Министерство образования Республики Беларусь

Учреждение образования
«Гомельский государственный университет
имени Франциска Скорины»

А. С. ПОМАЗ

СИСТЕМЫ АВТОМАТИЗИРОВАННОГО ПРОЕКТИРОВАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Тексты лекций

Гомель
УО «ГГУ им. Ф. Скорины»
2009

Министерство образования Республики Беларусь

Учреждение образования
«Гомельский государственный университет
имени Франциска Скорины»

А. С. ПОМАЗ

СИСТЕМЫ АВТОМАТИЗИРОВАННОГО ПРОЕКТИРОВАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Тексты лекций

для студентов математических специальностей

Гомель
УО «ГГУ им. Ф. Скорины»
2009

УДК 681.3.06
ББК 32.973.26
П 55

Рецензенты:
В. Д. Левчук, доцент, кандидат технических наук;
кафедра математических проблем управления учреждения образования «Гомельский государственный университет имени Франциска Скорины».

Рекомендовано к изданию научно-методическим советом учреждения образования «Гомельский государственный университет имени Франциска Скорины»

Помаз, А. С.

П 55 Системы автоматизированного проектирования программного обеспечения : тексты лекций для студентов математических специальностей / А. С. Помаз; М-во образования РБ, Гомельский государственный университет им. Ф. Скорины. – Гомель : ГГУ им. Ф. Скорины, 2009. – 110 с.

ISBN 978-985-439-371-1

В настоящее время в связи с повышением требований к качеству программного обеспечения все большую актуальность приобретает проектирование программных комплексов. В связи с этим получение знаний и формирование навыков проектирования программного обеспечения является неотъемлемой частью подготовки специалистов в области разработки программных средств. Данные тексты лекций ставят своей целью оказание помощи студентам в усвоении и практическом применении знаний по анализу и проектированию программных комплексов.

Адресованы студентам специальностей 1-40 01 01 «Программное обеспечение информационных технологий», 1-31 03 03 01 – «Прикладная математика (научно-производственная деятельность)».

УДК 681.3.06
ББК 32.973.26

ISBN 978–985–439–371–1

Содержание

Введение

В настоящее время в инженерии программного обеспечения актуальными стали следующие проблемы:

– аппаратная сложность опережает возможности построения программного обеспечения (в дальнейшем ПО), использующего потенциальные возможности аппаратуры;

– возможности разработки программ отстают от требований к новому ПО;

– 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 (в дальнейшем ПС). Одним из важных этапов создания ПО является проектирование, обеспечивающее качество создаваемых программных продуктов.

Тексты лекций ставят своей целью оказание помощи студентам в овладении знаниями по проектированию программного обеспечения и усвоении навыков по проектированию программных комплексов с использованием языка UML.

1 Организация процесса конструирования

Тема 1 Определение технологии конструирования программного обеспечения

1.1 Определение технологии конструирования программного обеспечения

1.1 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 [1], [2].

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.

Тема 2 Классический жизненный цикл

2.1 Классический жизненный цикл
2.2 Макетирование

2.1 Классический жизненный цикл

Старейшей парадигмой процесса разработки ПО является классический жизненный цикл (автор Уинстон Ройс, 1970) [2].

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 (рисунок 2.1).

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». Необходи-

мость 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.

Проектирование состоит в создании представлений:

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

Рисунок 2.1 – Классический жизненный цикл разработки ПО

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 согласно требованиям заказчика.

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

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

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

Недостатки классического жизненного цикла:

1) реальные проекты часто требуют отклонения от стандартной последовательности шагов;

2) цикл основан на точной формулировке исходных требований к ПО (реально в начале проекта требования заказчика определены лишь частично);

3) результаты проекта доступны заказчику только в конце работы.

2.2 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 адаптации 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 рисунке 2.2, макетирование основывается на многократном повторении итераций, в которых участвуют заказчик и разработчик.

Рисунок 2.2 – Макетирование

Последовательность действий при макетировании представлена на рисунке 2.3.

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.

Рисунок 2.3 – Последовательность действий при макетировании

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.

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.

Тема 3 Стратегии конструирования ПО

3.1 Инкрементная модель
3.2 Быстрая разработка приложений
3.3 Спиральная модель
3.4 Компонентно-ориентированная модель

3.1 Инкрементная модель

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 таблице 3.1.

Таблица 3.1 – Характеристики стратегий конструирования

Стратегия конструирования	**В начале процесса определены все требования?**	**Множество циклов конструирования?**
Однократный проход	Да	Нет
Инкрементная (запланированное улучшение продукта)	Нет	Да
Эволюционная	Нет	Да

Инкрементная модель является классическим примером инкрементной стратегии конструирования (рисунок 3.1). 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.

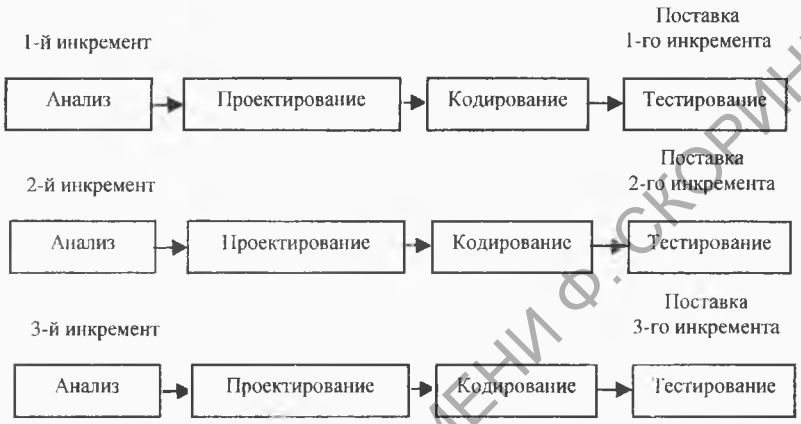

Рисунок 3.1 – Инкрементная модель

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

3.2 Быстрая разработка приложений

Модель быстрой разработки приложений (Rapid Application Development) — второй пример применения инкрементной стратегии конструирования (рисунок 3.2).

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).

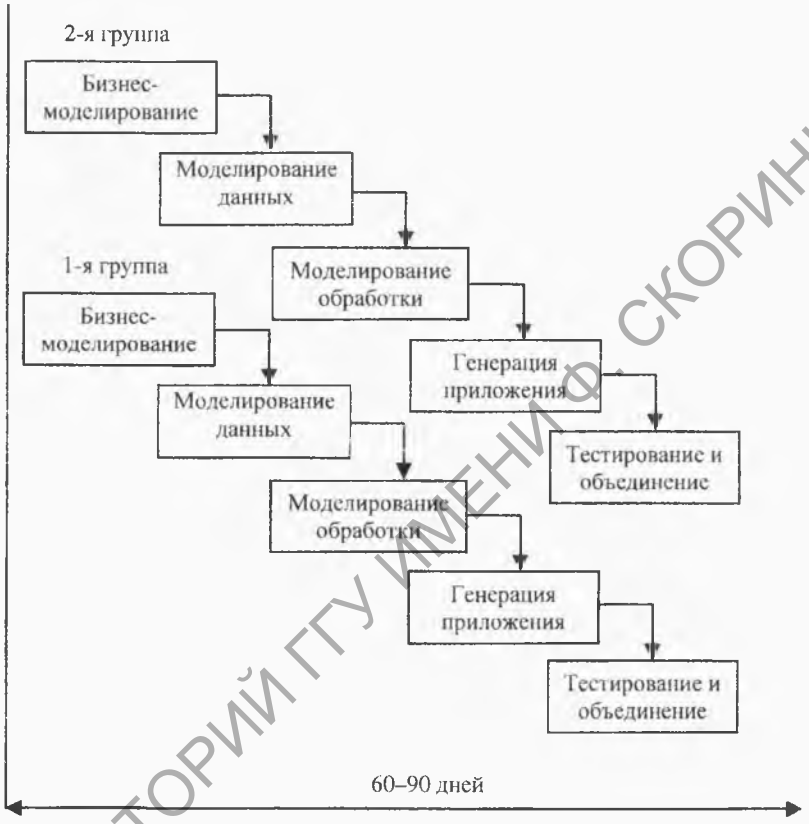

Рисунок 3.2 – Модель быстрой разработки приложений

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, определяют-

ся 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).

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).

3.3 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.

Как показано на рисунке 3.3, 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.

Рисунок 3.3 – 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

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 оцени-

вает 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.

3.4 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 (рисунок 3.4).

Рисунок 3.4 – 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.

Тема 4 XP-процесс

4.1 Тяжеловесные и облегченные процессы
4.2 XP-процесс
4.3 Модели качества процессов конструирования

4.1 Тяжеловесные и облегченные процессы

В 21 веке потребности общества в программном обеспечении информационных технологий достигли экстремальных значений. Программная индустрия не может справиться от потока самых разнообразных заказов.

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. Иными словами, человеческие слабости в расчет не принимаются, а объем необходимой документации во многих случаях огромен.

В последние годы появилась группа новых, облегченных (lightweight) процессов. Теперь их называют подвижными (agile) процессами. 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.

4.2 XP-процесс

Экстремальное программирование (eXtreme Programming, XP) — облегченный (подвижный) процесс (или методология), главный автор которого — Кент Бек (1999). 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.

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 упорядо-

ченном процессе. Просто в XP эти принципы, как показано в таблице 4.1, достигают «экстремальных значений».

Таблица 4.1 – 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	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 рисунке 4.1.

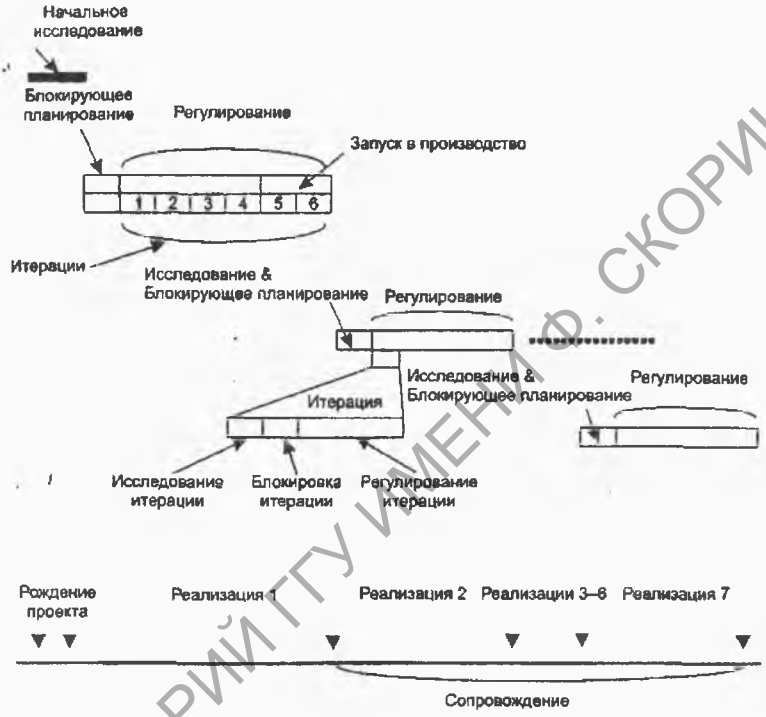

Рисунок 4.1 – Идеальный XP-процесс

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.

4.3 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. 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 CMM 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 организован-

ности процесса, но и даются рекомендации по дальнейшему совершенствованию процессов.

Очень важно отметить, что модель CMM ориентирована на построение системы постоянного улучшения процессов. В ней зафиксированы пять уровней зрелости (рисунок 4.2) и предусмотрен плавный, поэтапный подход к совершенствованию процессов — можно поэтапно получать подтверждения об улучшении процессов после каждого уровня зрелости.

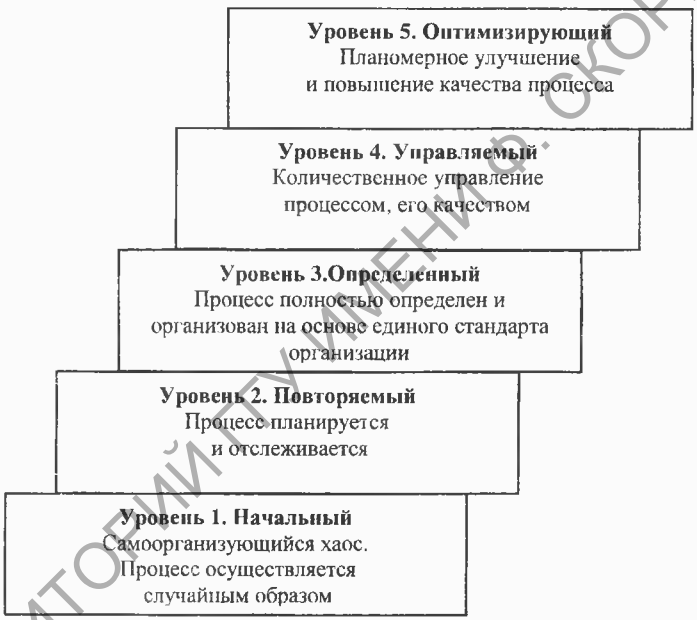

Рисунок 4.2 – Пять уровней зрелости модели CMM

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 соответ-

ствуют 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. 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.

2 Основы объектно-ориентированного представления программных систем

Тема 1 Базовые принципы объектно-ориентированного представления программных систем

1.1 Абстрагирование и инкапсуляция
1.2 Модульность
1.3 Иерархическая организация

1.1 Абстрагирование и инкапсуляция

Рассмотрение любой сложной системы требует применения техники декомпозиции — разбиения на составляющие элементы. Известны две схемы декомпозиции: алгоритмическая декомпозиция и объектно-ориентированная декомпозиция.

В основе алгоритмической декомпозиции лежит разбиение по действиям — алгоритмам. Эта схема представления применяется в обычных программных системах (ПС). Объектно-ориентированная декомпозиция обеспечивает разбиение по автономным лицам — объектам реального (или виртуального) мира. Эти лица (объекты) — более крупные элементы, каждый из них несет в себе и описания действий, и описания данных.

Объектно-ориентированное представление ПС основывается на принципах абстрагирования, инкапсуляции, модульности и иерархической организации. Каждый из этих принципов не нов, но их совместное применение рассчитано на проведение объектно-ориентированной декомпозиции. Это определяет модификацию их содержания и механизмов взаимодействия друг с другом. Обсудим данные принципы [1], [2], [3], [4], [5], [6], [7].

Аппарат абстракции — удобный инструмент для борьбы со сложностью реальных систем. Создавая понятие в интересах какой-либо задачи, мы отвлекаемся (абстрагируемся) от несущественных характеристик конкретных объектов, определяя только существенные характеристики. Например, в абстракции «часы» мы выделяем характеристику «показывать время», отвлекаясь от таких характеристик конкретных часов, как форма, цвет, материал, цена, изготовитель. Итак,

абстрагирование сводится к формированию абстракций. Каждая абстракция фиксирует основные характеристики объекта, которые отличают его от других видов объектов и обеспечивают ясные понятийные границы.

Абстракция концентрирует внимание на внешнем представлении объекта, позволяет отделить основное в поведении объекта от его реализации. Абстракцию удобно строить путем выделения обязанностей объекта.

Инкапсуляция и абстракция — взаимодополняющие понятия: абстракция выделяет внешнее поведение объекта, а инкапсуляция содержит и скрывает реализацию, которая обеспечивает это поведение. Инкапсуляция достигается с помощью информационной закрытости. Обычно скрываются структура объектов и реализация их методов.

Инкапсуляция является процессом разделения элементов абстракции на секции с различной видимостью. Инкапсуляция служит для отделения интерфейса абстракции от ее реализации.

1.2 Модульность

В языках C++, Object Pascal, Ada 95 абстракции классов и объектов формируют логическую структуру системы. При производстве физической структуры эти абстракции помещаются в модули. В больших системах, где классов сотни, модули помогают управлять сложностью. Модули служат физическими контейнерами, в которых объявляются классы и объекты логической разработки.

Модульность определяет способность системы подвергаться декомпозиции на ряд сильно связанных и слабо сцепленных модулей.

Общая цель декомпозиции на модули: уменьшение сроков разработки и стоимости ПС за счет выделения модулей, которые проектируются и изменяются независимо. Каждая модульная структура должна быть достаточно простой, чтобы быть полностью понятой. Изменение реализации модулей должно проводиться без знания реализации других модулей и без влияния на их поведение.

Определение классов и объектов выполняется в ходе логической разработки, а определение модулей — в ходе физической разработки системы. Эти действия сильно взаимосвязаны, осуществляются итеративно.

1.3 Иерархическая организация

Мы рассмотрели три механизма для борьбы со сложностью:

– абстракцию (она упрощает представление физического объекта);

– инкапсуляцию (закрывает детали внутреннего представления абстракций);

– модульность (дает путь группировки логически связанных абстракций).

Прекрасным дополнением к этим механизмам является иерархическая организация — формирование из абстракций иерархической структуры. Определением иерархии в проекте упрощаются понимание проблем заказчика и их реализация — сложная система становится обозримой человеком.

Иерархическая организация задает размещение абстракций на различных уровнях описания системы.

Двумя важными инструментами иерархической организации в объектно-ориентированных системах являются:

– структура из классов («is a»-иерархия);

– структура из объектов («part of»-иерархия).

Чаще всего «is a»-иерархическая структура строится с помощью наследования. Наследование определяет отношение между классами, где класс разделяет структуру или поведение, определенные в одном другом (единичное наследование) или в нескольких других (множественное наследование) классах.

Тема 2 Общая характеристика объектов

2.1 Общая характеристика объектов
2.2 Виды отношений между объектами

2.1 Общая характеристика объектов

Объект — это конкретное представление абстракции. Объект обладает индивидуальностью, состоянием и поведением. Структура и поведение подобных объектов определены в их общем классе. Термины «экземпляр класса» и «объект» взаимозаменяемы. На рисунке 2.1 приведен пример объекта по имени Стул, имеющего определенный набор свойств и операций.

Индивидуальность — это характеристика объекта, которая отличает его от всех других объектов.

Состояние объекта характеризуется перечнем всех свойств объекта и текущими значениями каждого из этих свойств (рисунок 2.1).

Рисунок 2.1 – Представление объекта с именем Стул

Объекты не существуют изолированно друг от друга. Они подвергаются воздействию или сами воздействуют на другие объекты.

Поведение характеризует то, как объект воздействует на другие объекты (или подвергается воздействию) в терминах изменений его состояния и передачи сообщений. Поведение объекта является функцией как его состояния, так и выполняемых им операций (купить, продать, взвесить, переместить, покрасить). Говорят, что состояние объекта представляет суммарный результат его поведения.

Операция обозначает обслуживание, которое объект предлагает своим клиентам. Возможны пять видов операций клиента над объектом:

1) модификатор (изменяет состояние объекта);
2) селектор (дает доступ к состоянию, но не изменяет его);
3) итератор (доступ к содержанию объекта по частям, в строго определенном порядке);
4) конструктор (создает объект и инициализирует его состояние);
5) деструктор (разрушает объект и освобождает занимаемую им

память).

Примеры операций приведены в таблице 2.1.

Таблица 2.1 – Разновидности операций

Вид операции	Пример операции
Модификатор	*пополнеть(кг)*
Селектор	*какойВес(): integer*
Итератор	*показатьАссортиментТоваров(): string*
Конструктор	*создатьРобот(параметры)*
Деструктор	*уничтожитьРобот()*

В чистых объектно-ориентированных языках программирования операции могут объявляться только как методы — элементы классов, экземплярами которых являются объекты. Гибридные языки (C++, Ada 95) позволяют писать операции как свободные подпрограммы (вне классов).

В общем случае все методы и свободные подпрограммы, ассоциированные с конкретным объектом, образуют его *протокол.* Таким образом, протокол определяет оболочку допустимого поведения объекта и поэтому заключает в себе цельное (статическое и динамическое) представление объекта.

Большой протокол полезно разделять на логические группировки поведения. Эти группировки, разделяющие пространство поведения объекта, обозначают *роли,* которые может играть объект. Принцип выделения ролей иллюстрирует рисунок 2.2.

С точки зрения внешней среды важное значение имеет такое понятие, как обязанности объекта. *Обязанности* означают обязательства объекта обеспечить определенное поведение. Обязанностями объекта являются все виды обслуживания, которые он предлагает клиентам. В мире объект играет определенные роли, выполняя свои обязанности.

В заключение отметим: наличие у объекта внутреннего состояния означает, что порядок выполнения им операций очень важен. Иначе говоря, объект может представляться как независимый автомат. По аналогии с автоматами можно выделять активные и пассивные объекты (рисунок 2.3).

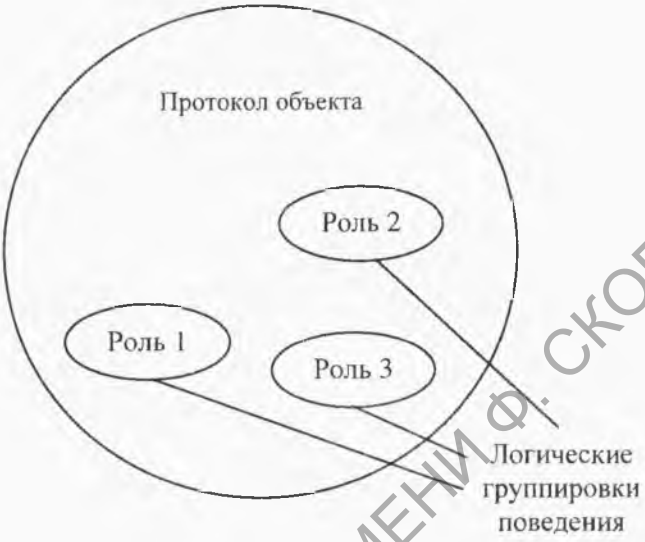

Рисунок 2.2 – Пространство поведения объекта

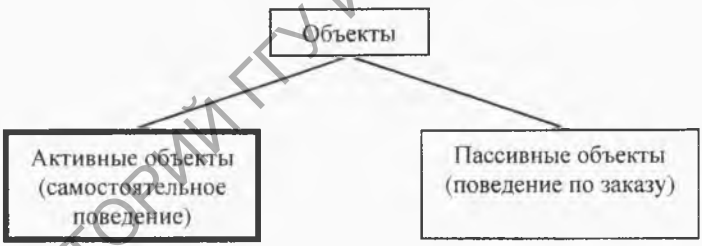

Рисунок 2.3 – Активные и пассивные объекты

Активный объект имеет собственный канал (поток) управления, пассивный — нет. Активный объект автономен, он может проявлять свое поведение без воздействия со стороны других объектов. Пассивный объект, наоборот, может изменять свое состояние только под воздействием других объектов.

2.2 Виды отношений между объектами

В поле зрения разработчика ПО находятся не объекты-одиночки, а взаимодействующие объекты, ведь именно взаимодействие объектов реализует поведение системы. У Г. Буча есть отличная цитата из Галла: «Самолет — это набор элементов, каждый из которых по своей природе стремится упасть на землю, но ценой совместных непрерывных усилий преодолевает эту тенденцию». Отношения между парой объектов основываются на взаимной информации о разрешенных операциях и ожидаемом поведении. Особо интересны два вида отношений между объектами: связи и агрегация.

Связь – это физическое или понятийное соединение между объектами. Объект сотрудничает с другими объектами через соединяющие их связи. Связь обозначает соединение, с помощью которого:

- объект-клиент вызывает операции объекта-поставщика;
- один объект перемещает данные к другому объекту.

Можно сказать, что связи являются рельсами между станциями-объектами, по которым ездят «трамвайчики сообщений».

Связи между объектами показаны на рисунке 2.4 с помощью соединительных линий.

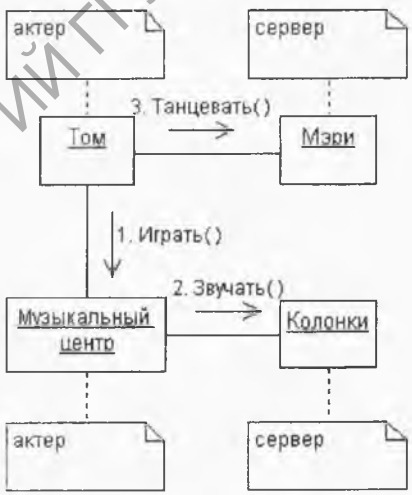

Рисунок 2.4 – Связи между объектами

Связи представляют возможные пути для передачи сообщений. Сами сообщения показаны стрелками, отмечающими их направления, и помечены именами вызываемых операций.

Как участник связи объект может играть одну из трех ролей:

– актер – объект, который может воздействовать на другие объекты, но никогда не подвержен воздействию других объектов;

– сервер – объект, который никогда не воздействует на другие объекты, он только используется другими объектами;

– агент – объект, который может как воздействовать на другие объекты, так и использоваться ими. Агент создается для выполнения работы от имени актера или другого агента.

Связи обозначают равноправные (клиент-серверные) отношения между объектами. Агрегация обозначает отношения объектов в иерархии «целое/часть». Агрегация обеспечивает возможность перемещения от целого (агрегата) к его частям (свойствам).

Агрегация может обозначать, а может и не обозначать физическое включение части в целое. На рисунке 2.5 приведен пример физического включения (композиции) частей (Двигателя, Сидений, Колес) в агрегат Автомобиль. В этом случае говорят, что части включены в агрегат по величине.

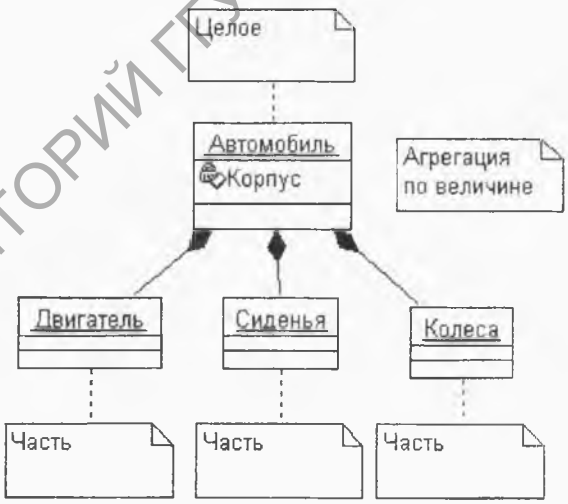

Рисунок 2.5 – Физическое включение частей в агрегат

На рисунке 2.6 приведен пример нефизического включения частей (Студента, Преподавателя) в агрегат Вуз. Очевидно, что Студент и Преподаватель являются элементами Вуза, но они не входят в него физически. В этом случае говорят, что части включены в агрегат по ссылке.

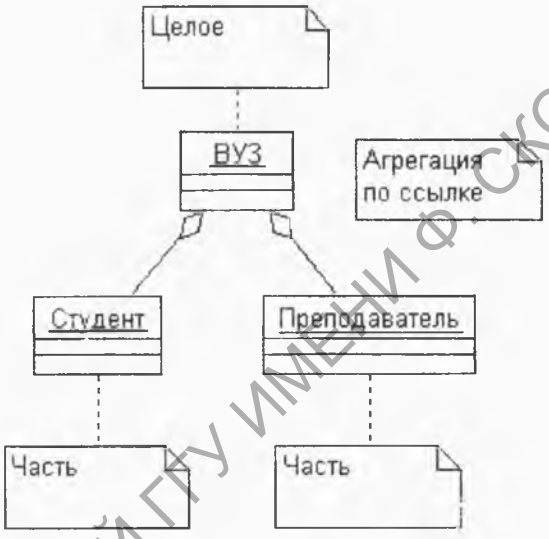

Рисунок 2.6 – Нефизическое включение частей в агрегат

При выборе вида отношения должны учитываться следующие факторы:

– связи обеспечивают низкое сцепление между объектами;

– агрегация инкапсулирует части как секреты целого.

Рассмотрим два объекта, А и В, между которыми имеется связь. Для того чтобы объект А мог послать сообщение в объект В, надо, чтобы В был виден для А.

Различают четыре формы видимости между объектами.

– объект-поставщик (сервер) глобален для клиента;

– объект-поставщик (сервер) является параметром операции клиента;

– объект-поставщик (сервер) является частью объекта-клиента;

– объект-поставщик (сервер) является локально объявленным

объектом в операции клиента.

На этапе анализа вопросы видимости обычно опускают. На этапах проектирования и реализации вопросы видимости по связям обязательно должны рассматриваться.

Тема 3 Классы

3.1 Общая характеристика классов
3.2 Виды отношений между классами
3.3 Ассоциации классов

3.1 Общая характеристика классов

Понятия объекта и класса тесно связаны. Тем не менее существует важное различие между этими понятиями. Класс — это абстракция существенных характеристик объекта.

Класс — описание множества объектов, которые разделяют одинаковые свойства, операции, отношения и семантику (смысл).

Как показано на рисунке 3.1, различают внутреннее представление класса (реализацию) и внешнее представление класса (интерфейс).

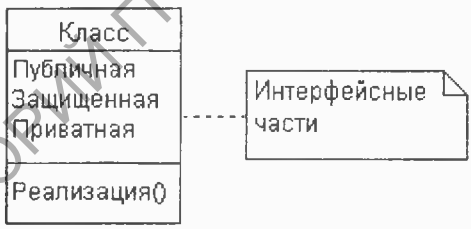

Рисунок 3.1 – Структура представления класса

Интерфейс объявляет возможности (услуги) класса, но скрывает его структуру и поведение. Иными словами, интерфейс демонстрирует внешнему миру абстракцию класса, его внешний облик. Интерфейс в основном состоит из объявлений всех операций, применимых к экземплярам класса. Он может также включать объявления типов, переменных, констант и исключений, необходимых для полноты данной абстракции.

Интерфейс может быть разделен на 3 части:

1) публичную *(public),* объявления которой доступны всем клиентам;

2) защищенную *(protected),* объявления которой доступны только самому классу, его подклассам и друзьям;

3) приватную *(private),* объявления которой доступны только самому классу и его друзьям.

Другом класса называют класс, который имеет доступ ко всем частям этого класса (публичной, защищенной и приватной). Иными словами, от друга у класса нет секретов.

Реализация класса описывает секреты поведения класса. Она включает реализации всех операций, определенных в интерфейсе класса.

3.2 Виды отношений между классами

Классы, подобно объектам, не существуют в изоляции. Напротив, с отдельной проблемной областью связывают ключевые абстракции, отношения между которыми формируют структуру из классов системы. Всего существует четыре основных вида отношений между классами:

– ассоциация (фиксирует структурные отношения — связи между экземплярами классов);

– зависимость (отображает влияние одного класса на другой класс);

– обобщение-специализация («is a»–отношение);

– целое-часть («part of»–отношение).

Для покрытия основных отношений большинство объектно-ориентированных языков программирования поддерживает следующие отношения:

1) ассоциация;
2) наследование;
3) агрегация;
4) зависимость;
5) конкретизация;
6) метакласс;
7) реализация.

Ассоциации обеспечивают взаимодействия объектов, принадлежащих разным классам. Они являются клеем, соединяющим воедино все элементы программной системы. Благодаря ассоциациям мы полу-

чаем работающую систему. Без ассоциаций система превращается в набор изолированных классов-одиночек. Наследование — наиболее популярная разновидность отношения *обобщение-специализация.* Альтернативой наследованию считается делегирование. При делегировании объекты *делегируют* свое поведение родственным объектам. При этом классы становятся не нужны.

Агрегация обеспечивает отношения *целое-часть,* объявляемые для экземпляров классов.

Зависимость часто представляется в виде частной формы — *использования,* которое фиксирует отношение между клиентом, запрашивающим услугу, и сервером, предоставляющим эту услугу.

Конкретизация выражает другую разновидность отношения *обобщение-специализация.* Применяется в таких языках, как Ada 95, C++, Эйфель. Отношения метаклассов поддерживаются в языках SmallTalk и CLOS. Метакласс — это класс классов, понятие, позволяющее обращаться с классами как с объектами.

Реализация определяет отношение, при котором класс-приемник обеспечивает свою собственную реализацию интерфейса другого класса-источника. Иными словами, здесь идет речь о наследовании интерфейса. Семантически реализация — это «скрещивание» отношений зависимости и обобщения-специализации.

3.3 Ассоциации классов

Ассоциация обозначает семантическое соединение классов.

Пример: в системе обслуживания читателей имеются две ключевые абстракции — Книга и Библиотека. Класс Книга играет роль элемента, хранимого в библиотеке. Класс Библиотека играет роль хранилища для книг.

Ассоциация обозначает только семантическую связь. Она не указывает направление и точную реализацию отношения. Ассоциация пригодна для анализа проблемы, когда нам требуется лишь идентифицировать связи. С помощью создания ассоциаций мы приходим к пониманию участников семантических связей, их ролей, мощности (количества элементов).

Пример отношения ассоциации между классами приведен на рисунке 3.2.

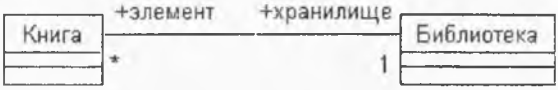

Рисунок 3.2 – Ассоциация

Очевидно, что ассоциация предполагает двухсторонние отношения:

– для данного экземпляра Книги выделяется экземпляр Библиотеки, обеспечивающий ее хранение;

– для данного экземпляра Библиотеки выделяются все хранимые Книги.

Здесь показана ассоциация *один-ко-многим*. Каждый экземпляр Книги имеет указатель на экземпляр Библиотеки. Каждый экземпляр Библиотеки имеет набор указателей на несколько экземпляров Книги.

Ассоциация *один-ко-многим,* введенная в примере, означает, что для каждого экземпляра класса Библиотека есть 0 или более экземпляров класса Книга, а для каждого экземпляра класса Книга есть один экземпляр Библиотеки. Эту множественность обозначает *мощность ассоциации.* Мощность ассоциации бывает одного из трех типов:

– один-к-одному;

– один-ко-многим;

– многие-ко-многим.

Примеры ассоциаций с различными типами мощности приведены на рисунке 3.3, они имеют следующий смысл:

– у европейской жены один муж, а у европейского мужа одна жена;

– у восточной жены один муж, а у восточного мужа сколько угодно жен;

– у заказа один клиент, а у клиента сколько угодно заказов;

– человек может посещать сколько угодно зданий, а в здании может находиться сколько угодно людей.

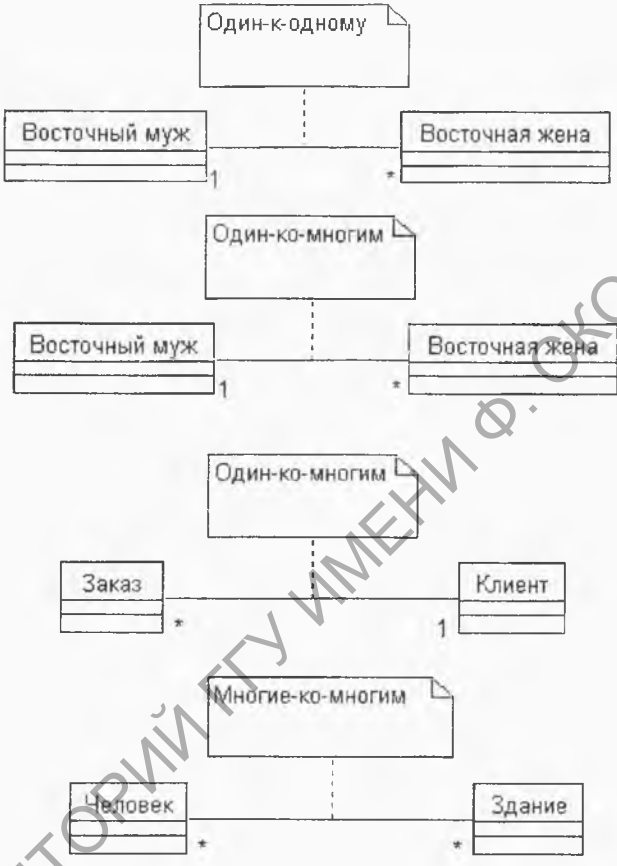

Рисунок 3.3 – Ассоциации с различными типами мощности

3 Язык моделирования UML

Тема 1 Основы языка UML

3.1 Сущность и особенности языка UML
3.2 Общая структура языка UML

3.1 Сущность и особенности языка UML

Унифицированный язык моделирования UML (Unified Modeling Language) представляет собой общецелевой язык визуального моделирования, который разработан для спецификации, визуализации, проектирования и документирования компонентов программного обеспечения, бизнес-процессов и других систем [1], [2], [5], [8], [12]. Язык UML одновременно является простым и мощным средством моделирования, который может быть эффективно использован для построения концептуальных, логических и графических моделей сложных систем самого различного целевого назначения. Этот язык вобрал в себя наилучшие качества методов программной инженерии, которые с успехом использовались на протяжении последних лет при моделировании больших и сложных систем.

Язык UML основан на некотором числе базовых понятий, которые могут быть изучены и применены большинством программистов и разработчиков, знакомых с методами объектно-ориентированного анализа и проектирования. При этом базовые понятия могут комбинироваться и расширяться таким образом, что специалисты объектного моделирования получают возможность самостоятельно разрабатывать модели больших и сложных систем в самых различных областях приложений.

Язык UML предназначен для решения следующих задач:

1) предоставить в распоряжение пользователей легко воспринимаемый и выразительный язык визуального моделирования, специально предназначенный для разработки и документирования моделей сложных систем самого различного целевого назначения;

2) снабдить исходные понятия языка UML возможностью расширения и специализации для более точного представления моделей систем в конкретной предметной области;

3) описание языка UML должно поддерживать такую спецификацию моделей, которая не зависит от конкретных языков программирования и инструментальных средств проектирования программных сис-

тем;

4) описание языка UML должно включать в себя семантический базис для понимания общих особенностей ООАП;

5) поощрять развитие рынка объектных инструментальных средств;

6) способствовать распространению объектных технологий и соответствующих понятий ООАП;

7) интегрировать в себя новейшие и наилучшие достижения практики ООАП.

С точки зрения визуального моделирования, UML можно охарактеризовать следующим образом. UML предоставляет выразительные средства для создания визуальных моделей, которые единообразно понимаются всеми разработчиками, вовлеченными в проект, и являются средством коммуникации в рамках проекта.

Унифицированный язык моделирования (UML) обладает следующими достоинствами:

– не зависит от объектно-ориентированных (ОО) языков программирования;

– не зависит от используемой методологии разработки проекта;

– может поддерживать любой объектно-ориентированный язык программирования.

UML является открытым и обладает средствами расширения базового ядра. На UML можно содержательно описывать классы, объекты и компоненты в различных предметных областях, часто сильно отличающихся друг от друга.

3.2 Общая структура языка UML

С самой общей точки зрения описание языка UML состоит из двух взаимодействующих частей, таких как:

семантика языка UML. Представляет собой некоторую метамодель, которая определяет абстрактный синтаксис и семантику понятий объектного моделирования на языке UML;

– нотация языка UML. Представляет собой графическую нотацию для визуального представления семантики языка UML.

Абстрактный синтаксис и семантика языка UML описываются с использованием некоторого подмножества нотации UML. В дополнение к этому, нотация UML описывает соответствие или отображение графической нотации в базовые понятия семантики. Таким образом, с функциональной точки зрения эти две части дополняют друг друга.

При этом семантика языка UML описывается на основе некоторой метамодели, имеющей три отдельных представления: абстрактный синтаксис, правила корректного построения выражений и семантику.

Семантика определяется для двух видов объектных моделей: структурных моделей и моделей поведения. Структурные модели, известные также как статические модели, описывают структуру сущностей или компонентов некоторой системы, включая их классы, интерфейсы, атрибуты и отношения. Модели поведения, называемые иногда динамическими моделями, описывают поведение или функционирование объектов системы, включая их методы, взаимодействие и сотрудничество между ними, а также процесс изменения состояний отдельных компонентов и системы в целом.

Для решения столь широкого диапазона задач моделирования разработана достаточно полная семантика для всех компонентов графической нотации. Требования семантики языка UML конкретизируются при построении отдельных видов диаграмм. Нотация языка UML включает в себя описание отдельных семантических элементов, которые могут применяться при построении диаграмм.

Формальное описание самого языка UML основывается на некоторой общей иерархической структуре модельных представлений, состоящей из четырех уровней:

1) мета-метамодель;
2) метамодель;
3) модель;
4) объекты пользователя.

Уровень *мета-метамодели* образует исходную основу для всех метамодельных представлений. Главное предназначение этого уровня состоит в том, чтобы определить язык для спецификации метамодели. Мета-метамодель определяет модель языка UML на самом высоком уровне абстракции и является наиболее компактным ее описанием. С другой стороны, мета-метамодель может специфицировать несколько метамоделей, чем достигается потенциальная гибкость включения дополнительных понятий.

Метамодель является экземпляром или конкретизацией мета-метамодели. Главная задача этого уровня – определить язык для спецификации моделей. Данный уровень является более конструктивным, чем предыдущий, поскольку обладает более развитой семантикой базовых понятий. Все основные понятия языка UML – это понятия уровня метамодели. Примеры таких понятий – класс, атрибут, операция, компонент, ассоциация и многие другие.

Модель в контексте языка UML является экземпляром метамодели в том смысле, что любая конкретная модель системы должна использовать только понятия метамодели, конкретизировав их применительно к данной ситуации. Это уровень для описания информации о конкретной предметной области. Однако если для построения модели используются понятия языка UML, то необходима полная согласованность понятий уровня модели с базовыми понятиями языка UML уровня метамодели.

Конкретизация понятий модели происходит на уровне *объектов*. В настоящем контексте объект является экземпляром модели, поскольку содержит конкретную информацию относительно того, чему в действительности соответствуют те или иные понятия модели.

Тема 2 Предметы в UML

2.1 Структурные предметы
2.2 Предметы поведения
2.3 Группирующие и поясняющие предметы

2.1 Структурные предметы

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.

Класс – описание множества объектов, которые разделяют одинаковые свойства, операции, отношения и семантику (смысл). Класс реализует один или несколько интерфейсов. Как показано на рисунке 2.1, графически класс отображается в виде прямоугольника, обычно включающего секции с именем, свойствами (атрибутами) и операциями.

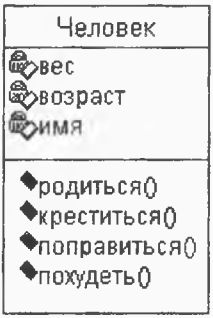

Рисунок 2.1 – Обозначение класса в нотации UML

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 рисунке 2.2. Имя интерфейса обычно начинается с буквы «I». Интерфейс редко показывают самостоятельно. Обычно его присоединяют к классу или компоненту, который реализует интерфейс.

Рисунок 2.2 – Обозначение интерфейса в нотации UML

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 рисунке 2.3, графически кооперация изображается как пунктирный эллипс, в который вписывается ее имя.

Рисунок 2.3 – Обозначение кооперации в нотации UML

Актер – набор согласованных ролей, которые могут играть пользователи при взаимодействии с системой (ее элементами Use Case). Каждая роль требует от системы определенного поведения. Обозначение актера приведено на рисунке 2.4.

Рисунок 2.4 – Обозначение актера в нотации UML

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 рисунке 2.5, элемент Use Case изображается как эллипс, в который вписывается его имя.

Рисунок 2.5 – Обозначение элемента Use Case

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 рисунке 2.6, активный класс изображается как утолщенный прямоугольник, обычно включающий имя, свойства (атрибуты) и операции.

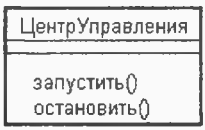

Рисунок 2.6 – Обозначение активного класса в нотации UML

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 кооперации). Как показано на рисунке 2.7, компонент изображается как прямоугольник с вкладками, обычно включающий имя.

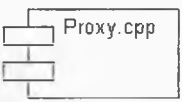

Рисунок 2.7 – Обозначение компонента в нотации UML

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 рисунке 2.8, узел изображается как куб с именем.

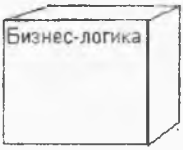

Рисунок 2.8 – Обозначение узла в нотации UML

2.2 Предметы поведения

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

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 рисунке 2.9, сообщение изображается в виде направленной линии с именем ее операции.

Рисунок 2.9 – Сообщение

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 рисунке 2.10, состояние изображается как закругленный прямоугольник, обычно включающий его имя и его подсостояния (если они есть).

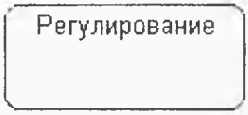

Рисунок 2.10 – Обозначение конечного автомата

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, кооперациями и объектами).

2.3 Группирующие и поясняющие предметы

Группирующие предметы – организационные части UML-моделей. Это контейнеры, 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 рисунке 2.11, пакет изображается как папка с закладкой, на которой обозначено его имя и, иногда, его содержание.

Рисунок 2.11 – Обозначение пакета

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 рисунке 2.12, примечание изображается в виде прямоугольника с загнутым углом, в который вписывается текстовый или графический комментарий.

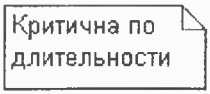

Рисунок 2.12 – Обозначение примечания

Тема 3 Отношения в UML

3.1 Отношение зависимости
3.2 Отношение ассоциации
3.3 Отношение обобщения
3.4 Отношение реализации

3.1 Отношение зависимости

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.

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 рисунке 3.1, зависимость изображается в виде пунктирной линии, возможно направленной на независимый предмет и иногда имеющей метку.

Рисунок 3.1 – Отношение зависимости

3.2 Отношение ассоциации

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 рисунке 3.2, ассоциация изображается в виде сплошной линии, возможно направленной, иногда имеющей метку и часто включающей другие «украшения», такие как мощность и имена ролей.

Рисунок 3.2 – Отношение ассоциации

3.3 Отношение обобщения

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 рисунке 3.3, обобщение изображается в виде сплошной стрелки с полым наконечником, указывающим на родителя.

Рисунок 3.3 – Отношение обобщения

3.4 Отношение реализации

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 рисунке

3.4, реализация изображается как нечто среднее между обобщением и зависимостью.

Рисунок 3.4 – Отношение реализации

Тема 4 Диаграммы в UML

4.1 Виды диаграмм
4.2 Механизмы расширения

4.1 Виды диаграмм

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 пред-

ставление 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 системы. 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 кооперации. 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 кооперации.

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.

4.2 Механизмы расширения

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 расширения. 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 рисунке 4.1 введено простое ограничение на свойство *сумма* класса *Сессия Банкомата* — его значение должно быть кратно 20.

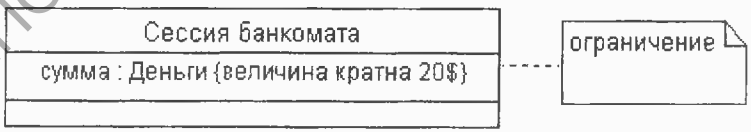

Рисунок 4.1 – Ограничения

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.

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 рисунке 4.2 класс <Текстовый процессор> расширен путем явного указания его версии и автора.

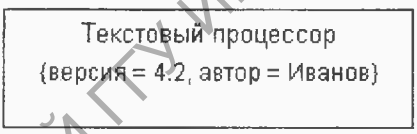

Рисунок 4.2 – Расширение класса

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 рисунке 4.3. Стереотип «exception» говорит о том, что класс <ПотеряЗначимости> теперь рассматривается как специальный класс, которому, положим, разрешается только генерация и обработка сигналов исключений. Осо-

бые возможности метакласса получил класс <ЭлементМодели>. Кроме того, здесь показано применение стереотипа «call» к отношению зависимости (у него появился новый смысл).

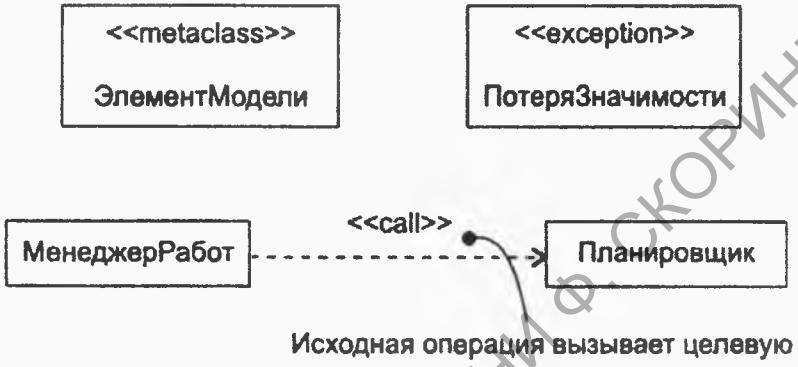

Рисунок 4.3 – Пример стереотипа

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.

4 Статические модели программных систем

Тема 1 Диаграмма классов

1.1 Назначение диаграммы классов
1.2 Вершины в диаграммах классов
1.3 Операции классов
1.4 Организация свойств и операций

1.1 Назначение диаграммы классов

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. 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.

1.2 Вершины в диаграммах классов

Итак, вершина в диаграмме классов — класс. Обозначение класса показано на рисунке 1.1.

Рисунок 1.1 – 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.

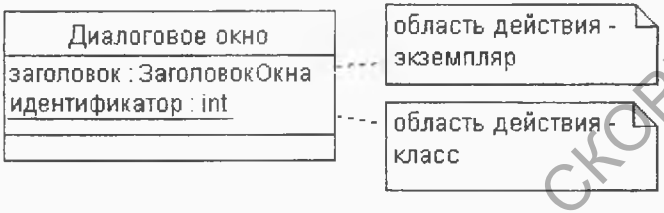

Рисунок 1.2 – Свойства уровней класса и экземпляра

Общий синтаксис представления свойства имеет вид:

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

Рассмотрим видимость и характеристики свойств. В языке UML определены три уровня видимости (таблица 1.1):

Таблица 1.1 – Уровни видимости класса

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

Если видимость не указана, считают, что свойство объявлено с публичной видимостью. Определены три характеристики свойств (таблица 1.2):

Таблица 1.2 – Характеристики свойства класса

Название	Назначение
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

Если характеристика не указана, считают, что свойство объявлено с характеристикой changeable. Примеры объявления свойств (таблица 1.3):

Таблица 1.3 – Примеры объявления свойств

Объявление	Комментарий
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

1.3 Операции классов

Общий синтаксис представления операции имеет вид:
Видимость Имя (Список Параметров): ВозвращаемыйТип {Характеристики}
Примеры объявления операций (таблица 1.4):

Таблица 1.4 – Примеры объявления операций

Название	Назначение
записать	Только имя
+ записать	Видимость и имя
Зарегистрировать(и: Имя, ф: Фамилия)	Имя и параметры

Окончание таблицы 1.4

Название	Назначение
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 (таблица 1.5):

Таблица 1.5 – Значения Направления

Название	Назначение
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 (таблица 1.6):

Таблица 1.6 – Характеристики операций

Название	Назначение
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).

Окончание таблицы 1.6

Название	Назначение
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

1.4 Организация свойств и операций классов

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. Как приведено на рисунке 1.3, в длинных списках свойств и операций разрешается группировка — каждая группа начинается со своего стереотипа.

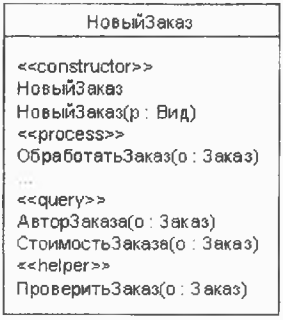

Рисунок 1.3 – Стереотипы для характеристик класса

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 рисунке 1.4, <КонтроллерУглов> — это класс–*singleton*, а для класса <ДатчикУгла> разрешены три экземпляра.

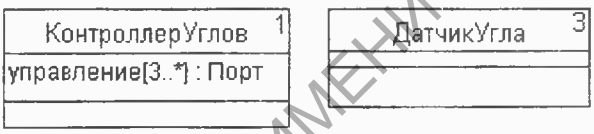

Рисунок 1.4 – Множественность класса

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>).

Тема 2 Отношения в диаграммах классов

2.1 Отношение ассоциации
2.2 Отношение зависимости
2.3 Отношение обобщения
2.4 Отношение реализации

2.1 Отношение ассоциации

Отношения в диаграммах классов показаны на рисунке 2.1.

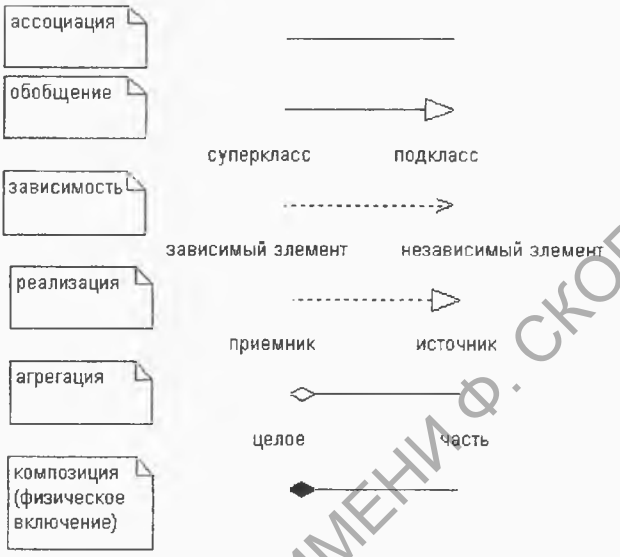

Рисунок 2.1 – Отношения в диаграммах классов

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 рисунке 2.2, имени можно придать направление — достаточно добавить треугольник направления, который указывает направление, заданное для чтения имени.

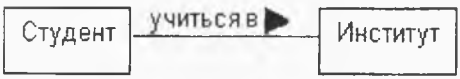

Рисунок 2.2 – Имена ассоциаций

Когда класс участвует в ассоциации, он играет в этом отношении определенную роль. Как показано на рисунке 2.3, роль определяет, каким представляется класс на одном конце ассоциации для класса на противоположном конце ассоциации.

Рисунок 2.3 – Роли ассоциации

Запись мощности на одном конце ассоциации определяет количество объектов, соединяемых с каждым объектом на противоположном конце ассоциации (рисунок 2.4). 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.

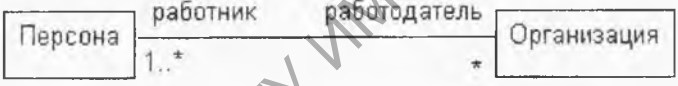

Рисунок 2.4 – Мощности ассоциации

В языке UML ассоциации могут иметь свойства. Как показано на рисунке 2.5, 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.

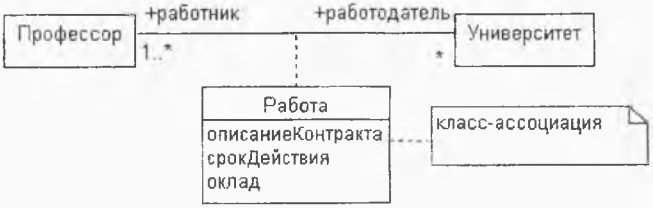

Рисунок 2.5 – Класс-ассоциация

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).

2.2 Отношение зависимости

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 рисунке 2.6 показана зависимость класса Заказ от класса Книга, так как <Книга> используется в операциях <проверка Доступности>, добавить и удалить класса <Заказ>.

Рисунок 2.6 – Отношения зависимости

На этом рисунке изображена еще одна зависимость, которая показывает, что класс <ПросмотрЗаказа> использует класс <Заказ>. Причем <Заказ> ничего не знает о <ПросмотрЗаказа>. Данная зависимость помечена стереотипом «friend», который расширяет простую зависимость, определенную в языке.

2.3 Отношение обобщения

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.

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 (рисунок 2.7).

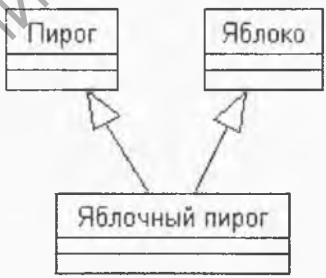

Рисунок 2.7 – Множественное наследование

Еще более сложные проблемы возникают при наследовании от двух классов, имеющих общего родителя. Говорят, что в результате образуется ромбовидная решетка наследования (рисунок 2.8).

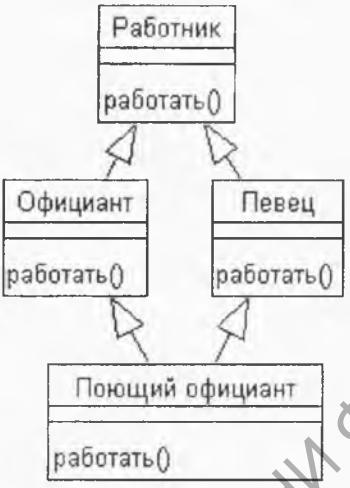

Рисунок 2.8 – Ромбовидная решетка наследования

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.

2.4 Отношение реализации

Реализация — семантическое отношение между классами, в котором класс-приемник выполняет реализацию операций интерфейса класса-источника. Например, на рисунке 2.9 показано, что класс <Каталог> должен реализовать интерфейс <IОбработчикКаталога>, то есть <IОбработчикКаталога> рассматривается как источник, а <Каталог> — как приемник.

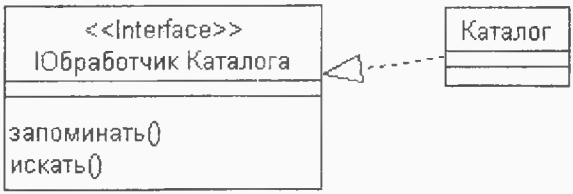

Рисунок 2.9 – Реализация интерфейса

Интерфейс <ІОбработчикКаталога> 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.).

Тема 3 Деревья наследования

3.1 Особенности деревьев наследования
3.2 Пример дерева наследования

3.1 Особенности деревьев 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 рисунке 3.1 показаны абстрактные классы *<Млекопитающее>*, *<Собака>*, *<Кошка>*.

Кроме того, здесь имеются конкретные классы <Охотничья собака>, <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>.

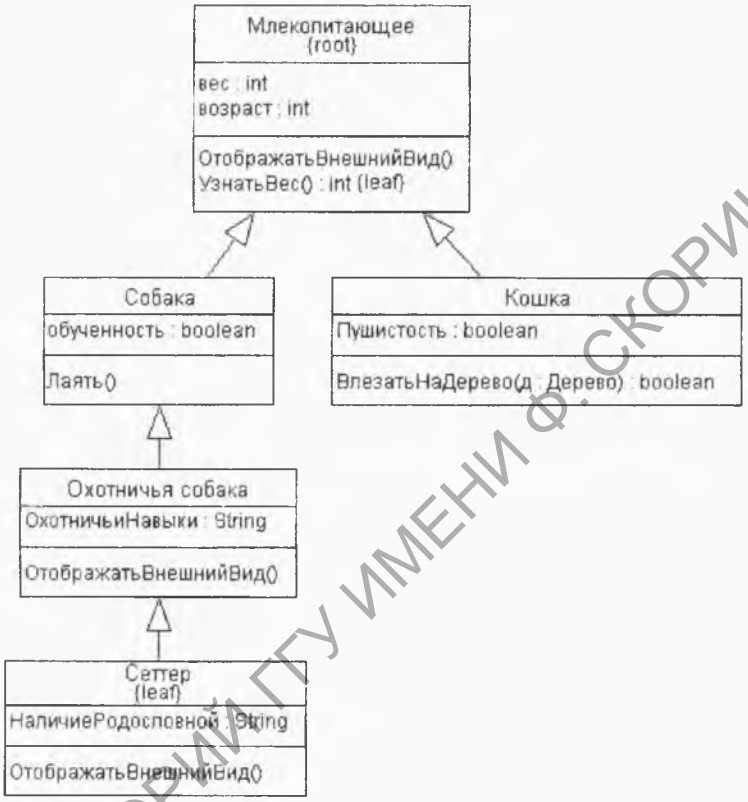

Рисунок 3.1 – Абстрактность и полиморфизм

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 <Млекопитающее>.

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. На-

пример, <Отображать ВнешнийВид()> и <ВлезатьНаДерево(дуб)> — полиморфные операции. К тому же операция *<Млекопитающее::ОтображатьВнешнийВид()>* является абстрактной, то есть неполной и требующей для своей реализации потомка. Имя абстрактной операции записывается курсивом (как и имя класса). С другой стороны, <Млекопитающее::УзнатьВес()> — конечная операция, что отмечается характеристикой leaf. Это значит, что операция не полиморфна и не может перекрываться.

3.2 Пример дерева наследования

В качестве второго примера на рисунке 3.2 приведена диаграмма классов для информационной системы театра. Эту систему образует 6 классов.

Классы-агрегаты <Театр> и <Труппа> имеют операции добавления и удаления своих частей, которые включаются в агрегаты по ссылке. Частями <Театра> являются <Зрители> и <Труппы>, а частями <Труппы> — <Актеры>. Отношения агрегации между классом <Театр> и классами <Труппа> и <Зритель> слегка отличны. Театр может состоять из одной или нескольких трупп, но каждая труппа находится в одном и только одном театре. С другой стороны, в театр может ходить любое количество зрителей (включая нулевое количество), причем зритель может посещать один или несколько театров. Между классами <Труппа> и <Актер> существуют два отношения — агрегация и ассоциация. Агрегация показывает, что каждый актер работает в одной или нескольких труппах, а в каждой труппе должен быть хотя бы один актер. Ассоциация отображает, что каждой труппой управляет только один актер — художественный руководитель, а некоторые актеры не являются руководителями.

Ассоциация между классами Спектакль и Актер фиксирует, что в спектакле должен быть занят хотя бы один актер, впрочем, актер может играть в любом количестве спектаклей (или вообще может ничего не играть).

Между классами Спектакль и Зритель тоже определена ассоциация. Она поясняет, что зритель может смотреть любое число спектаклей, а на каждом спектакле может быть любое число зрителей.

Между классами Спектакль и Зритель тоже определена ассоциация. Она поясняет, что зритель может смотреть любое число спектаклей, а на каждом спектакле может быть любое число зрителей.

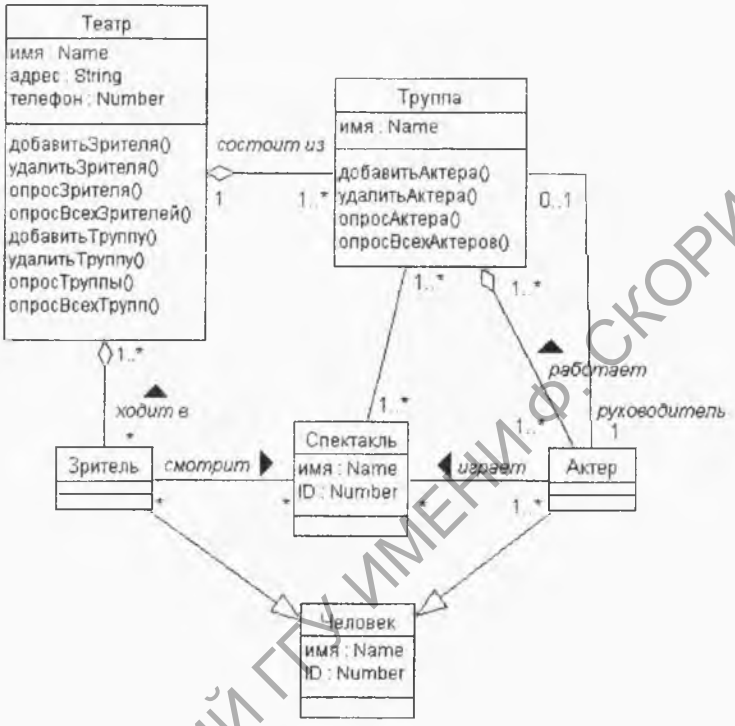

Рисунок 3.2 – Диаграмма классов театра

И наконец, на диаграмме отображены два отношения наследования, утверждающие, что и в зрителях, и в актерах есть человеческое начало.

5 Динамические модели программных комплексов

Тема 1 Диаграммы схем состояний

1.1 Моделирование поведения программной системы
1.2 Особенности диаграммы схем состояний
1.3 Условные переходы
1.4 Вложенные состояния

1.1 Моделирование поведения программной системы

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 [5], [8], [9], [10]. Эти средства ориентированы не только на собственно программные системы, но и на отображение требований заказчика к поведению таких систем.

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.

1.2 Особенности 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 рисунке 1.1, состояние изображается как закругленный прямоугольник, обычно включающий его имя и подсостояния (если они есть).

Рисунок 1.1 – Обозначение состояния

Переходы между состояниями отображаются помеченными стрелками (рисунок 1.2).

Рисунок 1.2 – Переходы между состояниями

На рисунке 1.2 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); flt.убратьПомехи( ) - вызов двух операций;
- send Ник. Привет - посылка сигнала в объект Ник.

Для отображения перехода в начальное состояние принято обозначение, показанное на рисунке 1.3.

Рисунок 1.3 – Переход в начальное состояние

Соответственно, обозначение перехода в конечное состояние имеет вид, представленный на рисунке 1.4.

Рисунок 1.4 – Переход в конечное состояние

В качестве примера на рисунке 1.5 показана диаграмма схем состояний для системы охранной сигнализации.

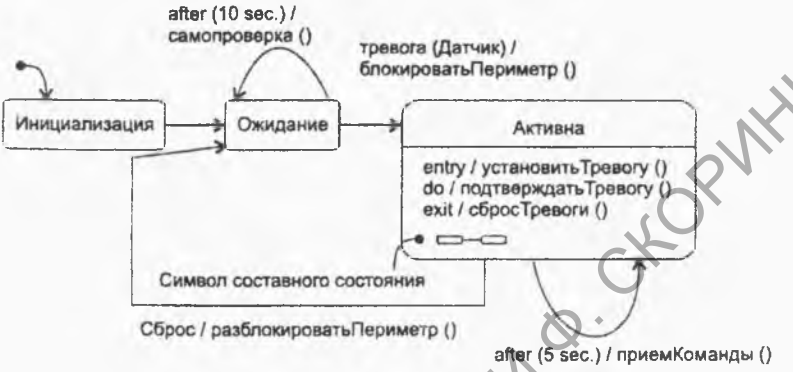

Рисунок 1.5 - Диаграмма схем состояний системы охранной сигнализации

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()).

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

Например, как показано на рисунке 1.6, при входе в состояние Активна выполняется операция установитьТревогу() из класса Контроллер, а при выходе из состояния — операция сбросТревоги().

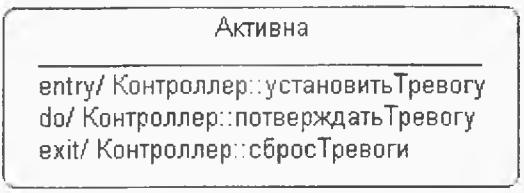

Рисунок 1.6 – Входные и выходные действия и деятельность в состоянии Активна

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().

1.3 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 рисунок 1.7.

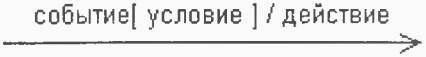

Рисунок 1.7 – Обозначение условного перехода

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 рисунке 1.8. Он происходит по событию питание подано, но только в том случае, если достигнут боевой режим лазера.

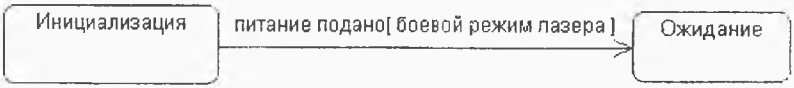

Рисунок 1.8 – Условный переход между состояниями

1.4 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 рисунке 1.9 показано составное состояние, содержащее в себе два подсостояния.

Рисунок 1.9 – Обозначение подсостояний

На рисунке 1.10 приведена внутренняя структура составного состояния Активна.

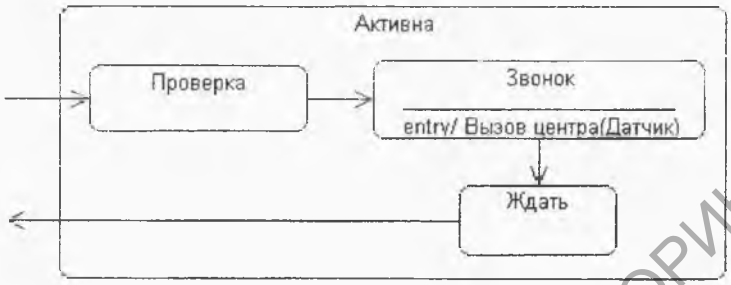

Рисунок 1.10 – Переходы в состоянии Активна

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 рисунке 1.11, подобная семантика переходов отображается значком истории — буквой H внутри кружка.

При первом посещении состояния Активна автомат не имеет истории, поэтому происходит простой переход в подсостояние Проверка. Предположим, что в подсостоянии 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).

Рисунок 1.11 – Историческое состояние

Для обозначения составного состояния, имеющего внутри себя скрытые (не показанные на диаграмме) подсостояния, используется символ, приведенный на рисунке 1.12.

Рисунок 1.12 – Символ состояния со скрытыми подсостояниями

Тема 2 Диаграммы деятельности

2.1 Графические обозначения в диаграммах деятельности
2.2 Пример диаграммы деятельности

2.1 Графические обозначения в диаграммах деятельности

Диаграмма деятельности представляет особую форму конечного автомата, в которой показываются процесс вычислений и потоки ра-

бот. В ней выделяются не обычные состояния объекта, а состояния выполняемых вычислений — действия. 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 действие, которое изображено на рисунке 2.1.

Действие считается атомарным (действие нельзя прервать) и выполняется за один квант времени, его нельзя подвергнуть декомпозиции.

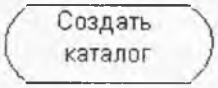

Рисунок 2.1 – Состояние действия

Если нужно представить сложное действие, которое можно подвергнуть дальнейшей декомпозиции (разбить на ряд более простых действий), то используют состояние поддеятельности. Изображение состояния поддеятельности содержит пиктограмму в правом нижнем углу (рисунок 2.2).

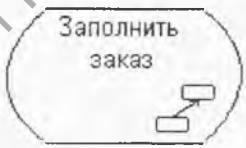

Рисунок 2.2 – Состояние поддеятельности

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

Переходы между вершинами — действиями — 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 ис-

ходящей 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).

2.2 Пример диаграммы деятельности

Пример диаграммы деятельности приведен на рисунке 2.3.

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.

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.

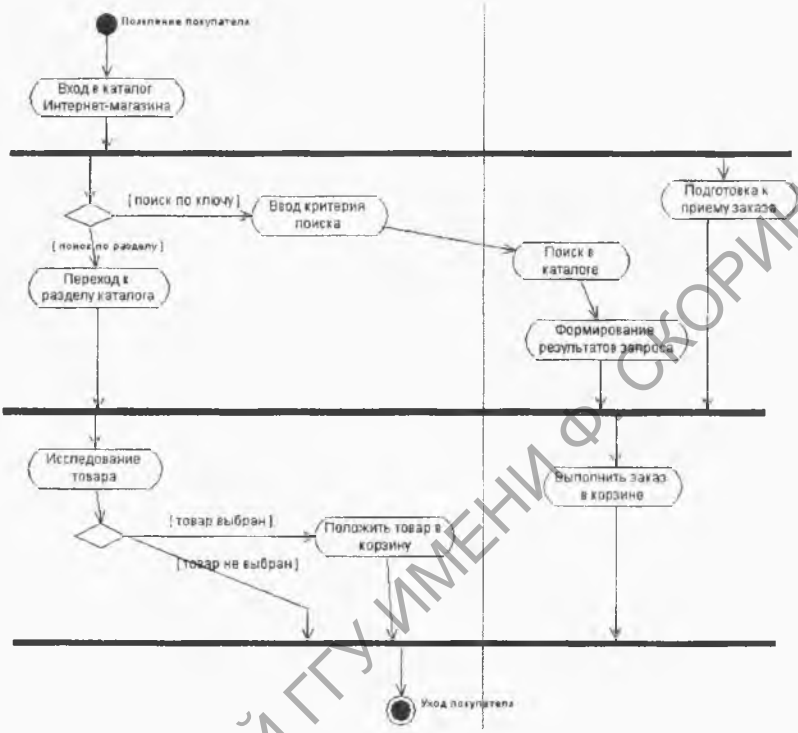

Рисунок 2.3 – Диаграмма деятельности покупателя в Интернет-магазине

Тема 3 Диаграммы сотрудничества

3.1 Диаграммы взаимодействия
3.2 Особенности диаграмм сотрудничества
3.3 Пример диаграммы сотрудничества

3.1 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.

3.2 Особенности диаграмм 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 рисунке 3.1.

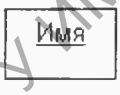

Рисунок 3.1 – Обозначение объекта

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 (таблица 3.1):

Таблица 3.1 – Стандартные стереотипы видимости

Название	Семантика
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 (таблица 3.2):

Таблица 3.2 – Разновидности действий

Действие	Семантика
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»

Окончание таблицы 3.2

Действие	Семантика
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 (таблица 3.3):

Таблица 3.3 – Примеры записи сообщений

Сообщение	Семантика
Коорд := ТекущПоложение(самолет)	Вызов операции, возврат значения
оповещение( )	Посылка сигнала
УстановитьМаршрут(x)	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), приведенный на рисунке 3.2.

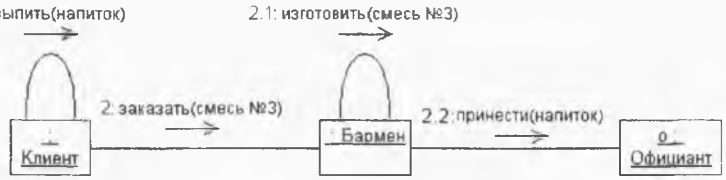

Рисунок 3.2 – Поток синхронных сообщений

Здесь сообщение 2.1 : изготовить(смесь №3) определено как первое сообщение, вложенное во второе сообщение 2 : заказать(смесь №3) 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. Асинхронный поток приведен на рисунке 3.3. 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 подтверждение вызова определено как второе сообщение в последовательности.

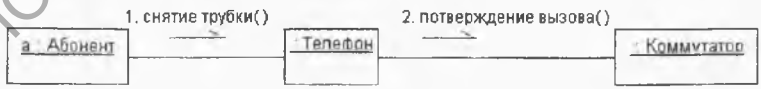

Рисунок 3.3 – Поток асинхронных сообщений

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 рисунке 3.4.

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.

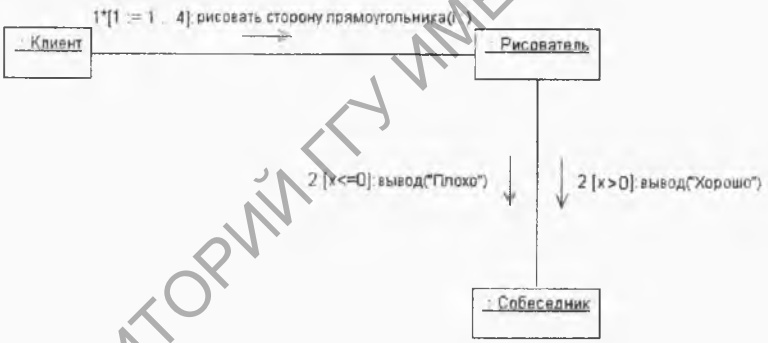

Рисунок 3.4 – Итерация и ветвление

3.3 Пример диаграммы сотрудничества

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.

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

В качестве примера на рисунке 3.5 приведена диаграмма сотрудничества системы управления полетом летательного аппарата.

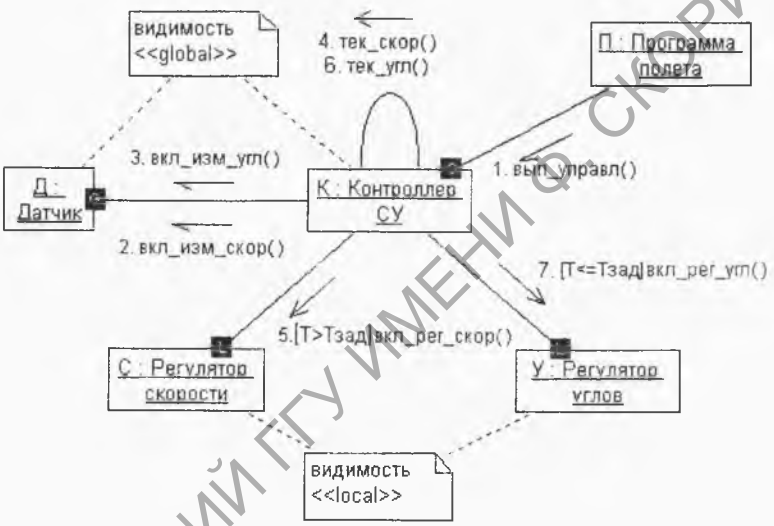

Рисунок 3.5 – Диаграмма сотрудничества системы управления полетом

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

- вкл_рег_скор( );
- вкл_рег_угл();
- тек_скор();
- тек_угл( ).

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 полета меньше или равно заданному периоду.

Тема 4 Диаграммы последовательности

4.1 Особенности диаграмм последовательности
4.2 Линия жизни объекта
4.3 Фокус управления

4.1 Особенности диаграмм 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*.

Как показано на рисунке 4.1, 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.

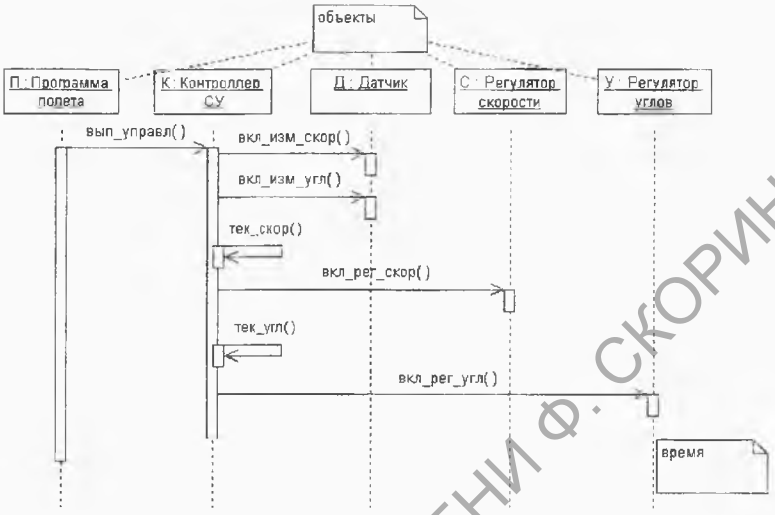

Рисунок 4.1 – Диаграмма последовательности системы управления полетом

От диаграмм сотрудничества диаграммы последовательности отличают две важные характеристики.

4.2 Линия жизни объекта

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 рисунке 4.2, уничтожение линии жизни отмечается пометкой X в конце линии:

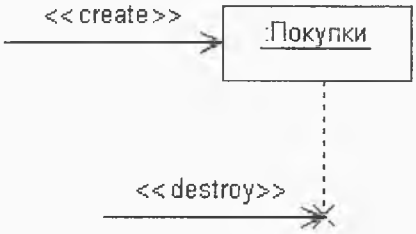

Рисунок 4.2 – Создание и уничтожение объекта

4.3 Фокус управления

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 (рисунок 4.3).

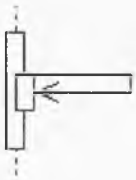

Рисунок 4.3 – Вложение фокусов управления

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 объекта могут быть снова слиты (рисунок 4.4);

Рисунок 4.4 – Параллельные линии жизни

2) ветвление показывается множеством стрелок, идущих из одной точки. Каждая стрелка отмечается сторожевым условием (рисунок 4.5).

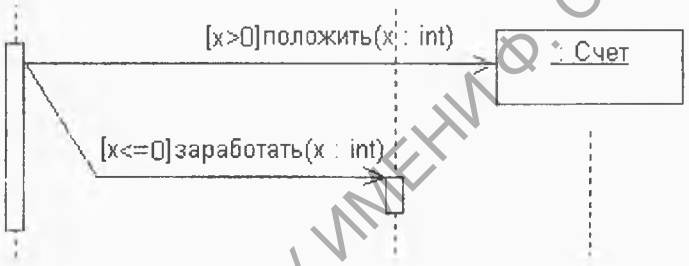

Рисунок 4.5 – Ветвление

Тема 5 Диаграммы Use Case

5.1 Назначение диаграмм Use Case
5.2 Актеры и элементы Use Case
5.3 Отношения в диаграммах Use Case

5.1 Назпачение диаграмм Use Case

Диаграмма Use Case определяет поведение системы с точки зрения пользователя. Диаграмма Use Case рассматривается как главное средство для первичного моделирования динамики системы, используется для выяснения требований к разрабатываемой системе, фиксации этих требований в форме, которая позволит проводить дальнейшую разработку. В русской литературе диаграммы Use Case часто называют диаграммами прецедентов, или диаграммами вариантов использования.

В состав диаграмм Use Case входят элементы Use Case, актеры, а также отношения зависимости, обобщения и ассоциации. Как и другие диаграммы, диаграммы Use Case могут включать примечания и ограничения. Кроме того, диаграммы Use Case могут содержать пакеты, используемые для группировки элементов модели в крупные фрагменты.

5.2 Актеры и элементы Use Case

Вершинами в диаграмме Use Case являются актеры и элементы Use Case. Их обозначения показаны на рисунке 5.1.

Актеры представляют внешний мир, нуждающийся в работе системы. Элементы Use Case представляют действия, выполняемые системой в интересах актеров.

Рисунок 5.1 - Обозначения актера и элемента Use Case

Актер — это роль объекта вне системы, который прямо взаимодействует с ее частью — конкретным элементом (элементом Use Case). Различают актеров и пользователей. Пользователь — это физический объект, который использует систему. Он может играть несколько ролей и поэтому может моделироваться несколькими актерами. Справедливо и обратное — актером могут быть разные пользователи.

Например, для коммерческого летательного аппарата можно выделить двух актеров: пилота и кассира. Сидоров — пользователь, который иногда действует как пилот, а иногда — как кассир. Как изображено на рисунке 5.2, в зависимости от роли Сидоров взаимодействует с разными элементами Use Case.

Рисунок 5.2 – Модель Use Case

Элемент Use Case — это описание последовательности действий (или нескольких последовательностей), которые выполняются системой и производят для отдельного актера видимый результат.

Один актер может использовать несколько элементов Use Case, и наоборот, один элемент Use Case может иметь несколько актеров, использующих его. Каждый элемент Use Case задает определенный путь использования системы. Набор всех элементов Use Case определяет полные функциональные возможности системы.

5.3 Отношения в диаграммах Use Case

Между актером и элементом Use Case возможен только один вид отношения — ассоциация, отображающая их взаимодействие (рисунок 5.3).

Рисунок 5.3 – Отношение ассоциации

Как и любая другая ассоциация, она может быть помечена именем, ролями, мощностью.

Между актерами допустимо отношение обобщения (рисунок 5.4), означающее, что экземпляр потомка может взаимодействовать с такими же разновидностями экземпляров элементов Use Case, что и экземпляр родителя.

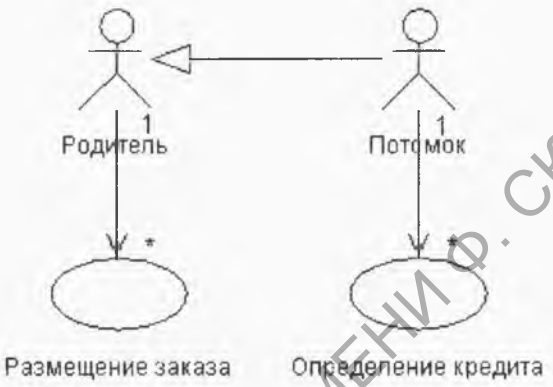

Рисунок 5.4 – Отношение обобщения между актерами

Между элементами Use Case определены отношение обобщения и две разновидности отношения зависимости — включения и расширения.

Отношение обобщения (рисунок 5.5) фиксирует, что потомок наследует поведение родителя. Кроме того, потомок может дополнить или переопределить поведение родителя. Элемент Use Case, являющийся потомком, может замещать элемент Use Case, являющийся родителем, в любом месте диаграммы.

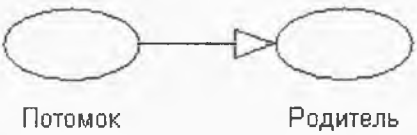

Рисунок 5.5 – Отношение обобщения между элементами Use Case

Отношение включения (рисунок 5.6) между элементами Use Case означает, что базовый элемент Use Case *явно* включает поведение дру-

гого элемента Use Case в точке, которая определена в базе. Включаемый элемент Use Case никогда не используется самостоятельно — его конкретизация может быть только частью другого, большего элемента Use Case. Отношение включения является примером отношения делегации. При этом в отдельное место (включаемый элемент Use Case) помещается определенный набор обязанностей системы. Далее остальные части системы могут агрегировать в себя эти обязанности (при необходимости).

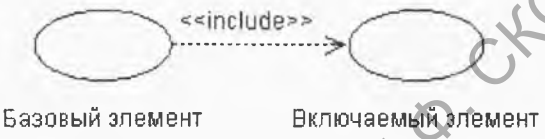

Рисунок 5.6 – Отношение включения между элементами Use Case

Отношение расширения (рисунок 5.7) между элементами Use Case означает, что базовый элемент Use Case *неявно* включает поведение другого элемента Use Case в точке, которая определяется косвенно расширяющим элементом Use Case. Базовый элемент Use Case может быть автономен, но при определенных условиях его поведение может расширяться поведением из другого элемента Use Case. Базовый элемент Use Case может расширяться только в определенных точках — точках расширения. Отношение расширения применяется для моделирования выбираемого поведения системы.

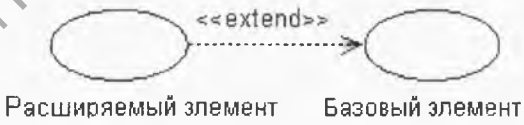

Рисунок 5.7 – Отношение расширения между элементами Use Case

Таким способом можно отделить обязательное поведение от необязательного поведения. Например, можно использовать отношение расширения для отдельного подпотока, который выполняется только при определенных условиях, находящихся вне поля зрения базового

элемента Use Case. Наконец, можно моделировать отдельные потоки, вставка которых в определенную точку управляется актером.

Пример простейшей диаграммы Use Case, в которой использованы отношения включения и расширения, приведен на рисунке 5.8.

На рисунке 5.9 приведена диаграмма Use Case для обслуживания заказчика. На данной диаграмме проиллюстрировано использование отношений включения и расширения, а также отношения обобщения.

Рисунок 5.8 – Простейшая диаграмма Use Case для банка

Рисунок 5.9 – Диаграмма Use Case для обслуживания заказчика

6 Модели реализации объектно–ориентированных программных систем

Тема 1 Основные понятия разработки программных систем

1.1 Диаграммы реализации
1.2 Компоненты
1.3 Интерфейсы
1.4 Компоновка системы

1.1 Диаграммы реализации

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], [9], [10], [11].

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.

1.2 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 (рисунок 1.1).

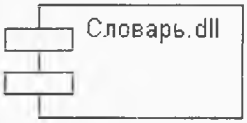

Рисунок 1.1 – Обозначение компонента

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. Различия компонентов и классов:

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.

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 кооперaций. Итак, в скафандре – физической реализации – располагается набор логики.

Теперь уместно перейти к обсуждению интерфейсов.

1.3 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 рисунке 1.2, интерфейс изображается в форме пиктограммы. Компонент Образ.java, который реализует интерфейс, соединяется со значком интерфейса (кружком) <НаблюдательОбраза> простой линией. Компонент РыцарьПечальногоОбраза.java, который использует интерфейс, связан с ним отношением зависимости.

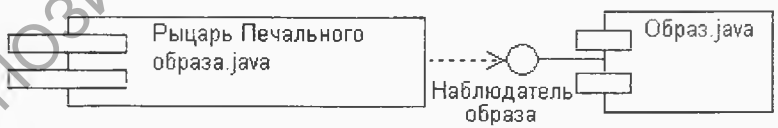

Рисунок 1.2 – Представление интерфейса в форме пиктограммы

Второй способ представления интерфейса проиллюстрирован на рисунке 1.3. Здесь используется развернутая форма изображения интерфейса, в которой могут показываться его операции. Класс или ком-

понент, который реализует интерфейс, подключается к нему отношением реализации.

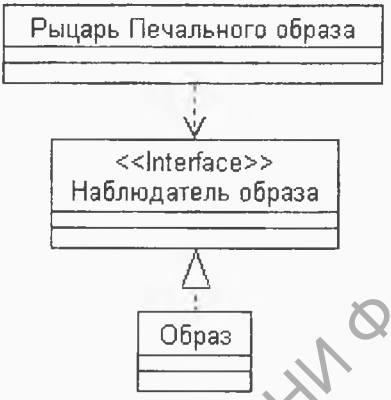

Рисунок 1.3 – Развернутая форма представления интерфейса

Класс или компонент, который получает доступ к услугам другого компонента через интерфейс, по-прежнему подключается к интерфейсу отношением зависимости.

По способу связи компонента с интерфейсом различают:

– экспортируемый интерфейс — тот, который компонент или класс реализует и предлагает как услугу клиентам (приведен на рисунке 1.4);

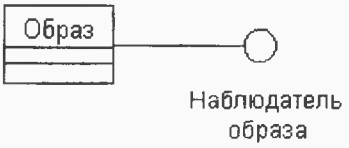

Рисунок 1.4 – Экспортируемый интерфейс

– импортируемый интерфейс — тот, который компонент или класс использует как услугу другого компонента или класса (приведен на рисунке 1.5).

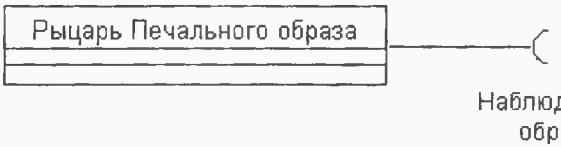

Рисунок 1.5 – Импортируемый интерфейс

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.

1.4 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 создателей аппаратного обеспечения?

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 следующих особенностях 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.

Тема 2 Компонентные диаграммы

2.1 Разновидности компонентов

2.2 Моделирование программного текста системы
2.3 Моделирование реализации системы

2.1 Разновидности компонентов

Мир современных компонентов достаточно широк и разнообразен. В языке UML для обозначения новых разновидностей компонентов используют механизм стереотипов. Приведем стандартные стереотипы, предусмотренные в UML для компонентов (таблица 2.1).

Таблица 2.1 – Разновидности компонентов

Стереотип	Семантика
<<executable>>	Компонент, который может выполняться в физическом узле
<<library>>	Статическая или динамическая объектная библиотека
<<file>>	Компонент, который представляет файл, содержащий исходный код или данные
<<table>>	Компонент, который представляет таблицу базы данных
<<document>>	Компонент, который представляет документ

В языке UML не определены пиктограммы для перечисленных стереотипов.

2.2 Моделирование программного текста системы

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.

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-файлах) 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.

В качестве примера на рисунке 2.1 приведена компонентная диаграмма, где изображены файлы исходного кода, используемые для построения библиотеки Визуализациями. Имеются четыре заголовочных файла (Визуализация.h, ВизЯдро.h, Прил.h, ТабЦветов.h), которые представляют исходный код для спецификации определенных классов.

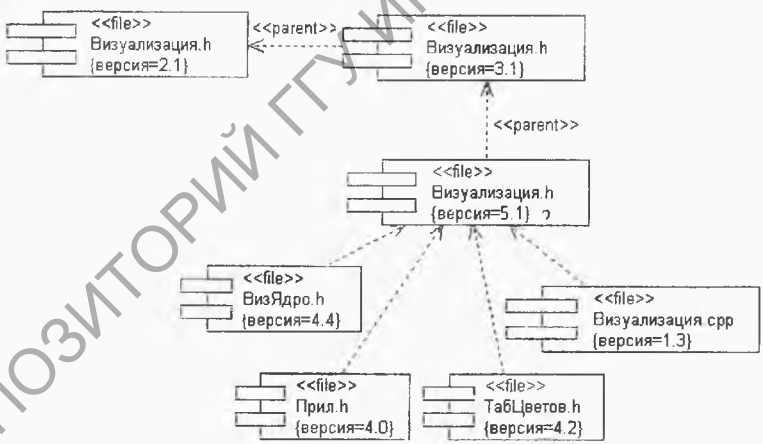

Рисунок 2.1 – Моделирование исходного кода

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.

2.3 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.

Например, на рисунке 2.2 показана часть реализации системы, группируемая вокруг исполняемого элемента Видеоклип.exe.

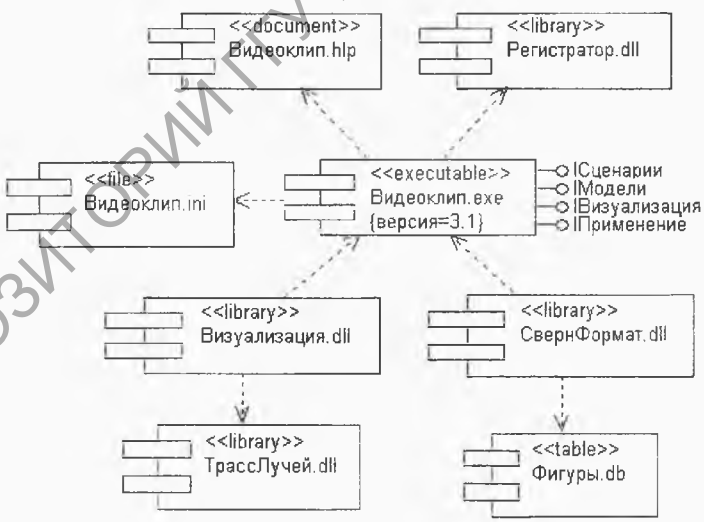

Рисунок 2.2 – Моделирование реализации системы

Здесь изображены четыре библиотеки (Регистратор.dll, СвернФормат.dll, Визуализация.dll, ТрассЛучей.dll), один документ (Видеоклип, hlp), один простой файл (Видеоклип.ini),а также таблица базы данных (Фигуры.tbl). В диаграмме указаны отношения зависимости, существующие между компонентами.

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

Литература

1 Pressman, R. S. Software Engineering: A Practioner's Approach / R. S. Pressman. – McGraw–Hill, 2000. – 943 с.

2 Соммервилл, И. Инженерия программного обеспечения / И. Соммервил. – М., 2002. – 624 с.

3 Буч, Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++ / Г. Буч. – СПб., 1998. – 560 с.

4 Graham, I. Object–Oriented Methods. Principles & Practice. / I. Graham. – Addison-Wesley, 2001. – 853 с.

5 Jacobcon, I. Object-Oriented Software Engineering / I. Jacobcon, M. [et. al.] – Addison–Wesley, 1993. – 528 с.

6 Page-Jones, M. Fundamentals of Object-Oriented Design in UML / M. Page-Jones. – Addison–Wesley, 2001. – 479 с.

7 Rumbaugh, J. Object Oriented Modeling and Design / J. Rumbaugh, M. [et. al.] – Prentice Hall, 1991. – 500 с.

8 Буч, Г. Язык UML. Руководство пользователя / Г. Буч, Д. Рамбо Д., А. Джекобсон. – М., 2000. – 432 с.

9 Rumbaugh, J. The Unified Modeling Language Reference Manual / J. Rumbaugh, I. Jacobson, G. Booch – Addison–Wesley, 1999. – 567 с.

10 Ambler, S. W. The Object Primer / S. W. Ambler. – Cambrige University Press, 2001. – 541 с.

11 Терри, К. Визуальное моделирование с помощью IBM® Rational® Software Architect и UML™ / К. Терри, Д. Палистрант. – М., 2007. – 192 с.

12 Орлов, С. А. Технологии разработки программного обеспечения : учебник / С. А. Орлов. – СПб., 2004. – 527 с.

Учебное издание

ПОМАЗ Андрей Сергеевич

СИСТЕМЫ АВТОМАТИЗИРОВАННОГО ПРОЕКТИРОВАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

ТЕКСТЫ ЛЕКЦИЙ

для студентов математических специальностей

Редактор *В. И. Шкредова*
Корректор *В. В. Калугина*

Лицензия №02330/0133208 от 30.04.04.
Подписано в печать 18.09.08. Формат 60 x 84 1/16.
Бумага писчая №1. Гарнитура «Таймс». Усл. печ.л. 6,39.
Уч.-изд. л.6,89. Тираж 100 экз. Заказ № 3

2160 – 00

Отпечатано с оригинал-макета на ризографе
учреждения образования
«Гомельский государственный университет
имени Франциска Скорины»
Лицензия № 02330/0056611 от 16.02.04.
246019, г. Гомель, ул. Советская, 104