Оглавление

[**ПРАКТИЧЕСКАЯ РАБОТА № 1** 2](#_Toc503476829)

[Оценка качественных показателей программного средства 2](#_Toc503476830)

[**ЛАБОРАТОРНАЯ РАБОТА № 1** 6](#_Toc503476831)

[Система контроля версий 6](#_Toc503476832)

[**ЛАБОРАТОРНАЯ РАБОТА № 2** 74](#_Toc503476833)

[Issue Tracker на примере BitBucket 74](#_Toc503476834)

[**ПРАКТИЧЕСКАЯ РАБОТА № 2, 3** 85](#_Toc503476835)

[Расчёт параметров надёжности программного обеспечения по математической модели Шумана и Миллса 85](#_Toc503476836)

[**ПРАКТИЧЕСКАЯ РАБОТА № 4** 93](#_Toc503476837)

[Составление наборов тестовых данных для функционального тестирования. Стратегия «чёрного ящика» 93](#_Toc503476838)

[**ПРАКТИЧЕСКАЯ РАБОТА № 5** 99](#_Toc503476839)

[Составление наборов тестовых данных для структурного тестирования. Стратегия «белого ящика» 99](#_Toc503476840)

# **ПРАКТИЧЕСКАЯ РАБОТА № 1**

### Оценка качественных показателей программного средства

Методика оценки качественных показателей ПС основана на составлении метрики ПС. В практической работе необходимо выполнить следующее:

1. Выбрать показатели качества (не менее 5) и сформулировать их сущность. Каждый показатель должен быть существенным, т.е. должны быть ясны потенциальные выгоды его использования.

Показатели представить в виде таблицы.

Таблица 1

|  |  |  |  |
| --- | --- | --- | --- |
| Показатели качества | Сущность показателя | Экспертная оценка (вес) | Оценка, установленная экспериментом |
|  |  |  |  |

**Определение комплексных показателей качества**

**1 Функциональные возможности (Functionality)**

* Пригодность (Suitability)
* Правильность (Accuracy)
* Способность к взаимодействию (Interoperability)
* Согласованность (Compliance)
* Защищённость (Security)
* Соответствие стандартам и правилам проектирования

**2 Надёжность (Reliability)**

* Стабильность (Maturity)
* Устойчивость к ошибке (Fault tolerance)
* Восстанавливаемость (Recoverability)

**3 Практичность (Usability)**

* Понятность (Understandability)
* Обучаемость (Learnability)
* Простота использования (Operability)

**4 Эффективность (Efficiences)**

* Характер изменения во времени (Time behavior)
* Характер изменения ресурсов (Resource behavior)

**5 Сопровождаемость (Maintainability)**

* Анализируемость (Analysability)
* Изменяемость (Changeability)
* Устойчивость (Stability)
* Тестируемость (Testability)
* Удобство для анализа

**6 Мобильность (Portability)**

* Адаптируемость (Adaptability)
* Простота внедрения (Installability)
* Соответствие (Conformance)
* Взаимозаменяемость (Replaceabilily)

**Примечания:**

1. Взаимозаменяемость используется вместо совместимости для того, чтобы избежать возможной путаницы со способностью к взаимодействию.
2. Взаимозаменяемость с конкретным программным средством не предполагает, что данное средство заменимо рассматриваемым программным средством.
3. Взаимозаменяемость может включать атрибуты простоты внедрения и адаптируемости. Понятие было введено в качестве отдельной подхарактеристики из-за его важности.
4. Установить веса показателей .
5. Для каждого показателя установить конкретную численную оценку  от 0 до 1, исходя из следующего:

0 – свойство в ПС присутствует, но качество его неприемлемо;

0.5 - 1 – свойство в ПС присутствует и обладает приемлемым качеством;

1 – свойство в ПС присутствует и обладает очень высоким качеством. Возможно присвоение промежуточных значений в соответствии с мнением оценивающего лица относительно полезности того или иного свойства ПС.

1. Определить качество ПС как иерархическую взвешенную сумму весов отдельных показателей. 
2. Определить среднее значение оценки качества ПС по формуле:



1. Представить выходные данные:

* перечень всех показателей с оценкой 0 с указанием причин такой оценки;
* гистограмму, показывающую распределение показателей по интервалам оценок;
* какие дефекты ПС обнаружены в результате анализа показателей качества.

**Задание**

Выбрать показатели качества для:

ОС семейства Windows

1. XP

2. Windows Vista

3. Windows 7

4. Windows 8/8.1

5. Windows X

заполнить таблицу, проставить коэффициенты  и , рассчитать среднее значение оценки качества ПС, перечислить все показатели с оценкой 0 с указанием причин оценки, построить гистограмму и написать какие дефекты ПС обнаружены в результате. Работу представить в виде отчёта.

**Пример**

Рассмотрим оценку показателей качества на примере ОС Linux

|  |  |  |  |
| --- | --- | --- | --- |
| Показатели качества | Сущность показателя | Экспертная оценка (вес) | Оценка, установленная экспериментом |
| Функциональная возможность | Пригодность | 0,15 | 1 |
| Защищённость | 0,2 | 1 |
| Надёжность | Стабильность | 0,15 | 1 |
| Устойчивость к ошибке | 0,3 | 1 |
| Практичность | Простота использования | 0,05 | 0,5 |
| Сопровождаемость | Анализируемость | 0,05 | 0,7 |
| Тестируемость | 0,05 | 0,6 |
| Мобильность | Адаптируемость | 0,05 | 0,8 |
| Простота внедрения | 0 | 0,5 |



Установка и настройка ОС Linux – это сложная процедура, связанная, непростой разметкой жёсткого диска, повышенным вниманием во время установки; установкой драйверов устройств, настройкой программного обеспечения. Эти работы выполняются высококвалифицированным специалистом в IT-области.

# **ЛАБОРАТОРНАЯ РАБОТА № 1**

### Система контроля версий

**Цель работы**

Получить опыт практической работы с системой контроля версий на примере BitBucket и TortoiseGit.

**Введение**

Контроль версий подразумевает под собой комплекс методов, направленных на систематизацию изменений, вносимых разработчиками в программный продукт в процессе его разработки и сопровождения, сохранение целостности системы после изменений, предотвращение нежелательных и непредсказуемых эффектов. Также использование систем контроля версий позволяет сделать процесс внесения изменений более формальным.

Система управления версиями гарантирует, что каждый автор всегда работает с самой последней версией файла, а также исключает возможность случайной перезаписи любым из авторов работы своих коллег.

*BitBucket* – это сервис для разработчиков программного обеспечения, предоставляющий хостинг для проектов.

Одним из основных сервисов, предоставляемых BitBucket, является контроль версий. Доступные системы контроля версий – Git и Mercurial. Также доступны сервисы просмотра исходного кода, issue-трекинга, ведения wiki проекта и другие.

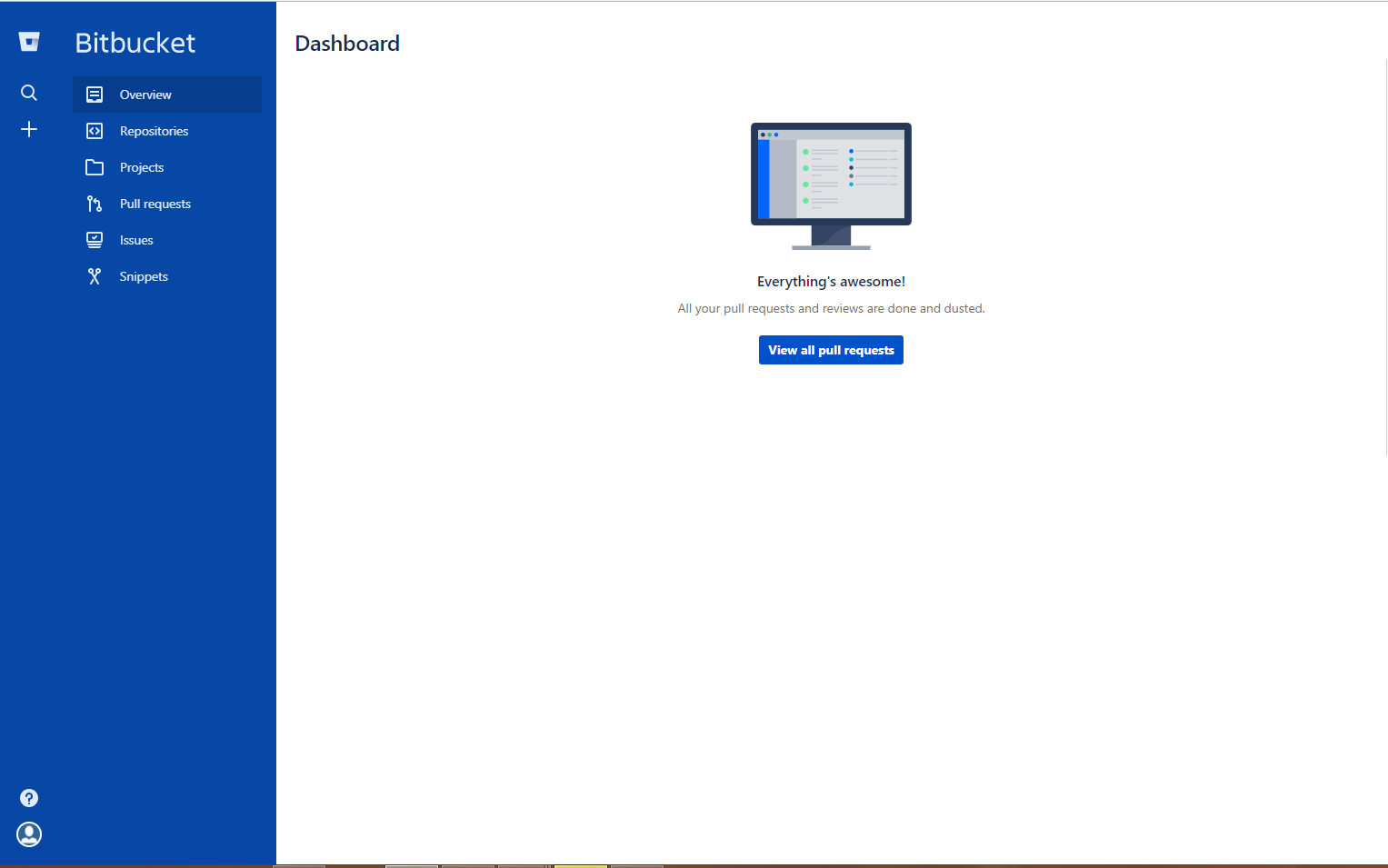
**Создание проекта на BitBucket**

1. Зайдите на сайт https://bitbucket.org. Авторизуйтесь под своим аккаунтом google, либо создайте новый аккаунт. При регистрации необходимо выбрать «5 users» в пункте «Plan».
2. Для создания частного репозитория необходимо нажать **Create → Create Repository** в верхнем правом углу на главной странице (рис. 1.1). Однако в данной лабораторной работе рекомендуется сначала создать команду через пункт меню **Teams → Create team**. На данном этапе вы можете не добавлять никого в команду (это понадобится для выполнения лаб. раб. №2). После создания команды вам будет предложено создать репозиторий.

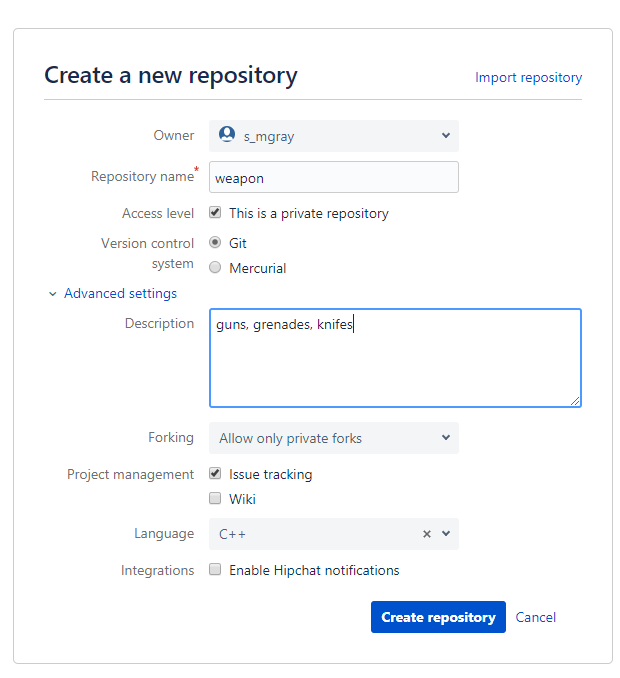
В открывшейся странице (рис. 1.2):

* + введите название проекта в поле «Name». Для того чтобы осмысленно выбрать название проекта
  + введите краткое описание в поле «Description»
  + выберите тип репозитория «Git» в поле «Repository type»
  + отметьте пункт «Issue tracking» в поле «Project Management»
  + выберите язык программирования (можно «C++») в поле «Language»
  + нажмите кнопку «Create Repository»

После создания репозитория будет доступно его меню (рис. 1.1, слева). Пункт меню «Clone» позволяет получить адрес репозитория для дальнейшей работы с ним. Пункт меню «Source» в группе «Actions» предназначен для просмотра исходного кода, пункт меню «Commits» позволяет просмотреть историю изменений, пункт «Issues» предназначен для учета ошибок и задач. Кроме того, есть возможность скачать содержимое репозитория с помощью пункта «Downloads».



*Рис. 1.1 – Главная страница BitBucket*



*Рис. 1.2 – Создание репозитория*

**Система контроля версий Git**

*BitBucket* – это централизованная система управления версиями. В настоящее время она используется многими сообществами разработчиков программного обеспечения. BitBucket предоставляет возможность создания как открытых, так и закрытых репозиториев. Создание закрытых репозиторией, рассчитанных на число пользователей более пяти, являются платной услугой.

*Git* является распределённой системой управления версиями. Распределённость подразумевает хранение исходного кода в разных хранилищах. Хранилища могут располагаться как на локальном диске, так и на сетевом сервере.

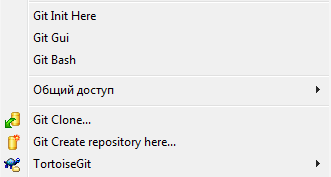
Для организации совместной работы над файлами в большинстве системах контроля версий используется модель копирование – изменение – слияние. Для файлов, не допускающих слияние (например, бинарные файлы), можно использовать модель блокирование – изменение – разблокирование. Однако система Git ориентирована на работу не с самим файлами как таковыми, а с их изменениями, то есть «единицей обработки» для Git является «изменение файла».

Для работы с Git будем использовать бесплатный клиент TortoiseGit, выполненный как расширение оболочки Windows.

**Начало работы с BitBucket**

Для того чтобы скачать репозиторий проекта, необходимо сделать следующее:

1. Создайте папку, в которой будет храниться ваша локальная копия проекта (для удобства лучше хранить все репозитории в одной папке, например, C:\Git).
2. Нажимаем на созданную папку правой кнопкой мыши и из контекстного меню выбираем «Git Clone…» (рис. 1.3).

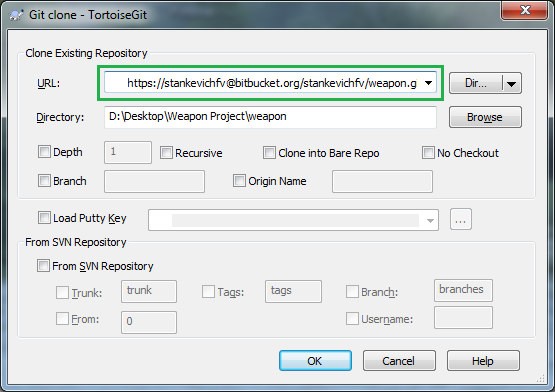
*Рис. 1.3 – Контекстное меню папки*

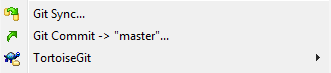
1. В появившемся диалоговом окне (рис. 1.4) в поле «URL» введите полный URL репозитория Git (https://..., без команды *git clone*). (рис. 1.4, пункт «Clone»)
2. Нажмите кнопку «OK», начнется скачивание репозитория с Git-сервера.
3. Затем система попросить ввести пароль от репозитория BitBucket.

**Модификация проекта**

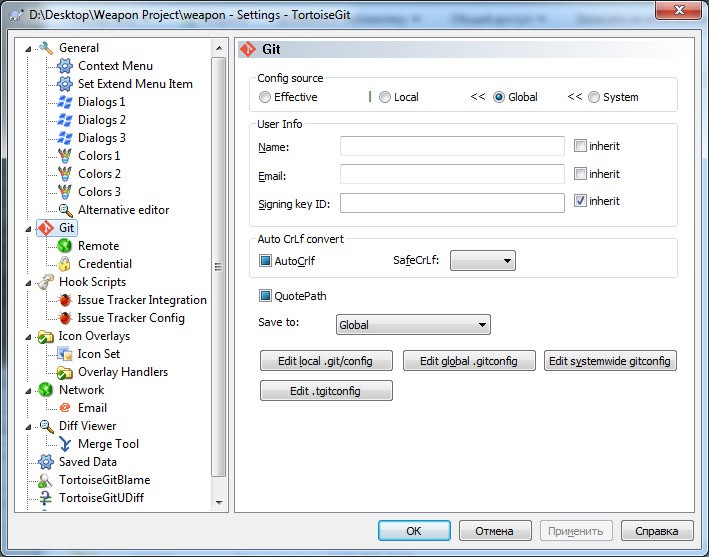
C локальной рабочей копией можно работать как с обычной папкой: создавать, редактировать, удалять файлы и/или папки.

1. Добавим новый файл в папку проекта (файл SampleProgram.cpp).
2. Нажмите правой кнопкой мыши по папке проекта, из контекстного меню выберите «Git Commit - > “master”…» (рис 1.5).

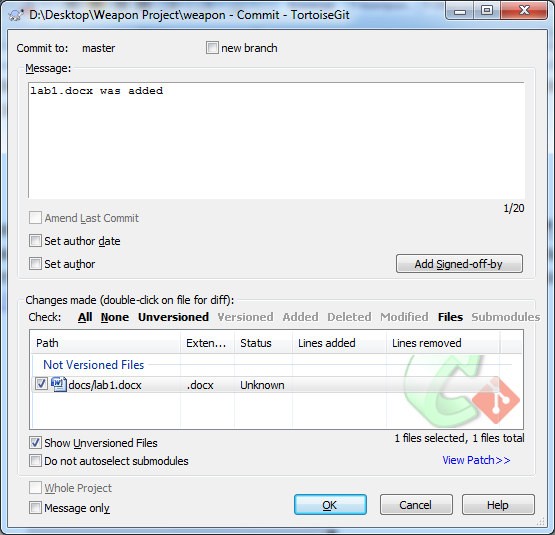
*Рис. 1.4 – Диалоговое окно «Git clone»*

*Рис 1.5 – Контекстное меню*

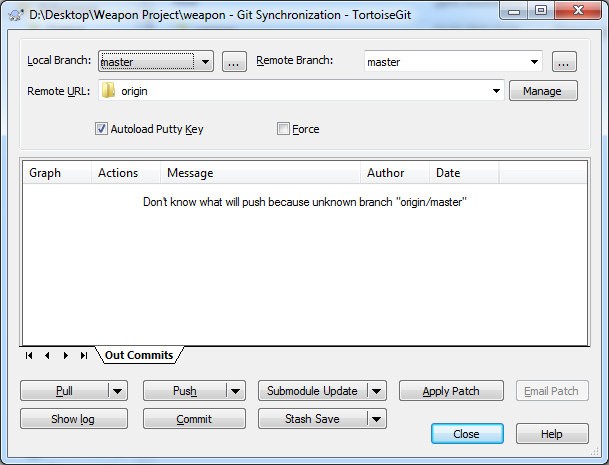
1. При первом коммите TortoiseGit запросит ввести имя пользователя и электронную почту (рис. 1.6)

*Рис. 1.6 – Ввод имени и e-mail*

1. В открывшемся окне (рис. 1.7) в секции «Message» введите краткое описание внесенных изменений (это рекомендуется делать всегда, особенно если не вы один работаете над проектом).
2. В секции «Changes made (double-click on file for diff)» вы увидите список всех внесенных изменённых: добавленных, измененных и удалённых файлов локальной рабочей копии. У изменённых файлов будут автоматически отмечены флажком, остальные – нет.
3. Установите флажки у тех файлов, изменения которых должны быть загружены на git-сервер (есть флажок – файл обновляется, добавляется, удаляется; нет флажка – остаётся без изменений). Двойной щелчок по файлу запустит утилиту Tortoise Merge, предназначенную для сравнения версий файлов, которая покажет последнюю версию файла из репозитория Git в левом окне и текущую рабочую копию в правом. Изменения будут выделены: удалённые строки зачёркнуты, добавленные выделены.

*Рис. 1.7 – Диалоговое окно «Commit».*

1. После выделения нужных файлов или всех сразу, нажмите кнопку «OK». Произойдет фиксация внесенных вами изменений в локальном репозитории. Каждый раз при выполнении фиксации в репозитории создаётся новое состояние дерева файловой системы, называемое *ревизией*. Каждой ревизии назначается уникальный номер. Начальная ревизия не содержит ничего, кроме пустой корневой папки.
2. Для того чтобы отправить изменения на сервер необходимо нажать правой кнопкой мыши по папке проекта снова, из контекстного меню выбрать «Git Sync» (рис. 1.5).
3. В появившемся окне (рис. 1.8) нажмите «Push» и введите пароль.

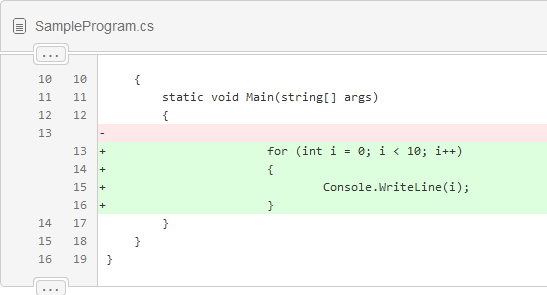


*Рис. 1.8 – Диалоговое окно «Git Sync».*

**Просмотр изменений**

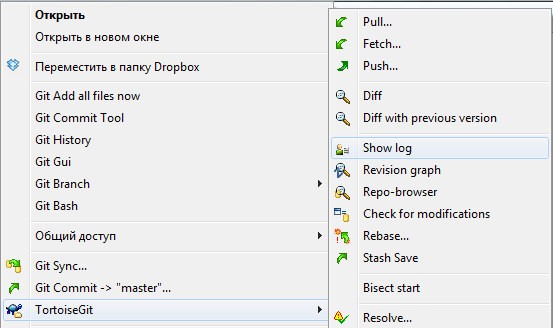
Откройте добавленный ранее файл (SampleProgram.cpp или любой другой) в любом текстовом редакторе и внесите в него изменение. Выполните «Commit» и «Push», как было описано выше.

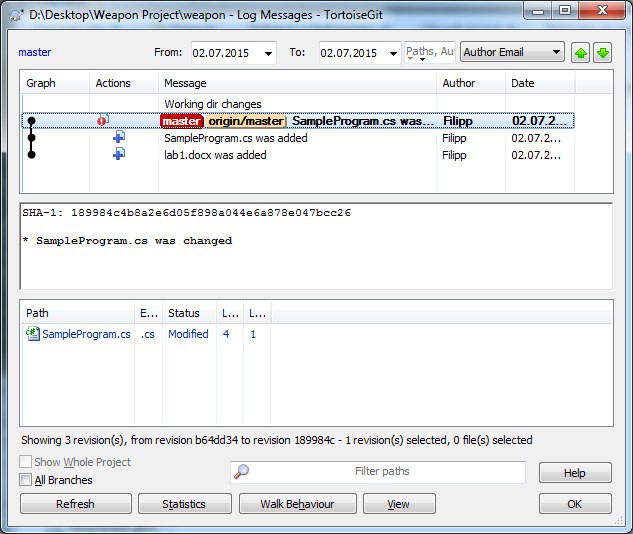
Для просмотра изменений в проекте можно использовать инструменты сайта https://bitbucket.org (пункт меню «Commits» рис. 1.9 изменения выделяются цветом), так и возможности TortoiseGit (рис. 1.10 – 1.11).

*Рис. 1.9 – Просмотр изменений*

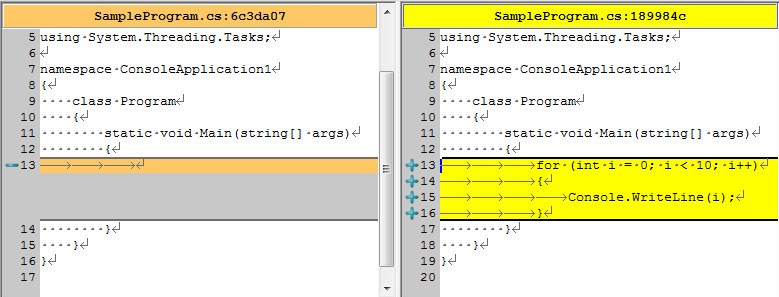
Для просмотра изменений средствами TortoiseGit необходимо вызвать контекстное меню любой папки или файла проекта (рис. 1.10).

TortoiseGit позволяет просмотреть журнал изменений любого файла/каталога (пункт меню «Show log») или построить граф ревизий (пункт меню «Revision graph»). Граф ревизий полезен для больших проектов с несколькими отдельными ветвями разработки. Для отображения изменений в программном коде используется журнал изменений (рис. 1.11). Он показывает список всех фиксаций изменений в файле или папке, а также детальные сообщения разработчиков.

После выбора «Show log» для просмотра отдельных изменений в коде щелкните два раза на изменении из списка. В результате откроется окно утилиты Tortoise Merge, отображающее внесенные изменения (рис. 1.12).

*Рис. 1.10 – Контекстное меню*

*Рис. 1.11 – Журнал изменений*

*Рис. 1.12 – Внесенные изменения*

**Получение последней версии проекта**

Для получения изменений с сервера необходимо нажать кнопку «Pull» в диалоговом окне «Git Sync» (рис. 1.8) и дождаться, пока изменения загрузятся с сервера. Рекомендуется всегда перед «Commit» выполнять действие «Pull». Также «Pull» можно выполнить посредством контекстного меню папки проекта пункт **TortoiseGit → Pull** (рис. 1.10).

**Структура проекта в хранилище**

При разработке программного обеспечения основная задача делится на более мелкие, которые выполняются либо последовательно, либо параллельно относительно друг друга.

При разработке выбирается основная ветвь разработки (в неё входят наиболее важные этапы, оказывающие влияние на весь проект), от неё отходят дочерние ветви, содержащие менее важные процессы. Таким образом, при разработке программных продуктов необходимо обеспечивать возможность одновременной разработки. Стратегически грамотное ветвление позволяет сохранять порядок и последовательность при работе с большим числом версий программного обеспечения.



*Рис. 1.13 – Пример эволюции ветвей*

Формально Git не накладывает каких-либо ограничений на файловую структуру проекта – она может быть какой угодно в рамках правил именования объектов файловой системы. Тем не менее, существуют рекомендации, призванные облегчить работу с ветвями и метками.

Основанная линия разработки проекта называется *trunk*. Дочерние же линии проекта именуются *branches.* Имя конкретной ветки (*branch*) можно выбрать в соответствии с разрабатываемым в ней функционалом, например, *branch\_new\_gui*.

**Задание**

1. Создайте новый проект на BitBucket.
2. Используя TortoiseGit, поместите в него текстовый документ, созданный, например, в Word.
3. Работа в паре – первый участник вносит изменение в документ – второй просматривает изменения, вносит исправления. Просмотр истории изменений.

**Требования и рекомендации:**

Название проекта и все комментарии к коммитам должны быть осмысленными. Это необходимо для того чтобы было удобнее понимать суть проекта и изменений, а также отслеживать их. Кроме того, это поможет человеку, увидевшему проект в первый раз, разобраться в нём.

**Контрольные вопросы:**

1. Что такое системы контроля версий?
2. Какие системы контроля версий вы знаете?
3. Какие существуют основные операции в системе контроля версий?

**Список литературы:**

* 1. Scott Chacon. Pro Git [Электронный ресурс]. – Режим доступа: https://git- scm.com/book/ru/v2, свободный.
  2. Mike McQuaid. Git in Practice, Manning, 2014.
  3. Bitbucket Documentation [Электронный ресурс]. – Режим доступа: https://confluence.atlassian.com/display/BITBUCKET/Bitbucket+Documentation+Home, свободный.

# **ЛАБОРАТОРНАЯ РАБОТА № 2**

### Issue Tracker на примере BitBucket

**Цель работы:**

Получить опыт практической работы в паре при создании, разработки ПО, а также его тестировании и исправлении дефектов.

**Введение**

*Issue tracker* – это система для отслеживания и управления заявками (ошибками в ПО, различными вопросами в тех. поддержку и задачами, которые необходимо решить).

Разновидностью Issue tracker систем являются багтрекеры – системы отслеживания/учёта ошибок.

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

По сути это программа или веб-приложение, в котором пользователи могут добавлять записи об ошибках (задачах, вопросах), назначать ответственное за устранение ошибки (решение задачи) лицо и статус (открыта заявка, в процессе решения, закрыта или требует каких-либо уточнений). Здесь же по каждому вопросу пользователи могут оставлять комментарии и какую-то дополнительную информацию.

Развитые системы позволяют также автоматизировано вести учёт времени, разделять каждую задачу на подзадачи, автоматизировано рассылают уведомления о поступлении заявок/ошибок или комментариев к ним, об изменении статуса и т.д.

Задачам и вопросам можно назначать приоритеты.

При необходимости записи все записи группируются по принадлежности к проекту и его версии (в случае разработки ПО; в случае оказания услуг или тех. поддержки группировка может быть по номеру договора, оказываемой услуге или иному процессу) и могут быть отфильтрованы по этим признакам.

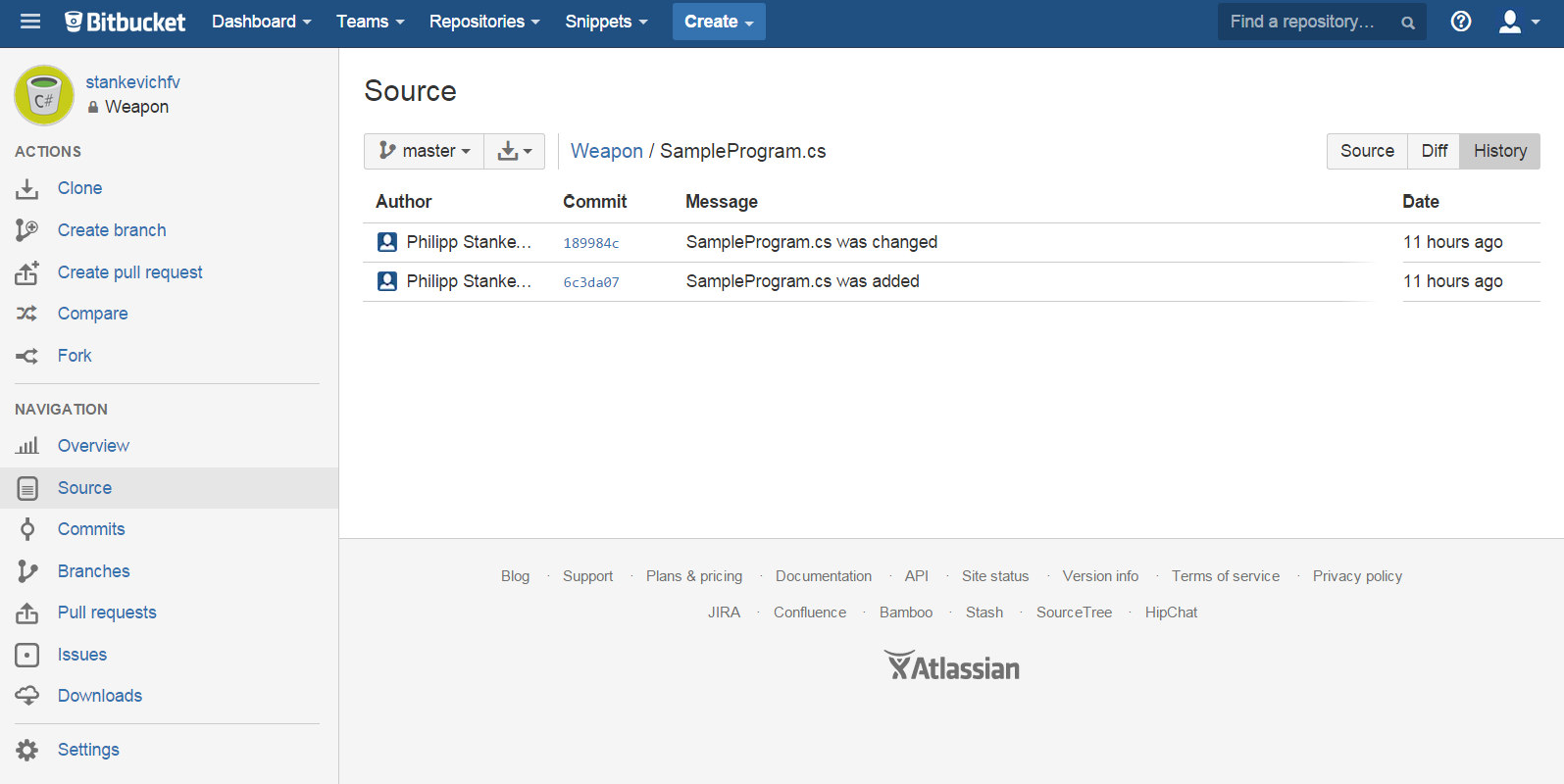
Иногда для каждой задачи указывается стоимость, временные рамки исполнения, зависимости от других задач (какие задачи не могут быть выполнены без выполнения данной задачи или из-за невыполнения каких задач не может быть выполнена данная).

Наиболее сложные и функциональные системы часто имеют возможность взаимодействия с другими системами для получения дополнительных данных или установки связей всех этапов работ (так, например, Issue tracking системы, используемые при разработке ПО могут быть связаны с системами контроля версий, репозиториями исходных кодов, базами знаний и т. д.).

**Работа в BitBucket**

*BitBucket* – это сервис для разработчиков программного обеспечения.

Один из основных сервисов – система контроля версий. Доступные системы контроля версий – Git и Mercurial. В системе контроля версий есть репозиторий для хранения исходного кода (пункт меню «Source»), и можно посмотреть любую версию исходного кода. Также можно просмотреть разницу (diff) между двумя последовательными версиями или посмотреть историю изменений отдельного файла (рис. 2.1).

*Рис. 2.1 – Система контроля версий в BitBucket*

Помимо контроля версий BitBucket также предоставляет возможность использования Issue Tracker (пункт меню «Issues»). Если в программе была найдена ошибка, то её можно там зарегистрировать с описанием, менеджер проекта может назначить её исправление определенному разработчику, а тот исправить.

**Состав информации о дефекте**

Главный компонент такой системы – база данных, содержащая сведения об обнаруженных дефектах. Эти сведения могут включать в себя:

* номер (идентификатор) дефекта;
* кто сообщил о дефекте;
* дата и время, когда был обнаружен дефект;
* версия продукта, в которой обнаружен дефект;
* серьёзность (критичность) дефекта и приоритет решения;
* описание шагов для выявления дефекта (воспроизведения неправильного поведения программы);
* кто ответственен за устранение дефекта;
* обсуждение возможных решений и их последствий;
* текущее состояние (статус) дефекта;
* версия продукта, в которой дефект исправлен.

Кроме того, развитые системы предоставляют возможность прикреплять файлы, помогающие описать проблему (например, дамп памяти или скриншот).

**Жизненный цикл дефекта**

Как правило, система отслеживания ошибок использует тот или иной вариант «жизненного цикла» ошибки, стадия которого определяется текущим состоянием, или статусом, в котором находится ошибка.

Типичный ЖЦ дефекта:

1. Новый – дефект зарегистрирован специалистом по тестированию
2. Назначен – назначен ответственный за исправление дефекта
3. Разрешён – дефект переходит обратно в сферу ответственности тестировщика. Как правило, сопровождается резолюцией, например:
   * Исправлено (исправления включены в версию *№...*)
   * Дубль (повторяет дефект, уже находящийся в работе)
   * Не исправлено (работает в соответствии со спецификацией, имеет слишком низкий приоритет, исправление отложено до следующей версии и т.п.)
   * «У меня всё работает» (запрос дополнительной информации об условиях, в которых дефект проявляется)
4. Далее специалист по тестированию ПО проводит проверку исправления, в зависимости от чего дефект либо снова переходит в статус *Назначен* (если он описан как исправленный, но не исправлен), либо в статус *Закрыт*.
5. Открыт повторно – дефект вновь найден в другой версии.

Система может предоставлять администратору возможность настроить, какие пользователи могут просматривать и редактировать ошибки в зависимости от их состояния, переводить их в другое состояние или удалять.

В корпоративной среде, система отслеживания ошибок может использоваться для получения отчётов, показывающих продуктивность программистов при исправлении ошибок. Однако, часто такой подход не даёт достаточно точных результатов, из-за того, что разные ошибки имеют различную степень серьёзности и сложности. При этом серьёзность проблемы не имеет прямого отношения к сложности устранения ошибки.

**Примеры систем отслеживания ошибок**

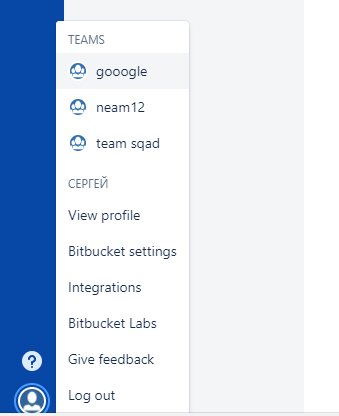
**Свободно распространяемые**

**Проприетарные** **Разное**

* Redmine;
* BUGS – the Bug Genie;
* Bugzilla;
* eTraxis;
* GNATS;
* Mantis bug tracking system;
* Trac;
* EmForge;
* Picket;
* Flyspray;
* DEVPROM.
* Atlassian JIRA;
* Bontq;
* PVCS Tracker;
* Project Kaiser;
* TrackStudio Enterprise;
* YouTrack.
* BugTracker.NET;
* ClearQuest;
* Intland CodeBeamer;
* LifeTask.ru;
* FlySpray;
* StarTeam.

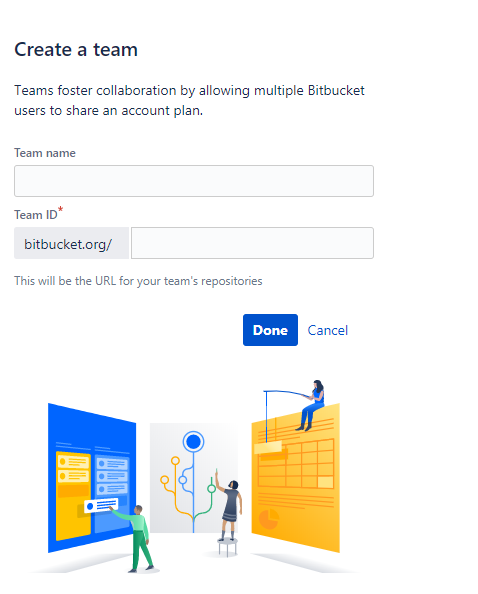
**Задание**

1. Необходимо протестировать Windows-приложение (необходимо создать или использовать графический интерфейс написанного ранее консольного приложения), для этого студентам необходимо разбиться на пары. Один будет выполнять роль *разработчика*, другой – *специалиста по тестированию*.
2. В репозитории необходимо выбрать вашу команду, созданную в лаб. раб. № 1 (рис. 2.2)



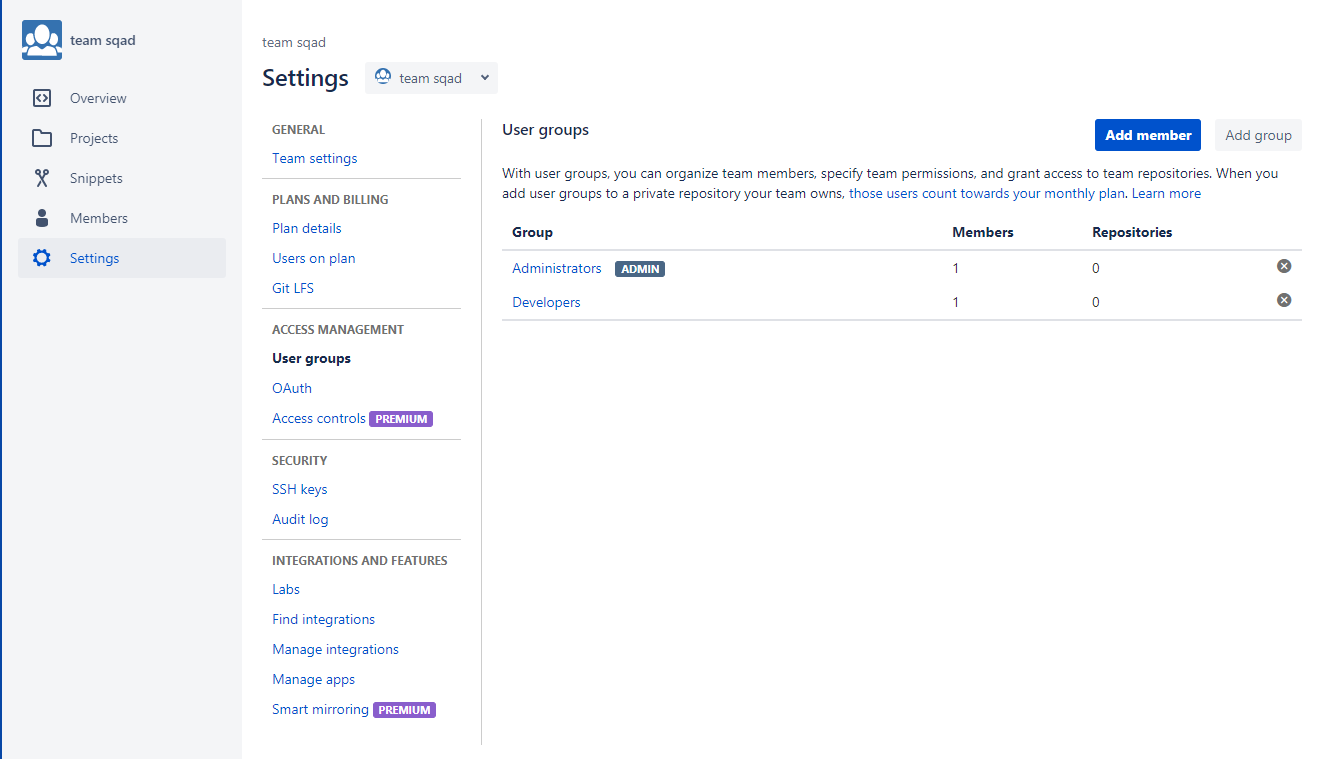
*Рис. 2.2 – Выбор команды*

Если команда не была создана, то необходимо выбрать пункт **Team → Create team** (рис. 2.3).



*Рис. 2.3 – Создание команды*

1. Необходимо нажать кнопку «Manage team» в левом верхнем углу окна (рис. 2.2), затем добавить новую группу «Quality engineers» нажав кнопку «Add Group» и выставить уровень доступа «read» этой группы. Результат показан на рис. 2.4.

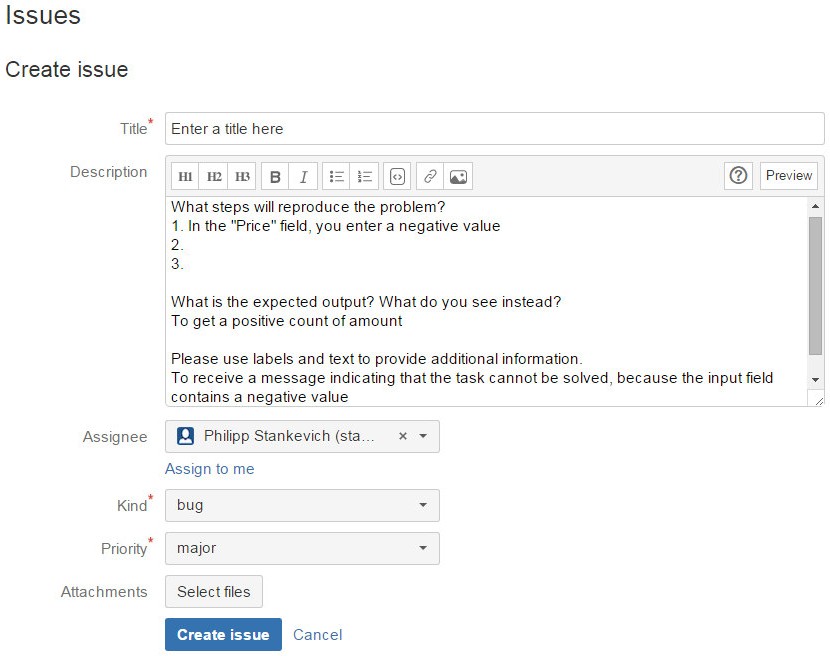


*Рис. 2.4 – Создание группы и добавление в неё пользователя*

1. Затем необходимо ввести имя (или e-mail) пользователя, который будет исполнять роль специалиста по тестированию, в блок группы «Quality engineers» и нажать кнопку «Add».
2. Специалист по тестированию выполняет «Clone» проекта, см. п 1 задания лаб. 2. Инструкцию по выполнению клонирования см. в лаб. раб. №1. Затем специалист по тестированию проверяет программу на наличие ошибок, в качестве которых может служить заполнение поля ввода отрицательными или некорректными значениями. Далее специалист по тестированию создает *issue* типа *bug* в Issue Tracker (пункт «Issues» в меню на главной странице репозитория), с описание найденных ошибок.

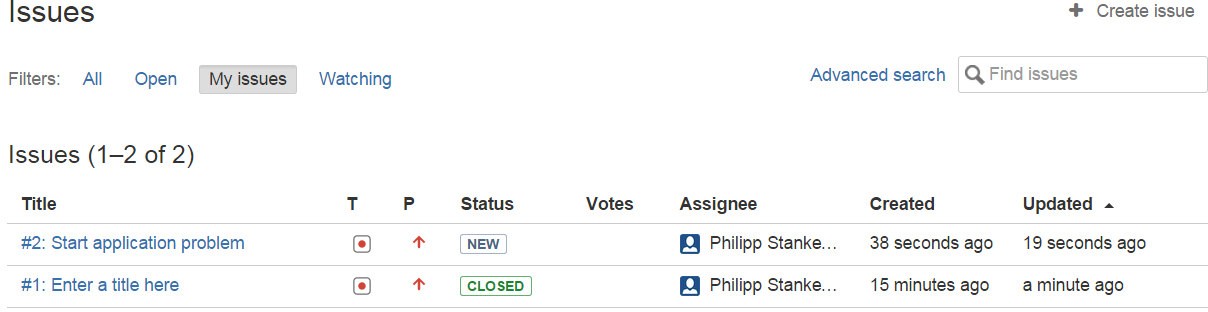
В поле *Title* вводится название ошибки, в поле *Description* – информация о найденной ошибке, т.е. здесь описываются шаги воспроизведения ошибки, результат, который ожидается в итоге, текущий результат. Пример: вы вводите в поле «Цена» отрицательное значение, но в итоге он получаете сообщение об ошибке, т.к. поле ввода содержит отрицательное значение. В поле «Assignee» необходимо выбрать имя разработка из списка. Также можно задать приоритет в поле *Priority*, в зависимости от серьёзности ошибки. Разработчик должен руководствоваться приоритетом при определении порядка исправления ошибок: сначала должны исправляться ошибки с наибольшим приоритетом.

Также необходимо приложить скриншот ошибки, который поможет разработчику локализировать и исправить данную ошибку. Пример создания *issue* представлен на рис. 2.5.



*Рис. 2.5 – Описание ошибки в Issue Tracker*

1. Разработчик просматривает найденный дефект в Issue Tracker, можно воспользоваться пунктом меню My Issues (рис. 2.6). Затем разработкой проверяет описанный сценарий и исправляет ошибку в программе, выполняет Commit и Push изменений с помощью TortoiseGit (см. лабораторную работу № 1). Затем разработчик заходит в Issue Tracker, наживает кнопку «Resolve» (либо через меню «Workflow») (рис. 2.7–2.8), в поле комментарий вводит номер ревизии (номер внесённых изменений), см. пункт меню «Commits». Также можно выполнить редактирование *issue* с помощью кнопки «Edit» (рис. 2.8).



*Рис. 2.6 – My Issues*

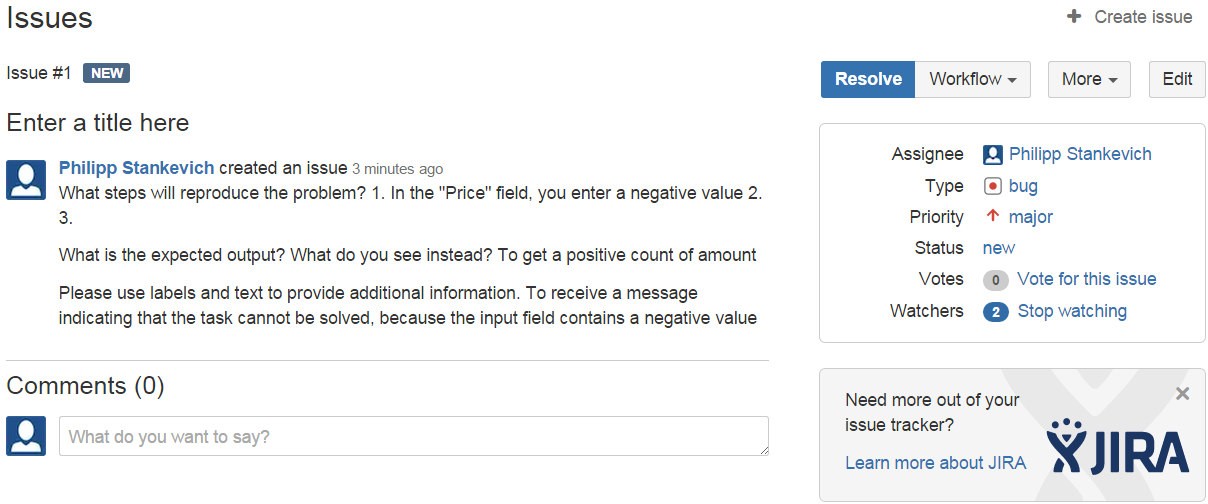
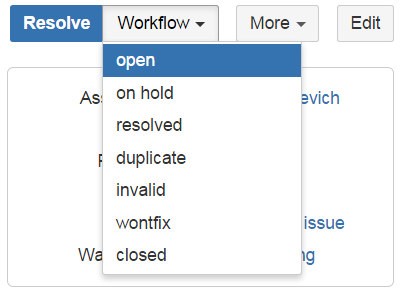


Рис. 2.7. Issue после создания



*Рис. 2.8 – Workflow*

1. Специалист по тестированию обновляет свой локальный репозиторий (Pull) и запускает проект. Проверяет дефект, если он исправлен, то ставит статус Closed, в противном случае выполнит Open дефекта с соответствующим комментарием.
2. Специалисту по тестированию необходимо создать как минимум два дефекта в Issue Tracker, затем *разработчик* и *специалист по тестированию* меняются ролями.

**Контрольные вопросы:**

1. Что такое Issue Tracker? Где и для чего применяется?
2. Что является разновидностью Issue Tracker? (дать определение)
3. Какие сведения может включать в себя база данных об обнаруженных ошибках?
4. Привести пример жизненного цикла дефекта (с пояснениями).

**Список литературы:**

1. Scott Chacon. Pro Git [Электронный ресурс]. – Режим доступа: https://git- scm.com/book/ru/v2, свободный.
2. Mike McQuaid. Git in Practice, Manning, 20 14
3. Ron Patton. Software Testing, 2nd ed., Sams Publishing, 2006.
4. Савин Р. Тестирование dot com – М. : Дело, 2007 – 312 с.
5. Котляров В.П. Основы тестирования ПО – М. : Интернет-Ун-т Информ. Технологий, 2006. – 288 с.

# **ПРАКТИЧЕСКАЯ РАБОТА № 2, 3**

### Расчёт параметров надёжности программного обеспечения по математической модели Шумана и Миллса

**Теоретические сведения**

Надёжность ПО – это свойство ПО сохранять работоспособность в течении определенного периода времени, в определённых условиях эксплуатации с учётом последствий для пользователя каждого отказа.

Параметры надёжности ПО такие же, как и аппаратного (технического) обеспечения. Причинами отказа ПО являются ошибки, вызванные либо внутренними свойствами ПО, либо средой функционирования ПО математической модели (ММ) ПО подразделяются на три класса:

1. динамические модели;
2. статические; при этом считается, что время прогона ПО делится на дискретное и непрерывное. К ММ дискретного времени относятся: ММ Шумана, модифицированная ММ Шумана, ММ La Padula и ММ Шика-Вольвертона. К динамическим ММ дискретного времени относятся: ММ Миллса, ММ Липова, простая интуитивная ММ, ММ Коркорэна и ММ Нельсона
3. эмпирические модели: ММ сложности и ММ, определяющая время доводки программ.

Будем определять параметры расчёта ПО по моделям Шумана и модели Миллса.

**1. ММ Шумана.**

Это динамическая модель дискретного времени. Используются тестовые прогоны программы. Считается, что количество исходных ошибок в ПО постоянно, т.е. const, новые ошибки при корректировании ПО не вносятся.

В ММ Шумана вводятся две модели:

* + - 1. математическая модель ошибки;
      2. математическая модель надёжности ПО.

Математическая модель ошибки выглядит так:

, (1)

где  экспериментальное количество ошибок в ПО в момент ;

 экспериментальное количество инструкций машинного языка;

 экспериментальное количество ошибок, найденных и исправленных к моменту времени ;

 экспериментальное количество ошибок, оставшихся в ПО.

1. Математическая модель ПО есть функция , , , ,

где *t* – время отладки ПО с начала его запуска;

 – время отладки с начала интеграции ПО;

 – функция количества ошибок;

 – оценка коэффициента пропорциональности.

Среднее время надёжности есть

. (2)

Оценка  определяется как

 (3)

Оценка функциональности ПО  есть

 (4)

где  – время прогона ПО;

*t* – время работы ПО.

Неизвестные параметры  и оцениваются путём их определения в двух точках *А* и *В* ПО так, чтобы выполнялось неравенство, т.е.

 (5)

при этом

 (6)

Теперь получим

 (7)

 (8)

Количество ошибок на интервале будем обозначать через  и , их значение мы определяем путём непосредственного подсчёта значений ошибок из таблицы значений.

Далее подсчитываются интенсивности появления ошибок  и  на интервалах *А* и *В* соответственно

 (9)

 (10)

**Пример.** В тестируемой программе 1000 (*I*) операторов, в процессе последовательных 10 тестовых прогонов были получены данные, сведённые в таблицу 1.

Таблица 1

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| № прогона | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Количество ошибок, (х) | 5 | 4 | 2 | 5 | 7 | 2 | 3 | 3 | 2 | 3 |
| Время прогона  (мсек) | 4 | 4 | 3 | 4 | 6 | 4 | 4 | 5 | 3 | 2 |

Определить интервалы *А* и *В* так, чтобы выполнялись условия:

 и 





,  т.е. 11 < 28.

Определим оценки интенсивности  и 





Подсчитываем числовые значения  и  и имеем

(ошибок)



Подсчитываем значения функции надёжности  при различных значениях  и . Имеем

Далее делаем различные значения *t*, например 1,2,3,4,5,6 и т.д. и считаем значения и стром графики зависимости  от *t*.

Таблица вариантов 2

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| № варианта | № прогона | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | Время прогона *t* |
| 1 | Количество ошибок () | 1 | 2 | 2 | 2 | 1 | 1 | 3 | 2 | 1 | 3 | 1 |
| Время прогона () | 2 | 2 | 1 | 1 | 2 | 2 | 3 | 2 | 4 | 2 |
| 2 | Количество ошибок () | 3 | 2 | 1 | 2 | 3 | 4 | 2 | 1 | 2 | 3 | 2 |
| Время прогона () | 3 | 2 | 3 | 1 | 2 | 3 | 2 | 2 | 2 | 3 |
| 3 | Количество ошибок () | 4 | 1 | 2 | 3 | 2 | 2 | 5 | 2 | 3 | 1 | 3 |
| Время прогона () | 4 | 2 | 3 | 4 | 4 | 5 | 3 | 3 | 2 | 1 |
| 4 | Количество ошибок () | 3 | 2 | 3 | 2 | 2 | 3 | 4 | 2 | 3 | 2 | 3 |
| Время прогона () | 3 | 2 | 4 | 4 | 2 | 3 | 3 | 2 | 3 | 4 |
| 5 | Количество ошибок () | 2 | 3 | 4 | 2 | 3 | 4 | 5 | 4 | 3 | 2 | 2 |
| Время прогона () | 4 | 2 | 3 | 2 | 3 | 4 | 3 | 2 | 5 | 3 |
| 6 | Количество ошибок () | 1 | 2 | 4 | 2 | 3 | 3 | 4 | 4 | 2 | 3 | 1 |
| Время прогона () | 4 | 3 | 2 | 1 | 3 | 2 | 3 | 3 | 2 | 4 |
| 7 | Количество ошибок () | 2 | 3 | 3 | 4 | 5 | 1 | 2 | 3 | 2 | 3 | 2 |
| Время прогона () | 4 | 3 | 5 | 2 | 3 | 2 | 4 | 3 | 2 | 2 |
| 8 | Количество ошибок () | 3 | 1 | 2 | 4 | 3 | 2 | 3 | 4 | 2 | 3 | 2 |
| Время прогона () | 3 | 2 | 2 | 3 | 4 | 2 | 3 | 2 | 4 | 2 |
| 9 | Количество ошибок () | 2 | 2 | 3 | 4 | 3 | 2 | 4 | 3 | 3 | 2 | 3 |
| Время прогона () | 2 | 3 | 5 | 3 | 2 | 4 | 3 | 5 | 2 | 4 |
| 10 | Количество ошибок () | 3 | 3 | 4 | 5 | 4 | 4 | 3 | 2 | 3 | 3 | 3 |
| Время прогона () | 4 | 2 | 3 | 5 | 3 | 3 | 2 | 5 | 3 | 1 |
| 11 | Количество ошибок () | 3 | 3 | 4 | 4 | 4 | 2 | 3 | 1 | 2 | 3 | 2 |
| Время прогона () | 4 | 2 | 5 | 3 | 2 | 1 | 1 | 2 | 3 | 3 |
| 12 | Количество ошибок () | 4 | 4 | 2 | 2 | 3 | 3 | 4 | 4 | 2 | 3 | 4 |
| Время прогона () | 3 | 3 | 2 | 2 | 4 | 4 | 5 | 3 | 4 | 4 |
| 13 | Количество ошибок () | 3 | 3 | 2 | 4 | 3 | 4 | 5 | 2 | 3 | 2 | 3 |
| Время прогона () | 3 | 4 | 3 | 3 | 2 | 4 | 4 | 4 | 3 | 2 |
| 14 | Количество ошибок () | 4 | 3 | 2 | 3 | 2 | 4 | 2 | 3 | 4 | 3 | 2 |
| Время прогона () | 3 | 4 | 4 | 3 | 2 | 3 | 5 | 3 | 2 | 1 |
| 15 | Количество ошибок () | 3 | 2 | 1 | 1 | 2 | 3 | 2 | 3 | 2 | 1 | 2 |
| Время прогона () | 4 | 3 | 2 | 2 | 1 | 3 | 2 | 3 | 1 | 2 |

**2. ММ Миллса**

Использование этой модели предполагает необходимость перед началом тестирования искусственно вводить в программу некоторое количество ошибок («засорять»). Ошибки вносятся случайным образом и фиксируются в протоколе искусственных ошибок. Предполагается, что ошибки имеют равную вероятность быть найденными в процессе тестирования.

Тестируя программу в течение некоторого времени, собирается статистика об ошибках в ПО. В сам момент оценки, по имеющемуся протоколу искусственных ошибок делятся на собственные (*n*) и искусственные. Первая часть модели Миллса связана с использованием в расчётах формулы, которая задаётся так:



где – целая часть от значения дроби, округляемая в большую сторону;

 – количество первоначальных ошибок с ПО;

 – количество искусственно внесённых ошибок;

 – количество найденных собственных ошибок;

 – количество обнаруженных к моменту оценки искусственных ошибок.

Вторая часть модели связана с проверкой гипотезы о значении . Для этого подсчитывается вероятность по формуле

 (11)

где *k* – реальное количество собственных ошибок в программе.

Величина  является мерой доверия к модели Миллса и показывает, насколько правильно вычислено значение . Варианты для исследования модели Миллса сведены в таблицу 3. (Данные  и записываются после проведения расчётов в таблицу 3).

Таблица 3

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| № прогона |  |  |  |  |  |  |
| 1 | 10 | 3 | 2 | 2 |  |  |
| 2 | 20 | 3 | 4 | 3 |  |  |
| 3 | 15 | 4 | 2 | 2 |  |  |
| 4 | 25 | 3 | 1 | 3 |  |  |
| 5 | 35 | 2 | 1 | 1 |  |  |
| 6 | 45 | 3 | 2 | 2 |  |  |
| 7 | 40 | 5 | 2 | 3 |  |  |
| 8 | 30 | 4 | 2 | 3 |  |  |
| 9 | 25 | 3 | 1 | 3 |  |  |
| 10 | 35 | 2 | 1 | 3 |  |  |
| 11 | 45 | 1 | 0 | 0 |  |  |
| 12 | 50 | 4 | 3 | 3 |  |  |
| 13 | 35 | 3 | 2 | 2 |  |  |
| 14 | 32 | 2 | 1 | 1 |  |  |
| 15 | 33 | 5 | 2 | 5 |  |  |

Отчёт по заданию должен содержать:

* + - 1. результаты расчётов и данные, необходимые для сделанного студентом варианта;
      2. выводы, говорящие о том, какая ММ и по каким соображениям предпочтительней;
      3. сравнительный анализ двух рассмотренных ММ ПО;
      4. вариант студента из таблицы 2 и таблицы 3 также помещается в отчёт.

# **ПРАКТИЧЕСКАЯ РАБОТА № 4**

### Составление наборов тестовых данных для функционального тестирования. Стратегия «чёрного ящика»

**Основные теоретические положения**

При функциональном тестировании программа рассматривается как «чёрный ящик», и целью тестирования является выяснение обстоятельств, в которых поведение программы не соответствует спецификации.

*Исчерпывающее* тестирование, т.е. тестирование на всех возможных наборах данных, в большинстве случаев невозможно (потребовались бы гигантские затраты времени и средств). А для программ, где исполнение команды зависит от предшествующих ей событий, необходимо также проверить и все возможные последовательности! Но, с другой стороны, необходимо обеспечить требуемое качество программного обеспечения. Возникает вопрос: как обнаружить максимальное количество ошибок, ограничив при этом подмножество возможных входных данных?

Существуют различные методы формирования тестовых наборов. В числе основных **критериев «чёрного ящика»** следующие:

1. **покрытие функций**. Для каждой из функций, реализуемых программой, требуется подобрать и выполнить хотя бы один тест;
2. **покрытие классов входных данных.** Критерий тестирования классов входных данных требует классифицировать входные данные, разделить их на классы таким образом, чтобы все данные из одного класса были равнозначны с точки зрения проверки правильности программы. Считается, что если программа работает правильно на одном наборе входных данных из этого класса, то она будет правильно работать на любом другом наборе данных из этого же класса. Критерий требует выполнения хотя бы одного теста для каждого класса входных данных;
3. **покрытие классов выходных данных**. Аналогичен предыдущему критерию, только проверяются не входные данные, а выходные;
4. **покрытие области допустимых значений** (тестирование границ класса). Для переменной, возможные значения которой перечислены (ноты, цвет, пол, диагноз и т.п.), следует убедиться, что на все указанные значения программа реагирует правильно и не принимает вместо них никаких иных значений. Если класс допустимых значений представляет собой числовой диапазон, то выделяются нормальные условия (в середине класса), граничные (экстремальные) условия и исключительные условия (выход за границу класса);
5. **тестирование длины набора данных** (можно считать частным случаем тестирования области допустимых значений). Определяется допустимое количество элементов в наборе. Если программа последовательно обрабатывает элементы некоторого набора данных, и имеет смысл проверить следующие ситуации:

* пустой набор (не содержит ни одного элемента);
* единичный набор (состоит из одного-единственного элемента);
* слишком короткий набор (если предусмотрена минимально допустимая длина);
* набор минимально возможной длины (если такая предусмотрена); нормальный набор (состоит из нескольких элементов);
* набор из нескольких частей (если такое возможно. Например, если программа читает литеры из текстового файла или печатает текст, то как она отнесется к переходу на следующую строку? На следующую страницу?);
* набор максимально возможной длины (если такая предусмотрена);
* слишком длинный набор (с длиной больше максимально допустимой);

1. тестирование **упорядоченности набора данных**. Важно для задач сортировки и поиска экстремумов. В этом случае имеет смысл проверить следующие ситуации (классы входных данных): данные не упорядочены; данные упорядочены в прямом порядке; данные упорядочены в обратном порядке; в наборе имеются повторяющиеся значения; экстремальное значение находится в середине набора; экстремальное значение находится в начале набора; экстремальное значение находится в конце набора; в наборе несколько совпадающих экстремальных значений.

Критерии покрытия функций, классов входных и выходных данных хорошо согласуются друг с другом и обычно каждой из функций ПП соответствуют свои классы входных и выходных данных. Например, программа учёта кадров предприятия имеет функции: принять на работу, уволить с работы, перевести с одной должности на другую, выдать кадровую сводку. Соответственно классы входных данных – приказы о приёме, об увольнении, о переводе, заявка на кадровую сводку, а выходных – записи о приёме, об увольнении, о переводе, кадровая сводка.

В данной практической работе предлагается рассмотреть два **метода формирования тестовых наборов**:

1. на основе классов эквивалентности;
2. на основе граничных значений классов эквивалентности.

Оба метода основаны на исследовании входных данных. Они не позволяют проверять результаты, получаемые при различных сочетаниях данных. Для построения тестов, проверяющих сочетания данных, применяют методы, использующие булеву алгебру.

**Эквивалентное разбиение.** Метод эквивалентного разбиения заключается в следующем. Область всех возможных наборов входных данных программы по каждому параметру разбивают на конечное число групп - классов эквивалентности. Наборы данных такого класса объединяют по принципу обнаружения одних и тех же ошибок: если набор какого-либо класса обнаруживает некоторую ошибку, то предполагается, что все другие тесты этого класса эквивалентности тоже обнаружат эту ошибку и наоборот.

Разработку тестов методом эквивалентного разбиения осуществляют в два этапа: на первом выделяют классы эквивалентности, а на втором - формируют тесты.

Формирование классов эквивалентности является эвристическим процессом, однако целесообразным считают выделять в отдельные классы эквивалентности наборы, содержащие допустимые и недопустимые значения некоторого параметра. При этом существует ряд правил:

* если некоторый параметр х может принимать значения в интервале [1, 999], то выделяют один правильный класс 1 < х < 999 и два неправильных: х < 1 и х > 999;
* если входное условие определяет диапазон значений порядкового типа, например, «в автомобиле могут ехать от одного до шести человек», то определяется один правильный класс эквивалентности и два неправильных: ни одного и более шести человек;
* если входное условие описывает множество входных значений и есть основания полагать, что каждое значение программист трактует особо, например, «типы графических файлов: bmp, jpeg, vsd», то определяют правильный класс эквивалентности для каждого значения и один неправильный класс, например, txt;
* если входное условие описывает ситуацию «должно быть», например, «первым символом идентификатора должна быть буква», то определяется один правильный класс эквивалентности (первый символ - буква) и один неправильный (первый символ - не буква);
* если есть основание считать, что различные элементы класса эквивалентности трактуются программой неодинаково, то данный класс разбивается на меньшие классы эквивалентности.

Таким образом, классы эквивалентности выделяют, перебирая ограничения, установленные для каждого входного значения в техническом задании или при уточнении спецификации. Каждое ограничение разбивают на две или более групп. При этом используют специальные бланки - таблицы классов эквивалентности:

|  |  |  |
| --- | --- | --- |
| Ограничение на значение параметра | Правильные классы эквивалентности | Неправильные классы эквивалентности |
|  |  |  |

Правильные классы включают правильные данные, неправильные классы - неправильные данные. Для правильных и неправильных классов тесты проектируют отдельно. При построении тестов правильных классов учитывают, что каждый тест должен проверять по возможности максимальное количество различных входных условий. Такой подход позволяет минимизировать общее число необходимых тестов. Для каждого неправильного класса эквивалентности формируют свой тест. Последнее обусловлено тем, что определённые проверки с ошибочными входами скрывают или заменяют другие проверки с ошибочными входами.

**Анализ граничных значений.** *Граничные значения -* это значения на границах классов эквивалентности входных значений или около них. Анализ показывает, что в этих местах резко увеличивается возможность обнаружения ошибок. Например, если в программе анализа вида треугольника было записано А + В ≥ С вместо А + В > С, то задание граничных значений приведет к ошибке: линия будет отнесена к одному из видов треугольника.

Применение метода анализа граничных значений требует определённой степени творчества и специализации в рассматриваемой проблеме. Тем не менее, существует несколько общих правил для применения этого метода:

* если входное условие описывает область значений, то следует построить тесты для границ области и тесты с неправильными входными данными для ситуаций незначительного выхода за границы области, например, если описана область [-1.0, +1.0], то должны быть сгенерированы тесты: -1.0, +1.0,-1.001 и +1.001;
* если входное условие удовлетворяет дискретному ряду значений, то следует построить тесты для минимального и максимального значений и тесты, содержащие значения большие и меньшие этих двух значений, например, если входной файл может содержать от 1 до 255 записей, то следует проверить 0, 1, 255 и 256 записей;
* если существуют ограничения выходных значений, то целесообразно аналогично тестировать и их: конечно, не всегда можно получить результат вне выходной области, но тем не менее стоит рассмотреть эту возможность;
* если некоторое входное или выходное значение программы является упорядоченным множеством, например, это последовательный файл, линейный список или таблица, то следует сосредоточить внимание на первом и последнем элементах этого множества.

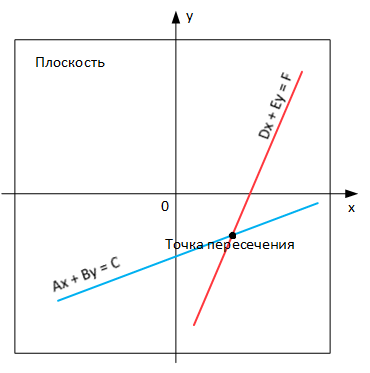
Помимо указанных граничных значений, целесообразно поискать другие. Однако следует помнить, что граничные значения могут быть едва уловимы и определение их связано с большими трудностями, что является недостатком этого метода.

**Пример.** Пусть необходимо выполнить тестирование программы, определяющей точку пересечения двух прямых на плоскости. При этом она должна определять параллельность прямой одной из осей координат.

В основе программы лежит решение системы линейных уравнений (СЛАУ):

Прямая 1: Ax + By = C;

Прямая 2: Dx + Ey = F.



*По методу эквивалентных разбиений* формируем для каждого коэффициента (A, B, C, D, E, F) один правильный класс эквивалентности (коэффициент – вещественное число – число с дробной частью) и один неправильный (коэффициент – не вещественное число). Откуда генерируем 7 тестов:

1) все коэффициенты - вещественные числа (1 тест);

2-7) поочередно каждый из коэффициентов - не вещественное число (6 тестов).

*По методу граничных значений* можно считать, что для исходных данных граничные значения отсутствуют, т.е. коэффициенты - «любые» вещественные числа.

Для результатов получаем, что возможны варианты: единственное решение, прямые сливаются - множество решений, прямые параллельны - отсутствие решений. Следовательно, целесообразно предложить тесты с результатами внутри областей возможных значений результатов:

8) результат - единственное решение 

определитель 

9) результат - множество решений 

определитель 

определитель 

определитель 

10) результат - отсутствие решений 

определитель 

определитель 

или

определитель 

и с результатами на границе:

11) 

12) 

13) 

14) 

**Задачи**

**Задача 1.** Есть программа, которая интерпретирует три целых числа, вводимых с клавиатуры, как длины сторон треугольника и выводит сообщение, о том, какой это треугольник: равносторонний, равнобедренный или неравносторонний. Напишите на листе бумаги тесты (последовательности входных данных и ожидаемые результаты), которые, как вам кажется, будут адекватно проверять эту программу. Рекомендуется запись тестов в виде таблицы:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Критерий формирования теста | № теста | Исходные данные | | | Ожидаемый результат |
| a | b | c |
|  |  |  |  |  |  |

**Задача 2.** В программе «Деканат» переменная, обозначающая количество студентов в группе, может принимать значения от 1 до 30. Составьте тестовый набор для этой переменной.

**Задача 3.** В компьютерной обучающей системе тестовые задания для контроля знаний берутся из файла типа .txt, где каждое задание занимает одну строку. Для формирования теста указывается имя файла, количество заданий в тесте и количество вариантов теста (не более 10). Варианты должны различаться не менее чем тремя заданиями. Если заданий в файле недостаточно для реализации этого требования, выдается сообщение «Недостаточно заданий», если файл не найден - «файл отсутствует». Увидеть составленный вариант теста для контроля знаний можно, нажав на соответствующую кнопку «Вариант №».

Составьте тестовые наборы для проверки перечисленных функций. Рекомендуется оформить их в виде таблицы, подобно задаче 1.

**Задача 4.** Используя методы эквивалентного разбиения и граничных значений, ***составьте*** тестовые наборы для проверки перечисленных ниже функций программы "Геометрические фигуры". Протестируйте указанную программу.

В программе "Геометрические фигуры" (файл geometry.exe) после ее запуска пользователю предоставляется выбрать вариант геометрической программы. Для выбора пользователь должен нажать на кнопку с названием нужного варианта. Предусмотрены следующие варианты:

***Вариант 1.*** Программа определяет тип треугольника. Возможные результаты: прямоугольный, остроугольный, тупоугольный, равнобедренный, правильный (равносторонний)

***Вариант 2.*** Четырехугольник задается координатами вершин. Программа проверяет, является ли он квадратом

***Вариант 3.*** Четырехугольник задается координатами вершин. Программа проверяет, является ли он ромбом

**Контрольные вопросы**

1. Почему стратегия функционального тестирования называется также стратегией "черного ящика"?

2. По каким критериям осуществляется функциональное тестирование?

3. В чем заключается метод эквивалентного разбиения?

4. По какому принципу формируют классы эквивалентности?

5. Почему анализ граничных значений считается одним из наиболее полезных методов проектирования тестов?

# **ПРАКТИЧЕСКАЯ РАБОТА № 5**

### Составление наборов тестовых данных для структурного тестирования. Стратегия «белого ящика»

**Основные теоретические положения**

При использовании стратегии «белого ящика» тестовые наборы формируют путём анализа маршрутов, предусмотренных алгоритмом. Под ***маршрутами*** при этом понимают последовательности операторов программы, которые выполняются при конкретном варианте исходных данных. Тестирование, проводимое по составленным таким образом тестам, называют структурным или тестированием по маршрутам.

В основе структурного тестирования лежит концепция максимально полного тестирования всех маршрутов программы. Так, если алгоритм программы включает ветвление, то при одном наборе исходных данных может быть выполнена последовательность операторов, реализующая действия, которые предусматривает одна ветвь, а при втором − другая. Соответственно, для программы будут существовать маршруты, различающиеся выбранным при ветвлении вариантом.

Считают, что программа проверена полностью, если с помощью тестов удаётся осуществить выполнение программы по всем возможным маршрутам передач управления. Однако нетрудно видеть, что даже в программе среднего уровня сложности число неповторяющихся маршрутов может быть очень велико, и, следовательно, полное или исчерпывающее тестирование маршрутов, как правило, невозможно.

***Последовательность составления тестов*** следующая:

1. На основе алгоритма (текста) программы формируется потоковый граф (или граф-схема). Узлы (вершины) потокового графа соответствуют линейным участкам программы, включают один или несколько операторов программы. Дуги (ориентированные ребра) потокового графа отображают поток управления в программе (передачи управления между операторами). Различают операторные и предикатные узлы. Из операторного узла выходит одна дуга, а из предикатного − две дуги. *Предикатные узлы соответствуют* ***простым*** *условиям в программе.* Составное условие программы (условие, в котором используется одна или несколько булевых операций (OR, AND)) отображается в несколько предикатных узлов.

На рис 5.1 показан пример такого преобразования для фрагмента программы: *if a OR b then x else у end if.*

*Рис. 5.1 – Прямое отображение фрагмента программы в потоковый граф и преобразованный потоковый граф*

1. Выбирается критерий тестирования. Формирование тестовых наборов для тестирования маршрутов может осуществляться по нескольким критериям: покрытие маршрутов, покрытие операторов, покрытие решений (переходов), покрытие условий, покрытие решений/условий, комбинаторное покрытие условий, покрытие потоков данных, покрытие циклов.
2. Подготавливаются тестовые варианты, инициирующие выполнение каждого пути. Каждый тестовый вариант формируется в следующем виде:

*Исходные данные (ИД):*

*Ожидаемые результаты (ОЖ.РЕЗ.):*

Тесты, составленные по критерию **покрытие маршрутов** и обеспечивающие проверку базового множества, гарантируют однократное выполнение каждого оператора и выполнение каждого условия по True-ветви и по False-ветви. Базовое множество образуют все независимые пути графа. Независимым называется любой путь от начального до конечного узла графа, который вводит новый оператор обработки или новое условие. В терминах потокового графа независимый путь должен содержать дугу, не входящую в ранее определенные пути. Независимые пути формируются в порядке от самого короткого к самому длинному.

Число независимых линейных путей в базовом множестве определяется ***цикломатической сложностью алгоритма***, которая вычисляется одним из трёх способов:

* цикломатическая сложность V(G) равна количеству регионов потокового графа; Регион - замкнутая область, образованная дугами и узлами. Окружающая граф среда рассматривается как дополнительный регион. Например, показанный на рис. 5.1 граф имеет три региона − Rl, R2, R3;
* цикломатическая сложность определяется по формуле
* V(G) = E-N+2,

где Е − количество дуг, N − количество узлов потокового графа;

* цикломатическая сложность V(G) = р +1, где р – количество предикатных узлов G.

**Пример 1**.

Цикломатическая сложность алгоритма на рис. 5.1:

1) V(G) = 3 региона;

2) V(G) = 7 дуг - 6 узлов + 2 = 3;

3) V(G) = 2 предикатных узлов + 1 = 3.

Независимые пути:

Путь 1: а - х – end if.

Путь 2: а - b - х – end if.

Путь 3: а - b - y – end if.

Критерий **покрытия операторов** подразумевает такой подбор тестов, чтобы каждый оператор программы выполнялся, по крайней мере, один раз. Это необходимое, но недостаточное условие для приемлемого тестирования.

Для реализации критерия **покрытие решений (переходов)** необходимо такое количество и состав тестов, чтобы результат проверки каждого условия (т.е. решение) принимал значения «истина» или «ложь», по крайней мере, один раз. Нетрудно видеть, что критерий покрытия решений удовлетворяет критерию покрытия операторов, но является более «сильным».

Критерий **покрытия** условий является ещё более «сильным» по сравнению предыдущими. В этом случае формируют некоторое количество тестов, достаточное для того, чтобы все возможные результаты каждого условия в решении были выполнены, по крайней мере, один раз. Однако, как и в случае покрытия решений, этот критерий не всегда приводит к выполнению каждого оператора, по крайней мере, один раз. К критерию требуется дополнение, заключающееся в том, что каждой точке входа управление должно быть передано, по крайней мере, один раз.

Согласно **покрытию решений/условий** тесты должны составляться так, чтобы, по крайней мере, один раз выполнились все возможные результаты каждого условия и все результаты каждого решения, и каждому оператору управление передавалось, по крайней мере, один раз.

Критерий **комбинаторное покрытие условий** требует создания такого множества тестов, чтобы все возможные комбинации результатов условий в каждом решении и все операторы выполнялись, по крайней мере, один раз.

**Пример 2.**

Требуется выполнить структурное тестирование текста программы, которая определяет значение х в зависимости от значений параметров процедуры.

*Procedure т (a, b:real; var x:real);*

*begin*

*if(a>1) and (b=0) then x:=x /a;*

*if(a=2) or (x>l) then x:=x+1;*

*end;*

Для формирования тестов программу представляют в виде графа, вершины которого соответствуют операторам программы, а дуги представляют возможные варианты передачи управления (рис. 5.2).

**Покрытие операторов** будет реализовано при а = 2, b = 0, х = 3.

Однако, хотя исходные данные заданы так, чтобы все операторы программы были выполнены хотя бы один раз, для проверки программы этого явно недостаточно. Например, из второго условия следует, что переменная *х* может принимать любое значение. Кроме того, если при написании программы в первом условии указано, что (а > 1) or (b = 0), или, если во втором условии вместо х > 1 записано х > 0, то эти ошибки обнаружены не будут. Также существует путь 1-2-4-6*,* в котором *х* вообще не меняется и, если здесь есть ошибка, она не будет обнаружена.



*Рис. 5.2 – Схема алгоритма процедуры (слева) и её граф передач управления*

По методу **покрытия решений (переходов)** рассматриваемую программу можно протестировать двумя тестами, покрывающими либо пути: 1-2-4-6, 1-2-3-4-5-6, либо пути: 1-2-3-4-6, 1-2-4-5-6, например:

а = 3, b = 0, х = 3 — путь 1-2-3-4-5-6;

а = 2, b = 1, х = 1 — путь 1-2-4-5-6

Однако путь, где х не меняется, будет проверен с вероятностью 50 %: если во втором условии вместо х > 1 записано х < 1, то этими двумя тестами ошибка обнаружена не будет.

**Покрытие условий** проверяет четыре условия:

l) a>l; 2) b = 0; 3) а = 2; 4) х>1.

Необходимо реализовать все возможные ситуации:

Тесты, удовлетворяющие этому условию:

а *=* 2, b = 0, х = 4 — путь 1-2-3-4-5-6, условия: 1 - да, 2 - да, 3-да, 4-да

а = 1, b = 1, х = 1 — путь 1-2-4-6, условия: 1 - нет, 2 - нет, 3 – нет, 4-нет.

Критерий покрытия условий часто удовлетворяет критерию покрытия решений, но не всегда. Тесты критерия покрытия условий для ранее рассмотренных примеров покрывают результаты всех решений, но это случайное совпадение. Например, тесты:

а = 1, b = 0, х = 3 — путь 1-2-3-6, условия: 1 - нет, 2 - да, 3 - нет, 4 - да;

а = 2, b = 1, х = 1 — путь 1-2-3-4-5-6, условия: 1 - да, 2 - нет, 3 - да, 4 - нет

покрывают результаты всех условий, но только два из четырех результатов решений: не выполняется результат «истина» первого решения и результат «ложь» второго.

Основной недостаток метода – недостаточная чувствительность к ошибкам в логических выражениях.

**Комбинаторное покрытие условий** требует покрыть тестами восемь комбинаций:

1) а>1, b = 0; 5) а = 2, х>1;

2) а>1, b ≠ 0; 6) а = 2, х<1;

3) а<1, b = 0; 7) а ≠ 2, х>1;

4) а<1, b ≠ 0; 8) а ≠ 2, х<1.

Эти комбинации можно проверить четырьмя тестами:

а = 2, b = 0, х = 4 — проверяет комбинации (1), (5);

а = 2, b = 1, х = 1 — проверяет комбинации (2), (6);

а = 1, b = 0, х = 2 — проверяет комбинации (3), (7);

а = 1, b = 1, х = 1— проверяет комбинации (4), (8).

В данном случае то, что четырём тестам соответствуют четыре пути, является совпадением. Представленные тесты не покрывают всех путей, например, ACD. Поэтому иногда необходима реализация восьми тестов.

Таким образом, для программ, содержащих только одно условие на каждое решение, минимальным является набор тестов, который проверяет все результаты каждого решения и передает управление каждому оператору, по крайней мере, один раз.

Для программ, содержащих вычисления, каждое из которых требует проверки более чем одного условия, минимальный набор тестов должен:

─ генерировать все возможные комбинации результатов проверок условий для каждого вычисления;

─ передавать управление каждому оператору, по крайней мере, один раз.

Термин «возможных» употреблен здесь потому, что некоторые комбинации условий могут быть нереализуемы. Например, для комбинации k<0 и k>40 задать k невозможно.

**Тестовое покрытие циклов**

Цикл — наиболее распространенная конструкция алгоритмов, реализуемых в программном обеспечении. При проверке циклов основное внимание обращается на правильность конструкций циклов. Количество наборов тестов для проверки циклов по принципу «белого ящика» зависит от типа цикла. Различают 4 типа циклов: простые, вложенные, объединённые, неструктурированные.

Для проверки ***простых*** циклов с количеством повторений *n* может использоваться один из следующих наборов тестов:

1) прогон всего цикла;

2) только один проход цикла;

3) два прохода цикла;

4) *t* проходов цикла, где *t*<*n*;

5) *n* - 1, *n*, *n* + 1 проходов цикла.

С увеличением уровня **вложенности** циклов количество возможных путей резко возрастает. Это приводит к нереализуемому количеству тестов. Для сокращения количества тестов применяется специальная методика, использующая понятия объемлющего и вложенного циклов (рис. 5.3)



*Рис. 5.3 – Объемлющий и вложенный циклы*

Порядок тестирования вложенных циклов следующий:

* Выбирается самый внутренний цикл. Устанавливаются минимальные значения параметров всех остальных циклов.
* Для внутреннего цикла проводятся тесты простого цикла. Добавляются тесты для исключенных значений и значений, выходящих за пределы рабочего диапазона.
* Переходят в следующий по порядку объемлющий цикл. Выполняют его тестирование. При этом сохраняются минимальные значения параметров для всех внешних (объемлющих) циклов и типовые значения для всех вложенных циклов.
* Работа продолжается до тех пор, пока не будут протестированы все циклы.

Если каждый из циклов независим от других **(объединённые циклы)**, то используется техника тестирования простых циклов. При наличии зависимости (например, конечное значение счётчика первого цикла используется как начальное значение счётчика второго цикла) используется методика для вложенных циклов.

**Неструктурированные циклы** тестированию не подлежат. Этот тип циклов должен быть переделан с помощью структурированных программных конструкций.

**Задачи**

Составить блок-схему алгоритма процедуры и её потоковый граф (см. пример рис 5.2), составить тестовые наборы для тестирования маршрутов по критериям:

покрытия операторов;

покрытия решений (переходов);

покрытия условий;

покрытия решений/условий.

Составить отчёт, который содержит тему, цель работы, вариант, решение задачи и вывод.

***Варианты заданий для структурного тестирования***

Вариант 1.

procedure m(a,b: real; var x: real)

begin

if (a>0) and (b<0) then x:=x+1;

if ((a=2) or (x>3)) and (b>-10) then x:=x-1;

end;

Вариант 2.

procedure m(a,b,с: real; var x: real)

begin

if (a>0) and (b<0) and (x>6) then x:=x+1;

if (a=4) or (c<0) then x:=x+11;

end;

Вариант 3.

procedure m(a,b: real; var x: real)

begin

if (a<=6) and (b<0) then x:=x+1;

if (a=7) then x:=x-1

else if (x>3) then x=x\*2;

end;

Вариант 4.

procedure m(a,b: real; var x: real)

begin

if (a>0) then

if (b<0) then x:=x+1

else x=x\*2;

if (a>2) or (x=0) then x:=x+1;

end;

Вариант 5.

procedure m(a,b: real; var x: real)

begin

if (a>0) and (b<0) then x:=x+1

else if ((a=2) or (x>3)) and (b>-10) then x:=x-1;

end;

Вариант 6.

procedure m(a,b,с: real; var x: real)

begin

if (a>0) and (b<0) and (x>6)

then x:=x+1

else if (a=4) or (c<0)

then x:=x+11;

end;

**Контрольные вопросы**

1. Почему стратегия структурного тестирования называется также стратегией "белого ящика"?

2. Что показывает цикломатическая сложность алгоритма?

3. В чем отличие критериев покрытия условий и покрытия решений/условий

4. Какой из критериев "белого ящика" считается самым сильным и почему?

5. Приведите порядок тестирования вложенных циклов.

**Библиографический список**

1. Бейзер, Б. Тестирование черного ящика: технологии функционального тестирования программного обеспечения и систем / Б. Бейзер. - СПб.: Питер, 2004 . - 317 с.

2. Гагарина, Л. Г. Технология разработки программного обеспечения : учебное пособие / Л. Г. Гагарина, Е. В. Кокорева, Б. Д. Виснадул; под ред. Л. Г. Гагариной . - М. : ФОРУМ , 2011. - 399 с.

3. Дастин,Э. Автоматизированное тестирование программного обеспечения/ Э.Дастин, Д.Рэшка, Д.Пол - М.: Лори, 2003.-567с.

4. Иванова, Г.С. Технология программирования: учебник для вузов / Г.С.Иванова. - М.:КНОРУС, 2011.-333с.

5. Орлов, С. Технологии разработки программного обеспечения: учебник / С.Орлов. – СПб.:Питер,2002. – 464с.

6. Орлов, С. А. Технологии разработки программного обеспечения: современный курс по программной инженерии: учебник для вузов / С.А. Орлов, Б. Я. Цилькер. - СПб.: Питер, 2012 . - 608 с.

7. Плаксин, М.А. Тестирование и отладка программ - для профессионалов будущих и настоящих/ М.А.Плаксин. - М.:БИНОМ, 2007.-167с.

8. Сергушичева, А.П. Жизненный цикл программного продукта: учебное пособие / А.П. Сергушичева. − Вологда: ВоГТУ, 2010. − 148 с