ХАРЬКОВСКИЙ НАЦИОНАЛЬНИЙ УНИВЕРСИТЕТ
РАДИОЕЛЕКТРОНИКИ

Методические указания

к лабораторным работам и самостоятельной подготовке по дисциплине

«Современные технологии создания программных систем»

для студентов всех форм обучения
направления 6.050201 Системная инженерия
специальности 8.05020101 Компьютеризированные системы управления и автоматика

Электронное издание

Утверждено
кафедрой «Системотехники».
Протокол № 7/1 от 08.01.2014 р.

Харьков
2014

Методические указания к лабораторным работам и самостоятельной подготовке по дисциплине «Современные технологии создания программных систем» для студентов всех форм обучения направления 6.050201 Системная инженерия специальности 8.05020101 Компьютеризированные системы управления и автоматика [Электронное издание] / Учред. Ю.В. Мищеряков, Харьков: ХНУРЕ, 2014. – 47 с.

Учредитель: Ю.В. Мищеряков

Рецензент:

Оглавление

ОБЩИЕ ПОЛОЖЕНИЯ

Курс "Операционные системы" (ОС) для студентов направления "Системная инженерия" (СИ) является частью фундаментальной подготовки и вместе с тем началом специальной подготовки. Материалы этого курса используются при изучении множества последующих дисциплин, связанных с процессами обработки информации или с принципами функционирования ЭВМ.

Курс ОС состоит из следующих разделов:

– Системы контроля версий;

– Тестирование программных систем.

Значение операционных систем особенно возросло в связи с развитием компьютерной техники. Данная дисциплина дает теоретическую базу для проектирования программных комплексов, систем автоматизированного поиска информации, проектирования подсистем и элементов информационных систем. Это в свою очередь позволяет ставить и успешно решать проблемы, возникающие в теории и практике разработки программных систем.

В данных методических указаниях подобраны занятия с тем, чтобы закрепить знания и навыки по курсу. Для каждого занятия имеются задания, даны контрольные вопросы и указана необходимая литература.

1 GIT НАСТРОЙКА И УПРАВЛЕНИЕ РЕПОЗИТОРИЯМИ

1.1 Цель работы

Ознакомиться с правилами установки и настройки системы контроля версий, принципами создания и управления репозиториями.

1.2 Указания по подготовке к выполнению работы

При подготовке к работе необходимо ознакомится с теоретическим описанием принципов установки и настройки систем контроля версий, принципами создания и управления репозиториями. Внимательно проработать вопросы, связанные с настройкой контроля доступа.

При подготовке к работе необходимо изучить конспект лекций по указанной теме, методические указания, а также разделы, указанные в [1-3].

1.3 Обзор темы работы

Система контроля версий (СКВ) — это система, регистрирующая изменения в одном или нескольких файлах с тем, чтобы в дальнейшем была возможность вернуться к определённым старым версиям этих файлов. Обычно СКВ используются для контроля версий исходных кодов программ, но на самом деле под версионный контроль можно поместить файлы практически любого типа.

СКВ можно разделить на следующие виды:

- Локальные системы контроля версий (ЛСКВ)
- Централизованные системы контроля версий (ЦСКВ) например, CVS, Subversion и Perforce;
- Распределённые системы контроля версий (РСКВ) например, Git, Mercurial, Bazaar или Darcs.

ЦСКВ – есть центральный сервер, на котором хранятся все файлы под версионным контролем, и ряд клиентов, которые получают копии файлов из

него. Много лет это было стандартом для систем контроля версий (см. рисунок 1.1)

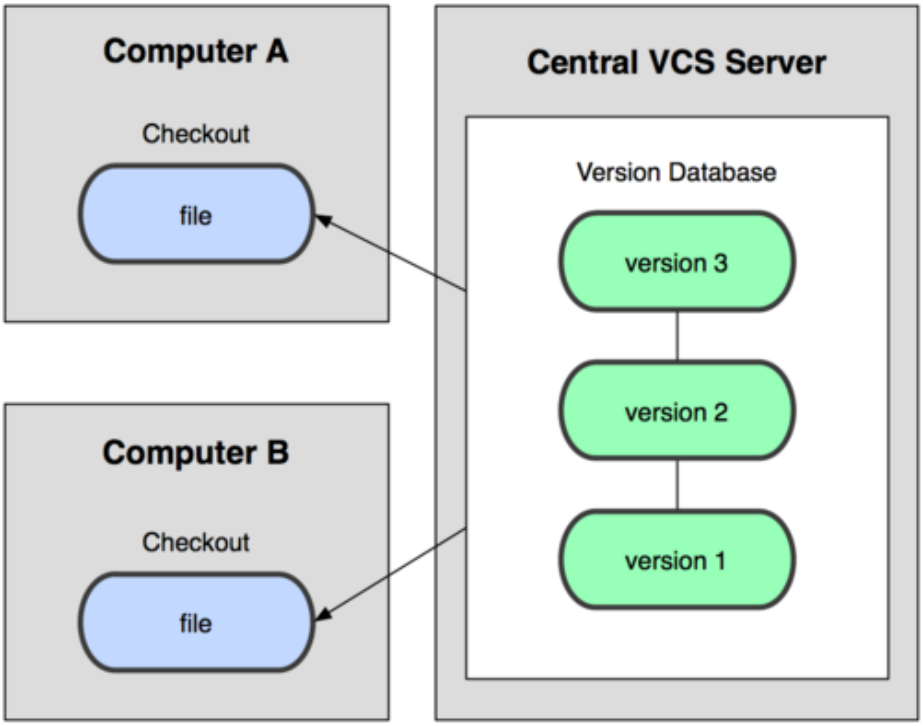

Рисунок 1.1 Схема централизованного контроля версий.

Однако при таком подходе есть и несколько серьёзных недостатков. Наиболее очевидный — централизованный сервер является уязвимым местом всей системы. Если сервер выключается на час, то в течение часа разработчики не могут взаимодействовать, и никто не может сохранить новой версии своей работы.

Локальные системы контроля версий подвержены той же проблеме: если вся история проекта хранится в одном месте, вы рискуете потерять всё.

РСКВ – клиенты не просто выгружают последние версии файлов, а полностью копируют весь репозиторий. Поэтому в случае, когда "умирает" сервер, через который шла работа, любой клиентский репозиторий может быть скопирован обратно на сервер, чтобы восстановить базу данных. Каждый раз, когда клиент забирает свежую версию файлов, он создаёт себе полную копию всех данных (см. рисунок 1.2).

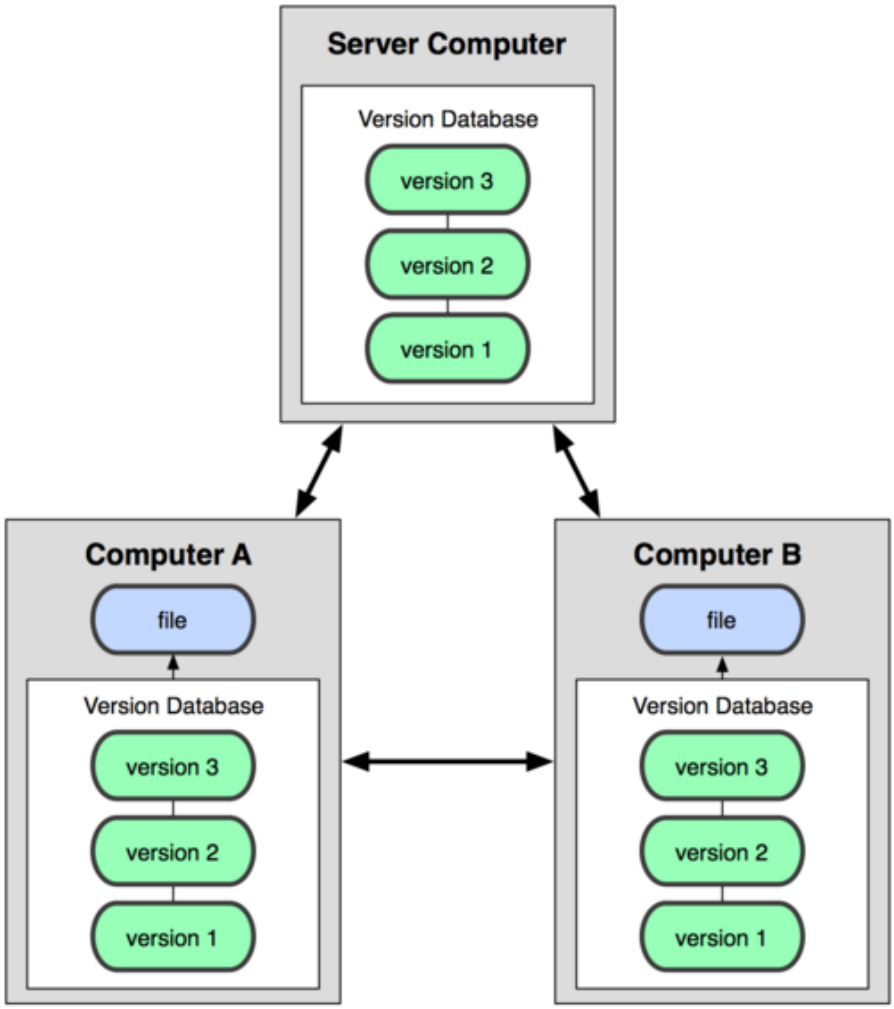

Рисунок 1.2 Схема распределённой системы контроля версий.

Принцип работы GIT

Git считает хранимые данные набором слепков небольшой файловой системы. Каждый раз, когда фиксируется текущая версия проекта, Git, по сути, сохраняет слепок того, как выглядят все файлы проекта на текущий момент. Ради эффективности, если файл не менялся, Git не сохраняет файл снова, а делает ссылку на ранее сохранённый файл (см. рисунок 1.3).

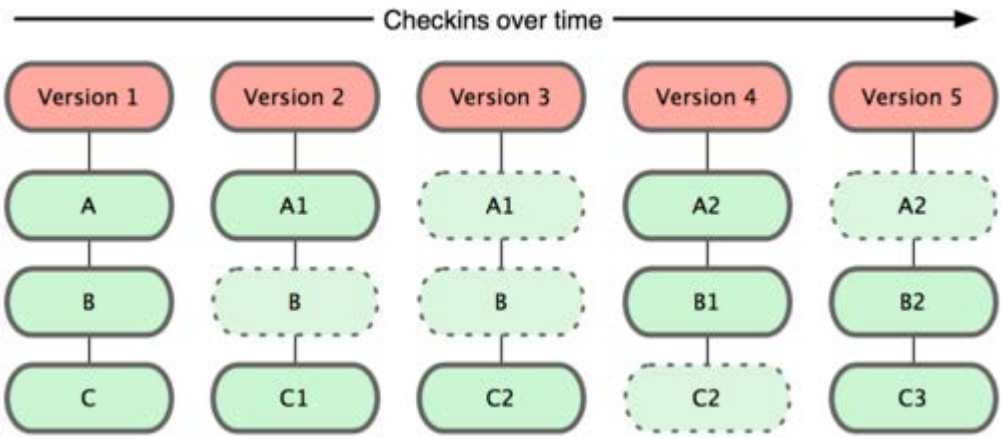

Рисунок 1.3 Git хранит данные как слепки состояний проекта во времени.

В Git'е файлы могут находиться в одном из трёх состояний: зафиксированном, изменённом и подготовленном. "Зафиксированный" значит, что файл уже сохранён в вашей локальной базе. К изменённым относятся файлы, которые поменялись, но ещё не были зафиксированы. Подготовленные файлы — это изменённые файлы, отмеченные для включения в следующий коммит.

Таким образом, в проектах, использующих Git, есть три части: каталог Git'a (Git directory), рабочий каталог (working directory) и область подготовленных файлов (staging area).

Каталог Git'a — это место, где Git хранит метаданные и базу данных объектов вашего проекта. Это наиболее важная часть Git'a, и именно она копируется, когда вы клонируете репозиторий с другого компьютера.

Рабочий каталог — это извлечённая из базы копия определённой версии проекта. Эти файлы достаются из сжатой базы данных в каталоге Git'a и помещаются на диск для того, чтобы вы их просматривали и редактировали.

Область подготовленных файлов — это обычный файл, обычно хранящийся в каталоге Git'a, который содержит информацию о том, что должно войти в следующий коммит. Иногда его называют индексом (index), но в последнее время становится стандартом называть его областью подготовленных файлов (staging area).

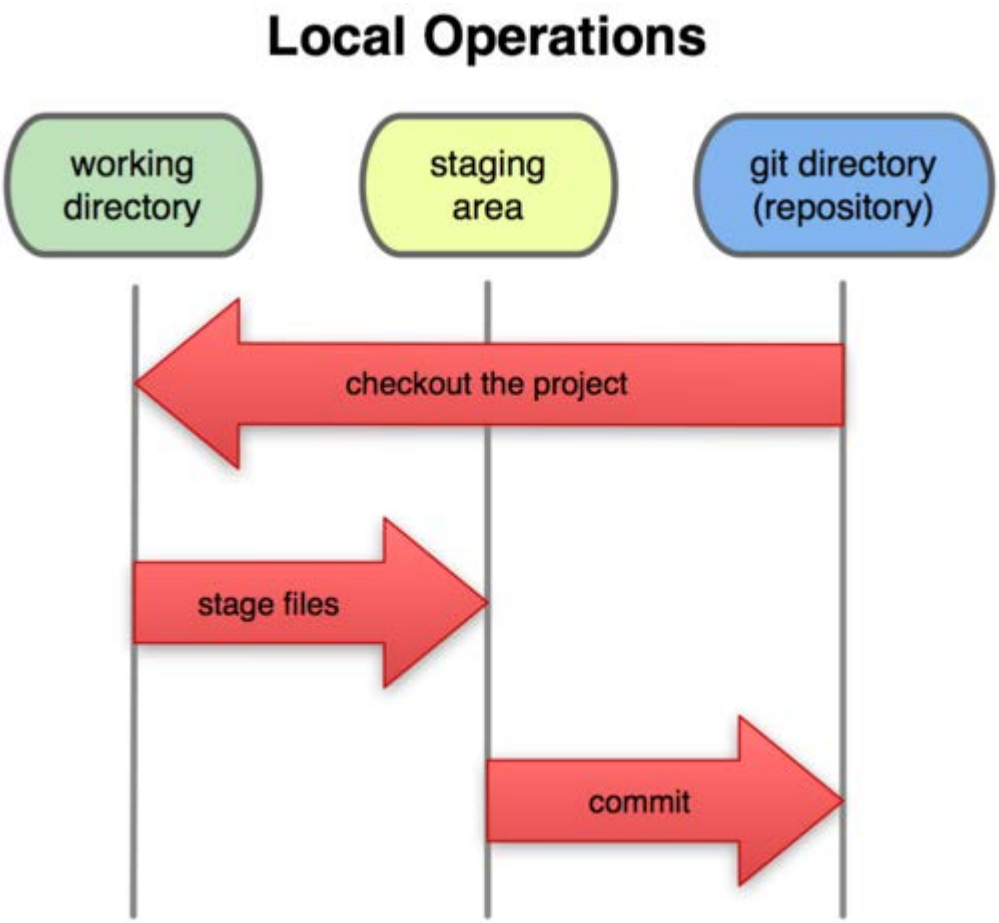

Рисунок 1.4 Рабочий каталог, область подготовленных файлов, каталог Git'a.

Стандартный рабочий процесс с использованием Git'a выглядит примерно так:

- Вы вносите изменения в файлы в своём рабочем каталоге.
- Подготавливаете файлы, добавляя их слепки в область подготовленных файлов.
- Делаете коммит, который берёт подготовленные файлы из индекса и помещает их в каталог Git'a на постоянное хранение.

Если рабочая версия файла совпадает с версией в каталоге Git'a, файл считается зафиксированным. Если файл изменён, но добавлен в область подготовленных данных, он подготовлен. Если же файл изменился после выгрузки из БД, но не был подготовлен, то он считается изменённым.

Первоначальная настройка Git

В состав Git'a входит утилита git config, которая позволяет просматривать и устанавливать параметры, контролирующие все аспекты работы Git'a и его внешний вид. Эти параметры могут быть сохранены в трёх местах:

Файл /etc/gitconfig содержит значения, общие для всех пользователей системы и для всех их репозиториев. Если при запуске git config указать параметр --system, то параметры будут читаться и сохраняться именно в этот файл.

Файл ~/.gitconfig хранит настройки конкретного пользователя. Этот файл используется при указании параметра --global.

Конфигурационный файл в каталоге Git'a (.git/config) в том репозитории, где вы находитесь в данный момент. Эти параметры действуют только для данного конкретного репозитория. Настройки на каждом следующем уровне подменяют настройки из предыдущих уровней, то есть значения в .git/config перекрывают соответствующие значения в /etc/gitconfig.

В системах семейства Windows Git ищет файл .gitconfig в каталоге $HOME (C:\Documents and Settings\$USER или C:\Users\$USER для большинства пользователей). Кроме того Git ищет файл /etc/gitconfig, но уже относительно корневого каталога MSys, который находится там, куда вы решили установить Git, когда запускали инсталлятор.

Первое, что следует сделать после установки Git'a, — указать имя пользователя и адрес электронной почты. Это важно, потому что каждый коммит в Git'e содержит эту информацию, и она включена в коммиты, передаваемые вами, и не может быть далее изменена:

```
$ git config --global user.name "John Doe"
$ git config --global user.email johndoe@example.com
```

Если указана опция `--global`, то эти настройки достаточно сделать только один раз, поскольку в этом случае Git будет использовать эти данные для всего, что вы делаете в этой системе. Если для каких-то отдельных проектов вы хотите указать другое имя или электронную почту, можно выполнить эту же команду без параметра `--global` в каталоге с нужным проектом.

Создание Git-репозитория

Для создания Git-репозитория существуют два основных подхода. Первый подход — импорт в Git уже существующего проекта или каталога. Второй — клонирование уже существующего репозитория с сервера.

Создание репозитория в существующем каталоге

Для существующего проекта необходимо перейти в проектный каталог и в командной строке ввести:

```
$ git init
```

Эта команда создаёт в текущем каталоге новый подкаталог с именем .git содержащий все необходимые файлы репозитория — основу Git-репозитория. На этом этапе проект ещё не находится под версионным контролем.

Необходимо добавить под версионный контроль существующие файлы (в отличие от пустого каталога), нужно проиндексировать эти файлы и осуществить первую фиксацию изменений. Осуществить это можно с помощью нескольких команд `git add` указывающих индексируемые файлы, а затем `commit`:

```
$ git add *.c
$ git add README
$ git commit -m 'initial project version'
```

Клонирование существующего репозитория

Клонирование репозитория с сревера осуществляется командой `git clone [url]`. Например, если вы хотите клонировать библиотеку Ruby Git, известную как Grit, вы можете сделать это следующим образом:

```
$ git clone git://github.com/schacon/grit.git
```

Эта команда создаёт каталог с именем `grit`, инициализирует в нём каталог `.git`, скачивает все данные для этого репозитория и создаёт (checks out) рабочую копию последней версии. Если зайти в новый каталог `grit`, вы увидите в нём проектные файлы, пригодные для работы и использования. Если вы хотите клонировать репозиторий в каталог, отличный от `grit`, можно это указать в следующем параметре командной строки:

```
$ git clone git://github.com/schacon/grit.git mygrit
```

Эта команда делает всё то же самое, что и предыдущая, только результирующий каталог будет назван mygrit.

В Git'е реализовано несколько транспортных протоколов, которые вы можете использовать. В предыдущем примере использовался протокол `git://`, вы также можете встретить `http(s)://` или `user@server:/path.git`, использующий протокол передачи SSH.

Жизненный цикл файлов в Git

После достижения некоторой стадии проекта Вам нужно фиксировать “снимки” состояния (snapshots).

Каждый файл в рабочем каталоге может находиться в одном из двух состояний: под версионным контролем (отслеживаемые) и нет (неотслеживаемые).

Отслеживаемые файлы — это те файлы, которые были в последнем слепке состояния проекта (snapshot); они могут быть неизменёнными, изменёнными или подготовленными к коммиту (staged).

Неотслеживаемые файлы — это всё остальное, любые файлы в рабочем каталоге, которые не входили в ваш последний слепок состояния и не подготовлены к коммиту.

Как только вы отредактируете файлы, Git будет рассматривать их как изменённые, т.к. вы изменили их с момента последнего коммита. Вы индексируете (stage) эти изменения и затем фиксируете все индексированные изменения, а затем цикл повторяется (см. рисунок 1.5).

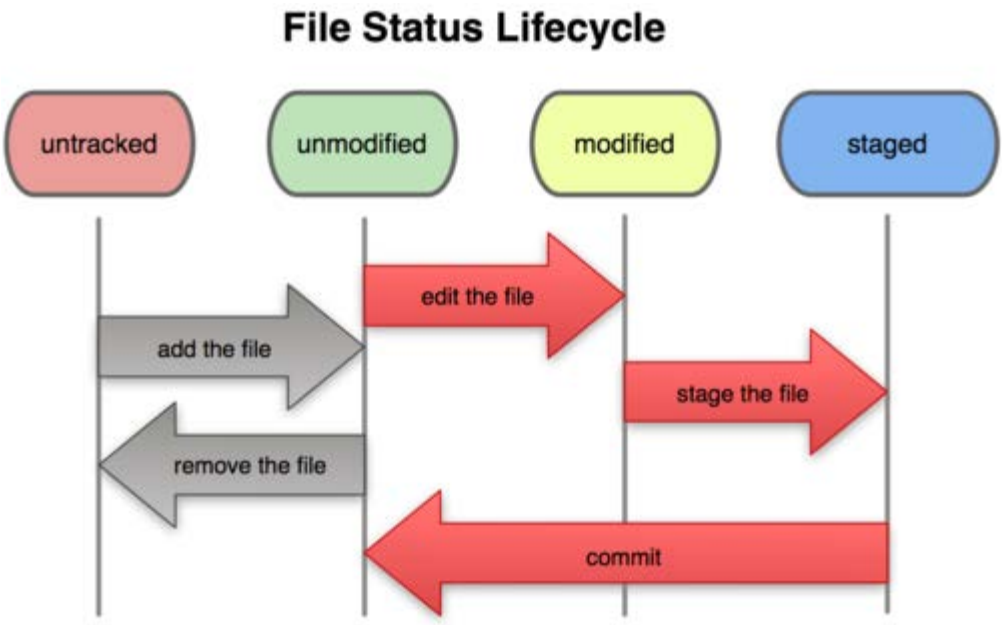

Рисунок 1.5 Жизненный цикл состояний файлов.

Индексация изменённых файлов

После модификации файлов проекта они должны быть заново проиндексированы командой `git add`.

С тем чтобы при индексации игнорировать часть файлов проекта можно создать файл .gitignore с перечислением шаблонов соответствующих таким файлам.

Фиксация изменений

Теперь, когда ваш индекс настроен так, как вам и хотелось, вы можете зафиксировать свои изменения. Запомните, всё, что до сих пор не проиндексировано — любые файлы, созданные или изменённые вами, и для которых вы не выполнили `git add` после момента редактирования — не войдут в этот коммит. Они останутся изменёнными файлами на вашем диске. Простейший способ зафиксировать изменения — это набрать `git commit`:

```
$ git commit
```

Эта команда откроет текстовый редактор в котором необходимо ввести комментарий к коммиту.

1.4 Задания к работе

1. Установить Git.
2. Создать собственный локальный репозиторий.
3. Добавить в него проект.
4. Создать файлы проекта, проиндексировать и сделать снимок.
5. Осуществить редактирование файлов проекта.
6. Заново проиндексировать и делать снимок.
7. Просмотреть историю снимков.

1.5 Содержание отчета к работе

Отчет должен содержать:

Цель работы, задание, распечатки вывода Git по всем выполняемым работам.

2 GIT ВЕТВЛЕНИЯ, СЛИЯНИЯ

2.1 Цель работы

Ознакомиться с правилами ветвления и слияния в распределенном репозитории.

2.2 Указания по подготовке к выполнению работы

При подготовке к работе необходимо ознакомится с теоретическим описанием принципов ветвления и слияния в распределенном репозитории. Внимательно проработать вопросы, связанные с ветвлением и слиянием в распределенном репозитории.

2.3 Обзор темы работы

Почти каждая СКВ имеет в какой-то форме поддержку ветвления. Ветвление означает, что вы отклоняетесь от основной линии разработки и продолжаете работу, не вмешиваясь в основную линию. Во многих СКВ это в некотором роде дорогостоящий процесс, зачастую требующий от вас создания новой копии каталога с исходным кодом, что может занять продолжительное время для больших проектов.

Ветка в Git'e — это просто легковесный подвижный указатель на один из этих коммитов. Ветка по умолчанию в Git'e называется `master`. Когда вы создаёте коммиты на начальном этапе, вам дана ветка `master`, указывающая на последний сделанный коммит. При каждом новом коммите она сдвигается вперёд автоматически (см. рисунок 2.1).

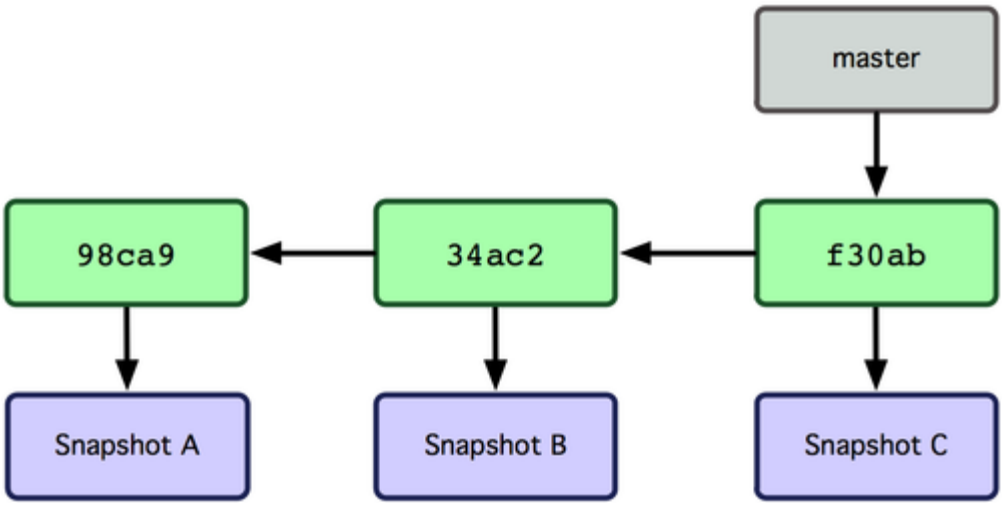

Рисунок 2.1 Ветка указывает на историю коммитов.

Что произойдёт, если создать новую ветку? Создание ветки осуществляется командой `git branch`:

```
$ git branch testing
```

Эта команда создаст новый указатель на тот самый коммит, на котором вы сейчас находитесь (см. рисунок 2.2).

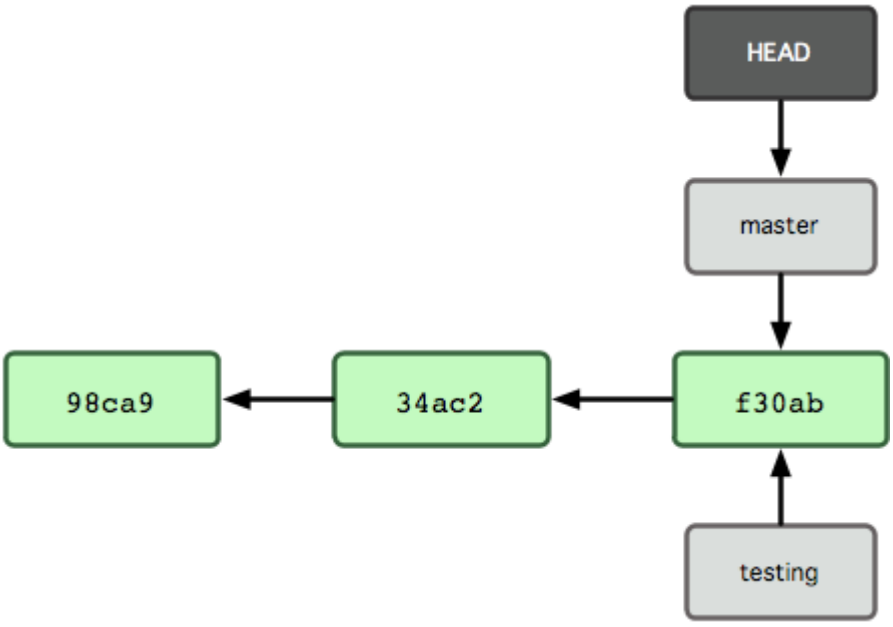

Рисунок 2.2 Файл HEAD указывает на текущую ветку.

Чтобы перейти на существующую ветку, надо выполнить команду `git checkout`.

```
$ git checkout testing
```

Это действие передвинет HEAD так, чтобы тот указывал на ветку `testing` (см. рисунок 2.3).

```
$ git commit -a -m 'made a change'
```

На рисунке 3-7 показан результат.

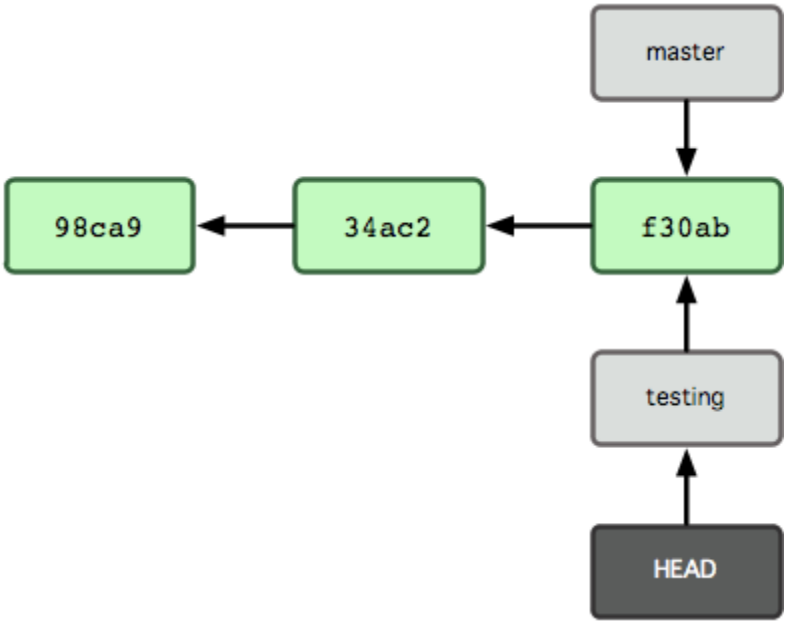

Рисунок 2.3 HEAD указывает на другую ветку после переключения веток.

Выполним ещё один коммит (см. рисунок 2.4):

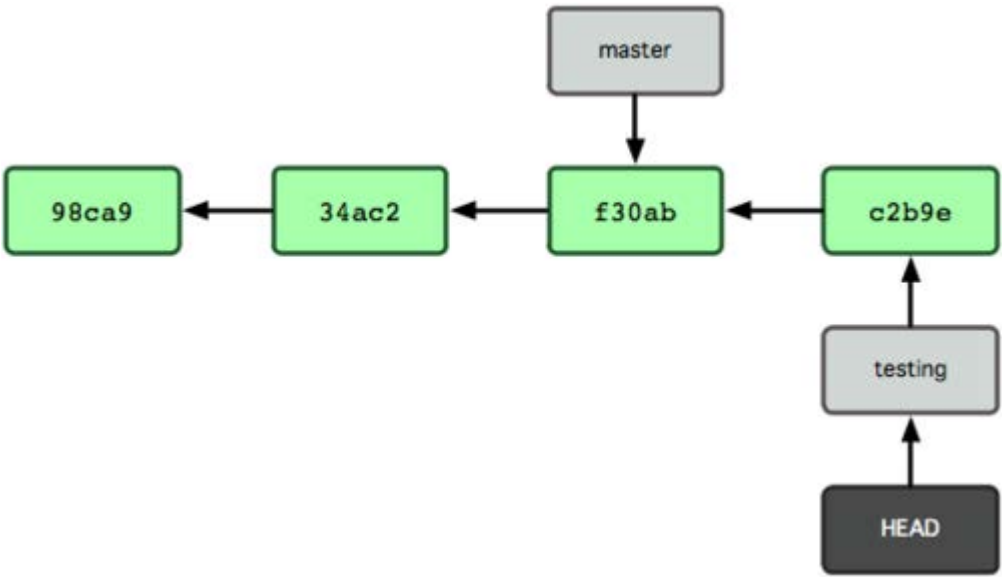

Рисунок 2.4 Ветка, на которую указывает HEAD, движется вперёд с каждым коммитом.

Ветка testing передвинулась вперёд, но ветка master всё ещё указывает на коммит, на котором вы были, когда выполняли git checkout, чтобы переключить ветки. Давайте перейдём обратно на ветку master (см. рисунок 2.5рисунок 2.4):

```
$ git checkout master
```

Эта команда выполнила два действия. Она передвинула указатель HEAD назад на ветку master и вернула файлы в вашем рабочем каталоге назад, в соответствие со снимком состояния, на который указывает master. Это также означает, что изменения, которые вы делаете, начиная с этого момента, будут ответвляться от старой версии проекта. Это, по сути, откатывает изменения, которые вы временно делали на ветке testing, так что дальше вы можете двигаться в другом направлении.

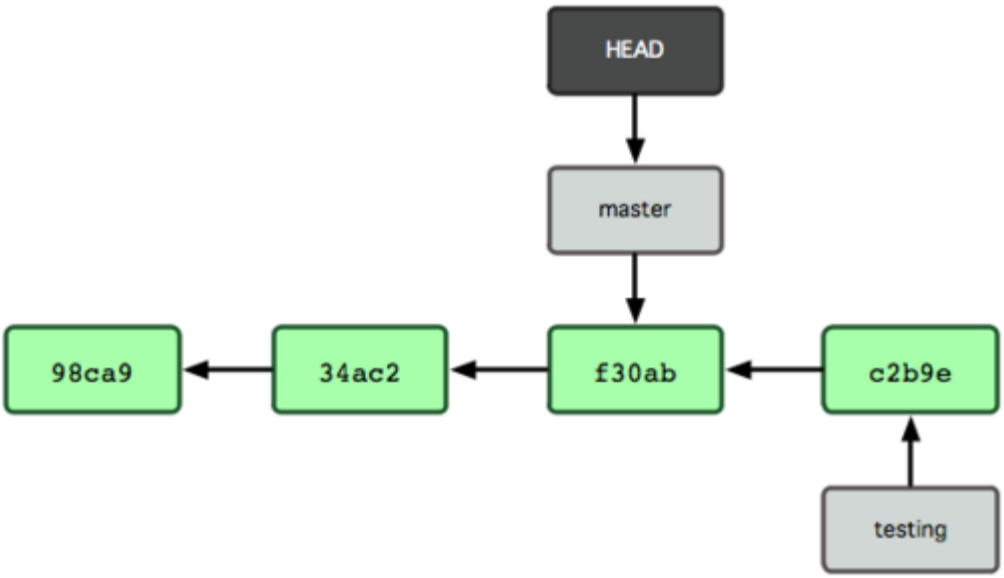

Рисунок 2.5 HEAD перемещается на другую ветку при checkout'e.

Теперь история проекта разветвилась (см. рис. 3-9). Вы создали новую ветку, перешли на неё, поработали на ней немного, переключились обратно на основную ветку и выполнили другую работу. Оба эти изменения изолированы в отдельных ветках: вы можете переключаться туда и обратно между ветками и слить их, когда будете готовы. И всё это было сделано простыми командами branch и checkout.

2.4 Задания к работе

1. Выполнить `checkout` для своего проекта из работы 1.
2. Разветвить проект, создав ветку `fix`.
3. Выполнить `checkout` для созданной ветки.
4. Внести изменения в проект и сделать `commit` ветки `fix`.
5. Создать новую ветку от `master` с именем `hotfix`, выполнить изменения в файлах, не затронутых веткой fix. Сделать `commit` ветки `hotfix`.
6. Осуществить слияние ветки `hotfix` с веткой `master` и удалить ветку `hotfix`.
7. Вновь загрузить ветку `fix`, сделать в ней изменения и выполнить commit.
8. Осуществить слияние ветки `fix` с веткой `master` и удалить ветку `fix`.
9. Повторить шаги 1-9 только в шаге 5 осуществить редактирование файлов затронутых веткой `fix`.
10. Пояснить результаты.

2.5 Содержание отчета к работе

Отчет должен содержать:

Цель работы, задание, распечатки вывода Git по всем выполняемым работам, выводы.

3 ТЕСТИРОВАНИЕ ПРОГРАММ МЕТОДАМИ «ЧЕРНОГО ЯЩИКА»

3.1 Цель работа:

Проанализировать методы тестирования программных продуктов, оценить различные методы с точки зрения детективности и покрывающей способности тестов

Лабораторная работа рассчитана на 4 академических часа.

3.2 Указания по подготовке к выполнению работы

Ознакомиться с лекционным материалом по теме "Тестирование программ" учебной дисциплины "Технология разработки программного обеспечения".

Изучить соответствующие разделы в литературе, приведенной в методических описаниях к лабораторной работе.

3.3 Обзор темы работы

Тестирование по принципу «черного ящика»

Одним из способов проверки программ является стратегия тестирования, называемая стратегией "черного ящика" или тестированием с управлением по данным. В этом случае программа рассматривается как "черный ящик" и такое тестирование имеет целью выяснения обстоятельств, в которых поведение программы не соответствует спецификации.

Для обнаружения всех ошибок в программе необходимо выполнить *исчерпывающее тестирование*, т.е. тестирование на всех возможных наборах данных. Для тех же программ, где исполнение команды зависит от предшествующих ей событий, необходимо проверить и все возможные последовательности.

Очевидно, что построение исчерпывающего входного теста для большинства случаев невозможно. Поэтому, обычно выполняется *"разумное"*

тестирование, при котором тестирование программы ограничивается прогонами на небольшом подмножестве всех возможных входных данных. Естественно при этом целесообразно выбрать наиболее подходящее подмножество (подмножество с наивысшей вероятностью обнаружения ошибок).

Правильно выбранный тест подмножества должен обладать следующими свойствами:

1) уменьшать, причем более чем на единицу число других тестов, которые должны быть разработаны для достижения заранее определенной цели «приемлемого» тестирования:

2) покрывать значительную часть других возможных тестов, что в некоторой степени свидетельствует о наличии или отсутствии ошибок до и после применения этого ограниченного множества значений входных данных.

Стратегия "черного ящика" включает в себя следующие методы формирования тестовых наборов:

- эквивалентное разбиение;
- анализ граничных значений;
- анализ причинно-следственных связей;
- предположение об ошибке.

3.3.1 Эквивалентное разбиение

Основу метода составляют два положения:

1. Исходные данные программы необходимо разбить на конечное число классов эквивалентности, так чтобы можно было предположить, что каждый тест, являющийся представителем некоторого класса, эквивалентен любому другому тесту этого класса. Иными словами, если тест какого-либо класса обнаруживает ошибку, то предполагается, что все другие тесты этого класса эквивалентности тоже обнаружат эту ошибку и наоборот

2. Каждый тест должен включать по возможности максимальное количество различных входных условий, что позволяет минимизировать общее число необходимых тестов.

Первое положение используется для разработки набора "интересных" условий, которые должны быть протестированы, а второе - для разработки минимального набора тестов.

Разработка тестов методом эквивалентного разбиения осуществляется в два этапа:

– выделение классов эквивалентности;

– построение тестов.

Выделение классов эквивалентности

Классы эквивалентности выделяются путем выбора каждого входного условия (обычно это предложение или фраза из спецификации) и разбиением его на две или более групп. Для этого используется таблица следующего вида:

Входное условие	Правильные классы эквивалентности	Неправильные классы эквивалентности

Правильные классы включают правильные данные, неправильные классы - неправильные данные.

Выделение классов эквивалентности является эвристическим процессом, однако при этом существует ряд правил:

– Если входные условия описывают *область* значений (например «целое данное может принимать значения от 1 до 999»), то выделяют один правильный класс $1 \le X \le 999$ и два неправильных $X<1$ и $X>999$.

– Если входное условие описывает *число* значений (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, «известные способы передвижения на АВТОБУСЕ, ГРУЗОВИКЕ, ТАКСИ, МОТОЦИКЛЕ или ПЕШКОМ»), то определяется правильный класс эквивалентности для каждого значения и один неправильный класс (например «на ПРИЦЕПЕ»).

– 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 любое основание считать, что различные элементы класса эквивалентности трактуются программой неодинаково, то данный класс разбивается на меньшие классы эквивалентности.

Построение тестов

Этот шаг заключается в использовании классов эквивалентности для построения тестов. Этот процесс включает в себя:

– Назначение каждому классу эквивалентности уникального номера.

– Проектирование новых тестов, каждый из которых покрывает как можно большее число непокрытых классов эквивалентности, до тех пор, пока все правильные классы не будут покрыты (только не общими) тестами.

– Запись тестов, каждый из которых покрывает один и только один из непокрытых неправильных классов эквивалентности, до тех пор, пока все неправильные классы не будут покрыты тестами.

Разработка индивидуальных тестов для неправильных классов эквивалентности обусловлено тем, что определенные проверки с ошибочными входами скрывают или заменяют другие проверки с ошибочными входами.

Недостатком метода эквивалентных разбиения в том, что он не исследует комбинации входных условий.

3.3.2 Анализ граничных значений.

Граничные условия - это ситуации, возникающие на, выше или ниже границ входных классов эквивалентности. Анализ граничных значений отличается от эквивалентного разбиения следующим:

– Выбор любого элемента в классе эквивалентности в качестве представительного при анализе граничных условий осуществляется таким образом, чтобы проверить тестом каждую границу этого класса.

– При разработке тестов рассматриваются не только входные условия (*пространство входов*), но и *пространство результатов*.

Применение метода анализа граничных условий требует определенной степени творчества и специализации в рассматриваемой проблеме. Тем не менее, существует несколько общих правил этого метода:

– Построить тесты для границ области и тесты с неправильными входными данными для ситуаций незначительного выхода за границы области, если входное условие описывает область значений (например, для области входных значений от -1.0 до +1.0 необходимо написать тесты для ситуаций -1.0, +1.0, -1.001 и +1.001).

– Построить тесты для минимального и максимального значений условий и тесты, большие и меньшие этих двух значений, если входное условие удовлетворяет дискретному ряду значений. Например, если входной файл может содержать от 1 до 255 записей, то проверить 0, 1, 255 и 256 записей.

– Использовать правило 1 для каждого выходного условия. Причем, важно проверить границы пространства результатов, поскольку не всегда границы входных областей представляют такой же набор условий, как и границы выходных областей. Не всегда также можно получить результат вне выходной области, но, тем не менее, стоит рассмотреть эту возможность.

– Использовать правило 2 для каждого выходного условия.

– Если вход или выход программы есть упорядоченное множество (например, последовательный файл, линейный список, таблица), то сосредоточить внимание на первом и последнем элементах этого множества.

– Попробовать свои силы в поиске других граничных условий.

Анализ граничных условий, если он применен правильно, является одним из наиболее полезных методов проектирования тестов. Однако следует помнить, что граничные условия могут быть едва уловимы и определение их связано с большими трудностями, что является недостатком этого метода. Второй недостаток связан с тем, что метод анализа граничных условий не позволяет проверять различные сочетания исходных данных.

3.3.3 Анализ причинно-следственных связей.

Метод анализа причинно-следственных связей помогает системно выбирать высокорезультативные тесты. Он дает полезный побочный эффект, позволяя обнаруживать неполноту и неоднозначность исходных спецификаций.

Для использования метода необходимо понимание булевской логики (логических операторов - и, или, не). Построение тестов осуществляется в несколько этапов.

1) Спецификация разбивается на «рабочие » участки, так как таблицы причинно-следственных связей становятся громоздкими при применении метода к большим спецификациям. Например, при тестировании компилятора в качестве рабочего участка можно рассматривать отдельный оператор языка.

2) В спецификации определяются множество причин и множество следствий. *Причина* есть отдельное входное условие или класс эквивалентности входных условий. *Следствие* есть выходное условие или преобразование системы. Каждым причине и следствию приписывается отдельный номер.

3) На основе анализа семантического (смыслового) содержания спецификации строится таблица истинности, в которой последовательно перебираются все возможные комбинации причин и определяются следствия каждой комбинации причин. Таблица снабжается примечаниями, задающими ограничения и описывающими комбинации причин и/или следствий, которые являются невозможными из-за синтаксических или внешних ограничений. Аналогично, при необходимости строится таблица истинности для класса эквивалентности.

Примечание. При этом можно использовать следующие приемы:

– По возможности выделять независимые группы причинно-следственных связей в отдельные таблицы.

– Истина обозначается "1". Ложь обозначается "0". Для обозначения безразличных состояний условий применять обозначение "X", которое предполагает произвольное значение условия (0 или 1).

4) Каждая строка таблицы истинности преобразуется в тест. При этом по возможности следует совмещать тесты из независимых таблиц.

Недостаток метода - неадекватно исследует граничные условия.

3.3.4 Предположение об ошибке.

Часто программист с большим опытом выискивает ошибки "без всяких методов". При этом он подсознательно использует метод "предположение об ошибке". Процедура метода предположения об ошибке в значительной степени основана на интуиции. Основная идея метода состоит в том, чтобы перечислить в некотором списке возможные ошибки или ситуации, в которых они могут появиться, а затем на основе этого списка составить тесты. Другими словами, требуется перечислить те специальные случаи, которые могут быть не учтены при проектировании.

3.4 Задания к работе

1. Разработать тесты для своего проекта.
2. Запустить тесты на выполнение.
3. Оценить покрытие тестами возможных наборов данных.
4. Добиться максимально возможного покрытия наборов данных.

3.5 Содержание отчета к работе

Отчет должен содержать:

Цель работы, задание, разработанные тесты, степень покрытия для задания.

4 МОДУЛЬНОЕ ТЕСТИРОВАНИЕ

4.1 Цель работы

Ознакомиться с процессом модульного тестирования. Научиться разрабатывать модульные тесты.

4.2 Указания по подготовке к выполнению работы

При подготовке к работе необходимо ознакомится с теоретическим описанием принципов тестирования, разработки модульных тестов, принципами обеспечения максимального покрытия кода.

4.3 Обзор темы работы

Модульное тестирование

Модульное тестирование - это тестирование программы на уровне отдельно взятых модулей, функций или классов. Цель *модульного тестирования* состоит в выявлении локализованных в модуле ошибок в реализации алгоритмов, а также в определении степени готовности системы к переходу на следующий уровень разработки и тестирования. *Модульное тестирование* проводится по принципу "белого ящика", то есть основывается на знании внутренней структуры программы, и часто включает те или иные методы анализа покрытия кода.

Модульное тестирование обычно подразумевает создание вокруг каждого модуля определенной среды, включающей заглушки для всех интерфейсов тестируемого модуля. Некоторые из них могут использоваться для подачи входных значений, другие для анализа результатов, присутствие третьих может быть продиктовано требованиями, накладываемыми компилятором и сборщиком.

На уровне *модульного тестирования* проще всего обнаружить дефекты, связанные с алгоритмическими ошибками и ошибками кодирования

алгоритмов, типа работы с условиями и счетчиками циклов, а также с использованием локальных переменных и ресурсов. Ошибки, связанные с неверной трактовкой данных, некорректной реализацией интерфейсов, совместимостью, производительностью и т.п. обычно пропускаются на уровне *модульного тестирования* и выявляются на более поздних стадиях тестирования.

Именно эффективность обнаружения тех или иных типов дефектов должна определять стратегию *модульного тестирования*, то есть расстановку акцентов при определении набора входных значений. У организации, занимающейся разработкой программного обеспечения, как правило, имеется историческая база данных (**Repository**) разработок, хранящая конкретные сведения о разработке предыдущих проектов: о версиях и сборках кода (**build**) зафиксированных в процессе разработки продукта, о принятых решениях, допущенных просчетах, ошибках, успехах и т.п. Проведя анализ характеристик прежних проектов, подобных заказанному организации, можно предохранить новую разработку от старых ошибок, например, определив типы дефектов, поиск которых наиболее эффективен на различных этапах тестирования.

В данном случае анализируется этап *модульного тестирования*. Если анализ не дал нужной информации, например, в случае проектов, в которых соответствующие данные не собирались, то основным правилом становится поиск локальных дефектов, у которых код, ресурсы и информация, вовлеченные в дефект, характерны именно для данного модуля. В этом случае на *модульном уровне* ошибки, связанные, например, с неверным порядком или форматом параметров модуля, могут быть пропущены, поскольку они вовлекают информацию, затрагивающую другие модули (а именно, спецификацию интерфейса), в то время как ошибки в алгоритме обработки параметров довольно легко обнаруживаются.

Являясь по способу исполнения структурным тестированием или тестированием "белого ящика", *модульное тестирование* характеризуется степенью, в которой тесты выполняют или покрывают логику программы (исходный текст). Тесты, связанные со структурным тестированием, строятся по следующим принципам:

– На основе анализа *потока управления*. В этом случае элементы, которые должны быть покрыты при прохождении тестов, определяются на

основе *структурных критериев* тестирования C0, C1,C2. К ним относятся вершины, дуги, пути *управляющего графа* программы (УГП), условия, комбинации условий и т. п.

- На основе анализа *потока данных*, когда элементы, которые должны быть покрыты, определяются при помощи *потока данных*, т. е. *информационного графа* программы.

Тестирование на основе потока управления. К особенностям использования *структурных критериев* тестирования C0, C1, C2 следует добавить критерий покрытия условий, заключающийся в покрытии всех логических (булевских) условий в программе. Критерии покрытия решений (ветвей - C1) и условий не заменяют друг друга, поэтому на практике используется комбинированный критерий покрытия условий/решений, совмещающий требования по покрытию и решений, и условий.

К популярным критериям относятся критерий покрытия функций программы, согласно которому каждая функция программы должна быть вызвана хотя бы один раз, и критерий покрытия вызовов, согласно которому каждый вызов каждой функции в программе должен быть осуществлен хотя бы один раз. Критерий покрытия вызовов известен также как критерий покрытия пар вызовов (call pair *coverage*).

Тестирование на основе потока данных. Этот вид тестирования направлен на выявление ссылок на неинициализированные переменные и избыточные присваивания (аномалий *потока данных*). Как основа для стратегии тестирования *поток данных* впервые был описан в [14] . Предложенная там стратегия требовала тестирования всех взаимосвязей, включающих в себя ссылку (использование) и определение переменной, на которую указывает ссылка (т. е. требуется покрытие дуг *информационного графа* программы). Недостаток стратегии в том, что она не включает критерий C1, и не гарантирует покрытия решений.

Стратегия требуемых пар [15] также тестирует упомянутые взаимосвязи. Использование переменной в предикате дублируется в соответствии с числом выходов решения, и каждая из таких требуемых взаимосвязей должна быть протестирована. К популярным критериям принадлежит критерий CP, заключающийся в покрытии всех таких пар дуг v и w, что из дуги v достижима дуга w, поскольку именно на дуге может произойти потеря значения переменной, которая в дальнейшем уже не

должна использоваться. Для "покрытия" еще одного популярного критерия Cdu достаточно тестировать пары (вершина, дуга), поскольку определение переменной происходит в вершине УГП, а ее использование - на дугах, исходящих из решений, или в вычислительных вершинах.

Методы проектирования тестовых путей для достижения заданной степени тестированности в структурном тестировании. Процесс построения набора тестов при структурном тестировании принято делить на три фазы:

- Конструирование УГП.
- Выбор тестовых путей.
- Генерация тестов, соответствующих тестовым путям.

Первая фаза соответствует статическому анализу программы, задача которого состоит в получении графа программы и зависящего от него и от критерия тестирования множества элементов, которые необходимо покрыть тестами.

На третьей фазе по известным путям тестирования осуществляется поиск подходящих тестов, реализующих прохождение этих путей.

Вторая фаза обеспечивает выбор тестовых путей. Выделяют три подхода к построению тестовых путей:

- *Статические методы*.
- *Динамические методы*.
- Методы Интеграционное тестирование

Интеграционное тестирование

Интеграционное тестирование - это тестирование части системы, состоящей из двух и более модулей. Основная задача *интеграционного тестирования* - поиск дефектов, связанных с ошибками в реализации и интерпретации интерфейсного взаимодействия между модулями.

С технологической точки зрения *интеграционное тестирование* является количественным развитием *модульного*, поскольку так же, как и *модульное тестирование*, оперирует интерфейсами модулей и подсистем и требует создания тестового окружения, включая заглушки (**Stub**) на месте отсутствующих модулей. Основная разница между *модульным* и *интеграционным тестированием* состоит в целях, то есть в типах обнаруживаемых дефектов, которые, в свою очередь, определяют стратегию

выбора входных данных и методов анализа. В частности, на уровне *интеграционного тестирования* часто применяются методы, связанные с покрытием интерфейсов, например, вызовов функций или методов, или анализ использования интерфейсных объектов, таких как глобальные ресурсы, средства коммуникаций, предоставляемых операционной системой.

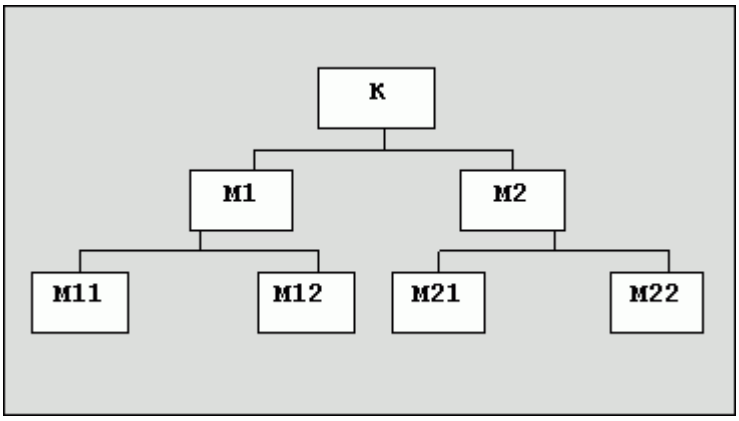

Рис. 5.1. Пример структуры комплекса программ

На Рис. 5.1 приведена структура комплекса программ K, состоящего из оттестированных на этапе *модульного тестирования* модулей M1, M2, M11, M12, M21, M22. Задача, решаемая методом *интеграционного тестирования*, - тестирование межмодульных связей, реализующихся при исполнении программного обеспечения комплекса K. *Интеграционное тестирование* использует модель "белого ящика" на модульном уровне. Поскольку тестировщику текст программы известен с детальностью до вызова всех модулей, входящих в тестируемый комплекс, применение структурных критериев на данном этапе возможно и оправдано.

Интеграционное тестирование применяется на этапе сборки модульно оттестированных модулей в единый комплекс. Известны два метода *сборки модулей*:

- **Монолитный**, характеризующийся одновременным объединением всех модулей в тестируемый комплекс
- **Инкрементальный**, характеризующийся пошаговым (помодульным) наращиванием комплекса программ с **пошаговым тестированием** собираемого комплекса. В инкрементальном методе выделяют две стратегии добавления модулей:

1) "Сверху вниз" и соответствующее ему *восходящее тестирование*.

2) "Снизу вверх" и соответственно *нисходящее тестирование*.

Особенности монолитного тестирования заключаются в следующем: для замены неразработанных к моменту тестирования модулей, кроме самого верхнего (К на Рис. 5.1), необходимо дополнительно разрабатывать **драйверы (test driver)** и/или **заглушки (stub)** [9], замещающие отсутствующие на момент сеанса тестирования модули нижних уровней.

Сравнение *монолитного* и инкрементального подхода дает следующее:

– *Монолитное тестирование* требует больших трудозатрат, связанных с дополнительной разработкой драйверов и заглушек и со сложностью идентификации ошибок, проявляющихся в пространстве собранного кода.

– Пошаговое тестирование связано с меньшей трудоемкостью идентификации ошибок за счет постепенного наращивания объема тестируемого кода и соответственно локализации добавленной области тестируемого кода.

– *Монолитное тестирование* предоставляет большие возможности распараллеливания работ особенно на начальной *фазе тестирования*.

Особенности *нисходящего тестирования* заключаются в следующем: организация среды для исполняемой очередности вызовов оттестированными модулями тестируемых модулей, постоянная разработка и использование заглушек, организация приоритетного тестирования модулей, содержащих операции обмена с окружением, или модулей, критичных для тестируемого алгоритма.

Например, порядок тестирования комплекса К (Рис. 5.1) при *нисходящем тестировании* может быть таким, как показано в примере 5.3, где тестовый набор, разработанный для модуля M_i, обозначен как $XY_i = (X, Y)_i$

---

1) $K->XY_K$
2) $M_1->XY_1$
3) $M_{11}->XY_{11}$
4) $M_2->XY_2$
5) $M_{22}->XY_{22}$
6) $M_{21}->XY_{21}$
7) $M_{12}->XY_{12}$

Пример 5.3. Возможный порядок тестов при нисходящем тестировании

Недостатки *нисходящего тестирования*:

- Проблема разработки достаточно "интеллектуальных" заглушек, т.е. заглушек, пригодных к использованию при моделировании различных режимов работы комплекса, необходимых для тестирования
- Сложность организации и разработки среды для реализации исполнения модулей в нужной последовательности
- Параллельная разработка модулей верхних и нижних уровней приводит к не всегда эффективной реализации модулей из-за подстройки (специализации) еще не тестированных модулей нижних уровней к уже оттестированным модулям верхних уровней

Особенности восходящего тестирования в организации порядка *сборки* и перехода к тестированию модулей, соответствующему порядку их реализации.

Например, порядок тестирования комплекса K (Рис. 5.1) при *восходящем тестировании* может быть следующим (пример. 5.4).

---

1) M_{11}->XY_{11}
2) M_{12}->XY_{12}
3) M_{1}->XY_{1}
4) M_{21}->XY_{21}
5) $M_{2}(M_{21}, Stub(M_{22}))$->$XY_{2}$
6) $K(M_{1}, M_{2}(M_{21}, Stub(M_{22}))$ ->XY_{K}
7) M_{22}->XY_{22}
8) M_{2}->XY_{2}
9) K->XY_{K}

Пример 5.4. Возможный порядок тестов при восходящем тестировании

Недостатки *восходящего тестирования*:

- Запаздывание проверки концептуальных особенностей тестируемого комплекса
- Необходимость в разработке и использовании драйверов

4.4 Задания к работе

5. Разработать модульные тесты для своего проекта.
6. Запустить тесты на выполнение.

7. Оценить покрытие тестами кода.
8. Добиться максимально возможного покрытия кода.

4.5 Содержание отчета к работе

Отчет должен содержать:

Цель работы, задание, разработанные тесты, степень покрытия для каждого модуля.

5 АВТОМАТИЗАЦИЯ ТЕСТИРОВАНИЯ

5.1 Цель работы

Ознакомиться со средствами автоматизации тестирования. Научиться разрабатывать сценарии тестирования.

5.2 Указания по подготовке к выполнению работы

При подготовке к работе необходимо ознакомится с теоретическим описанием принципов тестирования, разработки сценариев автоматизации тестов, принципами обеспечения максимального покрытия кода.

5.3 Обзор темы работы

Использование различных подходов к тестированию определяется их эффективностью применительно к условиям, определяемым промышленным проектом. В реальных случаях работа группы тестирования планируется так, чтобы разработка тестов начиналась с момента согласования требований к программному продукту (выпуск Requirement Book, содержащей высокоуровневые требования к продукту) и продолжалась параллельно с разработкой дизайна и кода продукта. В результате, к началу системного тестирования создаются тестовые наборы, содержащие тысячи тестов. Большой набор тестов обеспечивает всестороннюю проверку функциональности продукта и гарантирует качество продукта, но пропуск такого количества тестов на этапе системного тестирования представляет проблему. Ее решение лежит в области автоматизации тестирования, т.е. в автоматизации разработки.

```
Структура программы P теста
Загрузка теста (X,Y*)
Запуск тестируемого модуля
Сравнение полученных результатов Y с эталонными Y*
```

```
Структура тестируемого комплекса
ModF <-  МодF1
         МодF2
         МодF3 <- МодF31
                  МодF32
Структура тестирующего модуля
Mod  TestModF:
Mod TestМодF1
Мod TestМодF2
Мod TestМодF3
P TestМодF

Мod  TestModF1:
P TestМодF1

Мod  TestModF2:
P TestМодF2

Мod  TestModF3:
Mod TestМодF31
Мod TestМодF32
P TestМодF3
```

В этом примере приведены структура теста, структура тестируемого комплекса и структура тестирующего модуля. Особенностью структуры каждого из тестирующих модулей Mi является запуск тестирующей программы Pi после того как каждый из модулей Mij, входящих в контекст модуля Mi, оттестирован. В этом случае запуск тестирующего модуля обеспечивает рекурсивный спуск к программам тестирования модулей нижнего уровня, а затем исполняет тестирование вышележащих уровней в условиях оттестированности нижележащих. Тестовые наборы подобной структуры ориентированы на автоматическое управление пропуском тестового набора в тестовом цикле. Важным преимуществом подобной организации является возможность регулирования нижнего уровня, до которого следует доходить в цикле тестирования. В этом случае контекст редуцированных в конкретном тестовом цикле модулей помечается как базовый, не подлежащий тестированию. Например, если контекст модуля ModF3: (ModF31, ModF32) – помечен как базовый, то в результате рекурсивный спуск затронет лишь модули ModF1, ModF2, ModF3 и вышележащий модуль ModF. Описанный способ организации тестовых наборов с успехом применяется в системах автоматизации тестирования.

Собственно использование эффективной системы автоматизации тестирования сокращает до минимума (например, до одной ночи) время пропуска тестов, без которого невозможно подтвердить факт роста качества (уменьшения числа оставшихся ошибок) продукта. Системное тестирование осуществляется в рамках циклов тестирования (периодов пропуска разработанного тестового набора над build разрабатываемого приложения). Перед каждым циклом фиксируется разработанный или исправленный build, на который заносятся обнаруженные в результате тестового прогона ошибки. Затем ошибки исправляются, и на очередной цикл тестирования предъявляется новый build. Окончание тестирования совпадает с экспериментально подтвержденным заключением о достигнутом уровне качества относительно выбранного критерия тестирования или о снижении плотности не обнаруженных ошибок до некоторой заранее оговоренной величины. Возможность ограничить цикл тестирования пределом в одни сутки или несколько часов поддерживается исключительно за счет средств автоматизации тестирования.

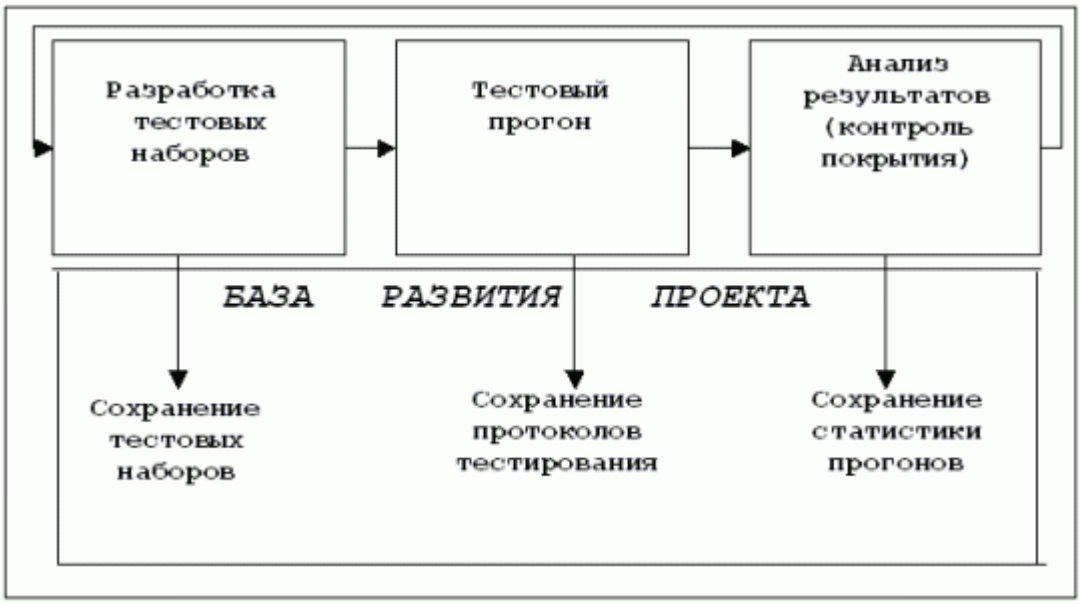

Рисунок 5.1 Рис. 8.1. Структура инструментальной системы автоматизации тестирования

На рисунок 5.1 представлена обобщенная структура системы автоматизации тестирования, в которой создается и сохраняется следующая информация:

– Набор тестов, достаточный для покрытия тестируемого приложения в соответствии с выбранным критерием тестирования – как результат ручной или автоматической разработки (генерации) тестовых наборов и драйвер/монитор пропуска тестового набора.

– Результаты прогона тестового набора, зафиксированные в Log-файле. Log-файл содержит трассы ("протоколы"), представляющие собой реализованные при тестовом прогоне последовательности некоторых событий (значений отдельных переменных или их совокупностей) и точки реализации этих событий на графе программы. В составе трасс могут присутствовать последовательности явно и неявно заданных меток, задающих пути реализации трасс на управляющем графе программы, совокупности значений переменных на этих метках, величины промежуточных результатов, достигнутых на некоторых метках и т.п.

– Статистика тестового цикла, содержащая: 1) результаты пропуска каждого теста из тестового набора и их сравнения с эталонными величинами; 2) факты, послужившие основанием для принятия решения о продолжении или окончании тестирования; 3) критерий покрытия и степень его удовлетворения, достигнутая в цикле тестирования.

Результатом анализа каждого прогона является список проблем, в виде ошибок и дефектов, который заносится в базу развития проекта. Далее происходит работа над ошибками, где каждая поднятая проблема идентифицируется, относится к соответствующему модулю и разработчику, приоритезируется и отслеживается, что обеспечивает гарантию ее решения (исправления или отнесения к списку известных проблем, решение которых по тем или иным причинам откладывается) в последующих build. Исправленный и собранный для тестирования build поступает на следующий цикл тестирования, и цикл повторяется, пока нужное качество программного комплекса не будет достигнуто. В этом итерационном процессе средства автоматизации тестирования обеспечивают быстрый контроль результатов исправления ошибок и проверку уровня качества, достигнутого в продукте. Некачественный продукт зрелая организация не производит.

Издержки тестирования

Интенсивность обнаружения ошибок на единицу затрат и надежность тесно связаны со временем тестирования и, соответственно, с гарантией качества продукта (рисунок 5.2A). Чем больше трудозатрат вкладывается в процесс тестирования, тем меньше ошибок в продукте остается незамеченными. Однако совершенство в индустриальном программировании имеет пределы, которые прежде всего связаны с затратами на получение программного продукта, а также с избытком качества, которое не востребовано заказчиком приложения. Нахождение оптимума – очень ответственная задача тестировщика и менеджера проекта.

Движение к уменьшению числа оставшихся ошибок или к качеству продукта приводит к применению различных методов отладки и тестирования в процессе создания продукта. На рисунок 5.2B приведен затратный компонент тестирования в зависимости от совершенствования применяемого инструментария и методов тестирования. На практике популярны следующие методы тестирования и отладки, упорядоченные по связанным с их применением затратам:

- Статические методы тестирования
- Модульное тестирование
- Интеграционное тестирование
- Системное тестирование
- Тестирование реального окружения и реального времени.

Зависимость эффективности применения перечисленных методов или их способности к обнаружению соответствующих классов ошибок (C) сопоставлена на рисунок 5.2 с затратами (B). График показывает, что со временем, по мере обнаружения более сложных ошибок и дефектов, эффективность низкозатратных методов падает вместе с количеством обнаруживаемых ошибок.

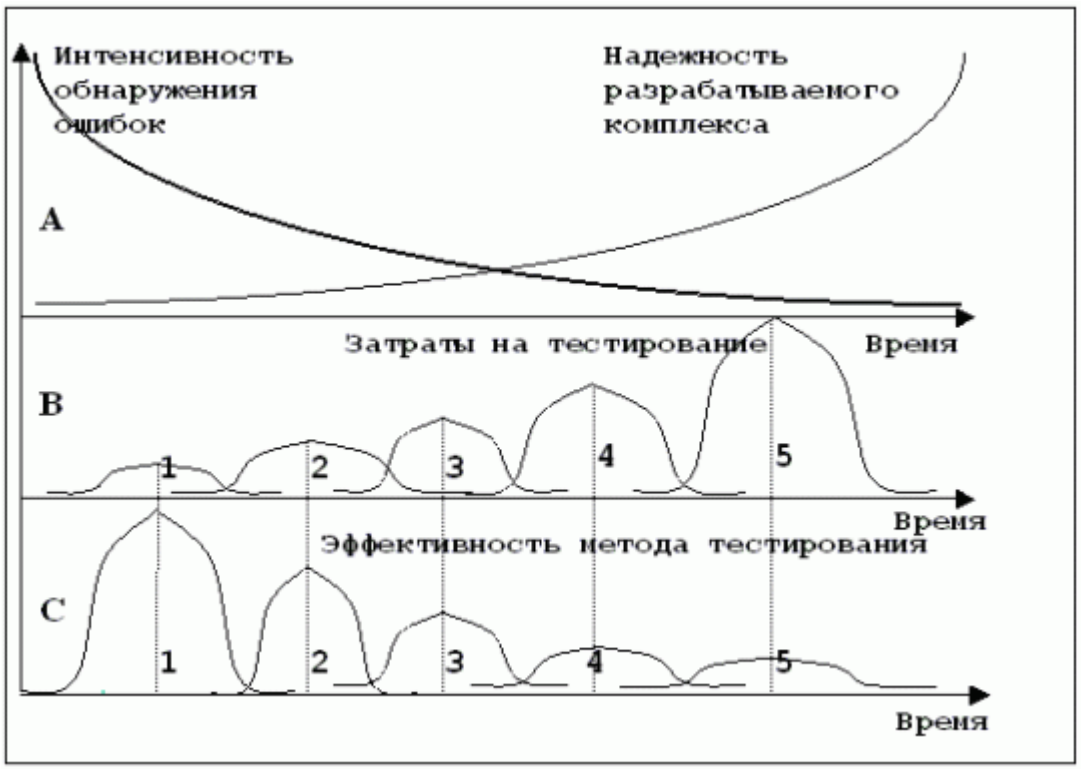

Рисунок 5.2 Рис. 8.2. Издержки тестирования

Отсюда следует, что все методы тестирования не только имеют право на существование, но и имеют свою нишу, где они хорошо обнаруживают ошибки, тогда как вне ниши их эффективность падает. Поэтому необходимо совмещать различные методы и стратегии отладки и тестирования с целью обеспечения запланированного качества программного продукта при ограниченных затратах, что достижимо при использовании процесса управления качеством программного продукта.

5.4 Задания к работе

1. Разработать сценарии тестирования для своего проекта.
2. Запустить тесты на выполнение.
3. Оценить покрытие тестами кода.
4. Добиться максимально возможного покрытия кода.

5.5 Содержание отчета к работе

Отчет должен содержать:

Цель работы, задание, разработанные тесты и сценарии, степень покрытия для каждого модуля.

ПЕРЕЧЕНЬ ВОПРОСОВ.

Системы контроля версий

3. Как скачать исходные коды?
4. Как обновить исходные коды?
5. Как создать новую ветку?
6. Как посмотреть все ветки?
7. Как переключиться в ветку?
8. Как просмотреть изменения?
9. Как создать патч файл?
10. Как применить патч?
11. Как применить патч с созданием новых файлов?
12. Как добавить все изменения?
13. Как добавить изменения по отдельности?
14. Как сделать commit?
15. Как просмотреть commit?
16. Как возвратиться к чистым исходным кодам?
17. Как переключиться в основную ветку?
18. Как удалить созданную ветку?
19. Как сделать откат всех изменений в коде?
20. Как скачать определённую ревизию (к примеру 10 ревизий назад)?

Автоматизация тестирования

1. Введение: тестирование - способ обеспечения качества программного продукта
2. Основные понятия тестирования
3. подходы к обоснованию истинности формул и программ и их связь с тестированием. Вопросы организации тестирования. Фазы тестирования, основные проблемы тестирования и поставлена задача выбора конечного набора тестов.
4. Требования к идеальному критерию тестирования и классы частных критериев. Особенности применения структурных и функциональных критериев на базе конкретных примеров. Особенности применения методов

стохастического тестирования и метод оценки скорости выявления ошибок. Мутационный критерий и на примере иллюстрируется техника работы с ним.

5. Оценка оттестированности проекта: метрики и методика интегральной оценки

6. Графовые модели проекта, метрики оценки оттестированности проекта, приводятся примеры плоской и иерархической моделей проекта.

7. Особенности модульного тестирования, подходы к тестированию на основе потока управления, потока данных, динамические и статические методы при структурном подходе. Взаимосвязь сборки модулей и методов интеграционного тестирования. Подходы монолитного, инкрементального, нисходящего и восходящего тестирования. Рассматриваются особенности интеграционного тестирования в процедурном программировании.

8. Интеграционное тестирование и его особенности для объектно-ориентированного программирования

9. Модель объектно-ориентированной программы, использующая понятие P-путей и MM-путей. Оценки сложности тестирования и методика тестирования объектно-ориентированной программы. Рассматривается пример интеграционного тестирования.

10. Разновидности тестирования: системное и регрессионное тестирование

11. Автоматизация тестирования

12. Особенности индустриального тестирования

13. Документирование и оценка индустриального тестирования

14. Описываются особенности документирования тестовых процедур для ручных и автоматизированных тестов, описаний тестовых наборов и тестовых отчетов. Рассматривается жизненный цикл дефекта. Обсуждаются метрики, используемые при тестировании.

15. Регрессионное тестирование: цели и задачи, условия применения, классификация тестов и методов отбора

16. Регрессионное тестирование: разновидности метода отбора тестов

17. Регрессионное тестирование: методики, не связанные с отбором тестов и методики порождения тестов

18. Регрессионное тестирование: алгоритм и программная система поддержки

19. Описание тестируемой системы и ее окружения. Планирование тестирования
20. Модульное тестирование на примере классов
21. Интеграционное тестирование
22. Системное тестирование
23. Ручное тестирование
24. Автоматизация тестирования с помощью скриптов
25. Автоматическая генерация тестов на основе формального описания
26. Описание ручного тестирования
27. Автоматизация тестирования с помощью скриптов
28. Описание автоматической генерации MSC тестов

ЛИТЕРАТУРА

1. В.В. Липаев. Тестирование компонентов и комплексов программ. Издательство: Синтег, 2010 г, - 400 с.

2. В.П. Котляров. Основы тестирования программного обеспечения. Издательство: Интернет-университет информационных технологий - ИНТУИТ.ру, 2006, - 360 с.

3. И.Н. Скопин. Основы менеджмента программных проектов. Издательство: Интернет-университет информационных технологий - ИНТУИТ.ру, 2004, - 336 с.

4. М. Плаксин. Тестирование и отладка программ - для профессионалов будущих и настоящих. Издательство: Бином. Лаборатория знаний, 2007 г, - 168 с.

5. Л. Криспин, Д. Грегори. Гибкое тестирование. Практическое руководство для тестировщиков ПО и гибких команд. Издательство: Вильямс, 2010 г, - 464 с.

Программное обеспечение и Интернет-ресурсы:

1. MS Visual Studio
2. MSDN

Электронное учебное издание

Методические указания
к лабораторным работам и самостоятельной подготовке по дисциплине
«Современные технологии создания программных систем»

для студентов всех форм обучения
направления 6.050201 Системная инженерия
специальности 8.05020101 Компьютеризированные системы управления и автоматика

Учредитель: Ю.В. Мищеряков

Ответственный за выпуск: _____

Авторская редакция