | 236 |
| Продолжительность рабочего дня | Тч | ч. | 8 |
| Тарифная ставка 1-го разряда | Тм1 | руб. | 600000 |
| Коэффициент премирования | Кп | ед. | 1,5 |
| Дополнительная заработная плата исполнителей | Нд | % | 20 |
| Ставка отчислений в фонд социальной защиты населения и на обязательное страхование | Нсз | % | 34,5 |
| Расходы на научные командировки | Нрнк | % | 30 |
| Прочие прямые расходы | Нпз | % | 20 |
| Накладные расходы | Нрн | % | 100 |
| Прогнозируемая прибыль | Урпi | % | 20 |
| Ставка НДС (при отсутствии льгот) | НДС | % | 20 |
| Налог на прибыль при отсутствии льгот | Нп | % | 18 |
| Норма расходов материалов | Нм | руб. | 380 |
| Норма расхода машинного времени | Нмв | маш-ч | 12 |
| Цена одного машино-часа | Цмi | руб. | 3000 |
| Норматив расходов на освоение ПО | Но | % | 10 |
| Норматив расходов на сопровождение ПО | Нс | % | 20 |

Таблица 5-1: исходные данные

На основе информации о функциях разрабатываемого ПО, используя нормативные данные, определяется объем функций и общий объем ПО (Таблица 5-2).

|  |  |  |
| --- | --- | --- |
| N функции | Наименование (содержание) | Объем функции (LOC) |
| 109 | Организация ввода/вывода информации в интерактивном режиме | 1397 |
| 206 | Обслуживание базы данных в интерактивном режиме | 1206 |
| 204 | Обработка наборов и записей базы данных | 425 |
| 207 | Манипулирование данными | 4325 |
| 208 | Организация поиска и поиск в базе данных | 384 |
| 506 | Обработка ошибочных и сбойных ситуаций | 145 |
| 507 | Обеспечение интерфейса между компонентами | 157 |
| 604 | Справка и обучение | 843 |
| 707 | Графический вывод результатов | 2358 |
| 605 | Вспомогательные и сервисные программы | 3786 |
|  | Итого | 15026 |

Таблица 5-2: Характеристика функций и их объем

Общий объём () ПО определяется исходя из количества и объёма функций, реализуемых программой и рассчитывается по формуле:



где – объём отдельной функций ПО; n – общее число функций.

В формуле для расчёта общего объёма ПО используем данные, приведенные в таблице 5.2.



На основании принятого к расчету объема и категории сложности определяется нормативная трудоемкость ПО.

(чел/дн)

На основании нормативной трудоемкости рассчитывается общая трудоемкость с учетом распределения ее по стадиям (:

,

где − трудоемкость разработки ПО на i-й стадии (чел./дн.); n − количество стадий разработки.

Трудоемкость стадий определяется на основе нормативной трудоемкости с учетом сложности, новизны, степени использования в разработке стандартных модулей ПО и удельного веса трудоемкости каждой стадий в общей трудоемкости ПО:

,

где −уточнённая трудоемкость разработки ПО на i-й стадии;

 − удельный вес трудоемкости i-й стадии разработки ПО в общей  
трудоемкости разработки ПО;

− коэффициент, учитывающий сложность ПО, вводится на всех стадиях;

 − коэффициент, учитывающий степень использования стандартных модулей ПО, вводится только на стадии рабочего проекта;

− коэффициент, учитывающий степень новизны ПО, вводится на всех стадиях.

Расчеты проводятся для всех стадий разработки ПО, результаты приведены в таблице 5-3.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Стадии | | | | | Итого |
| ТЗ | ЭП | ТП | РП | ВН |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 |
| Коэффициенты удельных весов трудоемкости стадии разработки () | 0,11 | 0,09 | 0,11 | 0,55 | 0,14 | 1,0 |
| Коэффициент сложности ПО | 1,12 | 1,12 | 1,12 | 1,12 | 1,12 | 1,12 |
| Коэффициент, учитывающий использование стандартных модулей () | - | - | - | 0,7 | - | - |
| Коэффициент новизны () | 1 | 1 | 1 | 1 | 1 | - |
| Распределение нормативной трудоемкости ПО () по стадиям, чел./дн | 68 | 56 | 68 | 240 | 87 | 519 |

Таблица 5-3 – Расчет общей трудоемкости разработки программного модуля и численности исполнителей с учетом стадий

Уточненная трудоемкость разработки программного модуля по всем стадиям составит 519 человеко-дня.

На основе общей трудоёмкости определяются плановое число разработчиков () и плановые сроки, необходимые для реализации проекта в целом (). При этом могут решаться следующие задачи:

расчет числа исполнителей при заданных сроках разработки проекта;

определение сроков разработки проекта при заданной численности исполнителей.

Численность исполнителей проекта рассчитывается по формуле:

,

где – общая трудоёмкость разработки проекта (чел/дн);  – срок разработки проекта.

Эффективный фонд времени одного работника рассчитывается по формуле:



Где  – количество дней в году;  – количество праздничных дней в году;  – количество выходных дней в году;  – количество дней отпуска.

Срок разработки проекта составляет 6 месяцев (0,5 года);

 дней.

(чел).

В расчете на 6 месяцев в среднем будет занято примерно 5 человек. Коэффициент премирования 1,5. В соответствии со штатным расписанием на разработке будут заняты:

1) четыре инженера-программиста второй категории (13 разряд; тарифный коэффициент – 3.04; продолжительность участия в разработке ПО - 72 дней).

2) один инженер-программист второй категории (13 разряд; тарифный коэффициент – 3.04; продолжительность участия в разработке ПО - 72 дней).

Месячная тарифная ставка каждого исполнителя () определяется путем умножения действующей месячной тарифной ставки 1-го разряда () на тарифный коэффициент (), соответствующий установленному тарифному разряду:

,

Часовая тарифная ставка рассчитывается путем деления месячной тарифной ставки на установленную при 40-часовой недельной норме рабочего времени расчетную среднемесячную норму рабочего времени в часах – 170 часов:

,

где  − часовая тарифная ставка (д.е.);  − месячная тарифная ставка (д.е.).

Определяются месячные и часовые тарифные ставки специалистов: ведущего инженера-программиста (,), трех инженеров-программистов 2-й категории (,).

(руб),

(руб),

(руб),

(руб).

На основе полученных выше результатов и исходных данных рассчитывается сумма основной заработной платы () всех исполнителей по формуле:

,

где  – количество исполнителей, занятых разработкой ПО;  – часовая тарифная ставка i-го исполнителя (руб);  – эффективный фонд рабочего времени i-го исполнителя (дней);  – количество часов работы в день (ч);  – коэффициент премирования.

 (руб)

Дополнительная заработная плата () определяется по формуле:

,

где  - норматив дополнительной заработной платы (исходя из исходных данных, ).

(руб).

Отчисление в фонд социальной защиты населения и на обязательное страхование () рассчитываются в соответствии с действующими законодательными актами по нормативу в процентном отношении к фонду основной и дополнительной зарплаты исполнителей, определенной по нормативу, в целом по организации:

,

где  – норматив отчислений в фонд социальной защиты населения(%).

Согласно исходным данным,  равен 34%, с учетом отчислений на обязательное страхование 34,5%.

 (руб).

Расходы по статье «Материалы»  определяются на основании сметы затрат, разрабатываемой на ПО с учетом действующих нормативов. По статье «Материалы» отражаются расходы на магнитные носители, бумагу, красящие ленты и другие материалы, необходимые для разработки ПО. Нормы расходов материалов определяются в расчете на 100 строк исходного кода. Сумма затрат на расходные материалы рассчитывается по формуле:

,

где  – норма расхода материалов в расчёте на 100 команд ПО (= 380 руб).

(руб).

Расходы по статье “Машинное время”  включают оплату машинного времени, необходимого для разработки и отладки ПО:

,

где  – цена одного машино-часа ( = 3000 руб.);

 – норматив расхода машинного времени на отладку 100 машинных команд (машино-часов). Для задач расчётного характера = 12.

(руб).

Затраты по статье «Накладные расходы»  рассчитываются по формуле:

,

где  – норматив накладных расходов в целом по организации.

 (руб).

Общая сумма расходов по всем статьям сметы  на ПО рассчитывается по формуле:

,



Рентабельность и прибыль по создаваемому ПО () определяется исходя из результатов анализа рыночных условий, переговоров с заказчиком (потребителем) и согласования с ним отпускной цены, включающей дополнительно налог на добавленную стоимость. Прибыль рассчитывается по формуле:

,

где  – прибыль от реализации ПО заказчику (руб);  – уровень рентабельности(%);  – себестоимость ПО (руб).

(руб).

Прогнозируемая цена ПО без налогов, включаемых в цену:ы



(руб).

С учетом того, что разработанный пакет составляет 1/10 часть системы, прогнозируемая цена ПО составит:

(руб).

Налог на добавленную стоимость (НДС):



(руб).

Расчет прогнозируемой отпускной цены на разрабатываемое ПО:



(руб).

Организация разработчик участвует в освоении ПО и несет соответствующие затраты, на которые составляется смета, оплачиваемая заказчиком по договору. Сметой предусматриваются не только затраты, но и налоги, предусмотренные законодательством, и прибыль организации-разработчика. Для упрощения расчетов до составления сметы затраты на освоение определяются по нормативу (НО = 10%) от себестоимости ПО в расчете на 3 месяца и рассчитываются по формуле:



где  – норматив расходов на освоение (%).

(руб).

С учетом того, что разработанный пакет составляет 1/10 часть системы:

(руб).

Затраты на сопровождение ПО (РС).

Организация сдатчик осуществляет сопровождение ПО и несет соответствующие расходы, которые оплачиваются заказчиком в соответствии с договором и сметой на сопровождение. Смета составляется по аналогии со сметой на освоение ПО. Для упрощения расчетов для составления сметы затраты на сопровождение определяются по установленному нормативу (НС = 20%) от себестоимости ПО и рассчитываются по формуле:



где НС – норматив расходов на сопровождение (%).

(руб).

С учетом того, что разработанный пакет составляет 1/10 часть системы:

(руб).

* 1. Расчет экономической эффективности от разработки ПО

Для определения экономического эффекта от использования нового ПО у потребителя необходимо сравнить расходы по всем основным статьям сметы затрат на эксплуатацию нового ПО (расходы на заработную плату с начислениями, затраты на расходные материалы, расходы на машинное время) с расходами по соответствующим статьям базового варианта. При этом за базовый вариант следует принимать аналогичное программное средство, используемое в действующей автоматизированной системе, или ручной вариант, если автоматизация отсутствует. При сравнении базового и нового вариантов ПО в качестве экономического эффекта будет выступать общая экономия всех видов ресурсов относительно базового варианта. Создание нового ПО окажется экономически целесообразным лишь в том случае, если все капитальные затраты окупятся за счет получаемой экономии в ближайшее время (до 2 лет).

Исходные данные для расчета экономии ресурсов в связи с применением новой системы приведены в таблице 5-4

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Наименование показателей | Обозначение | Единица измерения | Значение показателя | |
| В базовом варианте | В новом варианте |
| 1 | 2 | 3 | 4 | 5 |
| 1.Капитальные вложения, включая затраты пользователя на приобретение ПО | Кпр | Млн руб. | - | 304 |
| 2.Затраты на освоение ПО | Кос | млн руб. | - | 21 |
| 3.Затраты на сопро­вождение ПО | Кс | млн руб. | - | 42 |
| 4.Затраты на доукомплектование ВТ техническими средствами в связи с внедрением нового ПО | Ктс | Млн руб. | - | 0 |
| 5. Затраты на пополнение оборотных средств в связи с эксплуатацией нового ПО | Коб | Млн руб. | - | 0 |
| 6. Время простоя сервиса, обусловленное ПО, в день | П1, П2 | мин | 10 | 0 |
| 7. Стоимость одного часа простоя | Сп | руб. | 5000 | 5 000 |
| 8. Среднемесячная ЗП одного программиста | Зсм | руб. | - | 600 000 |
| 9.Коэффициент начислений на зарплату | Кнз |  | 1,1 | 1,1 |
| 10.Среднемесячное количество рабочих дней | Др | день | - | 21 |
| 11. Количество типовых задач, решаемых за год | Зт1, Зт2 | задача | 3000 | 3000 |
| 12. Объем выполняемых работ | А1, А2 | задача | 3000 | 3000 |
| 13. Средняя трудоемкость работ в расчете на 1 задачу | Тс1, Тс2 | человеко-часов на 1 задачу | 3 | 0,5 |
| 14. Количество часов работы в день | Тч | ч | 8 | 8 |
| 15. Ставка налога на прибыль | Нп | % | 24 | 18 |

Таблица 5-4 – Исходные данные для расчета экономии ресурсов в связи с применением нового программного модуля

Общие капитальные вложения (Ко) заказчика (потребителя), связанные с приобретением, внедрением и использованием системы, рассчитываются по формуле:



(млн руб).

Экономия затрат на заработную плату в расчете на 1 задачу ():

,

где  − среднемесячная заработная плата одного программиста (руб.);

,  − снижение трудоемкости работ в расчете на 1 задачу (человеко-часов);

 − количество часов работы в день (ч);

 − среднемесячное количество рабочих дней.

(руб).

Экономия заработной платы при использовании нового программного модуля (руб):



Где  − экономия заработной платы;  − количество типовых задач, решаемых за год (задач).

(руб).

Экономия с учетом начисления на зарплату ():

(руб).

Экономия за счет сокращения простоев сервиса () рассчитывается по формуле:



где  – плановый фонд работы сервиса (дней).

(руб).

Общая готовая экономия текущих затрат, связанных с использованием нового программного модуля (), рассчитывается по формуле:

,

(руб).

С учетом того, что разработанный пакет составляет 1/10 часть системы:

(руб).

Внедрение новой системы позволит пользователю сэкономить на текущих затратах, т.е. практически получить на эту сумму дополнительную прибыль. Для пользователя в качестве экономического эффекта выступает лишь чистая прибыль − дополнительная прибыль, остающаяся в его распоряжении (), которая определяется по формуле:



где  − ставка налога на прибыль (%).

(руб).

В процессе использования нового ПО чистая прибыль в конечном итоге возмещает капитальные затраты. Однако полученные при этом суммы результатов (прибыли) и затрат (капитальных вложений) по годам приводят к единому времени − расчетному году (за расчетный год принят 2013-й год) путем умножения результатов и затрат за каждый год на коэффициент дисконтирования α, который рассчитывается по формуле:

,

Где Е – норматив приведения разновременных затрат и результатов, равный 30%; tp− расчетный период; t − период, потоки которого приводятся к расчётному.

Коэффициент дисконтирования равен:

Для 2013 года = 1,

для 2014 года = 0,769,

для 2015 года = 0,591,

для 2016 года = 0,455.

Все рассчитанные данные экономического эффекта приведены в таблице 5-5.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Показатели | Ед. измерения | 2013 | 2014 | 2015 | 2016 |
| 1 | 2 | 3 | 4 | 5 | 6 |
| Результаты: |  |  |  |  |  |
| Прирост прибыли за счет экономии затрат (Пч) | млн руб. | 272 | 272 | 272 | 272 |
| То же с учетом фактора времени | млн руб. | 272 | 209.2 | 160.75 | 123.76 |
| Затраты: |  |  |  |  |  |
| Приобретение (Кпр) | млн руб. | 304 | - | - | - |
| Освоение ПО (Кос) | млн руб. | 21 | - | - | - |
| Сопровождение (Кс) | млн руб. | 42 | - | - | - |
| Доукомплектование ВТ техническими средствами (Ктс) | млн руб. | 0 | - | - | - |
| Пополнение оборотных средств (Коб) | млн руб. | 0 | - | - | - |
| Всего затрат: | млн руб. | 367 | - | - | - |
| То же с учетом фактора времени | млн руб. | 367 | - | - | - |
| Экономический эффект |  |  |  |  |  |
| Превышение результат над затратами | млн руб. | -367 | 209.2 | 160.75 | 123.76 |
| То же с нарастающим итогом | млн руб. | - | -157.8 | 2.95 | 126.71 |
| Коэффициент приведения | Единиц | 1 | 0,881 | 0,7762 | 0,6839 |

Таблица 5-5: Расчет экономического эффекта от использования нового программного модуля

В таком случае рентабельность продукта равна:

,

.

По данным таблицы срок окупаемости проекта составит 2 года.

Заключение

В этом дипломном проекте была реализована система, позволяющая упростить работу менеджеров программных проектов отслеживать динамику разработки. Эта система, позволяет быстро и легко экспортировать данные из багтрекинговой системы. Из этих данных система может выбрать задачи, решение которых, может уложиться во временные рамки спринта. Получив список задач, которые должны быть решены в рамках этого спринта, система предоставляет функционал, который оптимально распределяет эти задачи между разработчиками.

Во время выполнения задач, учитываются статистические данные разработчиков, специфик проекта и задач. Эти данные – риски проекта. Учет рисков позволяет более точно прогнозировать вероятность, что проект завершится вовремя. А так же используются при распределении задач между разработчиками.

На любом этапе разработке, может быть сгенерирован график сгорания, который показывает, каков объем работ был выполнен в сравнении с ожидаемым. Эти графики помогают обеспечивать обратную связь для заказчиков.

Так же предусмотрен механизм экспорта задач проекта в различные системы отчетности. Это позволяет сэкономить время на формировании отчетов, их заполнение и обновление вручную.

Таким образом система предоставляет механизмы упрощающие работу менеджеров по управлению задачами, их распределению и формированию отчетности. А так же предоставляет возможность формировать и визуализировать статус разработки проекта, что обеспечивает более эффективную и легкую работу с заказчиками.

Список источников

1. [Ado.net](http://citforum.ru/programming/csharp/ado_dot_net/)
2. [Mvc.net](http://msdn.microsoft.com/ru-ru/library/dd381412(v=vs.108).aspx)
3. [Платформа .net](http://bibliofond.ru/view.aspx?id=447431)
4. [Принципы XP](http://www.onembedding.com/59.html)
5. Х. Книберг: «Scrum и XP: заметки с передовой», «Scrum VS Kanban», 2011
6. A. Коуберн: «Парное программирование- преимущества и недостатки», 2009
7. Т. Демарко: «Deadline. Роман об управлении проектами», «Вальсируя с медведями: управление рисками при разработке ПО», 2006
8. Ф. Брукс: «Мифический человеко-месяц», 1995
9. Дж. Рейнвотер: «Как пасти котов. Наставление для программистов, руководящих другими программистами», 2011
10. А. Каптерев: «Книга о тактике. Интуиция и принятие решения», 2010
11. Т.Ф. Михнюк. «Охрана труда» Минск «ИВЦ Минфина», 2007.
12. В.П. Семич «Охрана труда при работе на персональных электронно-вычислительных машинах и другой офисной технике: Практ. пособие. – Мн.: Высш. шк.», 2001.
13. Н. Сибаров и др. «Охрана труда в вычислительных центрах. – М.: Машиностроение», 1990.

Приложение А: реализация соединения с базой данных

namespace Loop.DatabaseConnection

{

public interface IConnectToDB

{

bool OpenConnection();

bool OpenConnection(String connectionString);

void CloseConnection();

void ExecNonQuery(DbCommand args);

DataSet ExecSelect(DbCommand args);

void BeginTransaction();

void CommitTransaction();

void RollbackTransaction();

bool ConnectionIsOpen { get; }

}

}

namespace Loop.DatabaseConnection

{

public class ConnectToMsSql : IConnectToDB

{

protected bool transactionIsActive;

protected bool connectionIsOpen;

protected SqlTransaction transaction;

protected SqlConnection connection;

public bool ConnectionIsOpen

{

get

{

return connectionIsOpen;

}

}

public ConnectToMsSql()

{

transactionIsActive = false;

connectionIsOpen = false;

}

public bool OpenConnection()

{

if (!(connectionIsOpen))

{

connection = new SqlConnection(ConfigurationManager.ConnectionStrings["Loop"].ConnectionString);

connection.Open();

connectionIsOpen = true;

}

else

{

Logger.Instance.WriteToLog("Warning: connection is already opened.");

}

return connectionIsOpen;

}

public bool OpenConnection(String newConnectionString)

{

if (!(connectionIsOpen))

{

connection = new SqlConnection(newConnectionString);

connection.Open();

connectionIsOpen = true;

}

else

{

Logger.Instance.WriteToLog("Warning: connection is already opened.");

}

return connectionIsOpen;

}

public void CloseConnection()

{

if (connectionIsOpen)

{

if (transactionIsActive)

{

RollbackTransaction();

}

connection.Close();

connectionIsOpen = false;

}

else

{

Logger.Instance.WriteToLog("Error: there is no open connection."); ;

}

}

public void ExecNonQuery(DbCommand args)

{

if (connectionIsOpen)

{

if ((args != null) && (args.CommandText != String.Empty))

{

SqlCommand command = null;

if (!transactionIsActive)

{

command = new SqlCommand(args.CommandText, connection);

}

else

{

command = new SqlCommand(args.CommandText, connection, transaction);

}

if (args.Parameters != null)

{

command.Parameters.AddRange(args.Parameters);

}

command.ExecuteNonQuery();

}

else

{

Logger.Instance.WriteToLog("Error in command arguments.");

}

}

else

{

Logger.Instance.WriteToLog("Error: connection was not opened.");

}

}

public DataSet ExecSelect(DbCommand args)

{

if (connectionIsOpen)

{

if ((args != null) && (args.CommandText != String.Empty))

{

SqlCommand command = null;

if (!transactionIsActive)

{

command = new SqlCommand(args.CommandText, connection);

}

else

{

command = new SqlCommand(args.CommandText, connection, transaction);

}

if (args.Parameters != null)

{

command.Parameters.AddRange(args.Parameters);

}

var dataAdapter = new SqlDataAdapter(command);

var dataSet = new DataSet(connection.Database);

dataAdapter.Fill(dataSet);

return dataSet;

}

else

{

Logger.Instance.WriteToLog("Error in command arguments.");

}

}

else

{

Logger.Instance.WriteToLog("Error: connection was not opened.");

}

return null;

}

public void BeginTransaction()

{

if (connectionIsOpen)

{

if (!transactionIsActive)

{

transaction = connection.BeginTransaction(IsolationLevel.ReadCommitted);

transactionIsActive = true;

}

else

{

Logger.Instance.WriteToLog("Error: transaction is already began.");

}

}

else

{

Logger.Instance.WriteToLog("Error: connection was not opened.");

}

}

public void CommitTransaction()

{

if (connectionIsOpen)

{

if (transactionIsActive)

{

transaction.Commit();

transactionIsActive = false;

}

else

{

Logger.Instance.WriteToLog("Error: there is no active transaction.");

}

}

else

{

Logger.Instance.WriteToLog("Error: connection was not opened.");

}

}

public void RollbackTransaction()

{

if (connectionIsOpen)

{

if (transactionIsActive)

{

transaction.Rollback();

transactionIsActive = false;

}

else

{

Logger.Instance.WriteToLog("Error: there is no active transaction.");

}

}

else

{

Logger.Instance.WriteToLog("Error: connection was not opened.");

}

}

}

}

Приложение Б: подключение ядра приложения

namespace LoopUI.Controllers

{

//Class provides all functions that are needed for LoopUI

internal class CoreWrapper

{

private Core core;

private static CoreWrapper instance = new CoreWrapper(new Core(DataStorage.Instance));

private CoreWrapper()

{ }

private CoreWrapper(Core c)

{

core = c;

}

internal static CoreWrapper Instance

{

get

{

return instance;

}

}

/// <summary>

/// returns List of all risks

/// </summary>

/// <returns></returns>

internal List<IRisk> GetAllRisks()

{

return DataStorage.Instance.RiskActions.GetAllRisks();

}

/// <summary>

/// return List of all tasks

/// </summary>

/// <returns></returns>

internal List<ITask> GetAllTasks()

{

return DataStorage.Instance.TaskActions.GetAllTasks();

}

/// <summary>

/// return List of all users

/// </summary>

/// <returns></returns>

internal List<IUser> GetAllUsers()

{

return DataStorage.Instance.UserActions.GetAllUsers();

}

/// <summary>

/// return List of all sprints

/// </summary>

/// <returns></returns>

internal List<ISprint> GetAllSprints()

{

return DataStorage.Instance.SprintActions.GetAllSprints();

}

internal SelectList GetTaskPriorityList()

{

List<SelectListItem> result = new List<SelectListItem>();

List<ITaskPriority> list = DataStorage.Instance.TaskActions.GetAllTaskPriorities();

foreach (ITaskPriority t in list)

{

result.Add(new SelectListItem() { Text = t.Title, Value = t.Id.ToString() });

}

return new SelectList(result, "Value", "Text");

}

internal SelectList GetTaskStatusList()

{

List<SelectListItem> result = new List<SelectListItem>();

List<ITaskStatus> list = DataStorage.Instance.TaskActions.GetAllTaskStatuses();

foreach (ITaskStatus t in list)

{

result.Add(new SelectListItem() { Text = t.Title, Value = t.Id.ToString() });

}

return new SelectList(result, "Value", "Text");

}

internal SelectList GetActiveUsers()

{

List<SelectListItem> result = new List<SelectListItem>();

List<IUser> list = DataStorage.Instance.UserActions.GetActiveUsers();

foreach (IUser t in list)

{

result.Add(new SelectListItem() { Text = t.Name +" "+t.Surname, Value = t.Id.ToString() });

}

return new SelectList(result, "Value", "Text");

}

internal SelectList GetAllRiskTypes()

{

List<SelectListItem> result = new List<SelectListItem>();

List<IRiskType> riskTypes = DataStorage.Instance.RiskActions.GetAllRiskTypes();

foreach (IRiskType t in riskTypes)

{

result.Add(new SelectListItem() { Text = t.Title , Value = t.Id.ToString() });

}

return new SelectList(result, "Value", "Text");

}

}

}