КАЗАХСКИЙ НАЦИОНАЛЬНЫЙ УНИВЕРСИТЕТ ИМ.АЛЬ_ФАРАБИ

УЧЕБНО-МЕТОДИЧЕСКИЙ КОМПЛЕКС ДИСЦИПЛИНЫ

Проектирование и разработка программного обеспечения

Специальность _6М060200 - Информатика_
(шифр, название)

Форма обучения ____дневная____
(дневная, заочная)

г. Алматы 2018г.

УМК дисциплины составлен ___Макашев Е.П. к.ф.м.н.
(Ф.И.О., должность, ученая степень и звание составителя(ей))

На основании __типового учебного плана_____
(на основании каких документов)
_____

Рассмотрен и рекомендован на заседании кафедры
_____
От «___» _____ 2018 г., протокол №____
Зав. кафедрой _____ Ф.И.О.
(роспись)

Рекомендовано методическим Советом (бюро) факультета
«__»_____ 2018 г., протокол №__

Председатель _____ Ф.И.О.
(роспись)

Лекция 1

ВВЕДЕНИЕ

Известно, что основной задачей первых трех десятилетий компьютерной эры являлось развитие аппаратных компьютерных средств. Это было обусловлено высокой стоимостью обработки и хранения данных. В 80-е годы успехи микроэлектроники привели к резкому увеличению производительности компьютера при значительном снижении стоимости.

Основной задачей 90-х годов и начала XXI века стало совершенствование качества компьютерных приложений, возможности которых целиком определяются программным обеспечением (ПО).

Современный персональный компьютер теперь имеет производительность большой ЭВМ 80-х годов. Сняты практически все аппаратные ограничения на решение задач. Оставшиеся ограничения приходятся на долю ПО.

Чрезвычайно актуальными стали следующие проблемы:

- аппаратная сложность опережает наше умение строить ПО, использующее потенциальные возможности аппаратуры;
- наше умение строить новые программы отстает от требований к новым программам;
- нашим возможностям эксплуатировать существующие программы угрожает низкое качество их разработки.

Ключом к решению этих проблем является грамотная организация процесса создания ПО, реализация технологических принципов промышленного конструирования программных систем (ПС).

Определение технологии конструирования программного обеспечения

Технология конструирования программного обеспечения (ТКПО) — система инженерных принципов для создания экономичного ПО, которое надежно и эффективно работает в реальных компьютерах [64], [69], [71].

Различают методы, средства и процедуры ТКПО.

Методы обеспечивают решение следующих задач:

- планирование и оценка проекта;
- анализ системных и программных требований;
- проектирование алгоритмов, структур данных и программных структур;
- кодирование;
- тестирование;
- сопровождение.

Средства (утилиты) ТКПО обеспечивают автоматизированную или автоматическую поддержку методов. В целях совместного применения утилиты могут объединяться в системы автоматизированного конструирования ПО. Такие системы принято называть CASE-системами. Аббревиатура CASE расшифровывается как Computer Aided Software Engineering (программная инженерия с компьютерной поддержкой).

Процедуры являются «клеем», который соединяет методы и утилиты так, что они обеспечивают непрерывную технологическую цепочку разработки. Процедуры определяют:

- порядок применения методов и утилит;
- формирование отчетов, форм по соответствующим требованиям;
- контроль, который помогает обеспечивать качество и координировать изменения;
- формирование «вех», по которым руководители оценивают прогресс.

Процесс конструирования программного обеспечения состоит из последовательности шагов, использующих методы, утилиты и процедуры. Эти последовательности шагов часто называют парадигмами ТКПО.

Применение парадигм ТКПО гарантирует систематический, упорядоченный подход к промышленной разработке, использованию и сопровождению ПО. Фактически, парадигмы вносят в процесс создания ПО организующее инженерное начало, необходимость которого трудно переоценить.

Рассмотрим наиболее популярные парадигмы ТКПО.

Классический жизненный цикл

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.

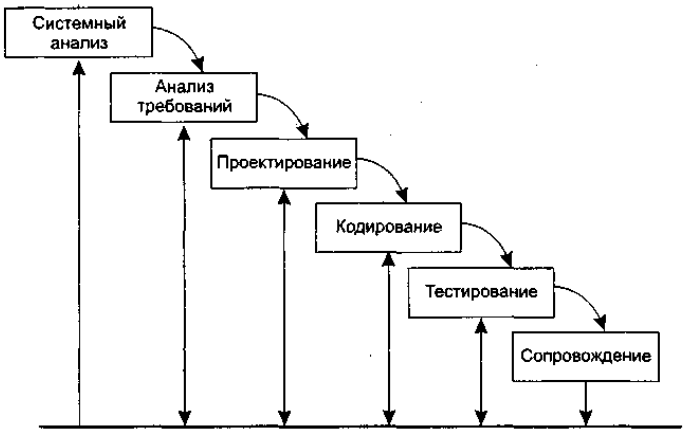

Рис. 1.1. Классический жизненный цикл разработки ПО

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.

Макетирование

Достаточно часто заказчик не может сформулировать подробные требования по вводу, обработке или выводу данных для будущего программного продукта. С другой стороны, разработчик может сомневаться в приспосабливаемое™ продукта под операционную систему, форме диалога с пользователем или в эффективности реализуемого алгоритма. В этих случаях целесообразно использовать макетирование.

Основная цель макетирования — снять неопределенности в требованиях заказчика.

Макетирование (прототипирование) — это процесс создания модели требуемого программного продукта.

Модель может принимать одну из трех форм:

1) бумажный макет или макет на основе ПК (изображает или рисует человеко-машинный диалог);
2) работающий макет (выполняет некоторую часть требуемых функций);
3) существующая программа (характеристики которой затем должны быть улучшены).

Как показано на рис. 1.2, макетирование основывается на многократном повторении итераций, в которых участвуют заказчик и разработчик.

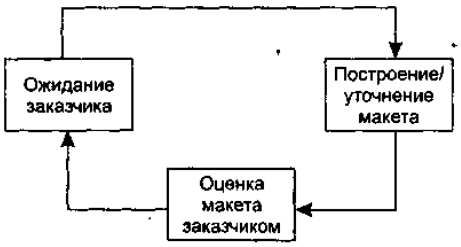

Рис. 1.2. Макетирование

Последовательность действий при макетировании представлена на рис. 1.3. Макетирование начинается со сбора и уточнения требований к создаваемому ПО Разработчик и заказчик встречаются и определяют все цели ПО, устанавливают, какие требования известны, а какие предстоит доопределить.

Затем выполняется быстрое проектирование. В нем внимание сосредоточивается на тех характеристиках ПО, которые должны быть видимы пользователю.

Быстрое проектирование приводит к построению макета.

Макет оценивается заказчиком и используется для уточнения требований к ПО.

Итерации повторяются до тех пор, пока макет не выявит все требования заказчика и, тем самым, не даст возможность разработчику понять, что должно быть сделано.

Достоинство макетирования: обеспечивает определение полных требований к ПО.

Недостатки макетирования:

- заказчик может принять макет за продукт;
- разработчик может принять макет за продукт.

Поясним суть недостатков. Когда заказчик видит работающую версию ПО, он перестает сознавать, что детали макета скреплены «жевательной резинкой и проволокой»; он забывает, что в погоне за работающим вариантом оставлены нерешенными вопросы качества и удобства сопровождения ПО. Когда заказчику говорят, что продукт должен быть перестроен, он начинает возмущаться и требовать, чтобы макет «в три приема» был превращен в рабочий продукт. Очень часто это отрицательно сказывается на управлении разработкой ПО.

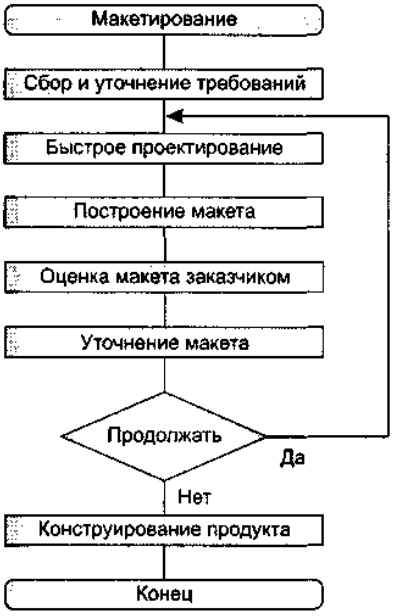

Рис. 1.3. Последовательность действий при макетировании

С другой стороны, для быстрого получения работающего макета разработчик часто идет на определенные компромиссы. Могут использоваться не самые подходящие язык программирования или операционная система. Для простой демонстрации возможностей может применяться неэффективный алгоритм. Спустя некоторое время разработчик забывает о причинах, по которым эти средства не подходят. В результате далеко не идеальный выбранный вариант интегрируется в систему.

Очевидно, что преодоление этих недостатков требует борьбы с житейским соблазном — принять желаемое за действительное.

Стратегии конструирования ПО

Существуют 3 стратегии конструирования ПО:

- *однократный проход* (водопадная стратегия) — линейная последовательность этапов конструирования;
- *инкрементная стратегия.* В начале процесса определяются все пользовательские и системные требования, оставшаяся часть конструирования выполняется в виде последовательности версий. Первая версия реализует часть запланированных возможностей, следующая версия реализует дополнительные возможности и т. д., пока не будет

получена полная система;

- *эволюционная стратегия.* Система также строится в виде последовательности версий, но в начале процесса определены не все требования. Требования уточняются в результате разработки версий.

Характеристики стратегий конструирования ПО в соответствии с требованиями стандарта IEEE/EIA 12207.2 приведены в табл. 1.1.

Таблица 1.1. Характеристики стратегий конструирования

Стратегия конструирования	В начале процесса определены все требования?	Множество циклов конструирования?	Промежуточное ПО распространяется?
Однократный проход	Да	Нет	Нет
Инкрементная (запланированное улучшение продукта)	Да	Да	Может быть
Эволюционная	Нет	Да	Да

Инкрементная 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.

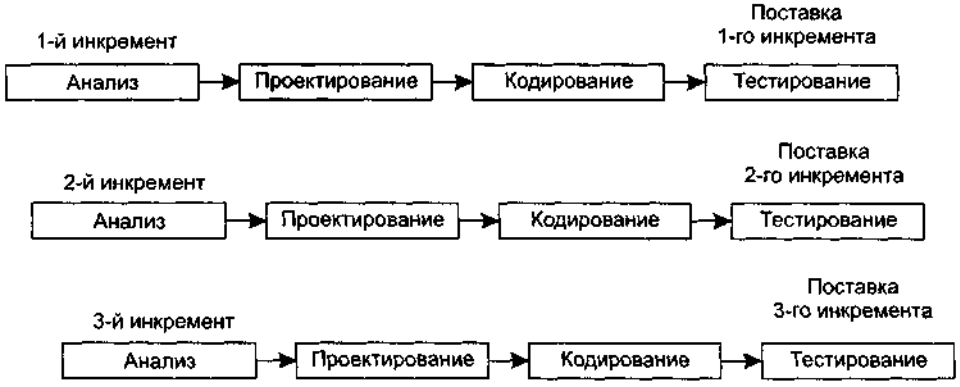

Рис. 1.4. Инкрементная модель

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

Быстрая разработка приложений

Модель быстрой разработки приложений (Rapid Application Development) — второй пример применения инкрементной стратегии конструирования (рис. 1.5).

RAD-модель обеспечивает экстремально короткий цикл разработки. RAD — высокоскоростная адаптация линейной последовательной модели, в которой быстрая разработка достигается за счет использования компонентно-ориентированного конструирования. Если требования полностью определены, а проектная область ограничена, RAD-процесс позволяет группе создать полностью функциональную систему за очень короткое время (60-90 дней). RAD-подход ориентирован на разработку информационных систем и выделяет следующие этапы:

- **бизнес-моделирование.** Моделируется информационный поток между бизнес-функциями. Ищется ответ на следующие вопросы: Какая информация руководит бизнес-процессом? Какая генерируется информация? Кто генерирует ее? Где информация применяется? Кто обрабатывает ее?
- **моделирование данных**. Информационный поток, определенный на этапе бизнес-моделирования, отображается в набор объектов данных, которые требуются для поддержки бизнеса. Идентифицируются характеристики (свойства, атрибуты) каждого объекта, определяются отношения между объектами;
- **моделирование обработки.** Определяются преобразования объектов данных, обеспечивающие реализацию бизнес-

функций. Создаются описания обработки для добавления, модификации, удаления или нахождения (исправления) объектов данных;

- **генерация приложения.** Предполагается использование методов, ориентированных на языки программирования 4-го поколения. Вместо создания ПО с помощью языков программирования 3-го поколения, RAD-процесс работает с повторно используемыми программными компонентами или создает повторно используемые компоненты. Для обеспечения конструирования используются утилиты автоматизации;
- **тестирование и объединение.** Поскольку применяются повторно используемые компоненты, многие программные элементы уже протестированы. Это уменьшает время тестирования (хотя все новые элементы должны быть протестированы).

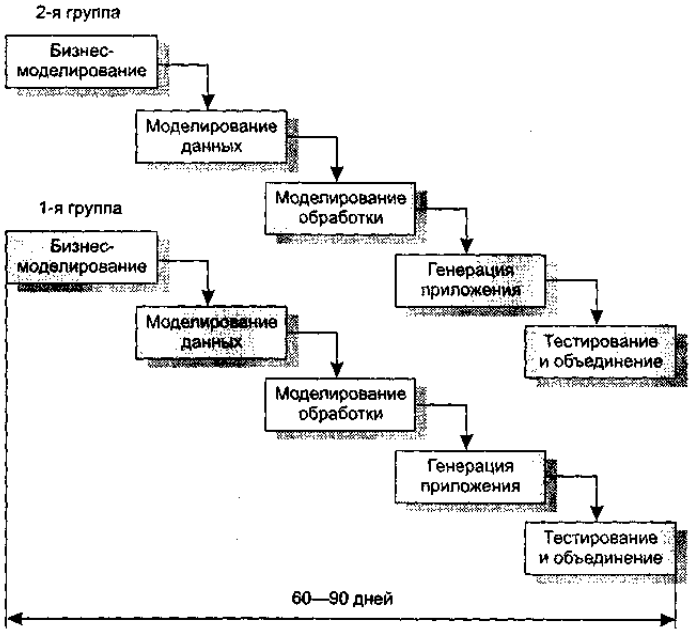

Рис. 1.5. Модель быстрой разработки приложений

Применение RAD возможно в том случае, когда каждая главная функция может быть завершена за 3 месяца. Каждая главная функция адресуется отдельной группе разработчиков, а затем интегрируется в целую систему.

Применение RAD имеет- и свои недостатки, и ограничения.

1. Для больших проектов в RAD требуются существенные людские ресурсы (необходимо создать достаточное количество групп).
2. RAD применима только для таких приложений, которые могут декомпозироваться на отдельные модули и в которых производительность не является критической величиной.
3. RAD не применима в условиях высоких технических рисков (то есть при использовании новой технологии).

Спиральная модель

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].

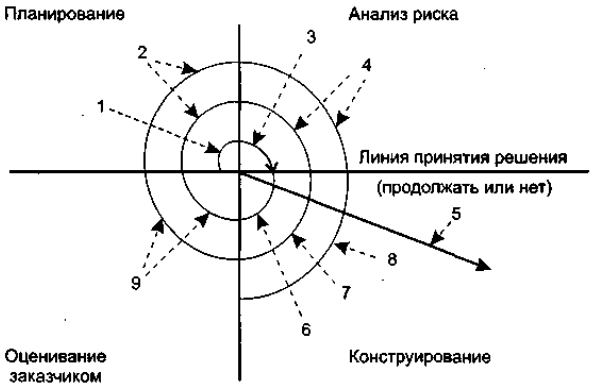

Рис. 1.6. Спиральная модель: *1* — начальный сбор требований и планирование проекта; *2* — та же работа, но на основе рекомендаций заказчика; 3 — анализ риска на основе начальных требований; *4* — анализ риска на основе реакции заказчика; 5 — переход к комплексной системе; 6 — начальный макет системы;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

Компонентно-ориентированная модель

Компонентно-ориентированная модель является развитием спиральной модели и тоже основывается на эволюционной стратегии конструирования. В этой модели конкретизируется содержание квадранта конструирования — оно отражает тот факт, что в современных условиях новая разработка должна основываться на повторном использовании существующих программных компонентов (рис. 1.7).

Рис. 1.7. Компонентно-ориентированная модель

Программные компоненты, созданные в реализованных программных проектах, хранятся в библиотеке. В новом программном проекте, исходя из требований заказчика, выявляются кандидаты в компоненты. Далее проверяется наличие этих кандидатов в библиотеке. Если они найдены, то компоненты извлекаются из библиотеки и используются повторно. В противном случае создаются новые компоненты, они применяются в проекте и включаются в библиотеку.

Достоинства компонентно-ориентированной модели:

1) уменьшает на 30% время разработки программного продукта;
2) уменьшает стоимость программной разработки до 70%;
3) увеличивает в полтора раза производительность разработки.

Тяжеловесные и облегченные процессы

В XXI веке потребности общества в программном обеспечении информационных технологий достигли экстремальных значений. Программная индустрия буквально «захлебывается» от потока самых разнообразных заказов. «Больше процессов разработки, хороших и разных!» — скандируют заказчики. «Сейчас, сейчас! Только об этом и думаем!» — отвечают разработчики.

Традиционно для упорядочения и ускорения программных разработок предлагались строго упорядочивающие тяжеловесные (heavyweight) процессы. В этих процессах прогнозируется весь объем предстоящих работ, поэтому они называются прогнозирующими (predictive) процессами. Порядок, который должен выполнять при этом человек-разработчик, чрезвычайно строг — «шаг вправо, шаг влево — виртуальный расстрел!». Иными словами, человеческие слабости в расчет не принимаются, а объем необходимой документации способен отнять покой и сон у «совестливого» разработчика.

В последние годы появилась группа новых, облегченных (lightweight) процессов [29]. Теперь их называют подвижными (agile) процессами [8], [25], [36]. Они привлекательны отсутствием бюрократизма, характерного для тяжеловесных (прогнозирующих) процессов. Новые процессы должны воплотить в жизнь разумный компромисс между слишком строгой дисциплиной и полным ее отсутствием. Иначе говоря, порядка в них достаточно для того, чтобы получить разумную отдачу от разработчиков.

Подвижные процессы требуют меньшего объема документации и ориентированы на человека. В них явно указано на необходимость использования природных качеств человеческой натуры (а не на применение действий, направленных наперекор этим качествам).

Более того, подвижные процессы учитывают особенности современного заказчика, а именно частые изменения его требований к программному продукту. Известно, что для прогнозирующих процессов частые изменения требований подобны смерти. В отличие от них, подвижные процессы адаптируют изменения требований и даже выигрывают от этого. Словом, подвижные процессы имеют адаптивную природу.

Таким образом, в современной инфраструктуре программной инженерии существуют два семейства процессов разработки:

- семейство прогнозирующих (тяжеловесных) процессов;
- семейство адаптивных (подвижных, облегченных) процессов.

У каждого семейства есть свои достоинства, недостатки и область применения:

- адаптивный процесс используют при частых изменениях требований, малочисленной группе высококвалифицированных разработчиков и грамотном заказчике, который согласен участвовать в разработке;
- прогнозирующий процесс применяют при фиксированных требованиях и многочисленной группе разработчиков разной квалификации.

XP-процесс

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.

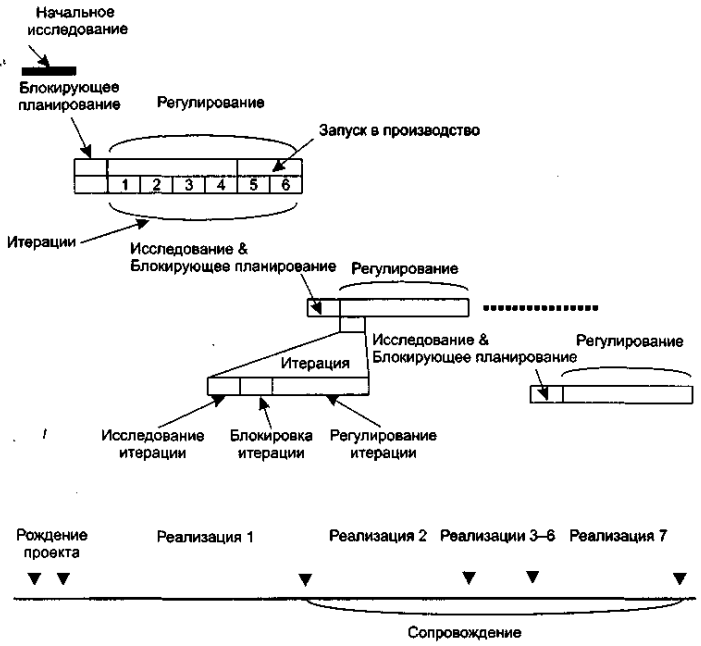

Рис. 1.8. Идеальный XP-процесс

Модели качества процессов конструирования

В современных условиях, условиях жесткой конкуренции, очень важно гарантировать высокое качество вашего процесса конструирования ПО. Такую гарантию дает сертификат качества процесса, подтверждающий его соответствие принятым международным стандартам. Каждый такой стандарт фиксирует свою модель обеспечения качества. Наиболее авторитетны модели стандартов ISO 9001:2000, ISO/ IEC 15504 и модель зрелости процесса конструирования ПО (Capability Maturity Model — CMM) Института программной инженерии при американском университете Карнеги-Меллон.

Модель стандарта ISO 9001:2000 ориентирована на процессы разработки из любых областей человеческой деятельности. Стандарт ISO/IEC 15504 специализируется на процессах программной разработки и отличается более высоким уровнем детализации. Достаточно сказать, что объем этого стандарта превышает 500 страниц. Значительная часть идей ISO/IEC 15504 взята из модели CMM.

Базовым понятием модели CMM считается *зрелость* компании [61], [62]. Незрелой называют компанию, где процесс конструирования ПО и принимаемые решения зависят только от таланта конкретных разработчиков. Как следствие, здесь

высока вероятность превышения бюджета или срыва сроков окончания проекта.

Напротив, в зрелой компании работают ясные процедуры управления проектами и построения программных продуктов. По мере необходимости эти процедуры уточняются и развиваются. Оценки длительности и затрат разработки точны, основываются на накопленном опыте. Кроме того, в компании имеются и действуют корпоративные стандарты на процессы взаимодействия с заказчиком, процессы анализа, проектирования, программирования, тестирования и внедрения программных продуктов. Все это создает среду, обеспечивающую качественную разработку программного обеспечения.

Таким образом, модель СММ фиксирует критерии для оценки зрелости компании и предлагает рецепты для улучшения существующих в ней процессов. Иными словами, в ней не только сформулированы условия, необходимые для достижения минимальной организованности процесса, но и даются рекомендации по дальнейшему совершенствованию процессов.

Очень важно отметить, что модель СММ ориентирована на построение системы постоянного улучшения процессов. В ней зафиксированы пять уровней зрелости (рис. 1.9) и предусмотрен плавный, поэтапный подход к совершенствованию процессов — можно поэтапно получать подтверждения об улучшении процессов после каждого уровня зрелости.

Рис. 1.9. Пять уровней зрелости модели СММ

Начальный уровень (уровень 1) означает, что процесс в компании не формализован. Он не может строго планироваться и отслеживаться, его успех носит случайный характер. Результат работы целиком и полностью зависит от личных качеств отдельных сотрудников. При увольнении таких сотрудников проект останавливается.

Для перехода на **повторяемый** уровень (уровень 2) необходимо внедрить формальные процедуры для выполнения основных элементов процесса конструирования. Результаты выполнения процесса соответствуют заданным требованиям и стандартам. Основное отличие от уровня 1 состоит в том, что выполнение процесса планируется и контролируется. Применяемые средства планирования и управления дают возможность повторения ранее достигнутых успехов.

Следующий, **определенный** уровень (уровень 3) требует, чтобы все элементы процесса были определены, стандартизованы и задокументированы. Основное отличие от уровня 2 заключается в том, что элементы процесса уровня 3 планируются и управляются на основе единого стандарта компании. Качество разрабатываемого ПО уже не зависит от способностей отдельных личностей.

С переходом на **управляемый** уровень (уровень 4) в компании принимаются количественные показатели качества как программных продуктов, так и процесса. Это обеспечивает более точное планирование проекта и контроль качества его результатов. Основное отличие от уровня 3 состоит в более объективной, количественной оценке продукта и процесса.

Высший, **оптимизирующий** уровень (уровень 5) подразумевает, что главной задачей компании становится постоянное улучшение и повышение эффективности существующих процессов, ввод новых технологий. Основное отличие от уровня 4 заключается в том, что технология создания и сопровождения программных продуктов планомерно и последовательно совершенствуется.

Каждый уровень СММ характеризуется *областью ключевых процессов* (ОКП), причем считается, что каждый последующий уровень включает в себя все характеристики предыдущих уровней. Иначе говоря, для 3-го уровня зрелости рассматриваются ОКП 3-го уровня, ОКП 2-го уровня и ОКП 1-го уровня. Область ключевых процессов образуют процессы, которые при совместном выполнении приводят к достижению определенного набора целей. Например, ОКП 5-го уровня образуют процессы:

- предотвращения дефектов;
- управления изменениями технологии;
- управления изменениями процесса.

Если все цели ОКП достигнуты, компании присваивается сертификат данного уровня зрелости. Если хотя бы одна цель не достигнута, то компания не может соответствовать данному уровню СММ.

Процесс руководства проектом

Руководство программным проектом — первый слой процесса конструирования ПО. Термин «слой» подчеркивает, что руководство определяет сущность процесса разработки от его начала до конца. Принцип руководства иллюстрирует рис. 2.1.

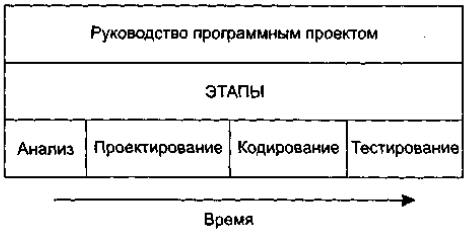

Рис. 2.1. Руководство в процессе конструирования ПО

На этом рисунке прямоугольник обозначает процесс конструирования, в нем выделены этапы, а вверху, над каждым из этапов, размещен слой деятельности *«руководство программным проектом».*

Для проведения успешного проекта нужно понять объем предстоящих работ, возможный риск, требуемые ресурсы, предстоящие задачи, прокладываемые вехи, необходимые усилия (стоимость), план работ, которому желательно следовать. Руководство программным проектом обеспечивает такое понимание. Оно начинается перед технической работой, продолжается по мере развития ПО от идеи к реальности и достигает наивысшего уровня к концу работ [32], [64], [69].

Начало проекта

Перед планированием проекта следует:

- установить цели и проблемную область проекта;
- обсудить альтернативные решения;
- выявить технические и управленческие ограничения.

Измерения, меры и метрики

Измерения помогают понять как процесс разработки продукта, так и сам продукт. Измерения процесса производятся в целях его улучшения, измерения продукта — для повышения его качества. В результате измерения определяется *мера* — количественная характеристика какого-либо свойства объекта. Путем непосредственных измерений могут определяться только опорные свойства объекта. Все остальные свойства оцениваются в результате вычисления тех или иных функций от значений опорных характеристик. Вычисления этих функций проводятся по формулам, дающим числовые значения и называемым *метриками.*

В *IEEE Standard Glossary of Software Engineering Terms* метрика определена как мера степени обладания свойством, имеющая числовое значение. В программной инженерии понятия *мера* и *метрика* очень часто рассматривают как синонимы.

Процесс оценки

При планировании программного проекта надо оценить людские ресурсы (в человеко-месяцах), продолжительность (в календарных датах), стоимость (в тысячах долларов). Обычно исходят из прошлого опыта. Если новый проект по размеру и функциям похож на предыдущий проект, вполне вероятно, что потребуются такие же ресурсы, время и деньги.

Анализ риска

На этой стадии исследуется область неопределенности, имеющаяся в наличии перед созданием программного продукта. Анализируется ее влияние на проект. Нет ли скрытых от внимания трудных технических проблем? Не станут ли изменения, проявившиеся в ходе проектирования, причиной недопустимого отставания по срокам? В результате принимается решение — выполнять проект или нет.

Планирование

Определяется набор проектных задач. Устанавливаются связи между задачами, оценивается сложность каждой задачи. Определяются людские и другие ресурсы. Создается сетевой график задач, проводится его временная разметка.

Трассировка и контроль

Каждая задача, помеченная в плане, отслеживается руководителем проекта. При отставании в решении задачи применяются утилиты повторного планирования. С помощью утилит определяется влияние этого отставания на промежуточную веху и общее время конструирования. Под вехой понимается временная метка, к которой привязано подведение промежуточных итогов.

В результате повторного планирования:

- могут быть перераспределены ресурсы;
- могут быть реорганизованы задачи;
- могут быть пересмотрены выходные обязательства.

КЛАССИЧЕСКИЕ МЕТОДЫ АНАЛИЗА

В этой главе рассматриваются классические методы анализа требований, ориентированные на процедурную реализацию программных систем. Анализ требований служит мостом между неформальным описанием требований, выполняемым заказчиком, и проектированием системы. Методы анализа призваны формализовать обязанности системы, фактически их применение дает ответ на вопрос: что должна делать будущая система?

Структурный анализ

Структурный анализ — один из формализованных методов анализа требований к ПО. Автор этого метода — Том Де Марко (1979) [27]. В этом методе программное изделие рассматривается как преобразователь информационного потока данных. Основной элемент структурного анализа — диаграмма потоков данных.

Диаграммы потоков данных

Диаграмма потоков данных ПДД — графическое средство для изображения информационного потока и преобразований, которым подвергаются данные при движении от входа к выходу системы. Элементы диаграммы имеют вид, показанный на рис. 3.1. Диаграмма может использоваться для представления программного изделия на любом уровне абстракции.

Пример системы взаимосвязанных диаграмм показан на рис. 3.2.

Диаграмма высшего (нулевого) уровня представляет систему как единый овал со стрелкой, ее называют основной или контекстной моделью. Контекстная модель используется для указания внешних связей программного изделия. Для детализации (уточнения системы) вводится диаграмма 1-го уровня. Каждый из преобразователей этой диаграммы — подфункция общей системы. Таким образом, речь идет о замене преобразователя *F* на целую систему преобразователей.

Дальнейшее уточнение (например, преобразователя *F3)* приводит к диаграмме 2-го уровня. Говорят, что ПДД1 разбивается на диаграммы 2-го уровня.

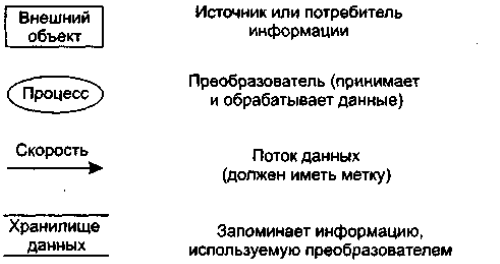

Рис. 3.1. Элементы диаграммы потоков данных

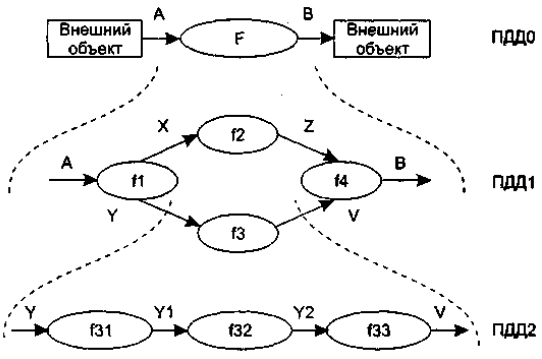

Рис. 3.2. Система взаимосвязанных диаграмм потоков данных

ПРИМЕЧАНИЕ

Важно сохранить непрерывность информационного потока и его согласованность. Это значит, что входы и выходы у каждого преобразователя на любом уровне должны оставаться прежними. В диаграмме отсутствуют точные указания на последовательность обработки. Точные указания откладываются до этапа проектирования.

Диаграмма потоков данных — это абстракция, граф. Для связи графа с проблемной областью (превращения в граф-модель) надо задать интерпретацию ее компонентов — дуг и вершин.

Описание потоков данных и процессов

Базовые средства диаграммы не обеспечивают полного описания требований к программному изделию. Очевидно, что должны быть описаны стрелки — потоки данных — и преобразователи — процессы. Для этих целей используются словарь требований (данных) и спецификации процессов.

Словарь требований (данных) содержит описания потоков данных и хранилищ данных. Словарь требований является неотъемлемым элементом любой CASE-утилиты автоматизации анализа. Структура словаря зависит от особенностей конкретной CASE-утилиты. Тем не менее можно выделить базисную информацию типового словаря требований.

Большинство *словарей* содержит следующую информацию.

1. *Имя* (основное имя элемента данных, хранилища или внешнего объекта).
2. *Прозвище* (Alias) — другие имена того же объекта.
3. *Где и как используется объект* — список процессов, которые используют данный элемент, с указанием способа использования (ввод в процесс, вывод из процесса, как внешний объект или как память).
4. *Описание содержания* — запись для представления содержания.
5. *Дополнительная информация* — дополнительные сведения о типах данных, допустимых значениях, ограничениях и т. д.

Спецификация процесса — это описание преобразователя. Спецификация поясняет: ввод данных в преобразователь, алгоритм обработки, характеристики производительности преобразователя, формируемые результаты.

Количество спецификаций равно количеству преобразователей диаграммы.

Расширения для систем реального времени

Как известно, программное изделие (ПИ) является дискретной моделью проблемной области, взаимодействующей с непрерывными процессами физического мира (рис. 3.3).

Рис. 3.3. Программное изделие как дискретная модель проблемной области

П. Вард и С. Меллор приспособили диаграммы потоков данных к следующим требованиям систем реального времени [73].

1. Информационный поток накапливается или формируется в непрерывном времени.
2. Фиксируется управляющая информация. Считается, что она проходит через систему и связывается с управляющей обработкой.
3. Допускается множественный запрос на одну и ту же обработку (из внешней среды).

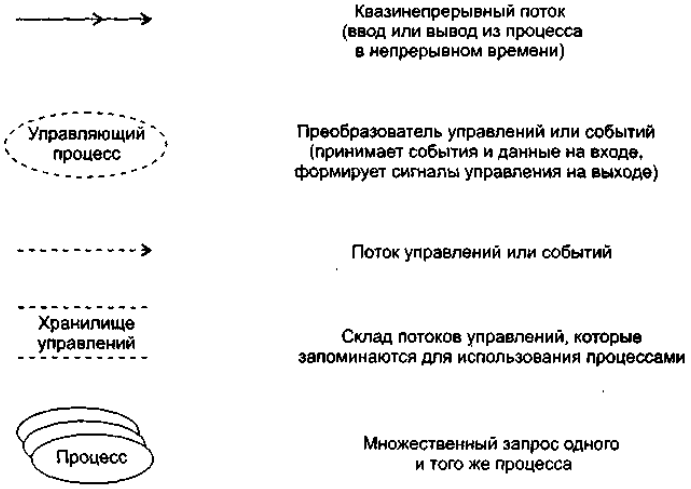

Рис. 3.4. Расширения диаграмм для систем реального времени

Новые элементы имеют обозначения, показанные на рис. 3.4.

Приведем два примера использования новых элементов.

Пример 1. Использование потоков, непрерывных во времени.

На рис. 3.5 представлена модель анализа программного изделия для системы слежения за газовой турбиной.

Рис. 3.5. Модель ПО для системы слежения за газовой турбиной

Видим, что здесь наблюдаемая температура измеряется непрерывно до тех пор, пока не будет найдено дискретное значение в наборе эталонов температуры. Преобразователь формирует регулирующие воздействия как непрерывный во времени вывод. Чем полезна эта модель?

Во-первых, инженер делает вывод, что для приема-передачи квазинепрерывных значений нужно использовать аналого-цифровую и цифроаналоговую аппаратуру.

Во-вторых, необходимость организации высокоскоростного управления этой аппаратурой делает критичным требование к производительности системы.

Пример 2. Использование потоков управления.

Рассмотрим компьютерную систему, которая управляет роботом (рис. 3.6).

Рис. 3.6. Модель ПО для управления роботом

Установка в прибор деталей, собранных роботом, фиксируется установкой бита в буфере состояния деталей (он показывает присутствие или отсутствие каждой детали). Информация о событиях, запоминаемых в буфере, посылается в виде строки битов в преобразователь «Наблюдение за прибором». Преобразователь читает команды оператора только тогда, когда управляющая информация (битовая строка) показывает наличие всех деталей. Флаг события (Старт-Стоп) посылается в управляющий преобразователь «Начать движение», который руководит дальнейшей командной обработкой. Потоки данных посылаются в преобразователь команд роботу при наличии события «Процесс активен».

Расширение возможностей управления

Д. Хетли и И. Пирбхаи сосредоточили внимание на аспектах управления программным продуктом [34]. Они выделили системные состояния и механизм перехода из одного состояния в другое. Д. Хетли и И. Пирбхаи предложили не вносить в ПДД элементы управления, такие как потоки управления и управляющие процессы. Вместо этого они ввели диаграммы управляющих потоков (УПД).

Диаграмма управляющих потоков содержит:

- обычные преобразователи (управляющие преобразователи исключены вообще);
- потоки управления и потоки событий (без потоков данных).

Вместо управляющих преобразователей в УПД используются указатели — ссылки на управляющую спецификацию УСПЕЦ. Как показано на рис. 3.7, ссылка изображается как косая пунктирная стрелка, указывающая на окно УСПЕЦ (вертикальную черту).

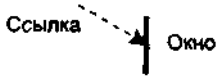

Рис. 3.7. Изображение ссылки на управляющую спецификацию

УСПЕЦ управляет преобразователями в ПДД на основе события, которое проходит в ее окно (по ссылке). Она предписывает включение конкретных преобразователей как результат конкретного события.

Иллюстрация модели программной системы, использующей описанные средства, приведена на рис. 3.8.

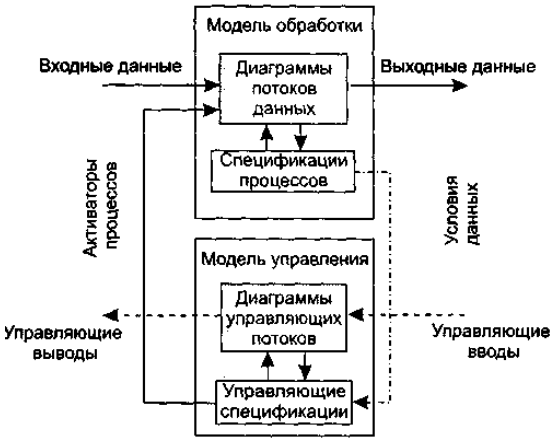

Рис. 3.8. Композиция модели обработки и управления

В модель обработки входит набор диаграмм потоков данных и набор спецификаций процессов. Модель управления образует набор диаграмм управляющих потоков и набор управляющих спецификаций. Модель обработки подключается к модели управления с помощью активаторов процессов. Активаторы включают в конкретной ПДД конкретные преобразователи. Обратная связь модели обработки с моделью управления осуществляется с помощью условий данных. Условия данных формируются в ПДД (когда входные данные преобразуются в события).

Модель системы регулирования давления космического корабля

Обсудим модель системы регулирования давления космического корабля, представленную на рис. 3.9.

Начнем с диаграммы потоков данных. Основной процесс в ПДД — Слежение и регулирование давления. На его входы поступают: измеренное Давление в кабине и Max давление: На выходе процесса — поток данных Изменение давления. Содержание процесса описывается в его спецификации ПСПЕЦ.

Спецификация процесса ПСПЕЦ может включать:

1) поясняющий текст (обязательно);
2) описание алгоритма обработки;
3) математические уравнения;
4) таблицы;
5) диаграммы.

Элементы со второго по пятый не обязательны.

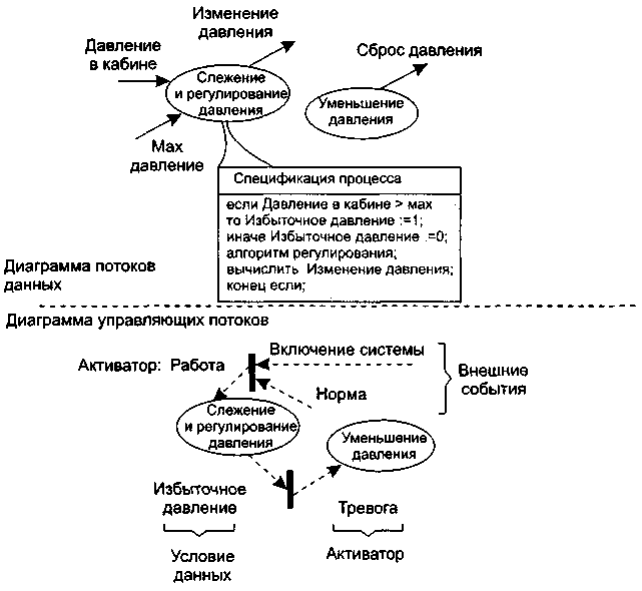

Рис. 3.9. Модель системы регулирования давления космического корабля

С помощью ПСПЕЦ разработчик создает описание для каждого преобразователя, которое рассматривается как:

- первый шаг создания спецификации требований к программному изделию;
- руководство для проектирования программ, которые будут реализовывать процессы.

В нашем примере спецификация процесса имеет вид

если Давление в кабине > max
то Избыточное давление:=11;
иначе Избыточное давление:=0;

алгоритм регулирования;
выч.Изменение давления;
конец если;

Таким образом, когда давление в кабине превышает максимум, генерируется управляющее событие Избыточное давление. Оно должно быть показано на диаграмме управляющих потоков *УПД*. Это событие входит в окно управляющей спецификации *УСПЕЦ*.

Управляющая спецификация моделирует поведение системы. Она содержит:

- таблицу активации процессов *(ТАП);*
- диаграмму переходов-состояний *(ДПС).*

Таблица активации процессов показывает, какие процессы будут вызываться (активироваться) в потоковой модели в результате конкретных событий.

ТАП включает три раздела — Входные события, Выходные события, Активация процессов. Логика работы *ТАП* такова: входное событие вызывает выходное событие, которое активирует конкретный процесс. Для нашей модели *ТАП* имеет вид, представленный в табл. 3.1.

Таблица 3.1. Таблица активации процессов

Входные события:			
Включение системы	1	0	0
Избыточное давление	0	1	0
Норма	0	0	1
Выходные события:			
Тревога	0	1	0
Работа	1	0	1
Активация процессов:			
Слежение и регулирование давления	1	0	1
Уменьшение давления	0	1	0

Видим, что в нашем примере входных событий три: два внешних события (Включение системы, Норма) и одно — условие данных (Избыточное Давление). Работа ТАП инициируется входным событием, «втекающим» в окно УСПЕЦ. В результате ТАП вырабатывает выходное событие — активатор. В нашем примере активаторами являются события Работа и Тревога. Активатор «вытекает» из окна УСПЕЦ, запуская в УПД конкретный процесс.

Другой элемент УСПЕЦ — Диаграмма переходов-состояний. ДПС отражает состояния системы и показывает, как она переходит из одного состояния в другое.

ДПС для нашей модели показана на рис. 3.10.

Системные состояния показаны прямоугольниками. Стрелки показывают переходы между состояниями. Стрелки переходов подписывают следующим образом: в числителе — событие, которое вызывает переход, в знаменателе — процесс, запускаемый как результат события.

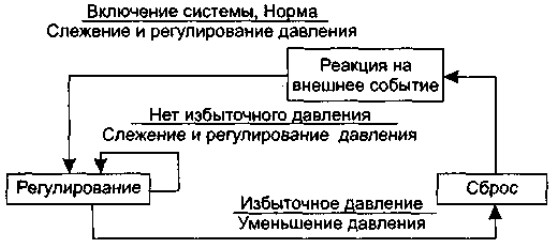

Рис. 3.10. Диаграмма переходов-состояний

Изучая ДПС, разработчик может анализировать поведение модели и установить, нет ли «дыр» в определении поведения.

Методы анализа, ориентированные на структуры данных

Элементами проблемной области для любой системы являются потоки, процессы и структуры данных. При структурном анализе активно работают только с потоками данных и процессами.

Методы, ориентированные на структуры данных, обеспечивают:

1) определение ключевых информационных объектов и операций;
2) определение иерархической структуры данных;
3) компоновку структур данных из типовых конструкций — последовательности, выбора, повторения;
4) последовательность шагов для превращения иерархической структуры данных в структуру программы.

Наиболее известны два метода: метод Варнье-Орра и метод Джексона.

В методе Варнье-Орра для представления структур применяют диаграммы Варнье [54].

Для построения диаграмм Варнье используют 3 базовых элемента: последовательность, выбор, повторение (рис. 3.11) [74].

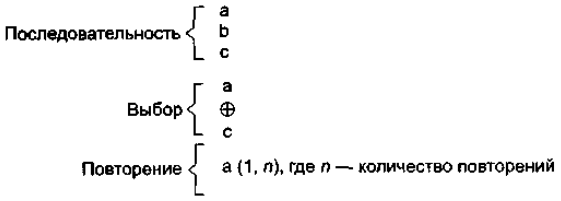

Рис. 3.11. Базовые элементы в диаграммах Варнье

Как показано на рис. 3.12, с помощью этих элементов можно строить информационные структуры с любым количеством уровней иерархии.

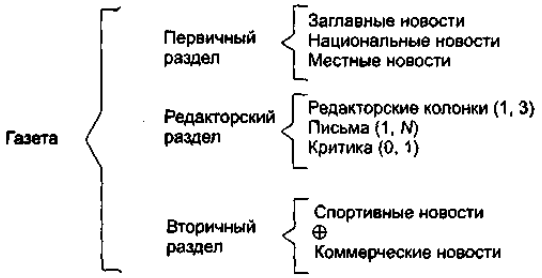

Рис. 3.12. Структура газеты в виде диаграммы Варнье

Как видим, для представления структуры газеты здесь используются три уровня иерархии.

ОСНОВЫ ПРОЕКТИРОВАНИЯ ПРОГРАММНЫХ СИСТЕМ

В этой главе рассматривается содержание этапа проектирования и его место в жизненном цикле конструирования программных систем. Дается обзор архитектурных моделей ПО, обсуждаются классические проектные характеристики: модульность, информационная закрытость, сложность, связность, сцепление и метрики для их оценки.

Особенности процесса синтеза программных систем

Известно, что технологический цикл конструирования программной системы (ПС) включает три процесса — анализ, синтез и сопровождение.

В ходе анализа ищется ответ на вопрос: «Что должна делать будущая система?». Именно на этой стадии закладывается фундамент успеха всего проекта. Известно множество неудачных реализаций из-за неполноты и неточностей в определении требований к системе.

В процессе синтеза формируется ответ на вопрос: «Каким образом система будет реализовывать предъявляемые к ней требования?». Выделяют три этапа синтеза: проектирование ПС, кодирование ПС, тестирование ПС (рис. 4.1).

Рассмотрим информационные потоки процесса синтеза.

Этап проектирования питают требования к ПС, представленные информационной, функциональной и поведенческой моделями анализа. Иными словами, модели анализа поставляют этапу проектирования исходные сведения для работы. Информационная модель описывает информацию, которую, по мнению заказчика, должна обрабатывать ПС. Функциональная модель определяет перечень функций обработки. Поведенческая модель фиксирует желаемую динамику системы (режимы ее работы). На выходе этапа проектирования — разработка данных, разработка архитектуры и процедурная разработка ПС.

Разработка данных — это результат преобразования информационной модели анализа в структуры данных, которые потребуются для реализации программной системы.

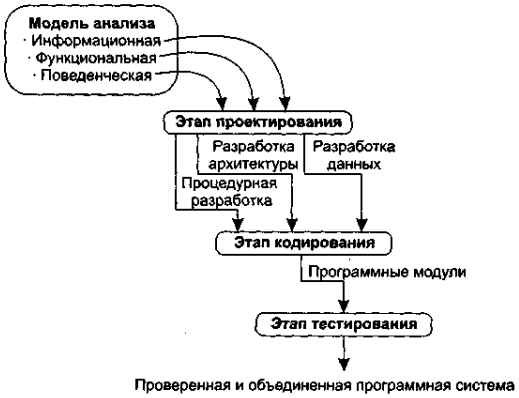

Рис. 4.1. Информационные потоки процесса синтеза ПС

Разработка архитектуры выделяет основные структурные компоненты и фиксирует связи между ними.

Процедурная разработка описывает последовательность действий в структурных компонентах, то есть определяет их содержание.

Далее создаются тексты программных модулей, проводится тестирование для объединения и проверки ПС. На проектирование, кодирование и тестирование приходится более 75% стоимости конструирования ПС. Принятые здесь решения оказывают решающее воздействие на успех реализации ПС и легкость, с которой ПС будет сопровождаться.

Следует отметить, что решения, принимаемые в ходе проектирования, делают его стержневым этапом процесса синтеза. Важность проектирования можно определить одним словом — качество. Проектирование — этап, на котором «выращивается» качество разработки ПС. Справедлива следующая аксиома разработки: может быть плохая ПС при хорошем проектировании, но не может быть хорошей ПС при плохом проектировании. Проектирование обеспечивает нас такими представлениями ПС, качество которых можно оценить. Проектирование — единственный путь, обеспечивающий правильную трансляцию требований заказчика в конечный программный продукт.

Особенности этапа проектирования

Проектирование — итерационный процесс, при помощи которого требования к ПС транслируются в инженерные представления ПС. Вначале эти представления дают только концептуальную информацию (на высоком уровне абстракции), последующие уточнения приводят к формам, которые близки к текстам на языках программирования.

Обычно в проектировании выделяют две ступени: предварительное проектирование и детальное проектирование. Предварительное проектирование формирует абстракции архитектурного уровня, детальное проектирование уточняет эти абстракции, добавляет подробности алгоритмического уровня. Кроме того, во многих случаях выделяют интерфейсное проектирование, цель которого — сформировать графический интерфейс пользователя (GUI). Схема информационных связей процесса проектирования приведена на рис. 4.2.

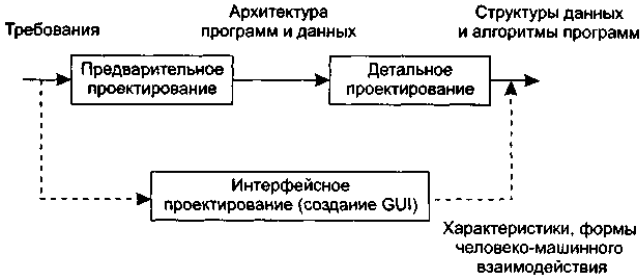

Рис. 4.2. Информационные связи процесса проектирования

Предварительное проектирование обеспечивает:

- идентификацию подсистем;
- определение основных принципов управления подсистемами, взаимодействия подсистем.

Предварительное проектирование включает три типа деятельности:

1. *Структурирование системы.* Система структурируется на несколько подсистем, где под подсистемой понимается независимый программный компонент. Определяются взаимодействия подсистем.
2. *Моделирование управления.* Определяется модель связей управления между частями системы.
3. *Декомпозиция подсистем на модули.* Каждая подсистема разбивается на модули. Определяются типы модулей и межмодульные соединения.

Рассмотрим вопросы структурирования, моделирования и декомпозиции более подробно.

Структурирование системы

Известны четыре модели системного структурирования:

- модель хранилища данных;
- модель клиент-сервер;
- трехуровневая модель;
- модель абстрактной машины.

В *модели хранилища данных* (рис. 4.3) подсистемы разделяют данные, находящиеся в общей памяти. Как правило, данные образуют БД. Предусматривается система управления этой базой.

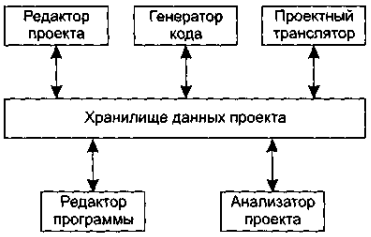

Рис. 4.3. Модель хранилища данных

Модель клиент-сервер используется для распределенных систем, где данные распределены по серверам (рис. 4.4). Для передачи данных применяют сетевой протокол, например TCP/IP.

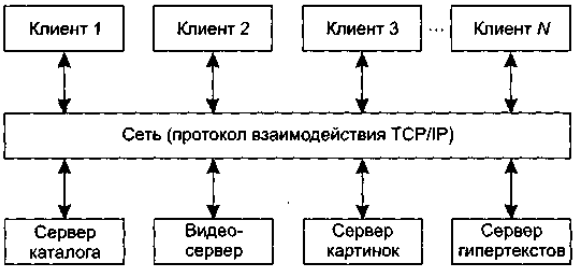

Рис. 4.4. Модель клиент-сервер

Трехуровневая модель является развитием модели клиент-сервер (рис. 4.5).

Рис. 4.5. Трехуровневая модель

Уровень графического интерфейса пользователя запускается на машине клиента. Бизнес-логику образуют модули, осуществляющие функциональные обязанности системы. Этот уровень запускается на сервере приложения. Реляционная СУБД хранит данные, требуемые уровню бизнес-логики. Этот уровень запускается на втором сервере — сервере базы данных.

Преимущества трехуровневой модели:

- упрощается такая модификация уровня, которая не влияет на другие уровни;
- отделение прикладных функций от функций управления БД упрощает оптимизацию всей системы.

Модель абстрактной машины отображает многослойную систему (рис. 4.6).

Каждый текущий слой реализуется с использованием средств, обеспечиваемых слоем-фундаментом.

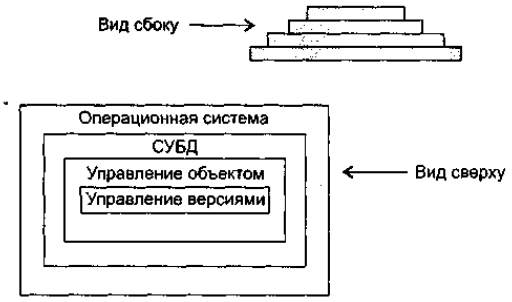

Рис. 4.6. Модель абстрактной машины

Моделирование управления

Известны два типа моделей управления:

- модель централизованного управления;
- модель событийного управления.

В модели централизованного управления одна подсистема выделяется как системный контроллер. Ее обязанности — руководить работой других подсистем. Различают две разновидности моделей централизованного управления: *модель вызов-возврат* (рис. 4.7) и *Модель менеджера* (рис. 4.8), которая используется в системах параллельной обработки.

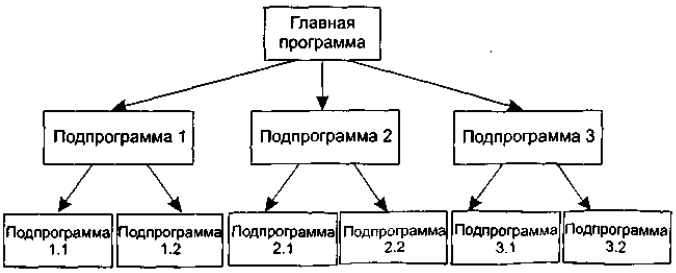

Рис. 4.7. Модель вызов-возврат

В модели событийного управления системой управляют внешние события. Используются две разновидности модели

событийного управления: широковещательная модель и модель, управляемая прерываниями.

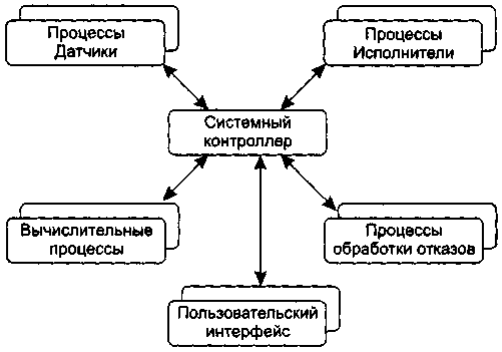

Рис. 4.8. Модель менеджера

В *широковещательной модели* (рис. 4.9) каждая подсистема уведомляет обработчика о своем интересе к конкретным событиям. Когда событие происходит, обработчик пересылает его подсистеме, которая может обработать это событие. Функции управления в обработчик не встраиваются.

Рис. 4.9. Широковещательная модель

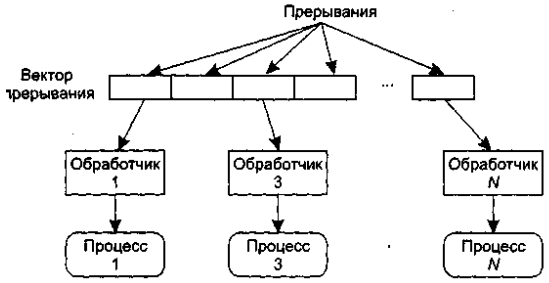

Рис. 4.10. Модель, управляемая прерываниями

В модели, управляемой прерываниями (рис. 4.10), все прерывания разбиты на группы — типы, которые образуют вектор прерываний. Для каждого типа прерывания есть свой обработчик. Каждый обработчик реагирует на свой тип прерывания и запускает свой процесс.

Декомпозиция подсистем на модули

Известны два типа моделей модульной декомпозиции:

- модель потока данных;
- модель объектов.

В основе модели потока данных лежит разбиение по функциям.

Модель объектов основана на слабо сцепленных сущностях, имеющих собственные наборы данных, состояния и наборы операций.

Очевидно, что выбор типа декомпозиции должен определяться сложностью разбиваемой подсистемы.

КЛАССИЧЕСКИЕ МЕТОДЫ ПРОЕКТИРОВАНИЯ

В этой главе рассматриваются классические методы проектирования, ориентированные на процедурную реализацию программных систем (ПС). Повторим, что эти методы появились в период революции структурного программирования. Учитывая, что на современном этапе программной инженерии процедурно-ориентированные ПС имеют преимущественно историческое значение, конспективно обсуждаются только два (наиболее популярных) метода: метод структурного проектирования и метод проектирования Майкла Джексона (этот Джексон не имеет никакого отношения к известному певцу). Зачем мы это делаем? Да чтобы знать исторические корни современных методов проектирования.

Метод структурного проектирования

Исходными данными для метода структурного проектирования являются компоненты модели анализа ПС, которая

представляется иерархией диаграмм потоков данных [34], [52], [58], [73], [77]. Результат структурного проектирования — иерархическая структура ПС. Действия структурного проектирования зависят от типа информационного потока в модели анализа.

Типы информационных потоков

Различают 2 типа информационных потоков:

1) поток преобразований;

2) поток запросов.

Как показано на рис. 5.1, в потоке преобразований выделяют 3 элемента: Входящий поток, Преобразуемый поток и Выходящий поток.

Потоки запросов имеют в своем составе особые элементы — запросы.

Назначение элемента-запроса состоит в том, чтобы запустить поток данных по одному из нескольких путей. Анализ запроса и переключение потока данных на один из путей действий происходит в центре запросов.

Структуру потока запроса иллюстрирует рис. 5.2.

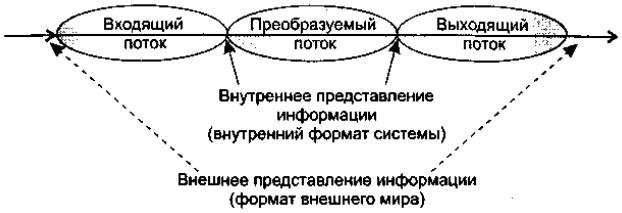

Рис. 5.1. Элементы потока преобразований

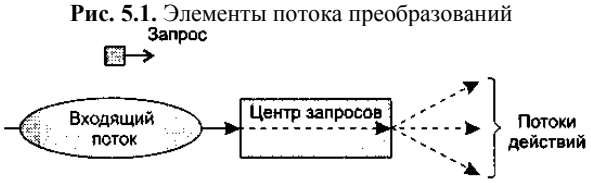

Рис. 5.2. Структура потока запроса

Проектирование для потока данных типа «преобразование»

Шаг 1. Проверка основной системной модели. Модель включает: контекстную диаграмму ПДД0, словарь данных и спецификации процессов. Оценивается их согласованность с системной спецификацией.

Шаг 2. Проверки и уточнения диаграмм потоков данных уровней 1 и 2. Оценивается согласованность диаграмм, достаточность детализации преобразователей.

Шаг 3. Определение типа основного потока диаграммы потоков данных. Основной признак потока преобразований — отсутствие переключения по путям действий.

Шаг 4. Определение границ входящего и выходящего потоков, отделение центра преобразований. Входящий поток — отрезок, на котором информация преобразуется из внешнего во внутренний формат представления. Выходящий поток обеспечивает обратное преобразование — из внутреннего формата во внешний. Границы входящего и выходящего потоков достаточно условны. Вариация одного преобразователя на границе слабо влияет на конечную структуру ПС.

Шаг 5. Определение начальной структуры ПС. Иерархическая структура ПС формируется нисходящим распространением управления. В иерархической структуре:

- модули верхнего уровня принимают решения;
- модули нижнего уровня выполняют работу по вводу, обработке и выводу;
- модули среднего уровня реализуют как функции управления, так и функции обработки.

Начальная структура ПС (для потока преобразования) стандартна и включает *главный контроллер* (находится на вершине структуры) и три подчиненных контроллера:

1. *Контроллер входящего потока* (контролирует получение входных данных).
2. *Контроллер преобразуемого потока* (управляет операциями над данными во внутреннем формате).
3. *Контроллер выходящего потока* (управляет получением выходных данных).

Данный минимальный набор модулей покрывает все функции управления, обеспечивает хорошую связность и слабое сцепление структуры.

Начальная структура ПС представлена на рис. 5.3.

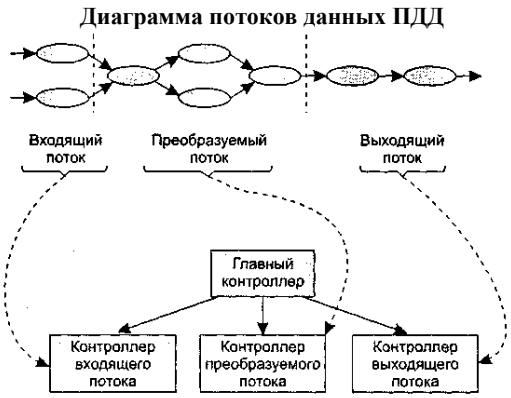

Рис. 5.3. Начальная структура ПС для потока «преобразование»

Шаг 6. Детализация структуры ПС. Выполняется отображение преобразователей ПДД в модули структуры ПС. Отображение выполняется движением по ПДД от границ центра преобразования вдоль входящего и выходящего потоков. Входящий поток проходится от конца к началу, а выходящий поток — от начала к концу. В ходе движения преобразователи отображаются в модули подчиненных уровней структуры (рис. 5.4).

Центр преобразования ПДД отображается иначе (рис. 5.5). Каждый преобразователь отображается в модуль, непосредственно подчиненный контроллеру центра.

Проходится преобразуемый поток слева направо.

Возможны следующие варианты отображения:

- 1 преобразователь отображается в 1 модуль;
- 2-3 преобразователя отображаются в 1 модуль;
- 1 преобразователь отображается в 2-3 модуля.

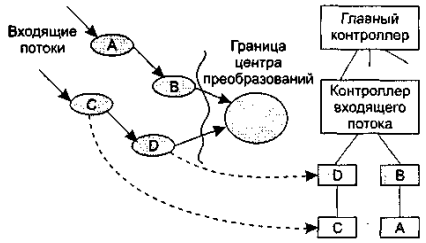

Рис. 5.4. Отображение преобразователей ПДД в модули структуры

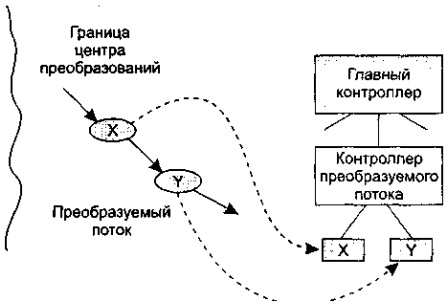

Рис. 5.5. Отображение центра преобразования ПДД

Для каждого модуля полученной структуры на базе спецификаций процессов модели анализа пишется сокращенное описание обработки.

Шаг 7. Уточнение иерархической структуры ПС. Модули разделяются и объединяются для:

1) повышения связности и уменьшения сцепления;
2) упрощения реализации;
3) упрощения тестирования;
4) повышения удобства сопровождения.

Проектирование для потока данных типа «запрос»

Шаг 1. Проверка основной системной модели. Модель включает: контекстную диаграмму ПДДО, словарь данных и спецификации процессов. Оценивается их согласованность с системной спецификацией.

Шаг 2. Проверки и уточнения диаграмм потоков данных уровней 1 и 2. Оценивается согласованность диаграмм, достаточность детализации преобразователей.

Шаг 3. Определение типа основного потока диаграммы потоков данных. Основной признак потоков запросов — явное переключение данных на один из путей действий.

Шаг 4. Определение центра запросов и типа для каждого из потоков действия. Если конкретный поток действия имеет тип

«преобразование», то для него указываются границы входящего, преобразуемого и выходящего потоков.

Шаг 5. Определение начальной структуры ПС. В начальную структуру отображается та часть диаграммы потоков данных, в которой распространяется поток запросов. Начальная структура ПС для потока запросов стандартна и включает входящую ветвь и диспетчерскую ветвь.

Структура входящей ветви формируется так же, как и в предыдущей методике.

Диспетчерская ветвь включает диспетчер, находящийся на вершине ветви, и контроллеры потоков действия, подчиненные диспетчеру; их должно быть столько, сколько имеется потоков действий.

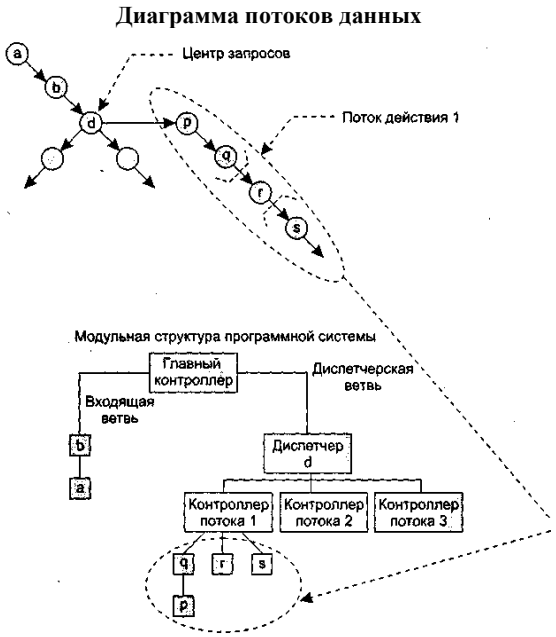

Рис. 5.6. Отображение в модульную структуру ПС потока действия 1

Шаг 6. Детализация структуры ПС. Производится отображение в структуру каждого потока действия. Каждый поток действия имеет свой тип. Могут встретиться поток-«преобразование» (отображается по предыдущей методике) и поток запросов. На рис. 5.6 приведен пример отображения потока действия 1. Подразумевается, что он является потоком преобразования.

Шаг 7. Уточнение иерархической структуры ПС. Уточнение выполняется для повышения качества системы. Как и при предыдущей методике, критериями уточнения служат: независимость модулей, эффективность реализации и тестирования, улучшение сопровождаемости.

СТРУКТУРНОЕ ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

В этой главе определяются общие понятия и принципы тестирования ПО (принцип «черного ящика» и принцип «белого ящика»). Читатель знакомится с содержанием процесса тестирования, после этого его внимание концентрируется на особенностях структурного тестирования программного обеспечения (по принципу «белого ящика»), описываются его достоинства и недостатки. Далее рассматриваются наиболее популярные способы структурного тестирования: тестирование базового пути, тестирование ветвей и операторов отношений, тестирование потоков данных, тестирование циклов.

Основные понятия и принципы тестирования 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.

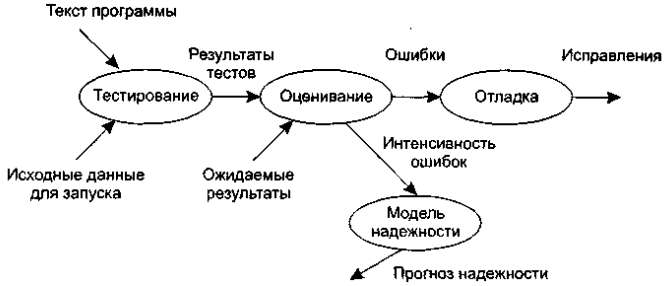

Рис. 6.1. Информационные потоки процесса тестирования

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»).

ФУНКЦИОНАЛЬНОЕ ТЕСТИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

В этой главе продолжается обсуждение вопросов тестирования ПО на уровне программных модулей. Впрочем, рассматриваемое здесь функциональное тестирование, основанное на принципе «черного ящика», может применяться и на уровне программной системы. После определения особенностей тестирования черного ящика в главе описываются популярные способы тестирования: разбиение по классам эквивалентности, анализ граничных значений, тестирование на основе диаграмм причин-следствий.

Особенности тестирования «черного ящика»

Тестирование «черного ящика» (функциональное тестирование) позволяет получить комбинации входных данных, обеспечивающих полную проверку всех функциональных требований к программе [14]. Программное изделие здесь рассматривается как «черный ящик», чье поведение можно определить только исследованием его входов и соответствующих выходов. При таком подходе желательно иметь:

- набор, образуемый такими входными данными, которые приводят к аномалиям поведения программы (назовем его *IT);*
- набор, образуемый такими выходными данными, которые демонстрируют дефекты программы (назовем его *OT).*

Как показано на рис. 7.1, любой способ тестирования «черного ящика» должен:

- выявить такие входные данные, которые с высокой вероятностью принадлежат набору IT;
- сформулировать такие ожидаемые результаты, которые с высокой вероятностью являются элементами набора OT.

Во многих случаях определение таких тестовых вариантов основывается на предыдущем опыте инженеров тестирования. Они используют свое знание и понимание области определения для идентификации тестовых вариантов, которые эффективно обнаруживают дефекты. Тем не менее систематический подход к выявлению тестовых данных, обсуждаемый в данной главе, может использоваться как полезное дополнение к эвристическому знанию.

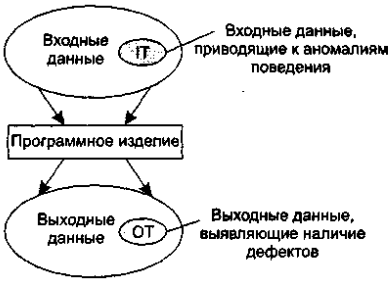

Рис. 7.1. Тестирование «черного ящика»

Принцип «черного ящика» не альтернативен принципу «белого ящика». Скорее это дополняющий подход, который обнаруживает другой класс ошибок.

Тестирование «черного ящика» обеспечивает поиск следующих категорий ошибок:

1) некорректных или отсутствующих функций;
2) ошибок интерфейса;
3) ошибок во внешних структурах данных или в доступе к внешней базе данных;
4) ошибок характеристик (необходимая емкость памяти и т. д.);
5) ошибок инициализации и завершения.

Подобные категории ошибок способами «белого ящика» не выявляются.

В отличие от тестирования «белого ящика», которое выполняется на ранней стадии процесса тестирования, тестирование «черного ящика» применяют на поздних стадиях тестирования. При тестировании «черного ящика» пренебрегают управляющей структурой программы. Здесь внимание концентрируется на информационной области определения программной системы.

Техника «черного ящика» ориентирована на решение следующих задач:

- сокращение необходимого количества тестовых вариантов (из-за проверки не статических, а динамических аспектов системы);
- выявление классов ошибок, а не отдельных ошибок.

Способ разбиения по эквивалентности

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

Например, если спецификация задает в качестве допустимых входных величин 5-разрядные целые числа в диапазоне 15 000...70 000, то класс эквивалентности допустимых ИД (исходных данных) включает величины от 15 000 до 70 000, а два класса эквивалентности недопустимых ИД составляют:

- числа меньшие, чем 15 000;
- числа большие, чем 70 000.

Класс эквивалентности включает множество значений данных, допустимых или недопустимых по условиям ввода.

Условие ввода может задавать:

1) определенное значение;
2) диапазон значений;
3) множество конкретных величин;
4) булево условие.

Сформулируем *правила формирования классов эквивалентности.*

1. Если условие ввода задает диапазон *n...m,* то определяются один допустимый и два недопустимых класса эквивалентности:
- V_Class={*n*.. .*m*} — допустимый класс эквивалентности;
- Inv_C1ass1={*x*|для любого *x: x* < *n*} — первый недопустимый класс эквивалентности;
- Inv_C1ass2={*y*|для любого *y: y* > *m*} — второй недопустимый класс эквивалентности.

2. Если условие ввода задает конкретное значение *a,* то определяется один допустимый и два недопустимых класса эквивалентности:
- V_Class={*a*};
- Inv_Class1 ={*x*|для любого *x: x* < *a*};
- Inv_C1ass2={*y*|для любого *y: y* > *a*}.

3. Если условие ввода задает множество значений *{a, b, c},* то определяются один допустимый и один недопустимый класс эквивалентности:
- V_Class={*a, b, c*};
- Inv_C1ass={*x*|для любого *x: (x* ≠ *a)&(x* ≠ *b)&(x* ≠ *c)}*.

4. Если условие ввода задает булево значение, например true, то определяются один допустимый и один недопустимый класс эквивалентности:
- V_Class={true};
- Inv_Class={false}.

После построения классов эквивалентности разрабатываются тестовые варианты. Тестовый вариант выбирается так, чтобы проверить сразу наибольшее количество свойств класса эквивалентности.

Способ 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.

Большинство разработчиков используют этот способ интуитивно. При применении описанных правил тестирование границ будет более полным, в связи с чем возрастет вероятность обнаружения ошибок.

Рассмотрим применение способов разбиения по эквивалентности и анализа граничных значений на конкретном примере. Положим, что нужно протестировать программу бинарного поиска. Нам известна *спецификация* этой программы. Поиск выполняется в массиве элементов *M,* возвращается индекс *I* элемента массива, значение которого соответствует ключу поиска *Key.*

Предусловия:

1) массив должен быть упорядочен;

2) массив должен иметь не менее одного элемента;

3) нижняя граница массива (индекс) должна быть меньше или равна его верхней границе.

Постусловия:

1) если элемент найден, то флаг Result=True, значение I — номер элемента;

2) если элемент не найден, то флаг Result=False, значение I не определено.

Для формирования классов эквивалентности (и их ребер) надо произвести разбиение области ИД — построить дерево разбиений. Листья дерева разбиений дадут нам искомые классы эквивалентности. Определим стратегию разбиения. На первом уровне будем анализировать выполнимость предусловий, на втором уровне — выполнимость постусловий. На третьем уровне можно анализировать специальные требования, полученные из практики разработчика. В нашем примере мы знаем, что входной массив должен быть упорядочен. Обработка упорядоченных наборов из четного и нечетного количества элементов может выполняться по-разному. Кроме того, принято выделять специальный случай одноэлементного массива. Следовательно, на уровне специальных требований возможны следующие эквивалентные разбиения:

1) массив из одного элемента;

2) массив из четного количества элементов;

3) массив из нечетного количества элементов, большего единицы.

Наконец на последнем, 4-м уровне критерием разбиения может быть анализ ребер классов эквивалентности. Очевидно, возможны следующие варианты:

1) работа с первым элементом массива;

2) работа с последним элементом массива;

3) работа с промежуточным (ни с первым, ни с последним) элементом массива.

Структура дерева разбиений приведена на рис. 7.3.

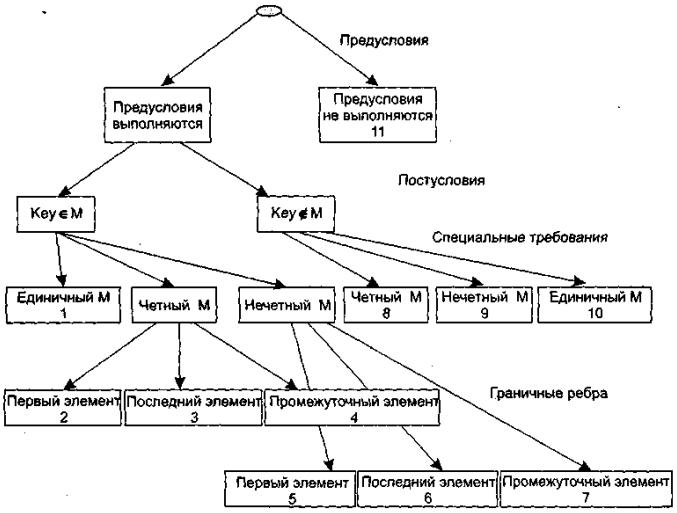

Рис. 7.3. Дерево разбиений области исходных данных бинарного поиска

Это дерево имеет 11 листьев. Каждый лист задает отдельный тестовый вариант. Покажем тестовые варианты, основанные на проведенных разбиениях.

Тестовый вариант 1 (единичный массив, элемент найден) **ТВ1:**
ИД: M=15; Key=15.
ОЖ.РЕЗ.: Resutt=True; I=1.
Тестовый вариант 2 (четный массив, найден 1-й элемент) **ТВ2:**
ИД: M=15, 20, 25,30,35,40; Key=15.
ОЖ.РЕЗ.: Result=True; I=1.
Тестовый вариант 3 (четный массив, найден последний элемент) **ТВ3:**
ИД: M=15, 20, 25, 30, 35, 40; Key=40.
ОЖ.РЕЗ:. Result=True; I=6.
Тестовый вариант 4 (четный массив, найден промежуточный элемент) **ТВ4:**
ИД: M=15,20,25,30,35,40; Key=25.
ОЖ.РЕЗ.: Result-True; I=3.
Тестовый вариант 5 (нечетный массив, найден 1-й элемент) **ТВ5:**
ИД: M=15, 20, 25, 30, 35,40, 45; Key=15.
ОЖ.РЕЗ.: Result=True; I=1.
Тестовый вариант 6 (нечетный массив, найден последний элемент) **ТВ6:**
ИД: M=15, 20, 25, 30,35, 40,45; Key=45.
ОЖ.РЕЗ.: Result=True; I=7.
Тестовый вариант 7 (нечетный массив, найден промежуточный элемент) **ТВ7:**
ИД: M=15, 20, 25, 30,35, 40, 45; Key=30.
ОЖ.РЕЗ.: Result=True; I=4.
Тестовый вариант 8 (четный массив, не найден элемент) **ТВ8:**
ИД: M=15, 20, 25, 30, 35,40; Key=23.
ОЖ.РЕЗ.: Result=False; I=?
Тестовый вариант 9 (нечетный массив, не найден элемент) **ТВ9;**
ИД: M=15, 20, 25, 30, 35, 40, 45; Key=24.
ОЖ.РЕЗ:. Result=False; I=?
Тестовый вариант 10 (единичный массив, не найден элемент) **ТВ10:**
ИД: M=15; Key=0.
ОЖ.РЕЗ.: Result=False; I=?
Тестовый вариант 11 (нарушены предусловия) **ТВ11:**
ИД: M=15, 10, 5, 25, 20, 40, 35; Key=35.
ОЖ.РЕЗ.: Аварийное донесение: Массив не упорядочен.

Способ диаграмм причин-следствий

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.

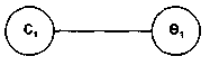

Рис. 7.4. Функция *тождество*

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.

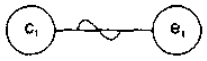

Рис. 7.5. Функция *не*

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.

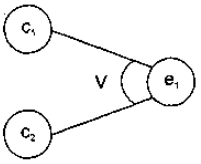

Рис. 7.6. Функция *или*

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.

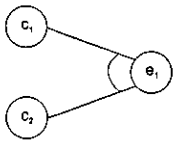

Рис. 7.7. Функция *и*

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).

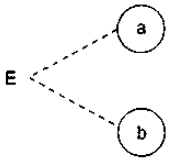

Рис. 7.8. Ограничение E (исключает, Exclusive)

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).

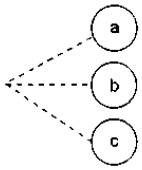

Рис. 7.9. Ограничение I (включает, Inclusive)

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.

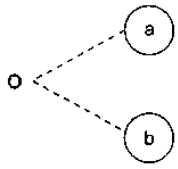

Рис. 7.10. Ограничение O (одно и только одно, Only one)

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).

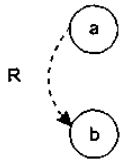

Рис. 7.11. Ограничение R (требует, Requires)

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.

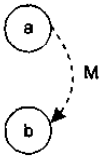

Рис. 7.12. Ограничение M (скрывает, Masks)

Для иллюстрации использования способа рассмотрим пример, когда программа выполняет расчет оплаты за электричество по среднему или переменному тарифу.

При расчете по среднему тарифу:

- при месячном потреблении энергии меньшем, чем 100 кВт/ч, выставляется фиксированная сумма;
- при потреблении энергии большем или равном 100 кВт/ч применяется процедура *A* планирования расчета.

При расчете по переменному тарифу:

- при месячном потреблении энергии меньшем, чем 100 кВт/ч, применяется процедура *A* планирования расчета;
- при потреблении энергии большем или равном 100 кВт/ч применяется процедура *B* планирования расчета.

Шаг 1. Причинами являются:

1) расчет по среднему тарифу;
2) расчет по переменному тарифу;
3) месячное потребление электроэнергии меньшее, чем 100 кВт/ч;
4) месячное потребление электроэнергии большее или равное 100 кВт/ч.

На основе различных комбинаций причин можно перечислить следующие следствия:

- 101 — минимальная месячная стоимость;
- 102 — процедура *A* планирования расчета;
- 103 — процедура *B* планирования расчета.

Шаг 2. Разработка графа причинно-следственных связей (рис. 7.13).

Узлы причин перечислим по вертикали у левого края рисунка, а узлы следствий — у правого края рисунка. Для следствия 102 возникает необходимость введения вторичных причин — 11 и 12, — их размещаем в центральной части рисунка.

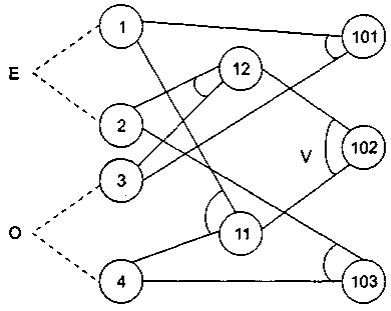

Рис. 7.13. Граф причинно-следственных связей

Шаг 3. Генерация таблицы решений. При генерации причины рассматриваются как условия, а следствия — как действия.

Порядок генерации.

1. Выбирается некоторое следствие, которое должно быть в состоянии «1».
2. Находятся все комбинации причин (с учетом ограничений), которые устанавливают это следствие в состояние «1». Для этого из следствия прокладывается обратная трасса через граф.
3. Для каждой комбинации причин, приводящих следствие в состояние «1», строится один столбец.

4. Для каждой комбинации причин доопределяются состояния всех других следствий. Они помещаются в тот же столбец таблицы решений.
5. Действия 1-4 повторяются для всех следствий графа.

Таблица решений для нашего примера показана в табл. 7.1.

Шаг 4. Преобразование каждого столбца таблицы в тестовый вариант. В нашем примере таких вариантов четыре.

Тестовый вариант 1 (столбец 1) **ТВ1:**
ИД: расчет по среднему тарифу; месячное потребление электроэнергии 75 кВт/ч.
ОЖ.РЕЗ.: минимальная месячная стоимость.
Тестовый вариант 2 (столбец 2) **ТВ2:**
ИД: расчет по переменному тарифу; месячное потребление электроэнергии 90 кВт/ч.
ОЖ.РЕЗ.: процедура *А* планирования расчета.
Тестовый вариант 3 (столбец 3) **ТВ3:**
ИД: расчет по среднему тарифу; месячное потребление электроэнергии 100 кВт/ч.
ОЖ.РЕЗ.: процедура *А* планирования расчета.
Тестовый вариант 4 (столбец 4) **ТВ4:**
ИД: расчет по переменному тарифу; месячное потребление электроэнергии 100 кВт/ч.
ОЖ.РЕЗ.: процедура *В* планирования расчета.

Таблица 7.1. Таблица решений для расчета оплаты за электричество

Номера столбцов — >			1	2	3	4
Условия	Причины	1	1	0	1	0
		2	0	1	0	1
		3	1	1	0	0
		4	0	0	1	1
	Вторичные причины	11	0	0	1	0
		12	0	1	0	0
Действия	Следствия	101	1	0	0	0
		102	0	1	1	0
		103	0	0	0	1

ОРГАНИЗАЦИЯ ПРОЦЕССА ТЕСТИРОВАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

В этой главе излагаются вопросы, связанные с проведением тестирования на всех этапах конструирования программной системы. Классический процесс тестирования обеспечивает проверку результатов, полученных на каждом этапе разработки. Как правило, он начинается с тестирования в малом, когда проверяются программные модули, продолжается при проверке объединения модулей в систему и завершается тестированием в большом, при котором проверяются соответствие программного продукта требованиям заказчика и его взаимодействие с другими компонентами компьютерной системы. Данная глава последовательно описывает содержание каждого шага тестирования. Здесь же рассматривается организация отладки ПО, которая проводится для устранения выявленных при тестировании ошибок.

Методика тестирования программных систем

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:

$$\lambda(t) = \frac{\lambda_0}{\lambda_0 \times p \times t + 1}, \qquad (8.$$

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.

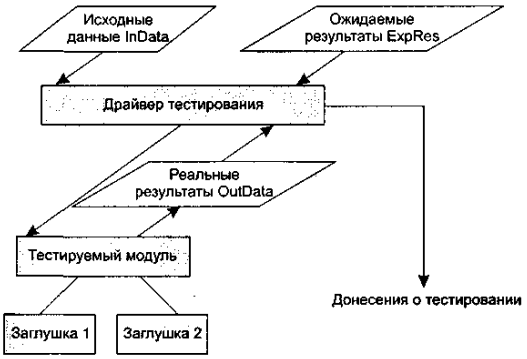

Рис. 8.2. Программная среда для тестирования модуля

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.

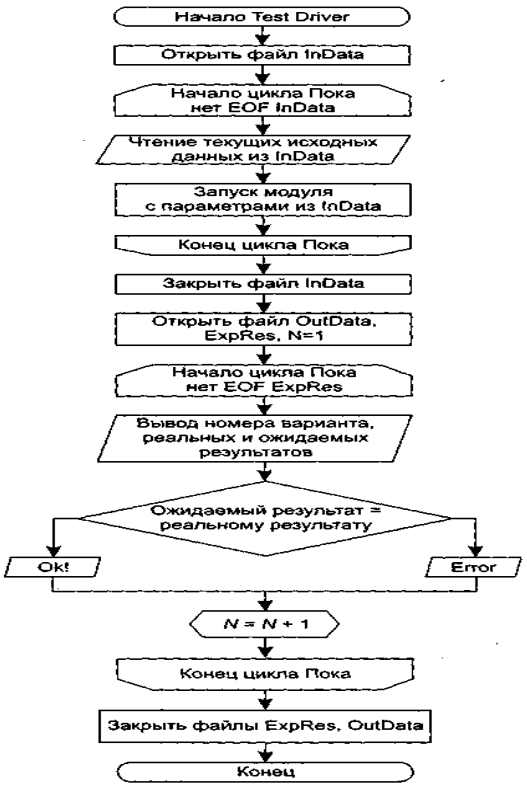

Рис. 8.3. Алгоритм работы драйвера тестирования

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.

Тестирование интеграции

Тестирование интеграции поддерживает сборку цельной программной системы.

Цель сборки и тестирования интеграции: взять модули, протестированные как элементы, и построить программную структуру, требуемую проектом [3].

Тесты проводятся для обнаружения ошибок интерфейса. Перечислим некоторые категории ошибок интерфейса:

- потеря данных при прохождении через интерфейс;
- отсутствие в модуле необходимой ссылки;
- неблагоприятное влияние одного модуля на другой;
- подфункции при объединении не образуют требуемую главную функцию;
- отдельные (допустимые) неточности при интеграции выходят за допустимый уровень;
- проблемы при работе с глобальными структурами данных.

Существует два варианта тестирования, поддерживающих процесс интеграции: нисходящее тестирование и восходящее тестирование. Рассмотрим каждый из них.

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, М2, М5. Следующим подключается модуль М8 или Мб (если это необходимо для правильного функционирования М2). 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 М2, М3, М4. На следующем уровне — модули М5, Мб и т. д.

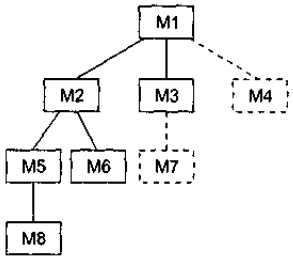

Рис. 8.4. Нисходящая интеграция системы

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 B — отображает проходящий параметр;
- заглушка C — возвращает величину из таблицы;
- заглушка D — выполняет табличный поиск по ключу (входному параметру) и возвращает связанный с ним выходной параметр.

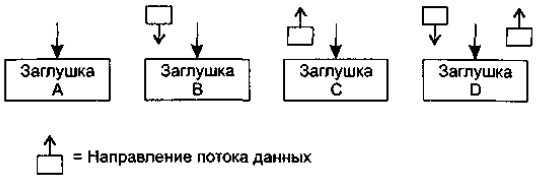

Рис. 8.5. Категории заглушек

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.

Третью возможность обсудим отдельно.

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.

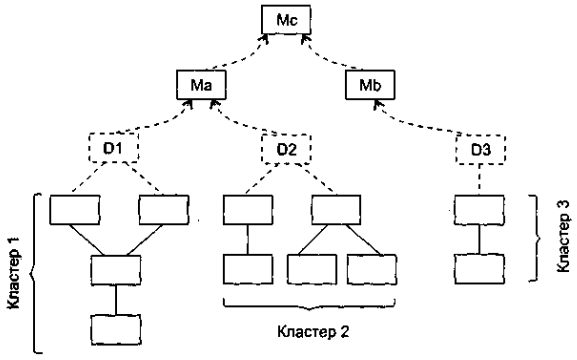

Рис. 8.6. Восходящая интеграция системы

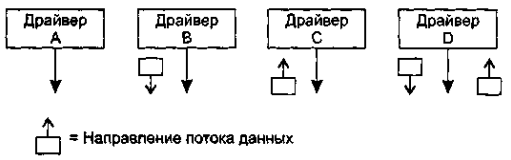

Рис. 8.7. Различные типы драйверов

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

Сравнение нисходящего и восходящего тестирования интеграции

Нисходящее тестирование:

1) *основной недостаток*— необходимость заглушек и связанные с ними трудности тестирования;

2) *основное достоинство* — возможность раннего тестирования главных управляющих функций.

Восходящее тестирование:

1) *основной недостаток* — система не существует как объект до тех пор, пока не будет добавлен последний модуль;

2) *основное достоинство* — упрощается разработка тестовых вариантов, отсутствуют заглушки.

Возможен комбинированный подход. В нем для верхних уровней иерархии применяют нисходящую стратегию, а для нижних уровней — восходящую стратегию тестирования [3], [13].

При проведении тестирования интеграции очень важно выявить критические модули. Признаки критического модуля:

1) реализует несколько требований к программной системе;
2) имеет высокий уровень управления (находится достаточно высоко в программной структуре);
3) имеет высокую сложность или склонность к ошибкам (как индикатор может использоваться цикломатическая сложность — ее верхний разумный предел составляет 10);
4) имеет определенные требования к производительности обработки.

Критические модули должны тестироваться как можно раньше. Кроме того, к ним должно применяться регрессионное тестирование (повторение уже выполненных тестов в полном или частичном объеме).

Тестирование правильности

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.

Системное тестирование

Системное тестирование подразумевает выход за рамки области действия программного проекта и проводится не только программным разработчиком. Классическая проблема системного тестирования — указание причины. Она возникает, когда разработчик одного системного элемента обвиняет разработчика другого элемента в причине возникновения дефекта. Для защиты от подобного обвинения разработчик программного элемента должен:

1) предусмотреть средства обработки ошибки, которые тестируют все вводы информации от других элементов системы;
2) провести тесты, моделирующие неудачные данные или другие потенциальные ошибки интерфейса ПС;
3) записать результаты тестов, чтобы использовать их как доказательство невиновности в случае «указания причины»;
4) принять участие в планировании и проектировании системных тестов, чтобы гарантировать адекватное тестирование

ПС.

В конечном счете системные тесты должны проверять, что все системные элементы правильно объединены и выполняют назначенные функции. Рассмотрим основные типы системных тестов [13], [52].

Тестирование восстановления

Многие компьютерные системы должны восстанавливаться после отказов и возобновлять обработку в пределах заданного времени. В некоторых случаях система должна быть отказоустойчивой, то есть отказы обработки не должны быть причиной прекращения работы системы. В других случаях системный отказ должен быть устранен в пределах заданного кванта времени, иначе заказчику наносится серьезный экономический ущерб.

Тестирование восстановления использует самые разные пути для того, чтобы заставить ПС отказать, и проверяет полноту выполненного восстановления. При автоматическом восстановлении оцениваются правильность повторной инициализации, механизмы копирования контрольных точек, восстановление данных, перезапуск. При ручном восстановлении оценивается, находится ли среднее время восстановления в допустимых пределах.

Тестирование безопасности

Компьютерные системы очень часто являются мишенью незаконного проникновения. Под проникновением понимается широкий диапазон действий: попытки хакеров проникнуть в систему из спортивного интереса, месть рассерженных служащих, взлом мошенниками для незаконной наживы.

Тестирование безопасности проверяет фактическую реакцию защитных механизмов, встроенных в систему, на проникновение.

В ходе тестирования безопасности испытатель играет роль взломщика. Ему разрешено все:

- попытки узнать пароль с помощью внешних средств;
- атака системы с помощью специальных утилит, анализирующих защиты;
- подавление, ошеломление системы (в надежде, что она откажется обслуживать других клиентов);
- целенаправленное введение ошибок в надежде проникнуть в систему в ходе восстановления;
- просмотр несекретных данных в надежде найти ключ для входа в систему.

Конечно, при неограниченном времени и ресурсах хорошее тестирование безопасности взломает любую систему. Задача проектировщика системы — сделать цену проникновения более высокой, чем цена получаемой в результате информации.

Стрессовое тестирование

На предыдущих шагах тестирования способы «белого» и «черного ящиков» обеспечивали полную оценку нормальных программных функций и качества функционирования. Стрессовые тесты проектируются для навязывания программам ненормальных ситуаций. В сущности, проектировщик стрессового теста спрашивает, как сильно можно расшатать систему, прежде чем она откажет?

Стрессовое тестирование производится при ненормальных запросах на ресурсы системы (по количеству, частоте, размеру-объему).

Примеры:

- генерируется 10 прерываний в секунду (при средней частоте 1,2 прерывания в секунду);
- скорость ввода данных увеличивается прямо пропорционально их важности (чтобы определить реакцию входных функций);
- формируются варианты, требующие максимума памяти и других ресурсов;
- генерируются варианты, вызывающие переполнение виртуальной памяти;
- проектируются варианты, вызывающие чрезмерный поиск данных на диске.

По существу, испытатель пытается разрушить систему. Разновидность стрессового тестирования называется *тестированием чувствительности.* В некоторых ситуациях (обычно в математических алгоритмах) очень малый диапазон данных, содержащийся в границах правильных данных системы, может вызвать ошибочную обработку или резкое понижение производительности. Тестирование чувствительности обнаруживает комбинации данных, которые могут вызвать нестабильность или неправильность обработки.

Тестирование производительности

В системах реального времени и встроенных системах недопустимо ПО, которое реализует требуемые функции, но не соответствует требованиям производительности.

Тестирование производительности проверяет скорость работы ПО в компьютерной системе. Производительность тестируется на всех шагах процесса тестирования. Даже на уровне элемента при проведении тестов «белого ящика» может оцениваться производительность индивидуального модуля. Тем не менее, пока все системные элементы не объединятся полностью, не может быть установлена истинная производительность системы. Иногда тестирование производительности сочетают со стрессовым тестированием. При этом нередко требуется специальный аппаратный и программный инструментарий. Например, часто требуется точное измерение используемого ресурса (процессорного цикла и т. д.). Внешний инструментарий регулярно отслеживает интервалы выполнения, регистрирует события (например, прерывания) и машинные состояния. С помощью инструментария испытатель может обнаружить состояния, которые приводят к деградации и

возможным отказам системы.

БАЗИС ЯЗЫКА ВИЗУАЛЬНОГО МОДЕЛИРОВАНИЯ

Для создания моделей анализа и проектирования объектно-ориентированных программных систем используют языки визуального моделирования. Появившись сравнительно недавно, в период с 1989 по 1997 год, эти языки уже имеют представительную историю развития.

В настоящее время различают три поколения языков визуального моделирования. И если первое поколение образовали 10 языков, то численность второго поколения уже превысила 50 языков. Среди наиболее популярных языков 2-го поколения можно выделить: язык Буча (G. Booch), язык Рамбо (J. Rumbaugh), язык Джекобсона (I. Jacobson), язык Коада-Йордона (Coad-Yourdon), язык Шлеера-Меллора (Shlaer-Mellor) и т. д [41], [64], [69]. Каждый язык вводил свои выразительные средства, ориентировался на собственный синтаксис и семантику, иными словами — претендовал на роль единственного и неповторимого языка. В результате разработчики (и пользователи этих языков) перестали понимать друг друга. Возникла острая необходимость унификации языков.

Идея унификации привела к появлению языков 3-го поколения. В качестве стандартного языка третьего поколения был принят Unified Modeling Language (UML), создававшийся в 1994-1997 годах (основные разработчики — три «amigos» Г. Буч, Дж. Рамбо, И. Джекобсон). В настоящее время разработана версия UML 1.4, которая описывается в данном учебнике [53]. Данная глава посвящена определению базовых понятий языка UML.

Унифицированный язык моделирования

UML — стандартный язык для написания моделей анализа, проектирования и реализации объектно-ориентированных программных систем [23], [53], [67]. UML может использоваться для визуализации, спецификации, конструирования и документирования результатов программных проектов. UML — это не визуальный язык программирования, но его модели прямо транслируются в текст на языках программирования (Java, C++, Visual Basic, Ada 95, Object Pascal) и даже в таблицы для реляционной БД.

Словарь UML образуют три разновидности строительных блоков: предметы, отношения, диаграммы.

Предметы — это абстракции, которые являются основными элементами в модели, отношения связывают эти предметы, диаграммы группируют коллекции предметов.

Предметы в UML

В UML имеются четыре разновидности предметов:

- структурные предметы;
- предметы поведения;
- группирующие предметы;
- поясняющие предметы.

Эти предметы являются базовыми объектно-ориентированными строительными блоками. Они используются для написания моделей.

Структурные предметы являются существительными в UML-моделях. Они представляют статические части модели — понятийные или физические элементы. Перечислим восемь разновидностей структурных предметов.

1. *Класс* — описание множества объектов, которые разделяют одинаковые свойства, операции, отношения и семантику (смысл). Класс реализует один или несколько интерфейсов. Как показано на рис. 10.1, графически класс отображается в виде прямоугольника, обычно включающего секции с именем, свойствами (атрибутами) и операциями.

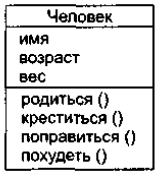

Рис. 10.1. Классы

2. *Интерфейс* — набор операций, которые определяют услуги класса или компонента. Интерфейс описывает поведение элемента, видимое извне. Интерфейс может представлять полные услуги класса или компонента или часть таких услуг. Интерфейс определяет набор спецификаций операций (их сигнатуры), а не набор реализаций операций. Графически интерфейс изображается в виде кружка с именем, как показано на рис. 10.2. Имя интерфейса обычно начинается с буквы «I». Интерфейс редко показывают самостоятельно. Обычно его присоединяют к классу или компоненту, который реализует интерфейс.
3. *Кооперация* (сотрудничество) определяет взаимодействие и является совокупностью ролей и других элементов, которые работают вместе для обеспечения коллективного поведения более сложного, чем простая сумма всех элементов. Таким образом, кооперации имеют как структурное, так и поведенческое измерения. Конкретный класс может участвовать в нескольких кооперациях. Эти кооперации представляют реализацию паттернов (образцов), которые формируют систему. Как показано на рис. 10.3, графически кооперация изображается как пунктирный эллипс, в который вписывается ее имя.

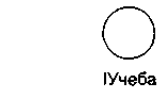

Рис. 10.2. Интерфейсы

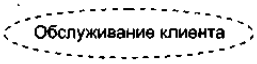

Рис. 10.3. Кооперации

4. *Актер* — набор согласованных ролей, которые могут играть пользователи при взаимодействии с системой (ее элементами Use Case). Каждая роль требует от системы определенного поведения. Как показано на рис. 10.4, актер изображается как проволочный человечек с именем.

Рис. 10.4. Актеры

5. *Элемент Use Case* (Прецедент) — описание последовательности действий (или нескольких последовательностей), выполняемых системой в интересах отдельного актера и производящих видимый для актера результат. В модели элемент Use Case применяется для структурирования предметов поведения. Элемент Use Case реализуется кооперацией. Как показано на рис. 10.5, элемент Use Case изображается как эллипс, в который вписывается его имя.

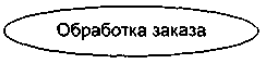

Рис. 10.5. Элементы Use Case

6. *Активный класс* — класс, чьи объекты имеют один или несколько процессов (или потоков) и поэтому могут инициировать управляющую деятельность. Активный класс похож на обычный класс за исключением того, что его объекты действуют одновременно с объектами других классов. Как показано на рис. 10.6, активный класс изображается как утолщенный прямоугольник, обычно включающий имя, свойства (атрибуты) и операции.

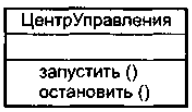

Рис. 10.6. Активные классы

7. *Компонент* — физическая и заменяемая часть системы, которая соответствует набору интерфейсов и обеспечивает реализацию этого набора интерфейсов. В систему включаются как компоненты, являющиеся результатами процесса разработки (файлы исходного кода), так и различные разновидности используемых компонентов (COM+-компоненты, Java Beans). Обычно компонент — это физическая упаковка различных логических элементов (классов, интерфейсов и сотрудничеств). Как показано на рис. 10.7, компонент изображается как прямоугольник с вкладками, обычно включающий имя.

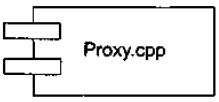

Рис. 10.7. Компоненты

8. *Узел* — физический элемент, который существует в период работы системы и представляет ресурс, обычно имеющий память и возможности обработки. В узле размещается набор компонентов, который может перемещаться от узла к узлу. Как показано на рис. 10.8, узел изображается как куб с именем.

Рис. 10.8. Узлы

Предметы поведения — динамические части UML-моделей. Они являются глаголами моделей, представлением поведения во времени и пространстве. Существует две основные разновидности предметов поведения.

1. *Взаимодействие* — поведение, заключающее в себе набор сообщений, которыми обменивается набор объектов в конкретном контексте для достижения определенной цели. Взаимодействие может определять динамику как совокупности объектов, так и отдельной операции. Элементами взаимодействия являются сообщения, последовательность действий (поведение, вызываемое сообщением) и связи (соединения между объектами). Как показано на рис. 10.9, сообщение изображается в виде направленной линии с именем ее операции.

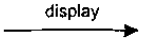

Рис. 10.9. Сообщения

2. *Конечный автомат* — поведение, которое определяет последовательность состояний объекта или взаимодействия, выполняемые в ходе его существования в ответ на события (и с учетом обязанностей по этим событиям). С помощью конечного автомата может определяться поведение индивидуального класса или кооперации классов. Элементами конечного автомата являются состояния, переходы (от состояния к состоянию), события (предметы, вызывающие переходы) и действия (реакции на переход). Как показано на рис. 10.10, состояние изображается как закругленный прямоугольник, обычно включающий его имя и его подсостояния (если они есть).

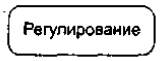

Рис. 10.10. Состояния

Эти два элемента — взаимодействия и конечные автоматы — являются базисными предметами поведения, которые могут включаться в UML-модели. Семантически эти элементы ассоциируются с различными структурными элементами (прежде всего с классами, сотрудничествами и объектами).

Группирующие предметы — организационные части UML-моделей. Это ящики, по которым может быть разложена модель. Предусмотрена одна разновидность группирующего предмета — пакет.

Пакет — общий механизм для распределения элементов по группам. В пакет могут помещаться структурные предметы, предметы поведения и даже другие группировки предметов. В отличие от компонента (который существует в период выполнения), пакет — чисто концептуальное понятие. Это означает, что пакет существует только в период разработки. Как показано на рис. 10.11, пакет изображается как папка с закладкой, на которой обозначено его имя и, иногда, его содержание.

Рис. 10.11. Пакеты

Поясняющие предметы — разъясняющие части UML-моделей. Они являются замечаниями, которые можно применить для описания, объяснения и комментирования любого элемента модели. Предусмотрена одна разновидность поясняющего предмета — примечание.

Примечание — символ для отображения ограничений и замечаний, присоединяемых к элементу или совокупности элементов. Как показано на рис. 10.12, примечание изображается в виде прямоугольника с загнутым углом, в который вписывается текстовый или графический комментарий.

Рис. 10.12. Примечания

Отношения в UML

В UML имеются четыре разновидности отношений:

1) зависимость;
2) ассоциация;
3) обобщение;
4) реализация.

Эти отношения являются базовыми строительными блоками отношений. Они используются при написании моделей.

1. *Зависимость* — семантическое отношение между двумя предметами, в котором изменение в одном предмете (независимом предмете) может влиять на семантику другого предмета (зависимого предмета). Как показано на рис. 10.13, зависимость изображается в виде пунктирной линии, возможно направленной на независимый предмет и иногда имеющей метку.

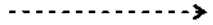

Рис. 10.13. Зависимости

2. *Ассоциация* — структурное отношение, которое описывает набор связей, являющихся соединением между объектами. Агрегация — это специальная разновидность ассоциации, представляющая структурное отношение между целым и его частями. Как показано на рис. 10.14, ассоциация изображается в виде сплошной линии, возможно направленной, иногда имеющей метку и часто включающей другие «украшения», такие как мощность и имена ролей.

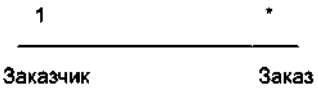

Рис. 10.14. Ассоциации

3. *Обобщение* — отношение специализации/обобщения, в котором объекты специализированного элемента (потомка, ребенка) могут заменять объекты обобщенного элемента (предка, родителя). Иначе говоря, потомок разделяет структуру и поведение родителя. Как показано на рис. 10.15, обобщение изображается в виде сплошной стрелки с полым наконечником, указывающим на родителя.

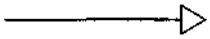

Рис. 10.15. Обобщения

4. *Реализация* — семантическое отношение между классификаторами, где один классификатор определяет контракт, который другой классификатор обязуется выполнять (к классификаторам относят классы, интерфейсы, компоненты, элементы Use Case, кооперации). Отношения реализации применяют в двух случаях: между интерфейсами и классами (или компонентами), реализующими их; между элементами Use Case и кооперациями, которые реализуют их. Как показано на рис. 10.16, реализация изображается как нечто среднее между обобщением и зависимостью.

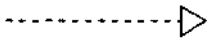

Рис. 10.16. Реализации

Диаграммы в UML

Диаграмма — графическое представление множества элементов, наиболее часто изображается как связный граф из вершин (предметов) и дуг (отношений). Диаграммы рисуются для визуализации системы с разных точек зрения, затем они отображаются в систему. Обычно диаграмма дает неполное представление элементов, которые составляют систему. Хотя один и тот же элемент может появляться во всех диаграммах, на практике он появляется только в некоторых диаграммах. Теоретически диаграмма может содержать любую комбинацию предметов и отношений, на практике ограничиваются малым количеством комбинаций, которые соответствуют пяти представлениям архитектуры ПС. По этой причине UML включает девять видов диаграмм:

1) диаграммы классов;
2) диаграммы объектов;
3) диаграммы Use Case (диаграммы прецедентов);
4) диаграммы последовательности;
5) диаграммы сотрудничества (кооперации);
6) диаграммы схем состояний;
7) диаграммы деятельности;
8) компонентные диаграммы;
9) диаграммы размещения (развертывания).

Диаграмма классов показывает набор классов, интерфейсов, сотрудничеств и их отношений. При моделировании объектно-ориентированных систем диаграммы классов используются наиболее часто. Диаграммы классов обеспечивают статическое проектное представление системы. Диаграммы классов, включающие активные классы, обеспечивают статическое представление процессов системы.

Диаграмма объектов показывает набор объектов и их отношения. Диаграмма объектов представляет статический «моментальный снимок» с экземпляров предметов, которые находятся в диаграммах классов. Как и диаграммы классов, эти диаграммы обеспечивают статическое проектное представление или статическое представление процессов системы (но с точки зрения реальных или фототипичных случаев).

Диаграмма Use Case (диаграмма прецедентов) показывает набор элементов Use Case, актеров и их отношений. С помощью диаграмм Use Case для системы создается статическое представление Use Case. Эти диаграммы особенно важны при организации и моделировании поведения системы, задании требований заказчика к системе.

Диаграммы последовательности и диаграммы сотрудничества — это разновидности диаграмм взаимодействия.

Диаграмма взаимодействия показывает взаимодействие, включающее набор объектов и их отношений, а также пересылаемые между объектами сообщения. Диаграммы взаимодействия обеспечивают динамическое представление системы.

Диаграмма последовательности — это диаграмма взаимодействия, которая выделяет упорядочение сообщений по времени.

Диаграмма сотрудничества (диаграмма кооперации) — это диаграмма взаимодействия, которая выделяет структурную организацию объектов, посылающих и принимающих сообщения. Диаграммы последовательности и диаграммы сотрудничества изоморфны, что означает, что одну диаграмму можно трансформировать в другую диаграмму.

Диаграмма схем состояний показывает конечный автомат, представляет состояния, переходы, события и действия. Диаграммы схем состояний обеспечивают динамическое представление системы. Они особенно важны при моделировании поведения интерфейса, класса или сотрудничества. Эти диаграммы выделяют такое поведение объекта, которое управляется событиями, что особенно полезно при моделировании реактивных систем.

Диаграмма деятельности — специальная разновидность диаграммы схем состояний, которая показывает поток от действия к действию внутри системы. Диаграммы деятельности обеспечивают динамическое представление системы. Они особенно важны при моделировании функциональности системы и выделяют поток управления между объектами.

Компонентная диаграмма показывает организацию набора компонентов и зависимости между компонентами. Компонентные диаграммы обеспечивают статическое представление реализации системы. Они связаны с диаграммами классов в том смысле, что в компонент обычно отображается один или несколько классов, интерфейсов или кооперций.

Диаграмма размещения (диаграмма развертывания) показывает конфигурацию обрабатывающих узлов периода выполнения, а также компоненты, живущие в них. Диаграммы размещения обеспечивают статическое представление размещения системы. Они связаны с компонентными диаграммами в том смысле, что узел обычно включает один или несколько компонентов.

Механизмы расширения в UML

UML — развитый язык, имеющий большие возможности, но даже он не может отразить все нюансы, которые могут возникнуть при создании различных моделей. Поэтому UML создавался как открытый язык, допускающий контролируемые

рас-. ширения. Механизмами расширения в UML являются:

- ограничения;
- теговые величины;
- стереотипы.

Ограничение (constraint) расширяет семантику строительного UML-блока, позволяя добавить новые правила или модифицировать существующие. Ограничение показывают как текстовую строку, заключенную в фигурные скобки {}. Например, на рис. 10.17 введено простое ограничение на свойство *сумма* класса *Сессия Банкомата* — его значение должно быть кратно 20. Кроме того, здесь показано ограничение на два элемента (две ассоциации), оно располагается возле пунктирной линии, соединяющей элементы, и имеет следующий смысл — владельцем конкретного счета не может быть и организация, и персона.

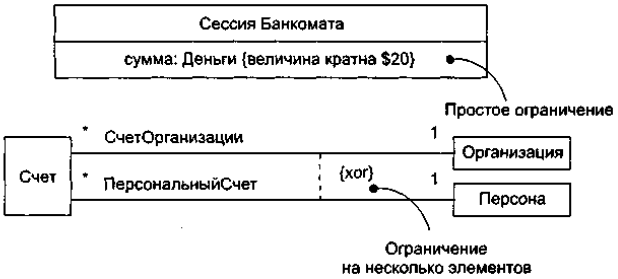

Рис. 10.17. Ограничения

Теговая величина (tagged value) расширяет характеристики строительного UML-блока, позволяя создать новую информацию в спецификации конкретного элемента. Теговую величину показывают как строку в фигурных скобках {}. Строка имеет вид

имя теговой величины = значение.

Иногда (в случае предопределенных тегов) указывается только имя теговой величины.

Отметим, что при работе с продуктом, имеющим много реализаций, полезно отслеживать версию и автора определенных блоков. Версия и автор не принадлежат к основным понятиям UML. Они могут быть добавлены к любому строительному блоку (например, к классу) введением в блок новых теговых величин. Например, на рис. 10.18 класс ТекстовыйПроцессор расширен путем явного указания его версии и автора.

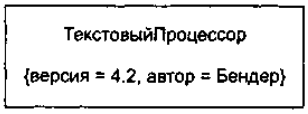

Рис. 10.18. Расширение класса

Стереотип (stereotype) расширяет словарь языка, позволяет создавать новые виды строительных блоков, производные от существующих и учитывающие специфику новой проблемы. Элемент со стереотипом является вариацией существующего элемента, имеющей такую же форму, но отличающуюся по сути. У него могут быть дополнительные ограничения и теговые величины, а также другое визуальное представление. Он иначе обрабатывается при генерации программного кода. Отображают стереотип как имя, указываемое в двойных угловых скобках (или в угловых кавычках).

Примеры элементов со стереотипами приведены на рис. 10.19. Стереотип «exception» говорит о том, что класс ПотеряЗначимости теперь рассматривается как специальный класс, которому, положим, разрешается только генерация и обработка сигналов исключений. Особые возможности метакласса получил класс ЭлементМодели. Кроме того, здесь показано применение стереотипа «call» к отношению зависимости (у него появился новый смысл).

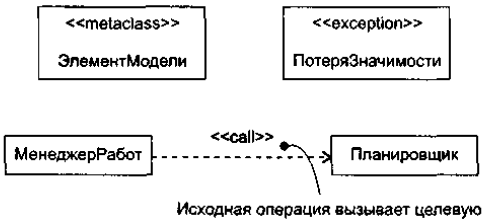

Рис. 10.19. Стереотипы

Таким образом, механизмы расширения позволяют адаптировать UML под нужды конкретных проектов и под новые программные технологии. Возможно добавление новых строительных блоков, модификация спецификаций существующих блоков и даже изменение их семантики. Конечно, очень важно обеспечить контролируемое введение расширений.

СТАТИЧЕСКИЕ МОДЕЛИ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ПРОГРАММНЫХ СИСТЕМ

Статические модели обеспечивают представление структуры систем в терминах базовых строительных блоков и отношений между ними. «Статичность» этих моделей состоит в том, что здесь не показывается динамика изменений системы во времени. Вместе с тем следует понимать, что эти модели несут в себе не только структурные описания, но и описания операций, реализующих заданное поведение системы. Основным средством для представления статических моделей являются диаграммы классов [8], [23], [53], [67]. Вершины диаграмм классов нагружены классами, а дуги (ребра) — отношениями между ними. Диаграммы используются:

- в ходе анализа — для указания ролей и обязанностей сущностей, которые обеспечивают поведение системы;
- в ходе проектирования — для фиксации структуры классов, которые формируют системную архитектуру.

Вершины в диаграммах классов

Итак, вершина в диаграмме классов — класс. Обозначение класса показано на рис. 11.1.

Рис. 11.1. Обозначение класса

Имя класса указывается всегда, свойства и операции — выборочно. Предусмотрено задание области действия свойства (операции). Если свойство (операция) подчеркивается, его областью действия является класс, в противном случае областью Действия является экземпляр (рис. 11.2).

Что это значит? Если областью действия свойства является класс, то все его экземпляры (объекты) используют общее значение этого свойства, в противном случае у каждого экземпляра свое значение свойства.

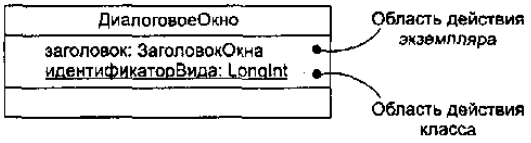

Рис. 11.2. Свойства уровней класса и экземпляра

Свойства

Общий синтаксис представления свойства имеет вид

Видимость Имя [Множественность]: Тип = НачальнЗначение {Характеристики}

Рассмотрим видимость и характеристики свойств.

В языке UML определены три уровня видимости:

public	Любой клиент класса может использовать свойство (операцию), обозначается символом +
	Любой наследник класса может использовать свойство (операцию), обозначается символом #
protected	Свойство (операция) может использоваться только самим классом, обозначается символом -
private	

ПРИМЕЧАНИЕ

Если видимость не указана, считают, что свойство объявлено с публичной видимостью.

Определены три характеристики свойств:

changeable	Нет ограничений на модификацию значения свойства
addOnly	Для свойств с множественностью, большей единицы; дополнительные значения могут быть добавлены, но после создания значение не может удаляться или изменяться
frozen	После инициализации объекта значение свойства не изменяется

ПРИМЕЧАНИЕ

Если характеристика не указана, считают, что свойство объявлено с характеристикой changeable.

Примеры объявления свойств:

начало	Только имя
+ начало	Видимость и имя
начало : Координаты	Имя и тип
имяфамилия [0..1] : String	Имя, множественность, тип
левыйУгол : Координаты=(0, 10)	Имя, тип, начальное значение
сумма : Integer {frozen}	Имя и характеристика

Операции

Общий синтаксис представления операции имеет вид

Видимость Имя (Список Параметров): ВозвращаемыйТип {Характеристики}

Примеры объявления операций:

записать	Только имя
+ записать	Видимость и имя
зарегистрировать) и: Имя, ф: Фамилия)	Имя и параметры
балансСчета ( ) : Integer	Имя и возвращаемый тип
нагревать ( ) (guarded)	Имя и характеристика

В сигнатуре операции можно указать ноль или более параметров, форма представления параметра имеет следующий синтаксис:

Направление Имя : Тип = ЗначениеПоУмолчанию

Элемент Направление может принимать одно из следующих значений:

in	Входной параметр, не может модифицироваться
out	Выходной параметр, может модифицироваться для передачи информации в вызывающий объект
inout	Входной параметр, может модифицироваться

Допустимо применение следующих характеристик операций:

leaf	Конечная операция, операция не может быть полиморфной и не может переопределяться (в цепочке наследования)
isQuery	Выполнение операции не изменяет состояния объекта
sequential	В каждый момент времени в объект поступает только один вызов операций. Как следствие, в каждый момент времени выполняется только одна операция объекта. Другими словами, допустим только один поток вызовов (поток управления)
guarded	Допускается одновременное поступление в объект нескольких вызовов, но в каждый момент времени обрабатывается только один вызов охраняемой операции. Иначе говоря, параллельные потоки управления исполняются последовательно (за счет постановки вызовов в очередь)
concurrent	В объект поступает несколько потоков вызовов операций (из параллельных потоков управления). Разрешается параллельное (и множественное) выполнение операции. Подразумевается, что такие операции являются атомарными

Организация свойств и операций

Известно, что пиктограмма класса включает три секции (для имени, для свойств и для операций). Пустота секции не означает, что у класса отсутствуют свойства или операции, просто в данный момент они не показываются. Можно явно определить наличие у класса большего количества свойств или атрибутов. Для этого в конце показанного списка проставляются три точки. Как показано на рис. 11.3, в длинных списках свойств и операций разрешается группировка — каждая группа начинается со своего стереотипа.

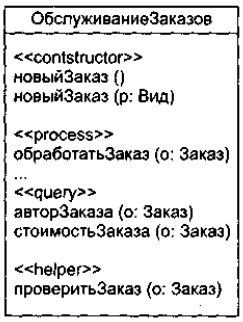

Рис. 11.3. Стереотипы для характеристик класса

Множественность

Иногда бывает необходимо ограничить количество экземпляров класса:

- задать ноль экземпляров (в этом случае класс превращается в утилиту, которая предлагает свои свойства и операции);
- задать один экземпляр (класс-singleton);
- задать конкретное количество экземпляров;

- не ограничивать количество экземпляров (это случай, предполагаемый по умолчанию).

Количество экземпляров класса называется его множественностью. Выражение множественности записывается в правом верхнем углу значка класса. Например, как показано на рис. 11.4, КонтроллерУглов — это класс-singleton, а для класса ДатчикУгла разрешены три экземпляра.

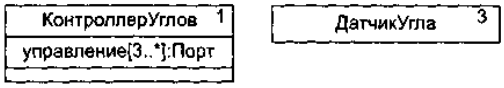

Рис. 11.4. Множественность

Множественность применима не только к классам, но и к свойствам. Множественность свойства задается выражением в квадратных скобках, записанным после его имени. Например, на рисунке заданы три и более экземпляра свойства Управление (в экземпляре класса КонтроллерУглов).

Отношения в диаграммах классов

Отношения, используемые в диаграммах классов, показаны на рис. 11.5.

Рис. 11.5. Отношения в диаграммах классов

Ассоциации отображают структурные отношения между экземплярами классов, то есть соединения между объектами. Каждая ассоциация может иметь метку — *имя,* которое описывает природу отношения. Как показано на рис. 11.6, имени можно придать направление — достаточно добавить треугольник направления, который указывает направление, заданное для чтения имени.

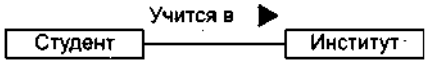

Рис. 11.6. Имена ассоциаций

Когда класс участвует в ассоциации, он играет в этом отношении определенную роль. Как показано на рис. 11.7, *роль* определяет, каким представляется класс на одном конце ассоциации для класса на противоположном конце ассоциации.

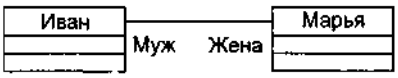

Рис. 11.7. Роли

Один и тот же класс в разных ассоциациях может играть разные роли. Часто важно знать, как много объектов может соединяться через экземпляр ассоциации. Это количество называется ложностью роли в ассоциации, записывается в виде выражения, задающего диапазон величин или одну величину (рис. 11.8).

Запись мощности на одном конце ассоциации определяет количество объектов, соединяемых с каждым объектом на противоположном конце ассоциации. Например, можно задать следующие варианты мощности:

- 5 — точно пять;
- * — неограниченное количество;
- 0..* — ноль или более;
- 1..* — один или более;
- 3..7 — определенный диапазон;
- 1..3, 7 — определенный диапазон или число.

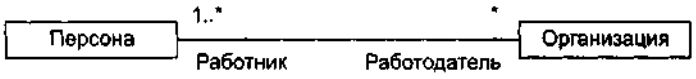

Рис. 11. 8. Мощность

Достаточно часто возникает следующая проблема — как для объекта на одном конце ассоциации выделить набор объектов на противоположном конце? Например, рассмотрим взаимодействие между банком и клиентом — вкладчиком. Как показано на рис. 11.9, мы устанавливаем ассоциацию между классом Банк и классом Клиент. В контексте Банка мы имеем НомерСчета, который позволяет идентифицировать конкретного Клиента. В этом смысле НомерСчета является атрибутом ассоциации. Он не является характеристикой Клиента, так как Клиенту не обязательно знать служебные параметры его счета. Теперь для данного экземпляра Банка и данного значения НомераСчета можно выявить ноль или один экземпляр Клиента. В UML для решения этой проблемы вводится *квалификатор* — атрибут ассоциации, чьи значения выделяют набор объектов, связанных с объектом через ассоциацию. Квалификатор изображается маленьким прямоугольником, присоединенным к концу ассоциации. В прямоугольник вписывается свойство — атрибут ассоциации.

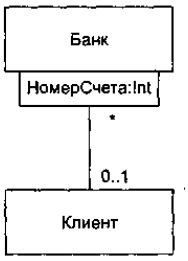

Рис. 11.9. Квалификация

Кроме того, роли в ассоциациях могут иметь пометки *видимости.* Например, на рис. 11.10 показаны ассоциации между Начальником и Женщиной, а также между Женщиной и Загадкой. Для данного экземпляра Начальника можно определить соответствующие экземпляры Женщины. С другой стороны, Загадка приватна для Женщины, поэтому она недоступна извне. Как показано на рисунке, из объекта Начальника можно перемещаться к экземплярам Женщины (и наоборот), но нельзя видеть экземпляры Загадки для объектов Женщины.

Рис. 11.10. Видимость

На конце ассоциации можно задать три уровня видимости, добавляя символ видимости к имени роли:

- по умолчанию для роли задается публичная видимость;
- приватная видимость указывает, что объекты на данном конце недоступны любым объектам вне ассоциации;
- защищенная видимость (protected) указывает, что объекты на данном конце недоступны любым объектам вне ассоциации, за исключением потомков того класса, который указан на противоположном конце ассоциации.

В языке UML ассоциации могут иметь свойства. Как показано на рис, 11.11, такие возможности отображаются с помощью классов-ассоциаций. Эти классы присоединяются к линии ассоциации пунктирной линией и рассматриваются как классы со свойствами ассоциаций или как ассоциации со свойствами классов.

Рис. 11.11. Класс-ассоциация

Свойства класса-ассоциации характеризуют не один, а пару объектов, в данном случае — пару экземпляров, Профессор и Университет.

Отношения агрегации и композиции в языке UML считаются разновидностями ассоциации, применяемыми для отображения структурных отношений между «целым» (агрегатом) и его «частями». *Агрегация* показывает отношение по ссылке (в агрегат включены только указатели на части), *композиция* — отношение физического включения (в агрегат включены сами части).

Зависимость является отношением использования между клиентом (зависимым элементом) и поставщиком (независимым элементом). Обычно операции клиента:

- вызывают операции поставщика;
- имеют сигнатуры, в которых возвращаемое значение или аргументы принадлежат классу поставщика.

Например, на рис. 11.12 показана зависимость класса Заказ от класса Книга, так как Книга используется в операциях проверкаДоступности, добавить и удалить класса Заказ.

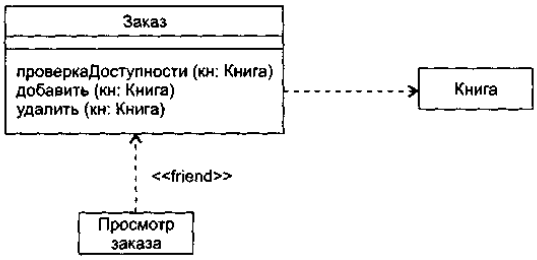

Рис. 11.12. Отношения зависимости

На этом рисунке изображена еще одна зависимость, которая показывает, что класс Просмотр Заказа использует класс Заказ. Причем Заказ ничего не знает о Просмотре Заказа. Данная зависимость помечена стереотипом «friend», который расширяет простую зависимость, определенную в языке. Отметим, что отношение зависимости очень разнообразно — в настоящее время язык предусматривает 17 разновидностей зависимости, различаемых по стереотипам.

Обобщение — отношение между общим предметом (суперклассом) и специализированной разновидностью этого предмета (подклассом). Подкласс может иметь одного родителя (один суперкласс) или несколько родителей (несколько суперклассов). Во втором случае говорят о множественном наследовании.

Как показано на рис. 11.13, подкласс Летающий шкаф является наследником суперклассов Летающий предмет и Хранилище вещей. Этому подклассу достаются в наследство все свойства и операции двух классов-родителей.

Множественное наследование достаточно сложно и коварно, имеет много «подводных камней». Например, подкласс Яблочный_Пирог не следует производить от суперклассов Пирог и Яблоко. Это типичное неправильное использование множественного наследования: потомок наследует все свойства от его родителя, хотя обычно не все свойства применимы к потомку. Очевидно, что Яблочный_Пирог является Пирогом, но не является Яблоком, так как пироги не растут на деревьях.

Рис. 11.13. Множественное наследование

Еще более сложные проблемы возникают при наследовании от двух классов, имеющих общего родителя. Говорят, что в результате образуется ромбовидная решетка наследования (рис. 11.14).

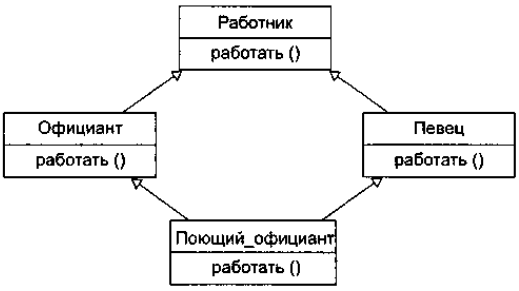

Рис. 11.14. Ромбовидная решетка наследования

Полагаем, что в подклассах Официант и Певец операция работать суперкласса Работник переопределена в соответствии с обязанностью подкласса (работа официанта состоит в обслуживании едой, а певца — в пении). Возникает вопрос — какую версию операции работать унаследует Поющий_официант? А что делать со свойствами, доставшимися в наследство от родителей и общего прародителя? Хотим ли мы иметь несколько копий свойства или только одну?

Все эти проблемы увеличивают сложность реализации, приводят к введению многочисленных правил для обработки особых случаев.

Реализация — семантическое отношение между классами, в котором класс-приемник выполняет реализацию операций интерфейса класса-источника. Например, на рис. 11.15 показано, что класс Каталог должен реализовать интерфейс Обработчик каталога, то есть Обработчик каталога рассматривается как источник, а Каталог — как приемник.

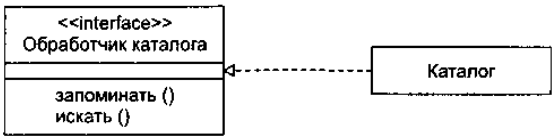

Рис. 11.15. Реализация интерфейса

Интерфейс Обработчик каталога позволяет клиентам взаимодействовать с объектами класса Каталог без знания той дисциплины доступа, которая здесь реализована (LIFO — последний вошел, первый вышел; FIFO — первый вошел, первый вышел и т. д.).

Деревья наследования

При использовании отношений обобщения строится иерархия классов. Некоторые классы в этой иерархии могут быть абстрактными. *Абстрактным* называют класс, который не может иметь экземпляров. Имена абстрактных классов записываются курсивом. Например, на рис. 11.16 показаны абстрактные классы *Млекопитающие, Собаки, Кошки.*

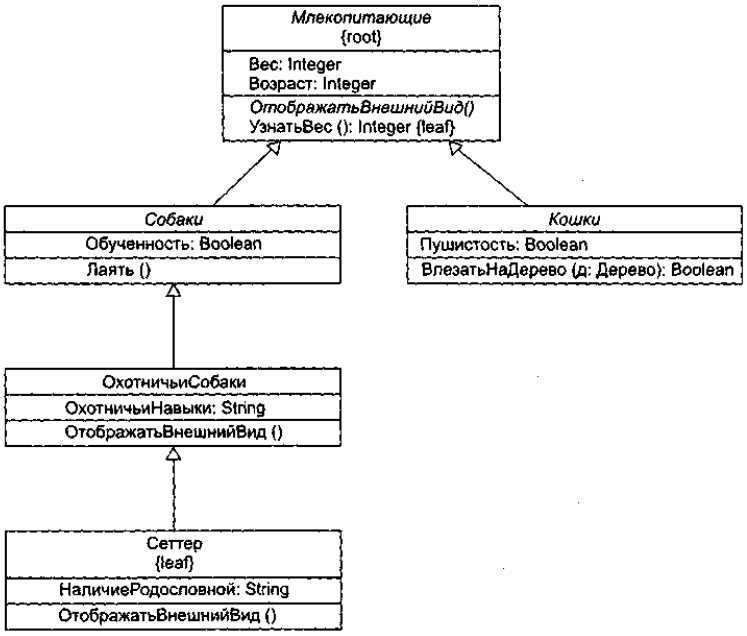

Рис. 11.16. Абстрактность и полиморфизм

Кроме того, здесь имеются конкретные классы ОхотничьиСобаки, Сеттер, каждый из которых может иметь экземпляры.

Обычно класс наследует какие-то характеристики класса-родителя и передает свои характеристики классу-потомку. Иногда требуется определить *конечный* класс, который не может иметь детей. Такие классы помечаются теговой величиной (характеристикой) leaf, записываемой за именем класса. Например, на рисунке показан конечный класс Сеттер.

Иногда полезно отметить *корневой* класс, который не может иметь родителей. Такой класс помечается теговой величиной (характеристикой) root, записываемой за именем класса. Например, на рисунке показан корневой класс *Млекопитающие.*

Аналогичные свойства имеют и операции. Обычно операция является полиморфной, это значит, что в различных точках иерархии можно определять операции с похожей сигнатурой. Такие операции из дочерних классов переопределяют поведение соответствующих операций из родительских классов. При обработке сообщения (в период выполнения) производится полиморфный выбор одной из операций иерархии в соответствии с типом объекта. Например, ОтображатьВнешнийВид () и ВлезатьНаДерево (дуб) — полиморфные операции. К тому же операция *Млекопитающие::ОтображатьВнешнийВид ( )* является абстрактной, то есть неполной и требующей для своей реализации потомка. Имя абстрактной операции записывается курсивом (как и имя класса). С другой стороны, Млекопитающие::УзнатьВес () — конечная операция, что отмечается характеристикой leaf. Это значит, что операция не полиморфна и не может перекрываться.

ДИНАМИЧЕСКИЕ МОДЕЛИ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ПРОГРАММНЫХ СИСТЕМ

Динамические модели обеспечивают представление поведения систем. «Динамизм» этих моделей состоит в том, что в них отражается изменение состояний в процессе работы системы (в зависимости от времени). Средства языка UML для создания динамических моделей многочисленны и разнообразны [8], [23], [41], [53], [67]. Эти средства ориентированы не только на собственно программные системы, но и на отображение требований заказчика к поведению таких систем.

Моделирование поведения программной системы

Для моделирования поведения системы используют:

- автоматы;
- взаимодействия.

Автомат (State machine) описывает поведение в терминах последовательности состояний, через которые проходит объект в течение своей жизни. Взаимодействие (Interaction) описывает поведение в терминах обмена сообщениями между объектами.

Таким образом, автомат задает поведение системы как цельной, единой сущности; моделирует жизненный цикл единого объекта. В силу этого автоматный подход удобно применять для формализации динамики отдельного трудного для понимания блока системы.

Взаимодействия определяют поведение системы в виде коммуникаций между его частями (объектами), представляя систему как сообщество совместно работающих объектов. Именно поэтому взаимодействия считают основным аппаратом для фиксации полной динамики системы.

Автоматы отображают с помощью:

- диаграмм схем состояний;

- диаграмм деятельности.

Взаимодействия отображают с помощью:

- диаграмм сотрудничества (кооперации);
- диаграмм последовательности.

Диаграммы схем состояний

Диаграмма схем состояний — одна из пяти диаграмм UML, моделирующих динамику систем. Диаграмма схем состояний отображает конечный автомат, выделяя поток управления, следующий от состояния к состоянию. Конечный автомат — поведение, которое определяет последовательность состояний в ходе существования объекта. Эта последовательность рассматривается как ответ на события и включает реакции на эти события.

Диаграмма схем состояний показывает:

1) набор состояний системы;

2) события, которые вызывают переход из одного состояния в другое;

3) действия, которые происходят в результате изменения состояния.

В языке UML состоянием называют период в жизни объекта, на протяжении которого он удовлетворяет какому-то условию, выполняет определенную деятельность или ожидает некоторого события. Как показано на рис. 12.1, состояние изображается как закругленный прямоугольник, обычно включающий его имя и подсостоя-ния (если они есть).

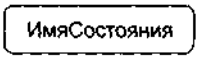

Рис. 12.1. Обозначение состояния

Переходы между состояниями отображаются помеченными стрелками (рис. 12.2).

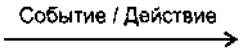

Рис. 12.2. Переходы между состояниями

На рис. 12.2 обозначено: Событие — происшествие, вызывающее изменение состояния, Действие — набор операций, запускаемых событием.

Иначе говоря, события вызывают переходы, а действия являются реакциями на переходы.

Примеры событий:

баланс < 0	Изменение в состоянии
помехи	Сигнал (объект с именем)
уменьшить(Давление)	Вызов действия
after (5 seconds)	Истечение периода времени
when (time = 16:30)	Наступление абсолютного момента времени

Примеры действий:

Кассир. прекратитьВыплаты( )	Вызов одной операции
flt:= new(Фильтр); Ш.убратьПомехи( )	Вызов двух операций
send Ник. привет	Посылка сигнала в объект Ник

ПРИМЕЧАНИЕ

Для отображения посылки сигнала используют специальное обозначение — перед именем сигнала указывают служебное слово send.

Для отображения перехода в начальное состояние принято обозначение, показанное на рис. 12.3.

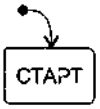

Рис. 12.3. Переход в начальное состояние

Соответственно, обозначение перехода в конечное состояние имеет вид, представленный на рис. 12.4.

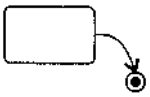

Рис. 12.4. Переход в конечное состояние

В качестве примера на рис. 12.5 показана диаграмма схем состояний для системы охранной сигнализации.

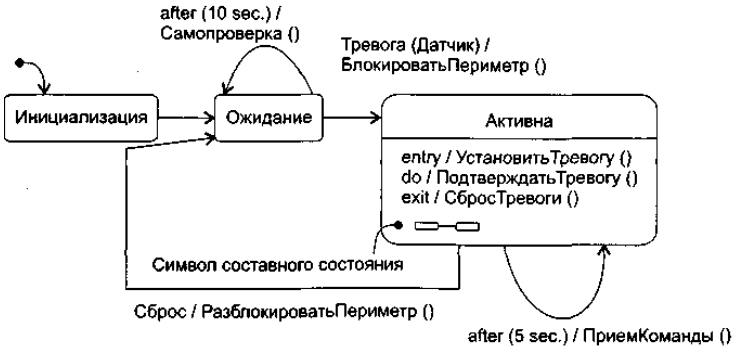

Рис. 12.5. Диаграмма схем состояний системы охранной сигнализации

Из рисунка видно, что система начинает свою жизнь в состоянии Инициализация, затем переходит в состояние Ожидание. В этом состоянии через каждые 10 секунд (по событию after (10 sec.)) выполняется самопроверка системы (операция Самопроверка ()). При наступлении события Тревога (Датчик) реализуются действия, связанные с блокировкой периметра охраняемого объекта, — исполняется операция БлокироватьПериметр() и осуществляется переход в состояние Активна. В активном состоянии через каждые 5 секунд по событию after (5 sec.) запускается операция ПриемКоманды(). Если команда получена (наступило событие Сброс), система возвращается в состояние Ожидание. В процессе возврата разблокируется периметр охраняемого объекта (операция РазблокироватьПериметр()).

Действия в состояниях

Для указания действий, выполняемых при входе в состояние и при выходе из состояния, используются метки entry и exit соответственно.

Например, как показано на рис. 12.6, при входе в состояние Активна выполняется операция УстановитьТревогу() из класса Контроллер, а при выходе из состояния — операция СбросТревоги().

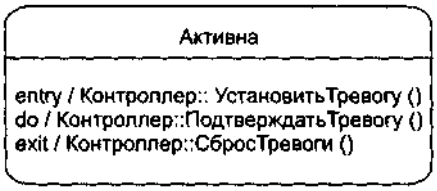

Рис. 12.6. Входные и выходные действия и деятельность в состоянии Активна

Действие, которое должно выполняться, когда система находится в данном состоянии, указывается после метки do. Считается, что такое действие начинается при входе в состояние и заканчивается при выходе из него. Например, в состоянии Активна это действие ПодтверждатьТревогу().

Условные переходы

Между состояниями возможны различные типы переходов. Обычно переход инициируется событием. Допускаются переходы и без событий. Наконец, разрешены условные или охраняемые переходы.

Правила пометки стрелок условных переходов иллюстрирует рис. 12.7.

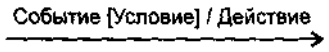

Рис. 12.7. Обозначение условного перехода

Порядок выполнения условного перехода:

1) происходит событие;
2) вычисляется условие УсловиеПерехода;
3) при УсловиеПерехода=true запускается переход и активизируется действие, в противном случае переход не выполняется.

Пример условного перехода между состояниями Инициализация и Ожидание приведен на рис. 12.8. Он происходит по событию ПитаниеПодано, но только в том случае, если достигнут боевой режим лазера.

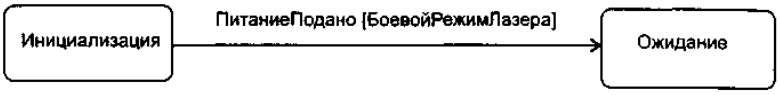

Рис. 12.8. Условный переход между состояниями

Вложенные состояния

Одной из наиболее важных характеристик конечных автоматов в UML является подсостояние. Подсостояние позволяет значительно упростить моделирование сложного поведения. Подсостояние — это состояние, вложенное в другое состояние. На рис. 12.9 показано составное состояние, содержащее в себе два подсостояния.

Рис. 12.9. Обозначение подсостояний

На рис. 12.10 приведена внутренняя структура составного состояния Активна.

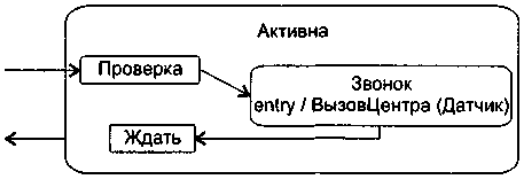

Рис. 12.10. Переходы в состоянии Активна

Семантика вложенности такова: если система находится в состоянии Активна, то она должна быть точно в одном из подсостояний: Проверка, Звонок, Ждать. В свою очередь, в подсостояние могут вкладываться другие подсостояния. Степень вложенности подсостояний не ограничивается. Данная семантика соответствует случаю последовательных подсостояний.

Возможно наличие параллельных подсостояний — они выполняются параллельно внутри составного состояния. Графически изображения параллельных подсостояний отделяются друг от друга пунктирными линиями.

Иногда при возврате в составное состояние возникает необходимость попасть в то его подсостояние, которое в прошлый раз было последним. Такое подсостояние называют историческим. Информация об историческом состоянии запоминается. Как показано на рис. 12.11, подобная семантика переходов отображается значком истории — буквой H внутри кружка.

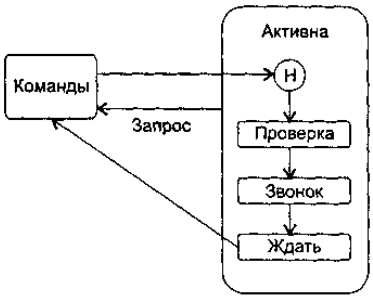

Рис. 12.11. Историческое состояние

При первом посещении состояния Активна автомат не имеет истории, поэтому происходит простой переход в подсостояние Проверка. Предположим, что в подсостоя-нии Звонок произошло событие Запрос. Средства управления заставляют автомат покинуть подсостояние Звонок (и состояние Активна) и вернуться в состояние Команды. Когда работа в состоянии Команды завершается, выполняется возврат в историческое подсостояние состояния Активна. Поскольку теперь автомат запомнил историю, он переходит прямо в подсостояние Звонок (минуя подсостояние Проверка).

Как показано на рис. 12.12, для обозначения составного состояния, имеющего внутри себя скрытые (не показанные на диаграмме) подсостояния, используется символ «очки».

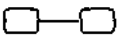

Рис. 12.12. Символ состояния со скрытыми подсостояниями

Диаграммы деятельности

Диаграмма деятельности представляет особую форму конечного автомата, в которой показываются процесс вычислений и потоки работ. В ней выделяются не обычные состояния объекта, а состояния выполняемых вычислений — состояния действий. При этом полагается, что процесс вычислений не прерывается внешними событиями. Словом, диаграммы деятельности очень похожи на блок-схемы алгоритмов.

Основной вершиной в диаграмме деятельности является состояние действия (рис. 12.13), которое изображается как прямоугольник с закругленными боковыми сторонами.

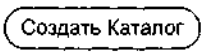

Рис. 12.13. Состояние действия

Состояние действия считается атомарным (действие нельзя прервать) и выполняется за один квант времени, его нельзя подвергнуть декомпозиции. Если нужно представить сложное действие, которое можно подвергнуть дальнейшей декомпозиции (разбить на ряд более простых действий), то используют состояние под-деятельности. Изображение состояния под-деятельности содержит пиктограмму в правом нижнем углу (рис. 12.14).

Рис. 12.14. Состояние под-деятельности

Фактически в данную вершину вписывается имя другой диаграммы, имеющей внутреннюю структуру.

Переходы между вершинами — состояниями действий — изображаются в виде стрелок. Переходы выполняются по окончании действий.

Кроме того, в диаграммах деятельности используются вспомогательные вершины:

- решение (ромбик с одной входящей и несколькими исходящими стрелками);
- объединение (ромбик с несколькими входящими и одной исходящей стрелкой);
- линейка синхронизации — разделение (жирная горизонтальная линия с одной входящей и несколькими исходящими стрелками);
- линейка синхронизации — слияние (жирная горизонтальная линия с несколькими входящими и одной исходящей стрелкой);
- начальное состояние (черный кружок);
- конечное состояние (незакрашенный кружок, в котором размещен черный кружок меньшего размера).

Вершина «решение» позволяет отобразить разветвление вычислительного процесса, исходящие из него стрелки помечаются сторожевыми условиями ветвления.

Вершина «объединение» отмечает точку слияния альтернативных потоков действий.

Линейки синхронизации позволяют показать параллельные потоки действий, отмечая точки их синхронизации при запуске (момент разделения) и при завершении (момент слияния).

Пример диаграммы деятельности приведен на рис. 12.15. Эта диаграмма описывает деятельность покупателя в Интернет-магазине. Здесь представлены две точки ветвления — для выбора способа поиска товара и для принятия решения о покупке. Присутствуют три линейки синхронизации: верхняя отражает разделение на два параллельных процесса, средняя отражает и разделение, и слияние процессов, а нижняя — только слияние процессов.

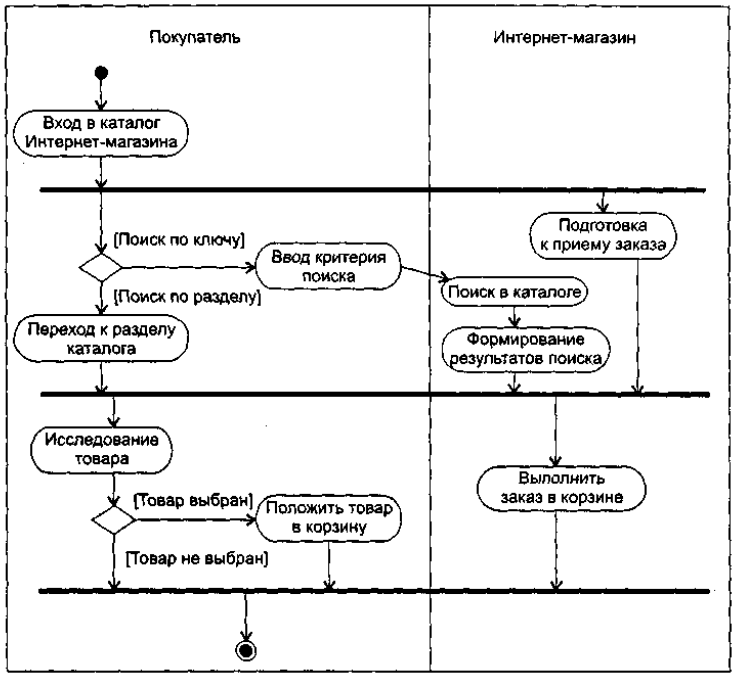

Рис. 12.15. Диаграмма деятельности покупателя в Интернет-магазине

Дополнительно на этой диаграмме показаны две плавательные дорожки — дорожка покупателя и дорожка магазина, которые разделены вертикальной линией. Каждая дорожка имеет имя и фиксирует область деятельности конкретного лица, обозначая зону его ответственности.

Диаграммы взаимодействия

Диаграммы взаимодействия предназначены для моделирования динамических аспектов системы. Диаграмма взаимодействия показывает взаимодействие, включающее набор объектов и их отношений, а также пересылаемые между

объектами сообщения. Существуют две разновидности диаграммы взаимодействия — диаграмма последовательности и диаграмма сотрудничества. Диаграмма последовательности — это диаграмма взаимодействия, которая выделяет упорядочение сообщений по времени. Диаграмма сотрудничества — это диаграмма взаимодействия, которая выделяет структурную организацию объектов, посылающих и принимающих сообщения. Элементами диаграмм взаимодействия являются участники взаимодействия — объекты, связи, сообщения.

Диаграммы сотрудничества

Диаграммы сотрудничества отображают взаимодействие объектов в процессе функционирования системы. Такие диаграммы моделируют сценарии поведения системы. В русской литературе диаграммы сотрудничества часто называют диаграммами кооперации.

Обозначение объекта показано на рис. 12.16.

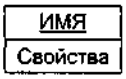

Рис. 12.16. Обозначение объекта

Имя объекта подчеркивается и указывается всегда, свойства указываются выборочно. Синтаксис представления имени имеет вид

ИмяОбъекта : ИмяКласса

Примеры записи имени:

Адам : Человек	Имя объекта и класса
: Пользователь	Только имя класса (анонимный объект)
мойКомпьютер	Только имя объекта (подразумевается, что имя класса известно)
агент :	Объект — сирота (подразумевается, что имя класса неизвестно)

Синтаксис представления свойства имеет вид

Имя : Тип = Значение

Примеры записи свойства:

номер:Телефон = "7350-420"	Имя, тип, значение
активен = True	Имя и значение

Объекты взаимодействуют друг с другом с помощью связей — каналов для передачи сообщений. Связь между парой объектов рассматривается как экземпляр ассоциации между их классами. Иными словами, связь между двумя объектами существует только тогда, когда имеется ассоциация между их классами. Неявно все классы имеют ассоциацию сами с собой, следовательно, объект может послать сообщение самому себе.

Итак, связь — это путь для пересылки сообщения. Путь может быть снабжен характеристикой видимости. Характеристика видимости проставляется как стандартный стереотип над дальним концом связи. В языке предусмотрены следующие стандартные стереотипы видимости:

«global»	Объект-поставщик находится в глобальной области определения
«local»	Объект-поставщик находится в локальной области определения объекта-клиента
«parameter»	Объект-поставщик является параметром операции объекта-клиента
«self»	Один и тот же объект является и клиентом, и поставщиком

Сообщение — это спецификация передачи информации между объектами в ожидании того, что будет обеспечена требуемая деятельность. Прием сообщения рассматривается как событие.

Результатом обработки сообщения обычно является действие. В языке UML моделируются следующие разновидности действий:

Вызов	В объекте запускается операция
Возврат	Возврат значения в вызывающий объект
Посылка(Send)	В объект посылается сигнал
Создание	Создание объекта, выполняется по стандартному сообщению «create»
Уничтожение	Уничтожение объекта, выполняется по стандартному сообщению «destroy»

Для записи сообщений в языке UML принят следующий синтаксис:

ВозврВеличина := ИмяСообщения (Аргументы),

где ВозврВеличина задает величину, возвращаемую как результат обработки сообщения.

Примеры записи сообщений:

Коорд := ТекущПоложение(самолетТ1)	Вызов операции, возврат значения
оповещение( )	Посылка сигнала
УстановитьМаршрут(х)	Вызов операции с действительным параметром
«create»	Стандартное сообщение для создания объекта

Когда объект посылает сообщение в другой объект (делегируя некоторое действие получателю), объект-получатель, в свою очередь, может послать сообщение в третий объект, и т. д. Так формируется поток сообщений — последовательность управления. Очевидно, что сообщения в последовательности должны быть пронумерованы. Номера записываются перед именами сообщений, направления сообщений указываются стрелками (размещаются над линиями связей).

Наиболее общую форму управления задает процедурный или вложенный поток (поток синхронных сообщений). Как показано на рис. 12.17, процедурный поток рисуется стрелками с заполненными наконечниками.

Рис. 12.17. Поток синхронных сообщений

Здесь сообщение 2.1 : Напиток : = Изготовить(СмесьN3) определено как первое сообщение, вложенное во второе сообщение 2 : Заказать(СмесьN3) последовательности, а сообщение 2.2 : Принести(Напиток) — как второе вложенное сообщение. Все сообщения процедурной последовательности считаются синхронными. Работа с синхронным сообщением подчиняется следующему правилу: передатчик ждет до тех пор, пока получатель не примет и не обработает сообщение. В нашем примере это означает, что третье сообщение будет послано только после обработки сообщений 2.1 и 2.2. Отметим, что степень вложенности сообщений может быть любой. Главное, чтобы соблюдалось правило: последовательность сообщений внешнего уровня возобновляется только после завершения вложенной последовательности.

Менее общую форму управления задает асинхронный поток управления. Как показано на рис. 12.18, асинхронный поток рисуется обычными стрелками. Здесь все сообщения считаются асинхронными, при которых передатчик не ждет реакции от получателя сообщения. Такой вид коммуникации имеет семантику почтового ящика — получатель принимает сообщение по мере готовности. Иными словами, передатчик и получатель не синхронизируют свою работу, скорее, один объект «избавляется» от сообщения для другого объекта. В нашем примере сообщение ПодтверждениеВызова определено как второе сообщение в последовательности.

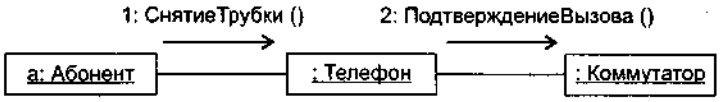

Рис. 12.18. Поток асинхронных сообщений

Помимо рассмотренных линейных потоков управления, можно моделировать и более сложные формы — итерации и ветвления.

Итерация представляет повторяющуюся последовательность сообщений. После номера сообщения итерации добавляется выражение

*[i := 1 .. n].

Оно означает, что сообщение итерации будет повторяться заданное количество раз. Например, четырехкратное повторение первого сообщения РисоватьСторонуПрямоугольника можно задать выражением

1*[1 := 1 .. 4] : РисоватьСторонуПрямоугольника(i)

Для моделирования ветвления после номера сообщения добавляется выражение условия, например: [x>0]. Сообщение альтернативной ветви помечается таким же номером, но с другим условием: [x<=0]. Пример итерационного и разветвляющегося потока сообщений приведен на рис. 12.19.

Здесь первое сообщение повторяется 4 раза, а в качестве второго выбирается одно из двух сообщений (в зависимости от значения переменной x). В итоге экземпляр рисователя нарисует на экране прямоугольное окно, а экземпляр собеседника выведет в него соответствующее донесение.

Таким образом, для формирования диаграммы сотрудничества выполняются следующие действия:

1) отображаются объекты, которые участвуют во взаимодействии;

2) рисуются связи, соединяющие эти объекты;

3) связи помечаются сообщениями, которые посылают и получают выделенные объекты.

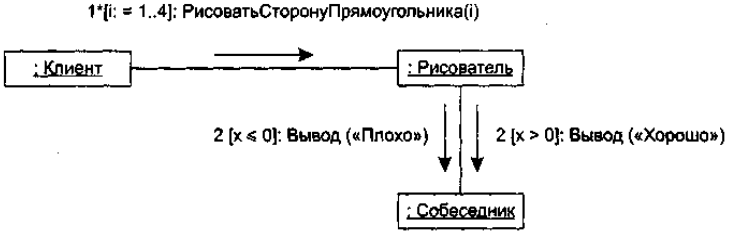

Рис. 12.19. Итерация и ветвление

В итоге формируется ясное визуальное представление потока управления (в контексте структурной организации сотрудничающих объектов).

В качестве примера на рис. 12.20 приведена диаграмма сотрудничества системы управления полетом летательного аппарата.

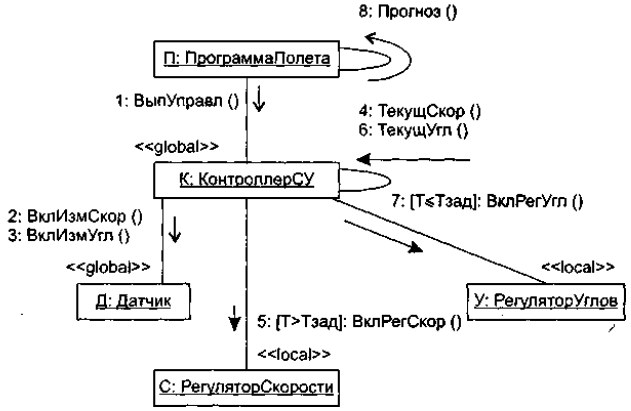

Рис. 12.20. Диаграмма сотрудничества системы управления полетом

На данной диаграмме представлены пять объектов, явно показаны характеристики видимости всех связей системы. Поток управления в системе включает восемь сообщений: четыре асинхронных и четыре синхронных сообщения. Экземпляр Контроллера СУ ждет приема и обработки сообщений:

- ВклРегСкор( );
- ВрРегУгл();
- ТекущСкор();
- ТекущУгл( ).

Порядок следования сообщений задан их номерами. Для пятого и седьмого сообщений указаны условия:

- включение Регулятора Скорости происходит, если относительное время полета T больше заданного периода $T_{зад}$;
- включение Регулятора Углов обеспечивается, если относительное время поле-; та меньше или равно заданному периоду.

Диаграммы последовательности

Диаграмма последовательности — вторая разновидность диаграмм взаимодействия. Отражая сценарий поведения в системе, эта диаграмма обеспечивает более наглядное представление порядка передачи сообщений. Правда, она не позволяет показать такие детали, которые видны на диаграмме сотрудничества (структурные характеристики объектов и связей).

Графически диаграмма последовательности — разновидность таблицы, которая показывает объекты, размещенные вдоль оси *X*, и сообщения, упорядоченные по времени вдоль оси *Y*.

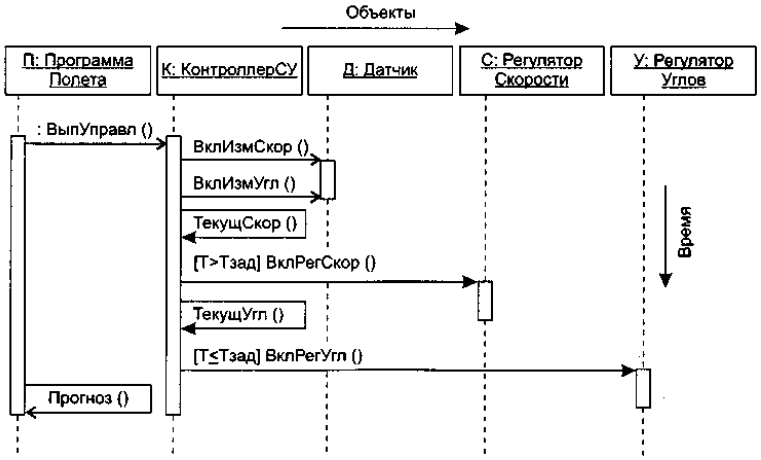

Рис. 12.21. Диаграмма последовательности системы управления полетом

Как показано на рис. 12.21, объекты, участвующие во взаимодействии, помещаются на вершине диаграммы, вдоль оси *X*. Обычно слева размещается объект, инициирующий взаимодействие, а справа — объекты по возрастанию подчиненности. Сообщения, посылаемые и принимаемые объектами, помещаются вдоль оси *Y* в порядке возрастания времени от вершины к основанию диаграммы. Используются те же синтаксис и обозначения синхронизации, что и в диаграммах сотрудничества. Таким образом, обеспечивается простое визуальное представление потока управления во времени.

От диаграмм сотрудничества диаграммы последовательности отличают две важные характеристики.

Первая характеристика — *линия жизни* объекта.

Линия жизни объекта — это вертикальная пунктирная линия, которая обозначает период существования объекта. Большинство объектов существуют на протяжении всего взаимодействия, их линии жизни тянутся от вершины до основания

диаграммы. Впрочем, объекты могут создаваться в ходе взаимодействия. Их линии жизни начинаются с момента приема сообщения «create». Кроме того, объекты могут уничтожаться в ходе взаимодействия. Их линии жизни заканчиваются с момента приема сообщения «destroy». Как представлено на рис. 12.22, уничтожение линии жизни отмечается пометкой X в конце линии:

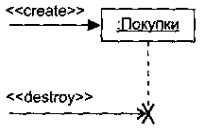

Рис. 12.22. Создание и уничтожение объекта

Вторая характеристика — *фокус управления.*

Фокус управления — это высокий тонкий прямоугольник, отображающий период времени, в течение которого объект выполняет действие (свою или подчиненную процедуру). Вершина прямоугольника отмечает начало действия, а основание — его завершение. Момент завершения может маркироваться сообщением возврата, которое показывается пунктирной стрелкой. Можно показать вложение фокуса управления (например, рекурсивный вызов собственной операции). Для этого второй фокус управления рисуется немного правее первого (рис. 12.23).

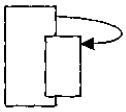

Рис. 12.23. Вложение фокусов управления

Замечания.

1. Для отображения «условности» линия жизни может быть разделена на несколько параллельных линий жизни. Каждая отдельная линия соответствует условному ветвлению во взаимодействии. Далее в некоторой точке линии жизни могут быть снова слиты (рис. 12.24).

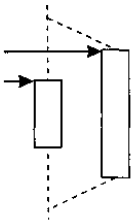

Рис. 12.24. Параллельные линии жизни

2. Ветвление показывается множеством стрелок, идущих из одной точки. Каждая стрелка отмечается сторожевым условием (рис. 12.25).

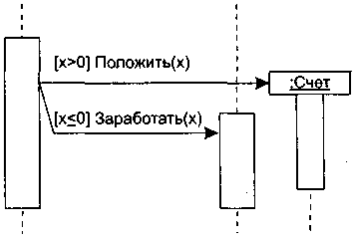

Рис. 12.25. Ветвление

МОДЕЛИ РЕАЛИЗАЦИИ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ПРОГРАММНЫХ СИСТЕМ

Статические и динамические модели описывают логическую организацию системы, отражают логический мир программного приложения. Модели реализации обеспечивают представление системы в физическом мире, рассматривая вопросы упаковки логических элементов в компоненты и размещения компонентов в аппаратных узлах [8], [23], [53], [67].

Компонентные диаграммы

Компонентная диаграмма — первая из двух разновидностей диаграмм реализации, моделирующих физические аспекты объектно-ориентированных систем. Компонентная диаграмма показывает организацию набора компонентов и зависимости между компонентами.

Элементами компонентных диаграмм являются компоненты и интерфейсы, а также отношения зависимости и реализации. Как и другие диаграммы, компонентные диаграммы могут включать примечания и ограничения. Кроме того, компонентные диаграммы могут содержать пакеты или подсистемы, используемые для группировки элементов модели в крупные фрагменты.

Компоненты

По своей сути компонент является физическим фрагментом реализации системы, который заключает в себе программный код (исходный, двоичный, исполняемый), сценарные описания или наборы команд операционной системз (имеются в виду командные файлы). Язык UML дает следующее определение.

Компонент — физическая и заменяемая часть системы, которая соответствует набору интерфейсов и обеспечивает реализацию этого набора интерфейсов.

Интерфейс — очень важная часть понятия «компонент», его мы обсудим в следующем подразделе. Графически компонент изображается как прямоугольник с вкладками, обычно включающий имя (рис. 13.1).

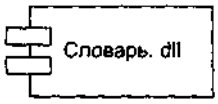

Рис. 13.1. Обозначение компонента

Компонент — базисный строительный блок физического представления ПО, поэтому интересно сравнить его с базисным строительным блоком логического представления ПО — классом.

Сходные характеристики компонента и класса:

- наличие имени;
- реализация набора интерфейсов;
- участие в отношениях зависимости;
- возможность быть вложенным;
- наличие экземпляров (экземпляры компонентов можно использовать только в диаграммах размещения).

Вы скажете — много общего. И тем не менее между компонентами и классами есть существенная разница, ее характеризует табл. 13.1.

Таблица 13.1. Различия компонентов и классов

№	Описание
1	Классы — логические абстракции, компоненты — физические предметы, которые живут в мире битов. В частности, компоненты могут «жить» в физических узлах, а классы лишены такой возможности
2	Компоненты являются физическими упаковками, контейнерами, инкапсулирующими в себе различные логические элементы. Они — элементы абстракций другого уровня
3	Классы имеют свойства и операции. Компоненты имеют только операции, которые доступны через их интерфейсы

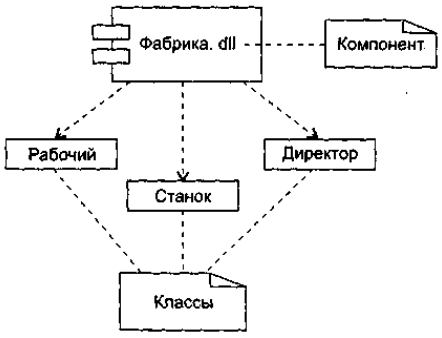

Рис. 13.2. Классы в компоненте

О чем говорят эти различия? Во-первых, класс не может «дышать» воздухом физического мира реализации. Ему нужен скафандр. Таким скафандром является компонент.

Во-вторых, им не жить друг без друга — пустые скафандры никому не нужны. Причем в скафандре-компоненте может находиться несколько классов и кооопераций. Итак, в скафандре — физической реализации — располагается набор логики. Как показано на рис. 13.2, с помощью отношения зависимости можно явно отобразить отношение между компонентом и классами, которые он реализует. Правда, чаще всего такие отношения не отображаются. Их удобно представлять в компонентной спецификации.

В-третьих, класс — душа нараспашку (он может даже показать свои свойства). Компонент всегда застегнут на все пуговицы (правда, из него торчат интерфейсные разъемы операций).

Теперь уместно перейти к обсуждению интерфейсов.

Интерфейсы

Интерфейс — список операций, которые определяют услуги класса или компонента. Образно говоря, интерфейс — это разъем, который торчит из ящичка компонента. С помощью интерфейсных разъемов компоненты стыкуются друг с другом, объединяясь в систему.

Еще одна аналогия. Интерфейс подобен абстрактному классу, у которого отсутствуют свойства и работающие операции, а есть только абстрактные операции (не имеющие тел). Если хотите, интерфейс похож на улыбку чеширского кота из правдивой истории об Алисе, где кот отдельно и улыбка отдельно. Все операции интерфейса открыты и видимы клиенту (в противном случае они потеряли бы всякий смысл). Итак, операции интерфейса только именуют предлагаемые услуги, не более того.

Очень важна взаимосвязь между компонентом и интерфейсом. Возможны два способа отображения взаимосвязи между компонентом и его интерфейсами. В первом, свернутом способе, как показано на рис. 13.3, интерфейс изображается в форме пиктограммы. Компонент Образ.java, который реализует интерфейс, соединяется со значком интерфейса (кружком) НаблюдательОбраза простой линией. Компонент РыцарьПечальногоОбраза.jaya, который использует интерфейс, связан с ним отношением зависимости.

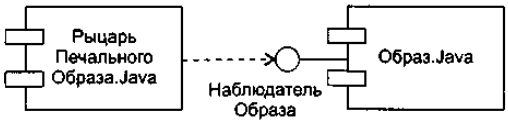

Рис. 13.3. Представление интерфейса в форме пиктограммы

Второй способ представления интерфейса иллюстрирует рис. 13.4. Здесь используется развернутая форма изображения интерфейса, в которой могут показываться его операции. Компонент, который реализует интерфейс, подключается к нему отношением реализации. Компонент, который получает доступ к услугам другого компонента через интерфейс, по-прежнему подключается к интерфейсу отношением зависимости.

Рис. 13.4. Развернутая форма представления интерфейса

По способу связи компонента с интерфейсом различают:

- экспортируемый интерфейс — тот, который компонент реализует и предлагает как услугу клиентам;
- импортируемый интерфейс — тот, который компонент использует как услугу другого компонента.

У одного компонента может быть несколько экспортируемых и несколько импортируемых интерфейсов.

Тот факт, что между двумя компонентами всегда находится интерфейс, устраняет их прямую зависимость. Компонент, использующий интерфейс, будет функционировать правильно вне зависимости от того, какой компонент реализует этот интерфейс. Это очень важно и обеспечивает гибкую замену компонентов в интересах развития системы.

Компоновка системы

За последние полвека разработчики аппаратуры прошли путь от компьютеров размером с комнату до крошечных «ноутбуков», обеспечивших возросшие функциональные возможности. За те же полвека разработчики программного обеспечения прошли путь от больших систем на Ассемблере и Фортране до еще больших систем на C++ и Java. Увы, но программный инструментарий развивается медленнее, чем аппаратный инструментарий. В чем главный секрет аппаратчиков? — спросят у аппаратчика-мальчиша программеры-буржуины.

Этот секрет — компоненты. Разработчик аппаратуры создает систему из готовых аппаратных компонентов (микросхем), выполняющих определенные функции и предоставляющих набор услуг через ясные интерфейсы. Задача конструкторов упрощается за счет повторного использования результатов, полученных другими.

Повторное использование — магистральный путь развития программного инструментария. Создание нового ПО из существующих, работоспособных программных компонентов приводит к более надежному и дешевому коду. При этом сроки разработки существенно сокращаются.

Основная цель программных компонентов — допускать сборку системы из двоичных заменяемых частей. Они должны обеспечить начальное создание системы из компонентов, а затем и ее развитие — добавление новых компонентов и замену некоторых старых компонентов без перестройки системы в целом. Ключ к воплощению такой возможности — интерфейсы. После того как интерфейс определен, к выполняемой системе можно подключить любой компонент, который удовлетворяет ему или обеспечивает этот интерфейс. Для расширения системы производятся компоненты, которые обеспечивают дополнительные услуги через новые интерфейсы. Такой подход основывается на особенностях компонента, перечисленных в табл. 13.2.

Таблица 13.2. Особенности компонента

Компонент физичен. Он живет в мире битов, а не логических понятий и не зависит от языка программирования
Компонент — заменяемый элемент. Свойство заменяемости позволяет заменить один компонент другим компонентом, который удовлетворяет тем же интерфейсам. Механизм замены оговорен современными компонентными моделями (COM, COM+, CORBA, Java Beans), требующими незначительных преобразований или предоставляющими утилиты, которые автоматизируют механизм
Компонент является частью системы, он редко автономен. Чаще компонент сотрудничает с другими компонентами и существует в архитектурной или технологической среде, предназначенной для его использования. Компонент связан и физически, и логически, он обозначает фрагмент большой системы

Компонент соответствует набору интерфейсов и обеспечивает реализацию этого набора интерфейсов

Вывод: компоненты — базисные строительные блоки, из которых может проектироваться и составляться система. Компонент может появляться на различных уровнях иерархии представления сложной системы. Система на одном уровне абстракции может стать простым компонентом на более высоком уровне абстракции.

Разновидности компонентов

Мир современных компонентов достаточно широк и разнообразен. В языке UML для обозначения новых разновидностей компонентов используют механизм стереотипов. Стандартные стереотипы, предусмотренные в UML для компонентов, представлены в табл. 13.3.

Таблица 13.3. Разновидности компонентов

Стереотип	Описание
«executable»	Компонент, который может выполняться в физическом узле (имеет расширение .exe)
«library»	Статическая или динамическая объектная библиотека (имеет расширение .dll)
«file»	Компонент, который представляет файл, содержащий исходный код или данные (имеет расширение .ini)
«table»	Компонент, который представляет таблицу базы данных (имеет расширение .tbl)
«document»	Компонент, который представляет документ (имеет расширение .hip)

В языке UML не определены пиктограммы для перечисленных стереотипов, применяемые на практике пиктограммы компонентов показаны на рис. 13.5-13.9.

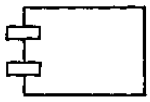

Рис. 13.5. Пиктограмма исполняемого элемента библиотеки

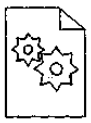

Рис. 13.6. Пиктограмма объектной

Рис. 13.7. Пиктограмма документа с исходным кодом или данными базы данных

Рис. 13.8. Пиктограмма таблицы

Рис. 13.9. Пиктограмма документа

Использование компонентных диаграмм

Компонентные диаграммы используют для моделирования статического представления реализации системы. Это представление поддерживает управление конфигурацией системы, составляемой из компонентов. Подразумевается, что для получения работающей системы существуют различные способы сборки компонентов.

Компонентные диаграммы показывают отношения:

- периода компиляции (среди текстовых компонентов);
- периода сборки, линковки (среди объектных двоичных компонентов);
- периода выполнения (среди машинных компонентов).

Рассмотрим типовые варианты применения компонентных диаграмм.

Моделирование программного текста системы

При разработке сложных систем программный текст (исходный код) разбросан по многим файлам исходного кода. При

использовании Java исходный код сохраняется в .java-файлах, при использовании C++ — в заголовочных файлах (.h-фай-лах) и телах (.cpp-файлах), при использовании Ada 95 — в спецификациях (.ads-файлах) и реализациях (.adb-файлах).

Между файлами существуют многочисленные зависимости компиляции. Если к этому добавить, что по мере разработки рождаются новые версии файлов, то становится очевидной необходимость управления конфигурацией системы, визуализации компиляционных зависимостей.

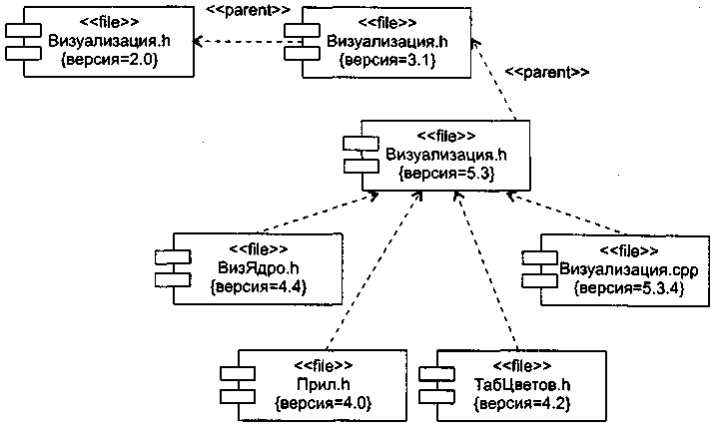

Рис. 13.10. Моделирование исходного кода

Рис. 13.11. Моделирование исходного кода с использованием пиктограмм

В качестве примера на рис. 13.10 приведена компонентная диаграмма, где изображены файлы исходного кода, используемые для построения библиотеки Визуализация.dll. Имеются четыре заголовочных файла (Визуализация.h, ВизЯдро.h, Прил.h, ТабЦветов.h), которые представляют исходный код для спецификации определенных классов. Файл реализации здесь один (Визуализация.cpp), он является реализацией одного из заголовков. Отметим, что для каждого файла явно указана его версия, причем для файла Визуализация.h показаны три версии и история их появления. На рис. 13.11 повторяется та же диаграмма, но здесь для обозначения компонентов использованы специальные пиктограммы.

Моделирование реализации системы

Реализация системы может включать большое количество разнообразных компонентов:

- исполняемых элементов;
- динамических библиотек;
- файлов данных;
- справочных документов;
- файлов инициализации;
- файлов регистрации;
- сценариев;
- файлов установки.

Моделирование этих компонентов, отношений между ними — важная часть управления конфигурацией системы.

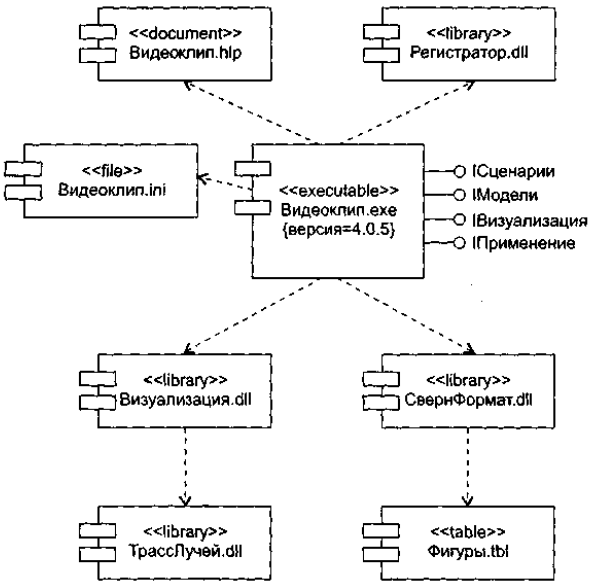

Рис. 13.12. Моделирование реализации системы

Например, на рис. 13.12 показана часть реализации системы, группируемая вокруг исполняемого элемента Видеоклип.exe. Здесь изображены четыре библиотеки (Регистратор.dll, СвернФормат.dll, Визуализация.dll, ТрассЛучей.dll), один документ (Видеоклип.hlp), один простой файл (Видеоклип.ini),атакже таблица базы данных (Фигуры.tbl). В диаграмме указаны отношения зависимости, существующие между компонентами.

Для исполняемого компонента Видеоклип.exe указан номер версии (с помощью пгеговой величины), представлены его экспортируемые интерфейсы (ICценарии, IВизуализация, IModели, IПрименение). Эти интерфейсы образуют API компонента «интерфейс прикладного программирования).

На рис. 13.13 повторяется та же диаграмма, моделирующая реализацию, но здесь для обозначения компонентов использованы специальные пиктограммы.

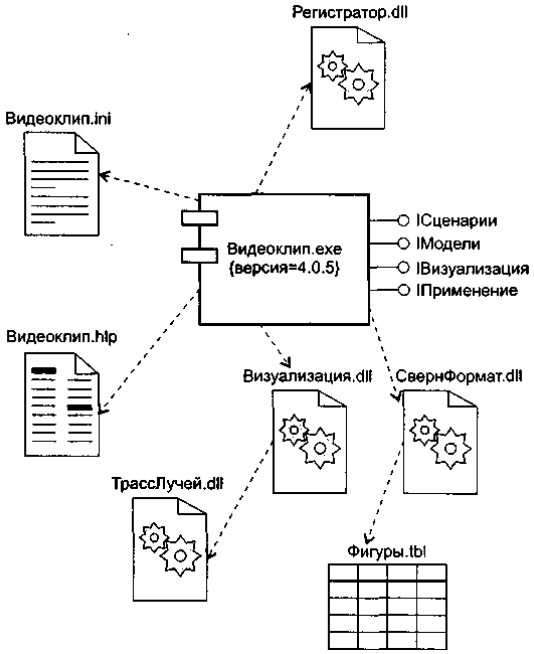

Рис. 13.13. Моделирование реализации с использованием пиктограмм

МЕТРИКИ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ПРОГРАММНЫХ СИСТЕМ

При конструировании объектно-ориентированных программных систем значительная часть затрат приходится на создание визуальных моделей. Очень важно корректно и всесторонне оценить качество этих моделей, сопоставив качеству числовую оценку. Решение данной задачи требует введения специального метрического аппарата. Такой аппарат развивает идеи классического оценивания сложных программных систем, основанного на метриках сложности, связности и сцепления. Вместе с тем он учитывает специфические особенности объектно-ориентированных решений. В этой главе обсуждаются наиболее известные объектно-ориентированные метрики, а также описывается методика их применения.

Метрические особенности объектно-ориентированных программных систем

Объектно-ориентированные метрики вводятся с целью:

- улучшить понимание качества продукта;
- оценить эффективность процесса конструирования;
- улучшить качество работы на этапе проектирования.

Все эти цели важны, но для программного инженера главная цель — повышение качества продукта. Возникает вопрос — как измерить качество объектно-ориентированной системы?

Для любого инженерного продукта метрики должны ориентироваться на его уникальные характеристики. Например, для электропоезда вряд ли полезна метрика «расход угля на километр пробега». С точки зрения метрик выделяют пять характеристик объектно-ориентированных систем: локализацию, инкапсуляцию, информационную закрытость, наследование и способы абстрагирования объектов. Эти характеристики оказывают максимальное влияние на объектно-ориентированные метрики.

Локализация

Локализация фиксирует способ группировки информации в программе. В классических методах, где используется функциональная декомпозиция, информация локализуется вокруг функций. Функции в них реализуются как процедурные модули. В методах, управляемых данными, информация группируется вокруг структур данных. В объектно-ориентированной среде информация группируется внутри классов или объектов (инкапсуляцией как данных, так и процессов).

Поскольку в классических методах основной механизм локализации — функция, программные метрики ориентированы на внутреннюю структуру или сложность функций (длина модуля, связность, цикломатическая сложность) или на способ, которым функции связываются друг с другом (сцепление модулей).

Так как в объектно-ориентированной системе базовым элементом является класс, то локализация здесь основывается на объектах. Поэтому метрики должны применяться к классу (объекту) как к комплексной сущности. Кроме того, между операциями (функциями) и классами могут быть отношения не только «один-к-одному». Поэтому метрики, отображающие способы взаимодействия классов, должны быть приспособлены к отношениям «один-ко-многим», «многие-ко-многим».

Инкапсуляция

Вспомним, что инкапсуляция — упаковка (связывание) совокупности элементов. Для классических ПС примерами низкоуровневой инкапсуляции являются записи и массивы. Механизмом инкапсуляции среднего уровня являются подпрограммы (процедуры, функции).

В объектно-ориентированных системах инкапсулируются обязанности класса, представляемые его свойствами (а для агрегатов — и свойствами других классов), операциями и состояниями.

Для метрик учет инкапсуляции приводит к смещению фокуса измерений с одного модуля на группу свойств и обрабатывающих модулей (операций). Кроме того, инкапсуляция переводит измерения на более высокий уровень абстракции (пример — метрика «количество операций на класс»). Напротив, классические метрики ориентированы на низкий уровень — количество булевых условий (цикломатическая сложность) и количество строк программы.

Информационная закрытость

Информационная закрытость делает невидимыми операционные детали программного компонента. Другим компонентам доступна только необходимая информация.

Качественные объектно-ориентированные системы поддерживают высокий уровень информационной закрытости. Таким образом, метрики, измеряющие степень достигнутой закрытости, тем самым отображают качество объектно-ориентированного проекта.

Наследование

Наследование — механизм, обеспечивающий тиражирование обязанностей одного класса в другие классы. Наследование распространяется через все уровни иерархии классов. Стандартные ПС не поддерживают эту характеристику.

Поскольку наследование — основная характеристика объектно-ориентированных систем, на ней фокусируются многие объектно-ориентированные метрики (количество детей — потомков класса, количество родителей, высота класса в иерархии наследования).

Абстракция

Абстракция — это механизм, который позволяет проектировщику выделять главное в программном компоненте (как свойства, так и операции) без учета второстепенных деталей. По мере перемещения на более высокие уровни абстракции мы игнорируем все большее количество деталей, обеспечивая все более общее представление понятия или элемента. По мере перемещения на более низкие уровни абстракции мы вводим все большее количество деталей, обеспечивая более удачное

представление понятия или элемента.

Класс — это абстракция, которая может быть представлена на различных уровнях детализации и различными способами (например, как список операций, последовательность состояний, последовательности взаимодействий). Поэтому объектно-ориентированные метрики должны представлять абстракции в терминах измерений класса. Примеры: количество экземпляров класса в приложении, количество родовых классов на приложение, отношение количества родовых к количеству неродовых классов.

Эволюция мер связи для объектно-ориентированных программных систем

В разделах «Связность модуля» и «Сцепление модулей» главы 4 было показано, что классической мерой сложности внутренних связей модуля является связность, а классической мерой сложности внешних связей — сцепление. Рассмотрим развитие этих мер применительно к объектно-ориентированным системам.

Связность объектов

В классическом методе Л. Констентайна и Э. Йордана определены семь типов связности.

1. ***Связность по совпадению.*** В модуле отсутствуют явно выраженные внутренние связи.
2. ***Логическая связность.*** Части модуля объединены по принципу функционального подобия.
3. ***Временная связность.*** Части модуля не связаны, но необходимы в один и тот же период работы системы.
4. ***Процедурная связность.*** Части модуля связаны порядком выполняемых ими действий, реализующих некоторый сценарий поведения.
5. ***Коммуникативная связность.*** Части модуля связаны по данным (работают с одной и той же структурой данных).
6. ***Информационная (последовательная) связность.*** Выходные данные одной части используются как входные данные в другой части модуля.
7. ***Функциональная связность.*** Части модуля вместе реализуют одну функцию.

Этот метод функционален по своей природе, поэтому наибольшей связностью здесь объявлена функциональная связность. Вместе с тем одним из принципиальных преимуществ объектно-ориентированного подхода является естественная связанность объектов.

Максимально связанным является объект, в котором представляется единая сущность и в который включены все операции над этой сущностью. Например, максимально связанным является объект, представляющий таблицу символов компилятора, если в него включены все функции, такие как «Добавить символ», «Поиск в таблице» и т. д.

Следовательно, восьмой тип связности можно определить так:

8. ***Объектная связность.*** Каждая операция обеспечивает функциональность, которая предусматривает, что все свойства объекта будут модифицироваться, отображаться и использоваться как базис для предоставления услуг.

Высокая связность — желательная характеристика, так как она означает, что объект представляет единую часть в проблемной области, существует в едином пространстве. При изменении системы все действия над частью инкапсулируются в едином компоненте. Поэтому для производства изменения нет нужды модифицировать много компонентов.

Если функциональность в объектно-ориентированной системе обеспечивается наследованием от суперклассов, то связность объекта, который наследует свойства и операции, уменьшается. В этом случае нельзя рассматривать объект как отдельный модуль — должны учитываться все его суперклассы. Системные средства просмотра содействуют такому учету. Однако понимание элемента, который наследует свойства от нескольких суперклассов, резко усложняется.

Обсудим конкретные метрики для вычисления связности классов.

Метрики связности по данным

Л. Отт и Б. Мехра разработали модель секционирования класса [55]. Секционирование основывается на экземплярных переменных класса. Для каждого метода класса получают ряд секций, а затем производят объединение всех секций класса. Измерение связности основывается на количестве лексем данных (data tokens), которые появляются в нескольких секциях и «склеивают» секции в модуль. Под лексемами данных здесь понимают определения констант и переменных или ссылки на константы и переменные.

Базовым понятием методики является секция данных. Она составляется для каждого выходного параметра метода. *Секция данных* — это последовательность лексем данных в операторах, которые требуются для вычисления этого параметра.

Например, на рис. 14.1 представлен программный текст метода SumAndProduct. Все лексемы, входящие в секцию переменной SumN, выделены рамками. Сама секция для SumN записывается как следующая последовательность лексем:

$N_1 \bullet SumN_1 \bullet I_1 \bullet SumN_2 \bullet O_1 \bullet I_2 \bullet 1_2 \bullet N_2 \bullet SumN_3\ SumN_4 \bullet I_3$.

```
procedure SumAndProduct
(N : integer;
 var SumN, ProdN : integer );
 var
    I : integer;
begin
    SumN := 0 ;
    ProdN := 1;
    for I := 1 to N do begin
                          SumN := SumN + I ;
                          ProdN := ProdN * I
                       end
end;
```

Рис. 14.1. Секция данных для переменной SumN

Заметим, что индекс в «l_2» указывает на второе вхождение лексемы «l» в текст метода. Аналогичным образом определяется секция для переменной ProdN:

$N_1 \bullet ProdN_1 \bullet I_1 \bullet ProdN_2 \bullet l_1 \bullet I_2 \bullet l_2 \bullet N_2 \bullet ProdN_3 \bullet ProdN_4 \bullet I_4$

Для определения отношений между секциями данных можно показать профиль секций данных в методе. Для нашего примера профиль секций данных приведен в табл. 14.1.

Таблица 14.1. Профиль секций данных для метода SumAndProduct

SumN	ProdN	Оператор
		procedure SumAndProduct
1	1	(Niinteger;
1	1	varSumN, ProdNiinteger)
		var
1	1	l:integer;
		begin
2		SumN:=0
	2	ProdN:=1
3	3	for l:=1 to N do begin
3		SumN:=SumN+l
	3	ProdN:=ProdN*l
		end
		end;

Видно, что в столбце переменной для каждой секции указывается количество лексем из *i*-й строки метода, которые включаются в секцию.

Еще одно базовое понятие методики — секционированная абстракция. *Секционированная абстракция* — это объединение всех секций данных метода. Например, секционированная абстракция метода SumAndProduct имеет вид

$$SA(SumAndProduct) = \{N_1 \cdot SumN_1 \cdot I_1 \cdot SumN_2 \cdot 0_1 \cdot I_2 \cdot I_2 \cdot N_2 \cdot SumN_3 \cdot SumN_4 \cdot I_3,$$
$$N_1 \cdot ProdN_1 \cdot I_1 \cdot ProdN_2 \cdot I_1 \cdot I_2 \cdot I_2 \cdot N_2 \cdot ProdN_3 \cdot ProdN_4 \cdot I_4\}.$$

Введем главные определения.

Секционированной абстракцией класса (Class Slice Abstraction) CSA(C) называют объединение секций всех экземплярных переменных класса. Полный набор секций составляют путем обработки всех методов класса.

Склеенными лексемами называют те лексемы данных, которые являются элементами более чем одной секции данных.

Сильно склеенными лексемами называют те склеенные лексемы, которые являются элементами всех секций данных.

Сильная связность по данным (StrongData Cohesion) — это метрика, основанная на количестве лексем данных, входящих во все секции данных для класса. Иначе говоря, сильная связность по данным учитывает количество сильно склеенных лексем в классе *C,* она вычисляется по формуле:

$$\mathrm{SDC}(C) = \frac{|SG(\mathrm{CSA}(C))|}{|\text{лексемы}(C)|},$$

где *SG*(CSA(*C*)) — объединение сильно склеенных лексем каждого из методов класса *C,* лексемы*(C)* — множество всех лексем данных класса *C.*

Таким образом, класс без сильно склеенных лексем имеет нулевую сильную связность по данным.

Слабая связность по данным (Weak Data Cohesion) — метрика, которая оценивает связность, базируясь на склеенных лексемах. Склеенные лексемы не требуют связывания всех секций данных, поэтому данная метрика определяет более слабый тип связности. Слабая связность по данным вычисляется по формуле:

$$\mathrm{WDC}(C) = \frac{|G(\mathrm{CSA}(C))|}{|\text{лексемы}(C)|},$$

где *G*(CSA(C)) — объединение склеенных лексем каждого из методов класса. Класс без склеенных лексем не имеет слабой связности по данным. Наиболее точной метрикой связности между секциями данных является *клейкость данных (Data Adhesiveness).* Клейкость данных определяется как отношение суммы из количеств секций, содержащих каждую склеенную лексему, к произведению количества лексем данных в классе на количество секций данных. Метрика вычисляется по формуле:

$$\mathrm{DA}(C) = \frac{\sum d \in G(\mathrm{CSA}(C)"d \in \text{Секции}}{|\text{лексемы}(C)| \times |\mathrm{CSA}(C)|}.$$

Приведем пример. Применим метрики к классу, профиль секций которого показан в табл. 14.2.

Таблица 14.2. Профиль секций данных для класса Stack

array top size	Класс Stack
	class Stack {int *array, top, size;
	public:

			Stack (int *s*) {
2 2			size=s;
2 2			array=new int [size];
2			top=0;}
			int IsEmpty () {
2			return top==0};
			int Size (){
2			return size};
			intVtop(){
3 3			return array [top-1]; }
			void Push (int item) {
2	2	2	if (top= =size)
			printf ("Empty stack. \n");
			else
3	3	3	array [top++]=item;}
			int Pop () {
1			if (IsEmpty ())
			printf ("Full stack. \n");
			else
1			--top;}
			};

Очевидно, что CSA(Stack) включает три секции с 19 лексемами, имеет 5 сильно склеенных лексем и 12 склеенных лексем. Расчеты по рассмотренным метрикам дают следующие значения:

SDC(CSA(Stack)) = 5/19 = 0,26

WDC(CSA(Stack)) = 12/19 = 0,63

DA(CSA(Stack)) =(7*2 + 5*3)/(19*3) = 0,51

УНИФИЦИРОВАННЫЙ ПРОЦЕСС РАЗРАБОТКИ ОБЪЕКТНО-ОРИЕНТИРОВАННЫХ ПС

В первой главе рассматривались основы организации процессов разработки ПО. В данной главе внимание сосредоточено на детальном обсуждении унифицированного процесса разработки объектно-ориентированного ПО, на базе которого возможно построение самых разнообразных схем конструирования программных приложений. Далее описывается содержание XP-процесса экстремальной разработки, являющегося носителем адаптивной технологии, применяемой в условиях частого изменения требований заказчика.

Эволюционно-инкрементная организация жизненного цикла разработки

Рассматриваемый подход является развитием спиральной модели Боэма [8], [40], [44], [57]. В этом случае процесс разработки программной системы организуется в виде эволюционно-инкрементного жизненного цикла. Эволюционная составляющая цикла основывается на доопределении требований в ходе работы, инкрементная составляющая — на планомерном приращении реализации требований.

В этом цикле разработка представляется как серия итераций, результаты которых развиваются от начального макета до конечной системы. Каждая итерация включает сбор требований, анализ, проектирование, реализацию и тестирование. Предполагается, что вначале известны не все требования, их дополнение и изменение осуществляется на всех итерациях жизненного цикла. Структура типовой итерации показана на рис. 15.1.

Видно, что критерием управления этим жизненным циклом является уменьшение риска. Работа начинается с оценки начального риска. В ходе выполнения каждой итерации риск пересматривается. Риск связывается с каждой итерацией так, что ее успешное завершение уменьшает риск. План последовательности реализаций гарантирует, что наибольший риск устраняется в первую очередь.

Такая методика построения системы нацелена на выявление и уменьшение риска в самом начале жизненного цикла. В итоге минимизируются затраты на уменьшение риска.

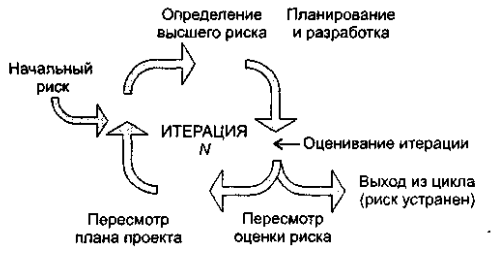

Рис. 15.1. Типовая итерация эволюционно-инкрементного жизненного цикла

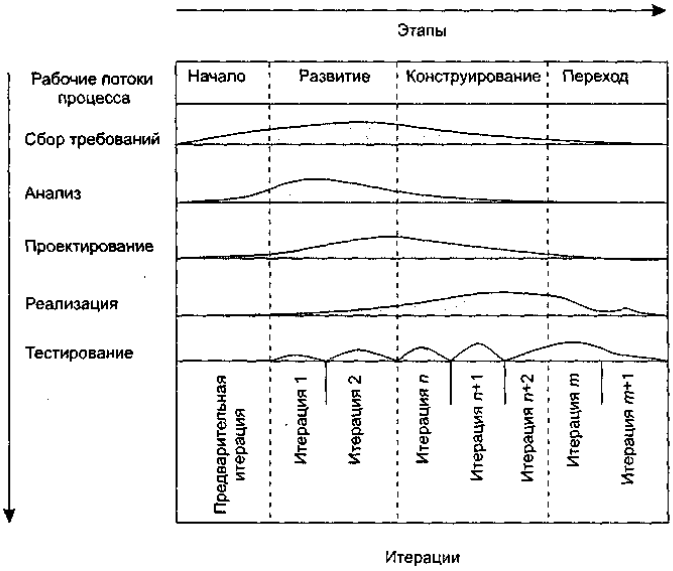

Рис. 15.2. Два измерения унифицированного процесса разработки

Как показано на рис. 15.2, в структуре унифицированного процесса разработки выделяют два измерения:

- горизонтальная ось представляет время и демонстрирует характеристики жизненного цикла процесса;
- вертикальная ось представляет рабочие потоки процесса, которые являются логическими группировками действий.

Первое измерение задает динамический аспект развития процесса в терминах циклов, этапов, итераций и контрольных вех. Второе измерение задает статический аспект процесса в терминах компонентов процесса, рабочих потоков, приводящих к выработке искусственных объектов (артефактов), и участников.

Этапы и итерации

По времени в жизненном цикле процесса выделяют четыре этапа:

- начало (Inception) — спецификация представления продукта;
- развитие (Elaboration) — планирование необходимых действий и требуемых ресурсов;
- конструирование (Construction) — построение программного продукта в виде серии инкрементных итераций;
- переход (Transition) — внедрение программного продукта в среду пользователя (промышленное производство, доставка и применение).

В свою очередь, каждый этап процесса разделяется на итерации. Итерация — это полный цикл разработки, вырабатывающий промежуточный продукт. По мере перехода от итерации к итерации промежуточный продукт инкрементно усложняется, постепенно превращаясь в конечную систему. В состав каждой итерации входят все рабочие потоки — от сбора требований до тестирования. От итерации к итерации меняется лишь удельный вес каждого рабочего потока — он зависит от этапа. На этапе Начало основное внимание уделяется сбору требований, на этапе Развитие — анализу и проектированию, на этапе Конструирование — реализации, на этапе Переход — тестированию. Каждый этап и итерация уменьшают некоторый риск и завершается контрольной вехой. К вехе привязывается техническая проверка степени достижения ключевых целей. По результатам проверки возможна модификация дальнейших действий.

Рабочие потоки процесса

Рабочие потоки процесса имеют следующее содержание:

- Сбор требований — описание того, что система должна делать;
- Анализ — преобразование требований к системе в классы и объекты, выявляемые в предметной области;
- Проектирование — создание статического и динамического представления системы, выполняющего выявленные требования и являющегося эскизом реализации;
- Реализация — производство программного кода, который превращается в исполняемую систему;
- Тестирование — проверка всей системы в целом.

Каждый рабочий поток определяет набор связанных артефактов и действий. Артефакт — это документ, отчет или выполняемый элемент. Артефакт может вырабатываться, обрабатываться или потребляться. Действие описывает задачи — шаги обдумывания, шаги исполнения и шаги проверки. Шаги выполняются участниками процесса (для создания или модификации артефактов).

Между артефактами потоков существуют зависимости. Например, модель Use Case, генерируемая в ходе сбора требований, уточняется моделью анализа из процесса анализа, обеспечивается проектной моделью из процесса проектирования, реализуется моделью реализации из процесса реализации и проверяется тестовой моделью из процесса тестирования.

Модели

Модель — наиболее важная разновидность артефакта. Модель упрощает реальность, создается для лучшего понимания разрабатываемой системы. Предусмотрены девять моделей, вместе они покрывают все решения по визуализации, спецификации, конструированию и документированию программных систем:

- бизнес-модель. Определяет абстракцию организации, для которой создается система;
- модель области определения. Фиксирует контекстное окружение системы;
- модель Use Case. Определяет функциональные требования к системе;
- модель анализа. Интерпретирует требования к системе в терминах проектной модели;
- проектная модель. Определяет словарь проблемы и ее решение;
- модель размещения. Определяет аппаратную топологию, в которой исполняется система;
- модель реализации. Определяет части, которые используются для сборки и реализации физической системы;
- тестовая модель. Определяет тестовые варианты для проверки системы;
- модель процессов. Определяет параллелизм в системе и механизмы синхронизации.

Технические артефакты

Технические артефакты подразделяются на четыре основных набора:

- набор требований. Описывает, что должна делать система;
- набор проектирования. Описывает, как должна быть сконструирована система;
- набор реализации. Описывает сборку разработанных программных компонентов;
- набор размещения. Обеспечивает всю информацию о поставляемой конфигурации.

Набор требований группирует всю информацию о том, что система должна делать. Он может включать модель Use Case, модель нефункциональных требований, модель области определения, модель анализа, а также другие формы выражения нужд пользователя.

Набор проектирования группирует всю информацию о том, как будет конструироваться система при учете всех ограничений (времени, бюджета, традиций, повторного использования, качества и т.д.).

Он может включать проектную модель, тестовую модель и другие формы выражения сущности системы (например, макеты).

Набор реализации группирует все данные о программных элементах, образующих систему (программный код, файлы конфигурации, файлы данных, программные компоненты, информацию о сборке системы).

Набор размещения группирует всю информацию об упаковке, отправке, установке и запуске системы.