**НПО. КОНСПЕКТ 2016 года. Часть 3**

**РАЗДЕЛ IV ТЕСТИРОВАНИЕ ПРОГРАММНЫХ СРЕДСТВ.**

**Тема 9** **РЕГРЕССИОННОЕ ТЕСТИРОВАНИЕ**

**Регрессия** (лат. regressio) – «обратное движение, возвращение». **Регрессионное тестирование (РТ, regression testing)** – тестирование, направленное на проверку того факта, что в ранее работоспособной функциональности не появились ошибки, вызванные изменениями в приложении или среде его функционирования. Фредерик Брукс в своей книге «Мифический человеко-месяц» писал: «Фундаментальная проблема при сопровождении программ состоит в том, что исправление одной ошибки с большой вероятностью (20-50 %) влечёт появление новой». Потому РТ является неотъемлемым инструментом обеспечения качества и активно используется практически в любом проекте [5].

**Другое определение**: РТ – это повторное тестирование уже протестированных программ после внесения в них изменений, чтобы обнаружить дефекты, внесенные или пропущенные в результате этих действий. Эти дефекты могут быть как в проверяемом компоненте, так и в связанном или несвязанным с ним. РТ выполняется, когда в ПО или его окружение вносятся изменения.

**Третье определение**: РТ заключается в повторном использовании поднабора тестов (которые до этого уже выполнялись) в новой версии приложения, чтобы убедиться, что функции, которые работали в предыдущей версии системы, по-прежнему работают так, как ожидается (Тамре, [55, с. 286]). Устранение ошибок или добавление новых функций может разрушить что-то, что раньше работало. Добавление новых функций может сделать недействительными старые регрессионные тесты. Таким образом, тестеру может понадобиться удалить или обновить существующие тесты, чтобы учесть функции нового продукта.

**Четвёртое определение** (Савин, [54, с. 271]): РТ как второй этап исполнения тестирования – это проверка того, что изменения, сделанные в ПО (для того, чтобы мир увидел новые фича), не поломали старые фича. Здесь следует отметить, что вместо термина «регрессионное» Савин употребляет термин «регрессивное» тестирование.

Глубина РТ оценивается риском пропуска дефектов в ПО, которое работало ранее. Тесты должны быть повторяемыми, если они должны использоваться для РТ. РТ может выполняться на всех уровнях тестирования и включает функциональное, и нефункциональное тестирование. Регрессионные наборы тестов запускаются множество раз и меняются медленно, поэтому РТ является хорошим кандидатом на автоматизацию.

Вывод о корректности функционирования системы может быть сделан после проведения РТ (Макгрегор, [52, с. 360]). После каждой модификации программы необходимо удостовериться, что на функциональность программы не оказал влияния модифицированный код. Если такое влияние обнаружено, говорят о регрессионном дефекте. Набор регрессионных тестов может быть усечен за счет удаления любых тестов, которые используют неустановленные опции.

Если рассматривается предлагаемая Куликовым полная схема классификации видов тестирования, которая приведена в [5, с. 65], то РТ по Куликову находится в одной группе с такими видами тестирования, как:

* Позитивное тестирование.
* Негативное тестирование.
* Функциональное тестирование.
* Нефункциональное тестирование (non-functional testing, включая отдельные его подвиды – тестирование безопасности, совместимости, интернационализации и т. д.).
* Инсталляционное тестирование (installation testing, installability testing).

• Повторное тестирование (re-testing, confirmation testing).

• Приёмочное тестирование (acceptance testing).

• Операционное тестирование (operational testing).

Эта группа выделена Куликовым по классификационному признаку «по целям и задачам» [5, с. 79]. Есть мнение, что РТ – это наилучший вариант использования автоматических тестов. Программные продукты для этого: Silk Test и др.

**9.1 Особенности регрессионного тестирования**

**9.1.1 Цели и задачи РТ. Цель 1** состоит в том, чтобы в соответствии с используемым критерием покрытия кода (например, критерием покрытия потока операторов или потока данных) гарантировать тот же уровень покрытия, что и при полном повторном тестировании программы. Для этого необходимо запускать тесты, относящиеся к измененным областям кода или функциональным возможностям.

**Цель 2** состоит в том, чтобы удостовериться, что ПП функционирует в соответствии со своей спецификацией, и что изменения не привели к внесению новых ошибок в ранее протестированный код. Эта цель всегда может быть достигнута повторным выполнением всех тестов регрессионного набора, но более перспективно отсеивать тесты, на которых выходные данные модифицированной и старой программы не могут различаться. В табл. 9.1 проведено сравнение повторного РТ с выборочным.

Таблица 9.1 – Сравнение повторного РТ с выборочным

|  |  |
| --- | --- |
| Повторный прогон всех тестов | Выборочное регрессионное тестирование |
| Прост в реализации | Требует дополнительных расходов при внедрении |
| Дорогостоящий и неэффективный | Способно уменьшать расходы за счет исключения лишних тестов |
| Обнаруживает все ошибки, которые были бы найдены при исходном тестировании | Может приводить к пропуску ошибок |

**9.1.2 Выбор тест-комплектов с тест-кейсами при проведении РТ** (Савин, [54, с. 271]). Допустим, у нас есть 5 тест-комплектов с тест-кейсами для новых фича, а также 21 тест-комплект с тест-кейсами для старых фича. Ситуация эта рождает как минимум два вопроса:

1 Какие из этих 21 тест-комплекта выбрать, чтобы:

• проверить именно те части ПО, которые могли быть поломаны?

• уложиться в срок, выделенный для РТ (например, 5 рабочих дней + два выходных дня, которые вполне могут стать рабочими)?

2 Что делать с РТ, когда после релиза к 21 тест-комплекту прибавятся еще 5 (тест-комплекты, которые проверяли новые фича, примкнут к остальным тест-комплектам и станут кандидатами для РТ) и еще, скажем, 10 после следующего релиза и т.д. (постоянно нарастающий снежный ком)? Итак, две темы:

1 Выбор тест-комплектов для РТ.

2 Решение проблемы противоречия между ограниченными ресурсами (например, время на РТ) и перманентно увеличивающимся количеством тест-комплектов.

Тема 1. Выбор тест-комплектов для РТ. Вопрос 1.1. Как узнать, какие части ПО могут быть поломаны? Можно предположить:

• к какой части ПО принадлежат новые фича (например, фича «Новые функциональности для Корзины» принадлежат к «Корзине») и

• какие старые фича напрямую зависят от части ПО с новыми фича (например, компонент «Оплата» использует данные, которые передаются ей компонентом «Корзины»).

Решение следующее:

**1-й группой** кандидатов для РТ у нас будут тест-комплекты, проверяющие часть ПО, к которой принадлежат новые фича. Например, при новых фича для «Корзины» в первую группу идут все тест-комплекты, непосредственно тестирующие «Корзину». **Рациональное объяснение**: если программист напортачил с кодом, то фича, тестируемые тест-комплектами первой группы, будут поломаны СКОРЕЕ ВСЕГО, так как являются частью ПО с измененным кодом.

**2-й группой** кандидатов для РТ у нас будут тест-комплекты, проверяющие старые фича, которые зависят от части ПО с новыми фича. Например, при новых фича для «Корзины» во вторую группу мы можем отнести тест-комплекты, проверяющие «Оплату». **Рациональное объяснение**: если даже программист НЕ сломал ничего, есть большая вероятность того, что код фича, напрямую зависящей от измененной части ПО, также нуждается в модификации (о необходимости которой и продюсер, и программист могли просто... забыть). Например, при изменениях в коде «Корзины» был легитимно (согласно спеку) изменен формат куки (cookie – файл с информацией о вашем заказе, хранящийся на вашем компьютере и используемый веб-сервером). Часть же ПО, которая заведует «Оплатой», не была модифицирована (или была модифицирована неверно), и она (эта часть ПО) просто не понимает новый формат куки, а следовательно, купить книгу не представляется возможным.

Итак, у нас 2 группы тест-комплектов:

Группа Номер тест-комплекта

1 #XS1111

#TS1222

#TS1333

2 #TS2444

#TS2555

#TS2777

#TS2888

#TS2999

Теперь **вопрос второй**: «Как уложиться в срок, выделенный для РТ?». Допустим, что у нас есть два тестировщика и неделя времени, т. е. 80 человеко-часов (112 – с выходными, 336 – без сна и отдыха). **Вопрос**: Сможем ли мы исполнить все 8 тест-комплектов за эти 80 часов? **Ответ**: Очевидно, что для этого нужно знать, сколько времени занимает исполнение каждого из этих тест-комплектов.

Допустим, что мы знаем, сколько времени занимает исполнение каждого тест-комплекта. **Оговорка**: в реальной жизни исполнение тест-комплектов, как правило, занимает гораздо меньше времени, чем в примере ниже, но нам нужна наглядность.

Группа Номер тест-комплекта Время на исполнение (в часах)

1 #XS1111 10

#TS1222 15

#TS1333 17

2 #TS2444 18

#TS2555 12

#TS2777 14

#TS2888 26

#TS2999 19

Итого, 131 час, что больше запланированных 80, и даже если мы будем работать в выходные, то не хватает 19 часов (131 минус 112).

Эти 19 часов могут быть, например, распределены на работу в сверхурочное время: примерно 2 часа 40 минут плюс к нашим восьми часам семь раз в неделю (19:7). Кстати, так и поступают во многих стартапах (новых ИТ-компаниях).

Но допустим, что наш www.testshop.rs не относится к этим многим и славится человечным отношением к своим работникам. Итак, нам, гуманистам, не хватает 51 часа (131 минус 80) для исполнения РТ. Что можно сделать? Среди прочих вещей, таких, как заимствование сотрудников из других отделов, можно сделать следующее: у нас есть приоритет каждого из тест-комплектов. Так давайте же исполним самые приоритетные из них!:

Группа Номер тест-комплекта Время на Приоритет

исполнение

(в часах)

1 #XS1111 10 1

#TS1222 15 3

#TS1333 17 4

2 #TS2444 18 4

#TS2555 12 2

#TS2777 14 1

#TS2888 26 3

#TS2999 19 2

Если мы исполним тест-комплекты

• только 1-го приоритета, то РТ займет 24 часа (10+ 14);

• только 1-го и 2-го, то – 55 часов (24 + 12 + 19);

• только 1, 2 и 3-го, то – 96 часов (55 + +5 + 26), это нам не подходит.

Итак, мы исполняем тест-комплекты 1-го и 2-го приоритетов. Оставшиеся 25 часов (80 минус 55) можно отдать на исполнение, например, наиболее приоритетных тест-кейсов из обоих этих тест-комплектов (самая лучшая идея). Концепция, думаю, понятна.

Решение проблемы противоречия. Проблема противоречия между ограниченными ресурсами (например, время на РТ) и постоянно растущим количеством тест-комплектов решается следующими способами:

а. Приоритизация тест-комплектов и тест-кейсов.

б. Оптимизация тест-комплектов.

в. Наем новых тестировщиков.

г. Автоматизация РТ.

а. О пользе приоритизации мы уже говорили. Странно, но во многих компаниях предпочитают изматывать людей, вместо того, чтобы приоритизировать тест-комплекты и тест-кейсы и исполнять лишь те из них, которые реально важны.

**9.2 Документирование результатов тестирования**

Документирование результатов РТ осуществляется как и для других видов тестирования – для каждого найденного бага составляется отчёт о дефекте (bug-report), п. 7.3.3.1, затем составляется отчёт о результатах тестирования (test result report, TRR), п. 7.3.3.2 (рис. 9.1) и финальный отчёт (final report).

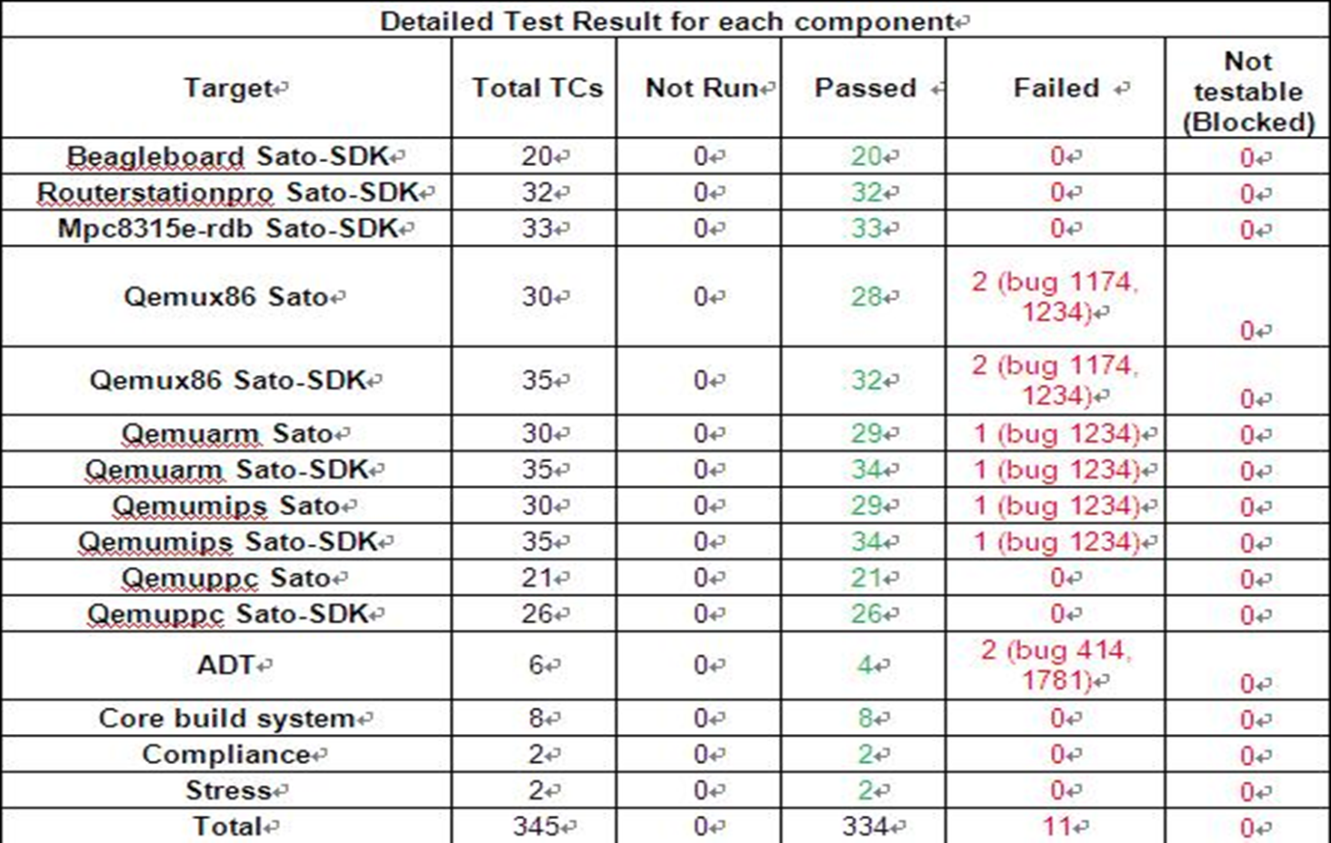


Рисунок 9.1 – Отчёт о результатах тестирования (test result report, TRR)

**Тема 10** **АВТОМАТИЗАЦИЯ ПРОЦЕССА ТЕСТИРОВАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ**

**10.1 Общие сведения об автоматизации ПО**

По степени автоматизированности тестирования различают [5, 58]:

– ручное тестирование (manual testing);

– автоматизированное тестирование (automated testing);

– смешанное/полуавтоматизированное тестирование (semi automated testing).

Ручное тестирование – это исполнение тестов без помощи каких-либо программ, автоматизирующих работу тестировщика. Ручное тестирование – очень трудоемкий процесс. Ручное тестирование заключается в выполнении задокументированной процедуры, где описана методика выполнения тестов, задающая порядок тестов и для каждого теста – список значений параметров, который подается на вход, и список результатов, ожидаемых на выходе. Такое тестирование является наиболее распространенным способом разработки тестов, тем более, что процесс тестирования не всегда поддаётся автоматизации. Этот способ создания тестов является наиболее гибким, однако производительность тестировщиков при ручном тестировании соизмерима с производительностью разработчиков при создании кода продукта.

Автоматизированное тестирование – вид тестирования, при котором используются специальные программные средства для выполнения тестов, сверки полученных фактических результатов с ожидаемыми значениями, установки предусловий для запуска тестов, а также для других функций, включая генерацию отчетов.

Смешанное/полуавтоматизированное тестирование (semi automated testing) – это сочетание ручного подхода с автоматизированным [5, 58].

**10.1.1 Эффективность автоматизации тестирования** [5, 58]. Автоматизация тестирования (test automation) – набор техник, подходов и инструментальных средств, позволяющих исключить человека из выполнения некоторых задач в процессе тестирования [58].

Инструментальное средство автоматизированного тестирования (test automation tool) – программа (или набор программ), позволяющая создавать, редактировать, отлаживать и выполнять автоматизированные тесты, а также собирать статистику их выполнения.

В табл. 10.1 представлены работы, составляющие суть процесса тестирования, и способы их возможного выполнения [58]. Как видно, большинство работ в области тестирования могут быть выполнены только ручным способом, и лишь при выполнении некоторых из них могут быть использованы подходы, методы и инструментальные средства автоматизации.

Таблица 10.1 – Виды работ по тестированию

|  |  |  |
| --- | --- | --- |
| Наименование работы | Способ выполнения |  |
| Анализ документации и спецификаций на наличие разнообразных ошибок и недоработок | Ручной |  |
|  |
| Анализ исходного кода на наличие логич ских ошибок | Ручной |  |
| Разработка тестов | Ручной |  |
| Создание алгоритмов формирования тестовых данных | Ручной |  |
| Первый запуск тестов | Ручной |  |
| Экспертная оценка результатов выполнения тестов | Ручной |  |
| Написание отчeтов об ошибках | Ручной |  |
| Отладка и обновление тестов | Ручной |  |
|  |
| Документирование процесса тестирования | Ручной |  |
|  |
| Выбор тестов, которые необходимо выполнить | Ручной |  |
|  |
| Построение матрицы покрытия требований тестами | Ручной или автоматизированный |  |
|  |
| Выполнение тестов | Ручной или автоматизированный |  |
|  |
| Запись результатов выполнения тестов | Ручной или автоматизированный |  |
|  |
| Статистическая оценка результатов выполнения тестов | Ручной или автоматизированный |  |
|  |
| Измерение числовых характеристик выполнения тестов (производительность) | Ручной или автоматизированный |  |
|  |

Опыт тестирования позволил выделить несколько областей, в которых автоматизация тестирования оказывается наиболее оправданной:

– *выполнение smoke test для крупных систем*, где приходится выполнять большое количество примитивных трудоёмких задач; в крупных системах смоук-тест может быть достаточно длинным и утомительным; при этом он, как правило, содержит достаточно простые для быстрой автоматизации операции; в данном случае автоматизация тестирования приносит ощутимый выигрыш;

– *регрессионное тестирование*, в силу того, что одни и те же тесты выполняются многократно; в РТ достаточно большой процент тестов повторяется снова и снова на каждом цикле тестирования; их автоматизация может сэкономить в конечном итоге много сил и времени;

– *проверка работоспособности стандартного, типичного для множества проектов функционала*; есть функции, повторяющиеся во множестве проектов. Они полностью идентичны или обладают незначительными отличиями; автоматизация тестирования таких функций экономит много времени; примеры таких функций: 1) аутентификация пользователя; 2) регистрация пользователя; 3 добавление товара в корзину; автоматизация тестирования такого функционала позволяет как сократить время, затраченное на тестирование, так и минимизировать риски – подобные тесты легко поддерживать; более того – их почти никогда не приходится подолгу отлаживать и дорабатывать, т. к. они просты;

– *длинные утомительные для человека тесты;* заполнение форм, регистрация тысяч пользователей и т. п. задачи крайне утомительны для человека и являются неприятной рутиной, которой хотелось бы избежать; автоматизация такой работы проста и приносит ощутимый эффект; дополнительным преимуществом автоматизации тестирования при заполнении форм, регистрации тысяч пользователей и т. п. задач является исключение человеческого фактора, заключающегося в том, что на подобных утомительных операциях люди часто совершают множество ошибок;

– проверка стандартных элементов управления; изолированная проверка стандартных элементов управления хорошо автоматизируется как в настольных приложениях, так и в веб-ориентированных;

– конфигурационное тестирование для проверки работоспособности приложения при тех или иных настройках (например, у приложения 100 параметров в файле конфигураций, каждый параметр имеет 100 значений, в результате получаем 10 000 вариантов конфигурационных файлов);

– распределённое тестирование, эмулирующее работу большого количества серверных и клиентских компонентов;

– тестирование web-ориентированных приложений для проверки «битых ссылок», соответствия HTML/CSS-кода стандартам и его совместимости с различными браузерами;

– нагрузочное, стрессовое тестирование и тестирование производительности. Например, необходимость проэмулировать работу нескольких десятков тысяч пользователей с одновременным сбором статистики по работе каждого из них и реакции приложения.

Успешная автоматизация даёт следующие преимущества:

– скорость выполнения тестирования;

– надёжность, поскольку устраняется человеческий фактор;

– мощность, т. к. человек никогда не сможет вручную добавить в базу данных 100 миллиардов записей, составленных из 500 полей каждая;

– сбор статистической информации и представление её в удобной для человека форме;

– средства автоматизации тестирования в сложных ошибочных ситуациях способны выполнять «низкоуровневые действия», сложные для человека.

В то же время автоматизация тестирования имеет недостатки:

– затягивание времени на тестирование; на разработку автоматизированных тестов уходит в 5–10 раз больше времени, чем на создание и однократное выполнение аналогичных ручных тестов;

– высокая стоимость создания автоматизированных тестов, включающая необходимость приобретения специального программного обеспечения;

– необходимость обучения специалистов;

– пропуск новых ошибок при регрессионном тестировании; автоматизированное регрессионное тестирование реже обнаруживает новые ошибки, т. к. тесты часто проходят строго по одному и тому же пути;

– непродуманность вопросов поддержки автоматизированных тестов приводит к тому, что в случае изменений в приложении автоматизированные тесты требуют серьёзной переработки.

В этих условиях каждый проект требует своего подхода к автоматизации. Приступая к автоматизации, следует учесть затраты времени на ручное выполнение тестов, количество повторений тестов, затраты времени на отладку и обновление автоматизированных тестов, затраты времени на выполнение автоматизированных тестов, затраты времени на поддержку автоматизированных тестов и т. д.

Приведем простой пример подсчета эффективности автоматизации тестов, сравнив время на подготовку и создание автоматизированного и ручного тестов. В общем случае специфика проверяемой программы роли не играет и данное сравнение применимо практически к любой ситуации. Если среднее время на создание теста при ручном тестировании (TestCreation, manual, *Tcm*) составляет

*Tcm =6* мин,

то при автоматизированном тестировании аналогичное время (TestCreation, automated, *Tca*) увеличивается в несколько раз по сравнению с *Tcm* до величины, предположим,

*Tca = 30* мин.

Среднее время на прогон теста вручную (TestExecution, manual, *Tem*) равно, допустим,

*Tem = 10*  мин,

а среднее время на прогон теста автоматически (TestExecution, automated, *Tea*) за счёт автоматизации снижается до величины

*Tea = 1* мин.

В этом случае, сравнив общую длительность ручного тестирования (*Т∑m)*

*Т∑m = Tcm + Tem  N*

с общей длительностью автоматизированиого тестирования (*Т∑a)*

*Т∑a = Tca + Tea  N,*

где N – количество итераций теста, можно увидеть, что только при повторении теста более 10 раз автоматизация становится экономически оправданной [58].

**10.1.2 Технология Record&Playback** [58]. Для ускорения процесса разработки автоматизированных тестов большинство средств автоматизации процессов тестирования используют технологию Record&Playback (Запись и воспроизведение). Суть технологии Record&Playback заключается в том, что средство автоматизации тестирования позволяет выполнить с тестируемым приложением некоторый набор действий, которые будут записаны на специальном языке программирования и представлены в виде скрипта. Скрипт – это логически законченная часть кода, сохраненная в отдельном файле и являющаяся программной реализацией определенного тест-кейса.

Для реализации технологии Record&Playback средство автоматизации тестирования должно иметь средство распознавания объектов, библиотеку объектов и среду выполнения.

Средство распознавания объекта – это утилита распознавания класса объекта и его свойств.

Библиотека объектов – это место, где полученная информация будет храниться в специальном формате.

Среда выполнения – это оболочка, которая позволяет исполнять и при необходимости модифицировать скрипты автоматизации.

Преимущества технологии Record&Playback:

– процесс создания исходного «скелета» тестов значительно ускоряется;

– средство автоматизации само собирает необходимую техническую информацию о тестируемом приложении.

Недостатки технологии Record&Playback:

– записанные таким образом тесты «жёстко закодированы» («hard-coded»);

– средство автоматизации записывает все, что происходит в момент записи теста (лишнее затем приходится удалять вручную);

– если в процессе записи тестировщик выполнил неверное действие, часть теста или даже весь тест придётся перезаписать;

– если приложение достаточно сильно изменилось, тест придётся полностью перезаписать.

Данная технология хороша в качестве помощника, но не позволяет создавать сложные тесты [58].

**10.1.3 Язык скриптов [58]**. Разработка сложных автоматизированных тестов выполняется с использования скриптовых языков. Язык скриптов – это высокоуровневый язык программирования, оперирующий терминами и обьектами среды, в которой он исполняется. Будучи скомпилированным в машинный код, программный скрипт заставляет тестируемую программу выполнять определенные действия, заменяя пользователя.

Языки скриптов в зависимости от архитектуры, выбранной поставщиком программного продукта автоматизации процессов тестирования, могут быть самыми разными. Например, средство автоматизации тестирования SilkTest для разработки скриптов использует язык 4Test, имеющий объектно-ориентированную архитектуру [58].

**10.1.4 Структура автоматизации тестирования** [58]. Программная часть автоматизации тестирования состоит из трёх компонентов:

– библиотеки функций (function library); представляет собой набор пользовательских функций, используемых скриптами; сновным критерием при её создании является дублирующаяся функциональность; Пример: функция авторизации пользователя, которая может вызываться многими тестами;

– библиотеки объектов (object repository); представляет собой описание всех графических объектов программного продукта; прежде, чем средство автоматизации процессов тестирования сможет работать с каким-либо элементом графического интерфейса пользователя, этот элемент должен быть описан в соответствующем формате и сохранен в специальном файле;

– библиотеки скриптов (script library); представляет собой набор скриптов, выполняющих задачи автоматического тестирования программного продукта; эта часть автоматизации является основным элементом всего процесса тестирования.

Помимо программной части для автоматизации процессов тестирования важны также наборы данных, включающие ожидаемые результаты, и данные, вводимые при исполнении тестов [58].

На рис. 10.1 представлена обобщенная структура системы автоматизации тестирования, в которой создается и сохраняется следующая информация: а) наборы тестов, достаточные для покрытия тестируемого приложения в соответствии с выбранным

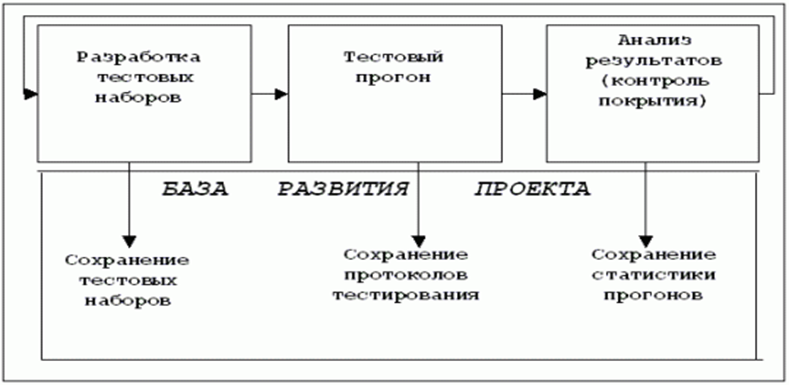


Рисунок 10.1 – Структура инструментальной системы автоматизации тестирования

критерием тестирования (как результат ручной или автоматической разработки (генерации) тестовых наборов) и драйвер/монитор пропуска (прогона) тестового набора; при этом сами тестовые наборы сохраняются в базе развития проекта (БРП), необходимой для заимствования их в следующих проектах; б) результаты прогона тестового набора, зафиксированные в Log-файле, где сохраняются протоколы тестирования. 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.

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 также сохраняются в БРП.

Результатом анализа каждого прогона является список проблем, в виде ошибок и дефектов, который заносится в БРПисходит 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.

**10.1.5 Требования к автоматизированному тесту [58].** Автоматизация тестирования начинается с ручного тестирования. Каждый скрипт базируется на ручном тесте, представленном с должным уровнем детализации. Автоматизированный тест подобен тесту, разработанному для ручного тестирования. По аналогии с ручным сценарием для автоматизированного сценария необходимо выделить последовательность шагов, выполнение которых приведет к значимому для пользователя, системы или автоматизируемого процесса результату, который, в свою очередь, может быть зафиксирован и оценён.

Однако есть ряд проблем, характерных только для автоматизированных тестов:

– необходимость синхронизации работы средства автоматизации и тестируемого приложения по времени;

– ожидаемый результат в автоматизированных тестах должен быть описан предельно чeтко с указанием конкретных признаков его корректности (рис. 10.2);

– тесты следует проектировать в виде тестов, управляемых данными (Data Driven Test), для работы с данными, принимающими широкий диапазон форм представлениязначений; использование технологии Data Driven Testing (DDT) (рис. 10.3, 10.4) также упрощает дальнейшую поддержку и модификацию тестов; DDT – это такой подход к тестированию, при котором тестовые данные хранятся отдельно от тест-кейсов; допустим, нам надо 10 раз выполнять одни и те же тесты с разными данными; для этого можно просто объявить массив с данными (Шаг Property), или же записывать данные в базу (data connection), или в отдельный файл (Excel, CSV; /CSV (от англ. Comma-Separated Values) – значения, разделённые запятыми/ или другие виды файлов, см. рис. 10.3), текстовый формат, предназначенный для представления табличных данных);

– тест может быь автоматизирован с использованием различных инструментальных средств, поэтому его следует описывать, избегая специфических для того или иного инструментального средства решений [58].

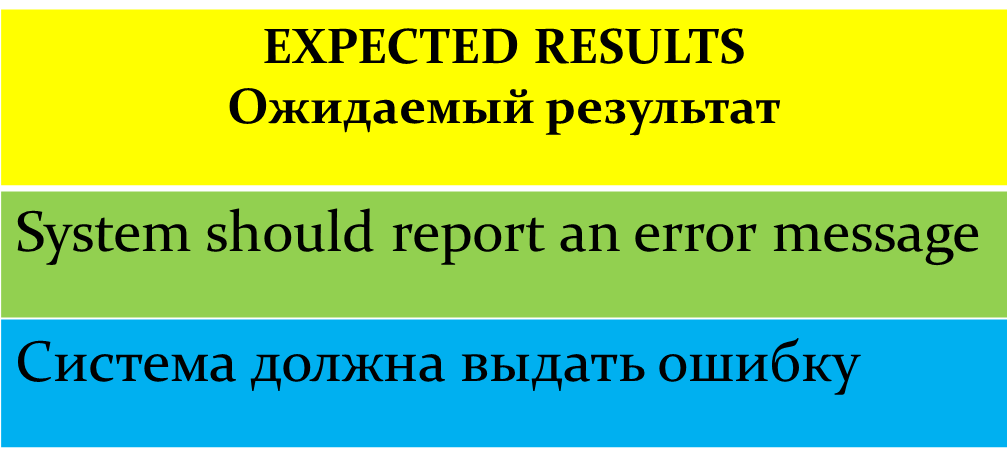


Рисунок 10.2 – Чёткое описание ожидаемого результата в автоматизированном тесте

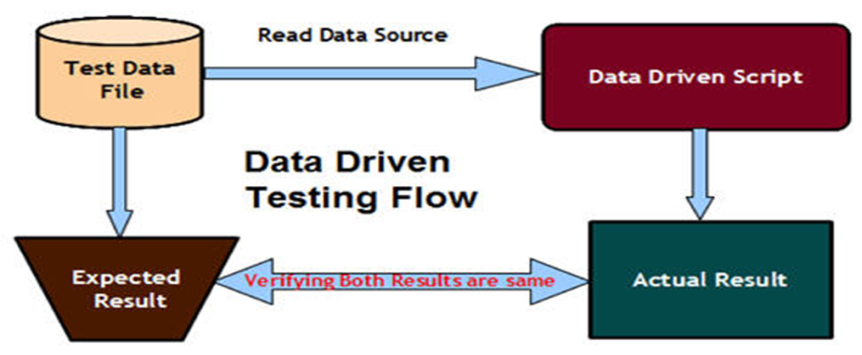


Рисунок 10.3 – Краткая иллюстрация технологии Data Driven Testing

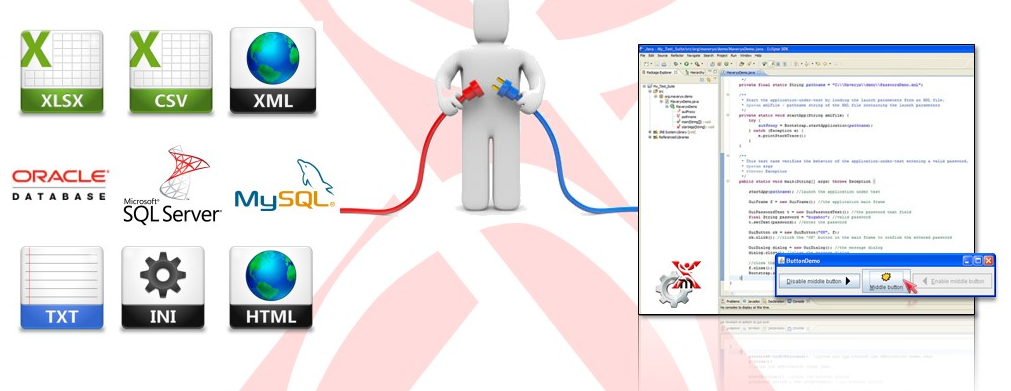


Рисунок 10.4 – Технологии Data Driven Testing, запись данных разных видов в отдельный файл

**10.1.6 Рассмотрим простейший пример ручного и автоматизированного тестирования.** *Ручное тестирование* заключается в выполнении тестового набора (задокументированной процедуры, где описана методика выполнения тестов, задающая порядок тестов и для каждого теста - список значений параметров, который подается на вход, и список результатов, ожидаемых на выходе). Поскольку процедура предназначена для выполнения человеком, в ее описание для краткости могут использоваться некоторые значения по умолчанию, ориентированные на здравый смысл, или ссылки на информацию, хранящуюся в другом документе. Пример выполнения фрагмента процедуры:

1 Подать на вход три разных целых числа.

2 Запустить тестовое исполнение.

3 Проверить, соответствует ли полученный результат таблице [ссылка на документ 1] с учетом поправок [ссылка на документ 2].

4 Убедиться в понятности и корректности выдаваемой сопроводительной информации

В приведенной процедуре тестировщик использует два дополнительных документа, а также собственное понимание того, какую сопроводительную информацию считать «понятной и корректной». Успех от использования процедурного подхода достигается в случае однозначного понимания тестировщиком всех пунктов процедуры. Например, в п. 1 приведенной процедуры не уточняется, из какого диапазона должны быть заданы три целых числа, и не описывается дополнительно, какие числа считаются «разными».

*Автоматизированное тестирование*. Попытка автоматизировать приведенный выше тест-сьют приводит к созданию скрипта, задающего тестируемому продукту три конкретных числа и перенаправляющего вывод продукта в файл с целью его анализа, а также содержащего конкретное значение желаемого результата, с которым сверяется получаемое при прогоне теста значение. Таким образом, вся необходимая информация должна быть явно помещена в текст (скрипт) теста, что требует дополнительных по сравнению с ручным подходом усилий. Также дополнительных усилий и времени требует создание разборщика вывода (программы согласования форматов представления эталонных значений из теста и вычисляемых при прогоне результатов) и, возможно, создание базы хранения состояний эталонных данных. Пример последовательности действий, закладываемых в скрипт:

Выдать на консоль имя или номер теста и время его начала.

Вызвать продукт с фиксированными параметрами.

Перенаправить вывод продукта в файл.

Проверить возвращенное продуктом значение. Оно должно быть равно ожидаемому (эталонному) результату, зафиксированному в тесте.

Проверить вывод продукта, сохраненный в файле (п. 3), на равенство заранее приготовленному эталону.

Выдать на консоль результаты теста в виде вердикта PASS/FAIL и в случае FAIL - краткого пояснения, какая именно проверка не прошла.

Выдать на консоль время окончания теста.

**10.1.7 Особенности тестов для автоматизированного тестирования** (по Куликову). Основная проблема автоматизированного тестирования: компьютер – это не человек, он не понимает никаких абстракций, а потому при автоматизированном тестировании будем помнить об **особенностях** компьютера. Поэтому тесты для автоматизированного тестирования имеют целый ряд особенностей по сравнению с тестами, выполняемыми вручную. Рассмотрим эти особенности подробнее.

**10.1.7.1 Особенность 1.** Точное описание ожидаемого результата. Ожидаемый результат в автоматизированных тестах должен быть описан предельно чётко с указанием конкретных признаков его корректности. ПК «не понимает» таких слов, как «загружается стандартная страница поиска» и т. п. (табл. 10.2).

Таблица 10.2 – Пояснение к особенности «Точное описание ожидаемого результата» теста

|  |  |
| --- | --- |
| Плохо | Хорошо |
|  |  |
| 1. Загружается стандартная страница  поиска. | 1. Загружается страница:  • TITLE = «Search page»  • Содержит форму с полями:  • input type=«text»  • input type=«submit» value=«Go!»  • Содержит логотип (logo.jpg)  • Не содержит иных графических  элементов.  • … |

**10.1.7.2 Особенность 2.** Разные инструменты. Поскольку тест может быть автоматизирован с использованием различных инструментальных средств, следует описывать его, избегая специфических для того или иного инструментального средства решений (табл. 10.3).

**10.1.7.3 Особенность 3.** Особенность **«**Проблемы синхронизации времени». Одной из неожиданно проявляющихся проблем до сих пор является синхронизация средства автоматизации и тестируемого приложения по времени: в случаях, когда для человека ситуация является понятной, средство автоматизации тестирования может среагировать неверно, «не дождавшись» определённого состояния тестируемого приложения. Это приводит к завершению неудачей тестов на корректно работающем приложении (табл. 10.4).

Таблица 10.3 – Пояснение к особенности «Разные инструменты»

|  |  |
| --- | --- |
| Плохо | Хорошо |
|  |  |
| 1. Кликнуть по ссылке «Search».  2. Использовать clickAndWait для  синхронизации тайминга. | 1. Кликнуть по ссылке «Search».  2. Дождаться завершения загрузки страницы. |

Таблица 10.4 – Пояснение к особенности «Проблемы синхронизации времени»

|  |  |
| --- | --- |
| Плохо | Хорошо |
|  |  |
| 1. Кликнуть по ссылке «Search».  2. Проверить, что TITLE загрузив-  шейся страницы равен «Search». | 1. Кликнуть по ссылке «Search».  2. ***Дождаться завершения загрузки страницы.***  3. Проверить, что TITLE загрузившейся страницы равен «Search». |

**10.1.7.4 Особенность 4.** Особенность **«**Технологические особенности». Необходимо хорошо разбираться в особенностях технологий, с помощью которых реализовано тестируемое приложение, т. к. технологические особенности оказывают сильное влияние на способы взаимодействия средства автоматизации тестирования и тестируемого приложения (табл. 10.5).

Таблица 10.5 – Пояснение к особенности «Технологические особенности»

|  |  |
| --- | --- |
| Плохо | Хорошо |
|  |  |
| 1. Отправить приложению сообщение WM\_CLICK в любое из видимых окон. | 1. Передать фокус ввода любому из несвёрнутых окон приложения.  2. Проэмулировать событие «нажата левая кнопка мыши» для активного окна. |

**10.1.7.5 Особенность 5.** Особенность «Различные наборы данных» Следует помнить о том, что входные данные для теста будут поступать извне (data-driven testing), а потому тест должен быть спроектирован готовым к работе с внешними данными, принимающими широкий диапазон форм представления и значений (табл. 10.6).

Таблица 10.6 – Пояснение к особенности «Различные наборы данных»

|  |  |
| --- | --- |
| Плохо | Хорошо |
|  |  |
| 1. Открыть http://www.tut.by | Открыть {START\_URL}. |

**10.1.7.6 Особенность 6.** Особенность «Универсальные подходы» По возможности следует использовать наиболее универсальные способы взаимодействия с тестируемым приложением. Это значительно сократит время поддержки тестов в случае, если приложение сильно изменится (например, добавится веб-интерфейс и т. п.) (табл. 10.7).

Таблица 10.7 – Пояснение к особенности «Универсальные подходы»

|  |  |
| --- | --- |
| Плохо | Хорошо |
|  |  |
| 1. Получить идентификатор поля ввода поискового запроса.  2. Передать в поле с полученным идентификатором набор событий WM\_KEY\_DOWN, WM\_KEY\_UP, в результате чего в поле должен быть введён поисковый запрос. | 1. Однозначно идентифицировать поле ввода поискового запроса.  2. Ввести в данный элемент интерфейса поисковый запрос. |

**10.1.7.7 Особенность 7.** Особенность «Профессиональное программирование». При проектировании автоматизированных тестов обязательно следует избегать т. н. «харкодинга» (hard-coding) и «магических значений» (magic values). Тест должен быть прозрачным с точки зрения программирования и подготовленным к работе с внешними данными (никакие данные не должны быть «намертво вшиты» в тест) (табл. 10.8).

Таблица 10.8 – Пояснение к особенности «Профессиональное программирование»

|  |  |
| --- | --- |
| Плохо | Хорошо |
|  |  |
| if ($date\_value == ‘2011.08.22’)  {  .  }  if ($status = 734)  {  .  } | if ($date\_value == date(‘Y.m.d’))  {  .  }  if ($status == POWER\_USER)  {  .  } |

**10.1.7.8 Особенность 8.** Особенность «Надёжные проверки». Любая проверка должна быть надёжной, т. е. гарантированно проверять именно ТО, что она должна проверить. Так, например, некий текст должен быть найден В НЕКОТОРОМ МЕСТЕ, а не ПРОСТО НА СТРАНИЦЕ (табл. 10.9).

Таблица 10.9 – Пояснение к особенности «Надёжные проверки»

|  |  |
| --- | --- |
| Плохо | Хорошо |
|  |  |
| |  |  |  | | --- | --- | --- | | verifyTextPresent | some text |  | | |  |  |  | | --- | --- | --- | | verifyText | id=SomeID | some text | |

**10.1.7.9 Особенность 9.** Особенность «Независимые тесты». Требование независимости тестов является одним из наиболее жёстких в автоматизации тестирования. Тесты могут выполняться в любой последовательности и/или в составе любого набора. Никогда не следует предполагать, что «вот эти два теста будут выполняться друг за другом, а потому во втором можно не готовить приложение». Автоматизированный тест следует писать так, как будто у нас есть только этот один тест, и больше ничего (табл. 10.10).

Таблица 10.10 – Пояснение к особенности «Независимые тесты»

|  |  |
| --- | --- |
| Плохо | Хорошо |
|  |  |
| 1. Из файла, созданного предыдущим тестом. | 1. Вызвать операцию создания файла.  2. Из созданного файла прочитать. |

Останавливаясь на данных для автоматизированных тестов и способов их генерации следует отметить, что автоматизированные тесты требуют ОГРОМНОГО количества входных данных – в 10-100-1000-10000 раз больше, чем для ручного тестирования. Источниками входных данных при этом могут быть:

• Случайные числа.

• Генерация по алгоритму.

• Получение из внешних источников.

• Собранные «рабочие данные».

• «Ручная» генерация.

Отдельно стоит рассмотреть вопрос действий и проверок. Правило: «ПРОВЕРКУ НЕЛЬЗЯ ЗАМЕНЯТЬ ДЕЙСТВИЕМ». Поясним это на примере:

Действие: войти в дверь.

Проверка: открыта ли дверь?

ПРАВИЛЬНАЯ ПОСЛЕДОВАТЕЛЬНОСТЬ ШАГОВ:

1. Проверить, что дверь открыта.

2. Войти в дверь.

В вышеприведенном примере шаг 1 (проверку) НЕЛЬЗЯ пропускать!

**10.2 Средства автоматизации тестирования программного обеспечения**

Продуктов для автоматизации тестирования ПО очень много. На рис. 10.5 приведена только часть из них, зато с указанием, платные они или нет. Ниже рассмотрены основные средства автоматизации тестирования [58].



Рисунок 10.5 – Платные и бесплатные средства автоматизации тестирования

**10.2.1 SilkTest** (Первый разработчик и поставщик: компания Segue Software Inc, которая в 2006 году была приобретена компанией Borland Software Corporation, а последняя в 2009 году была поглощена (куплена) британской компанией Micro Focus International). SilkTest – инструмент автоматизированного тестирования прикладных программ через графический интерфейс пользователя [58]. Программа предназначена для функционального, регрессионного, кросс-платформенного и локализационного тестирования для широкого диапазона технологий разработки приложений.

Основные возможности:

– использование технологии Data Driven Testing (DDT, тесты, управляемые данными);

– функция Code Completion для быстрой настройки тестов и разработки инфраструктуры автоматизации, интеграция с коллективными коммуникационными средствами Rational;

– выполнение проверок баз данных с помощью стандартного доступа через ODBC (протокол, который применяется для подключения базы данных Microsoft Access к такому внешнему источнику данных, как Microsoft SQL Server), что гарантирует точность выполнения сложных транзакций.

Поддерживаемые технологии: AJAX/Flex/ Web 2.0, Java, .NET клиент/сервер. Поддержка различных браузеров без модификации скриптов. Поддержка Eclipse, Java и JUnit. Поддерживаемые операционные системы: Microsoft Windows. Язык скриптов: 4Test [58].

SilkTest позволяет тестировать все виды приложений – настольные, веб и мобильные. Работа тестировщика в SilkTest подробно освещена в лабораторной работе. На рис. 10.6-10.10 показаны некоторые скринщоты средства SilkTest.

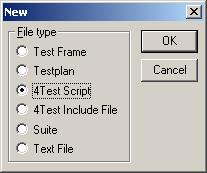
 

Рисунок 10.6 – Создание фрейма приложения Рисунок 10.7 – Запуск тестового приложения

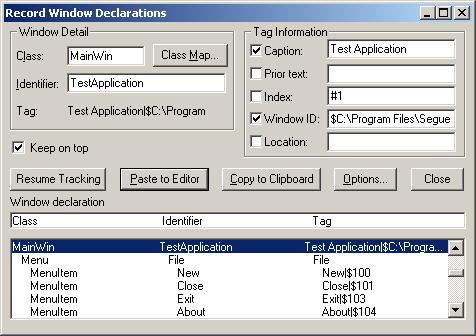


Рисунок 10.8 – Запись описаний окон. Окно Record Window Declaration

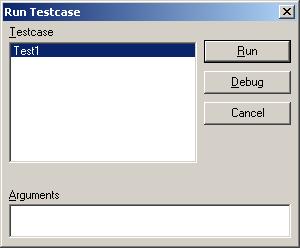
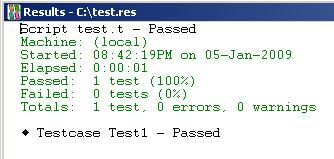
 

Рисунок 10.9 – Запуск записанного тест-кейса Рисунок 10.10 – Отчет о проделанной работе (во сколько времени скрипт начал работать, как долго работал, сколько произошло ошибок и предупреждений и т. п.)

**10.2.2 HP QuickTest Professional** (поставщик: Hewlett Packard Enterprise (HP)). Quicktest Professional (QuickTestPro или QTP) – основной инcтрумент автоматизации функционального тестирования [58]. QTP позволяет автоматизировать функциональные и регрессионные тесты путем записи действий пользователя при работе с тестируемым приложением и дальнейшего исполнения записанных действий с целью проверки работоспособности ПО. Записанные действия сохраняются в виде скриптов. Скрипты отображаются как VBScript (рис. 10.11) (Expert View) или как визуальные последовательные шаги с действиями (Keyword View). Каждый шаг скрипта может быть отредактирован и дополнен точками проверки (CheckPoint) для сравнения ожидаемого результата с фактическим.

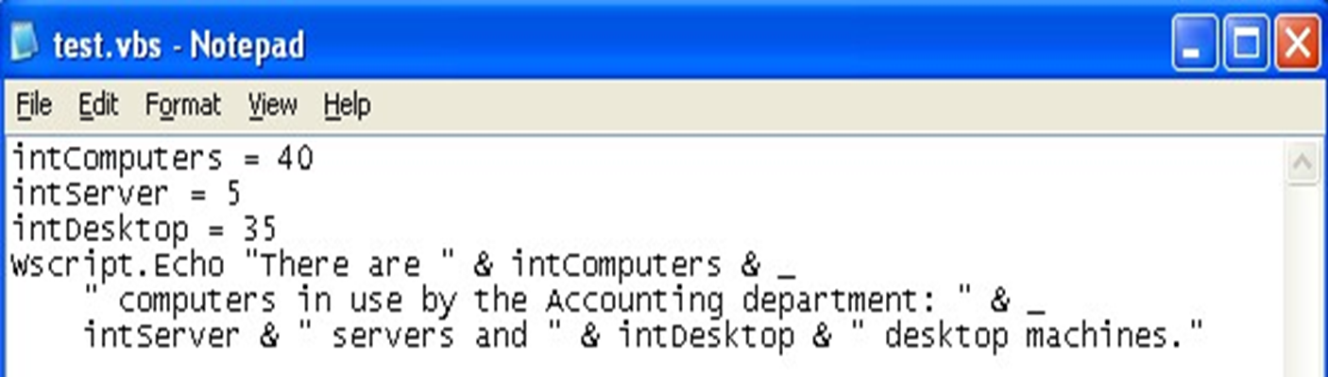


Рисунок 10.11 – Скрипты как VBScript в Notepad

Основные возможности:

– обработка исключительных ситуаций (Exception Handling) (рис. 10.12);

– формирование данных в таблицы с последующим использованием Data Driven Testing;

– работа со сложными UI-объектами;

– расширяемость за счет дополнительных модулей;

– формирование отчётов с результатами выполнения тестирования.

Поддерживаемые технологии: Web, Java, .Net, WPF, SAP, Oracle, Siebel, PeopleSoft, Delphi, Power Builder, Stingray 1, Terminal Emulator, Flex, Mainframe terminal emulator.

Поддерживаемые операционные системы: Microsoft Windows. Язык скриптов: VBScript. Тестируемые приложения: Web, Java, .NET и клиент-серверные приложения.

Особенности:

* В отличие от ряда других продуктов для автоматизации функционального тестирования (например, TestComplete, IBM Rational Robot), QTP позволяет контролировать генерируемый текст скрипта в процессе записи действий пользователя, за счёт чего снижается время, необходимое для разработки теста.

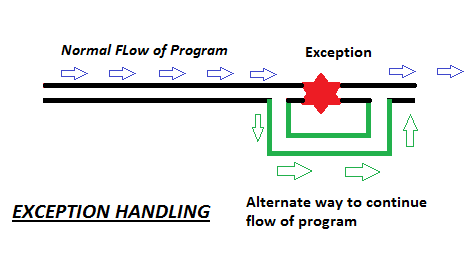


Рисунок 10.12 – Exception Handling

* В QTP информация обо всех объектах экранного интерфейса сохраняется в специальный репозиторий (Object Repository), что новому пользователю может показаться непрозрачным. Умолчания по выбору существенных свойств каждого типа объектов экранного интерфейса могут быть настроены отдельно, например, окно может определяться заголовком, а столбец таблицы – шириной и порядковым номером в таблице.
* Существует встроенный механизм сравнения текстовых данных с использованием регулярных выражений.

На рис. 10.13 показано одно из окон средства тестирования QTP. На рис. 10.14 показано окно, которое появляется при проверке синтаксиса скриптового языка VBscript (синтаксические ошибки возникают во время компиляции кода скрипта; компиляция не может быть завершена без исправления ошибок).

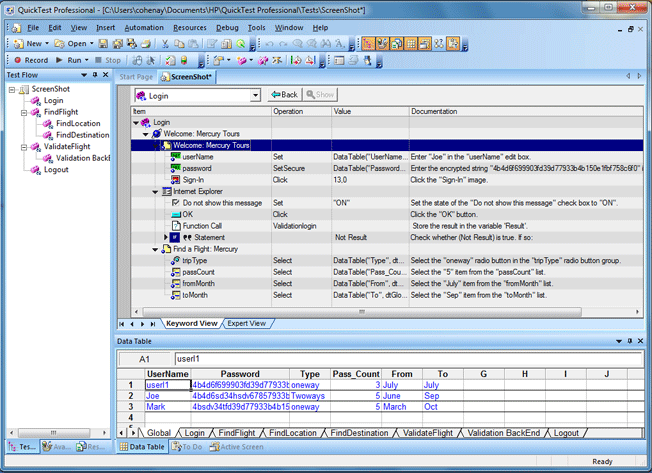


Рисунок 10.12 – Одно из окон средства тестирования QTP

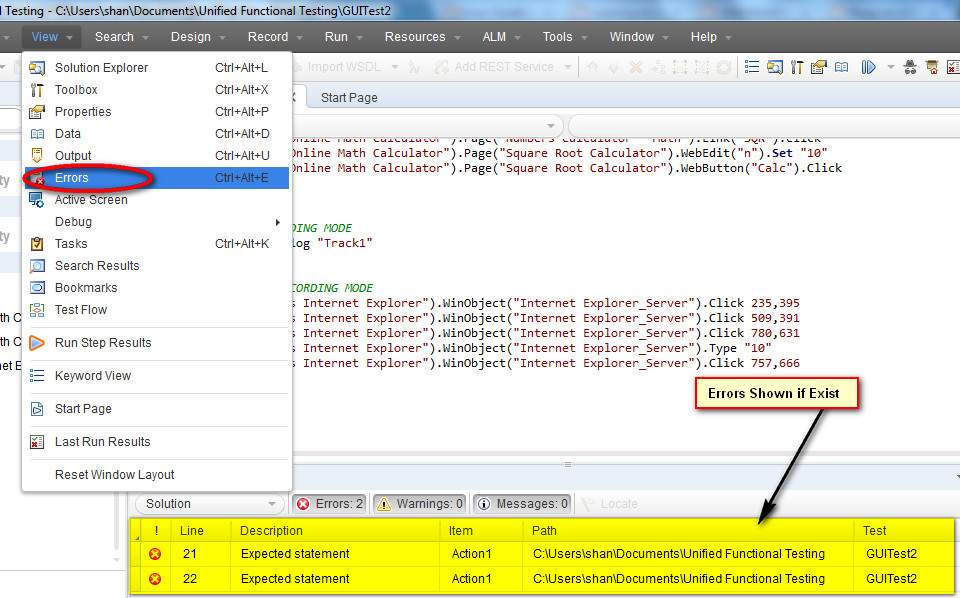


Рисунок 10.14 – Окно QTP, которое появляется при проверке синтаксиса скриптового языка VBscript

**10.2.3 Rational Functional Tester** (поставщик: IBM Rational Software, компания Rational Software до 2003 года была независимой компанией, но в 2003 году её поглотила корпорация IBM). Rational Functional Tester – инструмент автоматизированного тестирования, позволяющий выполнять функциональное тестирование, регрессионное тестирование, тестирование пользовательского интерфейса и тестирование, управляемое данными [58].

Основные возможности:

– интеграция с коллективными коммуникационными средствами Rational;

– интеграция в среду Eclipse, WebSphere Studio и Rational XDE Developer;

– возможность автоматизировать тестирование, устойчивое к частым изменениям пользовательского интерфейса приложений, благодаря технологии ScriptAssure;

– проверка динамических данных с использованием различных мастеров, точек проверки и шаблонов регулярных выражений;

– наличие автоматизированного мастера для создания Data Driven Test;

– выбор языка сценариев для разработки и настройки тестов: Java в среде Eclipse или Microsoft Visual Basic, .NET в среде Visual Studio .NET;

– поддержка пользовательских элементов управления благодаря прокси-объекту SDK (Java/.Net);

– поддержка функционального тестирования сред приложений Oracle ERP посредством поставляемых расширений.

Поддерживаемые технологии: .Net, Java, Siebel, SAP, терминальные приложения, приложения PowerBuilder, AJAX, Adobe ГFlex,Dojo Toolkit, GEF, документы Adobe PDF, приложения zSeries, iSeriesБиpSeries.

Поддерживаемые операционные системы: Microsoft Windows, Linux. Язык скриптов: VBScript, Java. Тестируемые приложения: Java-приложения [58].

На рис. 10.15 показано одно из окон средства тестирования Rational Functional Tester. На рис. 10.16 показано окно вставки действий, управляемых данными (Insert Data Driven Actions window) средства тестирования Rational Functional Tester.

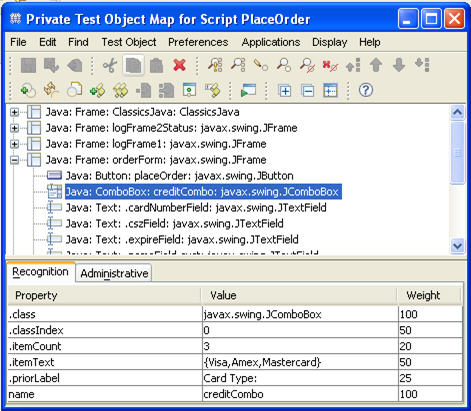


Рисунок 10.15 – Одно из окон средства тестирования Rational Functional Tester (Test Object Map for creditCombo)

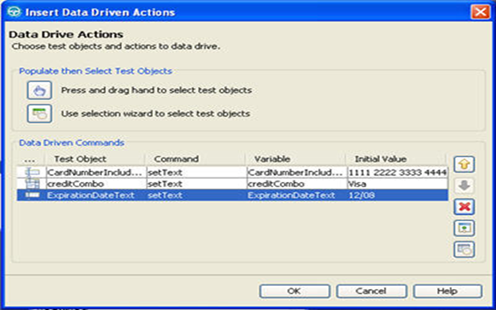


Рисунок 10.16 –Окно вставки действий, управляемых данными (Insert Data Driven Actions window)

**10.2.4 TestComplete**. TestComplete – полнофункциональная система для автоматизации тестирования приложений [58]. С помощью TestComplete можно выполнять функциональное, регрессионное и распределенное тестирование, а также тестирование работоспособности HTTP (на проектном уровне). TestComplete также включает поддержку для нагрузочного тестирования и модульного тестирования.

Основные возможности:

– встроенный редактор кода, помогающий тестировщикам писать скрипты вручную;

– режим Record/Playback с записью только ключевых действий;

– отладчик скриптов для построчного выполнения теста;

– распознавание объектов приложений, написанных на Delphi, C++Builder, .Net, WPF, Java и Visual Basic;

– использование шаблонов основных баг-трэкинговых систем: Microsoft Visual Studio 2005, 2008, 2010 Team System, BugZilla и AutomatedQA и AQdevTeam;

– компилятор TestComplete независим от языка программирования и основан на открытом API, COM интерфейсе;

– возможность автоматической фиксации снимков экрана во время записи и выполнения тестов для последующего сравнения ожидаемого и фактического результатов.

Поддерживаемые технологии: 32-битные и 64-битные Windows приложения, .NET (C#, VB.NET, JScript.NET, VCL.NET, C#Builder, Python .NET, Perl .NET, WPF, Java), Sybase PowerBuilder, Microsoft FoxPro, Microsoft Access, Microsoft InfoPath, Web Browsers (Internet Explorer, Firefox, Netscape Navigator), Visual C++, Visual Basic, Delphi, C++Builder, Flex, Flash, Silverlight, Windows Mobile Applications. Поддерживаемые операционные системы: Microsoft Windows 2000, XP, Server 2003, Server 2008, Vista, Windows 7. Язык скриптов: VBScript, JScript, DelphiScript, C++Script, C#Script. Тестируемые приложения: Web, Java, .NET и клиент-серверные приложения [58].

На рис. 10.17 показано окно средства тестирования TestComplete, служащее для определения названия проекта, расположения сценариев и языка. На рис. 10.18 показано окно средства тестирования TestComplete для просмотра результатов тестирования в веб-браузерах.

**10.2.5 Selenium.** Это средство для тестирования Web-приложений подробно рассматривается в курсе, являющемся продолжением курса НПО (тестирование веб-ориентированных приложений, ТВОП, ТВприл)

**10.2.7 Сравнение средств автоматизации тестирования**. Все рассмотренные выше средства автоматизации тестирования сходны по своим целям и задачам, но различаются реализацией выполнения этих задач. Это объяснимо – компании, их продающие, это конкуренты на рынке. И каждая из систем занимает свой сегмент рынка: есть более дешёвые продукты для программистских предприятий с небольшим объёмом тестирования (SilkTest, TestComplete и др.), но разработаны и более мощные, с расширенным перечнем выполняемых функций, однако более дорогие системы для предприятий со значительным объёмом тестирования (QuickTest Professional, Rational Functional Tester).

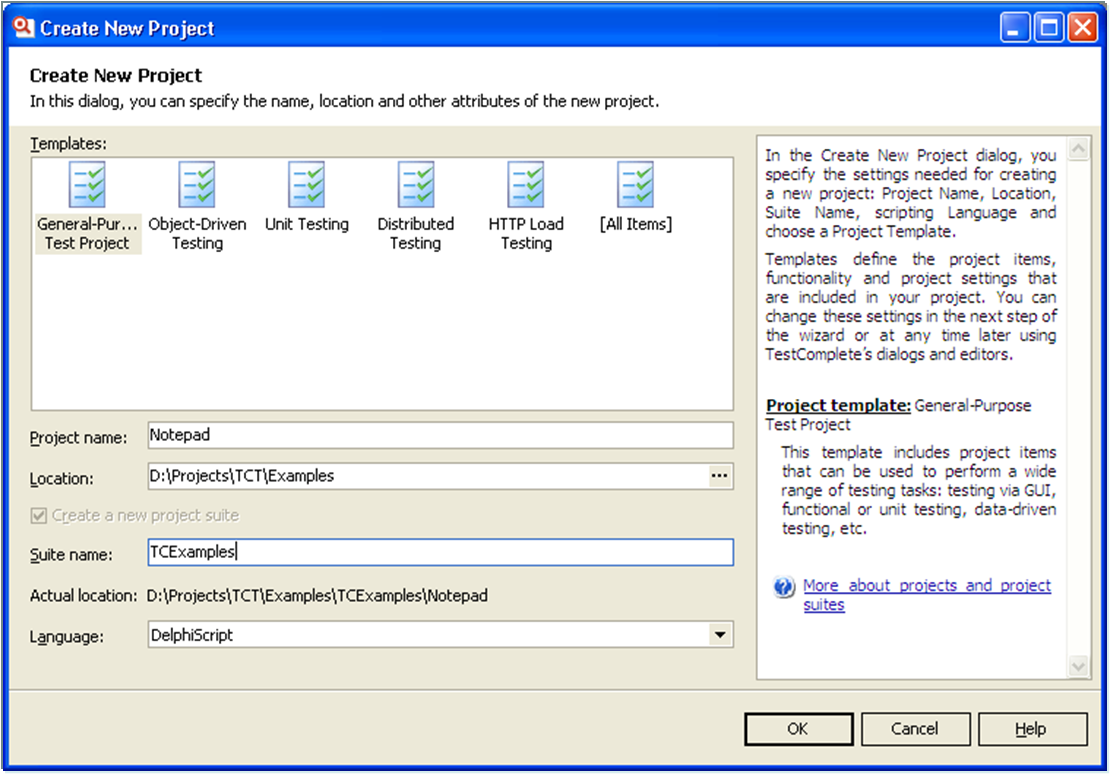


Рисунок 10.17 –Окно средства тестирования TestComplete, служащее для определения названия проекта, расположения сценариев и языка

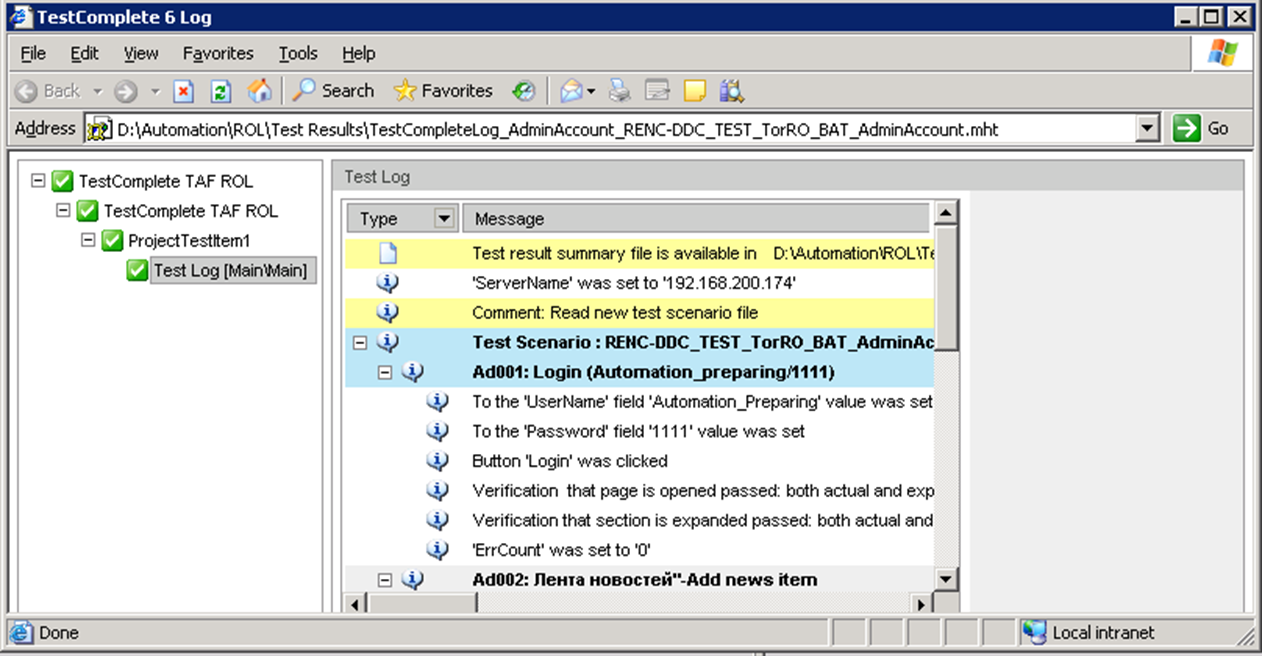


Рисунок 10.18 – Окно средства тестирования TestComplete для просмотра результатов тестирования в веб-браузерах

При сравнении перечисленных продуктов следует рассматривать и другие важные факторы: целостность инструмента, кривую обучения, уровень и качество коммерческой поддержки, уровень поддержки сообщества. Есть также мнение, что успех автоматизации гораздо большей степени зависит от навыков автоматизации у конкретного тестировщика, а не от мощности определенного инструмента.

**10.2.8 Другие инструменты автоматизированного тестирования (ИАТ) и литература.** Кроме вышеперечисленных пяти ИАТ в литературе часто упоминаются ещё 2 инструмента – **WinRunner** и **Rational Robot** [59].

Примером программной среды для функционального тестирования и тестирования веб-приложений является **WinRunner** – полнофункциональное приложение от компании Mercury Interactive Corp, которая в 2006 году за 4,5 млрд. долларов была приобретена компанией Hewlett Packard Enterprise. Считается, что в настоящее время WinRunner устарел, но о нём следует иногда вспоминать по причине того, что в 2000-е годы все студенты БГУИР дневной формы обучения специальности ПОИТ основную лабораторную работу по тестированию в курсе НПО выполняли именно на WinRunner. На рис. 10.19–10.21 показаны некоторые скриншоты средства WinRunner.

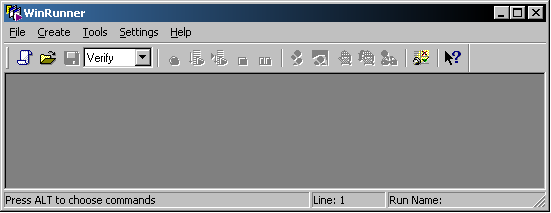


Рисунок 10.19 – Окно средства тестирования WinRunner

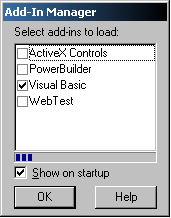
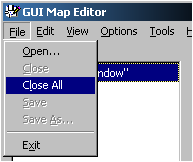
 

Рисунок 10.20 – Выбор подключаемых модулей Рисунок 10.21 – Окно редактора карты интерфейса

**Rational Robot** (поставщик: IBM Rational Software, тот же, что и у Rational Functional Tester, см п 10.1.3). Rational Robot является частью общего процесса тестирования и разработки RUP (Rational Unified Process – методологии разработки программного обеспечения, созданная компанией Rational Software, рис. 10.22). Язык скриптов: SQABasic. ИАТ Rational Robot используется для функционального тестирования и тестирования (в т. ч. нагрузочного) веб-приложений. На рис. 10.23–10.24 показаны некоторые скриншоты средства Rational Robot применительно к среде 1С.

В плане литературы по автоматизированному тестированию студентам рекомендуется коллективная монография Бахтизина, Куликова и Фадеевой [58] (чётко и кратко описаны SilkTest, HP QuickTest Professional, Rational Functional Tester, TestComplete и Selenium), монография Винниченко [59] (в ней хорошо описаны языки скриптов, обработка ошибок, различные функции и т д для ИАТ SilkTest, WinRunner и Rational Robot) и коллективная монография Элфрида Дастина (и других, в ней очень много полезного).

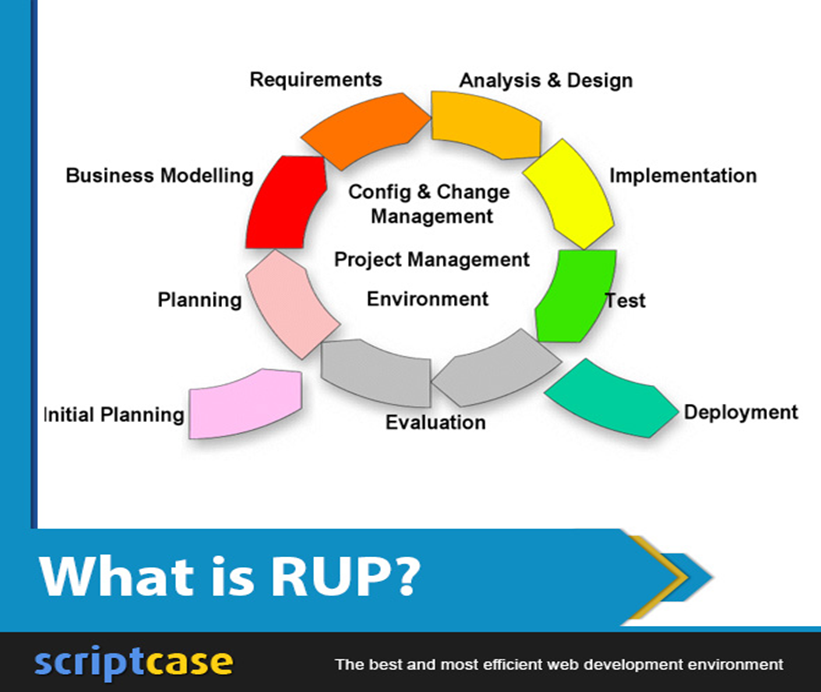


Рисунок 10.22 – Кратко о RUP (Rational Unified Process – методологии разработки программного обеспечения, созданная компанией Rational Software)

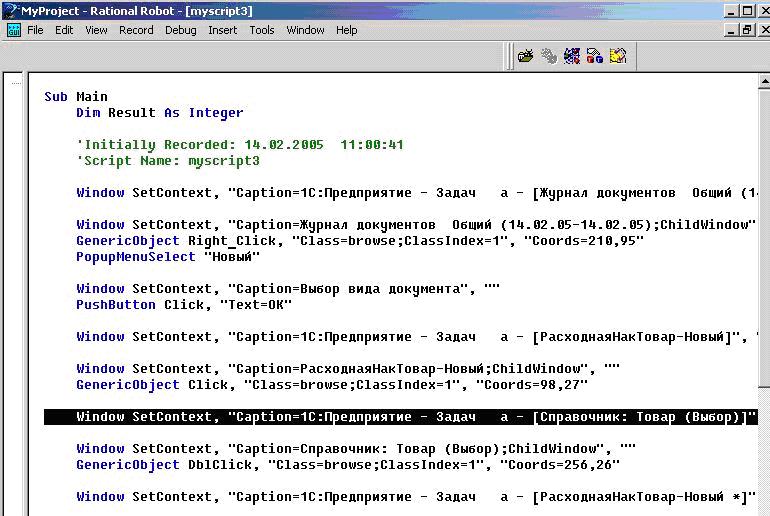


Рисунок 10.23 – Результат теста Rational Robot, записанный в виде скрипта на SQABasic

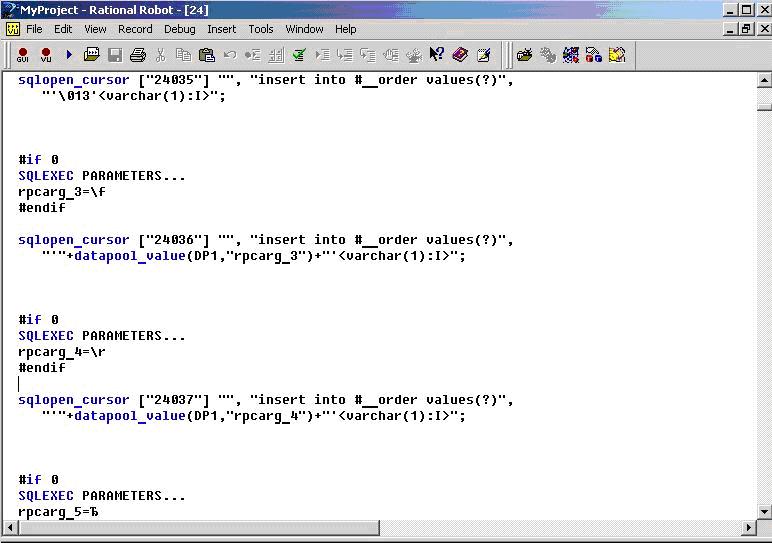


Рисунок 10.24 – Настройка в Rational Robot записи трассы по протоколу ODBC (англ. Open Database Connectivity) – протоколу от Microsoft программного интерфейса (API) доступа к базам данных)

**Тема 11** **ВЕРИФИКАЦИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ**

**11.1 Основные понятия и определения**

**11.1.1 Общие понятия**. Согласно глоссарию [3], стандартам ISO 9000 и их белорусскому аналогу СТБ 9000-2006 [57] верификация (verification) – это доказанное объективными результатами исследования подтверждение того, что определенные требования были выполнены.

Тестирование программного обеспечения не может доказать, что система, алгоритм или программа не содержит никаких ошибок и дефектов и удовлетворяет определённому свойству. Это может сделать формальная верификация, которая согласно Википедии представляет собой формальное доказательство соответствия или несоответствия формального предмета верификации его формальному описанию. При этом предметом верификации выступают алгоритмы, программы и другие объекты.

Одно из первых в СССР определений верификации дано В. В. Бахтизиным в монографии [56, с. 3]: верификация – выполнение формального доказательства того факта, что программа удовлетворяет своим спецификациям. При этом спецификация – любое описание ожидаемой функции, выполняемой программой, содержащее соотношения между аргументами и значениями ожидаемой функции [56, с. 3]. Здесь мы не забываем, что по Википедии спецификация программного обеспечения (спецификация требований ПО, англ. Software Requirements Specification, SRS) – законченное описание поведения программы, которую требуется разработать. Стандартом IEEE Standard 830-1998 «IEEE Recommended Practice for Software Requirements Specifications» определена рекомендуемая структура SRS, описанная в п. 01.2, последний абзац.

Главным недостатком метода тестирования программного обеспечения является невозможность гарантировать отсутствие ошибок в программах. Отсюда возникает вопрос, ЕСТЬ ЛИ КАКОЙ-ТО ДРУГОЙ МЕТОД, КРОМЕ ТЕСТИРОВАНИЯ, который способен гарантировать, что программа не содержит ошибок.

Существуют методы математического доказательства правильности программ, заключающиеся в том, что правильность программы может быть доказана, если выразить намерения программиста в некоторой формальной логической системе, сформулировать математические теоремы о программе, используя это описание и исходный текст программы, а затем доказать эти теоремы. Подобный подход носит название верификации программы. Как уже упоминалось, верификация программы заключается в выполнении формального доказательства, что программа удовлетворяет своим спецификациям [56].

Общим для различных методов верификации программ является представление программы в виде графа, с каждой дугой которого соотносится некоторый предикат, называемый утверждением [56].

Верификация нетривиальной программы очень сложна и трудоемка. В одной из работ было показано, что требуется около месяца труда квалифицированного специалиста для доказательства правильности программы, состоящей из 100 строк. Наибольшая верифицированная к 1984 году программа содержала около 2000 операторов. Но уже в 1984 году существовали некоторые автоматизированные системы верификации, например, для языка Паскаль, который имеет аксиоматическое построение, в отличие от распространенных языков типа Фортран или Алгол [56].

**11.1.2 Методы верификации.** По классификации В. В. Кулямина [63] методы верификации (здесь и далее до п. 11.1.3 текст заимствован из [63]) можно разделить на следующие группы (рис. 11.1):

**11.1.2.1** **Экспертиза** (software review, переводится также как просмотр, обзор, оценка, рецензирование, критический анализ и просто анализ). Экспертизой называются



Рисунок 11.1 – Схема классификации методов верификации по Кулямину [63]

все методы верификации, в которых оценка артефактов ЖЦ ПО выполняется людьми, непосредственно анализирующими эти артефакты. При этом под артефактами жизненного цикла ПО понимаются различные информационные сущности, документы и модели, создаваемые или используемые в ходе разработки и сопровождения ПО. Так, артефактами являются техническое задание, описание архитектуры, модель предметной области на каком-либо графическом языке, исходный код, пользовательская документация и т. д. Различные модели, используемые отдельными разработчиками при создании и анализе ПО, но не зафиксированные в виде доступных другим людям документов, не могут считаться артефактами.

Традиционно выделяются следующие виды экспертиз:

• Техническая экспертиза (technical review) – систематический анализ артефактов проекта квалифицированными специалистами для оценки их внутренней согласованности, точности, полноты, соответствия стандартам и принятым в организации процессам, а также соответствия друг другу и общим задачам проекта.

• Сквозной контроль (walkthrough) – метод экспертизы, в рамках которого один из членов команды проверки представляет ее участникам последовательно все характеристики проверяемого артефакта, а они анализируют его, задавая вопросы, внося замечания, отмечая возможные ошибки, нарушения стандартов и другие дефекты.

• Инспекция (software inspection) – последовательное изучение характеристик артефакта, обычно следующее некоторому плану, с целью обнаружения в нем ошибок и дефектов.

• Аудит (audit) – анализ артефактов и процессов жизненного цикла, выполняемый людьми, не входящими в команду проекта, для оценки соответствия этих артефактов и процессов задачам проекта, заключенному контракту, общим стандартам, друг другу и пр.

Исторически первой техникой экспертиз стала оценка ПО по Фагану (Fagan software inspection), использованная M. Fagan’ом в одном из проектов разработки ПО в IBM в 1972 году. Фактически, она представляет собой образец организации работ 36 (organizational pattern или process pattern), используемый для решения типовой задачи оценки соответствия друг другу двух артефактов жизненного цикла ПО. Эта техника определяет 4 роли участников и 6 шагов выполнения оценки. В ходе оценки сопоставляются первичный документ и вторичный документ, для создания которого первичный является входным. Многочисленные техники экспертиз, созданные после оценки ПО по Фагану, часто повторяют её основные элементы. Более подробно об экспертизах см. в [63].

**11.1.2.2 Статический анализ.** Методы статического анализа артефактов жизненного цикла можно разделить на два вида: контроль того, что все формализованные правила корректности построения этих артефактов выполнены, и поиск типичных ошибок и дефектов в них на основе некоторых шаблонов. Часто инструменты статического анализа используют оба типа проверок.

Чаще всего используется статический анализ исходного кода (для обзора могут использоваться следующие инструменты, реализующих наиболее сложные виды анализа:

* PolySpace Verifier,
* Coverity Prevent,
* Klocwork K7.

Проверенные на практике правила корректности кода или шаблоны типичных ошибок переносятся в среды разработки, такие как Eclipse или Microsoft Visual Studio, и постепенно становятся семантическими правилами языков программирования, их проверка возлагается уже на компиляторы этих языков. Поэтому статический анализ можно считать наиболее широко применяемым методом верификации.

Если в проекте используются языки описания архитектуры или графические языки проектирования, построенные с их помощью артефакты можно также проверять с помощью специализированных инструментов, которые также постепенно встраиваются в широко используемые среды моделирования, такие как Rational Rose.

Поэтому методы верификации при помощи статического анализа либо уже прошли апробацию на практике и используются в коммерческих инструментах и широко применяемых инструментах разработки общего назначения, либо все еще остаются в ранге новаторских, исследовательских работ. Исследовательские методы на данный момент, в основном, связаны с формализацией различных характеристик и свойств ПО и поэтому упомянуты в п. 11.1.2.3, посвящённом синтетическим подходам к верификации. Более подробно о методах статического анализа см. в [63].

**11.1.2.3 Формальные методы верификации.** Эти методы используют формальные модели требований, поведения и окружения ПО для анализа его свойств. Формальные методы верификации делятся на:

* дедуктивный анализ;
* проверку моделей;
* проверку согласованности.

Рассмотрим подробнее дедуктивный анализ. Этот метод верификации был предложен американским учёным в области теории вычислительных систем, лауреатом премии Тьюринга, Робертом Уильямом Флойдом (8.06.1936, Нью-Йорк, США – 25.09.2001, Стэнфорд, США) в 1967 году в работе [61]. Метод базируется на описании математической модели программы по Флойду, подробно рассмотренном в п. 11.1.3.

Дедуктивный анализ Флойдом был разделен на 2 этапа. Сначала доказывалась частичная корректность программы. Для этого использовался метод индуктивных утверждений Флойда (подраздел 11.2). Вторым этапом проводилось доказательство завершаемости программы. Для решения этой задачи использовался метод фундированных множеств Флойда, более подробно о котором см в [62].

Следует отметить, что дедуктивный анализ как метод верификации применялся Флойдом только для простейших моделей последовательных программ, т е для моделей без массивов, без использования адресной арифметики, без рекурсии или вызова подпрограмм, без взаимодействия с окружением (например, посредством операторов ввода-вывода). В настоящее время разработаны методы верификации для более сложных программ, свободных от вышеперечисленных ограничений, но эти методы намного сложнее предложенных Флойдом. Примером таких методов является метод Хоара доказательства частичной корректности программ, использующий аксиоматическую семантику элементарных конструкций и циклов.

**11.1.2.4 Динамические методы верификации.** К ним относятся:

* мониторинг;
* тестирование.

Более подробно о методах динамического анализа см. в [63].

**11.1.2.3 Синтетические методы верификации.** К ним относятся:

* тестирование на основе моделей;
* мониторинг формальных свойств;
* статический анализ формальных свойств;
* синтетические методы структурного тестирования.

Более подробно о синтетических методах верификации см. в [63].

**11.1.3 Описание математической модели программы по Флойду** (здесь и далее до подраздела 11.2 текст заимствован из [62]). Описание начнем [62] с нескольких вспомогательных определений**. Определение 1.** Каждая программа работает с конечным числом **переменных**. Переменные разделяются на три типа: *входные*, *промежуточные* и *выходные*. Вектора этих переменных мы будем обозначать

**x** = (*x1, x2, …, xa*), **y** = (*y1, y2, …, yb* ) и **z** = (z*1, z2, …, zc*) (11.1)

соответственно. Входные переменные содержат исходные входные значения и никогда не меняются во время работы программы. Промежуточные переменные используются для хранения промежуточных результатов в процессе вычисления. Выходные переменные содержат значения, вычисляемые данной программой.

Каждая переменная *v* может принимать значения из некоторого множества D*v*, которое называется *доменом* переменной. Также, мы будем выделять три непустых домена:

*входной домен* D**x** = D*x1* × D*x2* × … × D*xa;*

*домен программы* D**y** = D*y1* × D*y2* × … × D*yb;* (11.2)

*выходной домен* D**z** = D*z1* × D*z2* × … × D*zc*.

Множество значений всех переменных образует *универсальный домен* D. Это значит, что для любой переменной *v* выполнено соотношение: D*v* ⊆ D. Кроме того, мы выделим два специальных значения: Т (истина) и F (ложь). Функции, принимающие значение только из множества {Т, F} мы будем называть *предикатами* на области определения функции.

*Расширенным доменом* D*v*+ переменной *v* будем называть домен этой переменной, расширенный специальным значением ω (D*v*+ = D*v* ∪ {ω}). Значение ω будет использоваться для обозначения не инициализированного значения переменной.

**Определение 2.** Мы будем рассматривать 5 видов **операторов** программы над данным множеством переменных:

1. *Начальный оператор* START: **y** ← *f(***x***)*. Здесь *f* является функцией D**x** → D**y**, инициализирующей промежуточные переменные программы на основе значений ее входных переменных.
2. *Оператор присваивания* ASSIGNMENT: **y** ← *g(***x***,* **y***)*. Здесь *g* является функцией D**x** × D**y** → D**y**, вычисляющей новые значения промежуточных переменных.
3. *Условный оператор* TEST: *t(* **x***,* **y***)*. Здесь *t* является предикатом на множестве значений входных и промежуточных переменных программы.
4. *Оператор соединения* JOIN.
5. *Оператор завершения* HALT: **z** ← *h(* **x***,* **y***)*. Здесь *h* является функцией  
   D**x** × D**у** → D**z**, устанавливающей значения выходных переменных программы.

Графическое представление операторов показано на рис. 11.2. Для обеспечения уникальности одинаковых операторов в рамках одной программы, мы будем помечать каждый оператор уникальной меткой *i*. Таким образом, каждый оператор программы состоит из *метки оператора* и *тела оператора*, принадлежащего к одному из пяти

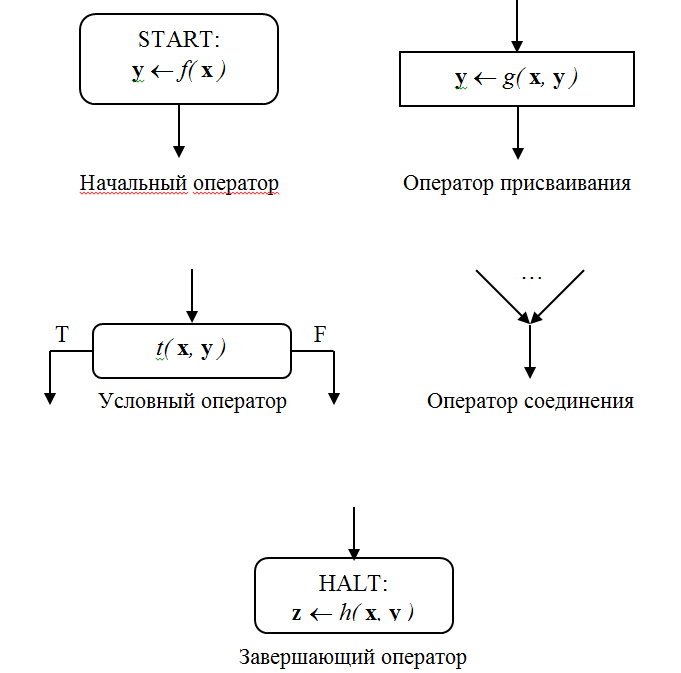


Рисунок 11.2 – Графическое представление операторов блок-схемы

возможных типов. Множество меток всех операторов программы P будет обозначаться как ΛP.

В качестве модели программы мы будем использовать блок-схемы. Определение 3. *Блок-схемой* называется тройка (V, N, E), где

V – конечное множество переменных программы,

N – конечное множество операторов блок-схемы,

E ⊆ N × {T, F, ε} × N – конечное множество связок блок-схемы, помеченных символами T, F или ε.

Заметим, что блок-схема соответствует ориентированному графу, вершинами которого являются операторы программы, а ребрами – ее связки. При этом все ребра помечены одним из трех символов.

**Определение 3.** *Корректно-определенной блок-схемой* мы будем называть блок-схему удовлетворяющую следующим требованиям:

1. В блок-схеме присутствует ровно один начальный оператор.
2. Любой оператор находится на ориентированном пути от начального оператора к некоторому завершающему оператору.
3. Число связок выходящих из каждого оператора и их раскраска соответствует типу оператора:
   1. Из начального оператора выходит ровно 1 дуга, помеченная «пустым» символом ε.
   2. Из оператора присваивания выходит ровно 1 дуга, помеченная «пустым» символом ε.
   3. Из условного оператора выходит ровно 2 дуги, причем одна из них помечена символом T, а другая – символом F.
   4. Из оператора соединения выходит ровно 1 дуга, помеченная «пустым» символом ε.
   5. Из завершающего оператора соединения не выходит ни одной дуги.
4. Число связок входящих в каждый оператора соответствует его типу:
   1. В начальный оператор не входит ни один оператор.
   2. В оператор присваивания, условный и завершающий оператор входит ровно одна дуга.
   3. В оператор соединения входит более или равно одной дуги.

Заметим, что для каждого оператора n и символа s корректно-определенной блок-схемы P существует не более одной связки (n, s, n') ∈ E. Если такая связка существует, то оператор n' мы будем называть последователем оператора n по метке s и обозначать как *succ*(n,s).

Далее мы будем рассматривать только корректно-определенные блок-схемы, и, поэтому, слово «корректно-определенная» будет опускаться. В графическом представлении блок-схем мы будем опускать некоторые детали. Например, ребра помеченные символом ε будут изображаться без соответствующей метки. Как правило, будут опускаться метки операторов блок-схемы, а в операторах соединения не все входящие ребра будут изображаться со стрелками на конце.

Для определения функций будет использоваться следующая нотация:

(*y1, y2, …, yb*) ← (*f1(* **x***,* **y***)*, *f2(***x***,* **y***)*, …, *fb(***x***,* **y***)*) (11.3)

Пример графического представления блок-схем можно увидеть на рис. 11.3, где представлена блок-схема программы целочисленного деления. В этом примере множество переменных V = {*x1*, *x2*, *y1*, *y2*, *z1*, *z2*} состоит из двух входных, двух промежуточных и двух выходных переменных. Доменом всех переменных является множество целых чисел.

**Определение 4.** *Конфигурацией* программы P будем называть пару (, σ), где

* + ∈ ΛP – метка текущего оператора программы,
  + σ = (d*1*, d*2*, …, d*a+b*) ∈ D**x***+* × D**y***+* – вектор значений входных и промежуточных переменных программы.

Если σ ∈ D**x***+* × D**y***+* – вектор значений входных и промежуточных переменных программы, а функция f( **x***,* **y** ): D**x***+* × D**y***+* → D**y***+* вычисляет новые значения переменных **y**, то вектор значений входных и промежуточных переменных программы, полученный путем замены в σ значений переменных **y** на f(σ), мы будем обозначать как σ[**y** ← *f(***x***,***y***)*].

**Определение 5.** Конечная или бесконечная последовательность конфигураций {C*i* | *i* = 1, …, n, …} программы P называется *вычислением*, если

1. Метка первой конфигурации программы является меткой начального оператора.
2. Значения всех входных переменных программы являются определенными (≠ ω) и неизменными во всех конфигурациях вычисления.
3. Значения промежуточных переменных в первой конфигурации являются неопределенными (равными ).

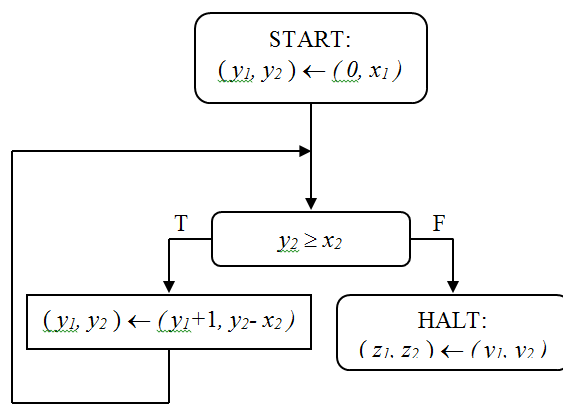


Рисунок 11.3 – Блок-схема программы целочисленного деления

1. Если метка i текущего оператора конфигурации Ci является меткой начального оператора START: y  f(x), то следующая конфигурация Ci+1 состоит из метки оператора succ(ni, ) и вектора значений переменных i+1 = i[y  f(x)].
2. Если метка i текущего оператора конфигурации Ci является меткой оператора присваивания ASSIGNMENT: y  g(x, y), то следующая конфигурация Ci+1 состоит из метки оператора succ(ni, ) и вектора значений переменных i+1 = i[y  g(x, y)].
3. Если метка *i* текущего оператора конфигурации C*i* является меткой условного оператора TEST: *t(***x***,* **y***)* и предикат *t(***x***,* **y***)* при значениях переменных σ*i* принимает значение T, то следующая конфигурация C*i+1* состоит из метки оператора *succ*(n*i*, T) и вектора значений переменных σ*i+1* = σ*i*.
4. Если метка *i* текущего оператора конфигурации C*i* является меткой условного оператора TEST: *t(***x***,* **y***)* и предикат *t(***x***,* **y***)* при значениях переменных σ*i* принимает значение F, то следующая конфигурация C*i+1* состоит из метки оператора *succ*(n*i*, F) и вектора значений переменных σ*i+1* = σ*i*.
5. Если метка *i* текущего оператора конфигурации C*i* является меткой оператора соединения JOIN, то следующая конфигурация C*i+1* состоит из метки оператора *succ*(n*i*, ε) и вектора значений переменных σ*i+1* = σ*i*.
6. Если метка *i* текущего оператора конфигурации C*i* является меткой завершающего оператора HALT: **z** ← *h(***x***,* **y***)*, то C*i* является последней конфигурацией вычисления.

**Лемма 1.** *Для каждой блок-схемы P и вектора значений ее входных переменных* ***x*** *существует единственное вычисление, в первой конфигурации которого значения входных переменных равны* ***x****.*

Каждой блок-схеме P мы поставим в соответствие функцию M[P], из входного домена блок-схемы в выходной домен, расширенный специальным значением ω (M[P]: D**x** → D**z**+, где D**z**+ = D**z** ∪ {ω}). Если вычисление блок-схемы P на векторе входных переменных **x** является конечным, то функция M[P](**x**) принимает значение *h(***x***,* **y**n*)*, *h* – функция завершающего оператора, последней конфигурации вычисления, а **y**n – вектор значений промежуточных переменных из последней конфигурации вычисления. Если вычисление блок-схемы P на векторе входных переменных **x** является бесконечным, то функция M[P](**x**) принимает значение ω.

**11.1.3.1** **Математическая модель требований**. Математическую модель требований к верифицируемой программе мы будем называть спецификацией программы. Семантика спецификации состоит в формальном описании требований к поведению программы. Требований к поведению программ может быть огромное множество, но в данном разделе мы будем рассматривать только требования к функциональности программ. Под требованиями к функциональности понимаются ограничения на результат вычисления программы в зависимости от значений ее входных данных.

**11.1.3.2** **Спецификации**. Спецификацией Φ программы над переменными V мы будем называть два предиката:

*входной предикат* φ: D**x** → {Т, F};

*выходной предикат* ψ: D**x** × D**z** → {Т, F}. (11.4)

*Выходной предикат* (или *постусловие*) определяет, какие значения выходных переменных программы являются корректными относительно значений входных переменных. А *входной предикат* (или *предусловие*) определяет, при каких значениях входных переменных требуется выполнение ограничений описанных в выходном предикате.

**11.1.3.3** **Задача верификации**. Здесь мы рассмотрим 2 аспекта – что такое корректность программ и это же применительно к конкретной программе (верификация программы целочисленного деления).

**11.1.3.3.1** **Корректность программ**. Пусть программа задана своей моделью в виде блок-схемы P, а ее спецификация Φ – предикатами φ и ψ. Мы будем говорить, что программа P *частично корректна* относительно φ и ψ, если для любого вектора значений входных переменных σ, такого, что φ(σ) и M[P](σ) ≠ ω, выполнено ограничение ψ(σ, M[P](σ)). Частичную корректность программы P относительно φ и ψ мы будем обозначать {φ}P{ψ}.

Программа P *полностью корректна* относительно φ и ψ, если для любого вектора значений входных переменных σ, такого что φ(σ) выполнены ограничения M[P](σ) ≠ ω и ψ( σ, M[P](σ)). Полную корректность программы P относительно φ и ψ мы будем обозначать 〈φ〉P〈ψ〉.

Заметим, что полная корректность P программы относительно входного предиката φ и выходного предиката T эквивалентна тому, что программа P завершается всегда, когда вектор значений входных переменных удовлетворяет φ. В этом случае мы будем говорить, что P *завершается* на φ.

**Лемма 2.** *Пусть даны программа* P *и спецификация* Φ = (φ, ψ). *В этом случае* 〈φ〉P〈ψ〉 *тогда и только тогда, когда* {φ}P{ψ} и 〈φ〉P〈T〉.

Исходя из данной леммы, для доказательства полной корректности программы достаточно доказать ее частичную корректность и завершаемость. Из определения корректности также следует, что и частичная, и полная корректность сохраняется при замене входного предиката на более сильный и выходного на более слабый. Таким образом, верна следующая лемма:

**Лемма 3.** *Пусть дана программа* P. *Пусть предикаты* φ, φ', ψ и ψ' *таковы, что формулы* φ' → φ и ψ → ψ' *истинны. Тогда*

*из* {φ}P{ψ} *следует* {φ'}P{ψ} *и* {φ}P{ψ'},

*из* 〈φ〉P〈ψ〉 *следует* 〈φ'〉P〈ψ〉 *и* 〈φ〉P〈ψ'〉. (11.5)

Следующая лемма позволяет по нескольким утверждениям о частичной и полной корректности получать новые утверждения:

**Лемма 4.** *Пусть дана программа* P. *Для любых программы* P *и предикатов* φ, ψ1 и ψ2 *выполнены следующие утверждения:*

*из* {φ}P{ψ1} *и* {φ}P{ψ2} *следует* {φ}P{ψ1 ∧ ψ2},

*из* 〈φ〉P〈ψ1〉 *и* 〈φ〉P〈ψ2〉 *следует* 〈φ〉P〈ψ1 ∧ ψ2〉. (11.6)

**11.1.3.3.2 Верификация программы целочисленного деления**. В дальнейшем, мы сформулируем методы доказательства тотальной корректности программ в общем случае, но сначала обратимся к примеру. Для этого вернемся к программе целочисленного деления, блок-схема которой представлена на рис. 11.3. Мы докажем ее полную корректность относительно спецификации, заданной следующими предикатами:

φ ≡ (*x1* ≥ 0) ∧ (*x2* > 0)

ψ ≡ (*x1* = *z1x2* + *z2*) ∧ (0 ≤ *z2* < *x2*) (11.7)

Входной предикат спецификации утверждает, что нас будет интересовать поведение программы только на неотрицательных значениях переменной *x1* и положительных значениях переменной *x2*. Выходной предикат определяет, что значения выходных переменных программы должно удовлетворять определению целочисленного деления с остатком.

Доказательство полной корректности будет разбито на два этапа. Сначала мы докажем, что программа является частично корректной относительно входного предиката φ0 ≡ (*x1* ≥ 0) ∧ (*x2* ≥ 0) и выходного предиката ψ. А затем мы докажем завершаемость программы на φ. Из этого, по леммам 2 и 3, будет следовать требуемое утверждение.

Заметим, что входной предикат φ0, используемый при доказательстве частичной корректности, является более слабым, чем φ. Это связано с тем, что при значении входной переменной *x2* равном 0, программа является частично корректной, но не завершается.

**11.1.3.3.2.1 Частичная корректность*.*** Поставим в соответствие начальному оператору блок-схемы входной предикат φ0, завершающему оператору – выходной предикат ψ, а ребру между оператором соединения и условным оператором – промежуточный предикат p, задаваемый формулой P(*x1*, *x2*, *y1*, *y2*) ≡ (*x1* = *y1x2* + *y2*) ∧ (*y2* ≥ 0). Это ребро на рис. 11.4 обозначено буквой B.

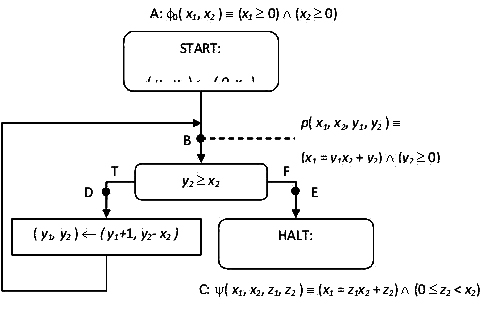


Рисунок 11.4 – Программа целочисленного деления и ее предикаты

1. Рассмотрим путь (рис. 11.5) от начального оператора программы до ребра B. После выполнения начального оператора переменные принимают следующие значения:

|  |  |
| --- | --- |
| *x1* | *x1* |
| *x2* | *x2* |
| *y1* | 0 |
| *y2* | *x1* |

Рисунок 11.5 – Путь от начального оператора программы до ребра B

Таким образом,

P(*x1*, *x2*, *y1*, *y2*) ≡ (*x1* = *y1x2* + *y2*) ∧ (*y2* ≥ 0) ≡

≡ (*x1* = 0⋅*x2* + *x1*) ∧ (*x1* ≥ 0) ≡ (*x1* ≥ 0). (11.8)

А последнее неравенство является истинным в предположении, что выполнено предусловие программы φ0 ≡ (*x1* ≥ 0) ∧ (*x2* ≥ 0). То есть: φ0 ⇒ (*x1* ≥ 0).

2. Предположим, что предикат P истинен в точке B и рассмотрим путь B-D-B. После выполнения условного оператора значения переменных не изменяются, то есть в точке D предикат p также будет истинен, а так как точка D лежит на ребре, помеченном символом T, то в точке D будет истинно следующее утверждение:

(*x1* = *y1x2* + *y2*) ∧ (*y2* ≥ 0) ∧ (*y2* ≥ *x2*). (11.9)

Докажем, что после выполнения последующего оператора присваивания предикат P также будет истинен:

D: (*x1* = *y1x2* + *y2*) ∧ (*y2* ≥ 0) ∧ (*y2* ≥ *x2*);

B: p( *x1*, *x2*, *y1* + 1, *y2* − *x2* ) ≡ (*x1* = (*y1+*1)⋅*x2* + *y2* − *x2*) ∧ (*y2* − *x2* ≥ 0);

(*x1* = *y1x2* + *y2*) ∧ (*y2* ≥ 0) ∧ (*y2* ≥ *x2*) ⇒ (11.10)

⇒ (*x1* = (*y1+*1)⋅*x2* + *y2* − *x2*) ∧ (*y2* − *x2* ≥ 0);

(*x1* = *y1x2* + *y2*) ∧ (*y2* ≥ 0) ∧ (*y2* ≥ *x2*) ⇒ (*x1* = *y1*⋅*x2* + *y2*) ∧ (*y2* − *x2* ≥ 0).

3. И в завершении, рассмотрим последний путь: от точки B до завершающего оператора. Предположим, что в точке B истинен предикат P. Тогда при попадании в точку E будет истинно следующее утверждение:

(*x1* = *y1x2* + *y2*) ∧ (*y2* ≥ 0) ∧ (*y2* < *x2*). (11.11)

Докажем, что после завершающего оператора будет истинен выходной предикат ψ:

E: (*x1* = *y1x2* + *y2*) ∧ (*y2* ≥ 0) ∧ (*y2* < *x2*);

C: ψ( *x1*, *x2*, *y1*, *y2* ) ≡ (*x1* = *y1x2* + *y2*) ∧ (0 ≤ *y2* < *x2*). (11.2)

Из рассмотренных свойств программы следует, что для любого конечного вычисления программы целочисленного деления при значениях входных переменных удовлетворяющих предусловию, значения выходных переменных будут удовлетворять постусловию. Или другими словами, программа целочисленного деления является частично корректной относительно спецификации Φ0 = (φ0, ψ).

**11.1.3.3.2.2 Завершаемость (завершимость).** До сих пор, мы доказали корректность программы только условно. Мы доказали, что если программа завершается, то её результат удовлетворяет предъявляемым к нему требованиям. Теперь докажем, что программа целочисленного деления действительно завершается при значениях входных переменных, удовлетворяющих входному предикату φ.

Рассмотрим блок-схему программы целочисленного деления (рис. 11.6) в аспекте доказательства завершимости программы. Во время доказательства частичной корректности программы, мы доказали, что если значения входных переменных удовлетворяют предикату φ0, то при всяком прохождении точки B значения входных и промежуточных переменных будут удовлетворять следующему условию: (*x1* = *y1x2* + *y2*) ∧ (*y2* ≥ 0). Из этого следует, что если значения входных переменных удовлетворяют более сильному предикату φ, то при всяком прохождении точки B значение промежуточной переменной *y2* будет неотрицательным. С другой стороны, значения входных переменных не изменяются в ходе выполнения программы, поэтому предикат φ является истинным в любой промежуточной точке. Отсюда следует, что в точке B будет выполнено следующее условие: (*y2* ≥ 0) ∧ (*x2* > 0).

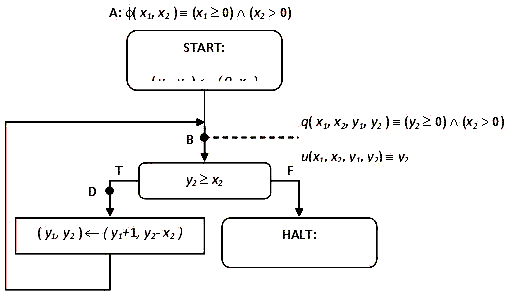


Рисунок 11.6 – Доказательство завершимости программы целочисленного деления

Рассмотрим цикл B-D-B. После прохождения этого пути значение переменной *y2* уменьшится на положительную величину (*x2*). С другой стороны, значение переменной *y2* останется неотрицательным. И так как не существует бесконечной убывающей последовательности неотрицательных целых чисел, то цикл B-D-B не может выполняться бесконечное число раз, если значения входных переменных программы удовлетворяли предикату φ. Следовательно, программа целочисленного деления завершается на входном предикате φ.

Мы доказали, что {φ0}P*div*{ψ} и 〈φ〉P*div*〈T〉. Отсюда, по леммам 2 и 3, следует 〈φ〉P*div*〈ψ〉, то есть **полная корректность программы** целочисленного деления относительно φ и ψ.

**11.2 Метод индуктивных утверждений**

**11.2.1 Описание метода** (здесь и далее до п. 11.2.2 текст заимствован из [64]).Пусть P блок-схема, а α – путь в этой блок-схеме, начинающийся со связки e0 и завершающийся связкой ek. Дадим несколько вспомогательных определений. Определим предикат допустимости Rα(**x**,**y**) : D**x** × D**y** → { Т, F } и функцию пути rα(**x**,**y**): D**x** × D**y** → D**y**. Предикат допустимости Rα(**x**,**y**) определяет, какое должно быть значение входных и промежуточных переменных в начале пути, чтобы дальнейшее вычисление шло по пути α. Функция пути rα(**x**,**y**) определяет, как изменятся значения промежуточных переменных в результате исполнения всех операторов блок-схемы лежащих на пути α.

Наиболее простым способом вычисления предиката допустимости Rα(**x**,**y**) и функции пути rα(**x**,**y**) является *метод обратных подстановок*.

Пусть последовательность операторов на пути α есть n1, …, nk. Для каждого m ∈ {1,…, k+1}, определим предикат Rαm(**x**,**y**) и функцию rαm(**x**,**y**). При этом для m ≤ k, функция rα(**x**,**y**) будет являться функцией пути, проходящего через операторы nm, …, nk, а предикат Rαm(**x**,**y**) будет являться предикатом допустимости для этого же пути. Таким образом,

Rα(**x**,**y**) ≡ Rα1(**x**,**y**),

rα(**x**,**y**) ≡ rα1(**x**,**y**). (11.13)

Предикаты Rαm(**x**,**y**) и функции rα(**x**,**y**) мы определим индукцией по m.

*Базис индукции:*

Rαk+1( **x**, **y** ) ≡ T,

rαk+1( **x**, **y** ) ≡ **y.** (11.14)

*Индуктивное предположение:* Предположим, что Rαm+1(**x**,**y**) и rαm+1(**x**,**y**) уже определены для некоторого m ∈ {2, …, k+1}.

*Индуктивный переход:* Определим зависимости Rαm(**x**,**y**) и rαm(**x**,**y**) от оператора nm:

* + Если nm – оператор присваивания ASSIGNMENT: **y** ← *g(***x***,* **y***)*, то

Rαm(**x**, **y**) ≡ Rαm+1(**x**, *g(***x***,* **y***)*), (11.15)

rαm( **x**, **y** ) ≡ rαm+1(**x**, *g(***x***,* **y***)*). (11.16)

* + Если nm – условный оператор TEST: *t* **x***,* **y***)* и связка em помечена символом T, то

Rαm(**x**, **y**) ≡ Rαm+1(**x**, **y**), (11.17)

rαm(**x**, **y**) ≡ rαm+1(**x**, **y**) ∧ *t(***x***,* **y***).* (11.18)

* + Если nm – условный оператор TEST: *t(***x***,***y***)* и связка em помечена символом F, то

Rαm(**x**, **y**) ≡ Rαm+1(**x**, **y**), (11.19)

rαm(**x**, **y**) ≡ rαm+1(**x**, **y**) ∧ ¬*t(***x***,* **y***).* (11.20)

* + Если nm – оператор соединения JOIN, то

Rαm(**x**, **y**) ≡ Rαm+1(**x**, **y**), (11.21)

rαm( **x**, **y** ) ≡ rαm+1(**x**, **y**). (11.22)

Заметим, что операторы других типов (начальный и завершающий) не могут встречаться на пути, так как они не имеют либо входящих, либо выходящих связок. Для этих операторов, мы расширим определения предиката допустимости и функции пути. Если началом первой связки пути является начальный оператор START: **y** ← *f(***x***)*, то мы будем говорить, что предикат допустимости R'α(**x**) ≡ Rα(**x**, *f(***x***)*), а функция пути r'α(**x**) ≡ rα(**x**, *f(***x***)*). Если концом последней связки пути является завершающий оператор HALT: **z** ← *h(***x***,* **y***)*, то мы будем говорить, что функция пути r''α(**x**, **y**) ≡ *h(* **x***,* rα(**x**, **y**) *)* (r''α(**x**, **y**) : D**x** × D**y** → D**z**).

***Индуктивные утверждения.*** Пусть P – блок-схема, а Φ = (φ, ψ) – её спецификация. Рассмотрим следующий метод доказательства частичной корректности программы P относительно спецификации Φ.

***Шаг 1. Точки сечения.*** Выберем подмножество связок блок-схемы. Эти связки мы будем называть *точками сечения*. Выбранное множество точек сечения должно быть таким, чтобы каждый цикл в блок-схеме содержал, по крайней мере, одну точку сечения. Все ориентированные пути между точками сечения, не содержащие других точек сечения, мы будем называть *промежуточными базовыми путями*. Пути, начинающиеся в начальном операторе, заканчивающиеся в точке сечения и не содержащие других точек сечения, мы будем называть *начальными базовыми путями*. Пути, начинающиеся в точке сечения, заканчивающиеся в одном из завершающихся операторов и не содержащие других точек сечения, мы будем называть *конечными базовыми путями*.

***Шаг 2. Индуктивные утверждения****.* Выберем для каждой точки сечения *i* предикат p*i*(**x**, **y**), который характеризует отношение между переменными блок-схемы при прохождении данной связки. Часто, эти предикаты называют *индуктивными утверждениями*. Кроме того, свяжем входной предикат φ(**x**) с начальным оператором блок-схемы, а выходной предикат ψ(**x**, **z**) – со всеми завершающими операторами.

***Шаг 3. Условия верификации****.* На третьем шаге, сконструируем для каждого промежуточного базового пути α, начинающегося в точке сечения *i* и завершающегося в точке сечения *j*, условия верификации:

∀ **x** ∈ D**x** ∀ **y** ∈ D**x** [p*i*(**x**, **y**) ∧ Rα(**x**, **y**) ⇒ p*j*(**x**, rα(**x**, **y**))]. (11.23)

Эти условия утверждают, что если предикат p*i*(**x**, **y**) истинен для некоторых значений переменных **x** и **y**, и эти значения такие, что начиная с точки сечения *i* вычисление пойдет по пути α, то предикат p*j*(**x**, **y**) будет истинен для значений переменных **x** и **y** после прохождения по пути α.

Для начального базового пути α, завершающегося в точке сечения *j*, условия верификации будут выглядеть следующим образом:

∀ **x** ∈ D**x** [ φ(**x**) ∧ R'α(**x**) ⇒ p*j*( **x**, r'α(**x**))]. (11.24)

В этом случае условия утверждают, что если входной предикат φ(**x**) истинен для некоторых значений входных переменных, и эти значения такие, что начальное вычисление пойдет по пути α, то предикат p*j*(**x**, **y**) будет истинен для значений переменных **x** и **y**, после прохождения по пути α.

Для конечного базового пути α, начинающегося в точке сечения *i*, условия верификации конструируются следующим образом:

∀ **x** ∈ D**x** ∀ **y** ∈ D**x** [p*i*(**x**, **y**) ∧ Rα(**x**, **y**) ⇒ ψ(**x**, r''α(**x**, **y**))]. (11.25)

Здесь условия верификации утверждают, что если предикат p*i*(**x**, **y**) истинен для некоторых значений переменных **x** и **y**, и эти значения такие, что начиная из точки сечения *i* вычисление пойдет по пути α, то предикат ψ(**x**, **z**) будет истинен для значений переменных **x** и **z**, после завершения работы блок-схемы при прохождения по пути α.

Для вырожденного случая, когда начальный базовый путь α является в то же время завершающим базовым путем, условия верификации будут выглядеть следующим образом:

∀ **x** ∈ D**x** [φ(**x**) ∧ R'α(**x**) ⇒ ψ(**x**, r''α(**x**, r'α(**x**)))]. (11.26)

В этом случае условия утверждают, что если входной предикат φ(**x**) истинен для некоторых значений входных переменных, и эти значения такие, что вычисление пойдет по пути α, то выходной предикат ψ(**x**, **z**) будет истинен для значений переменных **x** и **z**, после завершения работы блок-схемы при прохождении по пути α.

**Лемма 5.** *Пусть все условия верификации истинны. Пусть дано вычисление блок-схемы* P*, входные переменные которого удовлетворяют входному предикату* φ. *Тогда для каждого прохода вычисления* C*k* – C*k+1* *через точку сечения i предикат* p*i*(**x**, **y**) *будет истинен на значениях переменных* **x** *и* **y** *в конфигурации* C*k*.

**Теорема 1. *(Метод индуктивных утверждений Флойда).*** *Пусть даны блок-схема* P *и ее спецификация* Φ = (φ, ψ). *Выполним следующие действия:*

1. *Выберем точки сечения;*
2. *Найдем подходящий набор индуктивных утверждений;*
3. *Построим условия верификации для всех базовых путей.*

*Если все условия верификации истинны, то блок-схема* P *частично корректна относительно спецификации* Φ*.*

Все шаги метода Флойда, за исключением второго, могут быть выполнены относительно автоматически. А вот выбор подходящего набора индуктивных утверждений требует хорошего понимания функционирования программы и поэтому сложно поддается автоматизации.

**11.2.2 Краткое изложение, преимущества и недостатки метода** (здесь и далее до подраздела 11.3 текст заимствован из [64]).

**11.2.2.1 Краткое изложение.** Согласно методу индуктивных утверждений программа снабжается утверждениями о свойствах её переменных в нкоторых точках:

a) входные переменные не меняются в процессе исполнения программ;

b) описываются состояния переменных в промежуточных точках;

c) выходные переменные описываются с помощью отношениями между переменными после завершения программы.

Верификация состоит в последовательной демонстрации того, что из входных переменных и преобразований, выполненных на первом шаге следует истинность утверждения, сформированного в следующей промежуточной точке. Для верификации программ необходимо три языка:

* язык записи текстов программ;
* язык формулировки условий верификации;
* язык формирования и доказательства корректности.

**11.2.2.2** Доказательство корректности методу индуктивных утверждений имеет следующие **преимущества**:

1. Представляет собой чёткий формализованный процесс.

2. Требует анализа. Процесс доказательства корректности дает возможность рассматривать части программ, которое в противном случае анализируется лишь случайно.

3. Проясняет промежуточные результаты вычислений. Выписывании выражений заставляет программиста четко сформировать свои предположения о результатах вычислений в выбранных точках программы.

4. Выявляет зависимости. В процессе доказательства программ начинает понимать какие предположения о входных данных не явно испытывается в различных частях программы.

**11.2.2.2 Недостатки метода индуктивных утверждений**:

1. Сложность; даже для небольших простых программ выкладки очень сложны, что может привести к ошибкам.

2. Ошибки, Из-за сложности метода легко допустить ошибки и при формировании доказываемых утверждений и при доказательстве.

3. Трудности работы с массивами.

4. Отсутствие мощного математического аппарата.

5. Высокая трудоёмкость: для проверки программы требуется затраты труда, в 2 – 6 раз превышающие затраты для её написания.

6. Отсутствие выразительности. Часто нелегко сформировать выгодно утверждение для того что интуитивно представляется очень простым вычислением:

7. Трудность понимания.

8. Необходимость обучения. Для применении этого метода требуется длительное обучение и тренировка.

**11.3 Структура процесса верификации в соответствии с положениями действующих стандартов (например, СТБ ИСО/МЭК 12207-2003, ISO/IEC 12207:2008)**

**11.3.1 Общие положения**. Стандарт *СТБ ИСО/МЭК 12207-2003* дает (здесь и далее до пункта 11.3.2 текст заимствован из [31]) такое определение процесса верификации: «***Процесс верификации*** является процессом определения того, что программные продукты функционируют в полном соответствии с требованиями и условиями, реализованными в предшествующих работах». Данное определение является не совсем понятным для лиц, некомпетентных в области верификации. Кроме того, оно сужает применимость верификации только до программных продуктов. Для пояснения данного определения следует рассмотреть само понятие верификации. Термин *верификация* *(verification)* обозначает подтверждение с помощью экспертизы и представления объективных доказательств того, что конкретные требования полностью реализованы (см начало темы и стандарты МСО). В процессе разработки верификация связана с экспертизой результатов данной работы с целью определения их соответствия установленным на входе данной работы требованиям. Таким образом, верификация может применяться не только к программным продуктам, но и к любым другим результатам работы. Например, верификации могут подвергаться требования, системная и программная архитектура, документация, методы, планы и т.п.

В этой связи более понятным и близким к истине является следующее определение процесса верификации. ***Процесс верификации*** – это процесс определения того, что результаты работы соответствуют требованиям или условиям, установленным на входе данной работы.

Процесс верификации может включать анализ, проверку и тестирование. Объектами анализа и проверки могут являться, например, документация и исходные тексты программных модулей. Объектами тестирования могут являться исполнимые коды программных модулей, компонентов, промежуточных и конечного программных продуктов. Данный процесс называется ***процессом независимой верификации***, если организация–исполнитель не зависит от поставщика, разработчика, оператора или персонала сопровождения.

Процесс верификации состоит из двух работ (рис. 11.7, причём нумерация процессов, работ и задач на рис. 11.7 соответствует принятой в СТБ ИСО/МЭК 12207-2003). Общее число задач по данным работам равно 13.

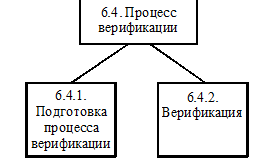


Рисунок 11.7 – Процесс верификации в соответствии с положениями действующего в РБ стандарта СТБ ИСО/МЭК 12207-2003

Как следует из нумерации процессов, работ и задач на рис. 11.7, процесс верификации в соответствии с положениями действующего в РБ стандарта СТБ ИСО/МЭК 12207-2003относится к группе вспомогательных процессов ЖЦ ПП. Состав задач работ 6.4.1 и 6.4.2, входящих в процесс верификации, показан в табл. 11.1, 11.2.

При этом указанное в столбце «Тип выходного результата задачи» табл. 11.1 «Описание» подразумевает «Описание результата выполнения каждой задачи (6.4.1.1, 6.4.1.2 и т д) в составе плана верификации, который в свою очередь является результатом выполнения задачи 6.4.1.5 «Разработка плана верификации» той же работы 6.4.1 «Подготовка процесса верификации».

Таблица 11.1 – Состав задач работы 6.4.1 «Подготовка процесса верификации»

|  |  |  |
| --- | --- | --- |
| № задачи в  ИСО/МЭК  12207 | Описание задачи | Тип выходного результата задачи |
| 6.4.1.1 | Определение необходимости в проекте работ по верификации и степени их организационной независимости, анализ критичности проектных требований | Описание |

|  |  |  |
| --- | --- | --- |
| 6.4.1.2 | Установка процесса верификации (при необходимости) | Описание |
| 6.4.1.3 | Выбор соответствующей независимой квалифицированной организации (при необходимости) | Описание |
| 6.4.1.4 | Определение верифицируемых работ и продуктов, выбор работ и задач верификации | Описание |
| 6.4.1.5 | Разработка плана верификации | План |
| 6.4.1.6 | Реализация плана проведения верификации, обеспечение доступности его результатов заказчику, при необходимости организация связи с процессом решения проблем | Отчет, протокол |

Таблица 11.2 – Состав задач работы 6.4.1 «Верификация»

|  |  |  |
| --- | --- | --- |
| № задачи в  ИСО/МЭК  12207 | Описание задачи | Тип выходного результата задачи |
| 6.4.2.1 | Верификация договора по критериям |  |
| 6.4.2.2 | Верификация процесса по критериям |  |
| 6.4.2.3 | Верификация требований по критериям |  |
| 6.4.2.4 | Верификация результатов проектирования по критериям |  |
| 6.4.2.5 | Верификация исходных текстов программных модулей по критериям |  |
| 6.4.2.6 | Верификация сборки по критериям |  |
| 6.4.2.7 | Верификация документации по критериям |  |

В описании каждой задачи в табл. 11.2 имеется приписка «по критериям». Эти критерии рассмотрены ниже.

***Договор*** должен быть верифицирован (см. задачу 6.4.2.1) по следующим *критериям*:

* возможность поставщика удовлетворять установленным требованиям;
* непротиворечивость требований и охват ими потребностей пользователя;
* наличие соответствующих процедур для внесения изменений в установленные требования и для решения проблем;
* наличие процедур по взаимодействию и кооперации между участниками договора;
* наличие критериев и процедур, предусмотренных в соответствии с установленными требованиями.

***Процесс*** должен быть верифицирован (см. задачу 6.4.2.2) по следующим *критериям*:

* соответствие и своевременность установления требований к планированию проекта;
* пригодность, реализуемость, выполнимость в соответствии с планом и условиями договора выбранных для проекта процессов;
* применимость стандартов, процедур и условий к процессам проекта;
* укомплектованность и обученность персонала в соответствии с условиями договора.

***Требования*** должны быть верифицированы (см. задачу 6.4.2.3) по следующим *критериям*:

* непротиворечивость, выполнимость и тестируемость требований к системе;
* распределение требований к системе между объектами технических и программных средств и ручных операций в соответствии с критериями проектирования;
* непротиворечивость, выполнимость, тестируемость и точность отражения требований к системе в требованиях к программным средствам;
* правильность (подтвержденная соответствующими методами) критических требований к программным средствам, в том числе по безопасности и защите.

***Результаты проектирования*** должны быть верифицированы (см. задачу 6.4.2.4) по следующим *критериям*:

* правильность, соответствие установленным требованиям и учет этих требований;
* реализация соответствующей последовательности событий, исходных данных, выходных результатов, интерфейсов, логики; соответствие временным ограничениям и ограничениям размера; обнаружение, локализация и устранение ошибок;
* возможность дальнейшего использования с учетом требований;
* правильность, подтвержденная соответствующими методами, реализации требований безопасности, защиты и других критических требований.

***Исходные тексты программных модулей*** должны быть верифицированы (см. задачу 6.4.2.5) по следующим *критериям*:

* трассируемость с результатами проектирования и требованиями; тестируемость, правильность и соответствие требованиям и стандартам программирования;
* реализация соответствующей последовательности событий, соответствующих интерфейсов, правильных данных и логики управления; завершенность; соответствие временным ограничениям и ограничениям размера; обнаружение, локализация и устранение ошибок;
* соответствие результатам проектирования и требованиям;
* возможность дальнейшего использования с учетом результатов тестирования и требований;
* правильность, подтвержденная соответствующими методами, реализации требований безопасности, защиты и других критических требований.

***Сборка*** должна быть верифицирована (см. задачу 6.4.2.6) по следующим *критериям*:

* полнота и правильность сборки программных компонентов и модулей каждого программного объекта в соответствующий программный объект;
* полнота и правильность сборки технических и программных объектов и ручных операций в систему;
* выполнение задач сборки в соответствии с планом сборки.

***Документация*** должна быть верифицирована (см. задачу 6.4.2.7) по следующим *критериям*:

* соответствие, полнота и непротиворечивость документации;
* своевременность подготовки документации;
* соблюдение установленных процедур управления конфигурацией документов.

11.3.2 **Непрерывность процесса верификации**. Как следует из п. 11.3.1, верификация используется практически на всех стадиях ЖЦ ПП: сначала верифицируется договор, затем своевременность установления требований к проекту (верификация процесса). Далее происходит верификация требований к ПП, результатов проектирования и текстов программных модулей, верификация сборки ПП и документации проекта. Отсюда следует непрерывность процесса верификации, показанная на рис. 11.8:



Рисунок 11.8 – Наглядное представление непрерывность процесса верификации