Оглавление

1. Жизненный цикл и этапы разработки ПО. Эволюция моделей жизненного цикла ПО.

Старейшей парадигмой процесса разработки ПО является классический жизненный цикл (автор Уинстон Ройс, 1970) [65].

Очень часто классический жизненный цикл называют каскадной или водопадной моделью, подчеркивая, что разработка рассматривается как последовательность этапов, причем переход на следующий, иерархически нижний этап происходит только после полного завершения работ на текущем этапе (рис. 1.1).

Охарактеризуем содержание основных этапов.

Подразумевается, что разработка начинается на системном уровне и проходит через анализ, проектирование, кодирование, тестирование и сопровождение. При этом моделируются действия стандартного инженерного цикла.

Системный анализ задает роль каждого элемента в компьютерной системе, взаимодействие элементов друг с другом. Поскольку ПО является лишь частью большой системы, то анализ начинается с определения требований ко всем системным элементам и назначения подмножества этих требований программному «элементу». Необходимость системного подхода явно проявляется, когда формируется интерфейс ПО с другими элементами (аппаратурой, людьми, базами данных). На этом же этапе начинается решение задачи планирования проекта ПО. В ходе планирования проекта определяются объем проектных работ и их риск, необходимые трудозатраты, формируются рабочие задачи и план-график работ.

Анализ требований относится к программному элементу — программному обеспечению. Уточняются и детализируются его функции, характеристики и интерфейс.

Все определения документируются в *спецификации анализа.* Здесь же завершается решение задачи планирования проекта.

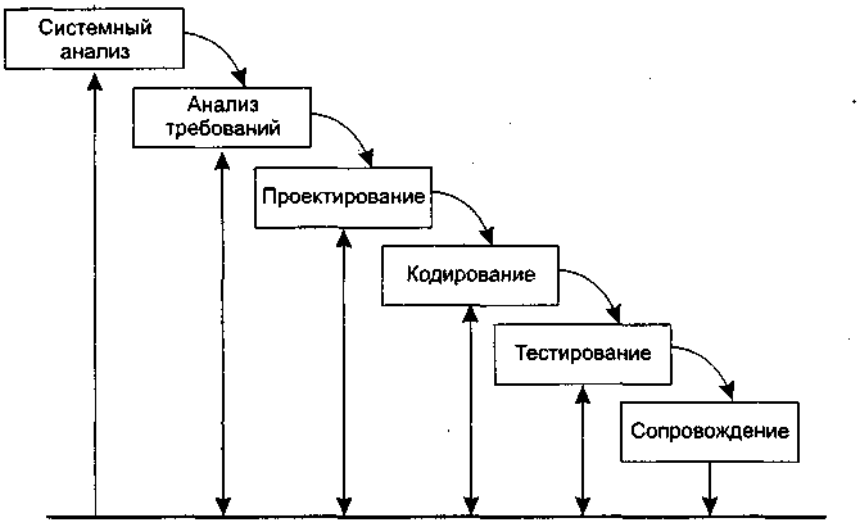


Рис. 1.1. Классический жизненный цикл разработки ПО

Проектирование состоит в создании представлений:

- архитектуры ПО;
- модульной структуры ПО;
- алгоритмической структуры ПО;
- структуры данных;
- входного и выходного интерфейса (входных и выходных форм данных).

Исходные данные для проектирования содержатся в *спецификации анализа,* то есть в ходе проектирования выполняется трансляция требований к ПО во множество проектных представлений. При решении задач проектирования основное внимание уделяется качеству будущего программного продукта.

Кодирование состоит в переводе результатов проектирования в текст на языке программирования.

Тестирование — выполнение программы для выявления дефектов в функциях, логике и форме реализации программного продукта.

Сопровождение — это внесение изменений в эксплуатируемое ПО. Цели изменений:

- исправление ошибок;
- адаптация к изменениям внешней для ПО среды;
- усовершенствование ПО по требованиям заказчика.

Сопровождение ПО состоит в повторном применении каждого из предшествующих шагов (этапов) жизненного цикла к существующей программе но не в разработке новой программы.

Как и любая инженерная схема, классический жизненный цикл имеет достоинства и недостатки.

Достоинства классического жизненного цикла: дает план и временной график по всем этапам проекта, упорядочивает ход конструирования.

Недостатки классического жизненного цикла:

1) реальные проекты часто требуют отклонения от стандартной последовательности шагов;
2) цикл основан на точной формулировке исходных требований к ПО (реально в начале проекта требования заказчика определены лишь частично);
3) результаты проекта доступны заказчику только в конце работы.

2. Каскадная модель ЖЦ

Первоначально (1970-1985 годы) была предложена и использовалась каскадная схема разработки программного обеспечения (рис. 1.10), которая предполагала, что переход на следующую стадию осуществляется после того, как полностью будут завершены проектные операции предыдущей стадии и получены все исходные данные для следующей стадии. Достоинствами такой схемы являются:

• получение в конце каждой стадии законченного набора проектной документации, отвечающего требованиям полноты и согласованности;

• простота планирования процесса разработки.

Именно такую схему и используют обычно при блочно-иерархическом подходе к разработке сложных технических объектов, обеспечивая очень высокие параметры эффективности разработки. Однако данная схема оказалась применимой только к созданию систем, для которых в самом начале разработки удавалось точно и полно сформулировать все требования.Это уменьшало вероятность возникновения в процессе разработки проблем, связанных с принятием неудачного решения на предыдущих стадиях. На практике такие разработки встречается крайне редко.

Рис. 1.10. Каскадная схема разработки программного обеспечения

В целом необходимость возвратов на предыдущие стадии обусловлена следующими причинами:

• неточные спецификации, уточнение которых в процессе разработки может привести к необходимости пересмотра уже принятых решений;

• изменение требований заказчика непосредственно в процессе разработки;

• быстрое моральное устаревание используемых технических и программных средств;

• отсутствие удовлетворительных средств описания разработки на стадиях постановки задачи, анализа и проектирования.

Отказ от уточнения (изменения) спецификаций приведет к тому, что законченный продукт не будет удовлетворять потребности пользователей. При отказе от учета смены оборудования и программной среды пользователь получит морально устаревший продукт. А отказ от пересмотра неудачных проектных решений приводит к ухудшению структуры

программного продукта и, соответственно, усложнит, растянет по времени и удорожит процесс его создания. Реальный процесс разработки, таким образом, носит итерационный характер.
Модель с промежуточным контролем. Схема, поддерживающая итерационный характер процесса разработки, была названа схемой с промежуточным контролем (рис. 1.11). Контроль, который выполняется по данной схеме после завершения каждого этапа, позволяет при необходимости вернуться на любой уровень и внести необходимые изменения.

Основная опасность использования такой схемы связана с тем, что разработка никогда не будет завершена, постоянно находясь в состоянии уточнения и усовершенствования.

Рис. 1.11. Схема разработки программного обеспечения с промежуточным контролем

Примечание. Народная мудрость в подобных случаях говорит «лучшее - враг хорошего». Осталось только понять, что можно считать «хорошим» и как все-таки добиться лучшего...

3. Спиральная модель ЖЦ ПО.

Спиральная модель — классический пример применения эволюционной стратегии конструирования.

Спиральная модель (автор Барри Боэм, 1988) базируется на лучших свойствах классического жизненного цикла и макетирования, к которым добавляется новый элемент — анализ риска, отсутствующий в этих парадигмах [19].

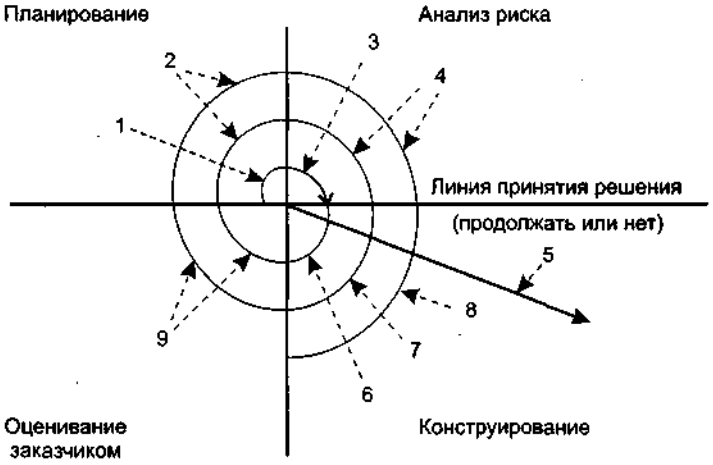


Рис. 1.6. Спиральная модель: *1* — начальный сбор требований и планирование проекта; *2* — та же работа, но на основе рекомендаций заказчика; 3 — анализ риска на основе начальных требований; *4* — анализ риска на основе реакции заказчика; 5 — переход к комплексной системе; 6 — начальный макет системы; 7 — следующий уровень макета; *8* — сконструированная система; 9 — оценивание заказчиком

Как показано на рис. 1.6, модель определяет четыре действия, представляемые четырьмя квадрантами спирали.

1. Планирование — определение целей, вариантов и ограничений.
2. Анализ риска — анализ вариантов и распознавание/выбор риска.
3. Конструирование — разработка продукта следующего уровня.
4. Оценивание — оценка заказчиком текущих результатов конструирования.

Интегрирующий аспект спиральной модели очевиден при учете радиального измерения спирали. С каждой итерацией по спирали (продвижением от центра к периферии) строятся все более полные версии ПО.

В первом витке спирали определяются начальные цели, варианты и ограничения, распознается и анализируется риск. Если анализ риска показывает неопределенность требований, на помощь разработчику и заказчику приходит макетирование (используемое в квадранте конструирования). Для дальнейшего определения проблемных и уточненных требований может быть использовано моделирование. Заказчик оценивает инженерную (конструкторскую) работу и вносит предложения по модификации (квадрант оценки заказчиком). Следующая фаза планирования и анализа риска базируется на предложениях заказчика. В каждом цикле по спирали результаты анализа риска формируются в виде «продолжать, не продолжать». Если риск слишком велик, проект может быть остановлен.

В большинстве случаев движение по спирали продолжается, с каждым шагом продвигая разработчиков к более общей модели системы. В каждом цикле по спирали требуется конструирование (нижний правый квадрант), которое может быть реализовано классическим жизненным циклом или макетированием. Заметим, что количество действий по разработке (происходящих в правом нижнем квадранте) возрастает по мере продвижения от центра спирали.

Достоинства спиральной модели:

1) наиболее реально (в виде эволюции) отображает разработку программного обеспечения;

2) позволяет явно учитывать риск на каждом витке эволюции разработки;

3) включает шаг системного подхода в итерационную структуру разработки;

4) использует моделирование для уменьшения риска и совершенствования программного изделия.

Недостатки спиральной модели:

1) новизна (отсутствует достаточная статистика эффективности модели);

2) повышенные требования к заказчику;

3) трудности контроля и управления временем разработки.

4. Инкрементальная модель.

Инкрементная модель является классическим примером инкрементной стратегии конструирования (рис. 1.4). Она объединяет элементы последовательной водопадной модели с итерационной философией макетирования.

Каждая линейная последовательность здесь вырабатывает поставляемый инкремент ПО. Например, ПО для обработки слов в 1-м инкременте реализует функции базовой обработки файлов, функции редактирования и документирования; во 2-м инкременте — более сложные возможности редактирования и документирования; в 3-м инкременте — проверку орфографии и грамматики; в 4-м инкременте — возможности компоновки страницы.

Первый инкремент приводит к получению базового продукта, реализующего базовые требования (правда, многие вспомогательные требования остаются нереализованными).

План следующего инкремента предусматривает модификацию базового продукта, обеспечивающую дополнительные характеристики и функциональность.

По своей природе инкрементный процесс итеративен, но, в отличие от макетирования, инкрементная модель обеспечивает на каждом инкременте работающий продукт.

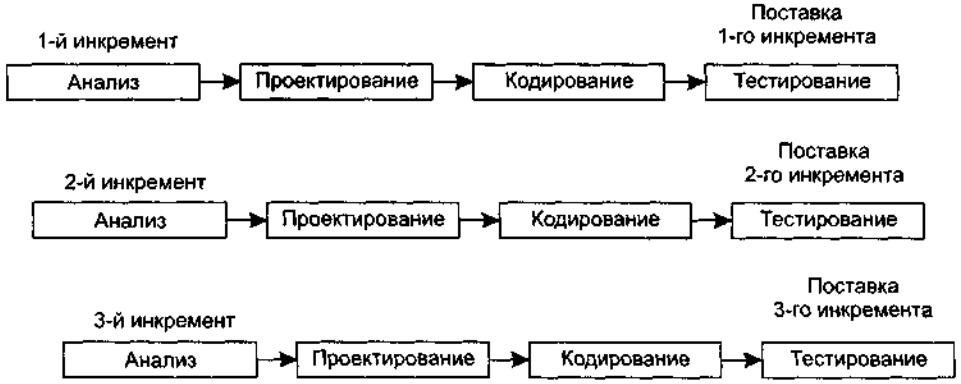


Рис. 1.4. Инкрементная модель

Забегая вперед, отметим, что современная реализация инкрементного подхода — экстремальное программирование XP (Кент Бек, 1999) [10]. Оно ориентировано на очень малые приращения функциональности.

5. Принципы методологии экстремального программирования.

Экстремальное программирование (eXtreme Programming, XP) — облегченный (подвижный) процесс (или методология), главный автор которого — Кент Бек (1999) [11]. XP-процесс ориентирован на группы малого и среднего размера, строящие программное обеспечение в условиях неопределенных или быстро изменяющихся требований. XP-группу образуют до 10 сотрудников, которые размещаются в одном помещении.

Основная идея XP — устранить высокую стоимость изменения, характерную для приложений с использованием объектов, паттернов* и реляционных баз данных. Поэтому XP-процесс должен быть высокодинамичным процессом. XP-группа имеет дело с изменениями требований на всем протяжении итерационного цикла разработки, причем цикл состоит из очень коротких итераций. Четырьмя базовыми действиями в XP-цикле являются: кодирование, тестирование, выслушивание заказчика и проектирование. Динамизм обеспечивается с помощью четырех характеристик: непрерывной связи с заказчиком (и в пределах группы), простоты (всегда выбирается минимальное решение), быстрой обратной связи (с помощью модульного и функционального тестирования), смелости в проведении профилактики возможных проблем.

* Паттерн является решением типичной проблемы в определенном контексте.

Большинство принципов, поддерживаемых в XP (минимальность, простота, эволюционный цикл разработки, малая длительность итерации, участие пользователя, оптимальные стандарты кодирования и т. д.), продиктованы здравым смыслом и применяются в любом упорядоченном процессе. Просто в XP эти принципы, как показано в табл. 1.2, достигают «экстремальных значений».

Таблица 1.2. Экстремумы в экстремальном программировании

Практика здравого смысла	XP-экстремум	XP-реализация
Проверки кода	Код проверяется все время	Парное программирование
Тестирование	Тестирование выполняется все время, даже с помощью заказчиков	Тестирование модуля, функциональное тестирование
Проектирование	Проектирование является частью ежедневной деятельности каждого разработчика	Реорганизация (refactoring)
Простота	Для системы выбирается простейшее проектное решение, поддерживающее ее текущую функциональность	Самая простая вещь, которая могла бы работать
Архитектура	Каждый постоянно работает над уточнением архитектуры	Метафора
Тестирование интеграции	Интегрируется и тестируется несколько раз в день	Непрерывная интеграция
Короткие итерации	Итерации являются предельно короткими, продолжаются секунды, минуты, часы, а не недели, месяцы или годы	Игра планирования

Тот, кто принимает принцип «минимального решения» за хакерство, ошибается, в действительности XP — строго упорядоченный процесс. Простые решения, имеющие высший приоритет, в настоящее время рассматриваются как наиболее ценные части системы, в отличие от проектных решений, которые пока не нужны, а могут (в условиях изменения требований и операционной среды) и вообще не понадобиться.

Базис XP образуют перечисленные ниже двенадцать методов.

1. Игра планирования (Planning game) — быстрое определение области действия следующей реализации путем объединения деловых приоритетов и технических оценок. Заказчик формирует область действия, приоритетность и сроки с точки зрения бизнеса, а разработчики оценивают и прослеживают продвижение (прогресс).
2. Частая смена версий (Small releases) — быстрый запуск в производство простой системы. Новые версии реализуются в очень коротком (двухнедельном) цикле.
3. Метафора (Metaphor) — вся разработка проводится на основе простой, общедоступной истории о том, как работает вся система.
4. Простое проектирование (Simple design) — проектирование выполняется настолько просто, насколько это возможно в данный момент.
5. Тестирование (Testing) — непрерывное написание тестов для модулей, которые должны выполняться безупречно; заказчики пишут тесты для демонстрации законченности функций. «Тестируй, а затем кодируй» означает, что входным критерием для написания кода является «отказавший» тестовый вариант.
6. Реорганизация (Refactoring) — система реструктурируется, но ее поведение не изменяется; цель — устранить дублирование, улучшить взаимодействие, упростить систему или добавить в нее гибкость.
7. Парное программирование (Pair programming) — весь код пишется двумя программистами, работающими на одном компьютере.
8. Коллективное владение кодом (Collective ownership) — любой разработчик может улучшать любой код системы в любое время.
9. Непрерывная интеграция (Continuous integration) — система интегрируется и строится много раз в день, по мере завершения каждой задачи. Непрерывное регрессионное тестирование, то есть повторение предыдущих тестов, гарантирует, что изменения требований не приведут к регрессу функциональности.
10. 40-часовая неделя (40-hour week) — как правило, работают не более 40 часов в неделю. Нельзя удваивать рабочую неделю за счет сверхурочных работ.
11. Локальный заказчик (On-site customer) — в группе все время должен находиться представитель заказчика, действительно готовый отвечать на вопросы разработчиков.
12. Стандарты кодирования (Coding standards) — должны выдерживаться правила, обеспечивающие одинаковое представление программного кода во всех частях программной системы.

Игра планирования и частая смена версий зависят от заказчика, обеспечивающего набор «историй» (коротких описаний), характеризующих работу, которая будет выполняться для каждой версии системы. Версии генерируются каждые две недели, поэтому разработчики и заказчик должны прийти к соглашению о том, какие истории будут осуществлены в пределах двух недель. Полную функциональность, требуемую заказчику, характеризует пул историй; но для следующей двухнедельной итерации из пула выбирается подмножество историй, наиболее важное для заказчика. В любое время в пул могут быть добавлены новые истории, таким образом, требования могут быстро изменяться. Однако процессы двухнедельной генерации основаны на наиболее важных функциях, входящих в текущий пул, следовательно, изменчивость управляется. Локальный заказчик обеспечивает поддержку этого стиля итерационной разработки.

«Метафора» обеспечивает глобальное «видение» проекта. Она могла бы рассматриваться как высокоуровневая архитектура, но XP подчеркивает желательность проектирования при минимизации проектной документации. Точнее говоря, XP предлагает непрерывное перепроектирование (с помощью реорганизации), при котором нет нужды в детализированной проектной документации, а для инженеров сопровождения единственным надежным источником информации является программный код. Обычно после написания кода проектная документация выбрасывается. Проектная документация сохраняется только в том случае, когда заказчик временно теряет способность

придумывать новые истории. Тогда систему помещают в «нафталин» и пишут руководство страниц на пять-десять по «нафталиновому» варианту системы. Использование реорганизации приводит к реализации простейшего решения, удовлетворяющего текущую потребность. Изменения в требованиях заставляют отказываться от всех «общих решений».

Парное программирование — один из наиболее спорных методов в XP, оно влияет на ресурсы, что важно для менеджеров, решающих, будет ли проект использовать XP. Может показаться, что парное программирование удваивает ресурсы, но исследования доказали: парное программирование приводит к повышению качества и уменьшению времени цикла. Для согласованной группы затраты увеличиваются на 15%, а время цикла сокращается на 40-50%. Для Интернет-среды увеличение скорости продаж покрывает повышение затрат. Сотрудничество улучшает процесс решения проблем, улучшение качества существенно снижает затраты сопровождения, которые превышают стоимость дополнительных ресурсов по всему циклу разработки.

Коллективное владение означает, что любой разработчик может изменять любой фрагмент кода системы в любое время. Непрерывная интеграция, непрерывное регрессионное тестирование и парное программирование XP обеспечивают защиту от возникающих при этом проблем.

«Тестируй, а затем кодируй» — эта фраза выражает акцент XP на тестировании. Она отражает принцип, по которому сначала планируется тестирование, а тестовые варианты разрабатываются параллельно анализу требований, хотя традиционный подход состоит в тестировании «черного ящика». Размышление о тестировании в начале цикла жизни — хорошо известная практика конструирования ПО (правда, редко осуществляемая практически).

Основным средством управления XP является метрика, а среда метрик — «большая визуальная диаграмма». Обычно используют 3-4 метрики, причем такие, которые видимы всей группе. Рекомендуемой в XP метрикой является «скорость проекта» — количество историй заданного размера, которые могут быть реализованы в итерации.

При принятии XP рекомендуется осваивать его методы по одному, каждый раз выбирая метод, ориентированный на самую трудную проблему группы. Конечно, все эти методы являются «не более чем правилами» — группа может в любой момент поменять их (если ее сотрудники достигли принципиального соглашения по поводу внесенных изменений). Защитники XP признают, что XP оказывает сильное социальное воздействие, и не каждый может принять его. Вместе с тем, XP — это методология, обеспечивающая преимущества только при использовании законченного набора базовых методов.

Рассмотрим структуру «идеального» XP-процесса. Основным структурным элементом процесса является XP-реализация, в которую многократно вкладывается базовый элемент — XP-итерация. В состав XP-реализации и XP-итерации входят три фазы — исследование, блокировка, регулирование. Исследование (exploration) — это поиск новых требований (историй, задач), которые должна выполнять система. Блокировка (commitment) — выбор для реализации конкретного подмножества из всех возможных требований (иными словами, планирование). Регулирование (steering) — проведение разработки, воплощение плана в жизнь.

XP рекомендует: первая реализация должна иметь длительность 2-6 месяцев, продолжительность остальных реализаций — около двух месяцев, каждая итерация длится приблизительно две недели, а численность группы разработчиков не превышает 10 человек. XP-процесс для проекта с семью реализациями, осуществляемый за 15 месяцев, показан на рис. 1.8.

Процесс инициируется начальной исследовательской фазой.

Фаза исследования, с которой начинается любая реализация и итерация, имеет клапан «пропуска», на этой фазе принимается решение о целесообразности дальнейшего продолжения работы.

Предполагается, что длительность первой реализации составляет 3 месяца, длительность второй — седьмой реализаций — 2 месяца. Вторая — седьмая реализации образуют период сопровождения, характеризующий природу XP-проекта. Каждая итерация длится две недели, за исключением тех, которые относят к поздней стадии реализации — «запуску в производство» (в это время темп итерации ускоряется).

Наиболее трудна первая реализация — пройти за три месяца от обычного старта (скажем, отдельный сотрудник не зафиксировал никаких требований, не определены ограничения) к поставке заказчику системы промышленного качества очень сложно.

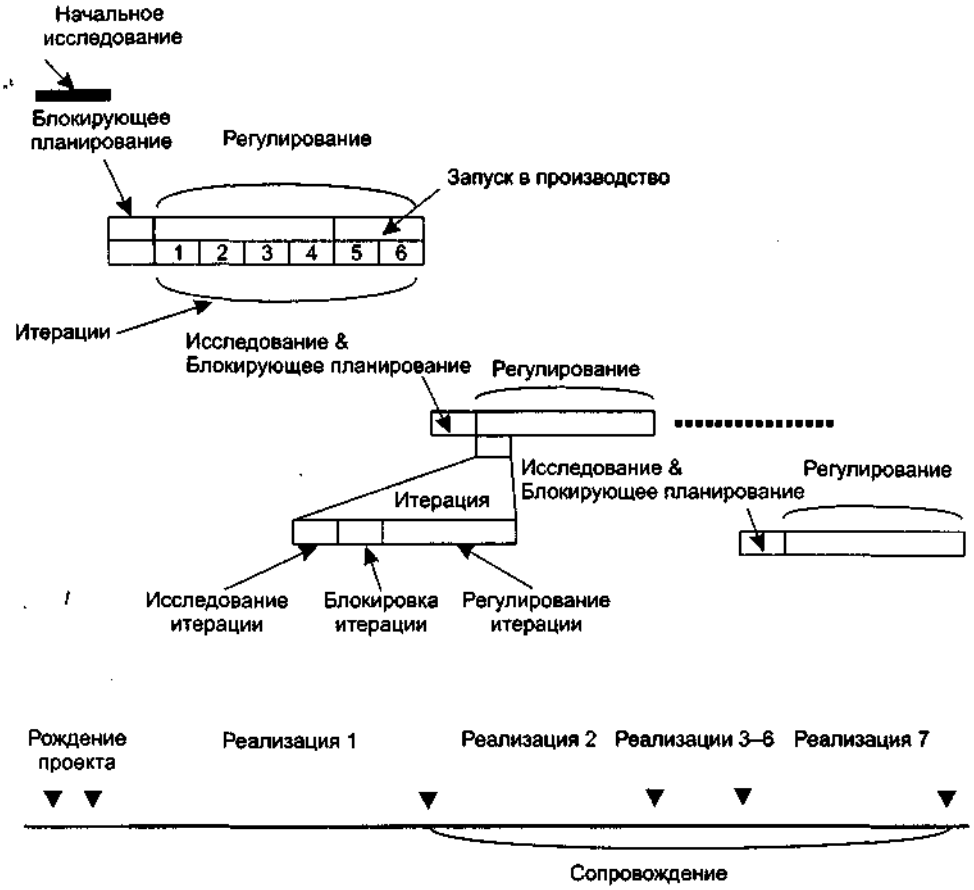


Рис. 1.8. Идеальный XP-процесс

6. Коллективная работа по созданию программ. Поддержка управления проектами с помощью программы MS Project.

Организацию коллективного владения кодом иллюстрирует рис. 15.19.

Коллективное владение кодом позволяет каждому разработчику выдвигать новые идеи в любой части проекта, изменять любую строку программы, добавлять функциональность, фиксировать ошибку и проводить реорганизацию. Один человек просто не в состоянии удержать в голове проект нетривиальной системы. Благодаря коллективному владению кодом снижается риск принятия неверного решения (главным разработчиком) и устраняется нежелательная зависимость проекта от одного человека.

Работа начинается с создания тестов модуля, она должна предшествовать программированию модуля. Тесты необходимо помещать в библиотеку кодов вместе с кодом, который они тестируют. Тесты делают возможным коллективное создание кода и защищают код от неожиданных изменений. В случае обнаружения ошибки также создается тест, чтобы предотвратить ее повторное появление.

Кроме тестов модулей, создаются тесты приемки, они основываются на пользовательских историях. Эти тесты испытывают систему как «черный ящик» и ориентированы на требуемое поведение системы.

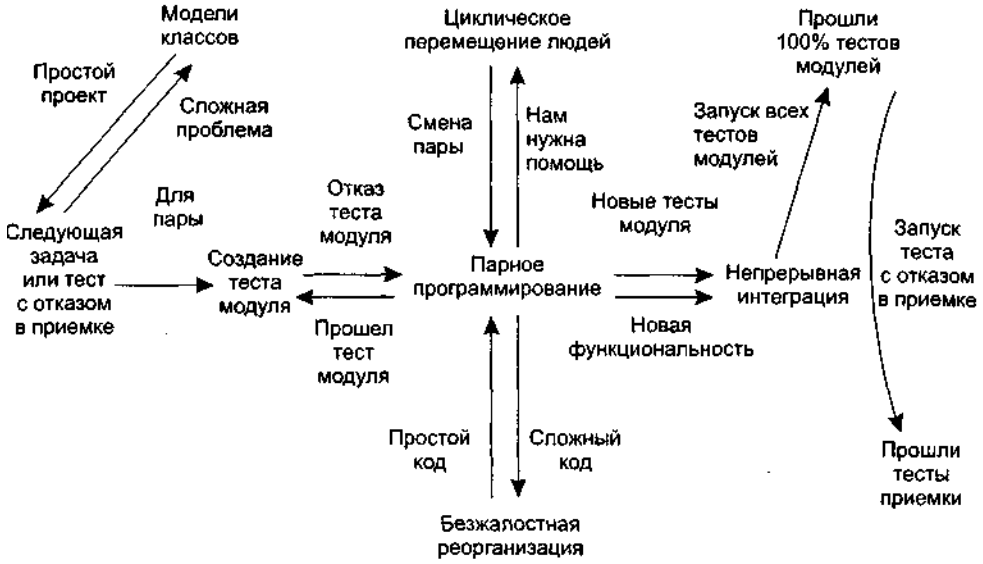


Рис. 15.19. Организация коллективного владения кодом

На основе результатов тестирования разработчики включают в очередную итерацию работу над ошибками. Вообще, следует помнить, что тестирование — один из краеугольных камней XP.

Все коды в проекте создаются парами программистов, работающими за одним компьютером. Парное программирование приводит к повышению качества без дополнительных затрат времени. А это, в свою очередь, уменьшает расходы на будущее сопровождение программной системы.

Оптимальный вариант для парной работы — одновременно сидеть за компьютером, передавая друг другу клавиатуру и мышь. Пока один человек набирает текст и думает (тактически) о создаваемом методе, второй думает (стратегически) о размещении метода в классе.

Во время очередной итерации всех сотрудников перемещают на новые участки работы. Такие перемещения помогают устранить изоляцию знаний и «узкие места». Особенно полезна смена одного из разработчиков при парном программировании.

Замечено, что программисты очень консервативны. Они продолжают использовать код, который трудно сопровождать, только потому, что он все еще работает. Боязнь модификации кода у них в крови. Это приводит к предельному понижению

эффективности систем. В XP считают, что код нужно постоянно обновлять — удалять лишние части, убирать ненужную функциональность. Этот процесс называют реорганизацией кода (refactoring). Поощряется безжалостная реорганизация, сохраняющая простоту проектных решений. Реорганизация поддерживает прозрачность и целостность кода, обеспечивает его легкое понимание, исправление и расширение. На реорганизацию уходит значительно меньше времени, чем на сопровождение устаревшего кода. Увы, нет ничего вечного — когда-то отличный модуль теперь может быть совершенно не нужен.

И еще одна составляющая коллективного владения кодом — непрерывная интеграция.

Без последовательной и частой интеграции результатов в систему разработчики не могут быть уверены в правильности своих действий. Кроме того, трудно вовремя оценить качество выполненных фрагментов проекта и внести необходимые коррективы.

По возможности XP-разработчики должны интегрировать и публично отображать, демонстрировать код каждые несколько часов. Интеграция позволяет объединить усилия отдельных пар и стимулирует повторное использование кода.

Расширенное семейство продуктов Microsoft Project 2002 сочетает в себе средства управления проектами, доступа к информации и поддержки коллективной работы, а также является мощной платформой управления проектами (рис. 1). Новое семейство Microsoft Project 2002 состоит из следующих продуктов (ранее в версиях 98, 2000 был только один продукт):

- Microsoft Project Standard (для бизенес-менеджеров). Это обновленная версия основного продукта управления проектами от Microsoft для управления проектами, в том числе для планирования и формирования графиков выполнения проектов. В сочетании с сервером Microsoft Project Server позволяет организовать коллективную работу над проектом в масштабах рабочей группы на предприятии;

- Microsoft Project Professional (для профессиональных менеджеров). Новое приложение, которое с Microsoft Project Server 2002 обеспечивает поддержку коллективной работы над проектами и предоставляет средства анализа и управления ресурсами в масштабах крупного предприятия (включает всю функциональность Microsoft Project Standard);

- Microsoft Project Server 2002 (сервер групповой работы). Продукт семейства Microsoft .NET Server, который в сочетании с Micros-oft Project Professional и Microsoft Project Standard обеспечивает полноценную поддержку коллективной работы над проектами, а также содержит средства анализа и управления ресурсами в масштабах всего предприятия;

- Microsoft Project Web Access (Microsoft Project Server Client Access Licenses). Web-интерфейс, предоставляющий доступ к информации о проектах и средствах анализа для руководителей и членов групп, которым не нужен полный набор функций управления в Microsoft Project. Эти пользователи могут обращаться к информации о проектах через Web-браузер Internet Explorer (версия не ниже 5.0).

Применяемая система SmartTags является интерфейсным решением, облегчающим работу пользователя, отмечая элементы проекта, которые кажутся «подозрительными». Кроме того, в Microsoft Project Server имеется целый спектр оригинальных возможностей: напоминание по электронной почте, списки оперативных задач с отслеживанием, библиотека проектных документов, отслеживание версий проектов и документов, вопросы по задачам и их отслеживание, управляемые замечания по задачам, отслеживание связей задач, документов, вопросов, замечаний и др.

Microsoft Project Server позволяет легко построить групповой органайзер задач. Пользователи могут создавать списки дел (ToDo) и назначать, кто их может видеть. Используя поставляемую вместе с Microsoft Project Server систему SharePoint, можно организовать библиотеку документов, привязывая их к задачам не ссылкой на файл, а через библиотеку с «версионностью» и блокировками документов.

Замечания к задачам не позволяют организовать переписку класса «Запрос-Ответ», новая функция отслеживания запросов не только позволяет «прикрепить» переписку к задачам проекта, но отлеживать ее прохождение. Участники рабочих групп проекта могут получать от Microsoft Project Server уведомления о необходимости предпринять некоторые действия, например, ответить на запрос о состоянии дел по задачам. Важным является также то, что сервером рассылок уведомлений теперь является Microsoft Project Server и пользователи могут настроить свои персональные опции по работе уведомлений.

Сервер, который раньше назывался Microsoft Project Central, стал называться Microsoft Project Server, а под названием Project Central развертывается новый проектный Web-сервис. Этот Web-сервис (www.proiectcentral.com) сделан для небольших рабочих групп в компаниях разного масштаба.

Среди механизмов управление ресурсами в Microsoft Project 2002 можно отметить:
- блокировки пула ресурсов;
- связь пула ресурсов с MS Exchange и Active Directory;
- поиск ресурсов по признакам и занятости;
- корпоративный пул ресурсов;
- ролевые ресурсы;
- мастер оптимизации ресурсов;
- Web-центр управления ресурсами.

Microsoft Project 2002 Professional позволяет пользователям получить отчетность современного уровня на базе аналитического OLAP-сервиса MS SQL.

Впервые Microsoft оборудовала свой проектный инструментарий средствами моделирования корпоративного класса «что-если?». Используя Portfolio Modeler, можно проанализировать, как разные сценарии развития проектов скажутся на сроках, себестоимости, загрузке ресурсов и т. д.

7. Проектирование программных продуктов с помощью языка универсального языка моделирования UML. Поддержка проектирования в CASE-системе IBM Rational Rose.

Унифицированный язык объектно-ориентированного моделирования Unified Modeling Language (UML) явился средством достижения компромисса между этими подходами. Существует достаточное количество инструментальных средств, поддерживающих с помощью UML жизненный цикл информационных систем, и, одновременно, UML является достаточно гибким для настройки и поддержки специфики деятельности различных команд разработчиков.
UML представляет собой объектно-ориентированный язык моделирования, обладающий следующими основными характеристиками:
является языком визуального моделирования, который обеспечивает разработку репрезентативных моделей для организации взаимодействия заказчика и разработчика ИС, различных групп разработчиков ИС;
содержит механизмы расширения и специализации базовых концепций языка.
UML включает внутренний набор средств моделирования (модулей?) ("ядро"), которые сейчас приняты во многих методах и средствах моделирования. Эти концепции необходимы в большинстве прикладных задач, хотя не каждая концепция необходима в каждой части каждого приложения. Пользователям языка предоставлены возможности:
строить модели на основе средств ядра, без использования механизмов расширения для большинства типовых приложений;
добавлять при необходимости новые элементы и условные обозначения, если они не входят в ядро, или специализировать компоненты, систему условных обозначений (нотацию) и ограничения для конкретных предметных областей.

UML обеспечивает поддержку всех этапов жизненного цикла ИС и предоставляет для этих целей ряд графических средств – диаграмм.

На этапе создания концептуальной модели для описания бизнес-деятельности используются модели бизнес-прецедентов и диаграммы видов деятельности, для описания бизнес-объектов – модели бизнес-объектов и диаграммы последовательностей.

На этапе создания логической модели ИС описание требований к системе задается в виде модели и описания системных прецедентов, а предварительное проектирование осуществляется с использованием диаграмм классов, диаграмм последовательностей и диаграмм состояний.

На этапе создания физической модели детальное проектирование выполняется с использованием диаграмм классов, диаграмм компонентов, диаграмм развертывания.

Ниже приводятся определения и описывается назначение перечисленных диаграмм и моделей применительно к задачам проектирования ИС (в скобках приведены альтернативные названия диаграмм, использующиеся в современной литературе).

Диаграммы прецедентов (диаграммы вариантов использования, use case diagrams) – это обобщенная модель функционирования системы в окружающей среде.

Диаграммы видов деятельности (диаграммы деятельностей, activity diagrams) – модель бизнес-процесса или поведения системы в рамках прецедента.

Диаграммы взаимодействия (interaction diagrams) – модель процесса обмена сообщениями между объектами, представляется в виде диаграмм последовательностей (sequence diagrams) или кооперативных диаграмм (collaboration diagrams).

Диаграммы состояний (statechart diagrams) – модель динамического поведения системы и ее компонентов при переходе из одного состояния в другое.

Диаграммы классов (class diagrams) – логическая модель базовой структуры системы, отражает статическую структуру системы и связи между ее элементами.

Диаграммы базы данных (database diagrams) — модель структуры базы данных, отображает таблицы, столбцы, ограничения и т.п.

Диаграммы компонентов (component diagrams) – модель иерархии подсистем, отражает физическое размещение баз данных, приложений и интерфейсов ИС.

Диаграммы развертывания (диаграммы размещения, deployment diagrams) – модель физической архитектуры системы, отображает аппаратную конфигурацию ИС.

Rational Rose - мощное CASE-средство для проектирования программных систем любой сложности. Одним из достоинств этого программного продукта будет возможность использования диаграмм на языке UML. Можно сказать, что Rational Rose является графическим редактором UML диаграмм.

В распоряжение проектировщика системы Rational Rose предоставляет следующие типы диаграмм, последовательное создание которых позволяет получить полное представление о всей проектируемой системе и об отдельных ее компонентах :

Use case diagram (диаграммы прецедентов);
Deployment diagram (диаграммы топологии);
Statechart diagram (диаграммы состояний);
Activity diagram (диаграммы активности);
Interaction diagram (диаграммы взаимодействия);
Sequence diagram (диаграммы последовательностей действий);
Collaboration diagram (диаграммы сотрудничества);
Class diagram (диаграммы классов);
Component diagram (диаграммы компонент).

8. Диаграммы вариантов использования. Использование диаграмм для представления функций разрабатываемого ПО.

Любые (в том числе и программные) системы проектируются с учетом того, что в процессе своей работы они будут использоваться людьми и/или взаимодействовать с другими системами. Сущности, с которыми взаимодействует система в процессе своей работы, называются экторами, причем каждый эктор ожидает, что система будет вести себя строго определенным, предсказуемым образом. Попробуем дать более строгое определение эктора. Для этого воспользуемся замечательным визуальным словарем по UML Zicom Mentor:
Эктор (actor) - это множество логически связанных ролей, исполняемых при взаимодействии с прецедентами или сущностями (система, подсистема или класс). Эктором может быть человек или другая система, подсистема или класс, которые представляют нечто вне сущности.

Графически эктор изображается либо " человечком ", подобным тем, которые мы рисовали в детстве, изображая членов своей семьи, либо символом класса с соответствующим стереотипом, как показано на рисунке. Обе формы представления имеют один и тот же смысл и могут использоваться в диаграммах. "Стереотипированная" форма чаще применяется для представления системных экторов или в случаях, когда эктор имеет свойства и их нужно отобразить (рис. 2.1).

Внимательный читатель сразу же может задать вопрос: а почему эктор, а не актер? Согласны, слово "эктор" немного режет слух русского человека. Причина же, почему мы говорим именно так, проста - эктор образовано от слова action, что в переводе означает действие. Дословный же перевод слова "эктор" - действующее лицо - слишком длинный и неудобный для употребления. Поэтому мы будем и далее говорить именно так.

Рис. 2.1.

Тот же внимательный читатель мог заметить промелькнувшее в определении эктора слово "прецедент". Что же это такое? Этот вопрос заинтересует нас еще больше, если вспомнить, что сейчас мы говорим о диаграмме прецедентов. Итак,
Прецедент (use-case) - описание отдельного аспекта поведения системы с точки зрения пользователя (Буч).

Определение вполне понятное и исчерпывающее, но его можно еще немного уточнить, воспользовавшись тем же Zicom Mentor 'ом:
Прецедент (use case) - описание множества последовательных событий (включая варианты), выполняемых системой, которые приводят к наблюдаемому эктором результату. Прецедент представляет поведение сущности, описывая взаимодействие между экторами и системой. Прецедент не показывает, "как" достигается некоторый результат, а только "что" именно выполняется.

Прецеденты обозначаются очень простым образом - в виде эллипса, внутри которого указано его название. Прецеденты и экторы соединяются с помощью линий. Часто на одном из концов линии изображают стрелку, причем направлена она к тому, у кого

запрашивают сервис, другими словами, чьими услугами пользуются. Это простое объяснение иллюстрирует понимание прецедентов как сервисов, пропагандируемое компанией IBM.
Прецеденты могут включать другие прецеденты, расширяться ими, наследоваться и т. д. Все эти возможности мы здесь рассматривать не будем. Как уже говорилось выше, цель этого обзора - просто научить читателя выделять диаграмму прецедентов, понимать ее назначение и смысл обозначений, которые на ней встречаются.

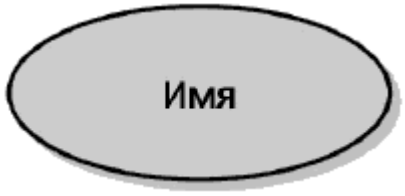


Рис. 2.2.

Кстати, к этому моменту мы уже потратили достаточно много времени на объяснение понятий и их условных обозначений. Наверное, пора уже, наконец, привести пример диаграммы прецедентов. Как вы думаете, что означает эта диаграмма (рис. 2.3)

рис. 2.3

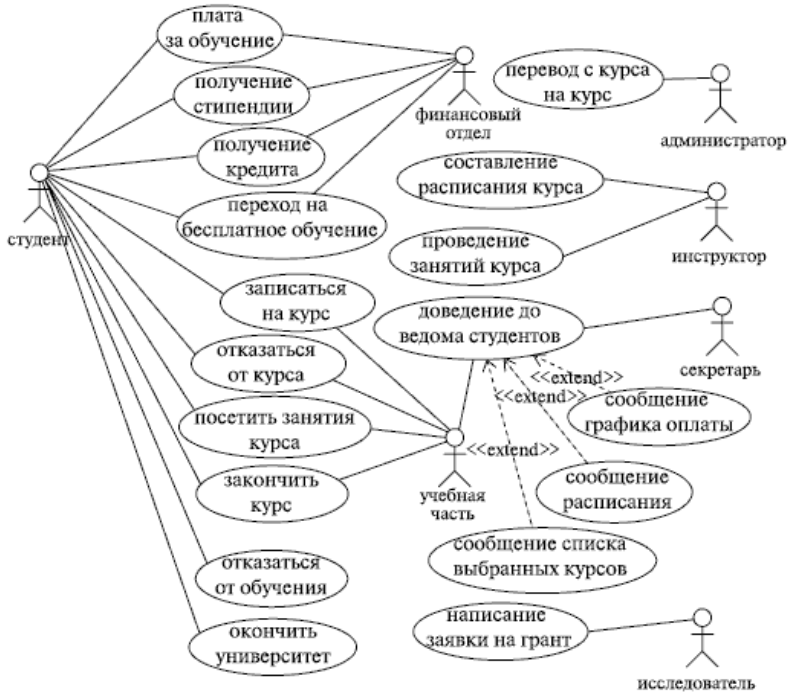


Рис. 2.4.

Из всего сказанного выше становится понятно, что диаграммы прецедентов относятся к той группе диаграмм, которые представляют динамические или поведенческие аспекты системы. Это отличное средство для достижения взаимопонимания между разработчиками, экспертами и конечными пользователями продукта. Как мы уже могли

убедиться, такие диаграммы очень просты для понимания и могут восприниматься и, что немаловажно, обсуждаться людьми, не являющимися специалистами в области разработки ПО.

Подводя итоги, можно выделить такие цели создания диаграмм прецедентов:
определение границы и контекста моделируемой предметной области на ранних этапах проектирования;
формирование общих требований к поведению проектируемой системы;
разработка концептуальной модели системы для ее последующей детализации;
подготовка документации для взаимодействия с заказчиками и пользователями системы

9. Диаграммы классов. Использование диаграмм для определения структурных компонентов ПО и связей между ними. Типы связей.

Центральное место в ООАП занимает разработка логической модели системы в виде диаграммы классов. Нотация классов в языке UML проста и интуитивно понятна всем, кто когда-либо имел опыт работы с CASE-инструментариями. Схожая нотация применяется и для объектов - экземпляров класса, с тем различием, что к имени класса добавляется имя объекта и вся надпись подчеркивается.
Диаграмма классов (class diagram) служит для представления статической структуры модели системы в терминологии классов объектно-ориентированного программирования. Диаграмма классов может отражать, в частности, различные взаимосвязи между отдельными сущностями предметной области, такими как объекты и подсистемы, а также описывает их внутреннюю структуру и типы отношений. На данной диаграмме не указывается информация о временных аспектах функционирования системы. С этой точки зрения диаграмма классов является дальнейшим развитием концептуальной модели проектируемой системы.

Диаграмма классов представляет собой некоторый граф, вершинами которого являются элементы типа "классификатор", которые связаны различными типами структурных отношений. Следует заметить, что диаграмма классов может также содержать интерфейсы, пакеты, отношения и даже отдельные экземпляры, такие как объекты и связи. Когда говорят о данной диаграмме, имеют в виду статическую структурную модель проектируемой системы. Поэтому диаграмму классов принято считать графическим представленном таких структурных взаимосвязей логической модели системы, которые не зависят или инвариантны от времени.

Диаграмма классов состоит из множества элементов, которые в совокупности отражают декларативные знания о предметной области. Эти знания интерпретируются в базовых понятиях языка UML, таких как классы, интерфейсы и отношения между ними и их составляющими компонентами. При этом отдельные компоненты этой диаграммы могут образовывать пакеты для представления более общей модели системы. Если диаграмма классов является частью некоторого пакета, то ее компоненты должны соответствовать элементам этого пакета, включая возможные ссылки на элементы из других пакетов.

Кроме внутреннего устройства или структуры классов на соответствующей диаграмме указываются различные отношения между классами. При этом совокупность типов таких отношений фиксирована в языке UML и предопределена семантикой этих типов отношений. Базовыми отношениями или связями в языке UML являются:
Отношение зависимости (dependency relationship)
Отношение ассоциации (association relationship)
Отношение обобщения (generalization relationship)
Отношение реализации (realization relationship)

Каждое из этих отношений имеет собственное графическое представление на диаграмме, которое отражает взаимосвязи между объектами соответствующих классов.

Отношение зависимости

Отношение зависимости в общем случае указывает некоторое семантическое отношение между двумя элементами модели или двумя множествами таких элементов, которое не является отношением ассоциации, обобщения или реализации. Оно касается только самих

элементов модели и не требует множества отдельных примеров для пояснения своего смысла. Отношение зависимости используется в такой ситуации, когда некоторое изменение одного элемента модели может потребовать изменения другого зависимого от него элемента модели.

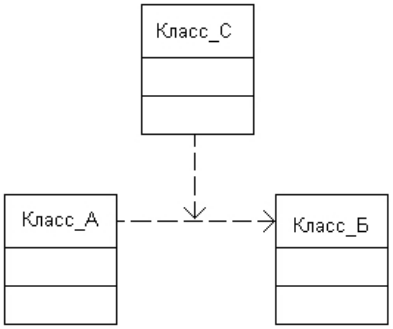


Рис. 5.3. Графическое изображение отношения зависимости на диаграмме классов

Отношение зависимости является наиболее общей формой отношения в языке UML. Все другие типы рассматриваемых отношений можно считать частным случаем данного отношения. Однако важность выделения специфических семантических свойств и дополнительных характеристик для других типов отношений обусловливают их самостоятельное рассмотрение при построении диаграмм.

Отношение ассоциации

Отношение ассоциации соответствует наличию некоторого отношения между классами. Данное отношение обозначается сплошной линией с дополнительными специальными символами, которые характеризуют отдельные свойства конкретной ассоциации. В качестве дополнительных специальных символов могут использоваться имя ассоциации, а также имена и кратность классов-ролей ассоциации. Имя ассоциации является необязательным элементом ее обозначения. Если оно задано, то записывается с заглавной (большой) буквы рядом с линией соответствующей ассоциации.

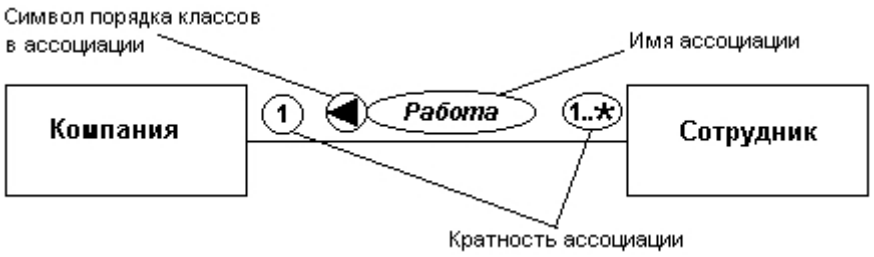


Рис. 5.5. Графическое изображение отношения бинарной ассоциации между классами

Отношение агрегации

Отношение агрегации имеет место между несколькими классами в том случае, если один из классов представляет собой некоторую сущность, включающую в себя в качестве составных частей другие сущности.

Данное отношение имеет фундаментальное значение для описания структуры сложных систем, поскольку применяется для представления системных взаимосвязей типа "часть-целое". Раскрывая внутреннюю структуру системы, отношение агрегации показывает, из каких компонентов состоит система и как они связаны между собой. С точки зрения модели отдельные части системы могут выступать как в виде элементов, так и в виде подсистем, которые, в свою очередь, тоже могут образовывать составные компоненты или подсистемы. Это отношение по своей сути описывает декомпозицию или разбиение сложной системы на более простые составные части, которые также могут быть подвергнуты декомпозиции, если в этом возникнет необходимость в последующем.

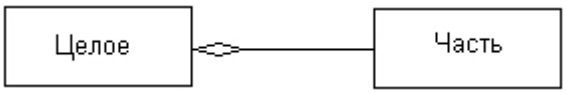


Рис. 5.8. Графическое изображение отношения агрегации в языке UML

Отношение композиции

Отношение композиции, как уже упоминалось ранее, является частным случаем отношения агрегации. Это отношение служит для выделения специальной формы отношения "часть-целое", при которой составляющие части в некотором смысле находятся внутри целого. Специфика взаимосвязи между ними заключается в том, что части не могут выступать в отрыве от целого, т. е. с уничтожением целого уничтожаются и все его составные части.

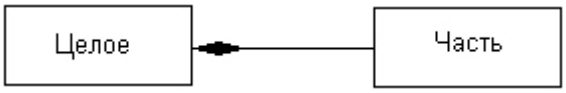


Рис. 5.10. Графическое изображение отношения композиции в языке UML

В качестве дополнительных обозначений для отношений композиции и агрегации могут использоваться дополнительные обозначения, применяемые для отношения ассоциации. А именно, указание кратности класса ассоциации и имени данной ассоциации, которые не являются обязательными. Применительно к описанному выше примеру класса "Окно_программы" его диаграмма классов может иметь следующий вид (рис. 5.11).

Отношение обобщения

Отношение обобщения является обычным таксономическим отношением между более общим элементом (родителем или предком) и более частным или специальным элементом (дочерним или потомком). Данное отношение может использоваться для представления взаимосвязей между пакетами, классами, вариантами использования и другими элементами языка UML.

Применительно к диаграмме классов данное отношение описывает иерархическое строение классов и наследование их свойств и поведения. При этом предполагается, что класс-потомок обладает всеми свойствами и поведением класса-предка, а также имеет свои собственные свойства и поведение, которые отсутствуют у класса-предка. На диаграммах отношение обобщения обозначается сплошной линией с треугольной стрелкой на одном из концов (рис. 5.12). Стрелка указывает на более общий класс (класс-

предок или суперкласс), а ее отсутствие - на более специальный класс (класс-потомок или подкласс).

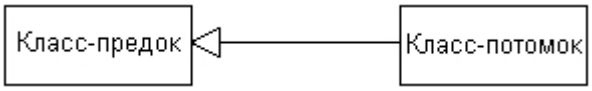


Рис. 5.12. Графическое изображение отношения обобщения в языке UML

С целью упрощения обозначений на диаграмме классов совокупность линий, обозначающих одно и то же отношение обобщения, может быть объединена в одну линию. В этом случае данные отдельные линии изображаются сходящимися к единственной .стрелке, имеющей с ними общую точку пересечения.

Рядом со стрелкой обобщения может размещаться строка текста, указывающая на некоторые дополнительные свойства этого отношения. Данный текст будет относиться ко всем линиям обобщения, которые идут к классам-потомкам. Другими словами, отмеченное свойство касается всех подклассов данного отношения. При этом текст следует рассматривать как ограничение, и тогда он записывается в фигурных скобках.

В качестве ограничений могут быть использованы следующие ключевые слова языка UML:
{complete} - означает, что в данном отношении обобщения специфицированы все классы-потомки, и других классов-потомков у данного класса-предка быть не может. Пример - класс Клиент_банка является предком для двух классов: Физическое_лицо и Компания, и других классов-потомков он не имеет. На соответствующей диаграмме классов это можно указать явно, записав рядом с линией обобщения данную строку-ограничение;
{disjoint} - означает, что классы-потомки не могут содержать объектов, одновременно являющихся экземплярами двух или более классов. В приведенном выше примере это условие также выполняется, поскольку предполагается, что никакое конкретное физическое лицо не может являться одновременно и конкретной компанией. В этом случае рядом с линией обобщения можно записать данную строку-ограничение;
{incomplete} - означает случай, противоположный первому. А именно, предполагается, что на диаграмме указаны не все классы-потомки. В последующем возможно восполнить их перечень не изменяя уже построенную диаграмму. Пример - диаграмма класса "Автомобиль", для которой указание всех без исключения моделей автомобилей соизмеримо с созданием соответствующего каталога. С другой стороны, для отдельной задачи, такой как разработка системы продажи автомобилей конкретных моделей, в этом нет необходимости. Но указать неполноту структуры классов-потомков все же следует;
{overlapping} - означает, что отдельные экземпляры классов-потомков могут принадлежать одновременно нескольким классам. Пример - класс "Многоугольник" является классом-предком для класса "Прямоугольник" и класса "Ромб". Однако существует отдельный класс "Квадрат", экземпляры которого одновременно являются объектами первых двух классов. Вполне естественно такую ситуацию указать явно с помощью данной строки-ограничения.

10. Диаграммы состояний. Использование диаграмм для представления процесса изменения состояний проектируемой системы.

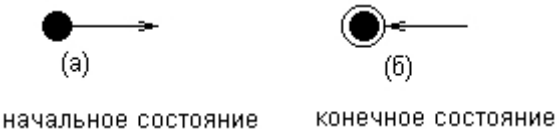
Понятие состояния (state) является фундаментальным не только в метамоде-ли языка UML, но и в прикладном системном анализе. Ранее в главе 1 кратко были рассмотрены особенности представления динамических характеристик сложных систем, традиционно используемых для моделирования поведения. Вся концепция динамической системы основывается на понятии состояния системы. Однако семантика состояния в языке UML имеет целый ряд специфических особенностей.

В языке UML под состоянием понимается абстрактный метакласс, используемый для моделирования отдельной ситуации, в течение которой имеет место выполнение некоторого условия. Состояние может быть задано в виде набора конкретных значений атрибутов класса или объекта, при этом изменение их отдельных значений будет отражать изменение состояния моделируемого класса или объекта.

Рис. 6.2. Графическое изображение состояний на диаграмме состояний

Состояние на диаграмме изображается прямоугольником со скругленными вершинами (рис. 6.2). Этот прямоугольник, в свою очередь, может быть разделен на две секции горизонтальной линией. Если указана лишь одна секция, то в ней записывается только имя состояния (рис. 6.2, а). В противном случае в первой из них записывается имя состояния, а во второй - список некоторых внутренних действий или переходов в данном состоянии (рис. 6.2, б). При этом под действием в языке UML понимают некоторую атомарную операцию, выполнение которой приводит к изменению состояния или возврату некоторого значения (например, "истина" или "ложь").

Начальное состояние представляет собой частный случай состояния, которое не содержит никаких внутренних действий (псевдосостояния). В этом состоянии находится объект по умолчанию в начальный момент времени. Оно служит для указания на диаграмме состояний графической области, от которой начинается процесс изменения состояний. Графически начальное состояние в языке UML обозначается в виде закрашенного кружка (рис. 6.4, а), из которого может только выходить стрелка, соответствующая переходу.



Конечное (финальное) состояние представляет собой частный случай состояния, которое также не содержит никаких внутренних действий (псевдосостояния). В этом состоянии будет находиться объект по умолчанию после завершения работы автомата в конечный момент времени. Оно служит для указания на диаграмме состояний графической области, в которой завершается процесс изменения состояний или жизненный цикл данного объекта. Графически конечное состояние в языке UML обозначается в виде закрашенного кружка, помещенного в окружность (рис. 6.4, б), в которую может только входить стрелка, соответствующая переходу.

Простой переход (simple transition) представляет собой отношение между двумя последовательными состояниями, которое указывает на факт смены одного состояния другим. Пребывание моделируемого объекта в первом состоянии может сопровождаться

выполнением некоторых действий, а переход во второе состояние будет возможен после завершения этих действий, а также после удовлетворения некоторых дополнительных условий. В этом случае говорят, что переход срабатывает, Или происходит срабатывание перехода. До срабатывания перехода объект находится в предыдущем от него состоянии, называемым исходным состоянием, или в источнике (не путать с начальным состоянием - это разные понятия), а после его срабатывания объект находится в последующем от него состоянии (целевом состоянии).

Переход осуществляется при наступлении некоторого события: окончания выполнения деятельности (do activity), получении объектом сообщения или приемом сигнала. На переходе указывается имя события Кроме того, на переходе могут указываться действия, производимые объектом в ответ на внешние события при переходе из одного состояния в другое. Срабатывание перехода может зависеть не только от наступления некоторого события, но и от выполнения определенного условия, называемого сторожевым условием. Объект перейдет из одного состояния в другое в том случае, если произошло указанное событие и сторожевое условие приняло значение "истина"

11. Диаграммы деятельности. Использование диаграмм для детализации особенностей алгоритмической реализации выполняемых системой операций.

Для моделирования процесса выполнения операций в языке UML используются так называемые диаграммы деятельности. Применяемая в них графическая нотация во многом похожа на нотацию диаграммы состояний, поскольку на диаграммах деятельности также присутствуют обозначения состояний и переходов. Отличие заключается в семантике состояний, которые используются для представления не деятельностей, а действий, и в отсутствии на переходах сигнатуры событий. Каждое состояние на диаграмме деятельности соответствует выполнению некоторой элементарной операции, а переход в следующее состояние срабатывает только при завершении этой, операции в предыдущем состоянии. Графически диаграмма деятельности представляется в форме графа деятельности, вершинами которого являются состояния действия, а дугами - переходы от одного состояния действия к другому.

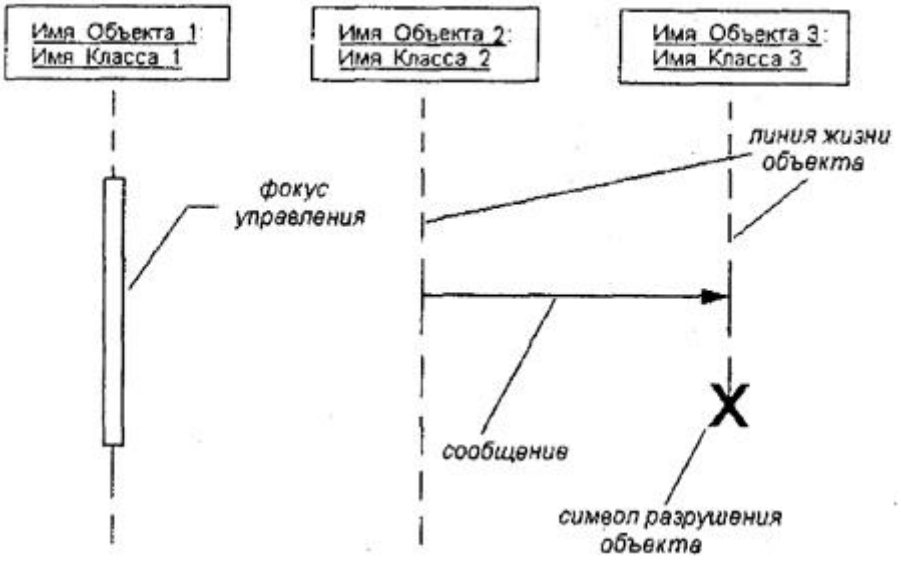
Таким образом, диаграммы деятельности можно считать частным случаем диаграмм состояний. Именно они позволяют реализовать в языке UML особенности процедурного и синхронного управления, обусловленного завершением внутренних деятельностей и действий. Метамодель UML предоставляет для этого необходимые термины и семантику. Основным направлением использования диаграмм деятельности является визуализация особенностей реализации операций классов, когда необходимо представить алгоритмы их выполнения. При этом каждое состояние может являться выполнением операции некоторого класса либо ее части, позволяя использовать диаграммы деятельности для описания реакций на внутренние события системы.

В контексте языка UML деятельность (activity) представляет собой некоторую совокупность отдельных вычислений, выполняемых автоматом. При этом отдельные элементарные вычисления могут приводить к некоторому результату или действию (action). На диаграмме деятельности отображается логика или последовательность перехода от одной деятельности к другой, при этом внимание фиксируется на результате деятельности. Сам же результат может привести к изменению состояния системы или возвращению некоторого значения.
При построении диаграммы деятельности используются только нетриггерные переходы, т. е. такие, которые срабатывают сразу после завершения деятельности или выполнения соответствующего действия. Этот переход переводит деятельность в последующее состояние сразу, как только закончится действие в предыдущем состоянии. На диаграмме такой переход изображается сплошной линией со стрелкой.
В общем случае действия на диаграмме деятельности выполняются над теми или иными объектами. Эти объекты либо инициируют выполнение действий, либо определяют некоторый результат этих действий. При этом действия специфицируют вызовы, которые передаются от одного объекта графа деятельности к другому. Поскольку в таком ракурсе объекты играют определенную роль в понимании процесса деятельности, иногда возникает необходимость явно указать их на диаграмме деятельности.

Для графического представления объектов, как уже упоминалось в главе 5, используются прямоугольник класса, с тем отличием, что имя объекта подчеркивается. Далее после имени может указываться характеристика состояния объекта в прямых скобках. Такие прямоугольники объектов присоединяются к состояниям действия отношением зависимости пунктирной линией со стрелкой. Соответствующая зависимость определяет состояние конкретного объекта после выполнения предшествующего действия.

12. Диаграммы последовательности и кооперации. Использование диаграмм для описания процесса взаимодействия отдельных элементов ПО.

На диаграмме последовательности изображаются исключительно те объекты, которые непосредственно участвуют во взаимодействии и не показываются возможные статические ассоциации с другими объектами. Для диаграммы последовательности ключевым моментом является именно динамика взаимодействия объектов во времени. При этом диаграмма последовательности имеет как бы два измерения. Одно - слева направо в виде вертикальных линий, каждая из которых изображает линию жизни отдельного объекта, участвующего во взаимодействии. Графически каждый объект изображается прямоугольником и располагается в верхней части своей линии жизни (рис. 8.1). Внутри прямоугольника записываются имя объекта и имя класса, разделенные двоеточием. При этом вся запись подчеркивается, что является признаком объекта, который, как известно, представляет собой экземпляр класса.



Крайним слева на диаграмме изображается объект, который является инициатором взаимодействия (объект 1 на рис. 8.1). Правее изображается другой объект, который непосредственно взаимодействует с первым. Таким образом, все объекты на диаграмме последовательности образуют некоторый порядок, определяемый степенью активности этих объектов при взаимодействии друг с другом.

Второе измерение диаграммы последовательности - вертикальная временная ось, направленная сверху вниз. Начальному моменту времени соответствует самая верхняя часть диаграммы. При этом взаимодействия объектов реализуются посредством сообщений, которые посылаются одними объектами другим. Сообщения изображаются в виде горизонтальных стрелок с именем сообщения и также образуют порядок по времени своего возникновения. Другими словами, сообщения, расположенные на диаграмме последовательности выше, инициируются раньше тех, которые расположены ниже. При этом масштаб на оси времени не указывается, поскольку диаграмма последовательности моделирует лишь временную упорядоченность взаимодействий типа "раньше-позже". Линия жизни объекта (object lifeline) изображается пунктирной вертикальной линией, ассоциированной с единственным объектом на диаграмме последовательности. Линия жизни служит для обозначения периода времени, в течение которого объект существует в системе и, следовательно, может потенциально участвовать во всех ее взаимодействиях. Если объект существует в системе постоянно, то и его линия жизни должна продолжаться по всей плоскости диаграммы последовательности от самой верхней ее части до самой нижней (объекты 1 и 2 на рис. 8.1).

Прежде всего, на диаграмме кооперации в виде прямоугольников изображаются участвующие во взаимодействии объекты, содержащие имя объекта, его класс и, возможно, значения атрибутов. Далее, как и на диаграмме классов, указываются ассоциации между объектами в виде различных соединительных линий. При этом можно явно указать имена ассоциации и ролей, которые играют объекты в данной ассоциации. Дополнительно могут быть изображены динамические связи - потоки сообщений. Они представляются также в виде соединительных линий между объектами, над которыми располагается стрелка с указанием направления, имени сообщения и порядкового номера в общей последовательности инициализации сообщений.

В отличие от диаграммы последовательности, на диаграмме кооперации изображаются только отношения между объектами, играющими определенные роли во взаимодействии. С другой стороны, на этой диаграмме не указывается время в виде отдельного измерения. Поэтому последовательность взаимодействий и параллельных потоков может быть определена с помощью порядковых номеров. Следовательно, если необходимо явно специфицировать взаимосвязи между объектами в реальном времени, лучше это делать на диаграмме последовательности.

Поведение системы может описываться на уровне отдельных объектов, которые обмениваются между собой сообщениями, чтобы достичь нужной цели или реализовать некоторый сервис. С точки зрения аналитика или конструктора важно представить в проекте системы структурные связи отдельных объектов между собой. Такое статическое представление структуры системы как совокупности взаимодействующих объектов и обеспечивает диаграмма кооперации.

Таким образом, с помощью диаграммы кооперации можно описать полный контекст взаимодействий как своеобразный временной "среза" совокупности объектов, взаимодействующих между собой для выполнения определенной задачи или бизнес-цели программной системы.

Рис. 9.1. Общее представление кооперации на диаграммах уровня спецификации

13. Диаграммы размещения. Использование диаграмм для описания физического представления ПО.

Диаграмма развертывания предназначена для визуализации элементов и компонентов программы, существующих лишь на этапе ее исполнения (runtime). При этом представляются только компоненты-экземпляры программы, являющиеся исполнимыми файлами или динамическими библиотеками. Те компоненты, которые не используются на этапе исполнения, на диаграмме развертывания не показываются. Так, компоненты с исходными текстами программ могут присутствовать только на диаграмме компонентов. На диаграмме развертывания они не указываются.

Диаграмма развертывания содержит графические изображения процессоров, устройств, процессов и связей между ними. В отличие от диаграмм логического представления, диаграмма развертывания является единой для системы в целом, поскольку должна всецело отражать особенности ее реализации. Эта диаграмма, по сути, завершает процесс ООАП для конкретной программной системы и ее разработка, как правило, является последним этапом спецификации модели.

Итак, перечислим цели, преследуемые при разработке диаграммы развертывания:
Определить распределение компонентов системы по ее физическим узлам.
Показать физические связи между всеми узлами реализации системы на этапе ее исполнения.
Выявить узкие места системы и реконфигурировать ее топологию для достижения требуемой производительности.

Для обеспечения этих требований диаграмма развертывания разрабатывается совместно системными аналитиками, сетевыми инженерами и системотехниками. Далее рассмотрим отдельные элементы, из которых состоят диаграммы развертывания.
Узел (node) представляет собой некоторый физически существующий элемент системы, обладающий некоторым вычислительным ресурсом. В качестве вычислительного ресурса узла может рассматриваться наличие по меньшей мере некоторого объема электронной или магнитооптической памяти и/или процессора.

Кроме собственно изображений узлов на диаграмме развертывания указываются отношения между ними. В качестве отношений выступают физические соединения между узлами и зависимости между узлами и компонентами, изображения которых тоже могут присутствовать на диаграммах развертывания.

Соединения являются разновидностью ассоциации и изображаются отрезками линий без стрелок. Наличие такой линии указывает на необходимость организации физического канала для обмена информацией между соответствующими узлами. Характер соединения может быть дополнительно специфицирован примечанием, помеченным значением или ограничением.

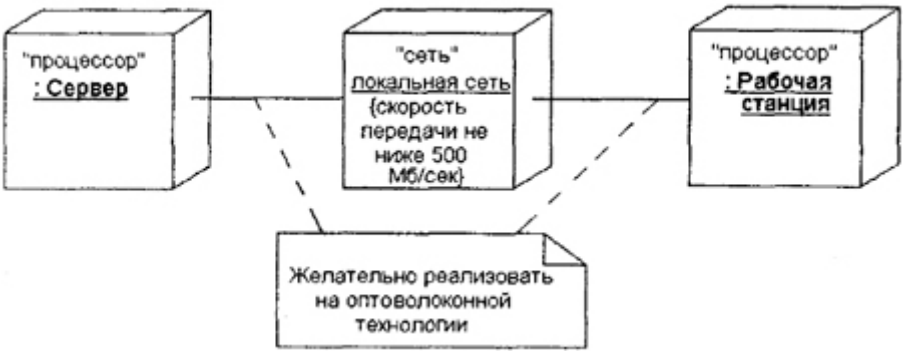


Рис. 11.4. Фрагмент диаграммы развертывания с соединениями меходу узлами

14. Основные понятия и принципы тестирования ПО. Использование способов «белого ящика» и «черного ящика» при тестировании ПО.

Тестирование — процесс выполнения программы с целью обнаружения ошибок. Шаги процесса задаются тестами.

Каждый тест определяет:

- свой набор исходных данных и условий для запуска программы;
- набор ожидаемых результатов работы программы.

Другое название теста — тестовый вариант. Полную проверку программы гарантирует *исчерпывающее тестирование.* Оно требует проверить все наборы исходных данных, все варианты их обработки и включает большое количество тестовых вариантов. Увы, но исчерпывающее тестирование во многих случаях остается только мечтой — срабатывают ресурсные ограничения (прежде всего, ограничения по времени).

Хорошим считают тестовый вариант с высокой вероятностью обнаружения еще не раскрытой ошибки. Успешным называют тест, который обнаруживает до сих пор не раскрытую ошибку.

Целью проектирования тестовых вариантов является систематическое обнаружение различных классов ошибок при минимальных затратах времени и стоимости.

Важен ответ на вопрос: что может тестирование?

Тестирование обеспечивает:

- обнаружение ошибок;
- демонстрацию соответствия функций программы ее назначению;
- демонстрацию реализации требований к характеристикам программы;
- отображение надежности как индикатора качества программы.

А чего не может тестирование? Тестирование не может показать отсутствия дефектов (оно может показывать только присутствие дефектов). Важно помнить это (скорее печальное) утверждение при проведении тестирования.

Рассмотрим информационные потоки процесса тестирования. Они показаны на рис. 6.1.

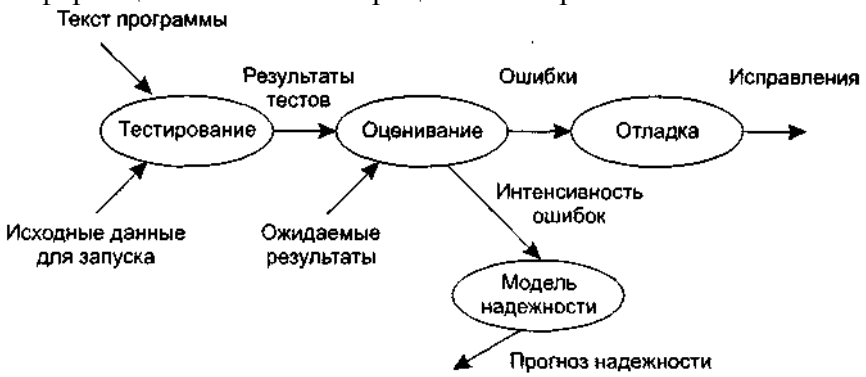


Рис. 6.1. Информационные потоки процесса тестирования

На входе процесса тестирования три потока:

- текст программы;
- исходные данные для запуска программы;
- ожидаемые результаты.

Выполняются тесты, все полученные результаты оцениваются. Это значит, что реальные результаты тестов сравниваются с ожидаемыми результатами. Когда обнаруживается несовпадение, фиксируется ошибка — начинается отладка. Процесс отладки непредсказуем по времени. На поиск места дефекта и исправление может потребоваться час, день, месяц. Неопределенность в отладке приводит к большим трудностям в планировании действий.

После сбора и оценивания результатов тестирования начинается отображение качества и надежности ПО. Если регулярно встречаются серьезные ошибки, требующие проектных изменений, то качество и надежность ПО подозрительны, констатируется необходимость усиления тестирования. С другой стороны, если функции ПО реализованы правильно, а обнаруженные ошибки легко исправляются, может быть сделан один из двух выводов:

- качество и надежность ПО удовлетворительны;
- тесты не способны обнаруживать серьезные ошибки.

В конечном счете, если тесты не обнаруживают ошибок, появляется сомнение в том, что тестовые варианты достаточно продуманы и что в ПО нет скрытых ошибок. Такие ошибки будут, в конечном итоге, обнаруживаться пользователями и корректироваться разработчиком на этапе сопровождения (когда стоимость исправления возрастает в 60-100 раз по сравнению с этапом разработки).

Результаты, накопленные в ходе тестирования, могут оцениваться и более формальным способом. Для этого используют модели надежности ПО, выполняющие прогноз надежности по реальным данным об интенсивности ошибок.

Существуют 2 принципа тестирования программы:

- функциональное тестирование (тестирование «черного ящика»);
- структурное тестирование (тестирование «белого ящика»).

Тестирование «черного ящика»

Известны: функции программы.

Исследуется: работа каждой функции на всей области определения.

Как показано на рис. 6.2, основное место приложения тестов «черного ящика» — интерфейс ПО.

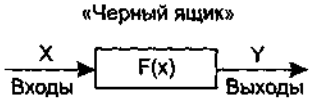


Рис. 6.2. Тестирование «черного ящика»

Эти тесты демонстрируют:

- как выполняются функции программ;
- как принимаются исходные данные;
- как вырабатываются результаты;
- как сохраняется целостность внешней информации.

При тестировании «черного ящика» рассматриваются системные характеристики программ, игнорируется их внутренняя логическая структура. Исчерпывающее тестирование, как правило, невозможно. Например, если в программе 10 входных величин и каждая принимает по 10 значений, то потребуется 10^{10} тестовых вариантов. Отметим также, что тестирование «черного ящика» не реагирует на многие особенности программных ошибок.

Тестирование «белого ящика»

Известна: внутренняя структура программы.

Исследуются: внутренние элементы программы и связи между ними (рис. 6.3).

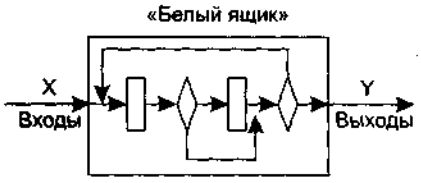


Рис. 6.3. Тестирование «белого ящика»

Объектом тестирования здесь является не внешнее, а внутреннее поведение программы. Проверяется корректность построения всех элементов программы и

правильность их взаимодействия друг с другом. Обычно анализируются управляющие связи элементов, реже — информационные связи. Тестирование по принципу «белого ящика» характеризуется степенью, в какой тесты выполняют или покрывают логику (исходный текст) программы. Исчерпывающее тестирование также затруднительно. Особенности этого принципа тестирования рассмотрим отдельно.

Обычно тестирование «белого ящика» основано на анализе управляющей структуры программы [2], [13]. Программа считается полностью проверенной, если проведено исчерпывающее тестирование маршрутов (путей) ее графа управления.

В этом случае формируются тестовые варианты, в которых:

- гарантируется проверка всех независимых маршрутов программы;
- проходятся ветви True, False для всех логических решений;
- выполняются все циклы (в пределах их границ и диапазонов);
- анализируется правильность внутренних структур данных.

Недостатки тестирования «белого ящика»:

1. Количество независимых маршрутов может быть очень велико. Например, если цикл в программе выполняется k раз, а внутри цикла имеется n ветвлений, то количество маршрутов вычисляется по формуле
$$m = \sum_{i=1}^{k} n^i \ .$$
При $n = 5$ и $k = 20$ количество маршрутов $m = 10^{14}$. Примем, что на разработку, выполнение и оценку теста по одному маршруту расходуется 1 мс. Тогда при работе 24 часа в сутки 365 дней в году на тестирование уйдет 3170 лет.
2. Исчерпывающее тестирование маршрутов не гарантирует соответствия программы исходным требованиям к ней.
3. В программе могут быть пропущены некоторые маршруты.
4. Нельзя обнаружить ошибки, появление которых зависит от обрабатываемых данных (это ошибки, обусловленные выражениями типа if abs (a-b) < eps..., if(a+b+c)/3=a...).

Достоинства тестирования «белого ящика» связаны с тем, что принцип «белого ящика» позволяет учесть особенности программных ошибок:

1. Количество ошибок минимально в «центре» и максимально на «периферии» программы.
2. Предварительные предположения о вероятности потока управления или данных в программе часто бывают некорректны. В результате типовым может стать маршрут, модель вычислений по которому проработана слабо.
3. При записи алгоритма ПО в виде текста на языке программирования возможно внесение типовых ошибок трансляции (синтаксических и семантических).
4. Некоторые результаты в программе зависят не от исходных данных, а от внутренних состояний программы.

Каждая из этих причин является аргументом для проведения тестирования по принципу «белого ящика». Тесты «черного ящика» не смогут реагировать на ошибки таких типов.

15. Основные принципы способа тестирования базового пути.

Тестирование базового пути — это способ, который основан на принципе «белого ящика». Автор этого способа — Том МакКейб (1976) [49].

Способ тестирования базового пути дает возможность:

- получить оценку комплексной сложности программы;
- использовать эту оценку для определения необходимого количества тестовых вариантов.

Тестовые варианты разрабатываются для проверки базового множества путей (маршрутов) в программе. Они гарантируют однократное выполнение каждого оператора программы при тестировании.

Шаги способа тестирования базового пути:

На основе текста программы формируется потоковый граф.

Определяется цикломатическая сложность потокового графа.

Определяется и выписывается базовое множество независимых линейных путей.

Подготавливаются тестовые варианты, инициирующие выполнение каждого пути.

Свойства потокового графа:

Граф строится отображением управляющей структуры программы. В ходе отображения закрывающие конструкции условных операторов и операторов циклов могут рассматриваться как отдельные (фиктивные) операторы.

Узлы (вершины) потокового графа соответствуют линейным участкам программы, включают один или несколько операторов программы.

Дуги потокового графа отображают поток управления в программе (передачи управления между операторами).

Различают операторные и предикатные узлы. Из операторного узла выходит одна дуга, а из предикатного — две дуги.

Предикатные узлы соответствуют простым условиям в программе. Составное условие программы отображается в несколько предикатных узлов.

Замкнутые области, образованные дугами и узлами, называют регионами. Окружающая граф среда рассматривается как дополнительный регион.

Цикломатическая сложность – метрика ПО, которая обеспечивает количественную оценку логической сложности программы.

В способе тестирования базового пути цикломатическая сложность определяет:

- количество независимых путей в базовом множестве программы;
- верхнюю оценку количества тестов, которое гарантирует однократное выполнение всех операторов.

Независимым называется любой путь, который вводит новый оператор обработки или новое условие. В терминах потокового графа независимый путь должен содержать дугу, не входящую в ранее определенные пути.

Все независимые пути графа образуют базовое множество.

Свойства базового множества:

1. тесты, обеспечивающие его проверку, гарантируют:
   - однократное выполнение каждого оператора;
   - выполнение каждого условия по True-ветви и по False-ветви;
2. мощность базового множества равна цикломатической сложности потокового графа.

Способы определения цикломатической сложности:

1. цикломатическая сложность равна количеству регионов потокового графа;
2. определяется по формуле

$$V(G)=E-N+2,$$

где E — количество дуг, N — количество узлов потокового графа;

3. определяется по формуле

$$V(G) =p+ 1,$$

где p — количество предикатных узлов в потоковом графе G.

16. Разбиение на классы эквивалентности. Способы тестирования классов эквивалентности. Построение дерева разбиений.

Разбиение по эквивалентности — самый популярный способ тестирования «черного ящика» [3], [14].

В этом способе входная область данных программы *делится* на классы эквивалентности. Для каждого класса эквивалентности разрабатывается один тестовый вариант.

Класс эквивалентности — набор данных с общими свойствами. Обрабатывая разные элементы класса, программа должна вести себя одинаково. Иначе говоря, при обработке любого набора из класса эквивалентности в программе задействуется один и тот же набор операторов (и связей между ними).

На рис. 7.2 каждый класс эквивалентности показан эллипсом. Здесь выделены входные классы эквивалентности допустимых и недопустимых исходных данных, а также классы результатов.

Классы эквивалентности могут быть определены по спецификации на программу.

Рис. 7.2. Разбиение по эквивалентности

Чтобы удостовериться в правильности поведения программы при различных входных данных, в идеале следует протестировать все возможные значения для каждого элемента этих данных. (а также все возможные сочетания входных параметров)

Чтобы уменьшить количество тестируемых значений, производится

а) разбиение множества всех значений входной переменной на подмножества (классы эквивалентности), а затем

б) тестирование одного любого значения из каждого класса.

Все значения из каждого подмножества должны быть эквивалентны для наших тестов. То есть, если тест проходит успешно для одного значения из класса эквивалентности, он должен проходить успешно для всех остальных. И наоборот, если тест не проходит для одного значения, он не должен проходить для всех остальных.

Таким образом, метод классов эквивалентности можно разделить на три этапа:

1. Тестирование разрешенных значений
2. Тестирование граничных значений
3. Тестирование запрещенных значений

Часто в литературе второй и третий этапы называют отдельными методами, но сути это не меняет.

17. Тестирование ПО способом анализа граничных значений.

Как правило, большая часть ошибок происходит на границах области ввода, а не в центре. Анализ граничных значений заключается в получении тестовых вариантов, которые анализируют граничные значения [3], [14], [69]. Данный способ тестирования дополняет способ разбиения по эквивалентности.

Основные отличия анализа граничных значений от разбиения по эквивалентности:

1) тестовые варианты создаются для проверки только ребер классов эквивалентности;

2) при создании тестовых вариантов учитывают не только условия ввода, но и область вывода.

Сформулируем **правила анализа граничных значений.**

1. Если условие ввода задает диапазон *n...m*, то тестовые варианты должны быть построены:

- для значений *n* и *m;*
- для значений чуть левее *n* и чуть правее *m* на числовой оси.

Например, если задан входной диапазон -1,0...+1,0, то создаются тесты для значений -1,0, +1,0, - 1,001, +1,001.

2. Если условие ввода задает дискретное множество значений, то создаются тестовые варианты:

- для проверки минимального и максимального из значений;
- для значений чуть меньше минимума и чуть больше максимума.

Так, если входной файл может содержать от 1 до 255 записей, то создаются тесты для О, 1, 255, 256 записей.

3. Правила 1 и 2 применяются к условиям области вывода.

Рассмотрим пример, когда в программе требуется выводить таблицу значений. Количество строк и столбцов в таблице меняется. Задается тестовый вариант для минимального вывода (по объему таблицы), а также тестовый вариант для максимального вывода (по объему таблицы).

4. Если внутренние структуры данных программы имеют предписанные границы, то разрабатываются тестовые варианты, проверяющие эти структуры на их границах.

5. Если входные или выходные данные программы являются упорядоченными множествами (например, последовательным файлом, линейным списком, таблицей), то надо тестировать обработку первого и последнего элементов этих множеств.

18. Тестирование ПО способом потоков данных.

В предыдущих способах тесты строились на основе анализа управляющей структуры программы. В данном способе анализу подвергается информационная структура программы.

Работу любой программы можно рассматривать как обработку потока данных, передаваемых от входа в программу к ее выходу.

Рассмотрим пример.

Пусть потоковый граф программы имеет вид, представленный на рис. 6.8. В нем сплошные дуги — это связи по управлению между операторами в программе. Пунктирные дуги отмечают информационные связи (связи по потокам данных). Обозначенные здесь информационные связи соответствуют следующим допущениям:

- в вершине 1 определяются значения переменных *a, b;*
- значение переменной *a* используется в вершине 4;
- значение переменной *b* используется в вершинах 3, 6;
- в вершине 4 определяется значение переменной c, которая используется в вершине 6.

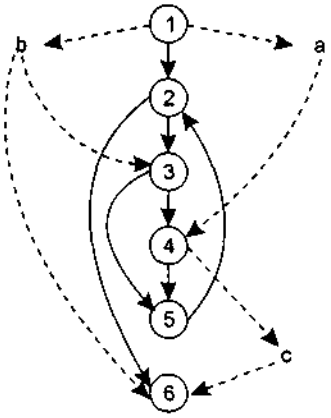


Рис. 6.8. Граф программы с управляющими и информационными связями

В общем случае для каждой вершины графа можно записать:

- множество определений данных

DEF(i) = { x | i -я вершина содержит определение x};

- множество использований данных:

USE (i) = { x | i -я вершина использует x}.

Под *определением данных* понимают действия, изменяющие элемент данных. Признак определения — имя элемента стоит в левой части оператора присваивания:

$$x{:=}f(\ldots).$$

Использование данных — это применение элемента в выражении, где происходит обращение к элементу данных, но не изменение элемента. Признак использования — имя элемента стоит в правой части оператора присваивания:

$$\#{:=}f(x).$$

Здесь место подстановки другого имени отмечено прямоугольником (прямоугольник играет роль метки-заполнителя).

Назовём *DU-цепочкой (цепочкой определения-использования)* конструкцию *[x, i,j]*, где *i,j* — имена вершин; x определена в *i*-й вершине *(*$x \in$ DEF(*i*)) и используется в *j* -й вершине *(*$x \in$ USE(*j*)).

В нашем примере существуют следующие DU-цепочки:

[a,1,4],[b, 1,3], [b, 1,6], [c, 4, 6].

Способ *DU-тестирования* требует охвата всех DU-цепочек программы. Таким образом, разработка тестов здесь проводится на основе анализа жизни всех данных программы.

Очевидно, что для подготовки тестов требуется выделение маршрутов — путей выполнения программы на управляющем графе. Критерий для выбора пути — покрытие максимального количества DU-цепочек.

Шаги способа DU-тестирования:

1) построение управляющего графа (УГ) программы;
2) построение информационного графа (ИГ);
3) формирование полного набора DU-цепочек;
4) формирование полного набора отрезков путей в управляющем графе (отображением набора DU-цепочек информационного графа, рис. 6.9);

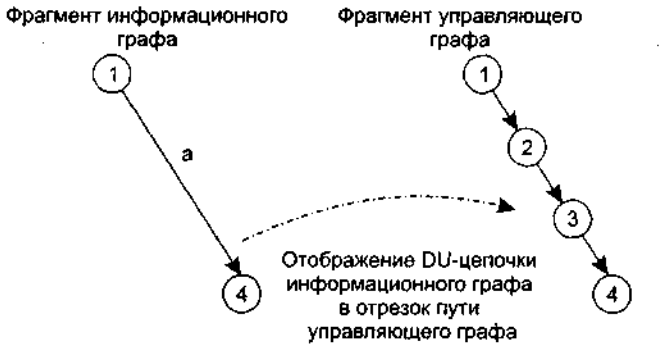


Рис. 6.9. Отображение DU-цепочки *в* отрезок пути

5) построение маршрутов — полных путей на управляющем графе, покрывающих набор отрезков путей управляющего графа;
6) подготовка тестовых вариантов.

Достоинства DU-тестирования:

- простота необходимого анализа операционно-управляющей структуры программы;
- простота автоматизации.

Недостаток DU-тестирования: трудности в выборе минимального количества максимально эффективных тестов.

Область использования DU-тестирования: программы с вложенными условными операторами и операторами цикла.

19. Тестирование ПО способом анализа причинно-следственных связей.

Диаграммы причинно-следственных связей — способ проектирования тестовых вариантов, который обеспечивает формальную запись логических условий и соответствующих действий [3], [64]. Используется автоматный подход к решению задачи.

Шаги способа:

1) для каждого модуля перечисляются причины (условия ввода или классы эквивалентности условий ввода) и следствия (действия или условия вывода). Каждой причине и следствию присваивается свой идентификатор;
2) разрабатывается граф причинно-следственных связей;
3) граф преобразуется в таблицу решений;
4) столбцы таблицы решений преобразуются в тестовые варианты.

Изобразим базовые символы для записи графов причин и следствий (cause-effect graphs).

Сделаем предварительные *замечания:*

1) причины будем обозначать символами c_i, а следствия — символами e_i;
2) каждый узел графа может находиться в состоянии 0 или 1 (0 — состояние отсутствует, 1 — состояние присутствует).

Функция тождество (рис. 7.4) устанавливает, что если значение c_1 есть 1, то и значение e_1 есть 1; в противном случае значение e_1 есть 0.

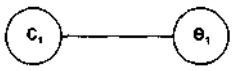


Рис. 7.4. Функция *тождество*

Функция не (рис. 7.5) устанавливает, что если значение c_1 есть 1, то значение e_1 есть 0; в противном случае значение e_1 есть 1.

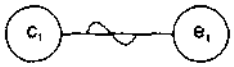


Рис. 7.5. Функция *не*

Функция или (рис. 7.6) устанавливает, что если c_1 или c_2 есть 1, то e_1 есть 1, в противном случае e_1 есть 0.

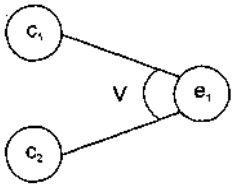


Рис. 7.6. Функция *или*

Функция и (рис. 7.7) устанавливает, что если и c_1 и c_2 есть 1, то e_1 есть 1, в противном случае e_1 есть 0.

Часто определенные комбинации причин невозможны из-за синтаксических или внешних ограничений. Используются перечисленные ниже обозначения ограничений.

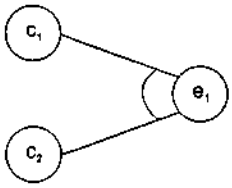


Рис. 7.7. Функция *и*

Ограничение E (исключает, Exclusive, рис. 7.8) устанавливает, что E должно быть истинным, если хотя бы одна из причин — *a* или *b* — принимает значение 1 *(a* и *b* не могут принимать значение 1 одновременно).

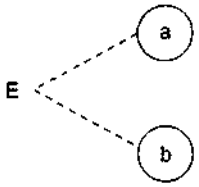


Рис. 7.8. Ограничение E (исключает, Exclusive)

Ограничение I (включает, Inclusive, рис. 7.9) устанавливает, что по крайней мере одна из величин, *a, b,* или c, всегда должна быть равной 1 (*a, b* и *c* не могут принимать значение 0 одновременно).

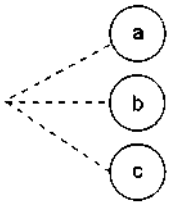


Рис. 7.9. Ограничение I (включает, Inclusive)

Ограничение O (одно и только одно, Only one, рис. 7.10) устанавливает, что одна и только одна из величин *a* или *b* должна быть равна 1.

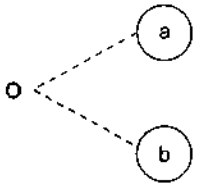


Рис. 7.10. Ограничение O (одно и только одно, Only one)

Ограничение R (требует, Requires, рис. 7.11) устанавливает, что если *a* принимает значение 1, то и *b* должна принимать значение 1 (нельзя, чтобы *a* было равно 1, а *b* - 0).

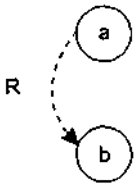


Рис. 7.11. Ограничение R (требует, Requires)

Часто возникает необходимость в ограничениях для следствий.

Ограничение M (скрывает, Masks, рис. 7.12) устанавливает, что если следствие *a* имеет значение 1, то следствие *b* должно принять значение 0.

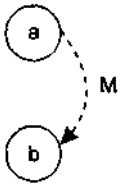


Рис. 7.12. Ограничение M (скрывает, Masks)

20. Организация процесса тестирования. Методика тестирования ПО, разработанного на основе процедурного подхода. Тестирование элементов. Алгоритм работы тестового драйвера.

Процесс тестирования объединяет различные способы тестирования в спланированную последовательность шагов, которые приводят к успешному построению программной системы (ПС) [3], [13], [64], [69]. Методика тестирования ПС может быть представлена в виде разворачивающейся спирали (рис. 8.1).

В начале осуществляется *тестирование элементов (модулей),* проверяющее результаты этапа *кодирования* ПС. На втором шаге выполняется *тестирование интеграции,* ориентированное на выявление ошибок этапа *проектирования* ПС. На третьем обороте спирали производится *тестирование правильности,* проверяющее корректность этапа *анализа требований* к ПС. На заключительном витке спирали проводится *системное тестирование,* выявляющее дефекты этапа *системного анализа* ПС.

Охарактеризуем каждый шаг процесса тестирования.

1. *Тестирование элементов.* Цель — индивидуальная проверка каждого модуля. Используются способы тестирования «белого ящика».

Рис. 8.1. Спираль процесса тестирования ПС

2. *Тестирование интеграции.* Цель — тестирование сборки модулей в программную систему. В основном применяют способы тестирования «черного ящика».
3. *Тестирование правильности.* Цель — проверить реализацию в программной системе всех функциональных и поведенческих требований, а также требования эффективности. Используются исключительно способы тестирования «черного ящика».
4. *Системное тестирование.* Цель — проверка правильности объединения и взаимодействия всех элементов компьютерной системы, реализации всех системных функций.

Организация процесса тестирования в виде эволюционной разворачивающейся спирали обеспечивает максимальную эффективность поиска ошибок. Однако возникает вопрос — когда заканчивать тестирование?

Ответ практика обычно основан на статистическом критерии: «Можно с 95%-ной уверенностью сказать, что провели достаточное тестирование, если вероятность безотказной работы ЦП с программным изделием в течение 1000 часов составляет по меньшей мере 0,995».

Научный подход при ответе на этот вопрос состоит в применении математической модели отказов. Например, для логарифмической модели Пуассона формула расчета текущей интенсивности отказов имеет вид:

$$\lambda(t) = \frac{\lambda_0}{\lambda_0 \times p \times t + 1},$$

где $\lambda(t)$ — текущая интенсивность программных отказов (количество отказов в единицу времени); λ_0 — начальная интенсивность отказов (в начале тестирования); p — экспоненциальное уменьшение интенсивности отказов за счет обнаруживаемых и устраняемых ошибок; t —время тестирования.

С помощью уравнения (8.1) можно предсказать снижение ошибок в ходе тестирования, а также время, требующееся для достижения допустимо низкой интенсивности отказов.

Тестирование элементов

Объектом тестирования элементов является наименьшая единица проектирования ПС — модуль. Для обнаружения ошибок в рамках модуля тестируются его важнейшие управляющие пути. Относительная сложность тестов и ошибок определяется как результат ограничений области тестирования элементов. Принцип тестирования — «белый ящик», шаг может выполняться для набора модулей параллельно.

Тестированию подвергаются:

- интерфейс модуля;
- внутренние структуры данных;
- независимые пути;
- пути обработки ошибок;
- граничные условия.

Интерфейс модуля тестируется для проверки правильности ввода-вывода тестовой информации. Если нет уверенности в правильном вводе-выводе данных, нет смысла проводить другие тесты.

Исследование внутренних структур данных гарантирует целостность сохраняемых данных.

Тестирование независимых путей гарантирует однократное выполнение всех операторов модуля. При тестировании путей выполнения обнаруживаются следующие категории ошибок: ошибочные вычисления, некорректные сравнения, неправильный поток управления [3].

Наиболее общими ошибками вычислений являются:

1) неправильный или непонятый приоритет арифметических операций;
2) смешанная форма операций;
3) некорректная инициализация;
4) несогласованность в представлении точности;
5) некорректное символическое представление выражений.

Источниками ошибок сравнения и неправильных потоков управления являются:

1) сравнение различных типов данных;
2) некорректные логические операции и приоритетность;
3) ожидание эквивалентности в условиях, когда ошибки точности делают эквивалентность невозможной;
4) некорректное сравнение переменных;
5) неправильное прекращение цикла;
6) отказ в выходе при отклонении итерации;
7) неправильное изменение переменных цикла.

Обычно при проектировании модуля предвидят некоторые ошибочные условия. Для защиты от ошибочных условий в модуль вводят пути обработки ошибок. Такие пути тоже должны тестироваться. Тестирование путей обработки ошибок можно ориентировать на следующие ситуации:

1) донесение об ошибке невразумительно;
2) текст донесения не соответствует, обнаруженной ошибке;

3) вмешательство системных средств регистрации аварии произошло до обработки ошибки в модуле;

4) обработка исключительного условия некорректна;

5) описание ошибки не позволяет определить ее причину.

И, наконец, перейдем к граничному тестированию. Модули часто отказывают на «границах». Это означает, что ошибки часто происходят:

1) при обработке *n*-го элемента *n*-элементного массива;

2) при выполнении *m*-й итерации цикла с *m* проходами;

3) при появлении минимального (максимального) значения.

Тестовые варианты, ориентированные на данные ситуации, имеют высокую вероятность обнаружения ошибок.

Тестирование элементов обычно рассматривается как дополнение к этапу кодирования. Оно начинается после разработки текста модуля. Так как модуль не является автономной системой, то для реализации тестирования требуются дополнительные средства, представленные на рис. 8.2.

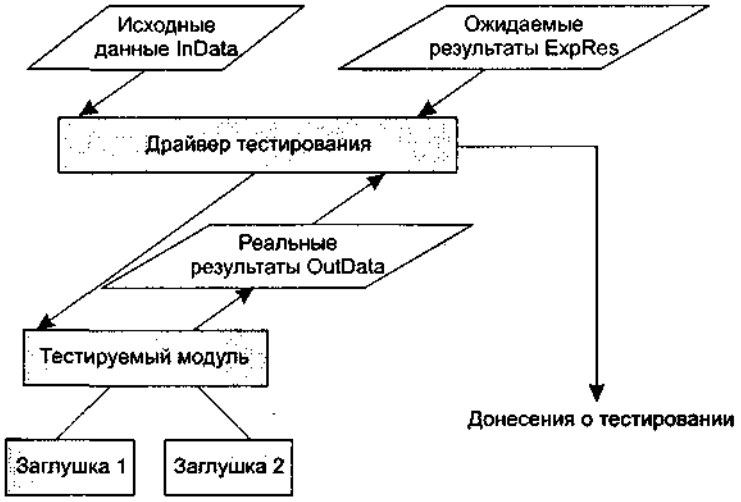


Рис. 8.2. Программная среда для тестирования модуля

Дополнительными средствами являются драйвер тестирования и заглушки. Драйвер — управляющая программа, которая принимает исходные данные (InData) и ожидаемые результаты (ExpRes) тестовых вариантов, запускает в работу тестируемый модуль, получает из модуля реальные результаты (OutData) и формирует донесения о тестировании. Алгоритм работы тестового драйвера приведен на рис. 8.3.

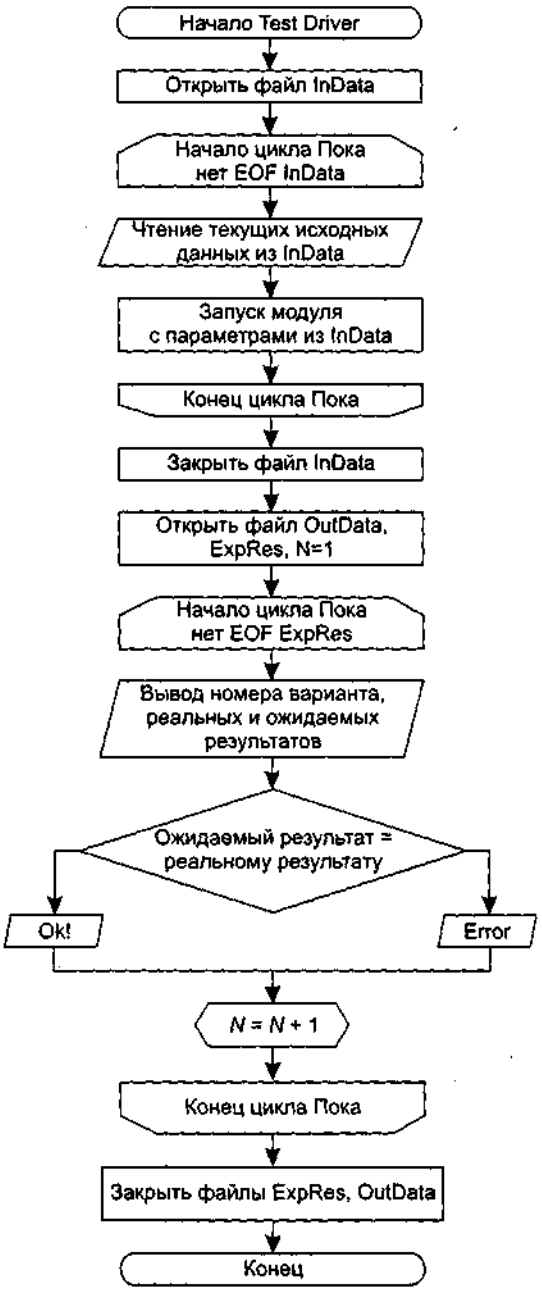


Рис. 8.3. Алгоритм работы драйвера тестирования

Заглушки замещают модули, которые вызываются тестируемым модулем. Заглушка, или «фиктивная подпрограмма», реализует интерфейс подчиненного модуля, может выполнять минимальную обработку данных, имитирует прием и возврат данных.

Создание драйвера и заглушек подразумевает дополнительные затраты, так как они не поставляются с конечным программным продуктом.

Если эти средства просты, то дополнительные затраты невелики. Увы, многие модули не могут быть адекватно протестированы с помощью простых дополнительных средств. В этих случаях полное тестирование может быть отложено до шага тестирования интеграции (где драйверы или заглушки также используются).

Тестирование элемента просто осуществить, если модуль имеет высокую связность. При реализации модулем только одной функции количество тестовых вариантов уменьшается, а ошибки легко предсказываются и обнаруживаются.

21. Нисходящее тестирование интеграции.

В данном подходе модули объединяются движением сверху вниз по управляющей иерархии, начиная от главного управляющего модуля. Подчиненные модули добавляются в структуру или в результате поиска в глубину, или в результате поиска в ширину.

Рассмотрим пример (рис. 8.4). Интеграция поиском в глубину будет подключать все модули, находящиеся на главном управляющем пути структуры (по вертикали). Выбор главного управляющего пути отчасти произволен и зависит от характеристик, определяемых приложением. Например, при выборе левого пути прежде всего будут подключены модули Ml, M2, M5. Следующим подключается модуль M8 или Мб (если это необходимо для правильного функционирования M2). Затем строится центральный или правый управляющий путь.

При интеграции поиском в ширину структура последовательно проходится по уровням-горизонталям. На каждом уровне подключаются модули, непосредственно подчиненные управляющему модулю — начальнику. В этом случае прежде всего подключаются модули M2, M3, M4. На следующем уровне — модули M5, Мб и т. д.

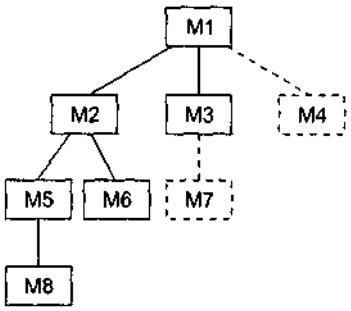


Рис. 8.4. Нисходящая интеграция системы

Опишем возможные шаги процесса нисходящей интеграции.

1. Главный управляющий модуль (находится на вершине иерархии) используется как тестовый драйвер. Все непосредственно подчиненные ему модули временно замещаются заглушками.
2. Одна из заглушек заменяется реальным модулем. Модуль выбирается поиском в ширину или в глубину.
3. После подключения каждого модуля (и установки на нем заглушек) проводится набор тестов, проверяющих полученную структуру.
4. Если в модуле-драйвере уже нет заглушек, производится смена модуля-драйвера (поиском в ширину или в глубину).
5. Выполняется возврат на шаг 2 (до тех пор, пока не будет построена целая структура).

Достоинство нисходящей интеграции: ошибки в главной, управляющей части системы выявляются в первую очередь.

Недостаток: трудности в ситуациях, когда для полного тестирования на верхних уровнях нужны результаты обработки с нижних уровней иерархии.

Существуют 3 возможности борьбы с этим недостатком:

1) откладывать некоторые тесты до замещения заглушек модулями;

2) разрабатывать заглушки, частично выполняющие функции модулей;

3) подключать модули движением снизу вверх.

Первая возможность вызывает сложности в оценке результатов тестирования.

Для реализации второй возможности выбирается одна из следующих категорий заглушек:

- заглушка А — отображает трассируемое сообщение;
- заглушка В — отображает проходящий параметр;
- заглушка С — возвращает величину из таблицы;

- заглушка D — выполняет табличный поиск по ключу (входному параметру) и возвращает связанный с ним выходной параметр.

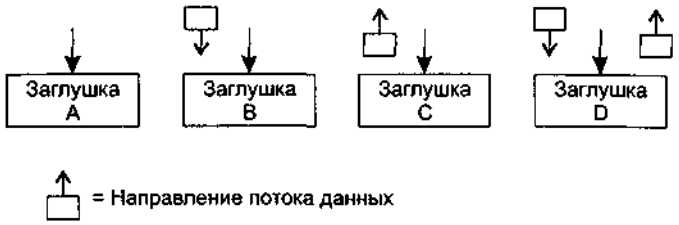


Рис. 8.5. Категории заглушек

Категории заглушек представлены на рис. 8.5.

Очевидно, что заглушка A наиболее проста, а заглушка D наиболее сложна в реализации.

Этот подход работоспособен, но может привести к существенным затратам, так как заглушки становятся все более сложными.

22. Восходящее тестирование интеграции.

При восходящем тестировании интеграции сборка и тестирование системы начинаются с модулей-атомов, располагаемых на нижних уровнях иерархии. Модули подключаются движением снизу вверх. Подчиненные модули всегда доступны, и нет необходимости в заглушках.

Рассмотрим шаги методики восходящей интеграции.

1. Модули нижнего уровня объединяются в кластеры (группы, блоки), выполняющие определенную программную подфункцию.
2. Для координации вводов-выводов тестового варианта пишется драйвер, управляющий тестированием кластеров.
3. Тестируется кластер.
4. Драйверы удаляются, а кластеры объединяются в структуру движением вверх. Пример восходящей интеграции системы приведен на рис. 8.6.

Модули объединяются в кластеры 1,2,3. Каждый кластер тестируется драйвером. Модули в кластерах 1 и 2 подчинены модулю Ma, поэтому драйверы D1 и D2 удаляются и кластеры подключают прямо к Ma. Аналогично драйвер D3 удаляется перед подключением кластера 3 к модулю Mb. В последнюю очередь к модулю Mc подключаются модули Ma и Mb.

Рассмотрим различные типы драйверов:

- драйвер A — вызывает подчиненный модуль;
- драйвер B — посылает элемент данных (параметр) из внутренней таблицы;
- драйвер C —отображает параметр из подчиненного модуля;
- драйвер D — является комбинацией драйверов B и C.

Очевидно, что драйвер A наиболее прост, а драйвер D наиболее сложен в реализации. Различные типы драйверов представлены на рис. 8.7.

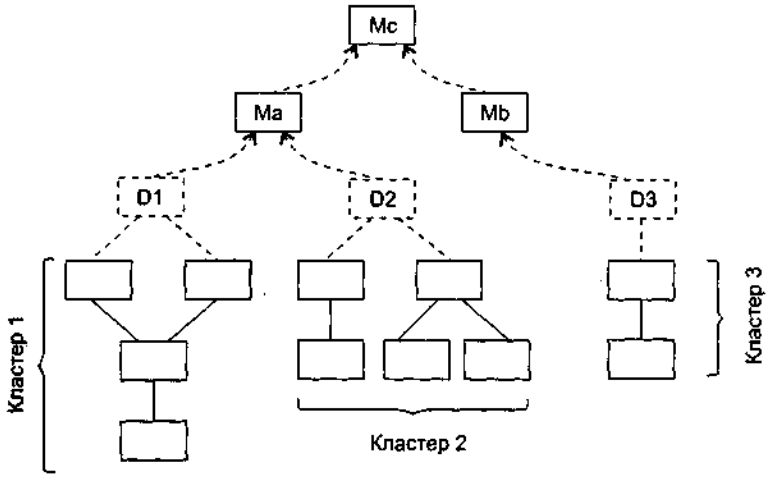


Рис. 8.6. Восходящая интеграция системы

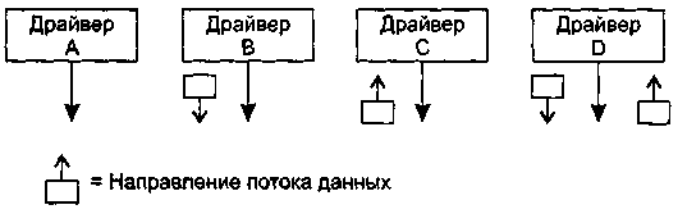


Рис. 8.7. Различные типы драйверов

По мере продвижения интеграции вверх необходимость в выделении драйверов уменьшается. Как правило, в двухуровневой структуре драйверы не нужны.

23. Тестирование правильности ПО. Основные компоненты конфигурации программной системы. Альфа-тестирование и бета-тестирование.

После окончания тестирования интеграции программная система собрана в единый корпус, интерфейсные ошибки обнаружены и откорректированы. Теперь начинается последний шаг программного тестирования — *тестирование правильности.* Цель — подтвердить, что функции, описанные в спецификации требований к ПС, соответствуют ожиданиям заказчика [64], [69].

Подтверждение правильности ПС выполняется с помощью тестов «черного ящика», демонстрирующих соответствие требованиям. При обнаружении отклонений от спецификации требований создается список недостатков. Как правило, отклонения и ошибки, выявленные при подтверждении правильности, требуют изменения сроков разработки продукта.

Важным элементом подтверждения правильности является проверка конфигурации ПС. Конфигурацией ПС называют совокупность всех элементов информации, вырабатываемых в процессе конструирования ПС. Минимальная конфигурация ПС включает следующие базовые элементы:

1) системную спецификация;
2) план программного проекта;
3) спецификацию требований к ПС; работающий или бумажный макет;
4) предварительное руководство пользователя;
5) спецификация проектирования;
6) листинги исходных текстов программ;
7) план и методику тестирования; тестовые варианты и полученные результаты;
8) руководства по работе и инсталляции;
9) exe-код выполняемой программы;
10) описание базы данных;
11) руководство пользователя по настройке;
12) документы сопровождения; отчеты о проблемах ПС; запросы сопровождения; отчеты о конструкторских изменениях;
13) стандарты и методики конструирования ПС.

Проверка конфигурации гарантирует, что все элементы конфигурации ПС правильно разработаны, учтены и достаточно детализированы для поддержки этапа сопровождения в жизненном цикле ПС.

Разработчик не может предугадать, как заказчик будет реально использовать ПС. Для обнаружения ошибок, которые способен найти только конечный пользователь, используют процесс, включающий альфа- и бета-тестирование.

Альфа-тестирование проводится заказчиком в организации разработчика. Разработчик фиксирует все выявленные заказчиком ошибки и проблемы использования.

Бета-тестирование проводится конечным пользователем в организации заказчика. Разработчик в этом процессе участия не принимает. Фактически, бета-тестирование — это реальное применение ПС в среде, которая не управляется разработчиком. Заказчик сам записывает все обнаруженные проблемы и сообщает о них разработчику. Бета-тестирование проводится в течение фиксированного срока (около года). По результатам выявленных проблем разработчик изменяет ПС и тем самым подготавливает продукт полностью на базе заказчика.

24. Основные принципы объектно-ориентированного тестирования.

Существует мнение, что объектно-ориентированное тестирование мало чем отличается от процедурно-ориентированного тестирования. Конечно, многие понятия, подходы и способы тестирования у них общие, но в целом это мнение ошибочно. Напротив, особенности объектно-ориентированных систем должны вносить и вносят существенные изменения как в последовательность этапов, так и в содержание этапов тестирования. Сгруппируем эти изменения по трем направлениям:

- расширение области применения тестирования;
- изменение методики тестирования;
- учет особенностей объектно-ориентированного ПО при проектировании тестовых вариантов.

Обсудим каждое из выделенных направлений отдельно.

Расширение области применения объектно-ориентированного тестирования

Разработка объектно-ориентированного ПО начинается с создания визуальных моделей, отражающих статические и динамические характеристики будущей системы. Вначале эти модели фиксируют исходные требования заказчика, затем формализуют реализацию этих требований путем выделения объектов, которые взаимодействуют друг с другом посредством передачи сообщений. Исследование моделей взаимодействия приводит к построению моделей классов и их отношений, составляющих основу логического представления системы. При переходе к физическому представлению строятся модели компоновки и размещения системы.

На конструирование моделей приходится большая часть затрат объектно-ориентированного процесса разработки. Если к этому добавить, что цена устранения ошибки стремительно растет с каждой итерацией разработки, то совершенно логично требование тестировать объектно-ориентированные модели анализа и проектирования.
Критерии тестирования моделей: правильность, полнота, согласованность.

Изменение методики при объектно-ориентированном тестировании

В классической методике тестирования действия начинаются с тестирования элементов, а заканчиваются тестированием системы. Вначале тестируют модули, затем тестируют интеграцию модулей, проверяют правильность реализации требований, после чего тестируют взаимодействие всех блоков компьютерной системы.

Проектирование объектно-ориентированных тестовых вариантов

Традиционные тестовые варианты ориентированы на проверку последовательности: ввод исходных данных — обработка — вывод результатов — или на проверку внутренней управляющей (информационной) структуры отдельных модулей. Объектно-ориентированные тестовые варианты проверяют состояния классов. Получение информации о состоянии затрудняют такие объектно-ориентированные характеристики, как инкапсуляция, полиморфизм и наследование.

Инкапсуляция

Информацию о состоянии класса можно получить только с помощью встроенных в него операций, которые возвращают значения свойств класса.

Полиморфизм

При вызове полиморфной операции трудно определить, какая реализация будет проверяться. Пусть нужно проверить вызов функции

y=functionA(x).

В стандартном ПО достаточно рассмотреть одну реализацию поведения, которая обеспечивает вычисление функции. В объектно-ориентированном ПО придется рассматривать поведение реализации Базовый_класс :: functionA(x), Производный_класс :: functionA(x), Наследиик_Производного_класса :: functionA(x). Здесь двойным двоеточием от имени операции отделяется имя класса, в котором размещена операция (это

обозначение UML). Таким образом, в объектно-ориентированном контексте каждый раз при вызове functionA(x) следует рассматривать набор различных поведений. Конечно, подход к тестированию базовых и производных классов в основном одинаков. Разница состоит только в учете используемых системных ресурсов.

Наследование

Наследование также может усложнить проектирование тестовых вариантов. Пусть Родительский_класс содержит операции унаследована() и переопределена(). Дочерний_класс переопределяет операцию переопределена() по-своему. Очевидно, что реализация Дочерний_класс::переопределена() должна повторно тестироваться, ведь ее содержание изменилось. Но надо ли повторно тестировать операцию Дочерний_класс::унаследована()?

Важно отметить, что для операции Дочерний_класс::унаследована() может проводиться только подмножество тестов. Если часть операции унаследована() не зависит от операции переопределен(), то есть нет ее вызова или вызова любого кода, который косвенно ее вызывает, то ее не надо повторно тестировать в дочернем классе.

Родительский_класс::переопределена() и Дочерний_класс::переопределена() —это две разные операции с различными спецификациями и реализациями. Каждая из них проверяется самостоятельным набором тестов. Эти тесты нацелены на вероятные ошибки: ошибки интеграции, ошибки условий, граничные ошибки и т. д. Однако сами операции, как правило, похожи. Наборы их тестов будут перекрываться. Чем лучше качество объектно-ориентированного проектирования, тем больше перекрытие. Таким образом, новые тесты надо формировать только для тех требований к операции Дочерний_класс::переопределена(), которые не покрываются тестами для операции Родительский_класс::переопределена().

25. Основные принципы отладки ПО.

Отладка — это локализация и устранение ошибок. Отладка является следствием успешного тестирования. Это значит, что если тестовый вариант обнаруживает ошибку, то процесс отладки уничтожает ее.

Итак, процессу отладки предшествует выполнение тестового варианта. Его результаты оцениваются, регистрируется несоответствие между ожидаемым и реальным результатами. Несоответствие является симптомом скрытой причины. Процесс отладки пытается сопоставить симптом с причиной, вследствие чего приводит к исправлению ошибки. Возможны два исхода процесса отладки:

1) причина найдена, исправлена, уничтожена;
2) причина не найдена.

Во втором случае отладчик может предполагать причину. Для проверки этой причины он просит разработать дополнительный тестовый вариант, который поможет проверить предположение. Таким образом, запускается итерационный процесс коррекции ошибки.

Возможные разные способы проявления ошибок:

1) программа завершается нормально, но выдает неверные результаты;
2) программа зависает;
3) программа завершается по прерыванию;
4) программа завершается, выдает ожидаемые результаты, но хранимые данные испорчены (это самый неприятный вариант).

Характер проявления ошибок также может меняться. Симптом ошибки может быть:

- постоянным;
- мерцающим;
- пороговым (проявляется при превышении некоторого порога в обработке — 200 самолетов на экране отслеживаются, а 201-й — нет);
- отложенным (проявляется только после исправления маскирующих ошибок).

В ходе отладки мы встречаем ошибки в широком диапазоне: от мелких неприятностей до катастроф. Следствием увеличения ошибок является усиление давления на отладчика — «найди ошибки быстрее!!!». Часто из-за этого давления разработчик устраняет одну ошибку и вносит две новые ошибки.

Различают две группы методов отладки:

- аналитические;
- экспериментальные.

Аналитические методы базируются на анализе выходных данных для тестовых прогонов. Экспериментальные методы базируются на использовании вспомогательных средств отладки (отладочные печати, трассировки), позволяющих уточнить характер поведения программы при тех или иных исходных данных.

Общая стратегия отладки — обратное прохождение от замеченного симптома ошибки к исходной аномалии (месту в программе, где ошибка совершена).

Цель отладки — найти оператор программы, при исполнении которого правильные аргументы приводят к неправильным результатам. Если место проявления симптома ошибки не является искомой аномалией, то один из аргументов оператора должен быть неверным. Поэтому надо перейти к исследованию предыдущего оператора, выработавшего этот неверный аргумент. В итоге пошаговое обратное прослеживание приводит к искомому ошибочному месту.

В разных методах прослеживание организуется по-разному. В аналитических методах — на основе логических заключений о поведении программы. Цель — шаг за шагом уменьшать область программы, подозреваемую в наличии ошибки. Здесь определяется корреляция между значениями выходных данных и особенностями поведения.

Основное *преимущество аналитических методов отладки* состоит в том, что исходная программа остается без изменений.

В экспериментальных методах для прослеживания выполняется:

1. Выдача значений переменных в указанных точках.
2. Трассировка переменных (выдача их значений при каждом изменении).
3. Трассировка потоков управления (имен вызываемых процедур, меток, на которые передается управление, номеров операторов перехода).

Преимущество экспериментальных методов отладки состоит в том, что основная рутинная работа по анализу процесса вычислений перекладывается на компьютер. Многие трансляторы имеют встроенные средства отладки для получения информации о ходе выполнения программы.

Недостаток экспериментальных методов отладки — в программу вносятся изменения, при исключении которых могут появиться ошибки. Впрочем, некоторые системы программирования создают специальный отладочный экземпляр программы, а в основной экземпляр не вмешиваются.