Донбасская государственная машиностроительная академия
Кафедра автоматизации производственных процессов

ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ СЛОЖНЫХ СИСТЕМ

Объектно-ориентированное моделирование сложных систем

Конспект лекций

(для студентов специальности
“Автоматизированное управление технологическими процессами”
заочного обучения)

Утверждено
на заседании кафедры АПП
_____протокол №__

УДК 658.01

Технология программирования сложных систем. Объектно-ориентированное моделирование сложных систем. Конспект лекций (для студентов специальности “Автоматизированное управление технологическими процессами.

В первой части конспекта лекций изложены принципы и методы моделирования, используемые в инженерном цикле разработки программных систем. Показаны последние научные и практические достижения в области технологии моделирования программных систем, приведены примеры решения наиболее важных вопросов, возникающих при моделировании сложных систем.

Редактор – без редактирования

Содержание

ВВЕДЕНИЕ

Для решения задач комплексной автоматизации управления промышленным предприятием применяются сложные программные системы с иерархической архитектурой, в которой технологический, производственный и административный уровни программного обеспечения должны охватывать определенные функции и иметь соответствующие средства взаимодействия с разнообразными программными приложениями. Основным направлением в реализации этих требований является ***развитие общей архитектурной платформы для объектно-ориентированных приложений на основе открытых спецификаций.***

Решение этой проблемы невозможно без знания теоретических основ объектно-ориентированного подхода и методов проектирования сложных программных систем.

Дисциплина «Технология программирования сложных систем» относится к циклу специальных дисциплин и обеспечивает системное восприятие современной концепции объектно-ориентированного подхода в моделировании и разработке программных систем.

Дисциплина состоит из двух частей:

1. Объектно-ориентированное моделирование сложных систем.

2. Унифицированный процесс разработки программных систем.

Цель дисциплины – освоение современных принципов, методов, инструментальных средств и технологий проектирования программного обеспечения сложных компьютерных систем.

В процессе изучения дисциплины предусмотрено выполнение лабораторных и практических работ, которые позволят сформировать умения разрабатывать объектно-ориентированные модели сложных программных систем с применением универсального языка моделирования UML и генерировать программный код с применением CASE-технологий.

1 СЛОЖНЫЕ СИСТЕМЫ И ПРОБЛЕМЫ ИХ МОДЕЛИРОВАНИЯ

1.1 Структурная сложность систем

Становление и развитие сложных больших систем – это одна из объективных и прогрессивных особенностей нынешнего этапа научно-технической революции, а создание таких систем требует адекватных, то есть ***системных*** методов [1].

Существует множество программ, которые задумываются, разрабатываются, сопровождаются и используются одним и тем же человеком. Такие программы относятся к решению прикладных задач и не являются сложными.

Предметом данной дисциплины являются ***промышленные программные продукты***, которые применяются, например, для решения следующих задач:

- управление событиями физического мира, для которых ресурсы времени ограничены (например, технологический уровень производства);
- задачи поддержания целостности информации объемом в сотни тысяч записей при параллельном доступе к ней с запросами и обновлениями (например, управление ресурсами производства);
- системы управления и контроля реальных процессов (например, диспетчеризация воздушного или железнодорожного транспорта).

Системы подобного типа обычно имеют большой жизненный цикл и предназначены для большого количества пользователей.

При разработке промышленных программ применяется та, или иная среда разработки, которая упрощает создание приложений в конкретных областях.

Существенная черта промышленной программы – высокий уровень сложности. Один разработчик практически не в состоянии охватить все аспекты такой системы.

Сложность вызывается четырьмя основными причинами:

- сложностью реальной предметной области, из которой исходит заказ на разработку программы;
- трудностью управления процессом разработки большого количества программ и приложений, которые должны, в конечном итоге, решать одну сложную задачу;
- необходимостью обеспечить достаточную гибкость программы;
- неудовлетворительными способами описания поведения больших дискретных систем.

Определены [2] ***пять общих признаков сложной системы***:

1. "Сложные системы часто являются иерархическими и состоят из взаимозависимых подсистем, которые в свою очередь также могут быть разделены на подсистемы, и т.д., вплоть до самого низкого уровням".

2. "Выбор, какие компоненты в данной системе считаются элементарными, относительно произволен и в большой степени оставляется на усмотрение исследователя".

3. "Внутрикомпонентная связь обычно сильнее, чем связь между компонентами. Это обстоятельство позволяет отделять "высокочастотные" взаимодействия внутри компонентов от "низкочастотной" динамики взаимодействия между компонентами".

4. "Иерархические системы обычно состоят из немногих типов подсистем, по-разному скомбинированных и организованных".

5. "Любая работающая сложная система является результатом развития работавшей более простой системы... Сложная система, спроектированная "с нуля", никогда не заработает. Следует начинать с работающей простой системы".

Проблемы, которые необходимо решить с помощью программного обеспечения, неизбежно содержат сложные элементы, множество различных, порой взаимоисключающих требований. На этапе ***анализа проблемы*** первые сложности обычно возникают из-за "нестыковки" между пользователями системы и ее разработчиками – пользователи смутно представляют и с трудом

могут объяснить, что нужно от будущей программной системы. Это в основном происходит из-за того, что каждый из участников процесса специализируется в своей области. У пользователей и разработчиков разные взгляды на сущность проблемы, и они делают различные выводы о возможных путях ее решения. Даже если пользователь точно знает, что ему нужно, очень трудно однозначно зафиксировать все его требования.

Дополнительные сложности возникают в результате изменений требований к программной системе уже ***в процессе разработки***. В основном требования корректируются из-за того, что само осуществление программного проекта часто изменяет проблему. Рассмотрение первых результатов (схем, прототипов) позволяет пользователям лучше понять и отчетливей сформулировать то, что им действительно нужно. В то же время этот процесс повышает квалификацию разработчиков в предметной области и позволяет им задавать более осмысленные вопросы, которые проясняют тёмные места в проектируемой системе.

Большая программная система – это крупное капиталовложение. В процессе её использования необходимо обеспечить ***сопровождение программного обеспечения, его эволюцию и сохранение эффективности***. При этом под *сопровождением* понимается устранение ошибок, а под *эволюцией* - внесение изменений в систему в ответ на изменившиеся к ней требования.

Основная задача разработчиков состоит в создании ***иллюзии простоты***, в защите пользователей от сложности описываемого предмета или процесса.

Размер исходных текстов программной системы не является её достоинством. Сегодня обычными стали программные системы, размер которых исчисляется десятками тысяч строк на языках высокого уровня. Ни один человек никогда не сможет полностью понять такую систему. Даже если правильно разложить ее на составные части, мы получим сотни, а иногда и тысячи отдельных модулей. Поэтому такой объем работ потребует привлечения команды разработчиков. Однако при этом будут возникать значительные трудности, связанные с ***организацией коллективной разработки***. Чем больше

разработчиков, тем сложнее связи между ними и тем сложнее координация, особенно если участники работ географически удалены друг от друга. Таким образом, при коллективном выполнении проекта главной задачей руководства является поддержание единства и целостности разработки.

Исполнение программы осуществляется на компьютере, то есть в системе с дискретными состояниями. Дискретные системы по самой своей природе имеют конечное число возможных состояний, хотя в больших системах это число будет очень велико. Переходы между дискретными состояниями не могут моделироваться непрерывными функциями. Каждое событие, внешнее по отношению к программной системе, может перевести ее в новое состояние, и, более того, переход из одного состояния в другое ***не всегда детерминирован***. При неблагоприятных условиях внешнее событие может нарушить текущее состояние системы из-за того, что ее создатели не смогли предусмотреть все возможные варианты. Это требует обязательного ***тестирования программных*** систем, однако всеобъемлющее тестирование провести невозможно. Поэтому и моделирования поведения больших дискретных систем, и тестирование ограничивается разумным уровнем уверенности в их правильности.

1.2 Базовая парадигма программной системы – жизненный цикл

Классический жизненный цикл называют *каскадной* или *водопадной моделью* этапов разработки программного продукта. Этим подчеркивается, что разработка программы рассматривается как последовательность этапов, причем переход на следующий иерархический (нижний) уровень происходит только после полного завершения работ на текущем уровне [2].

Разработка начинается на системном уровне и проходит через этапы анализа, проектирования, кодирования, тестирования и сопровождения (рис. 1.1).

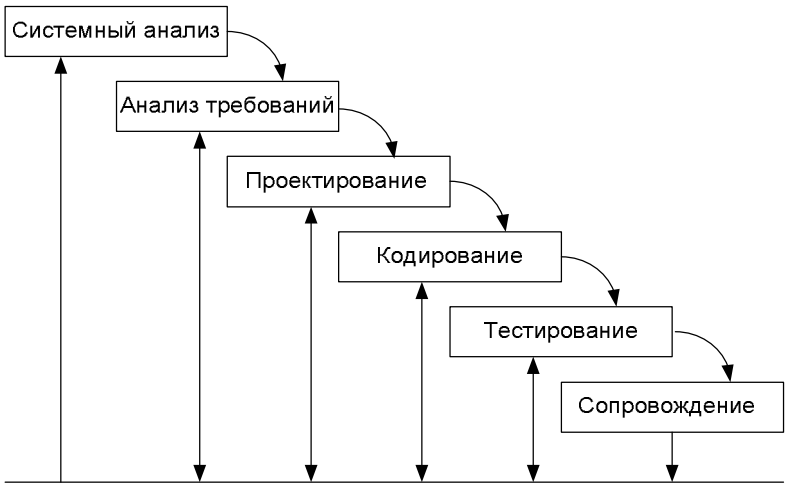

Рисунок 1.1 – Классический жизненный цикл разработки программы

Эти действия называются стандартным инженерным циклом.

Системный анализ определяет роль каждого элемента в компьютерной системе и взаимодействия элементов друг с другом. Поскольку программа является лишь частью большой системы, то анализ начинается с *определения требований* ко всем системным элементам.

Анализ требований относится к программному элементу – программному обеспечению. На этом этапе уточняются и детализируются его функции, характеристики и интерфейс. Все определения документируются в спецификации анализа. Здесь же происходит завершение планирования проекта.

На этапе ***проектирования*** создаются следующие представления:

- архитектуры программного обеспечения;
- модульной структуры программного обеспечения;
- алгоритмической структуры программного обеспечения;
- структуры данных;
- входного и выходного интерфейса (форм данных).

Исходные данные для проектирования содержатся в спецификации анализа. Таким образом, в ходе проектирования выполняется трансляция требований к программному обеспечению во множество проектных представлений. Основное внимание следует уделить качеству будущего

программного продукта. В ходе создания проекта определяются также риски, трудозатраты, объемы работ, формируются задачи и план-график работ.

Кодирование – это перевод результатов проектирования в текст на языке программирования.

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

Сопровождение – это внесение изменений в эксплуатируемое программное обеспечение с целью исправления ошибок, адаптации к изменениям внешней среды, усовершенствования программного обеспечения по требованиям заказчика. В процессе сопровождения происходит повторное применение всех предшествующих этапов жизненного цикла к существующей программе, однако только в отношении изменяемых компонентов программы.

Парадигма классического жизненного цикла дает план и временной график по всем этапам проекта, а также упорядочивает процесс программирования. Однако реальные проекты вынуждают иногда отклоняться от приведенной последовательности шагов, так как требования заказчика обычно определены лишь частично и результаты программирования доступны заказчику только в конце работ.

1.3 Проблемы декомпозиции сложных систем

При проектировании сложной программной системы необходимо разделять ее на все меньшие и меньшие подсистемы, каждую из которых можно совершенствовать независимо. Преимуществом такого подхода является то, что в этом случае мозг человека позволяет одновременно держать в уме информацию лишь о немногих частях системы.

В сложных системах между требованиями к системе, законом функционирования системы и алгоритмами реализации нет однозначного соответствия. Поэтому для декомпозиции систем следует применять следующие стратегии:

Функциональная декомпозиция. При этой декомпозиции ставится вопрос “Что делает система?”, а не “Как она работает?”. Наличие общих функций позволяет сгруппировать элементы в функциональные подсистемы.

Алгоритмическая декомпозиция (по физическому процессу). При управлении физическим процессом декомпозиция системы позволяет проанализировать соблюдение всех стадий процесса и учесть ограничения на каждой стадии при разработке алгоритмов.

Структурная (объектная) декомпозиция. Такая декомпозиция уместна, когда имеется стабильность границ подсистем, а между объектами одного типа (иерархического, логического, энергетического) существуют сильные связи.

Декомпозиция по жизненному циклу. Для такой декомпозиции необходимо выделить этапы цикла существования системы “от рождения до гибели” и определить, как будут изменяться законы функционирования подсистем в этих циклах. Очевидно, что в процессе эксплуатации системы она будет улучшаться путем внедрения оптимальных алгоритмов, тогда желательно такие изменения производить не во всей системе, а в отдельной, специально образованной подсистеме.

При проведении декомпозиции следует сначала выполнить функциональную декомпозицию, позволяющую разработать спецификацию требований к системе. Алгоритмическая декомпозиция позволит сконцентрировать внимание на порядке происходящих событий, а структурное разделение по объектам придаст особое значение агентам, которые являются либо объектами, либо субъектами действия.

Опыт показывает, что при знании функций системы полезнее начинать с ***объектной декомпозиции***. Такое начало поможет придать организованность сложной программной системе. Объектная декомпозиция имеет несколько очень важных преимуществ перед алгоритмической:

Объектная декомпозиция уменьшает размер программных систем за счет повторного использования общих механизмов, что приводит к существенной экономии затрат времени. Объектно-ориентированные системы более гибки и проще эволюционируют со временем, потому что их схемы базируется на устойчивых промежуточных формах.

Объектная декомпозиция существенно снижает риск при создании сложной программной системы, так как она развивается из меньших систем, в которых мы уже уверены.

Объектная декомпозиция помогает разобраться в сложной программной системе, предлагая разумные решения относительно выбора подпространства из большого пространства состояний.

При ***алгоритмической и объектной декомпозиции*** образуются два типа моделей – модель потока данных и модель объектов. В основе модели потока данных лежит разбиение по функциям, а в основе модели объектов – слабо сцепленные сущности, которые имеют собственные наборы данных, состояний и операций.

Таким образом, вопрос "Как наилучшим способом разделить сложную систему на подсистемы?" замыкается на проблеме создания таких моделей, которые фокусируют внимание на объектах, найденных в самой предметной области, и образуют то, что называют ***объектно-ориентированной декомпозицией.***

Декомпозиция системы обычно ограничивается на глубину 5-6 уровней. Функции, выделяемые при такой глубине, являются наиболее важными. Декомпозиция должна прекращаться тогда, когда выделенный элемент требует описания внутреннего алгоритма функционирования, несущественного для представления системы, или когда выделение объекта превращает систему в “черный ящик”.

В результате декомпозиции формируются строительные блоки программы – модули.

1.4 Особенности архитектурного представления модели

Цель моделирования заключается в том, чтобы стандартизовать этапы работы над проектом, а также автоматизировать процесс программирования.

При этом в процессе моделирования программы важно выявление ясной и относительно простой внутренней структуры программы, иногда называемой архитектурой. Моделирование подразумевает учет противоречивых требований. Его продуктами являются модели, позволяющие лучше понять структуру будущей системы, сбалансировать требования и наметить схему реализации.

Моделирование широко распространено во всех инженерных дисциплинах, в значительной степени из-за того, что оно реализует ***принципы декомпозиции, абстракции и иерархии***. Каждая модель описывает определенную часть рассматриваемой системы. С помощью моделей оценивается поведение системы в обычных и необычных ситуациях, контролируются неудачные решения и производятся соответствующие доработки.

Построение моделей крайне важно при проектировании сложных систем. Объектно-ориентированное проектирование предлагает богатый выбор моделей, которые представлены на рисунке 1.3.

Рисунок 1.3 – Объектно-ориентированные модели

Объектно-ориентированные модели проектирования отражают ***иерархию классов и объектов системы***. Эти модели покрывают весь спектр важнейших конструкторских решений, которые необходимо рассматривать при разработке сложной системы.

Обычно термин ***иерархия*** применяется в весьма приблизительном смысле. Сложные системы содержат ***много разных иерархий***. В программной системе металлорежущего станка, например, можно выделить программы расчета траектории движения, программы управления приводами, программы управления сменой инструмента и т.д. Такое разбиение дает структурную иерархию типа "быть частью" – первый тип декомпозиции. Однако систему можно разложить и совершенно другим способом. Например, если двигатель переменного тока – особый тип электродвигателя, то асинхронный двигатель – особый тип двигателя переменного тока. С другой стороны, понятие "двигатель" обобщает свойства, присущие всем двигателям. Такую декомпозицию можно назвать декомпозицией второго типа.

Таким образом, структурную иерархию системы можно рассмотреть с двух точек зрения – как иерархию первого и второго типа. Эти иерархии называются, соответственно, ***структурой классов*** и ***структурой объектов***. Структуры классов и объектов не являются независимыми – каждый элемент структуры объектов представляет специфический экземпляр определенного класса. Объектов в сложной системе обычно гораздо больше, чем классов. Если бы структура классов системы была неизвестна, пришлось бы повторять одни и те же сведения для каждого экземпляра класса. С введением структуры классов можно разместить в ней только общие свойства экземпляров. Так, например, в описании динамической системы могут применяться интегрирующие звенья с различными коэффициентами передачи. Эти интеграторы являются просто экземплярами обобщенного интегратора. Такой обобщенный компонент называется ***классом***, а отдельный интегратор схемы называется ***объектом*** этого класса.

Объединяя понятия ***структуры классов*** и ***структуры объектов*** **с** ***пятью признаками сложных систем***, можно представить практически все сложные системы одной (канонической) формой.

В результате такого представления образуются две ***ортогональные*** иерархии одной системы – классов и объектов. При этом каждая иерархия будет являться многоуровневой, причем классы и объекты более высокого уровня в этой системе будут построены из более простых классов и объектов, находящихся на нижнем уровне. Объекты одного уровня системы приобретают четко выраженные связи.

Объектно-ориентированное моделирование (ООМ) - это методология, основанная на представлении системы в виде совокупности объектов, каждый из которых является экземпляром определенного класса, а классы образуют иерархию наследования.

В данном определении можно выделить три части:

1) ООМ использует в качестве базовых элементов ***объекты,*** а не алгоритмы (иерархия "быть частью");

2) каждый объект является ***экземпляром*** какого-либо определенного ***класса;***

3) классы организованы ***иерархически***.

Объектный подход зарекомендовал себя как унифицирующая идея всей компьютерной науки, применимая не только в программировании, но также в проектировании интерфейса пользователя, баз данных и даже архитектуры компьютеров.

Структуры классов и объектов системы вместе называются ***архитектурой*** системы.

Выводы:

- Программам присуща сложность, которая нередко превосходит возможности человеческого разума.

- Задача разработчиков программных систем – создать у пользователя разрабатываемой системы иллюзию простоты.
- Сложные структуры часто принимают форму иерархий; полезны обе иерархии – и классов, и объектов.
- Познавательные способности человека ограничены, однако можно раздвинуть их рамки, используя декомпозицию, выделение абстракций и создание иерархий.
- Сложные системы можно исследовать, концентрируя основное внимание либо на объектах, либо на процессах; имеются веские основания использовать объектно-ориентированную декомпозицию, при которой мир рассматривается как упорядоченная совокупность объектов, которые в процессе взаимодействия друг с другом определяют поведение системы.
- Объектно-ориентированное моделирование и проектирование – метод, использующий объектную декомпозицию.
- Объектно-ориентированный подход имеет свою систему условных обозначений и предлагает богатый набор логических и физических моделей, с помощью которых можно получить представление о различных аспектах рассматриваемой системы.

2 ПРИНЦИПЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПОДХОДА В МОДЕЛИРОВАНИИ

2.1 Принципы моделирования

Модель – это упрощенное представления о реальности. Модель – это чертеж, схема, план, которые позволяют лучше понять систему на определенном уровне абстракции.

Моделирование позволяет решить четыре различные задачи:

- Визуализировать систему в её текущем или желаемом для нас состоянии.
- Определить структуру или поведение системы.
- Получить шаблон для автоматического проектирования программы.
- Документировать принятые решения.

Моделирование предназначено не только для создания больших систем – оно полезно в любых случаях. ***Однако моделировать сложную систему необходимо обязательно, поскольку иначе её невозможно воспринять как единое целое.***

Длительный опыт использования моделирования позволил сформулировать четыре основных принципа моделирования.

Во-первых, ***выбор модели оказывает определяющее влияние на подход к решению проблемы и на то, как будет выглядеть это решение***. Правильно выбранная модель позволяет осветить проблемы разработки и проникнуть в самую суть задачи, что при ином подходе было бы попросту невозможно. Ошибочно построенная модель будет заострять внимание на несущественных вопросах.

Если смотреть на систему глазами разработчика баз данных, то основное внимание будет уделяться моделям "сущность-связь", где поведение инкапсулировано в памяти и хранимых процедурах. Структурный аналитик, скорее всего, создал бы модель, в центре которой находятся алгоритмы и

передача данных от одного процесса к другому. Результатом труда разработчика, пользующегося объектно-ориентированным методом, будет система, архитектура которой основана на множестве классов и образцах взаимодействия, определяющих, как эти классы действуют совместно. Любой из этих вариантов может оказаться подходящим для данного приложения и методики разработки, хотя опыт подсказывает, что объектно-ориентированная точка зрения более эффективна при создании гибких архитектур, даже если система должна будет работать с большими базами данных или производить сложные математические расчеты. При этом надо учитывать, что различные точки зрения на мир приводят к созданию различных систем, со своими преимуществами и недостатками.

Второй принцип формулируется так: ***каждая модель может быть воплощена с разной степенью абстракции.***

При строительстве небоскреба может возникнуть необходимость показать его с высоты птичьего полета, например, чтобы с проектом могли ознакомиться инвесторы. В других случаях, наоборот, требуется самое детальное описание. То же происходит и при моделировании программного обеспечения. Иногда простая и быстро созданная модель пользовательского интерфейса – это самый подходящий вариант. В других случаях приходится работать на уровне битов, например, когда специфицируются межсистемные интерфейсы или устраняются узкие места в сети. В любом случае лучшей моделью будет та, которая позволяет выбрать уровень детализации в зависимости от того, кто и с какой целью на нее смотрит. Для аналитика или конечного пользователя наибольший интерес представляет вопрос "что", а для разработчика – вопрос "как". В обоих случаях необходима возможность рассматривать систему на разных уровнях детализации в разное время.

Третий принцип: ***лучшие модели - те, что ближе к реальности***.

Физическая модель здания, которая ведет себя не так, как изготовленная из реальных материалов, имеет лишь ограниченную ценность. Математическая модель самолета, для которой предполагаются идеальные условия работы и

безупречная сборка, может и не обладать некоторыми характеристиками, присущими настоящему изделию, что в ряде случаев приводит к фатальным последствиям. Лучше всего, если модели будут во всем соотноситься с реальностью, а там, где связь ослабевает, должно быть понятно, в чем заключается различие и что из этого следует. Поскольку модель всегда упрощает реальность, задача в том, чтобы это упрощение не повлекло за собой какие-либо существенные потери.

Возвращаясь к программному обеспечению, можно сказать, что "ахиллесова пята" структурного анализа – различие принятой в нем модели и модели системного проекта. Если этот разрыв не будет устранен, то поведение созданной системы с течением времени начнет все больше отличаться от задуманного. При объектно-ориентированном подходе можно объединить все почти независимые представления системы в единое семантическое целое.

Четвертый принцип заключается в том, что ***нельзя ограничиваться созданием только одной модели. Наилучший подход при разработке любой нетривиальной системы – использовать совокупность нескольких моделей, почти независимых друг от друга.***

При разработке конструкции здания никакой отдельный комплект чертежей не поможет прояснить до конца все детали. Понадобятся, как минимум, виды в разрезе, схемы электропроводки, центрального отопления и водопровода.

Ключевым определением здесь является выражение "*почти независимые*". В данном контексте оно означает, что модели могут создаваться и изучаться по отдельности, но вместе с тем остаются взаимосвязанными. Например, можно изучать только схемы электропроводки проектируемого здания, но при этом наложить их на поэтажный план и даже рассмотреть совместно с прокладкой труб на схеме водоснабжения.

Такой подход верен и в отношении объектно-ориентированных программных систем. Для понимания архитектуры подобной системы требуется несколько взаимодополняющих видов: вид с точки зрения прецедентов, или

вариантов использования (чтобы выявить требования к системе), вид с точки зрения проектирования (чтобы построить словарь предметной области и области решения), вид с точки зрения процессов (чтобы смоделировать распределение процессов и потоков в системе), вид с точки зрения реализации, позволяющий рассмотреть физическую реализацию системы, и вид с точки зрения развертывания, помогающий сосредоточиться на вопросах системного проектирования. Каждый из перечисленных видов имеет множество структурных и поведенческих аспектов, которые в своей совокупности составляют детальный чертеж программной системы.

В зависимости от природы системы некоторые модели могут быть важнее других. Так, при создании систем для обработки больших объемов данных более важны модели, обращающиеся к точке зрения статического проектирования. В приложениях, ориентированных на интерактивную работу пользователя, на первый план выходят представления с точки зрения статических и динамических прецедентов. В системах реального времени наиболее существенными будут представления с точки зрения динамических процессов. Наконец, в распределенных системах, таких как Web-приложения, основное внимание нужно уделять моделям реализации и развертывания.

2.2 Уменьшение сложности реальной системы средствами абстрагирования

Люди развили чрезвычайно эффективную технологию преодоления сложности – ***абстрагирование*** от нее. Будучи не в состоянии полностью воссоздать сложный объект, мы просто игнорируем не слишком важные детали и, таким образом, получаем обобщенную, идеализированную модель объекта.

Объектно-ориентированная модель имеет четыре главных элемента:

- абстрагирование;
- инкапсуляция;
- модульность;

- иерархия.

Эти элементы являются ***главными*** в том смысле, что без любого из них модель не будет объектно-ориентированной. Кроме главных, имеются еще три ***дополнительных элемента***:

- типизация;
- параллелизм;
- сохраняемость.

Называя их дополнительными, подразумевают, что они полезны в объектной модели, но не обязательны.

Абстрагирование является одним из основных методов, используемых для решения сложных задач. Абстрагирование проявляется в отвлечении от имеющихся различий при нахождении сходств между определенными объектами, ситуациями или процессами реального мира. Упрощенное описание или изложение системы подчеркивает детали, существенные для рассмотрения и использования, и опускает те, которые на данный момент несущественны. Идея квалифицируется как абстракция только тогда, когда она может быть изложена, понята и проанализирована независимо от механизма, который будет в дальнейшем принят для ее реализации. Суммируя эти разные точки зрения, получим следующее определение абстракции:

Абстракция выделяет существенные характеристики некоторого объекта, отличающие его от всех других видов объектов и, таким образом, четко определяет его концептуальные границы с точки зрения наблюдателя.

К этому полезно добавить дополнительный принцип, называемый ***принципом наименьшего удивления***, согласно которому абстракция должна охватывать все поведение объекта, но не больше и не меньше, и не привносить сюрпризов или побочных эффектов, лежащих вне ее сферы применимости.

Выбор правильного набора абстракций для заданной предметной области представляет собой главную задачу объектно-ориентированного моделирования.

Существует целый спектр абстракций, начиная с объектов, которые почти точно соответствуют реалиям предметной области, и кончая объектами, не имеющими право на существование:

- ***Абстракция сущности***. Объект представляет собой полезную модель некой сущности в предметной области.
- ***Абстракция поведения***. Объект состоит из обобщенного множества операций.
- ***Абстракция виртуальной машины***. Объект группирует операции, которые либо вместе используются более высоким уровнем управления, либо сами используют некоторый набор операций более низкого уровня.
- ***Произвольная абстракция***. Объект включает в себя набор операций, не имеющих друг с другом ничего общего.

Из этого набора наиболее предпочтительным типом абстракции являются ***абстракции сущности***, так как они прямо соответствуют сущностям предметной области.

Поведение объекта можно характеризовать ***услугами***, которые он оказывает другим объектам, и ***операциями***, которые он выполняет над другими объектами. Такой подход концентрирует внимание на ***внешних проявлениях объекта*** и приводит к идее ***контрактной модели*** программирования.

Контракт определяет ***ответственность*** объекта – это то поведение, за которое объект отвечает. Внешнее проявление объекта рассматривается с точки зрения его контракта с другими объектами, в соответствии с этим должно быть выполнено и его внутреннее устройство (часто во взаимодействии с другими объектами). Контракт фиксирует все обязательства, которые объект-сервер имеет перед объектом-клиентом.

Клиентом называется любой объект, использующий ресурсы другого объекта, называемого ***сервером***.

Каждая операция, предусмотренная этим контрактом, однозначно определяется ее формальными параметрами и типом возвращаемого значения.

Полный набор операций, которые клиент может осуществлять над другим объектом, вместе с правильным порядком, в котором эти операции вызываются, называется ***протоколом***. Протокол отражает все возможные способы, которыми объект может действовать или подвергаться воздействию, полностью определяя тем самым внешнее поведение абстракции со статической и динамической точек зрения.

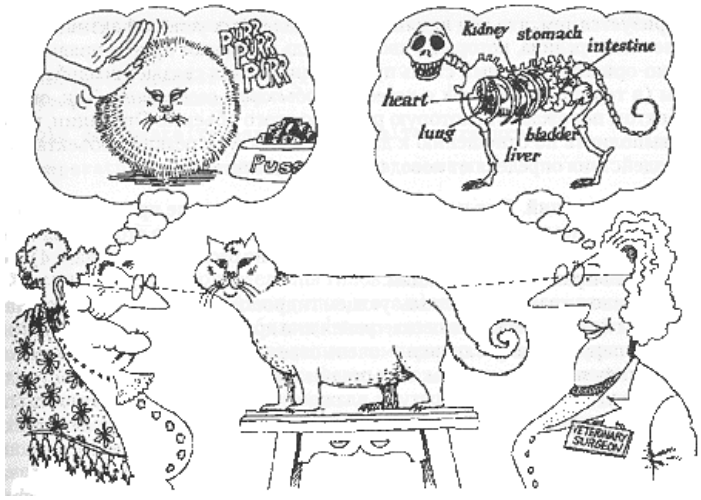

Рисунок 2.1 – Абстракция фокусируется на существенных с точки зрения наблюдателя характеристиках объекта [2]

Центральной идеей абстракции является понятие инварианта.

Инвариант - это некоторое логическое условие, значение которого (истина или ложь) должно сохраняться. Для каждой операции объекта можно задать ***предусловия*** (инварианты предполагаемые операцией) и ***постусловия*** (инварианты, которым удовлетворяет операция). Изменение инварианта нарушает контракт, связанный с абстракцией. В частности, если нарушено предусловие, то клиент не соблюдает свои обязательства и сервер не может выполнить свою задачу правильно. Если же нарушено постусловие, то свои обязательства нарушил сервер, и клиент не может более ему доверять. В случае нарушения какого-либо условия возбуждается исключительная ситуация, для разрешения которой должны быть предусмотрены альтернативы. Некоторые языки имеют средства для работы с исключительными ситуациями – объекты

могут возбуждать исключения, чтобы запретить дальнейшую обработку и предупредить о проблеме другие объекты.

Все абстракции обладают как статическими, так и динамическими свойствами. Например, файл как объект требует определенного объема памяти на конкретном устройстве, имеет имя и содержание. Эти атрибуты являются его статическими свойствами. Конкретные же значения каждого из перечисленных свойств динамичны и изменяются в процессе использования объекта – файл можно увеличить или уменьшить, изменить его имя и содержимое. В процедурном стиле программирования действия, изменяющие динамические характеристики объектов, составляют суть программы. Любые события связаны с вызовом подпрограмм и с выполнением операторов. Стиль программирования, ориентированный на правила, характеризуется тем, что под влиянием определенных условий активизируются определенные правила, которые в свою очередь вызывают другие правила, и т.д.

Объектно-ориентированный стиль программирования связан с воздействием на объекты. Так, операция над объектом порождает некоторую реакцию этого объекта. Операции, которые можно выполнить по отношению к данному объекту, и реакция объекта на внешние воздействия определяют поведение этого объекта.

Рассмотрим пример задания свойств абстракции.

Типичной задачей программирования задач автоматизации является контроль и регулирование. Одна из ключевых абстракций в этом процессе – датчик. В качестве примера возьмем датчик температуры, установленный в определенном месте. Зададим обязанности датчика – измерение температуры и сообщение её по запросу клиента. Зададим также возможные действия клиента – клиент может калибровать датчик и получать от него текущее значение температуры.

Для демонстрации описания абстрактного датчика температуры, использован язык C++.

```
// Температура по Цельсию
typedef float Temperature;
// Число, однозначно определяющее положение датчика
typedef unsigned int Location;
class TemperatureSensor {
public:
      TemperatureSensor (Location);
      ~TemperatureSensor();
      void calibrate(Temperature actualTemperature);
      Temperature currentTemperature() const;
private:
};
```

Одним из атрибутов класса является ***видимость класса***. Видимость класса может быть разделена на три части:

- закрытую (**private**), видимую только для самого класса;
- защищенную (**protected**), видимую также и для подклассов;
- открытую (**public**), видимую для всех.

Здесь два оператора определения типов **Temperature** и **Location** вводят удобные псевдонимы для простейших типов, что позволяет выражать абстракции на языке предметной области

Temperature – это числовой тип данных в формате с плавающей точкой для записи температуры. Значения типа **Location** обозначают места, где могут располагаться температурные датчики.

Класс **TemperatureSensor** – это только спецификация датчика, это еще не объекты. Настоящая начинка класса скрыта в его закрытой (**private**) части. Собственно датчики – это ***экземпляры,*** которые нужно создать, прежде чем ими можно будет оперировать. Например, можно написать так:

```
Temperature temperature;
TemperatureSensor station1Sensor(1);
TemperatureSensor station2Sensor(2);
temperature = station1Sensor.currentTemperature();
```

Рассмотрим инварианты, связанные с операцией **currentTemperature:**

- предусловие включает предположение, что датчик установлен в нужном месте;
- постусловие задает условие, что датчик возвращает значение температуры в градусах Цельсия.

До сих пор датчик предполагался пассивным – кто-то должен запросить у него температуру, и тогда он ответит. Однако датчик может быть активным – следить за температурой и извещать другие объекты, когда ее отклонение от заданного значения превышает допустимую величину. Абстракция от этого меняется мало – всего лишь несколько иначе формулируется ответственность объекта. Какие новые операции нужны ему в связи с этим? Обычной идиомой для таких случаев является обратный вызов. Клиент предоставляет серверу функцию обратного вызова, а сервер использует ее, когда считает нужным. Здесь можно записать, например:

```
class ActiveTemperatureSensor {
public:
ActiveTemperatureSensor (Location,
void (*f)(Location, Temperature));
    ~ActiveTemperatureSensor();
    void calibrate(Temperature actualTemperature);
    void establishSetpoint(Temperature setpoint,
        Temperature delta);
    Temperature currentTemperature() const;
private:
...
};
```

Новый класс **ActiveTemperatureSensor** стал лишь чуть сложнее, но вполне адекватно выражает новую абстракцию. Создавая экземпляр датчика, мы передаем ему при инициализации не только место, но и указатель на функцию обратного вызова, параметры которой определяют место установки и температуру. Новая функция установки **establishSetpoint** позволяет клиенту изменять порог срабатывания датчика температуры, а ответственность датчика состоит в том, чтобы вызывать функцию обратного вызова каждый раз, когда текущая температура **actualTemperature** отклоняется от **setpoint**

больше чем на **`delta`**. При этом клиенту становится известно место срабатывания и температура в нем, а дальше уже он сам должен знать, что с этим делать. Следует учесть, что клиент по-прежнему может запрашивать температуру по собственной инициативе.

Представленные здесь варианты протоколов показывают способы, которыми объект может действовать или подвергаться воздействию, а также внешнее поведение абстракции.

2.3 Инкапсуляция, как средство отделения интерфейса от реализации

Абстракция объекта всегда предшествует его реализации. При этом клиенту нет никакого дела до реализации класса, который его обслуживает, до тех пор, пока тот соблюдает свои обязательства. Когда решение о реализации принято, оно должно трактоваться как секрет абстракции, скрытый от большинства клиентов.

Абстракция и инкапсуляция дополняют друг друга – абстрагирование направлено на ***наблюдаемое поведение объекта***, а инкапсуляция – на ***внутреннее устройство объекта***. Чаще всего инкапсуляция выполняется посредством скрытия информации, то есть маскировкой всех внутренних деталей, не влияющих на внешнее поведение. Обычно скрываются и внутренняя структура объекта, и реализация его методов.

Инкапсуляция, таким образом, определяет четкие границы между различными абстракциями. Так, например, при проектировании базы данных программа не должна зависеть от физического представления данных, а должна отражать только их логическое строение. Благодаря этому объекты становятся защищенными от деталей реализации тех объектов, которые находятся на более низком уровне.

Принято считать, что абстракция должна работать только вместе с инкапсуляцией. Практически это означает наличие двух частей в классе – интерфейса и реализации. ***Интерфейс*** отражает внешнее поведение объекта,

описывая ***абстракцию поведения*** всех объектов данного класса. Внутренняя ***реализация*** описывает представление этой абстракции и механизмы достижения желаемого поведения объекта. Принцип разделения интерфейса и реализации соответствует сути вещей – в интерфейсной части собрано все, что касается взаимодействия данного объекта с любыми другими объектами, а реализация скрывает от других объектов все детали, не имеющие отношения к процессу взаимодействия объектов.

Итак, инкапсуляцию можно определить следующим образом:

Инкапсуляция – это процесс отделения друг от друга элементов объекта, определяющих его устройство и поведение; инкапсуляция служит для того, чтобы изолировать контрактные обязательства абстракции от их реализации.

Одной из ключевых абстракций в системах автоматизации является исполнительное устройство. Возьмем для примера нагреватель, поддерживающий заданную температуру в помещении. Нагреватель является абстракцией низкого уровня, поэтому можно ограничиться всего тремя действиями с этим объектом – включение, выключение и запрос состояния. Нагреватель не должен отвечать за поддержание температуры, это будет поведением *более высокого уровня*, реализуемым совместно – нагревателем, датчиком температуры и еще одним объектом. Однако поведение объекта *более высокого уровня* основывается на простом поведении нагревателя и датчика, к которым добавлено некоторое другое поведение, например, запаздывание или гистерезис. Благодаря добавленному поведению можно обойтись без частых включений и выключений нагревателя в состояниях, близких к граничным. Принятие такого решения о разделении ответственности позволяет сделать каждую абстракцию более цельной.

Сначала установим, как обычно, типы операций:

```
// Булевский тип
enum Boolean {FALSE, TRUE};
```

В дополнение к трем предложенным выше операциям (включение, выключение и запрос состояния), нужны обычные мета-операции создания и уничтожения объекта (конструктор и деструктор). Поскольку в системе может быть несколько нагревателей, необходимо каждому из них сообщить место, где он установлен, как это делалось с классом датчиков температуры **TemperatureSensor**. Представим класс **Heater** для абстрактных нагревателей в нотации C++:

```
class Heater {
public:
    Heater(Location);
    ~Heater();
    void turnOn();
    void tum0ff();
    Boolean is0n() const;
private:
};
```

Здесь имеется только то, что посторонним надо знать о классе **Heater**. Предположим далее, что управляющий компьютер соединен с датчиками и исполнительными устройствами с помощью последовательного интерфейса. Очевидно, что нагреватели будут коммутироваться с помощью блока реле, управляемого командами, которые поступают через последовательные интерфейсы. Для включения нагревателя передается текстовое имя команды, номер места нагревателя и еще одно число, используемое как сигнал включения нагревателя.

Класс, выражающий абстрактный последовательный порт, будет иметь следующее описание:

```
class SerialPort {
public:
    SerialPort();
    ~SerialPort();
    void write(char*);
    void write(int);
    static SerialPort ports[10];
private:
};
```

Экземпляры этого класса будут настоящими последовательными портами, в которые можно выводить строки и числа.

Добавим еще три параметра в класс **Heater**.

```
class Heater {
public:
...
protected:
     const Location repLocation;
     Boolean repIsOn;
     SerialPort* repPort;
};
```

Эти параметры **repLocation**, **repIsOn**, **repPort** образуют его инкапсулированное состояние. Правила **C++** таковы, что при компиляции программы, если клиент попытается обратиться к этим параметрам напрямую, будет выдано сообщение об ошибке.

Определим теперь реализации всех операций этого класса.

```
Heater::Heater(Location 1)
      : repLocation(1),
      repIsOn(FALSE),
      repPort(&SerialPort::ports[l]) {}
Heater::Heater() {}
void Heater::turnOn()
{
    if (!repls0n) {
         repPort->write("*");
         repPort->write(repLocation);
         repPort->write(1);
         repIsOn = TRUE;
    }
}
void Heater::turn0ff()
{
    if (repIsOn) {
         repPort->write("*");
         repPort->write(repLocation);
         repPort->write(0);
         repIsOn = FALSE;
    }
}
```

```
Boolean Heater::is0n() const
{
    return repIsOn;
}
```

Такой стиль реализации типичен для хорошо структурированных объектно-ориентированных систем – классы записываются экономно, поскольку их специализация осуществляется через подклассы.

Предположим, что по какой-либо причине изменилась архитектура аппаратных средств системы и вместо последовательных портов управление должно осуществляться через фиксированную область памяти. Нет необходимости изменять интерфейсную часть класса – достаточно переписать реализацию. Согласно правилам **C++**, после этого придется перекомпилировать измененный класс, но не другие объекты, если только они не зависят от временных и пространственных характеристик прежнего кода.

Разумная инкапсуляция локализует те особенности проекта, которые могут подвергнуться изменениям. По мере развития системы разработчики могут корректировать, например, длительность операции или объем памяти. Для этого необходимо изменить внутреннее представление объекта, чтобы реализовать более эффективные алгоритмы или оптимизировать алгоритм по критерию памяти, заменяя хранение данных их вычислением.

2.4 Модульность и иерархическая организация модели

Разделение программы на модули до некоторой степени позволяет уменьшить ее сложность. Однако гораздо важнее то, что внутри модульной программы создается множество хорошо определенных и документированных интерфейсов. Эти интерфейсы неоценимы для исчерпывающего понимания программы в целом. В языке **C++**, например, классы и объекты составляют логическую структуру системы, они помещаются в ***модули,*** образующие физическую структуру системы. Это свойство становится особенно полезным, когда система состоит из многих сотен классов.

Модульность – это разделение программы на фрагменты, которые компилируются по отдельности, но могут устанавливать связи с другими модулями. В **C++** раздельно компилируемые файлы также называются модулями. Реализация, то есть текст модуля, хранится в файлах с расширением **.c**. В программах на **C++** часто используются расширения **.ee, .cp** и **.cpp**.

Правильное разделение программы на модули является почти такой же сложной задачей, как выбор правильного набора абстракций. Для небольших задач допустимо описание всех классов и объектов в одном модуле. Однако для сложных программ лучшим решением будет сгруппировать в отдельный модуль логически связанные классы и объекты, оставив открытыми те элементы, которые необходимо видеть другим модулям. Следует учесть, что увлечение разбиением на модули может довести до абсурда, когда для внесения в проект изменений потребуется модифицировать и перекомпилировать сотни модулей.

Модуль – это фрагмент программного текста, который состоит из интерфейсной части и части-реализации. Модульность обеспечивает интеллектуальную возможность создавать сколь угодно сложную программу. Существует *оптимальное количество модулей*, которому соответствует минимальная стоимость программного обеспечения (рис. 2.2).

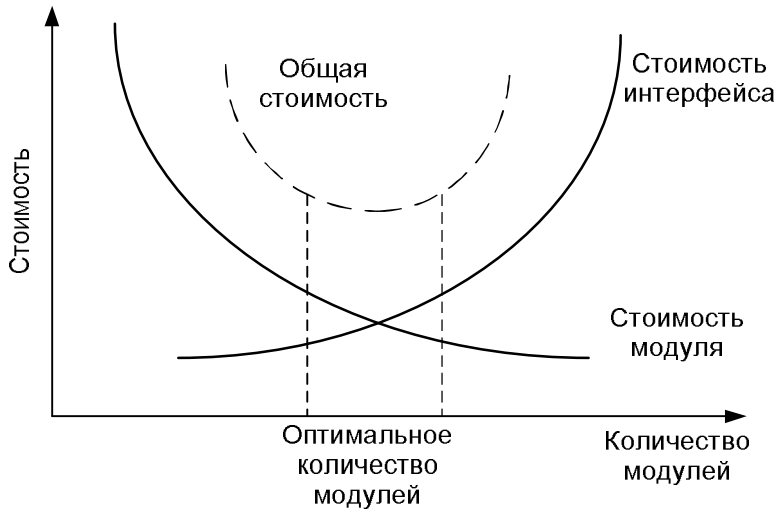

Рисунок 2.2 – Затраты на модульность

Это связано с тем, что при неглубокой декомпозиции образуются крупные модули, имеющие высокую сложность, а при глубокой декомпозиции создаются небольшие модули, но возрастает их количество.

Опыт показывает, что *оптимальный модуль* должен удовлетворять двум условиям:

- снаружи он проще, чем внутри;
- его проще использовать, чем построить.

В самой организации модуля следует придерживаться принципа информационной закрытости.

Информационная закрытость означает следующее:

- содержимое модулей должно быть скрыто друг от друга;
- все модули независимы, обмениваются только информацией, необходимой для работы;
- доступ к операциям и структурам данных модуля ограничен.

Благодаря принципу информационной закрытости модуля обеспечивается возможность разработки модулей различными коллективами, а также упрощается возможность модификации части программной системы.

Идеальный модуль играет роль ”черного ящика”, содержимое которого не видно клиентам.

Определенные требования существуют и относительно взаимной зависимости частей модуля друг от друга. Мерой взаимозависимости частей модуля является связность модуля. Существует 7 типов связности:

1. ***Связность по совпадению*** означает, что в модуле отсутствуют явно выраженные связи и каждая часть имеет свой поток управления.
2. ***Логическая связность*** означает, что части модуля объединены по принципу функционального подобия. Например, модуль состоит из разных подпрограмм обработки ошибок и клиент сам выбирает одну из них.
3. ***Временная связность*** – части модуля не связаны, но задействуются в один и тот же период работы системы. Например, модуль

инициализации содержит компоненты (части), которые связаны с работой всей системы и поэтому его переработка вызовет большие трудности.

4. ***Процедурная связность*** – части модуля участвуют в некотором сценарии. Например, имеется несколько таблиц, из которых в определенном порядке извлекаются данные, обрабатываются и снова размещаются в таблицах. Если в таком модуле заложить параметры таблиц, то любые изменения этих параметров потребуют модификации модуля в процессе сопровождения программы.
5. ***Коммуникативная связность*** – части модуля связаны по данным, то есть работают с одними и теми же данными, например, с данными зарплат сотрудников. Для удовлетворения различных клиентов обычно в таких модулях закладываются избыточные данные, что неудовлетворительно сказывается на качестве обслуживания клиентов необходимой информацией.
6. ***Информационная (последовательная) связность*** – выходные данные одной части модуля используются как входные данные в другой части. Элементы-обработчики модуля образуют конвейер для обработки данных. Сопровождение модулей с информационной связностью не вызывает трудностей.
7. ***Функциональная связность*** – части модуля вместе реализуют одну функцию. Например, при управлении оборудованием обычно применяется модуль автоматического режима, состоящий из программ вычисления ошибки, умножения ошибки на передаточную функцию регулятора, анализа ограничений выходного сигнала регулятора, компенсации нелинейностей и т.д. Приложения, построенные из функционально связанных модулей, сопровождать легче всего.

Следует учесть, что модули со связностью по совпадению, с логической и временной связностью являются результатом неправильной декомпозиции и

спланированной по ней архитектуры. Модуль с процедурной связностью можно отнести к результату небрежного планирования архитектуры программы.

Программист должен находить баланс между двумя противоположными тенденциями – стремлением скрыть информацию и необходимостью обеспечения видимости тех или иных абстракций в нескольких модулях. Для достижения этого следует стремиться строить модули так, чтобы объединить логически связанные абстракции и минимизировать взаимные связи между модулями. Исходя из этого, модульность определяется следующим образом:

Модульность – это свойство системы, которая была разложена на связные внутренне, но слабо связанные между собой модули.

Таким образом, принципы абстрагирования, инкапсуляции и модульности являются взаимодополняющими. Объект логически определяет границы определенной абстракции, а инкапсуляция и модульность делают их физически незыблемыми.

В процессе разделения системы на модули могут быть полезными два правила:

1. Модули служат в качестве ***элементарных и неделимых блоков программы***, которые могут использоваться в системе повторно. Поэтому при распределении классов и объектов по модулям необходимо это учитывать.
2. Многие компиляторы создают отдельный сегмент кода для каждого модуля. Поэтому могут появиться ***ограничения на размер модуля***. Динамика вызовов подпрограмм и расположение описаний внутри модулей может сильно повлиять на локальность ссылок и на управление страницами виртуальной памяти. При плохом разбиении процедур по модулям учащаются взаимные вызовы между сегментами, что приводит к потере эффективности кэш-памяти и частой смене страниц.

Таким образом, абстракция полезна для реализации наших умственных возможностей, направляя внимание на существенные характеристики системы,

инкапсуляция позволяет в какой-то степени убрать из поля зрения внутреннее содержание абстракций, а модульность упрощает задачу моделирования, объединяя логически связанные абстракции в группы. Но этого оказывается недостаточно.

Значительное упрощение в понимании сложных задач достигается за счет образования из абстракций иерархической структуры. Определим иерархию следующим образом:

Иерархия – это упорядочение абстракций, расположение их по уровням.

Основными видами иерархических структур применительно к сложным системам являются ***структура классов*** (иерархия "**`is-a`**") и ***структура объектов*** (иерархия "**`part of`**").

Важным элементом объектно-ориентированных систем и основным видом иерархии "**`is-a`**" является концепция наследования. Наследование означает такое отношение между классами (отношение родитель/потомок), когда один класс заимствует структурную или функциональную часть одного или нескольких других классов (соответственно, *одиночное* и *множественное наследование*). Иными словами, наследование создает такую иерархию абстракций, в которой подклассы наследуют строение от одного или нескольких суперклассов. Семантически, наследование описывает отношение типа "**`is-a`**". Например, медведь есть млекопитающее, дом есть недвижимость и "быстрая сортировка" есть сортирующий алгоритм. Таким образом, наследование порождает иерархию "обобщение-специализация", в которой подкласс представляет собой специализированный частный случай своего суперкласса. В наследственной иерархии общая часть структуры и поведения сосредоточена в наиболее общем суперклассе. В отсутствие наследования каждый класс становится самостоятельным блоком и должен разрабатываться "с нуля". Наследование позволяет вводить в обращение новые программы, разработанные на базе известных решений.

Выводы

- Абстракция определяет существенные характеристики некоторого объекта, которые отличают его от всех других видов объектов и, таким образом, абстракция четко очерчивает концептуальную границу объекта с точки зрения наблюдателя.
- Инкапсуляция – это процесс разделения устройства и поведения объекта; инкапсуляция служит для того, чтобы изолировать контрактные обязательства абстракции от их реализации.
- Модульность – это состояние системы, разложенной на внутренне связные и слабо связанные между собой модули.
- Иерархия – это ранжирование или упорядочение абстракций

3 ОБЪЕКТЫ, КАК МОДЕЛИ СУЩНОСТЕЙ РЕАЛЬНОЙ СИСТЕМЫ

3.1 Определение объектов

Объект определяется как осязаемая реальность, проявляющая четко выделяемое поведение. С точки зрения восприятия человеком объектом может быть:

- осязаемый и (или) видимый предмет;
- нечто, воспринимаемое мышлением;
- нечто, на что направлена мысль или действие.

Таким образом, объект моделирует часть окружающей действительности и существует во времени и пространстве.

Так, например, заводы разделяются на цеха – механические, металлургические, сборочные и т.д. Цеха подразделяются на участки, на каждом из которых установлено несколько единиц оборудования (штампы, прессы, станки). На производственных линиях можно увидеть множество емкостей с исходными материалами, из которых с помощью технологических процессов создаются готовые детали машин. Каждый осязаемый предмет может рассматриваться как объект. Токарный станок имеет четко очерченные границы, которые отделяют его от обрабатываемой на этом станке заготовки. Заготовки, в свою очередь, имеют четкие границы по отношению к оборудованию.

Существуют такие объекты, для которых определены явные концептуальные границы, но сами объекты представляют собой неосязаемые события или процессы. Например, химические процессы можно трактовать как объекты, так как они имеет четкие концептуальные границы, взаимодействует с другими объектами посредством упорядоченного и распределенного во времени набора операций и проявляет хорошо определенное поведение.

Другой пример относится к системе пространственного проектирования CAD/CAM. Два тела, например, сфера и куб, имеют, как правило, нерегулярное пересечение. Хотя эта линия пересечения не существует отдельно от сферы и

куба, она все же является самостоятельным объектом с четко определенными концептуальными границами.

Объекты могут быть осязаемыми, но иметь размытые физические границы, например, реки, туман или толпы людей. Существуют также понятия, явно не являющиеся объектами – время, красота, эмоции. Однако потенциально все перечисленное может характеризовать свойства, присущие объектам.

Объект обладает состоянием, поведением и идентичностью; структура и поведение схожих объектов определяет общий для них класс; термины «экземпляр класса» и «объект» взаимозаменяемы.

Поведение объекта определяется его историей – важна последовательность совершаемых над объектом действий. Такая зависимость поведения от событий и от времени объясняется тем, что у объекта есть внутреннее состояние. Для торгового автомата, например, состояние определяется суммой денег, опущенных до нажатия кнопки выбора. Другая важная информация – это набор воспринимаемых монет и запас, например, напитков.

Состояние объекта характеризуется перечнем (обычно статическим) всех свойств данного объекта и текущими (обычно динамическими) значениями каждого из этих свойств.

К числу свойств объекта относятся присущие ему или приобретаемые им характеристики, черты, качества или способности, делающие данный объект самим собой. Например, для лифта характерным является то, что он сконструирован для поездок вверх и вниз, а не горизонтально. Перечень свойств объекта является, как правило, статическим, поскольку эти свойства составляют неизменяемую основу объекта. Однако в ряде случаев состав свойств объекта может изменяться. Примером может служить робот с возможностью самообучения.

Все свойства имеют некоторые значения. Эти значения могут быть простыми количественными характеристиками, а могут ссылаться на другой объект. Состояние лифта может описываться числом 3, означающим номер этажа, на котором лифт в данный момент находится. Состояние торгового

автомата описывается в терминах других объектов, например, перечнем имеющихся в наличии напитков.

Тот факт, что всякий объект имеет состояние, означает, что всякий объект занимает также определенное пространство (физически или в памяти компьютера).

Предположим, что на языке **C++** нам нужно создать регистрационные записи о сотрудниках. Можно сделать это следующим образом:

```
struct PersonnelRecord {

     char name[100];
     int socialSecurityNumber;
     char department[10];
     float salary;
};
```

Каждый компонент в приведенной структуре обозначает конкретное свойство нашей абстракции регистрационной записи. Описание определяет не объект, а класс, поскольку оно не вводит какой-либо конкретный экземпляр. Для того чтобы создать объекты данного класса, необходимо написать следующее:

```
PersonnelRecord igor, gleb, anna, pavel, lena, denise;
```

В данном случае объявлено шесть различных объектов, каждый из которых занимает определенный участок в памяти. Хотя свойства этих объектов являются общими (их состояние представляется единообразно), в памяти объекты не пересекаются и занимают каждый свое место. На практике принято ограничивать доступ к состоянию объекта, а не делать его общедоступным, как в предыдущем определении класса. С учетом сказанного, изменим данное определение следующим образом:

```
class PersonnelRecord {
public:
     char* employeeName() const;
     int employeeSocialSecurityNumber() const;
     char* employeeDepartment() const;
protected:
```

```
    char name[100];
    int socialSecurityNumber;
    char department[10];
    float salary;
};
```

Новое определение несколько сложнее предыдущего, но по ряду соображений предпочтительнее. В частности, в новом определении реализация класса скрыта от других объектов. Если реализация класса будет в дальнейшем изменена, код придется перекомпилировать, но семантически клиенты не будут зависеть от этих изменений, то есть их код сохранится. Кроме того, решается также проблема занимаемой объектом памяти за счет явного определения операций, которые разрешены клиентам над объектами данного класса. В частности, всем клиентам дано право узнать имя, код социальной защиты и место работы сотрудника, но только особым клиентам, а именно, подклассам данного класса разрешено устанавливать значения указанных параметров. Только этим специальным клиентам разрешен доступ к сведениям о заработной плате (**`salary`**). Другое достоинство последнего определения связано с возможностью его повторного использования – механизм наследования позволяет повторно использовать абстракцию, а затем уточнить и специализировать ее.

Итак, все объекты в системе инкапсулируют некоторое состояние, и все состояние системы инкапсулировано в объекты. Однако инкапсуляция состояния объекта представляет собой только начало, которого недостаточно, чтобы можно было охватить полный смысл абстракций, вводимых при разработке. По этой причине необходимо разобраться, как объекты функционируют.

3.2 Поведение объектов и их состояния

Объекты не существуют изолированно, а подвергаются воздействию или сами воздействуют на другие объекты.

Поведение - это то, как объект действует и реагирует; поведение выражается в терминах состояния объекта и передачи сообщений.

Иными словами, поведение объекта – это его наблюдаемая и проверяемая извне деятельность, реализуемая через операции.

Операцией называется определенное воздействие одного объекта на другой с целью вызвать соответствующую реакцию.

Например, клиент может активизировать операции **`append`** и **`pop`** для того, чтобы управлять объектом-очередью (добавить или изъять элемент). Существует также операция **`length`**, которая позволяет определить размер очереди, но не может изменить это значение. В основном понятие *операции* над объектами совпадает с понятием *сообщение*, отличаясь только механизмом передачи. Часто эти два термина используются как синонимы.

Передача сообщений – это одна часть уравнения, задающего поведение.

Из приведенного определения следует, что состояние объекта также влияет на его поведение. Так, например, пользуясь торговым автоматом, клиент может сделать выбор, но поведение автомата будет зависеть от его состояния. Если в автомат не опущена достаточная сумма, то, скорее всего, ничего не произойдет. Если же сумма достаточна, автомат выдаст нам желаемое и тем самым изменит свое состояние. Итак, поведение объекта определяется выполняемыми над ним операциями и его состоянием, причем некоторые операции имеют побочное действие – они изменяют состояние. Концепция побочного действия позволяет уточнить определение состояния:

Состояние объекта представляет суммарный результат его поведения.

Наиболее интересны те объекты, состояние которых не статично, а изменяется и запрашивается операциями.

Операция – это услуга, которую класс может предоставить своим клиентам. На практике типичный клиент совершает над объектами операции пяти видов. Ниже приведены первые три (наиболее распространенные) операции:

Модификатор	Операция, которая изменяет состояние объекта.
Селектор	Операция, считывающая состояние объекта, но не меняющая состояния.
Итератор	Операция, позволяющая организовать доступ ко всем частям объекта в строго определенной последовательности.

Поскольку логика этих операций различна, полезно выбрать такой стиль программирования, который учитывает эти различия в коде программы. Две операции являются универсальными – они обеспечивают инфраструктуру, необходимую для создания и уничтожения экземпляров класса:

Конструктор	Операция создания объекта и/или его инициализации.
Деструктор	Операция, освобождающая состояние объекта и/или разрушающая сам объект.

В языке **C++** конструктор и деструктор составляют часть описания класса.

Таким образом, можно утверждать, что все методы – это операции, но не все операции – методы. Некоторые из них представляют собой свободные подпрограммы.

Объединяя определения состояния и поведения объекта, получим новое понятие – ответственности. ***Ответственности объекта имеют две стороны***, ***которые выражают смысл предназначения объекта и его место в системе***:

- знания, которые объект поддерживает;
- действия, которые объект может исполнить.

Ответственность понимается как совокупность всех услуг и всех контрактных обязательств объекта.

Таким образом, можно констатировать, что состояние и поведение объекта определяют исполняемые им роли, а те, в свою очередь, необходимы для выполнения ответственности данной абстракции.

3.3. Отношения между объектами

Отношения двух любых объектов основываются на предположениях, которыми один обладает относительно другого:

- об операциях, которые можно выполнять;
- об ожидаемом поведении.

Особый интерес для объектно-ориентированного анализа и проектирования представляют два типа иерархических соотношений объектов:

- связи;
- агрегация.

Связь – это физическое или концептуальное соединение между объектами, специфическое сопоставление, через которое клиент запрашивает услугу у объекта-сервера или через которое один объект находит путь к другому.

Участвуя в связи, объект может выполнять одну из следующих трех ролей:

- ***Актер*** (`Actor`). Этот объект может воздействовать на другие объекты, но сам никогда не подвергается воздействию других объектов. В определенном смысле этот объект соответствует понятию активный объект.
- ***Сервер***. Объект может только подвергаться воздействию со стороны других объектов, но он никогда не выступает в роли воздействующего объекта.
- ***Агент***. Такой объект может выступать как в активной, так и в пассивной роли. Как правило, объект-агент создается для выполнения операций в интересах какого-либо объекта-актера или другого агента.

Пример. В ряде технологических процессов требуется непрерывное регулирование температуры. Необходимо поднять температуру до заданного значения, выдержать заданное время и понизить до нормы. При этом профиль изменения температуры у разных процессов разный.

Абстракция нагрева имеет достаточно четкое поведение, что дает право на описание такого класса. Сначала определим тип. Значение типа задает прошедшее время в минутах.

```
// Число прошедших минут
typedef unsigned int Minute;
```

Теперь опишем сам класс **TemperatureRamp**, который по смыслу задает функцию времени от температуры:

```
class TemperatureRamp {
public:
    TemperatureRamp();
    virtual ~TemperatureRamp();
    virtual void clear();
    virtual void bind (Temperature, Minute);
    Temperature TemperatureAt (Minute);
protected:
...
};
```

Для задания поведения необходимо конкретизировать зависимость температуры от времени. Пусть, например, известно, что на 60-й минуте должна быть достигнута температура 250 C°, а на 180-й – 150 C°. Спрашивается, какой она должна быть на 120-й минуте? Это требует применения линейной интерполяции, так что требуемое от абстракции поведение усложняется. Вместе с тем, от нагревателя, поддерживающего требуемый профиль, этой абстракции не требуется.

Введем разделение понятий, при котором нужное поведение достигается взаимодействием трех объектов – экземпляра **TemperatureRamp**, нагревателя и контроллера. Класс **TemperatureController** можно определить так:

```
class TemperatureController {
public:
     TemperatureController(Location);
     ~TemperatureController();
     void process(const TemperatureRamp&);
     Minute schedule(const TemperatureRamp&) const;
private:
...
};
```

Тип **Location** был определен выше. Операция **process** обеспечивает основное поведение этой абстракции; ее назначение – передать график изменения температуры нагревателю, установленному в конкретном месте.

Например:

```
TemperatureRamp growingRamp;
TemperatureController rampController(7);
```

Теперь зададим пару точек и загрузим план в контроллер:

```
growingRamp.bind (250, 60);
growingRamp.bind(150, 180);
rampController.process(growingRamp);
```

В этом примере **rampController** – агент, ответственный за выполнение температурного плана. Он использует объект **growingRamp** как сервер. Эта связь проявляется хотя бы в том, что **rampController** явно получает **growingRamp** в качестве параметра одной из своих операций.

Применение операции **schedule** показывает, что объекты класса **TemperatureController** имеют достаточно интеллекта, чтобы определять расписание для конкретных профилей, и эта операция описывает дополнительное поведение абстракции **TemperatureController**. В некоторых энергоемких технологиях (например, плавка металлов) можно существенно выиграть, если учитывать остывание установки и тепло, остающееся после предыдущей плавки. Поскольку существует операция **schedule**, клиент может запросить объект **TemperatureController**, чтобы тот рекомендовал оптимальный момент запуска следующего нагрева.

При реализации системы необходимо обеспечить видимость связанных объектов. В предыдущем примере объект **rampController** видит объект **growingRamp**, поскольку оба они объявлены в одной области видимости. Кроме того, **growingRamp** передается объекту **rampController** в качестве параметра.

В принципе есть следующие четыре способа обеспечить видимость:

- Сервер глобален по отношению к клиенту.
- Сервер передан клиенту в качестве параметра операции.
- Сервер является частью клиента.
- Сервер локально порождается клиентом в ходе выполнения какой-либо операции.

Выбор того, или иного способа зависит от тактики проектирования.

Широко распространенным видом отношений является агрегация. В то время, как связи обозначают равноправные или "клиент-серверные" отношения между объектами, агрегация описывает отношения целого и части, приводящие к соответствующей иерархии объектов. В этом смысле агрегация – специализированный частный случай ассоциации. Так, объект **rampController** имеет связь с объектом **growingRamp** и атрибутом **h** класса **Heater** (нагреватель). В данном случае **rampController** – целое, а **h** – его часть. Другими словами, **h** – часть состояния **rampController**. Исходя из **rampController**, можно найти соответствующий нагреватель. Однако по **h** нельзя найти содержащий его объект (называемый также его контейнером), если только сведения о нем не являются случайно частью состояния **h**.

Агрегация может означать физическое вхождение одного объекта в другой, но не обязательно. Самолет состоит из крыльев, двигателей, шасси и прочих частей. С другой стороны, отношения акционера с его акциями – это агрегация, которая не предусматривает физического включения. Акционер монопольно владеет своими акциями, но они в него не входят физически. Это, несомненно, отношение агрегации, но скорее концептуальное, чем физическое.

Объект, являющийся атрибутом другого объекта (агрегата), имеет связь со своим агрегатом. Через эту связь агрегат может посылать ему сообщения.

Добавим в спецификацию класса **TemperatureController** описание нагревателя:

```
Heater h;
```

После этого каждый объект **TemperatureController** будет иметь свой нагреватель. В соответствии с определением класса Heater необходимо инициализировать нагреватель при создании нового контроллера, так как сам этот класс не предусматривает конструктора по умолчанию.

Определить конструктор класса **TemperatureController** можно следующим образом:

```
TemperatureController::TemperatureController(Location 1):h(1) {}
```

Выбирая одно из двух – связь или агрегацию, надо иметь в виду следующее. Агрегация иногда предпочтительнее, поскольку позволяет скрыть части в целом.

4 КЛАССЫ, КАК АБСТРАКЦИИ СУЩЕСТВЕННЫХ ХАРАКТЕРИСТИК ОБЪЕКТОВ

4.1 Общая характеристика классов

Понятия класса и объекта настолько тесно связаны, что невозможно говорить об объекте безотносительно к его классу. Однако существует важное различие этих двух понятий. В то время как объект обозначает конкретную сущность, определенную во времени и в пространстве, класс определяет лишь абстракцию существенного в объекте. Таким образом, можно говорить о классе "Металлорежущие станки", который включает характеристики, общие для всех металлорежущих станков.

Класс - это некое множество объектов, имеющих общую структуру и общее поведение.

Любой конкретный объект является просто экземпляром класса. Что же не является классом? Объект не является классом, хотя класс может быть объектом. Объекты, не связанные общностью структуры и поведения, нельзя объединить в класс, так как по определению они не связаны между собой ничем, кроме того, что все они объекты.

Важно отметить, что классы, как их понимают в большинстве существующих языков программирования, необходимы, но недостаточны для декомпозиции сложных систем. Некоторые абстракции так сложны, что не могут быть выражены в терминах простого описания класса. Например, графический интерфейс пользователя, база данных или система учета как целое, это явные объекты, но не классы. Можно попытаться выразить такие абстракции одним классом, но повторной используемости и возможности наследования не получится. Кроме того, такие большие абстракции приведут к созданию громоздкого интерфейса. Это плохая практика, так как большинство клиентов использует только малую часть из разработанного интерфейса.

В проектировании классов рекомендуется идея ***контрактного*** программирования.

Суть этой идеи заключается в том, что большие задачи разделяются на много маленьких задач, которые перепоручаются мелким субподрядчикам. По своей природе, ***класс – это генеральный контракт между абстракцией и всеми ее клиентами***. Выразителем обязательств класса служит его интерфейс, причем в языках с сильной типизацией потенциальные нарушения контракта можно обнаружить уже на стадии компиляции.

Идея контрактного программирования приводит к разграничению внешнего облика, то есть интерфейса, и внутреннего устройства класса, его реализации. Главное в интерфейсе – объявление операций, поддерживаемых экземплярами класса. К нему можно добавить объявления других классов, переменных, констант и исключительных ситуаций, уточняющих абстракцию, которую класс должен выражать. В свою очередь, реализация класса никому не интересна. По большей части реализация состоит в определении операций, объявленных в интерфейсе класса.

Интерфейс класса можно разделить на три части (**`public`**, **`protected`**, **`private`**).

Разные языки программирования предусматривают различные комбинации этих частей. В частности, в **`C++`** все три перечисленных уровня доступа определяются явно. Разработчик может задать права доступа к той или иной части класса, определив тем самым зону видимости клиента.

Состояние объекта задается в его классе через определения констант или переменных, помещаемых в его защищенной или закрытой части. Тем самым они инкапсулированы, и их изменения не влияют на клиентов.

Здесь возникает вопрос: *почему представление объекта определяется в интерфейсной части класса, а не в его реализации?*

Причины чисто практические – в противном случае понадобились бы объектно-ориентированные процессоры или очень хитроумные компиляторы.

Когда компилятор обрабатывает объявление объекта, например:

```
DisplayItem item1;
```

то он должен знать, сколько отвести под него памяти. Если бы эта информация содержалась в реализации класса, пришлось бы написать ее полностью, прежде, чем задействовать его клиентов. То есть, весь смысл отделения интерфейса от реализации был бы потерян.

В поведении простых классов можно разобраться, изучая операции их открытой части. Однако поведение более интересных классов, такое как перемещение объекта класса **DisplayItem** или составление расписания для экземпляра класса **TemperatureController,** включает взаимодействие разных операций, входящих в класс. Объекты таких классов действуют как маленькие машины, части которых взаимодействуют друг с другом, и так как все такие объекты имеют одно и то же поведение, можно использовать класс для описания их общей семантики, упорядоченной по времени и событиям. Динамику поведения таких объектов можно описать, используя модель конечного автомата.

4.2 Виды отношений между классами. Ассоциация и наследование

Классы, как и объекты, не существуют изолированно. В каждой проблемной области ключевые абстракции взаимодействуют многими интересными способами, что и должно быть отражено в модели.

Известны два основных типа отношений между классами. Во-первых, это отношение "обобщение/специализация" (общее и частное), известное как "**is-a**". Во-вторых, это отношение "целое/ часть", известное как "**part of**".

Языки программирования выработали несколько общих подходов к выражению отношений этих типов. В частности, большинство объектно-ориентированных языков непосредственно поддерживают разные комбинации следующих видов отношений:

- ассоциация;
- наследование;
- агрегация;
- использование;
- инстанцирование;
- метакласс.

Ассоциация является наиболее общим и неопределенным из шести перечисленных видов отношений. Обычно аналитик констатирует наличие ассоциации и, постепенно уточняя проект, превращает ее в какую-то более специализированную связь.

Пример. Желая автоматизировать розничную торговую точку, введем две абстракции – товары (**Product**) и продажи (**Sale**). Ассоциация между этими классами работает в обе стороны – задавшись товаром, можно выйти на сделку, в которой товар был продан, а пойдя от сделки, найти, что было продано.

В **C++** это можно выразить с помощью такого объявления классов:

```
class Product;
class Sale;
class Product {
public:
...
protected:
      Sale* lastSale;
};
class Sale {
public:
...
protected:
      Product** productSold;
};
```

Здесь применена ассоциация вида "один-ко-многим", для которой каждый экземпляр товара относится только к одной последней продаже, в то время как каждый экземпляр **Sale** может указывать на совокупность проданных товаров.

Как показывает этот пример, ***ассоциация – это семантическая (смысловая) связь***. По умолчанию, она не имеет направления, если не оговорено противное. Ассоциация, как в данном примере, подразумевает двухстороннюю связь и не объясняет, как классы общаются друг с другом. Можно только отметить семантическую зависимость, указав, какие роли классы играют друг для друга. Однако именно это и требуется на ранней стадии анализа.

Для определения отношения ассоциации необходимо зафиксировать участников, их роли и указать мощность отношения.

Мощность ассоциации в предыдущем примере обозначена как "один ко многим". Кроме этого на практике различают также мощности ассоциации "один-к-одному" и "многие-ко-многим".

Отношение "один-к-одному" обозначает очень узкую ассоциацию. Например, в розничной системе продаж примером могла бы быть связь между классом **`Sale`** и классом **`CreditCardTransaction`**, когда каждая продажа соответствует одному снятию денег с данной кредитной карточки. Примером отношения "многие-ко-многим" может служить ситуация, когда каждый объект класса **`Customer`** (покупатель) может инициировать транзакцию с несколькими объектами класса **`Saleperson`** (торговый агент), и каждый торговый агент может взаимодействовать с несколькими покупателями.

Наследование выражает отношение общего и частного. Однако одного наследования недостаточно, чтобы выразить все многообразие явлений и отношений жизни. Полезна также агрегация, отражающая отношения целого и части между экземплярами классов. Нелишне добавить отношение использования, означающее наличие связи между экземплярами классов. Имея дело с **`C++`**, не обойтись без инстанцирования, которое, подобно наследованию, является специфической разновидностью обобщения. "Метаклассовые" отношения – это нечто совершенно иное, в явном виде встречающееся только в языках **`Smalltalk`** и **`CLOS`**. Метакласс – это класс классов, что позволяет трактовать классы как объекты.

Наследование – это такое отношение между классами, когда один класс повторяет структуру и поведение другого класса (одиночное наследование) или других (множественное наследование) классов.

Класс, структура и поведение которого наследуются, называется суперклассом. Так, **TelemetryData**. является суперклассом для **ElectricalData**.

Пример. Находящиеся в полете космические зонды посылают на наземные станции информацию о состоянии своих основных систем, например, источников энергоснабжения и двигателей, а также результаты измерения с датчиков уровня радиации, масс-спектрометров, телекамер, фиксаторов столкновений с микрометеоритами и т.д. Вся совокупность передаваемой информации называется телеметрическими данными. Как правило, они передаются в виде потока данных, состоящего из заголовка, который включает временные метки и ключи для идентификации последующих данных (нескольких пакетов от подсистем и датчиков). Все это выглядит как простой набор разнотипных данных, поэтому для описания каждого типа данных телеметрии очевидна следующая структура программы:

```
class Time...

struct ElectricalData {

      Time timeStamp;
      int id;
      float fuelCell1Voltage, fuelCell2Voltage;
      float fuelCell1Amperes, fuelCell2Amperes;
      float currentPower;

};
```

Однако такое описание имеет ряд недостатков:

Структура класса **ElectricalData** не защищена, то есть клиент может вызвать изменение такой важной информации, как **timeStamp** или

currentPower (мощность, развиваемая обеими электробатареями, которую можно вычислить из тока и напряжения).

Структура является полностью открытой, ее модификация влияет на клиентов, так как возможно добавление новых элементов в структуру или изменение типа существующих элементов. Как минимум, приходится заново компилировать все описания, связанные каким-либо образом с этой структурой. Еще важнее, что внесение в структуру изменений может нарушить логику отношений с клиентами, следовательно, логику всей программы.

Описание структуры очень трудно для восприятия. По отношению к такой структуре можно выполнить множество различных действий (пересылка данных, вычисление контрольной суммы для определения ошибок и т.д.), но все они не будут связаны с приведенной структурой логически.

Наконец, при наличии нескольких сотен разновидностей телеметрических данных становится очевидным, что описание большого количества структур будет избыточным как из-за повторяемости структур, так и из-за наличия общих функций обработки.

Значительно лучше построить иерархию классов, в которой от общих классов с помощью наследования образуются более специализированные; например, следующим образом:

```
class TelemetryData {
public:
       TelemetryData();
       virtual ~TelemetryData();
       virtual void transmit();
       Time currentTime() const;
protected:
       int id;
       Time timeStamp;
};
```

В этом примере введен класс, имеющий конструктор, деструктор, который наследники могут переопределить, и функции **transmit** и **currentTime**, видимые для всех клиентов. Защищенные элементы **id** и **timeStamp** несколько

лучше инкапсулированы – они доступны только классу и его подклассам. Функция **currentTlrne** сделана открытой, благодаря чему значение **timeStamp** можно читать (но не изменять).

Теперь следует переопределить поведение **ElectricalData**:

```
class ElectricalData : public TelemetryData {

public:

     ElectricalData(float v1, float v2, float a1, float a2);
     virtual ~ElectricalData();
     virtual void.transmit();
     float currentPower() const;

protected:

     float fuelCell1Voltage, fuelCell2Voltage;
     float fuelCell1Amperes, fuelCell2Amperes;
};
```

Класс **ElectricalData** обозначен как наследник класса **TelemetryData**, но исходная структура дополнена четырьмя новыми элементами, а поведение переопределено (изменена функция **transmit**). Кроме того, добавлена функция **currentPower**.

Наследование подразумевает, что подклассы повторяют структуры их суперклассов. В предыдущем примере экземпляры класса **ElectricalData** содержат элементы структуры суперкласса (**id** и **timeStamp**) и более специализированные элементы (**fuelCell1Voltage**, **fuelCell2Voltage**, **fuelCell1Amperes**, **fuelCell2Amperes**).

Производный от суперкласса класс называется подклассом. Это означает, что наследование устанавливает между классами иерархию общего и частного. В этом смысле **ElectricalData** является более специализированным классом более общего **TelemetryData**. В подклассе структура и поведение исходного суперкласса дополняются и переопределяются. Наличие механизма наследования отличает объектно-ориентированные языки от объектных.

4.3 Отношения агрегации, использования и инстанцирования

Агрегация – это *непосредственное* (физическое) включение одного класса в другой (время жизни их неразрывно связано) или *косвенное* включение (классы создаются и уничтожаются независимо).

Рассмотрим снова класс **TemperatureController**:

```
class TemperatureController {
Public:
    TemperatureController (Location);
    ~TemratureController();
    void process(const TemperatureRamp&);
    Minute schedule(const TemperatureRamp&) const;
private:
    Heater h;
};
```

В приведенном примере класс **TemperatureController** это, несомненно, целое, а экземпляр класса **Heater** – одна из его частей. Здесь мы имеем агрегацию по значению, эта разновидность физического включения означает, что объект класса **Heater** не существует отдельно от объемлющего экземпляра класса **TemperatureController**.

Менее обязывающим является включение по ссылке. Можно изменить закрытую часть **TemperatureController** таким образом:

```
Heater* h;
```

В этом случае класс **TemperatureController** по-прежнему означает целое, но его часть, экземпляр класса **Heater**, содержится в целом косвенно. Теперь эти объекты живут отдельно друг от друга – их можно создавать и уничтожать независимо.

Агрегация является направленной, как и всякое отношение "целое/часть". Объект **Heater** входит в объект **TemperatureController**, а не наоборот. Физическое вхождение одного в другое нельзя "зациклить", а вот указатели

можно. При этом каждый из двух объектов может содержать указатель на другой объект.

Отношение ***использования*** между классами соответствует равноправной связи между их экземплярами. Это то, во что превращается ассоциация, если оказывается, что одна из ее сторон (клиент) пользуется услугами другой стороны (сервера).

В примере, приведенном выше, связь объектов **rampController** и **growingRamp** иллюстрируются через отношения использования между их классами **TemperatureController** и **TemperatureRamp**. Класс **TemperatureRamp** упомянут как часть сигнатуры функции-члена **process**. Это позволяет утверждать, что класс **TemperatureController** пользуется услугами класса **TemperatureRamp**.

Инстанцирование. Параметризованный класс не может иметь экземпляров, пока он не будет инстанцирован. Объявим две конкретных очереди – очередь целых чисел и очередь экранных объектов:

```
Queue<int> intQueue;
Queue<DisplayItem*> itemQueue;
```

Объекты **intQueue** и **itemQueue** – это экземпляры совершенно различных классов, которые даже не имеют общего суперкласса. Тем не менее, они получены из одного параметризованного класса **Queue**.

Это инстанцирование безопасно с точки зрения типов. По правилам **C++** будет отвергнута любая попытка поместить в очередь или извлечь из нее что-либо, кроме целых чисел и разновидностей **DisplayItem**, соответственно.

Метакласс – это класс, экземпляры которого суть классы. Метаклассы венчают объектную модель в чисто объектно-ориентированных языках.

Хотя в **C++** метаклассов нет, семантика его конструкторов и деструкторов обслуживать метаклассы. **C++** имеет средства поддержки и переменных класса, и операций метакласса.

4.4 Назначение операций класса

Описание интерфейса класса или модуля представляет собой сложную задачу. Обычно, исходя из структурного смысла класса, делается первое приближение, а затем, когда появляются клиенты класса, интерфейс уточняется, модифицируется и дополняется.

В пределах каждого класса принято иметь только примитивные операции (методы), отражающие отдельные аспекты поведения. Для этого желательно применять небольшое количество методов, что облегчает образование подклассов с переопределением поведения. Решение о количестве методов может быть обусловлено тем, что описание поведения в одном методе упрощает интерфейс, но усложняет и увеличивает размеры самого метода, а расщепление метода усложняет интерфейс.

В объектно-ориентированном моделировании и проектировании принято рассматривать методы класса как единое целое, поскольку все они взаимодействуют друг с другом для реализации протокола абстракции. Таким образом, определив поведение, нужно решить, в каком из классов это поведение реализуется. Для принятия такого решения можно использовать следующие критерии:

Повторная используемость	Будет ли это поведение полезно более чем в одном контексте?
Сложность	Насколько трудно реализовать такое поведение?
Применимость	Насколько данное поведение характерно для класса, в который мы хотим включить поведение?
Знание реализации	Надо ли для реализации данного поведения знать секреты класса?

Обычно операции объявляются как ***методы класса***, к объектам которого относятся данные действия. Однако в языке **C++** допускается описание операций

в виде свободных подпрограмм (утилит класса). Свободная подпрограмма в терминологии **C++** – это функция, не являющаяся элементом класса. Свободные подпрограммы не могут переопределяться подобно обычным методам, в них нет такой общности. Наличие утилит позволяет выполнить требование примитивности и уменьшить зацепление между классами, особенно если эти операции высокого уровня задействуют объекты многих различных классов.

Аспекты расхода памяти и времени. После принятия решения о необходимости конкретной функции и определения её семантики, следует принять решение об использовании времени и памяти этой функцией. Поскольку один объект посылает другому сообщение, эти два объекта должны быть каким-то образом синхронизированы. В случае многих потоков управления это означает, что передача сообщений сложнее, чем управление вызовами подпрограмм – для большинства языков программирования синхронизация просто не нужна, поскольку в них программы однопотоковые, и все объекты действуют последовательно. Однако в языках, поддерживающих параллелизм, нужно применить более изощренные системы передачи сообщений, чтобы избежать случаев, когда два потока работают одновременно и несогласованно с одним и тем же объектом. Объекты, семантика которых сохраняется при многопоточности, являются или синхронизированными, или защищенными.

В некоторых обстоятельствах следует отмечать параллельность как для отдельных операций, так и для объекта в целом, так как разные операции могут потребовать разных форм синхронизации. Выделяют следующие формы передачи сообщений:

- ***Синхронная*** – операция активизируется только при готовности передающего и принимающего сообщения объектов, ожидание взаимной готовности может быть неопределенно долгим.
- ***С учетом задержки*** – то же, что и синхронная, однако, в случае, если принимающий не готов, передающий не выполняет операцию.

- ***С ограничением времени*** – то же, что и синхронная, однако, посылающий будет ждать готовности принимающего не дольше некоторого времени.
- ***Асинхронная*** – операция выполняется вне зависимости от готовности принимающего.

Нужная форма выбирается для каждой операции отдельно, но только после того, как ее функциональная семантика определена.

Выводы.

1. В проектировании классов рекомендуется идея контрактного программирования. Класс – это генеральный контракт между абстракцией и всеми ее клиентами.
2. Между классами используются два основных типа отношений. Во-первых, это отношение "обобщение/специализация" (общее и частное), известное как "**`is-a`**". Во-вторых, это отношение "целое/часть", известное как "**`part of`**".
3. Для определения отношения ассоциации необходимо зафиксировать участников, их роли и указать мощность отношения.
4. В подклассе структура и поведение исходного суперкласса дополняются и переопределяются. Наличие механизма наследования отличает объектно-ориентированные языки от объектных.
5. Агрегация является направленной, как и всякое отношение "целое/часть". Физическое вхождение одного в другое нельзя "зациклить", а указатели можно.
6. Параметризованный класс не может иметь экземпляров, пока он не будет инстанцирован.

5 БАЗИС ЯЗЫКА МОДЕЛИРОВАНИЯ UML (*UNIFIED MODELING LANGAUGE*)

5.1 Направления использования UML

Язык UML, как объектно-ориентированный язык, был специально разработан для среды, в которой объекты распределены, параллельны и связаны.

Объекты распределены – это значит, что каждый объект поддерживает свое собственное состояние, отличное от других.

Объекты параллельны – это значит, что каждый из объектов потенциально может действовать параллельно с другими.

Объекты связаны – это значит, что каждый из объектов может отправлять сообщения другим объектам через сеть соединений.

UML относится к языкам визуального моделирования. В рамках языка UML все представления о модели фиксируются в виде графических конструкций, получивших название диаграмм. Язык UML предназначен для написания моделей анализа, проектирования и реализации ***объектно-ориентированных программных систем.***

UML – это стандартный инструмент для создания “чертежей” программного обеспечения. UML состоит из словаря и правил, позволяющих комбинировать эти слова и получать осмысленные конструкции. UML позволяет специфицировать решения, то есть построить точные, недвусмысленные модели, которые могут быть переведены языками программирования в программный код. UML – объектно-ориентированный язык, но в то же время он никак не связан с конкретными объектно-ориентированными языками программирования. Разработанный в терминах языка UML проект легко реализовать на любом существующем языке, поддерживающим объектно-ориентированную технологию. В этом смысле он выступает как универсальный язык моделирования. Он создавался с учетом способности адаптироваться к возникающим новым проблемам.

Первая разработка языка UML выполнена в компании **Rational** в 1995 году. За это время было предпринято несколько инициатив по расширению языка UML для новых прикладных областей, включая системы и базы данных Web. В 1999 году была закончена версия UML 1.3, предусматривавшая восемь видов диаграмм.

В 2003 году на Парижской сессии OMG был принят новый стандарт языка UML версии 2.0, который по сравнению с прежними версиями языка UML 1.3/1.4 имеет следующие особенности:

- расширены прикладные области анализа, встроена поддержка архитектуры времени выполнения задачи, *добавлено пять новых диаграмм*;
- уточнено представление *отношений* между объектами и классами, в результате чего упростилось моделирование конечных автоматов;
- *упрощены синтаксис и семантика языка*.

В новой версии UML 2.0 все диаграммы разделены на три категории:

1. ***Диаграммы поведения***. К этой категории относят диаграммы, которые отображают особенности поведения систем или бизнес-процессов. Эта категория включает в себя диаграммы деятельности (activity diagrams), диаграммы автоматов (state machine diagrams), диаграммы Use Case (use case diagrams).
2. ***Диаграммы взаимодействия***. К этой категории относят подмножество диаграмм поведения, которые выделяют взаимодействия объектов. Категория включает диаграммы коммуникации (communication diagrams), диаграммы обзора взаимодействия (interaction overview diagrams), диаграммы последовательности (sequence diagrams) и диаграммы синхронизации (timing diagrams).
3. ***Диаграммы структуры***. К этой категории относят диаграммы, отображающие элементы спецификации, которые не зависят от

времени. Категория включает диаграммы классов (class diagrams), диаграммы составной структуры (composite structure diagrams), компонентные диаграммы (component diagrams), диаграммы размещения (deployment diagrams), диаграммы объектов (object diagrams) и диаграммы пакетов (package diagrams).

В UML 2.0 даны следующие определения диаграмм.

Диаграмма деятельности изображает поведение, используя модель потока данных и управления.

Диаграмма классов показывает коллекцию декларативных (статических) элементов модели, таких как классы и типы, а также их содержание и отношения.

Диаграмма коммуникации отображает взаимодействия между линиями жизни (отдельными участниками), выделяя архитектуру внутренней структуры и ее соответствие передачам сообщений. Последовательность сообщений указывается путем их нумерации.

Компонентная диаграмма показывает организацию компонентов и зависимости между компонентами.

Диаграмма составной структуры изображает внутреннюю структуру классификатора (класса, компонента, элемента Use Case).

Диаграмма размещения изображает исполняемую архитектуру системы. Она представляет артефакты системы как узлы, соединяемые коммуникациями. Обычно узлы представляют или аппаратные средства, или исполняемые программы.

Диаграмма обзора взаимодействия – это вариант диаграммы деятельности, которая изображает управляемый поток взаимодействий, например, передача сообщений в сетевых структурах.

Объектная диаграмма отображает объекты и их отношения в конкретный момент времени. Она может рассматриваться как специальный вариант диаграммы классов или диаграммы коммуникации.

Диаграмма пакетов показывает, как элементы модели размещены в пакетах, какие зависимости существуют между пакетами.

Диаграмма последовательности отображает последовательность событий на линиях жизни объектов.

Диаграмма автомата отображает дискретное поведение, моделируемое через переходы в системах с конечным числом состояний.

Диаграмма синхронизации показывает изменение состояния или условия линии жизни во времени в ответ на внешние события.

Диаграмма Use Case показывает отношения между актерами и системой, а также варианты использования системы.

Язык UML позволяет создавать, тестировать и поддерживать в работе программные системы. С его помощью можно моделировать систему целиком, от концепции до исполняемого артефакта (элемента информации, используемого или порождаемого в процессе разработки программного обеспечения), решить проблему масштабируемости (наращивание, объединение программ или, наоборот, создание программ локального уровня), которая присуща сложным системам. Кроме того, язык UML используется для автоматической разработки программ с помощью специально созданных для этого ***CASE-инструментов***, прежде всего, семейства объектно-ориентированных **CASE**-средств – **Rational Rose**.

5.2 Особенности объектно-ориентированного подхода в UML

UML реализует объектно-ориентированный подход к разработке сложных систем следующим образом:

- Программная система представляется в виде множества самостоятельных ***сущностей***, взаимодействующих друг с другом. Каждая сущность отвечает за хранение информации, необходимой для её функционирования, и, кроме того, реализует своё собственное

поведение. ***С каждой сущностью связано понятие класса и объекта***.

- Каждый ***объект*** защищен системой правил, которые не позволяют окружающим объектам произвольно менять его данные или влиять на его поведение. Правила определяют способ взаимодействия с окружением, то есть интерфейс, и скрывают детали реализации. Иными словами – ***данные и методы инкапсулированы в объекте***.
- Под ***поведением*** объекта в UML понимаются любые правила взаимодействия объекта с внешним миром и с данными самого объекта.
- Процесс разделения сущностей на классы и построение общей классификации осуществляется с помощью механизма ***наследования и полиморфизма***.

Полиморфизм касается переопределения поведения объектов. В UML для описания полиморфизма вводятся понятия ***операции и метода***. Следует учесть, что с ***операцией*** связано качественное описание поведения объекта, а с ***методом*** – его конкретная реализация. Таким образом, при наследовании операции возможна передача свойств, присущих объектам класса-предка.

- ***При иерархическом представлении*** больших систем классы можно объединять в пакеты (группы) и использовать модульный подход в проектировании.
- При моделировании все изучаемые сущности рассматриваются с ***двух точек зрения***. *Первая* – это поиск группы близких по своим свойствам элементов и создание общего для всех них описания. *Вторая* – это указание механизма, позволяющего из абстрактного элемента получить конкретный, с присущими ему индивидуальными чертами, который затем будет использован для конструирования системы. В результате ***сущности*** представляются парами “тип-экземпляр”. Возможно образование нескольких пар, например, “класс-объект”,

“ассоциация-связь”, “параметр-значение”, “операция-вызов процедуры”. Для изображения элементов этих пар конкретный экземпляр и его обобщение на диаграммах представляются геометрически одинаково, так что можно разработать, например, и диаграммы классов и диаграммы объектов.

5.3 Базис UML – структурные предметы

Базовые элементы UML составляют ***язык моделирования***. Этот язык описывает как ***статические***, так и ***динамические*** состояния сложной модели. Знание этого языка позволяет понимать модель сложной системы так же, как понимаются блок-схемы алгоритмов, чертежи, структурные схемы математических моделей, электрические или гидравлические схемы.

Объектно-ориентированными базовыми блоками, которые используются в UML для написания моделей, являются четыре группы предметов:

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

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

1. ***Класс (class)*** – описание множества объектов, которые разделяют одинаковые свойства, операции, отношения и семантику (смысл). Класс реализует один или несколько интерфейсов. Графически класс отображается в виде прямоугольника, включающего секции, с именем, атрибутами (свойствами) и операциями (рис. 5.1).

Рисунок 5.1 –Изображение класса

2. ***Интерфейс*** ***(interface)*** – это набор операций, которые определяют услуги класса или компонента. Интерфейс описывает поведение элемента, видимое извне. Следует учесть, что интерфейс определяет *набор спецификаций операций* (сигнатуру), а не набор реализаций операций. Графически интерфейс изображается в виде кружка с именем (рис. 5.2). Имя начинается с буквы “I”.

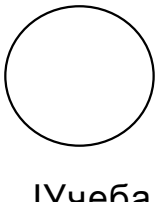

Рисунок 5.2 –Изображение интерфейса

3. ***Кооперация*** ***(collaboration)*** – (сотрудничество) определяет взаимодействие и является совокупностью ролей и элементов, которые совместно обеспечивают более сложное коллективное поведение. Таким образом, кооперации имеют как структурные, так и поведенческие измерения. Конкретный класс может участвовать в нескольких кооперациях. Эти кооперации представляют собой реализацию ***паттернов*** (образцов), которые формируют систему. Графически кооперация изображается как пунктирный эллипс, в который вписывается имя (рис. 5.3).

Рисунок 5.3 –Изображение кооперации

4. ***Актер (Actor)*** – это набор согласованных ролей, которые могут играть пользователи при взаимодействии с системой, точнее с её элементами (Use Case). Каждая роль требует от системы определенного поведения. Актер изображается как проволочный человечек и снабжается именем (рис. 5.4).

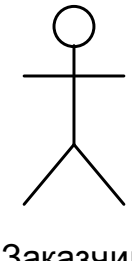

Рисунок 5.4 – Изображение актера

5. ***Элемент Use Case*** – это прецедент, последовательность действий, которая выполняется системой в интересах актера. В модели элемент Use Case применяется для структурирования предметов поведения. Элемент Use Case изображается в виде эллипса, в который вписывается имя (рис. 5.5).

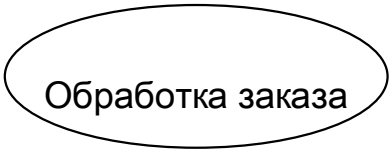

Рисунок 5.5 – Изображение элемента Use Case

6. ***Активный класс (active class)*** – класс, объекты которого имеют один или несколько процессов и поэтому могут инициировать управляющую деятельность. В отличие от обычного класса объекты активного класса действуют одновременно с объектами других классов. Графически активный класс изображается более толстыми линиями, чем обычный класс (рис. 5.6).

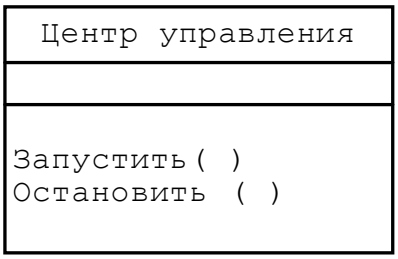

Рисунок 5.6 – Изображение активного класса

7. ***Компонент (component)*** – это физическая и заменяемая часть системы, которая соответствует набору интерфейсов и обеспечивает его реализацию. Компоненты – это файлы исходного кода, а также различные разновидности используемых файлов, например, **COM** (компонентная объектная модель). Обычно компонент – это физическая упаковка различных логических элементов (классов, интерфейсов, кооперации). Компонент изображается как прямоугольник с вкладками и с именем (рис. 5.7).

Рисунок 5.7 – Изображение компонента

8. ***Узел (node)*** – физический элемент, который существует в период работы системы и представляет собой ресурс, имеющий память и возможности обработки информации. В узле размещается набор компонентов, который может перемещаться от узла к узлу. Узел изображается как куб с именем (рис. 5.8).

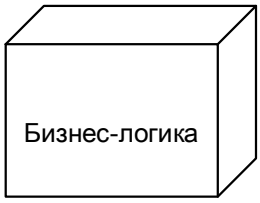

Рисунок 5.8 – Изображение узла

5.4 Предметы поведения, группирующие и поясняющие предметы

Предметы поведения – динамические части **UML**-моделей. Они являются ***глаголами*** моделей, представлением поведения во времени и в пространстве. Семантически эти элементы ассоциируются со структурными элементами.

Существуют две основные разновидности предметов поведения.

1. ***Взаимодействие (interaction)*** – 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). Сообщение (`message`) изображается в виде направленной линии с именем ее операции (рис. 5.9).

Рисунок 5.9 – Изображение сообщения

2. ***Конечный автомат (state machine)*** – поведение, которое определяет последовательность состояний объекта или взаимодействия, выполняемые в ходе его существования в ответ на события. С помощью конечного автомата может определяться поведение индивидуального класса или кооперации классов. Элементами конечного автомата являются состояния, переходы из одного состояния в другое, события (предметы, вызывающие переходы) и действия (реакции на переход). Состояние изображается как закругленный прямоугольник, включающий его имя и его подсостояния, если они есть (рис. 5.10).

Рисунок 2.10 – Изображение состояния конечного автомата

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

Пакет (package) – общий механизм для распределения элементов по группам. В пакет могут помещаться структурные предметы, предметы поведения и даже другие пакеты (аналогично файловой системе). В отличие от компонента, который существует в период выполнения, пакет – чисто концептуальное понятие, он существует только в период разработки.

Пакет изображается как папка с закладкой, на папке обозначено имя и, если необходимо, содержание (рис. 5.11).

Поясняющие предметы – части UML-моделей, которые применяются для замечаний, описания, объяснения и комментирования *любого* элемента модели. Предусмотрена одна разновидность поясняющего предмета – примечание.

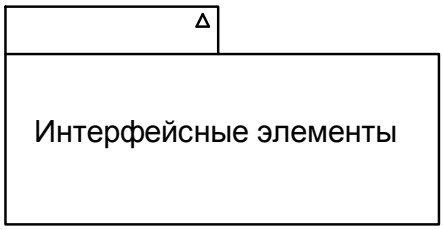

Рисунок 5.11 – Изображение пакета

Примечание (note) – это символ для отображения ограничений и замечаний, присоединяемых к элементу или к совокупности элементов. Примечание изображается в виде прямоугольника с загнутым углом, в который вписывается текстовый или графический материал (рис. 5.12).

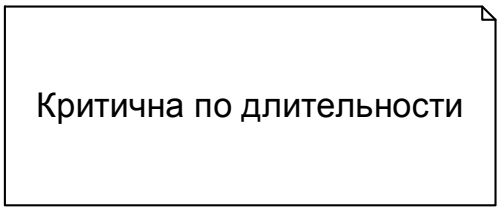

Рисунок 5.12 – Изображение примечания

5.5 Базовые блоки отношений

В **UML** имеется четыре разновидности отношений – *зависимости, ассоциации, обобщения и реализации*. Эти отношения являются базовыми строительными блоками и используются при написании моделей.

Зависимость – семантическое отношение между двумя предметами, в котором изменение в одном предмете (независимом) может влиять на семантику другого предмета (зависимого). Зависимость изображается в виде пунктирной линии, направленной на независимый предмет и иногда имеющей метку (рис. 5.13).

Рисунок 5.13 – Изображение зависимости

Ассоциация – это структурное отношение, которое описывает набор связей, являющихся соединением между объектами. При этом структурные отношения между целым и частями представляет собой специальную разновидность ассоциации – агрегацию. Ассоциации изображаются в виде сплошной линии, возможно направленной, иногда имеющей метку, часто включающей другие ”украшения”, например, имена ролей или мощность (рис. 5.14).

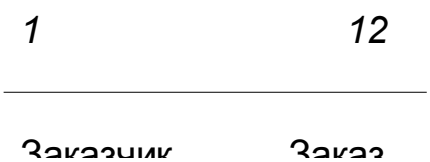

Рисунок 5.14 – Изображение ассоциации

Обобщение – это отношение специализации/обобщения, в котором объекты специализированного элемента (потомка, ребенка) могут заменять объекты обобщенного элемента (предка, родителя). Другими словами – потомок разделяет структуру и поведение родителя. Обобщения изображаются в виде сплошной стрелки с полым наконечником, указывающим на родителя (рис. 5.15).

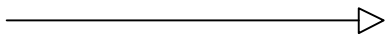

Рисунок 5.15 – Изображение обобщения

Реализация – семантическое отношение между классификаторами, где один классификатор определяет контракт, который другой классификатор обязуется выполнить. К классификаторам относят классы, интерфейсы, компоненты, элементы Use Case, кооперации. Отношения реализации применяют в двух случаях:

- между интерфейсами и классами или компонентами, которые эти отношения реализуют;
- между элементами Use Case и кооперациями, которые их реализуют.

Реализации изображаются как нечто среднее между зависимостью и обобщением – пунктирной стрелкой с полым наконечником (рис. 5.16).

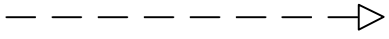

Рисунок 5.16 – Изображение реализации

5.6 Механизмы расширения в UML

Несмотря на большие возможности языка UML, он не в состоянии отразить все нюансы, возникающие при создании моделей. Поэтому UML создавался как открытый язык, допускающий расширения по усмотрению пользователя.

Механизмами расширения являются:

- ограничения;
- теговые величины;
- стереотипы.

Ограничение (constraint) расширяет семантику строительного UML-блока, позволяя добавить *новые правила* или модифицировать существующие. Ограничение показывается в виде текста, заключенного в фигурные скобки, например, {**ВремяПерезапуска = 2мс**}. Пример ограничения на *сумму* и на владельца счета (либо персона, либо организация, но не оба) показан на рисунке 5.17.

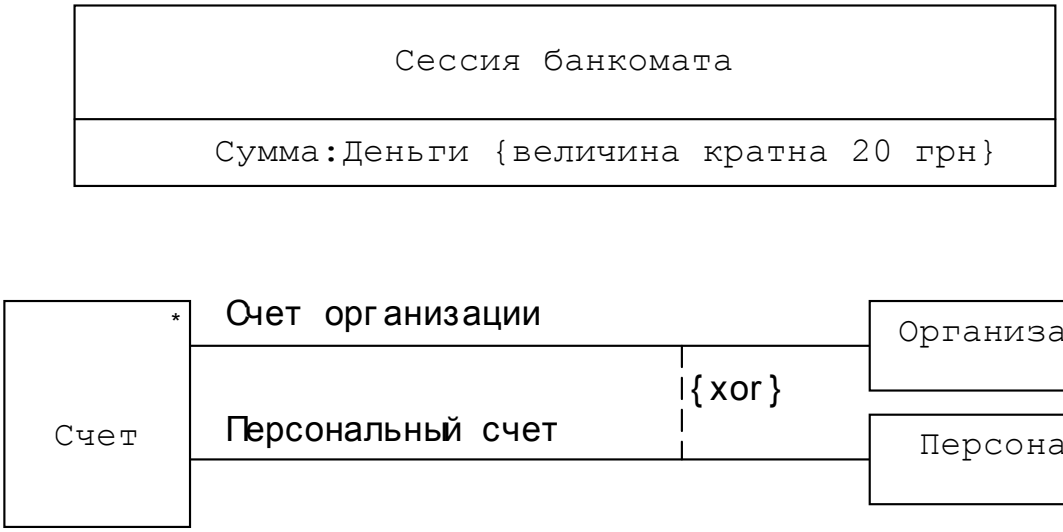

Рисунок 5.17 – Ограничения

Теговая величина (tagged value) расширяет характеристики строительного UML-блока, позволяя создать новую информацию в спецификации элемента. Теговую величину показывают как строку в фигурных скобках, например, {**версия = 4.2, автор = Белов**}.

Стереотип (stereotype) расширяет словарь языка, позволяет создавать новые виды строительных UML-блоков с учетом специфики 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, другое визуальное представление. Он будет иначе обрабатываться при генерации программного кода. Стереотип отображают как имя, указываемое в *двойных угловых* скобках или кавычках. Примеры элементов со стереотипами показаны на рисунке 5.18.

Стереотип **<<metaclass>>** задает возможности метакласса обычному элементу модели, стереотип **<<exception>>** говорит о том, что класс **ПотеряЗначимости** теперь рассматривается как специальный класс, которому, например, разрешается только генерация и обработка сигналов исключений, а стереотип **<<call>>** устанавливает новые отношения (исходная операция вызывает целевую операцию).

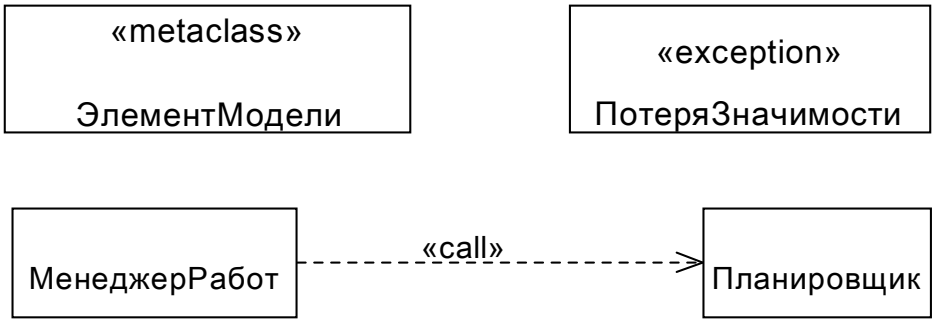

Рисунок 5.18 – Примеры стереотипов

Таким образом, механизмы расширения позволяют адаптировать **UML** под нужды конкретных проектов и под новые программные технологии.

5.7 Порядок разработки диаграмм при моделировании

Порядок разработки UML диаграмм при проектировании системы упрощенно можно представить следующим образом.

На первом этапе производится оценка внешней функциональности системы, выделяются все актеры и все прецеденты (Use case). Отношения между ними изображаются на серии диаграмм использования (Use case diagrams), которые содержат список выполняемых системой операций. Одним из основных преимуществ применения диаграммы вариантов использования является то, что она показывает какие функциональные возможности будут заложены в систему. Рассматривая действующих лиц,

пользователи системы выяснят, кто будет с ней взаимодействовать, определят сферу применения системы и что она будет делать. Это поможет им узнать также, что она не будет делать, и внести необходимые коррективы.

Дальнейшая работа над проектом осуществляется в рамках детализации представленных на диаграмме прецедентов. Для каждого прецедента строится описание его динамики в виде серии диаграмм последовательности (sequence diagrams) и диаграмм деятельности (activity diagrams). С помощью этих диаграмм проектировщики и разработчики системы могут определить классы, которые нужно создать, связь между ними, а также операции и обязательства (responsibilities) каждого класса. Диаграммы взаимодействия – это краеугольный камень, на котором возводится проект.

Выделенные объекты позволяют разработать диаграммы классов (class diagrams), которые определяют статическую структуру и описывают взаимоотношения объектов друг с другом.

Для описания сложной динамики поведения классов при их реагировании на события возникает потребность в диаграмме автоматов (state machine diagram).

Размещение объектов по программным модулям описывается в компонентных диаграммах (component diagrams), а распределение программных модулей по сети и компьютерам – в диаграммах распределения (deployment diagrams).

Итак, диаграммы следует разрабатывать в следующей последовательности: use case diagrams, sequence diagrams, class diagrams, state machine diagrams, component diagrams, deployment diagrams. Этот набор диаграмм должен присутствовать в моделях системы всегда, другие типы диаграмм необходимы для большей детализации системы.

6 РАЗРАБОТКА ДИАГРАММЫ ВАРИАНТОВ ИСПОЛЬЗОВАНИЯ

6.1 Основные элементы и отношения в диаграммах Use Case

Визуальное моделирование в UML можно представить как некоторый процесс поуровневого спуска от наиболее обшей и абстрактной концептуальной модели исходной системы к логической, а затем и к физической модели соответствующей программной системы. Для достижения этих целей вначале строится модель в форме так называемой диаграммы вариантов использования (**use case diagram**), которая описывает функциональное назначение системы или, другими словами, то, что система будет делать в процессе своего функционирования. Диаграмма вариантов использования является исходным концептуальным представлением или концептуальной моделью системы в процессе ее проектирования и разработки.

Разработка диаграммы вариантов использования преследует цели:

- Определить общие границы и контекст моделируемой предметной области на начальных этапах проектирования системы.
- Сформулировать общие требования к функциональному поведению проектируемой системы.
- Разработать исходную концептуальную модель системы для ее последующей детализации в форме логических и физических моделей.
- Подготовить исходную документацию для взаимодействия разработчиков системы с ее заказчиками и пользователями.

Суть данной диаграммы состоит в следующем. Проектируемая система представляется в виде множества сущностей или актеров, взаимодействующих с системой с помощью так называемых вариантов использования. При этом актером (actor) или действующим лицом называется любая сущность, взаимодействующая с системой *извне*. Это может быть человек, техническое устройство, программа или любая другая система, которая может служить

источником воздействия на моделируемую систему так, как определит сам разработчик. В свою очередь, вариант использования (use case) служит для описания сервисов, которые система предоставляет актеру. Другими словами, каждый вариант использования определяет некоторый набор действий, совершаемый системой при диалоге с актером. При этом ничего не говорится о том, каким образом будет реализовано взаимодействие актеров с системой.

Одно действующее лицо (actor) обычно использует часть функций системы. Поэтому все действующие лица можно условно разделить на группы и для каждой группы составить свой сценарий (use case). Список всех сценариев определяет функциональные требования к системе, на основе которых формулируется техническое задание.

В самом общем случае, диаграмма вариантов использования представляет собой граф специального вида, который является графической нотацией для представления конкретных вариантов использования, актеров, возможно некоторых интерфейсов, и отношений между этими элементами. При этом отдельные компоненты диаграммы могут быть заключены в прямоугольник, который обозначает проектируемую систему в целом. Следует отметить, что отношениями данного графа могут быть только некоторые фиксированные типы взаимосвязей между актерами и вариантами использования, которые в совокупности описывают сервисы или функциональные требования к моделируемой системе.

Вершинами в диаграмме использования являются актеры и элементы Use Case. Актеры представляют внешний мир, нуждающийся в работе системы. Элементы Use Case представляют действия, выполняемые системой в интересах актеров.

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 утверждение – актером могут быть разные пользователи.

Например, для коммерческого использования самолета можно выделить двух актеров – пилота и кассира, даже если это одно лицо (рис. 6.1).

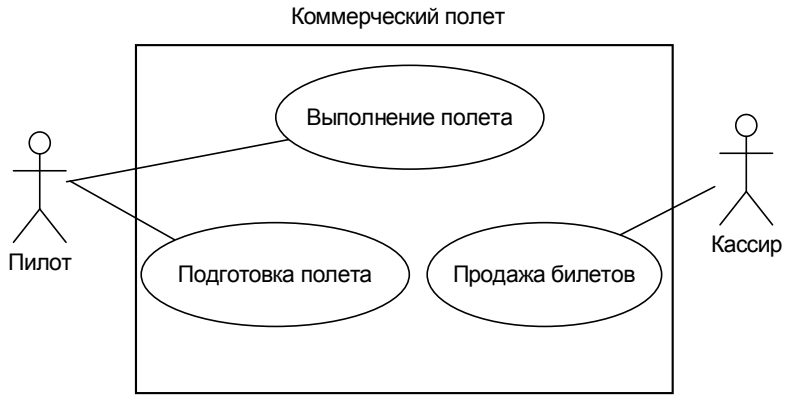

Рисунок 6.1 – Модель `Use Case` для коммерческого полета

Элемент `Use Case` – это описание последовательности действий, которые выполняются системой и производят для актера видимый результат. Один актер может использовать несколько элементов `Use Case`, и наоборот, каждый элемент может быть использован несколькими актерами.

Между актером и элементом `Use Case` может быть ***только один вид отношения*** – ***ассоциация***, отображающая их взаимодействие. Ассоциация может быть помечена именем, ролями, мощностью.

Следует учесть, что отношениями данного графа могут быть только некоторые фиксированные типы взаимосвязей между актерами и вариантами использования, которые в совокупности описывают сервисы или функциональные требования к моделируемой системе.

Между актерами допустимы *отношения обобщения*, то есть экземпляр потомка может взаимодействовать с теми же элементами `Use Case`, что и экземпляр родителя. Например, группа студентов является обобщением отдельных студентов этой группы (рис. 6.2).

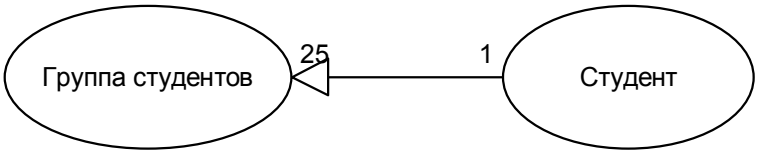

Рисунок 6.2 – Отношение обобщения между элементами

Между элементами `Use Case` *также допустимы обобщения*, которые фиксируют наследование поведения родителя.

Кроме обобщений между этими элементами определены два отношения зависимости – *включения и расширения*.

Отношение включения (include) между двумя элементами соответствует делегации обязанностей. Например, в базовый элемент **`Снять деньги`** может быть включен элемент **`Идентификация клиента`** и базовый элемент может быть расширен для определенных условий, например, запрос **`Вид валюты`** (рис. 6.3).

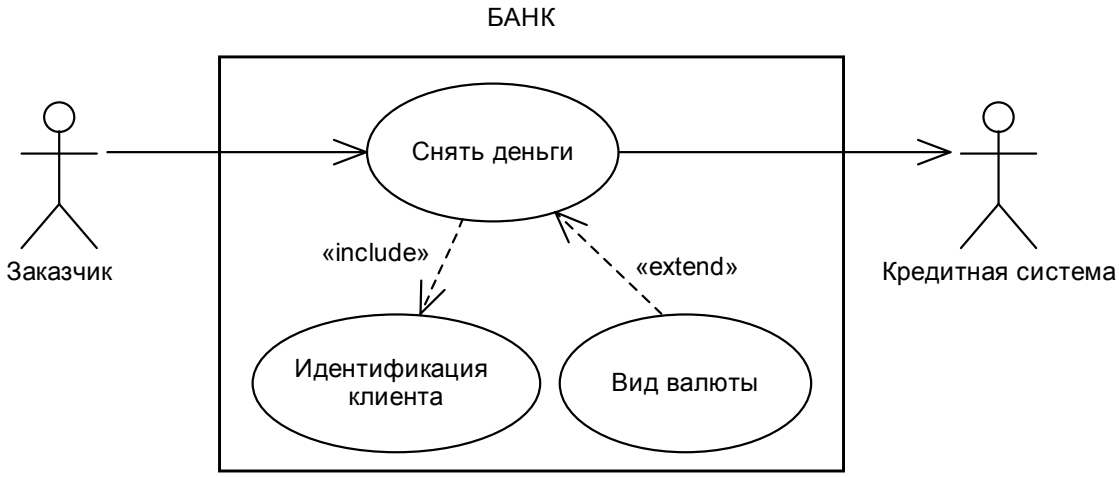

Рисунок 6.3 – Простейшая диаграмма с включением и расширением

Отношение расширения (extend) между элементами Use Case означает, что базовый элемент включает поведение другого элемента в точке, которая определяется косвенно расширяющим элементом Use Case. Это отношение применяется, если нужно отделить обязательное поведение от необязательного. Например, при заказе продукта элементом расширения будет

запрос каталога или другой информации. На рисунке 6.4 приведен пример диаграммы Use Case для обслуживания заказчика продукта с выделением отдельного потока, вставка которого в определенную точку управляется актером.

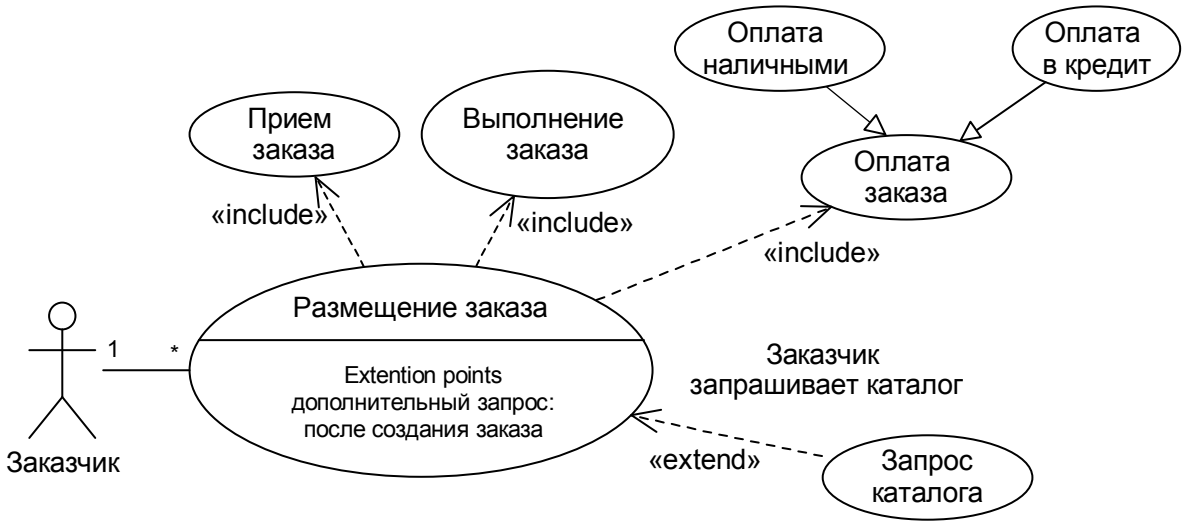

Рисунок 6.4 – Диаграмма Use Case для обслуживания заказчика

Как показано на рисунке 6.4, внутри элемента Use Case может быть дополнительная секция **Extention points**, в которой перечисляются точки расширения (**Запрос каталога**) и для справки указывается, что точка расширения размещена после действий, обеспечивающих создание заказа. На этом же рисунке отображены отношения наследования – элементы **Оплата наличными** и **Оплата в кредит** наследуют поведение элемента **Оплата заказа** и являются его специализациями.

6.2 Начальное описание элемента Use Case

Элемент Use Case описывает, *что должна делать система*, но не определяет *как* она должна это делать. При моделировании это правило

позволяет отделять внешнее представление системы от её внутреннего представления.

Поведение элемента `Use Case` описывается потоком событий. В потоке событий выделяют:

- основной поток и альтернативные потоки поведения;
- условия (как и когда?) для запуска и остановки элемента `Use Case;`
- условия взаимодействия (когда?) элемента `Use Case` с актерами;
- какими данными обмениваются актер и система.

В общем случае один элемент `Use Case` описывает набор последовательностей, а каждая последовательность представляет возможный поток событий. Начальное описание выполняется в текстовой форме, удобной для пользователя системы.

Основным источником информации для анализа и проектирования систем являются спецификации элементов `Use Case`. Важно, чтобы спецификации были представлены в полной и конструктивной форме.

Спецификация включает главный поток, подпотоки и альтернативные потоки поведения.

Пусть, например, необходимо разработать начальное описание модели информационной системы авиакассы.

Пусть главный поток системы образуется начальным (базовым) элементом `Use Case`, включающим ряд действий, которые выбираются клиентом – покупателем авиабилета: **`Создать, Удалить, Проверить, Выполнить, Выйти`**. Для включения главного потока система должна произвести регистрацию клиента.

Ряд действий главного потока образуют подпотоки S_i:

`S1` – создать заказ авиабилета. Система выводит поле для указания пункта назначения и даты полета. Покупатель авиабилета вводит пункт назначения и дату, далее система отображает параметры авиарейсов на эту дату. Покупатель

выбирает авиарейс, после чего система устанавливает соединение “покупатель-авиарейс” и производит возврат к начальному элементу Use Case.

S2 – удалить заказ авиабилета. Система отображает параметры заказа, покупатель подтверждает решение о ликвидации заказа, система удаляет связь “покупатель-авиарейс” и производит возврат к начальному элементу Use Case.

S3 – проверить заказ авиабилета. Система отображает параметры заказа, покупатель подтверждает его, после чего система осуществляется возврат к начальному элементу Use Case.

S4 – выполнение заказа. Система запрашивает параметры кредитной карты, принимает необходимую информацию, снимает требуемую для оплаты заказа сумму и информирует клиента об окончании процедуры заказа. После этого осуществляется возврат к начальному элементу Use Case.

Альтернативные потоки A_j необходимы для случаев различных отклонений, которые могут возникнуть в подпотоках, например:

- A1 – введен неправильный регистрационный номер покупателя;
- A2 – нет подходящих авиарейсов;
- A3 – введен неправильный пункт назначения или дата полета;
- A4 – не устанавливается связь с авиарейсом;
- A5 – введены некорректные параметры кредитной карты.

В каждом из этих потоков покупателю предлагается либо повторить ввод, либо прекратить элемент Use Case.

Таким образом, в начальном описании данной спецификации элемента Use Case зафиксированы один основной поток и девять вспомогательных потоков действий. Разработчик может выделить из начального элемента Use Case самостоятельные элементы Use Case.При этом, если самостоятельный элемент содержит подпоток, то его следует подключить к базовому элементу Use Case отношением **include**, а если самостоятельный элемент содержит альтернативный поток, то его следует подключить к базовому элементу Use Case отношением **extend**.

6.3 Разработка спецификации для диаграммы Use Case

При построении диаграмм Use Case наибольшие трудности связаны с применением отношений включения **include** и расширения **extend**. Пример диаграммы, в которой использованы отношения включения и расширения, приведен на рисунке 6.5.

В этой диаграмме применен один базовый элемент Use Case – **Сеанс банкомата**, который взаимодействует с актером **Клиент**. К базовому элементу подключены два расширяющих элемента Use Case (**Состояние, Снять деньги**) и два включаемых элемента Use Case (**Идентификация клиента, Проверка счета**).

В свою очередь, к элементу Use Case **Идентификация клиента** подключен включаемый элемент Use Case **Проверить достоверность**, а к элементу Use Case **Снять деньги** – расширяющий элемент Use Case **Захват карты** (он же расширяет элемент Use Case **Проверить достоверность**).

Из рисунка 6.5 видно, что элемент Use Case **Сеанс банкомата** имеет две точки расширения (**диалог возможен, выдача квитанции**), а элементы Use Case **Снять деньги** и **Проверить достоверность** – по одной точке расширения (**проверка снятия, проверка**). Возможна также *вставка поведения* в точки расширения из расширяющего элемента Use Case, причем вставка происходит, если выполняются условия:

- для расширяющего элемента Use Case **Состояние – запрос состояния**;
- для расширяющего элемента Use Case **Снять деньги – запрос снятия**;
- для расширяющего элемента Use Case **Захват карты – список подозрений**.

Рисунок 6.5 – Пример применения отношений включения и расширения

Для базового элемента Use Case эти условия являются внешними, то есть базовому элементу ничего не известно об условиях запрос состояния и **запрос снятия**. Аналогично, элементам Use Case **Снять деньги** и **Проверить достоверность** ничего не известно об условии **список подозрений**. Таким образом, условия расширения являются следствием событий, происходящих во внешней среде.

Стрелки расширения в диаграмме подписаны указанием стереотипа (угловые скобки), именем точки расширения (круглые скобки) и условиями расширения (квадратные скобки). Описание расширяющего элемента Use Case разделено на сегменты, каждый сегмент обслуживает одну точку расширения базового элемента Use Case.

Количество точек расширяющего элемента Use Case равно количеству точек расширения базового элемента Use Case. Первый сегмент расширяющего элемента Use Case начинается с условия расширения, которое записывается только один раз, причем его действие распространяется на все остальные сегменты.

Поведение базового элемента Use Case задается внутренним потоком событий, вплоть до первой точки расширения. В точке расширения возможно выполнение расширяющего элемента Use Case, после чего возобновляется работа внутреннего потока.

Пример спецификации элементов Use Case рассматриваемой диаграммы имеет следующий вид:

Элемент Use Case **Сеанс банкомата**

Include (Идентификация клиента)	//включение
Include (Проверка счета)	//включение
(диалог возможен)	//первая точка расширения
	напечатать заголовок квитанции
(выдача квитанции)	//вторая точка расширения
конец сеанса	

Поведение модели, задаваемое диаграммой Use Case рисунка 6.5, может быть представлено следующим образом.

Актер **Клиент** инициирует действия базового элемента Use Case **Сеанс банкомата**. На первом шаге запускается включаемый элемент Use Case **Идентификация клиента**. Этот элемент Use Case получает имя клиента и запускает элемент Use Case **Проверить достоверность**. При этом устанавливается соединение с базой данных клиентов и извлекаются параметры клиента.

Если к этому моменту исполняется условие расширения **список подозрений**, то срабатывает расширяющий элемент **Захват карты**, карта арестовывается, и работа системы прекращается.

В противном случае происходит возврат к элементу Use Case **Идентификация клиента**, который получает номер счета клиента и возвращает управление базовому элементу Use Case.

Базовый элемент Use Case переходит ко второму шагу работы – вызывает включаемый элемент Use Case **Проверка счета**, который устанавливает соединение с базой данных счетов и получает информацию о состоянии и ограничении счета.

Управление снова возвращается к базовому элементу Use Case, который переходит к первой точке расширения **диалог возможен**. В этой точке возможно подключение *одного из двух* расширяющих элементов Use Case:

- по условию **запрос состояния** запускается первый сегмент элемента Use Case **Состояние**, в результате чего отображается информация о состоянии счета и управление передается базовому элементу Use Case. В базовом элементе Use Case печатается заголовок квитанции и производится переход ко второй точке расширения **выдача квитанции**. Если элемент Use Case **Состояние** продолжает находиться в активном состоянии, запускается его второй сегмент – **печать квитанции**.
- по условию **запрос снятия** запускается первый сегмент элемента Use Case **Снять деньги**, далее выполняются процедуры, описанные в спецификации расширяющего элемента Use Case **Снять деньги.**

Сеанс работы банкомата завершается возвращением к базовому элементу Use Case.

6.4 Разработка модели требований

Основное назначение диаграмм `Use Case` – определение требований заказчика к будущему программному приложению.

Использование системы по её прямому назначению – это удовлетворение требований потребителей, пользователей.

Разделение проекта на варианты использования является таким способом изучения системы, который ориентирован на сам процесс, а не на его реализацию. Если при функциональной декомпозиции задача заключается в последовательной разбивке проблемы на небольшие фрагменты, с которыми будет работать готовая система, то при построении диаграммы вариантов использования необходимо сосредоточиться, прежде всего, на том, что ожидает от системы пользователь.

Варианты использования концентрируют внимание на том, ***что*** должна делать система, а не на том, ***как*** она должна это делать. Каждый вариант использования должен представлять собой завершенную транзакцию между пользователем и системой.

Базовый материал, необходимый для построения диаграмм вариантов использования, разработчик получает при непосредственном общении с потенциальными пользователями системы. Пользователям необходимо подробно описать функции или задачи, которые они выполняют. В результате беседы определяются выполняемые действия, действующие лица (кто инициирует действие) и список требований. Все это необходимо задокументировать (отобразить) на диаграммах. Например, фраза "Бухгалтер формирует прибыльную накладную на товар" приводит к такому пониманию:

- действующее лицо (Актер) – "Бухгалтер";
- вариант использования (прецедент) – "Сформировать прибыльную накладную " (прецедент обычно начинается из глагола);
- требование к системе – "система должна поддерживать операцию формирования приходно-расходных накладных на товар".

Далее необходимо изобразить всех актеров и те действия (прецеденты), которые они могут выполнять на одной или нескольких диаграммах Use case, а потом связать каждый прецедент с требованием.

После того, как все актеры и прецеденты были нанесены на диаграммы, необходимо убедиться в том, что найдены все варианты использования. Для этого следует поставить следующие вопросы:

- Присутствует ли конкретное функциональное требование хотя бы в одном варианте использования? Если требование не нашло отображения в варианте использования, оно не будет реализовано.
- Учтено ли, как будет работать с системой каждое заинтересованное лицо, какую информацию это лицо будет передавать системе, какую информацию оно будет получать от системы?
- Учтены ли все внешние системы, с которыми будет взаимодействовать данная система?
- Какой информацией каждая внешняя система будет обмениваться с данной системой?

После проведения такого анализа разрабатывается документ, который называется ***потоком событий*** (`flow of events`). Этот документ подробно описывает, что будут делать пользователи системы и что будет делать сама система. Обычно поток событий содержит:

Сжатое описание. Каждый вариант использования должен иметь связанное с ним краткое описание того, что он будет делать. Например, вариант использования "Сформировать налоговую накладную" системы учета материальных ценностей может содержать такое описание: Вариант использования "Сформировать налоговую накладную" разрешает бухгалтеру оформить и распечатать налоговую накладную для проведения расчетов в связи с продажей материалов внешней организации.

Предпосылки (`pre-conditions`) – это такие условия, которые должны быть выполнены, прежде чем вариант использования начнет свою работу. Например, таким условием может быть выполнение другого варианта

использования, или наличие у пользователя прав доступа, необходимых для запуска данного варианта использования.

Основной и альтернативный потоки событий. Конкретные детали вариантов использования отражаются в основном и альтернативном потоках событий. Поток событий поэтапно описывает, что должно происходить во время выполнения заложенной в варианты использования функциональности. Первичный и альтернативный потоки событий содержат описание того, каким образом запускается вариант использования, какие пути используются для выполнения вариантов использования, какие отклонения от основного потока событий возможны (альтернативные потоки), каким образом завершается вариант использования. Документируя поток событий, можно использовать нумерованные списки, ненумерованные списки, разбитый на параграфы текст и даже блок-схемы. Поток событий должен быть согласован с определенными раньше требованиями. При его изучении заказчики будут проверять, отвечает ли он их ожиданиям, а аналитики – отвечает ли он требованиям к системе.

Постусловия (`post-conditions`). Постусловиями называются такие условия, которые должны быть выполнены после завершения варианта использования. Например, в конце варианта использования можно установить флажок. Информация такого типа входит в состав постусловий. Как и в случае предпосылок, с помощью постусловий можно вводить сведения о порядке выполнения вариантов использования системы. Если, например, после одного из вариантов использования должен всегда выполняться другой, это можно описать как постусловие. Такие условия существуют не в каждом варианте использования.

В процессе создания диаграммы использования необходимо выполнить работу по поиску действующих лиц – актеров.

Действующее лицо **(`actor`)** – это то, что взаимодействует с создаваемой системой. Если варианты использования описывают все, что происходит внутри системы, действующее лицо определяет все, что находится вне нее.

Действующие лица делятся на три основных типа: пользователи системы, другие системы, взаимодействующие с данной, и время.

Первый тип действующих лиц – это физические лица. Они наиболее типичны и существуют практически в каждой системе. Называя действующих лиц, необходимо использовать их ролевые имена, а не те, что отвечают их должности.

Вторым типом действующих лиц является другая система. Например, информация о прибыли и расходах товарно-материальных ценностей на складах в системе учета материальных ценностей (УМЦ) может использоваться в системе учета финансово-расчетных операций (УФРО) для учета расчетов с поставщиками. В таком случае последняя становится действующим лицом.

Следует учесть, что если система становится действующим лицом, то она не будет изменяться вообще, действующие лица находятся вне сферы разработки и, следовательно, не подлежат контролю.

Третий, наиболее распространенный тип действующего лица – время, которое также не подлежит нашему контролю. Время становится действующим лицом, если от него зависит запуск каких-нибудь событий в системе.

С помощью *связи обобщения действующего лица* (**actor generalization relationship**) можно показать, что у нескольких действующих лиц существуют общие черты. Например, клиенты могут быть двух типов: корпоративные и индивидуальные. Это отношение моделируется с помощью нотации, представленной на рисунке 6.6.

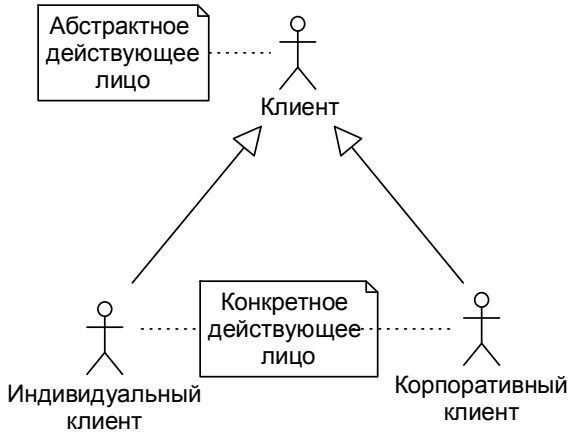

Рисунок 6.6 – Связь обобщения действующих лиц

Действующее лицо **`Клиент`** – абстрактный тип, который никогда не инстанцинируется непосредственно. Абстрактное действующее лицо нужно только для того, чтобы показать наличие двух типов клиентов.

6.5 Рекомендации по разработке диаграмм вариантов использования

Главное назначение диаграммы вариантов использования заключается в формализации функциональных требований к системе с помощью понятий соответствующего пакета и возможности согласования полученной модели с заказчиком на ранней стадии проектирования. Любой из вариантов использования может быть подвергнут дальнейшей декомпозиции на множество подвариантов использования отдельных элементов, которые образуют исходную сущность. Рекомендуемое общее количество актеров в модели — не более 20, а вариантов использования — не более 50. В противном случае модель теряет свою наглядность и, возможно, заменяет собой одну из некоторых других диаграмм.

Семантика построения диаграммы вариантов использования должна определяться следующими особенностями:

1. Экземпляр отдельного варианта использования инициализируется (запускается) посредством сообщения от экземпляра актера. В качестве отклика или ответной реакции на сообщение актера экземпляр варианта использования выполняет последовательность действий, установленную для данного варианта использования. Экземпляры актеров могут генерировать новые экземпляры сообщений *для других* экземпляров вариантов использования.

2. Взаимодействие актера с экземплярами вариантов использования будет продолжаться до тех пор, пока не закончится выполнение требуемой последовательности действий экземпляром варианта использования, и соответствующий экземпляр актера (и никакой другой) не получит требуемый экземпляр сервиса.

3. Варианты использования могут быть специфицированы в виде текста, а в последующем — с помощью операций и методов вместе с атрибутами, в виде *графа деятельности*, посредством автомата или любого другого механизма описания поведения, включающего предусловия и постусловия. Взаимодействие между вариантами использования и актерами может уточняться на диаграмме кооперации, когда описываются взаимосвязи между сущностью, содержащей эти варианты использования, и окружением или внешней средой этой сущности.

4. В случае, когда для представления иерархической структуры проектируемой системы используются подсистемы, система может быть определена в виде вариантов использования на всех уровнях. Отдельные подсистемы или классы могут выступать в роли таких вариантов использования. Вариант использования в целом может рассматриваться как суперсервис для уточняющих его подвариантов, которые, в свою очередь, могут рассматриваться как подсервисы исходного варианта использования.

5. Окружение вариантов использования нижнего уровня является самостоятельным элементом модели, который, в свою очередь, содержит другие элементы модели, определенные для этих вариантов использования.

6. Варианты использования классов соответствуют ***операциям*** этого класса, поскольку сервис класса является по существу выполнением операций данного класса. Некоторые варианты использования могут соответствовать применению только одной операции, в то время как другие — конечного множества операций, определенных в виде последовательности операций. В то же время одна операция может быть необходима для выполнения нескольких сервисов класса и поэтому будет появляться в нескольких вариантах использования этого класса.

7. Если в качестве моделируемой сущности выступает система или подсистема самого верхнего уровня, то отдельные пользователи вариантов использования этой системы моделируются *актерами*. Такие актеры, являясь

внутренними по отношению к моделируемым подсистемам нижних уровней, часто в явном виде не указываются, хотя и присутствуют неявно в модели подсистемы. Вместо этого варианты использования непосредственно обращаются к тем модельным элементам, которые содержат в себе подобные неявные актеры, то есть экземпляры которых играют роли таких актеров при взаимодействии с вариантами использования.

8. С системно-аналитической точки зрения построение диаграммы вариантов использования специфицирует не только функциональные требования к проектируемой системе, но и выполняет исходную *структуризацию предметной области*. Решение этой задачи требует умения выделять главное в модели системы и стремиться к возможно более точному представлению модели именно в форме диаграммы вариантов использования.

Если же варианты использования применяются для спецификации части системы, то они будут эквивалентны соответствующим вариантам использования в модели подсистемы для части соответствующего пакета. Важно учесть, что *все сервисы системы должны быть явно определены на диаграмме вариантов использования*, и никаких других сервисов, которые отсутствуют на данной диаграмме, проектируемая система не может выполнять по определению. Более того, если для моделирования реализации системы используются сразу несколько моделей, например, модель анализа и модель проектирования, то множество вариантов использования должно быть эквивалентно множеству вариантов использования модели в целом.

7 РАЗРАБОТКА ДИАГРАММ ВЗАИМОДЕЙСТВИЯ

7.1 Назначение и содержание диаграмм взаимодействия

Диаграммы взаимодействий (i**nteraction diagrams**) используются ***для моделирования динамических аспектов системы***. Сюда входит моделирование конкретных и прототипических экземпляров классов, интерфейсов, компонентов и узлов, а также сообщений, которыми они обмениваются. Диаграммы взаимодействий могут существовать автономно и служить для визуализации, специфицирования, конструирования и документирования динамики конкретного сообщества объектов, а могут использоваться для моделирования отдельного потока управления в составе прецедента.

Диаграммы взаимодействий важны также для создания исполняемых систем посредством прямого и обратного проектирования.

К диаграммам взаимодействия относятся два типа диаграмм:

1. ***Диаграмма последовательностей*** (`Sequence diagram`) – диаграмма взаимодействий, акцентирующая внимание на временной упорядоченности сообщений.
2. ***Диаграмма кооперации*** (`Collaboration diagram`) – диаграмма взаимодействий, основное внимание в которой уделяется структурной организации объектов, принимающих и отправляющих сообщения.

На диаграмме взаимодействия отображают один из процессов обработки информации в варианте использования. В силу того, что для одного варианта использования может быть несколько альтернативных потоков, то для данного варианта использования нужно создать несколько диаграмм взаимодействия.

Для создания ***диаграммы последовательности*** необходимо, прежде всего, расположить объекты, участвующие во взаимодействии, в верхней ее

части вдоль горизонтальной оси. Обычно инициирующий взаимодействие объект размещают слева, а остальные – правее, по принципу подчиненности объектов. Объекты, не относящиеся к выстроенной иерархии подчиненности, располагают ниже. От каждого объекта вниз проводится линия его жизни, на которой размещаются маленькие прямоугольники – фокусы управления. Фокус управления предназначен для отображения активного состояния объекта. Затем на вертикальной оси времени размещаются горизонтальные линии сообщений, которые объекты посылают и принимают. Это позволяет получить наглядную картину развития потока управления во времени (рис. 7.1).

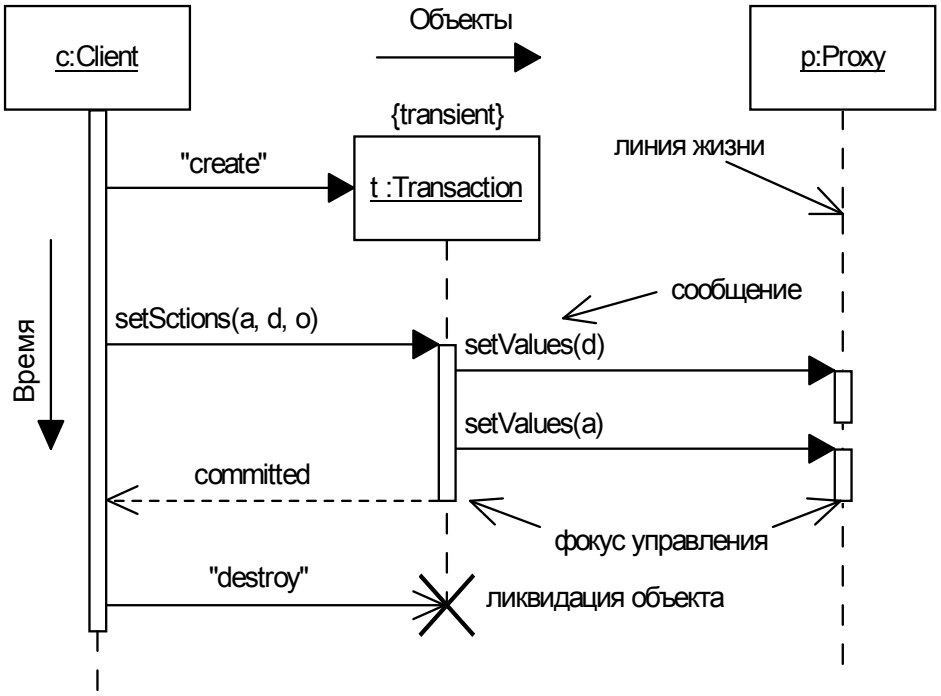

Рисунок 7.1 – Пример оформления диаграммы последовательностей

Для создания ***диаграммы кооперации*** нужно расположить участвующие во взаимодействии объекты в виде вершин графа. Связи, соединяющие эти объекты, изображаются в виде дуг этого графа. Связи дополняются пронумерованными сообщениями, которые объекты принимают и посылают. Это дает пользователю ясное визуальное представление о потоке управления в контексте структурной организации кооперирующихся объектов (рис. 7.2).

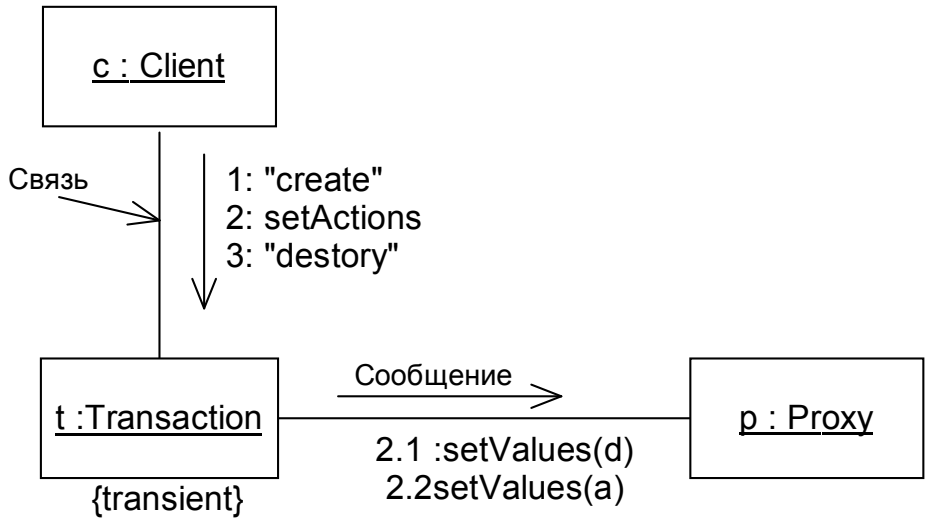

Рисунок 7.2 – Пример оформления кооперативной диаграммы

Сообщениями называется спецификация обмена данными между объектами. Сообщение представляет собой некую информацию, передаваемую в расчете на то, что в ответ последует определенное действие. Получение объектом сообщения можно считать экземпляром события. Чаще всего встречающийся в моделях вид сообщений – вызов одним объектом операции другого объекта, или вызов собственной операции. Сообщения представляют в виде линии со стрелкой и почти всегда добавляют название соответствующей операции, так как объект не может вызвать произвольную операцию.

Действие – это исполняемое предложение, которое образует абстракцию вычислительной процедуры. Действие может привести к изменению состояния.

UML позволяет моделировать действия нескольких видов:

call (вызвать) – вызывает операцию, применяемую к объекту. Объект может послать сообщение самому себе, что приведет к локальному вызову операции;

return (возвратить) – возвращает значение вызывающему объекту;

send (послать) – посылает объекту сигнал;

create (создать) – создает новый объект;

destroy (уничтожить) – удаляет объект.

7.2 Правила формирования диаграммы последовательностей

На диаграмме последовательности изображаются исключительно те объекты, которые непосредственно участвуют во взаимодействии и не показываются возможные статические ассоциации с другими объектами. Для диаграммы последовательности ключевым моментом является именно динамика взаимодействия объектов во времени. При этом диаграмма последовательности имеет как бы два измерения.

Одно измерение — слева направо в виде вертикальных линий, жизни объектов, участвующих во взаимодействии. Графически каждый объект изображается прямоугольником и располагается в верхней части своей линии жизни (рис. 7.1). Внутри прямоугольника записываются имя объекта и имя класса, разделенные двоеточием. При этом вся запись подчеркивается, что является признаком объекта, который представляет собой экземпляр класса. Каждому объекту на диаграмме последовательности должно быть дано уникальное имя. В отличие от имен классов, которые, как правило, носят общий характер, имена объектов всегда конкретные. На диаграмме взаимодействия может быть два объекта, которые являются экземплярами того самого класса.

Создавая диаграммы последовательности, следует учитывать, что объектам назначаются определенные обязательства. Объекты и их обязательства должны отвечать друг другу. **О**бъект, который инициирует взаимодействия, изображается на диаграмме крайним слева. Правее изображается другой объект, который непосредственно взаимодействует с первым. Таким образом, все объекты на диаграмме последовательности образуют некоторый порядок, определяемый степенью активности этих объектов при взаимодействии друг с другом.

Хорошим способом первоначального выявления некоторых объектов является изучение существительных в потоке событий. Можно также прочитать документы, которые описывают конкретный сценарий. Под сценарием

понимается конкретный экземпляр потока событий. Некоторые существительные в сценариях будут действующими лицами, другие – объектами, а третьи – атрибутами объекта. При разработке диаграмм последовательности существительные помогут определить, что может быть объектом. Если не понятно, что описывает существительное – объект или атрибут, следует определить, есть ли у него поведение. Если поведение отсутствует, то это, вероятно, атрибут. Если для существительного определяется поведение, то, скорее всего, это объект.

Другое измерение диаграммы последовательности — вертикальная временная ось, направленная сверху вниз. Начальному моменту времени соответствует самая верхняя часть диаграммы. Сообщения изображаются в виде горизонтальных стрелок с именем сообщения и образуют порядок по времени своего возникновения. При этом масштаб на оси времени не указывается, поскольку диаграмма последовательности моделирует лишь временную упорядоченность взаимодействий типа "раньше-позже". Линия жизни служит для обозначения периода времени, в течение которого объект существует в системе и, следовательно, может потенциально участвовать во всех ее взаимодействиях. Если объект существует в системе постоянно, то и его линия жизни должна продолжаться по всей плоскости диаграммы последовательности от самой верхней ее части до самой нижней.

Отдельные объекты, исполнив свою роль в системе, могут быть уничтожены (разрушены), чтобы освободить занимаемые ими ресурсы. Для таких объектов линия жизни обрывается в момент его уничтожения. Для обозначения момента уничтожения объекта в языке UML используется специальный символ в форме латинской буквы "X" (рис. 7.1). Ниже этого символа пунктирная линия не изображается, поскольку соответствующего объекта в системе уже нет, и этот объект должен быть исключен из всех последующих взаимодействий.

Вовсе не обязательно создавать все объекты в начальный момент времени. Отдельные объекты в системе могут создаваться по мере необходимости, существенно экономя ресурсы системы и повышая ее производительность. В этом случае прямоугольник такого объекта изображается не в верхней части диаграммы последовательности, а в той ее части, которая соответствует моменту создания объекта (объект **`t`** класса **`Transaction`** на рисунке 7.1). При этом прямоугольник объекта располагается в том месте диаграммы, которое по оси времени совпадает с моментом его возникновения в системе. Очевидно, что объект обязательно создается со своей линией жизни и, возможно, с фокусом управления.

Фокус управления изображается в форме вытянутого узкого прямоугольника, верхняя сторона которого обозначает начало получения фокуса управления объекта (начало активности), а ее нижняя сторона — окончание фокуса управления (окончание активности). Этот прямоугольник может заменять линию жизни объекта, если на всем ее протяжении он является активным. Периоды активности объекта могут чередоваться с периодами его пассивности или ожидания. В этом случае у такого объекта имеются несколько фокусов управления. Важно понимать, что получить фокус управления может только существующий объект, у которого в этот момент имеется линия жизни. Если же некоторый объект был уничтожен, то вновь возникнуть в системе он уже не может. Вместо него может быть создан другой экземпляр этого же класса, который, строго говоря, будет являться другим объектом.

В отдельных случаях инициатором взаимодействия в системе может быть актер или внешний пользователь. В этом случае актер изображается на диаграмме последовательности самым первым объектом слева со своим фокусом управления (рис. 7.3). Чаще всего актер и его фокус управления будут существовать в системе постоянно, отмечая характерную для пользователя активность в инициировании взаимодействий с системой. При этом сам актер может иметь собственное имя либо оставаться анонимным.

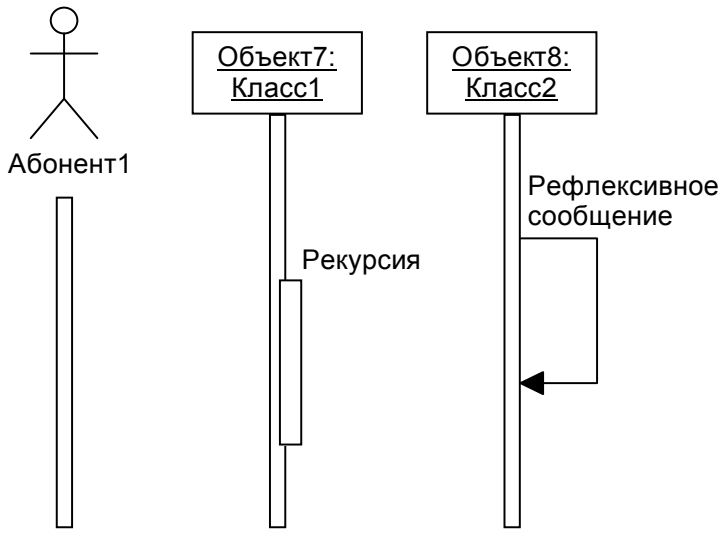

Рисунок 7.3 – Графические примитивы диаграмм последовательности

Объект может инициировать ***рекурсивное взаимодействие*** с самим собой. Необходимость такого взаимодействия связана с тем, что наличие во многих языках программирования специальных средств построения рекурсивных процедур требует визуализации соответствующих понятий в форме графических примитивов. На диаграмме последовательности ***рекурсия*** обозначается небольшим прямоугольником, присоединенным к правой стороне фокуса управления того объекта, для которого изображается это рекурсивное взаимодействие (объект 7 на рис. 7.3).

В отдельных случаях объект может посылать сообщения самому себе, инициируя так называемые ***рефлексивные сообщения*** (объект8 на рис. 7.3). Такие сообщения изображаются либо прямоугольником со стрелкой, начало и конец которой совпадают, либо дуговой стрелкой. Рефлексивные сообщения используют, например, при обработке нажатий на клавиши клавиатуры или при наборе цифр номера телефона абонента.

7.3 Правила моделирования взаимодействий

Взаимодействие объектов представляется сообщениями, которые являются спецификацией передачи информации и показывают, что один объект вызывает функцию другого. Дальше, когда будут определены операции классов, каждое сообщение станет операцией.

Графические примитивы, используемые для обозначения взаимодействий, приведены на рисунке 7.4.

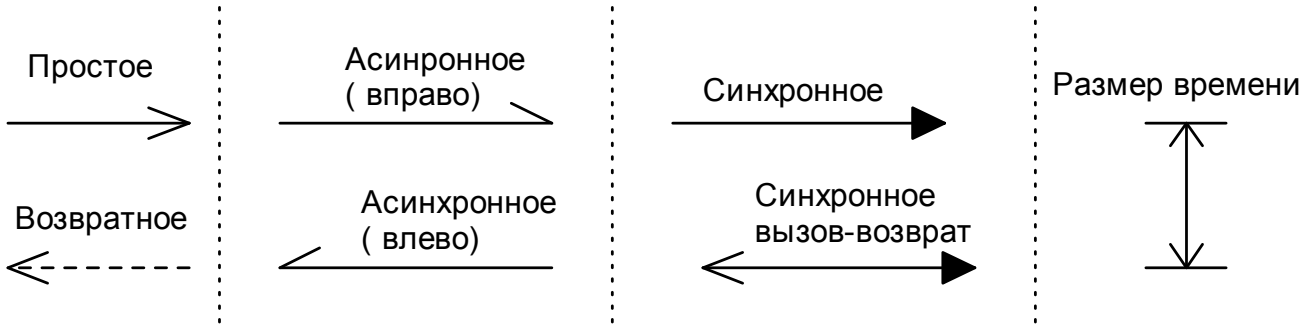

Рисунок 7.4 Обозначения разновидностей взаимодействий

В языке UML могут встречаться следующие разновидности сообщений:

– ***Простое*** (simple) – сообщение для вызова процедур, выполнения операций или обозначения отдельных потоков управления. Начало этой стрелки всегда соприкасается с фокусом управления или линией жизни того объекта-клиента, который инициирует это сообщение. Конец стрелки соприкасается с линией жизни того объекта, который принимает это сообщение и выполняет в ответ определенные действия. При этом принимающий объект зачастую получает и фокус управления, становясь активным.

– ***Возвратное*** (message return) – используется для обозначения возврата из процедуры и обозначается пунктирной линией.

– ***Синхронное*** (synchronous) – используется для задания процедурных или вложенных потоков.

– ***Асинхронное*** (asynchronous) – используется для обозначения простого (не вложенного) потока управления. Каждая такая стрелка указывает на прогресс одного шага потока. При этом сообщения могут возникать в произвольные моменты времени. Передача такого сообщения обычно сопровождается получением фокуса управления объектом, который его принял, а клиент, пославший сообщение серверу, продолжает свою работу, не ожидая подтверждения о получении.

– ***Синхронно-асинхронное*** (synchronous /asynchronous) – используется для обозначения таких сообщений, при которых клиент посылает запрос и ждет ответа пользователя. Такие сообщения следует применять, например, для обозначения входа в прерывания и выхода из прерываний.

Пример оформления фрагмента диаграммы последовательности с различными сообщениями показан на рисунке 7.5.

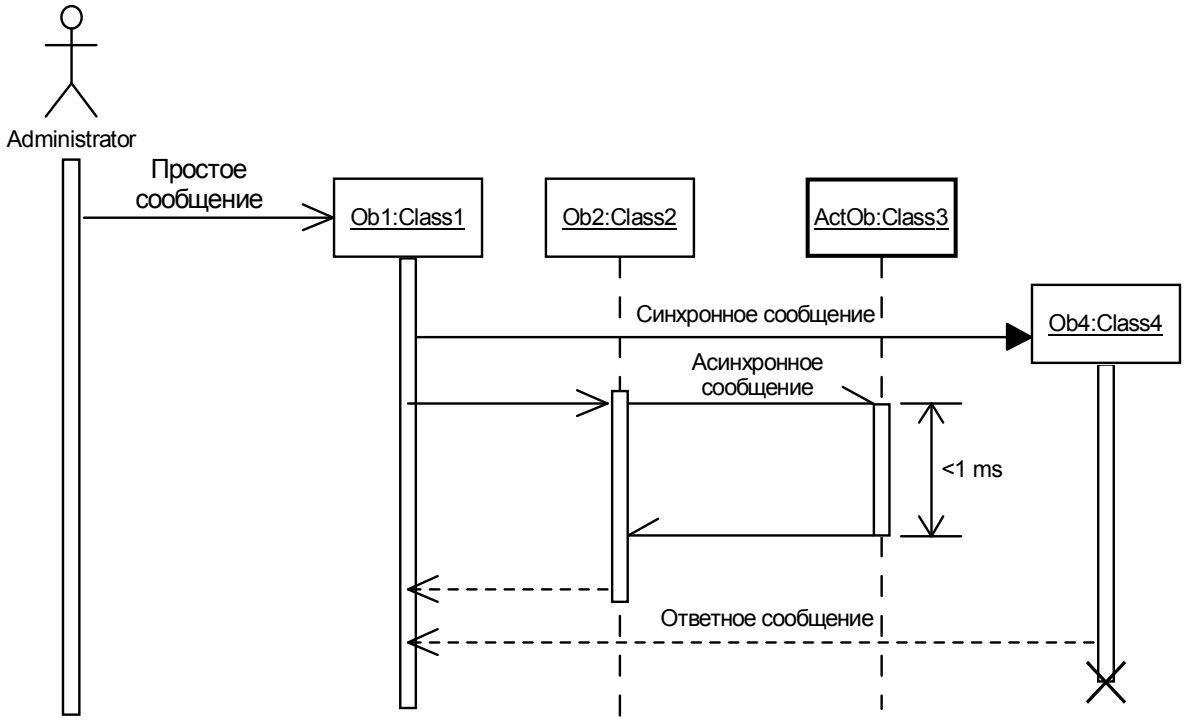

Рисунок 7.5 – Примеры оформления различных типов сообщений

Для указания того, что сообщения посылаются через регулярные временные интервалы, можно указать частоту сообщения. Для процессов, лимитированых по времени, можно указывать длительность операции, например, как показано на рисунке 7.5 (<1 ms).

Для записи сообщений в UML принят следующий синтаксис:

ВозврВеличина : = ИмяСообщения {Аргументы},

где **ВозврВеличина** обозначает результат обработки сообщения.

Примеры записи сообщений:

- "create" – вызов стандартной процедуры создания объекта;
- корд:=текущПоложение(осьX) – вызов операции с возвратом значения;
- оповещение( ) – посылка сигнала;
- установитьПозицию(a) – вызов операции с действительными параметрами.

Примеры записей возвращаемой величины:

Вызов – в объекте запускается операция;

Возврат – возврат значений в вызывающий объект;

Посылка (Send) – в объект посылается информационный сигнал;

Создание – создание объекта по сообщению "create";

Уничтожение – уничтожение объекта по сообщению "destroy".

7.4 Примеры построения диаграмм последовательности

В качестве примера рассмотрим построение диаграммы последовательности для моделирования процесса телефонного разговора с использованием обычной телефонной сети. Объектами в этом примере являются два абонента **a** и **b**, два телефонных аппарата **c** и **d**, коммутатор и собственно

разговор, как объект моделирования. Будем полагать, что **Коммутатор** и **Разговор** являются анонимными объектами.

Располагая объекты на предполагаемой диаграмме, будем рассматривать абонентов как актеров, причем абоненту “**a**” зададим активную роль, а абоненту “**b**” — пассивную роль. В связи с этим абонент “**a**” получает сразу фокус управления, а второй имеет только линию жизни.

Так как все объекты должны принадлежать определенному классу, то устанавливаем следующую классификацию:

- абоненты “**a**” и “**b**” являются экземплярами класса “**Абонент**”;
- телефонные аппараты “**c**” и “**d**” являются экземплярами класса “**Телефонный аппарат**”;
- безымянные объекты – “**Коммутатор**” и “**Разговор**”.

Процесс взаимодействия в этой системе начинается с поднятия трубки телефонного аппарата первым абонентом. Тем самым он посылает сообщение телефонному аппарату “**c**”, которое переводит этот аппарат в активное состояние и вызывает действие — подачу тонового сигнала в телефонную трубку для первого абонента. Следующее действие также инициируется первым абонентом — набор цифр телефонного номера. Это представлено в форме итеративного сообщения со знаком "*" слева от его имени.

Следует учесть, что поднятие телефонной трубки и набор цифр номера являются физическими действиями и поэтому изображаются в форме простых асинхронных сообщений. После набора номера телефона аппарат “**c**” рекурсивно вызывает процедуру посылки коммутационных импульсов на **Коммутатор**. **Коммутатор** инициирует создание нового объекта в моделируемой системе — “**Разговор**”. Фрагмент диаграммы последовательности для этого процесса изображен на рис. 7.6.

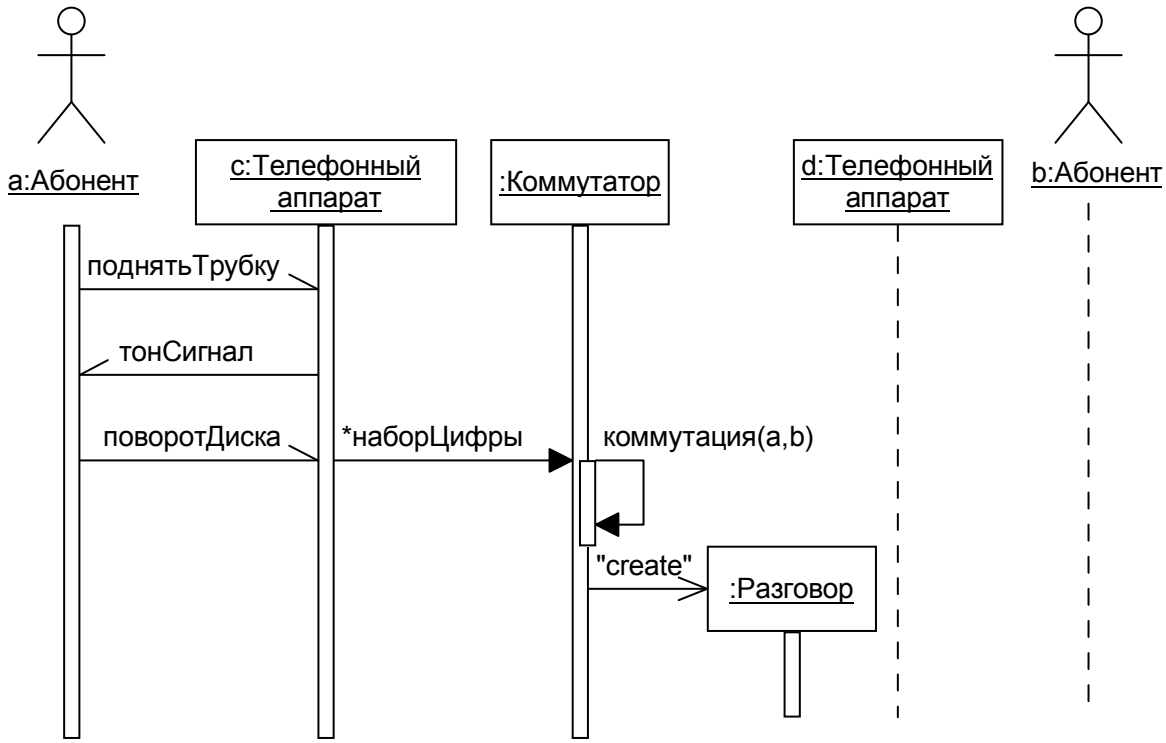

Рисунок 7.6 – Фрагмент моделирования подачи вызова на коммутатор

После создания анонимного объекта "**Разговор**" он сразу получает фокус активности и посылает сообщение телефонному аппарату **d** на выполнение действия — звонка вызова (рис. 7.7). После того, как второй абонент снимает трубку (асинхронное сообщение), устанавливается прямое соединение между абонентами **a** и **b**. Объект "**Разговор**" уничтожается после того, как абоненты опустят трубки. Окончательный вариант диаграммы последовательности может содержать некоторые временные ограничения и комментарии.

Таким образом, при моделировании временной упорядоченности потока управления необходимо выполнить следующие действия:

1. Определить сцену для взаимодействия, выяснив, какие объекты принимают в нем участие, разместить объекты на диаграмме последовательностей слева направо так, чтобы более важные объекты были расположены левее, провести для каждого объекта линию жизни, отметить на линиях жизни моменты рождения и смерти.

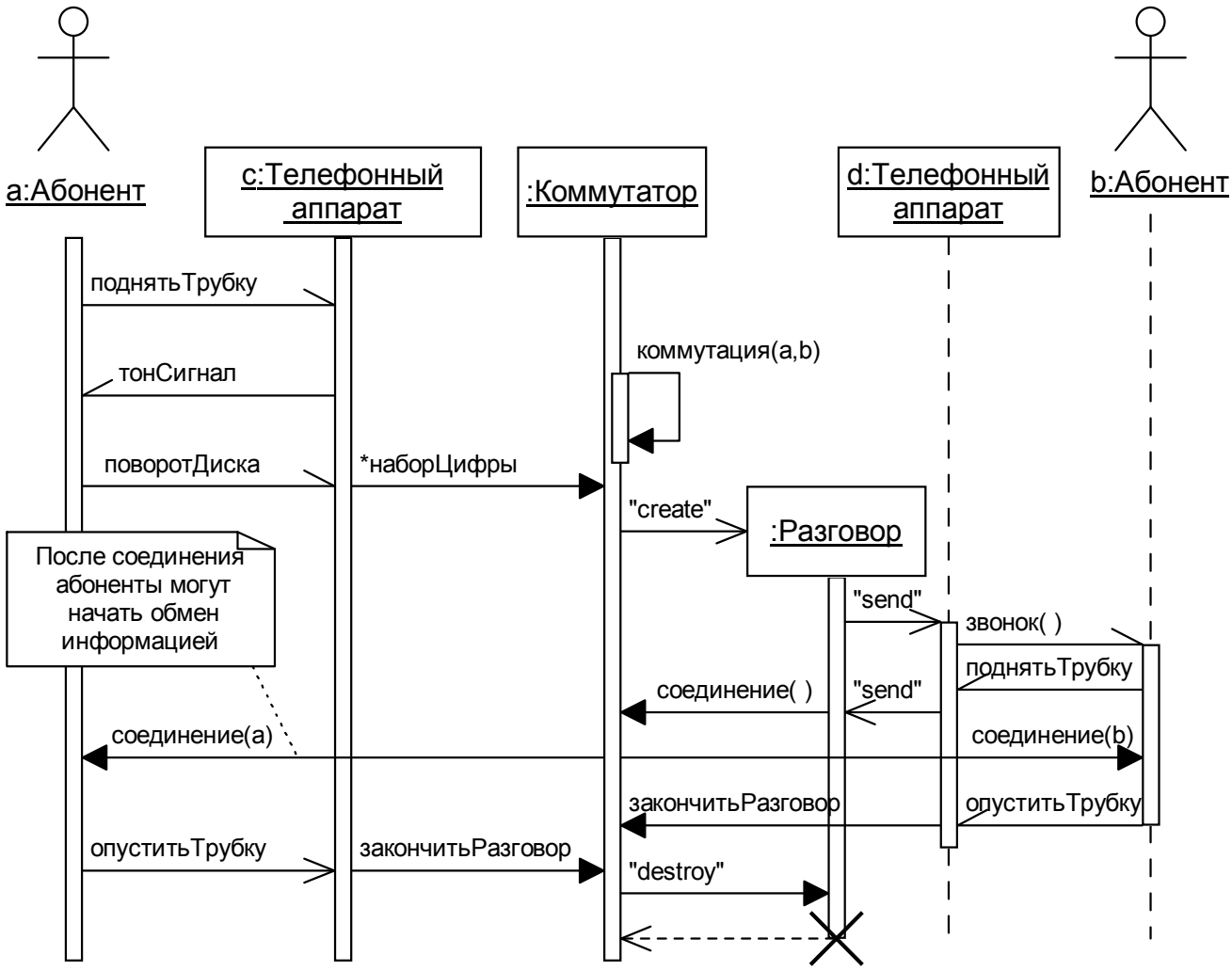

Рисунок 7.7 – Диаграмма последовательности для моделирования телефонного разговора

2. Начав с сообщения, инициирующего взаимодействие, расположить все последующие сообщения сверху вниз между линиями жизни объектов. Если необходимо объяснить семантику взаимодействия, показать свойства каждого сообщения (например, его параметры). Если необходимо специфицировать временные или пространственные ограничения, дополнить сообщения отметками времени и присоединить соответствующие ограничения.

3. Если требуется показать вложенность сообщений или точный промежуток времени, когда происходят вычисления, дополнить линии жизни объектов фокусами управления.

Еще один пример построения диаграммы последовательности относится к моделированию процесса управления электроприводом (рис. 7.8).

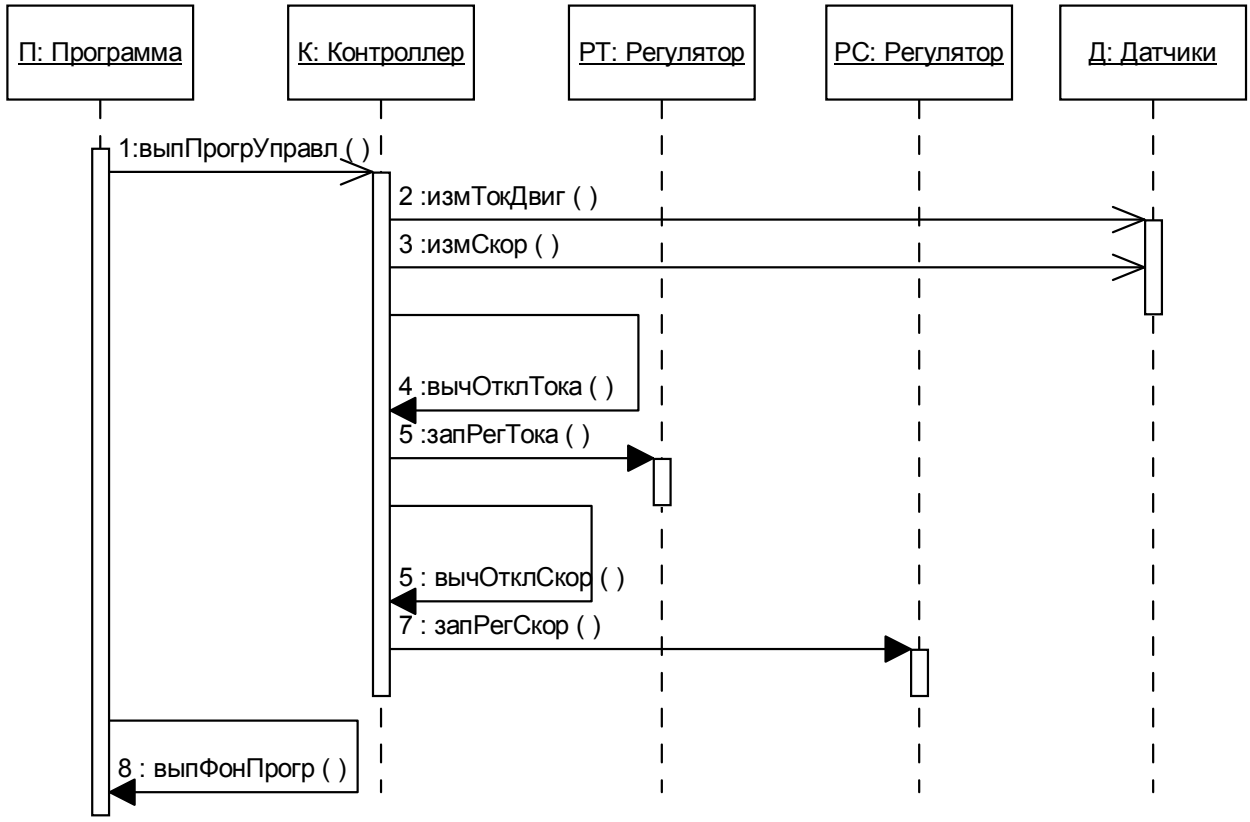

Рисунок 7.8 – Диаграмма последовательности системы управления электроприводом

На этой диаграмме показано взаимодействие объектов, составляющих предметные сущности процесса управления – программа П, контроллер К, регулятор тока РТ, регулятор скорости РС и датчики Д, включающие в себя датчики тока и скорости.

Последовательность выполнения операций обозначена порядковыми номерами:

1. Управляющая программа **П** посылает контроллеру **К** простое сообщение **выпПрогрУправл( )** – выполнить программу управления электроприводом.

2. Контроллер **К** получает фокус управления и направляет сообщение **измТокДвиг( )** – измерить текущее значение тока двигателя.
3. Датчик тока возвращает конкретное значение тока двигателя, после чего контроллер **К** направляет сообщение объекту **Д измСкор( )** – измерить текущее значение скорости двигателя.
4. Датчик скорости возвращает контроллеру конкретное значение скорости и котроллер переходит на выполнение рефлексивного сообщения **вычОтклТока( )** – вычислить отклонение тока.
5. Отклонение тока от заданного значения записывается в регистр регулятора тока.
6. Контроллер выполняет рефлексивное сообщение **вычОтклСкор ( )** – вычисляет отклонение скорости от заданного значения.
7. Отклонение скорости записывается в регистр регулятора скорости.
8. Управляющая программа вызывает диспетчера фонового режима для обслуживания пульта оператора.

7.4 Особенности разработки кооперативных диаграмм

Для создания диаграммы кооперации нужно расположить участвующие во взаимодействии объекты в виде вершин графа. Связи, соединяющие эти объекты, будут представлять собой сообщения, которые объекты принимают и посылают. Это дает пользователю ясное визуальное представление о потоке управления в контексте структурной организации кооперирующихся объектов.

У диаграмм кооперации есть два свойства, которые отличают их от диаграмм последовательностей.

Первое свойство – это путь. Для описания связи одного объекта с другим к дальней концевой точке этой связи можно присоединить стереотип пути, например, **local**, показывающий, что помеченный объект является локальным по отношению к отправителю сообщения (рис. 7.9).

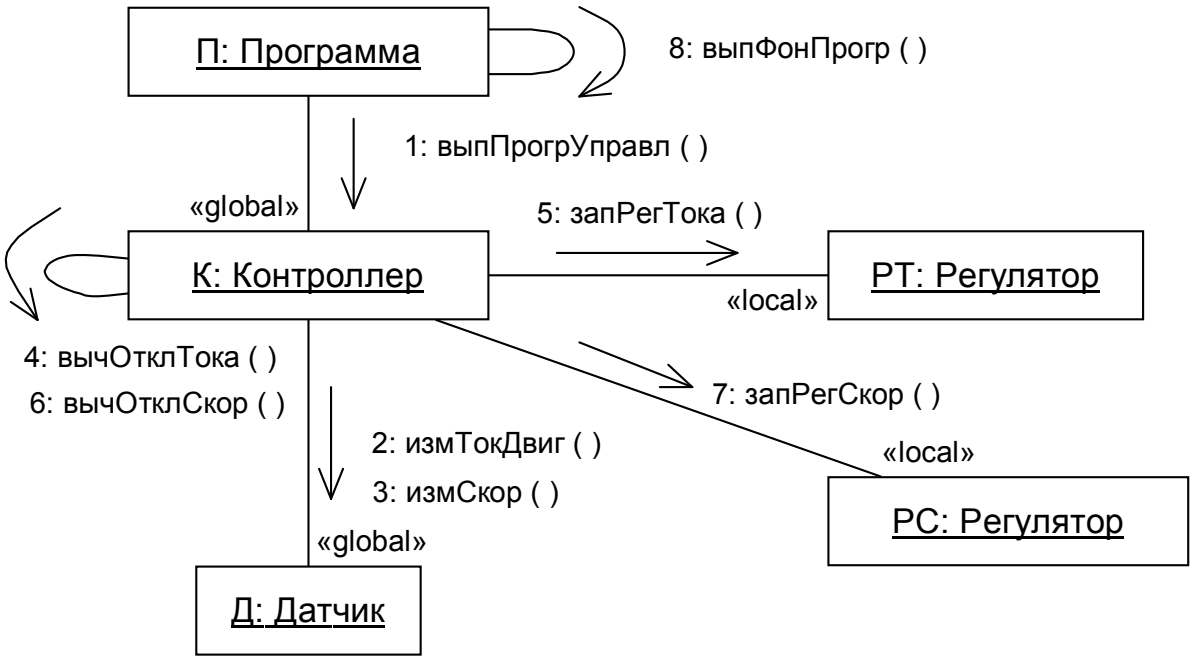

Рисунок 7.9 – Кооперативная диаграмма для системы управления электроприводом

Второе свойство – это порядковый номер сообщения. Для обозначения временной последовательности перед сообщением можно поставить его порядковый номер, начиная с единицы. Для обозначения вложенности используется десятичная нотация: 1 – первое сообщение; 1.1 – первое сообщение, вложенное в сообщение 1 и т. д. Уровень вложенности не ограничен.

Диаграммы кооперации предпочтительно применять для моделирования сложных потоков, содержащих итерации и ветвления. На диаграммах последовательностей такие потоки обычно не показывают. Итерация представляет собой повторяющуюся последовательность сообщений. Для ее моделирования перед номером сообщения последовательности ставится выражение итерации, например, **`* [i:= 1 ... n]`**, или просто *, если надо обозначить итерацию без детализации. Итерация показывает, что сообщения и все вложенные в него сообщения будут повторяться в соответствии со значением заданного выражения.

На дугах, соединяющих вершины, можно задать условие выполнения сообщения. Для моделирования условия перед порядковым номером сообщения ставится выражение, например, **[x>0]**. У всех альтернативных ветвей устанавливается один и тот же порядковый номер, но условия на каждой ветви должны быть заданы так, чтобы два из них не выполнялись одновременно.

7.5 Двухэтапный подход к разработке диаграмм взаимодействия

При разработке диаграмм взаимодействия часто применяется двухэтапный подход.

Сущность двухэтапного подхода заключается в следующем.

На первом этапе построения диаграммы взаимодействия на ней отображается информация высокого уровня, которая нужна конечным пользователям системы. Здесь сообщения еще не соотносятся с операциями, и объекты могут быть не соотнесены с классами. Эти диаграммы позволяют аналитикам, пользователям и всем заинтересованным в бизнес-процессах лицам увидеть, как будут развиваться события в системе.

На втором этапе, после того как пользователи придут к согласию по поводу полученной диаграммы, можно углубиться в детали. При этом диаграмма может потерять свою полезность для пользователей, но станет важна для разработчиков, тестировщиков и других участников проекта.

В начале второго этапа на диаграмму наносят некоторые новые объекты. Как правило, на каждой диаграмме взаимодействия есть объект, ответственный за управление последовательностью событий сценария. Все диаграммы взаимодействия для некоторого варианта использования могут иметь один и тот же управляющий объект.

Управляющий объект не реализует никаких бизнесов-процессов, он лишь посылает сообщение другим объектам. Управляющий объект отвечает за координацию действий других объектов и за делегирование ответственности. По этой причине такие объекты называют еще *объектами-менеджерами*.

Преимущество использования управляющего объекта состоит в обособлении бизнес-логики от логики, которая предопределяет последовательность событий. Если надо будет изменить последовательность, это затронет только управляющий объект.

На диаграмму можно нанести и другие объекты, которые отвечают за безопасность, обработку ошибок или за связь с базой данных. Многие из них бывают настолько общими, что допускают повторное использование в других применениях.

Предположим, нужно сохранить в базе данных информацию о новом материале – станок фрезерный.

Первый вариант – объект "станок фрезерный" снабжен необходимой информацией о базе данных, в этом случае он сохраняет себя сам.

Второй вариант – объект целиком отделен от логики базы данных и тогда его сохранением должен заниматься другой объект, который можно назвать менеджером *транзакции* (`transaction manager`).

Преимуществом второго варианта является возможность повторного использования объекта в другом применении с другой базой данных или вообще без базы данных. Кроме того, при изменении требований уменьшаются изменения в программе. Изменения в базе данных не повлияют на логику применения, и наоборот. Однако все эти детали не интересуют клиента, но неоценимые для разработчиков.

После того как все объекты созданы, их соотносят с классами. Объекты можно связать с существующими классами или создать для них новые. Потом нужно соотнести сообщение диаграммы с операциями. В конце концов, надо перейти к спецификациям самых объектов и сообщений и задать такие детали, как стойкость объекта, синхронизация и частота сообщений.

8 РАЗРАБОТКА ДИАГРАММЫ КЛАССОВ

8.1 Представление класса в диаграмме классов

Диаграмма классов (Class diagram) показывает коллекцию декларативных (статических) элементов модели, а также их содержание и отношения. По существу диаграмма классов показывает статическую структуру системы.

Построение диаграмм классов – это важнейший и трудоемкий этап в создании модели. Каждый класс имеет набор методов (Operations) и переменных (Attributes).

На диаграммах классов отображаются статические картины фрагментов системы и связей между ними. Обычно для описания системы создают несколько диаграмм классов. На одних показывают некоторое подмножество классов и отношения между классами подмножества. На других отображают одно и то же подмножество, но вместе с атрибутами и операциями классов. Третьи отвечают только пакетам классов и отношениям между ними. Для изображения полной картины системы можно разработать столько диаграмм классов, сколько нужно.

При наличии большого числа классов они могут объединяться в пакеты (packages). Объединять классы можно как угодно, однако существует несколько наиболее распространенных подходов.

Во-первых, можно группировать классы по стереотипу. В таком случае выходит один пакет с классами-сущностями, один с пограничными классами, один с управляющими классами и т.д. Этот подход может быть полезным с точки зрения размещения готовой системы, поскольку все пограничные классы, которые находятся на клиентских машинах, уже оказываются в одном пакете.

Второй подход заключается в объединении классов за их функциональностью. Например, в пакете **Security** (безопасность) будут содержаться все классы, которые отвечают за безопасность применения.

Преимущество этого метода заключается в возможности повторного использования пакетов. Если внимательно подойти к группированию классов, можно получить практически не зависимые один от другого пакеты. Например, пакет Security можно использовать и в других применениях.

В конце концов, применяют комбинацию двух указанных подходов. Для дальнейшей организации классов можно вкладывать пакеты один в один. На высоком уровне, например, можно сгруппировать классы за функциональностью, создав пакет **`Security`**. Внутри него можно создать другие пакеты, сгруппировав классы, которые отвечают за безопасность, за функциональностью или по стереотипу.

Диаграммы классов является хорошим инструментом проектирования. С их помощью разработчики могут видеть и планировать структуру системы еще до фактического написания кода, благодаря чему они в самом начале могут понять, хорошо ли спроектирована система.

Вершина в диаграмме классов – класс. Пиктограмма класса включает три секции (рис. 8.1).

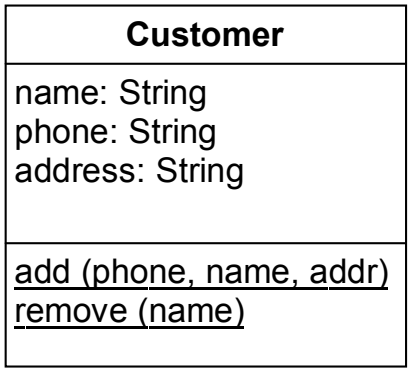

Рисунок 8.1 – Пример представления класса на диаграмме классов

В вершине указываются имя класса (верхняя секция), свойства (средняя секция) и операции (нижняя секция). Если свойство или операция подчеркивается, то областью действия является класс, в противном случае

областью действия является экземпляр класса, то есть у каждого экземпляра имеется свое значение свойства или выполняется своя операция.

Пустота секции не означает, что у класса отсутствуют свойства или операции. Для явного определения большого количества свойств или атрибутов в конце списка проставляется три точки. Длинные списки можно группировать – каждая группа начинается со своего стереотипа.

В UML существует несколько разновидностей класса: интерфейс, утилита и др. Все они похожие, но есть и отличия: интерфейс – это класс, в котором определены только операции, а утилита – это класс, все атрибуты и операции которого общедоступны (public). Для указания вида класса в UML введено понятие стереотипа (stereotype). Соответственно, классы-интерфейсы имеют стереотип "**interface**", а классы-утилиты – "**utility**". Существует стандартный набор стереотипов (параметризованный класс), которые, при необходимости, можно расширять.

Выявление класса позволяет найти границы предметной области. Кроме того, эта процедура представляет собой первый шаг в продумывании объектно-ориентированной декомпозиции разрабатываемой системы. Результатом этого этапа является справочник данных, который обновляется по мере своего развития. Все классы должны быть осмыслены в рассматриваемой прикладной области; классов, связанных с компьютерной реализацией, как, например, список, стэк и т. п. на этом этапе вводить не следует.

Начать нужно с выделения возможных классов из письменной постановки прикладной задачи (технического задания и другой документации, предоставленной заказчиком). При определении возможных классов нужно постараться выделить как можно больше классов, выписывая имя каждого класса, который приходит на ум. В частности, каждому существительному, встречающемуся в предварительной постановке задачи, может соответствовать класс. Поэтому при выделении возможных классов каждому такому существительному обычно сопоставляется возможный класс.

Далее список возможных классов должен быть проанализирован с целью исключения из него ненужных классов. Из списка удаляются:

- *избыточные классы*: если два или несколько классов выражают одинаковую информацию, следует сохранить только один из них;
- *нерелевантные классы* (не имеющие прямого отношения к проблеме): для каждого имени возможного класса оценивается, насколько он необходим в будущей системе;
- *нечетко определенные классы* с точки зрения рассматриваемой проблемы;
- *атрибуты*: некоторым существительным больше соответствуют не классы, а атрибуты, например: имя, возраст, вес, адрес и т. п.;
- *операции*: некоторым существительным больше соответствуют не классы, а имена операций, например, телефонный вызов вряд ли означает какой-либо класс;
- *роли*: некоторые существительные определяют имена ролей в объектной модели, например, владелец, водитель, начальник, служащий;
- *реализационные конструкции*: имена, которые больше связаны с программированием и компьютерной аппаратурой (подпрограмма, процесс, алгоритм, прерывание и т. п.), не следует на данном этапе сопоставлять с классом, так как они не отражают особенностей проектируемой прикладной системы.

После исключения имен всех ненужных (лишних) классов будет получен предварительный список классов, составляющих проектируемую систему.

Каждому классу на диаграмме необходимо дать уникальное имя. Как правило, большинство организаций разработчиков имеют уже постоянные, собственные соглашения в наименовании классов. В общем случае используются существительные в единственном числе. Обычно имена классов не содержат пробелов. Это делается по практическим причинам и из пониманий чтения – языки программирования, как правило, не поддерживают пробелов в

именах классов. Имена классов по возможности должны быть относительно короткими.

Количество экземпляров класса называется его множественностью. Выражение множественности записывается в правом верхнем углу значка класса. Так, например, **КонтроллерУглов** (рис. 8.2) – это единичный экземпляр класса-(singleton), а для класса **ДатчикУгла** разрешены три экземпляра.

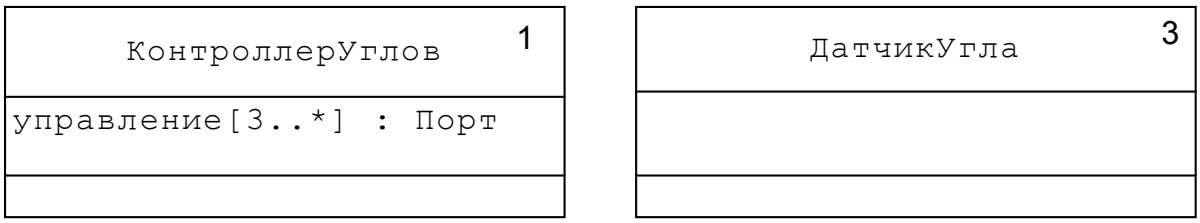

Рисунок 8.2 – Примеры задания множественности

8.2 Отличительные особенности классов

В UML определены абстрактные, базовые, параметризованные и конкретные классы.

Абстрактный класс – это класс, для которого не определены экземпляры объектов. Поведение этого класса дополняется и уточняется в подклассах.

Базовый класс – это наиболее общий класс объектов в какой-нибудь структуре классификации. Базовые классы часто используют в практических приложениях.

Параметризованный класс – это базовый класс, который служит основой для образования других классов путем замены параметров на значения. Только после “наполнения” параметров их значениями этот класс становится пригодным для создания объектов. Используется как “обобщенный класс”.

Конкретный класс – класс, для которого можно, непосредственно создать экземпляры объектов.

Поддержку класса обеспечивают утилиты класса.

Утилита класса (class utility) – это совокупность общедоступных процедур (операций). Например, в системе может быть совокупность математических функций, которые используются всей системой и не слишком подходят для какого-нибудь конкретного класса. Эти функции можно собрать вместе и объединить в утилиту класса. Утилиты классов часто применяются для расширения функциональных возможностей языка программирования или для многоразового использования в нескольких системах.

Для расширения словаря UML и создания новых видов строительных блоков, производных от существующих, но специфичных для конкретной задачи, применяются стереотипы классов. Так, например, можно создать стереотип **Form** (Форма) и назначить его всем окнам, используемым в программе. Такой стереотип можно применить в других программах.

Языком UML определены три основных стереотипа – ***граница*** (Boundary), ***объект*** (Entity) и ***управление*** (Control). Соответствующие этим стереотипам классы называются: ***пограничным классом*** (boundary classes), ***класс-сущностью*** (entity classes) и ***управляющим классом*** (control classes). Графические обозначения этих классов приведены на рисунке 8.3.

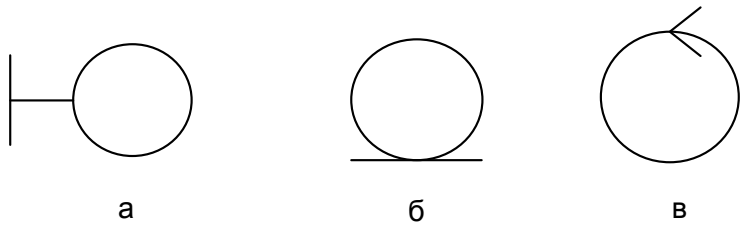

Рисунок 8.3 – Обозначения пограничного класса (а), класса-сущности (б) и управляющего класса (в)

Пограничный класс – это класс, который расположен на границе системы со всем другим миром. Он содержит в себе формы, отчеты, интерфейсы с аппаратурой (принтерами, сканерами) и интерфейсы с другими системами.

Для выявления пограничных классов необходимо исследовать диаграммы вариантов использования. Для каждого взаимодействия между действующим лицом и вариантом использования должен существовать хотя бы один пограничный класс. Именно он разрешает действующему лицу взаимодействовать с системой. Следует принять во внимание, что необязательно создавать пограничные классы для каждой пары "действующее лицо-вариант использования".

Класс-сущности содержат информацию, которая сохраняется постоянно. Класс-сущности можно найти в потоке событий и на диаграммах взаимодействия. Они имеют наибольшее значение для пользователя, и потому в их названиях часто применяют термины из предметной области.

Часто для каждого класса-сущности создают таблицу в базе данных. В этом заключается еще одно отличие от типичного подхода к конструированию системы. Вместо того, чтобы с самого начала задавать структуру базы данных, существует возможность разрабатывать ее на основе информации, получаемой из объектной модели. Это позволяет отслеживать соответствие всех полей базы данных и требований к системе. Требования определяют поток событий. Поток событий определяет объекты, классы и атрибуты классов. Каждый атрибут класса-сущности становится полем в базе данных. Применяя такой подход, легко отслеживать соответствие между полями базы данных и требованиями к системе, которая уменьшает вероятность сбора ненужной информации.

Управляющие классы создаются для координации действий других классов. Обычно у каждого варианта использования есть один управляющий класс, который контролирует последовательность событий этого варианта использования. Управляющий класс не несет в себе никакой функциональности – другие классы посылают ему мало сообщений. Но сам он посылает множество

сообщений. Управляющий класс делегирует ответственность другим классам. По этой причине управляющий класс часто называют классом-менеджером.

В системе могут применяться и другие управляющие классы, общие для нескольких вариантов использования. Например, класс **SecurityManager** (Менеджер безопасности) может отвечать за контроль событий, связанных с безопасностью, а класс **TransactionManager** (Менеджер транзакций) – заниматься координацией сообщений, которые относятся к транзакциям с базой данных. Могут быть и другие менеджеры для работы с такими элементами функционирования системы, как разделение ресурсов, распределенная обработка данных и обработка ошибок.

С помощью управляющих классов можно изолировать функциональность системы. Инкапсуляция в один класс, например, координации безопасности, минимизирует следствия внесенных изменений. Любые изменения логики, которые отвечают за безопасность системы, затронут только менеджера безопасности.

Кроме упомянутых выше стереотипов можно создавать и свои собственные.

8.3 Работа с атрибутами в диаграмме классов

Атрибут – поименованное свойство классификатора, описывающее диапазон значений, которые могут принимать экземпляры этого свойства.

Существует множество источников, где можно найти атрибуты. Важнейшим источником является описание варианта использования, в частности, имена существительные в потоке событий. Некоторые из них будут классами или объектами, другие – действующими лицами, и, в конце концов, последняя группа – атрибутами. Например, в потоке событий может быть написано: "Пользователь вводит имя сотрудника, его адрес, номер социальной страховки и номер телефона". Это означает, что в классе **Сотрудник** есть атрибуты: **Имя, Адрес, Номер страховки** и **Номер телефона**.

Атрибуты можно также проявить, изучая документацию, которая описывает требования к системе. Следует также изучить структуру базы данных. Если она уже определена, поля в ее таблицах дадут представление об атрибутах.

Часто есть однозначное соответствие между таблицами базы данных и классами-сущностями. Если возвратиться к предыдущему примеру, то в таблице **Сотрудник** должны быть поля **Имя, Адрес, Номер телефона** и **Номер страховки**, которые соответствую атрибутам класса **Сотрудник.** Следует отметить, однако, что не всегда существует такое однозначное соответствие. Подходы к проектированию баз данных и классов могут различаться. В частности, реляционные базы данных не поддерживают наследования непосредственно.

Очерчивая атрибуты, необходимо следить за тем, чтобы каждый из них можно было соотнести с требованиями к системе. Это помогает решить классическую проблему применения, которое собирает огромный объем никому не нужной информации. Каждое требование должно быть отработано применительно к конкретному потоку событий варианта использования или относительно существующей таблицы базы данных. Если это не удается сделать, нельзя быть уверенным в том, что данное требование нужно заказчику. Система и база данных должны создаваться одновременно. В этом заключен один из принципов современного подхода к программированию.

Итак, атрибут должен отвечать классу. Например, в классе **Сотрудник** конкретный сотрудник может носить имя и адрес, но не должен включать в себя сведения о продукции, которая выпускается на предприятии. Для таких сведений подошел бы класс **Продукция**.

Если класс содержит слишком много атрибутов, то возможно, такой класс следует разделить на два меньших. Так, класс, имеющий 10-15 атрибутов, может быть целиком приемлемым. Важно, чтобы все его атрибуты были нужны и действительно принадлежали этому классу. Следует также обратить внимание на классы, которые имеют слишком мало атрибутов. Вполне возможно, что все

нормально, например, управляющий класс имеет мало атрибутов. Однако это может быть и признаком необходимости объединения нескольких классов.

Все атрибуты должны иметь спецификацию. Для атрибута это: тип данных, значение по умолчанию, стереотип и видимость.

Одной из главных характеристик атрибута есть его тип данных. Он специфический для языка, который используется. Это может быть, например, тип **`string`** (ряд), **`integer`** (целое число), **`long`** (длинный) или **`boolean`** (булевый). Перед началом генерации кода необходимо указать тип данных каждого атрибута. Для указания типа даннях можно использовать типы данных языка программирования или определенные в модели имена классов.

У атрибутов могут быть стереотипы. Стереотип атрибута является способом его классификации. Например, некоторые атрибуты могут отвечать полям базы данных, а другие нет. Для каждого такого типа можно определить свой стереотип.

Атрибуты могут иметь значения по умолчанию. Значение по умолчанию задавать не обязательно, однако это значение используется при генерации кода для инициализации атрибута.

Представление атрибута (свойства) осуществляется следующим синтаксисом:

`Видимость Имя [Множественность]: Тип = НачальноеЗначение {Характеристики}`

Так как атрибуты содержатся внутри класса, они скрыты от других классов. В связи с этим нужно указать, какие классы имеют право читать и изменять атрибуты. Это свойство называется ***видимостью атрибута*** (`attribute visibility`).

Допустимы четыре значения этого параметра:

- ***public*** **(**общий, открытый**)**. Атрибут видим всем другим классам. Любой класс может просмотреть или изменить значение атрибута. В нотации `UML` общему атрибуту соответствует знак "+";

- **private (**закрытый, секретный**)**. Атрибут не видим никаким другим классам. В нотации UML закрытый атрибут обозначается знаком "–".
- **protected (**защищенный**)**. Атрибут доступный только самому классу и его преемникам. Нотация UML для защищенного атрибута – знак "#".
- **package or Implementation (**пакетный**)**. Атрибут есть общим, но только в пределах своего пакета. Данный тип видимости не обозначается никаким специальным значком.

В общем случае атрибуты рекомендуется делать закрытыми или защищенными. Это позволяет лучше контролировать сам атрибут и код. При использовании закрытых или защищенных атрибутов удается избежать ситуации, когда значение атрибута изменяется всеми классами системы. Вместо этого логика изменения атрибута будет содержаться в том самом классе, который и сам атрибут. Параметры видимости, которые задаются, влияют на скелетный код, который генерируется.

В некоторых случаях в классы включаются атрибуты или методы, которые относятся ко всему классу в целом, а не к отдельному экземпляру объекта этого класса. Для указания этого атрибут надо сделать статическим (static). В нотации UML статический атрибут обозначается символом "$".

Если атрибут создается с одного или нескольких других атрибутов, то его называют производным (derived). Например, класс **Прямоугольник** может иметь атрибуты **Ширину** и **Высоту**. Также у него может быть атрибут **Площадь**, которая исчисляется как произведение ширины и высоты. Так как атрибут **Площадь** выходит из двух других атрибутов, то он считается производным. В UML производные атрибуты обозначают символом "/".

Установленные для атрибута начальные значения могут быть изменены. Порядок такого изменения указывается в характеристике атрибута (свойства).

В языке UML определены три ***характеристики свойств***:

Chenqeable	Нет ограничения на модификацию значения свойства

addOnly	Для свойств множественностью, большей единицы; дополнительные значения могут быть добавлены, но после создания значений характеристика не может удаляться или изменяться
frozen	После инициализации объекта значение свойства не изменяется

Примеры объявления атрибутов (свойств):

начало	Только имя
+начало	Публичная видимость и имя
начало: Координаты	Имя : тип
имяФамилия [0…1] : String	Имя, множественность : тип (ряд имен)
левыйУгол : Координаты = (0,10)	Имя : тип = (начальное значение)
сумма : Integer {frozen}	Имя : тип, характеристика

8.4 Работа с операциями

Операция – это связанное с классом поведение. Операции определяют ответственности классов. При идентификации операций и анализе классов необходимо:

- Относиться с подозрением к любому классу, который имеет только одну или две операции. Возможно, класс написан совсем правильно, но его следует объединить с каким-нибудь другим классом.
- С наибольшим подозрением относиться к классу без операций. Как правило, класс инкапсулирует не только данные, но и поведение. Класс без поведения лучше моделировать как один или несколько атрибутов.
- С осторожностью относиться к классу со слишком большим числом операций. Набор ответственностей класса должен быть управляем. Если класс очень большой, им будет тяжело руководить. В такой ситуации лучше разделить класс на два меньших.

Общий синтаксис представления ***операции*** имеет вид:

Видимость Имя (Список Параметров): ВозвращаемыйТип {Характеристики}

Видимость операции устанавливается так же, как и для атрибутов.

Имя операции определяется её типом. Обычно применяются четыре типа операций.

Операции реализации (implementor operations) реализуют некоторую бизнес-функциональность. Такие операции можно найти, исследуя диаграммы взаимодействия. Диаграммы этого типа фокусируются на бизнес-функциональности и каждое сообщение диаграммы, скорее всего, можно соотнести с операцией реализации. Необходимо, чтобы каждую операцию реализации можно было проследить до соответствующего требования. Это достигается на разных этапах моделирования. Операция выводится из сообщения на диаграмме взаимодействия, сообщение выделяются из подробного описания потока событий, который создается на основе варианта использования, а последний – на основе требований. Возможность проследить всю эту цепочку гарантирует, что каждое требование будет воплощено в коде, а каждый фрагмент кода реализует какое-то требование.

Операции управления (manager operations) руководят созданием и разрушением объектов. К этой категории относятся конструкторы и деструкторы классов.

Операции доступа (access operations). Атрибуты обычно бывают закрытыми или защищенными. Тем не менее, другие классы иногда должны пересматривать или изменять их значение. Пусть, например, есть атрибут **Salary** (Заработок) класса **Employee** (Служащий), в отношении которого нежелательно, чтобы другие классы могли его изменять. В то же время следует предусмотреть контролируемый доступ к этому атрибуту. С этой целью к классу **Employee** прибавляются две операции доступа – **GetSalary** и **SetSalary**. Теперь другие классы могут

обращаться к общей операции **GetSalary**, которая получает значение атрибута **Salary** и возвращает его вызвавшему классу. Операция **SetSalary** также является общей, она помогает классу, который вызвал ее, установить новое значение атрибута **Salary**, если выполнены необходимые правила и условия. Такой подход дает возможность безопасно инкапсулировать атрибуты внутри класса, защитив их от других классов, но все-таки разрешает осуществлять контролируемый доступ.

Создание операций **Get** и **Set** (получение и изменения значения для каждого атрибута класса) является промышленным стандартом.

Вспомогательными (`helper operations`) называются такие операции класса, которые необходимы ему для выполнения его ответственностей, но о каких другие классы не должны ничего знать. Это закрытые и защищенные операции класса. Как и операции реализации, вспомогательные операции можно найти на диаграммах последовательности и кооперативных диаграмм. Часто такие операции являются рефлексивными сообщениями.

Для идентификации операций необходимо выполнить такие действия:

1. Изучить сообщения на диаграммах последовательности. Большая часть сообщений на этих диаграммах представляет собой операции реализации. Рефлексивные сообщения будут вспомогательными операциями.
2. Рассмотреть управляющие операции. Возможно, нужно прибавить конструкторы и деструкторы.
3. Рассмотреть операции доступа. Для каждого атрибута класса, с которым будут работать другие классы, необходимо создать операции **Get** и **Set**.

Параметры – это аргументы, получаемые операцией "на входе". Тип значения, которое возвращается, относится к результату действия операции.

В сигнатуре операции можно указать ноль или более параметров по следующей форме:

Направление Имя : Тип = ЗначениеПоУмолчанию

Элемент **Направление** может принимать одно из следующих значений:

In	Входной параметр, не может модифицироваться;
out	Выходной параметр, может модифицироваться для передачи информации в вызывающий объект;
inout	Входной параметр, может модифицироваться.

Допустимо применение следующих характеристик операций:

leaf	Конечная операция, не может переопределяться (находится в цепочке наследования);
isQuery	Выполнение операции не изменяет состояние;
sequential	В каждый момент времени в объект поступает только один вызов операций, то есть допустим только один поток вызовов, поток управления;
quarded	Допускается поступление нескольких вызовов, которые ставятся в очередь;
concurrent	В объект поступает несколько потоков вызовов операций, разрешается множественное выполнение операций, подразумевается, что операции являются атомарными.

Примеры объявления операций:

записать	Только имя;
+записать	Видимость и имя;
Зарегистрировать(и: Имя, ф: Фамилия)	Имя и параметры;
балансСчета ( ) : Integer	Имя и возвращаемый тип;
нагревать ( ) {quarded}	Имя и характеристика.

На некоторых диаграммах полезно показывать полную сигнатуру операций. Если же нужно упростить диаграмму, лучше оставить только названия.

8.5 Работа со связями

Для того, чтобы один класс мог послать сообщение другому классу, между ними должна существовать связь. Связи или отношения, используемые в диаграммах классов, включают в себя ассоциации, обобщения, зависимости, реализации, агрегации и композиции (рис. 8.4).

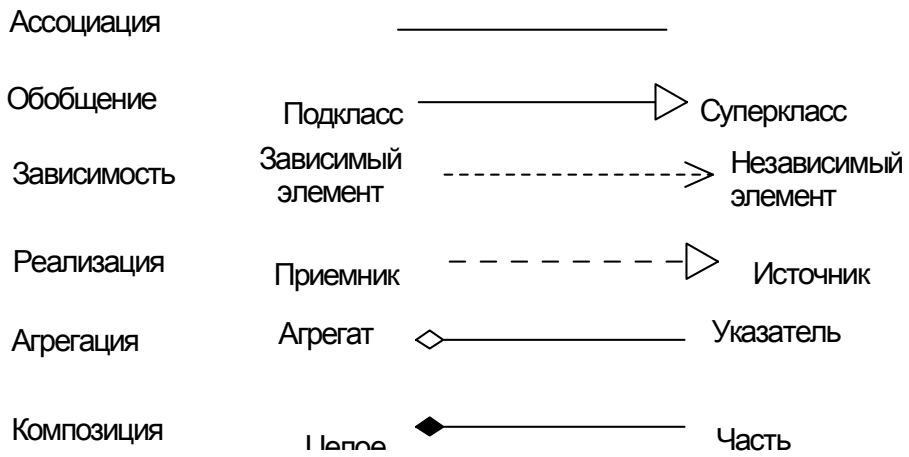

Рисунок 8.4 – Отношения в диаграммах классов

Ассоциации отображают структурные отношения, то есть *соединения* между объектами. Каждая ассоциация может иметь метку – имя, которое описывает природу отношения. Имени можно придать направление (рис. 7.5).

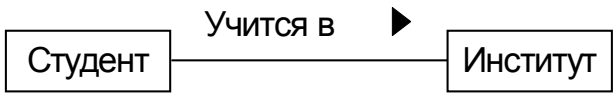

Рисунок 8.5 Пример имени ассоциации

Роль ассоциации – определить, каким представляется класс на одном конце ассоциации для класса на противоположном конце.

Чтобы учесть, сколько объектов может присоединяться через заданный экземпляр ассоциации, пользуются заданием *мощности* роли ассоциации. Так, например, можно применять следующие варианты задания мощности:

- 5 – точно пять;
- * – неограниченное количество;
- 1..* – один и более;
- 3..7 – определенный диапазон.

Вариант указания мощности приведен на рисунке 8.6.

Рисунок 8.6 – Пример указания мощности ассоциации

Ассоциации могут иметь свойства, тогда они отображаются с помощью классов ассоциаций (рис. 8.7).

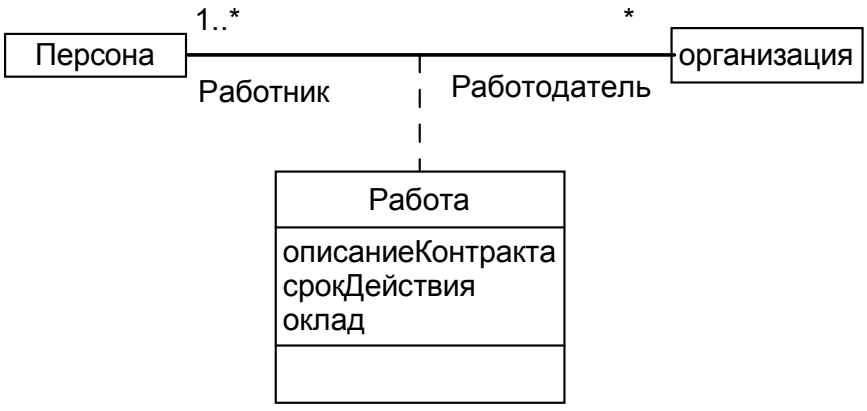

Рисунок 8.7 – Пример добавления класс-ассоциации

Обобщение – это отношение между общим предметом (суперклассом) и специализированной разновидностью этого предмета (подклассом). Подкласс

может иметь одного родителя (один суперкласс) или нескольких родителей, то есть иметь множественное наследование.

Зависимость является отношением использования между клиентом (зависимым элементом) и поставщиком (независимым элементом). Например, при реализации класса **Заказ**, в котором предусмотрены операции проверки наличия необходимого оборудования и оформления заказа, может оказаться, что нужного комплекта нет.

Агрегация и композиция считаются разновидностями отображения структурных отношений между целым (агрегатом) и частью. Агрегация показывает отношение по ссылке (в агрегат включены только указатели), а композиция – отношение физического включения (в агрегат включены сами части).

Реализация – это семантическое отношение между классами, в котором класс-приемник выполняет реализацию класса-источника. Например, при работе с каталогом интерфейс класса **Обработчик каталога** позволяет клиентам взаимодействовать с объектами класса **Каталог** без знания дисциплины доступа.

Для выявления связей необходимо изучить созданные к этому моменту элементы модели:

1. Следует начать из изучения диаграмм последовательности. Если класс **A** посылает сообщение классу **B**, между этими классами должна быть установлена связь. Как правило, – это ассоциации или зависимости.
2. Далее необходимо проверить классы на предмет наличия связей типа ***целое-часть***. Любой класс, который состоит из других классов, может принимать участие в связях агрегации.
3. Все классы, в которых есть несколько типов или вариантов, имеют связи обобщения. Например, имеется класс **Employee** (Сотрудник). Есть два типа сотрудников – с почасовой оплатой и с окладом. Это означает, что нужно создать классы **HourlyEmp** и **SalariedEmp**, которые наследуют класс **Employee**. Общие для всех сотрудников

атрибуты, операции и связи следует поместить в класс **Employee**. Атрибуты, операции и связи, уникальные для сотрудников какого-то типа, необходимо поместить в классы **HourlyEmp** и **SalariedEmp**.

Одной из особенностей хорошо спроектированного применения есть сравнительно небольшое количество связей в системе. Класс, у которого много связей, должен знать о большом числе других классов системы. В результате его тяжело будет использовать. Кроме того, сложно будет вносить изменения в готовый программный продукт. Если изменится какой-либо из классов, это может повлиять на многие классы.

8.6 Пример диаграммы классов

Пример диаграммы классов системы управления приводами подачи металлорежущего станка приведен на рисунке 8.8.

На диаграмме представлен активный класс **УправлПрограмма**, который имеет атрибут (свойство) **траекторияОбработкиДетали** и операции: **выполнятьПрограмму ( ), анализСостояния ( ), прогнозОкончУправления ( ).**

Имеется ассоциация между этим классом и классом **Контроллер СУ** в виде клиент-серверного отношения – экземпляры программы задают параметры движения, которые должны обеспечивать экземпляры контроллера.

Классу **Контроллер СУ** задано ограничение на множественность – он имеет три экземпляра, каждый из которых включает по одному экземпляру классов **Регулятор скорости** и **Регулятор тока**, а также по девять экземпляров класса **Датчик**.

Экземпляры **Регулятор скорости** и **Регулятор тока** включены в агрегат **Контроллер СУ** физически – здесь присутствует отношение *композиция*. Экземпляры **Датчик** включены по ссылке (агрегация), то есть

экземпляр **Контроллер СУ** имеет лишь указатели (адреса или регистры) на объекты-датчики.

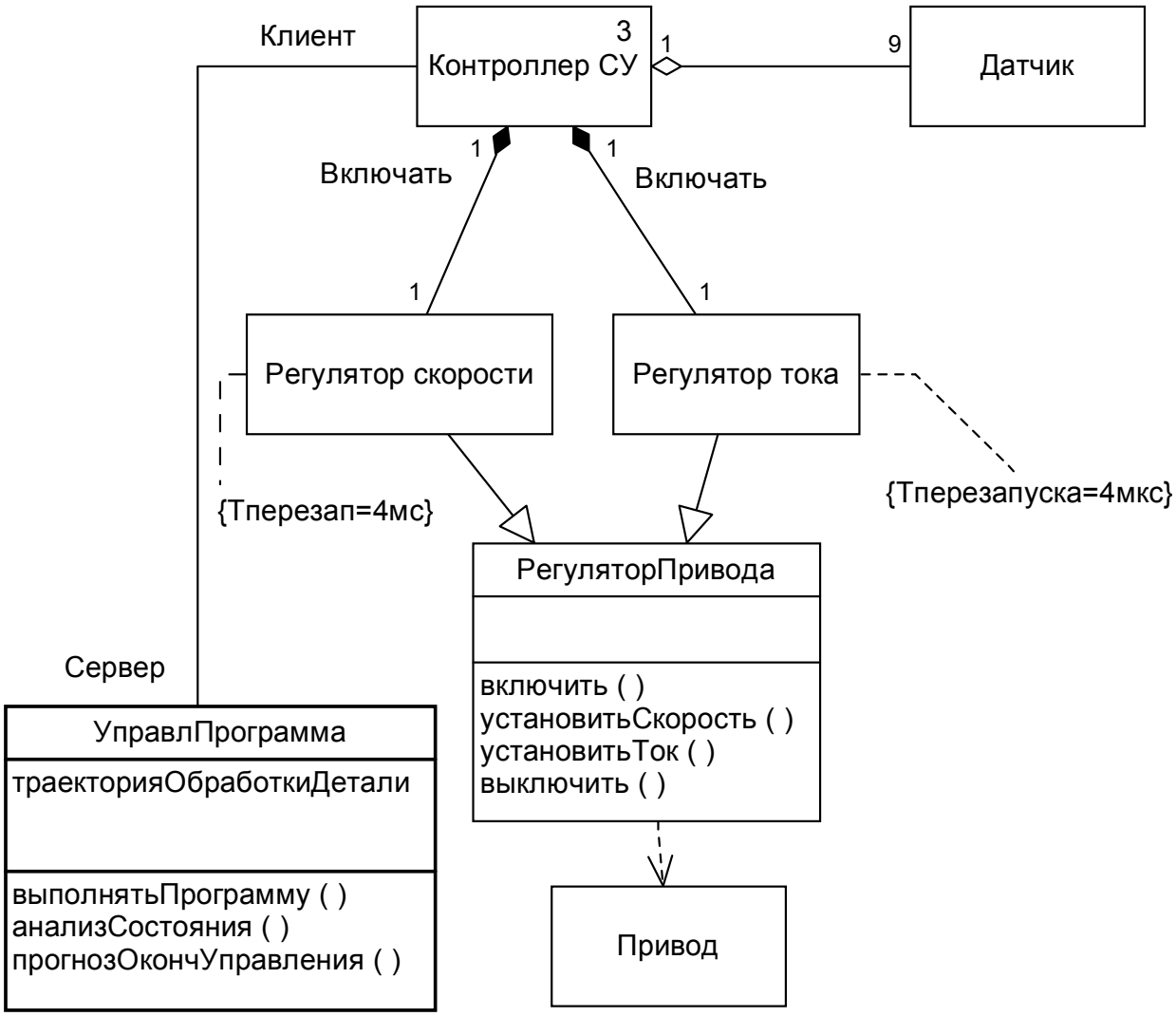

Рисунок 8.8 – Диаграмма классов системы управления приводами

Регулятор скорости и **Регулятор тока** – это подклассы абстрактного суперкласса **РегуляторПривода**, который передает им в наследство абстрактные операции **включить ( )** и **выключить ( )**. В свою очередь, класс **РегуляторПривода** использует конкретный класс **Привод**. Для класса **Регулятор тока** задано ограничение времени на повторное включение 4 мкс, для класса **Регулятор скорости** – на 4 мс.

На диаграмме классов объекты не существуют изолированно друг от друга – они либо подвергаются воздействию, либо сами воздействуют на другие объекты. При этом поведение объекта является функцией, как его состояния, так и выполняемых им операций.

Операции обозначают обслуживание, которое объект предлагает своим клиентам. Возможны пять видов операций ***клиента над объектом***:

1) ***модификатор*** – по этой операции изменяется состояние объекта, например, вес;
2) ***селектор*** – объект разрешает доступ к состоянию, но не изменяет его, например, предоставляет информацию;
3) ***итератор*** – содержание объекта доступно в определенном порядке, например, по номенклатуре;
4) ***конструктор*** – операцией создается объект и инициализируется его состояние, например, включается объект, обеспечивающий смену инструмента;
5) ***деструктор*** – операция разрушения объекта и освобождения памяти.

В приведенном на рисунке 8.8 примере операция **`выполнятьПрограмму ( )`** является модификатором, **`анализСостояния ( )`** – итератором, а **`прогнозОкончУправления ( )`** – селектором.

Литература

Основная

1. Брауде Э. Технология разработки программного обеспечения. – СПб.: Питер, 2004.- 655 с.
2. Буч Г. Объектно-ориентированное проектирование с примерами применения: Пер. с англ. – М.: Конкорд, 1992. – 519 с.
3. Буч Г., Рамбо Дж., Джекобсон А. Язык UML. Руководство пользователя: Пер. с англ. – М.: ДМК, 2000.
4. Иванова Г. С. Технология программирования: Учебник для вузов. – 2-е изд., стереотип. – М.: Изд-во МГТУ им. Н.Э. Баумана, 2003. – 320 с.
5. Орлов С. А. Технология разработки программного обеспечения: Учебник для вузов. 3-е изд. – СПб.: Питер, 2004. – 527 с.

Дополнительная

6. Леоненков А. В. Самоучитель UML – Санкт-Петербург: BHV, 2001.
7. Леоненков А. В. Объектно-ориентированный анализ и проектирование с использованием UML и IBM Rational Rose: Учебное пособие / А.В. Леоненков. – М.:Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2006. – 320 с.
8. Розенберг Д., Скотт К. Применение объектного моделирования с использованием UML и анализ прецедентов: Пер. с англ. – М.: ДМК Пресс, 2002. – 160 с.
9. Спицнадель В. Н. Основы системного анализа: Учеб. Пособие. – СПб.: Изд. дом ”Бизнес-пресса”, 2000 г. – 326 с.
10. Бенькович Е.С., Колесов Ю.Б., Сениченков Ю.Б. Практическое моделирование динамических систем – СПб.: БХВ-Петербург, 2002. – 464 с.

Глоссарий

OCL (Object Constraint Language) - язык ограничений объектов. Формальный язык для выражения ограничений без побочных эффектов.

UML (Unified Modeling Language) - Унифицированный язык моделирования, предназначенный для визуализации, специфицирования, конструирования и документирования артефактов программных систем.

Абстрактный класс - класс, для которого нельзя непосредственно создать экземпляры объектов.

Абстракция - важная характеристика сущности, отличающая ее от всех иных сущностей. Абстракция проводит границу между сущностями лишь с какой-то определенной точки зрения.

Автомат - поведение, которое специфицирует последовательность состояний, через которые проходит объект на протяжении своего жизненного цикла, реагируя на события, включая описание реакций на эти события.

Агрегат - класс, представляющий "целое" в отношении агрегирования.

Агрегирование - специальный вид ассоциации, описывающий отношение между агрегатом (целым) и компонентом (частью).

Актер - множество логически связанных ролей, исполняемых при взаимодействии с Прецедентами.

Активация - выполнение операции.

Активный класс - класс, экземплярами которого являются активные объекты.

Активный объект - объект, который владеет процессом или нитью и может инициировать управляющее воздействие.

Аргумент - фактическое значение, соответствующее формальному параметру.

Артефакт - элемент информации, используемый или порождаемый в процессе разработки программного обеспечения.

Асинхронное действие - запрос, при котором посылающий объект не дожидается получения результата.

Ассоциация - структурное отношение, описывающее набор связей, в котором

каждая из них представляет собой соединение между объектами; семантическое отношение между двумя или более классификаторами, в котором участвуют соединения между их экземплярами.

Атрибут - Поименованное свойство классификатора, описывающее диапазон значений, которые могут принимать экземпляры этого свойства.

Версия - относительно полный и самосогласованный набор артефактов, предназначенный для внутреннего или внешнего использования.

Взаимодействие - поведение, описываемое набором сообщений, которыми обмениваются между собой объекты в некотором контексте для достижения определенной цели.

Видимость - указывает, при каких обстоятельствах то или иное имя видимо и может быть использовано.

Внедрение - четвертая фаза цикла разработки программного обеспечения, в течение которой оно передается пользователям.

Временное выражение - выражение, результатом вычисления которого является абсолютный или относительный момент времени.

Временное ограничение - семантическое утверждение об абсолютном или относительном значении времени или временного интервала.

Временный объект - объект, который существует только до тех пор, пока выполняется создавший его процесс или нить.

Выражение действия - выражение, значением которого является набор действий.

Действие - выполнимое атомарное вычисление, которое приводит к изменению состояния системы или возврату значения.

Делегирование - способность объекта посылать сообщение другому объекту в ответ на получение сообщения.

Диаграмма - графическое представление множества элементов. Обычно изображается в виде графа с вершинами (сущностями) и ребрами (отношениями).

Диаграмма взаимодействия - диаграмма, на которой представлено

взаимодействие, состоящее из множества объектов и отношений между ними, включая и сообщения, которыми они обмениваются. Диаграммы взаимодействия относятся к динамическому виду системы. Этот обобщенный термин применяется к нескольким видам диаграмм, в которых делается акцент на взаимодействии объектов, в том числе к Диаграммам кооперации, последовательности и деятельности.

Диаграмма деятельности - диаграмма, на которой представлены переходы потока управления от одной деятельности к другой. Диаграммы деятельности относятся к динамическому аспекту поведения системы. Это разновидность диаграмм состояний, где все или большая часть состояний являются состояниями деятельности а все или большая часть переходов срабатывают при завершении деятельности в исходном состоянии.

Диаграмма классов - диаграмма, на которой представлено множество классов, интерфейсов, кооперацій и отношений между ними; диаграммы классов относятся к статическому виду системы. Иными словами, это диаграмма, на которой показано множество декларативных (статических) элементов.

Диаграмма компонентов - диаграмма, на которой изображена организация некоторого множества компонентов и зависимости между ними; относится к статическому виду системы.

Диаграмма кооперации - диаграмма взаимодействий, в которой основной акцент сделан на структурной организации объектов, посылающих и получающих сообщения. На этой диаграмме изображено, как организованы взаимодействия между экземплярами и какие между ними существуют связи.

Диаграмма объектов - диаграмма, на которой представлено множество объектов и отношений между ними в некоторый момент времени. Диаграммы объектов относятся к статическому виду системы с точки зрения проектирования или процессов.

Диаграмма последовательностей - диаграмма взаимодействия, в которой основной акцент сделан на временном упорядочении сообщений.

Диаграмма прецедентов - диаграмма, на которой представлено множество

прецедентов и актеров, а также отношения между ними. Диаграммы прецедентов относятся к статическому виду системы.

Диаграмма развертывания - диаграмма, на которой представлена конфигурация обрабатывающих узлов и размещенные на них компоненты; относится к статическому виду системы.

Диаграмма состояний - диаграмма, на которой изображен автомат; диаграммы состояний относятся к динамическому виду системы.

Динамическая классификация - семантическая разновидность обобщения, при которой объект может изменять тип или роль.

Динамический вид- аспект системы, в котором основное внимание уделено ее поведению.

Дополнение - деталь элемента спецификации, добавляемая к его базовому графическому символу.

Дорожка - разбиение диаграммы взаимодействия для распределения ответственности за действия.

Зависимость - семантическое отношение между двумя сущностями, при которой изменение одной (независимой) сущности может повлиять на семантику другой (зависимой).

Задача - путь выполнения программы, динамической модели или иного представления потока управления; процесс или нить.

Запрос - спецификация стимула, посылаемого объекту.

Значение - элемент области определения типа.

Иерархия вложенности - иерархия в пространстве имен, состоящая из элементов и отношений агрегирования между ними.

Импорт: в контексте пакетов - зависимость, показывающая пакет, на классы которого можно ссылаться внутри данного пакета (включая и рекурсивно вложенные в него пакеты).

Имя - название сущности, отношения или диаграммы; строка, идентифицирующая элемент.

Инкрементный подход: в контексте цикла разработки программного

обеспечения - процесс непрерывного развития архитектуры системы, когда каждая новая версия содержит улучшения по сравнению с предыдущей.

Интерфейс - множество операций, составляющее спецификацию услуг, которые предоставляет класс или компонент.

Исполнение - прогон динамической модели.

Использование - зависимость, при которой один элемент (клиент) для правильного функционирования требует наличия другого элемента (поставщика).

Исследование - вторая фаза цикла разработки программного обеспечения, в ходе которой определяется общее видение продукта и его архитектура.

Итеративный подход: в контексте Шакла разработки программного обеспечения - процесс управления потоком исполняемых версий.

Итерация - четко очерченный перечень работ, для которых определены конечная цель и критерий оценки. В результате нескольких итераций должна быть выпущена версия для внутреннего или внешнего использования.

Каркас - архитектурный образец (паттерн), обеспечивающий расширяемый шаблон приложений в некоторой предметной области.

Квалификатор - атрибут ассоциации, значения которого разбивают множество объектов, связанных с некоторым объектом посредством данной ассоциации, на непересекающиеся подмножества.

Класс - описание множества объектов, обладающих общими атрибутами, операциями, отношениями и семантикой.

Класс-ассоциация - элемент модели, обладающий свойствами как класса, так и ассоциации. Класс-ассоциацию можно рассматривать либо как ассоциацию, обладающую свойствами класса, либо как класс, обладающий свойствами ассоциации.

Классификатор - механизм, с помощью которого описываются структурные и поведенческие особенности. К числу классификаторов относятся классы, интерфейсы, типы данных, сигналы, компоненты, узлы, прецеденты и подсистемы,

Клиент - классификатор, запрашивающий услугу у другого классификатора.

Комментарий - аннотация, присоединенная к элементу или множеству элементов.

Композит - класс, который связывается с одним или несколькими классами посредством отношения композиции.

Композиция - форма агрегирования, в которой целое владеет своими частя ми, имеющими одинаковое время жизни. Части с нефиксированной кратностью могут быть созданы после создания самого композита, но, будучи созданными, живут и умирают вместе с ним; такие части могут быть и явно удалены до момента уничтожения композита.

Компонент - физическая заменяемая часть системы, реализующая спецификацию интерфейсов.

Конкретный класс - класс, для которого можно, непосредственно создать экземпляры объектов.

Контейнер - объект, назначение которого - быть вместилищем других объектов; он предоставляет операции для доступа или последовательного обхода своего содержимого.

Контекст - множество взаимосвязанных элементов, предназначенное для определенной цели, например для специфицирования операции.

Концевая точка ассоциации - точка, в которой ассоциация соединяется с классификатором.

Концевая точка связи - экземпляр концевой точки ассоциации.

Кооперация - множество ролей и других элементов, совместно работающих для обеспечения кооперативного поведения, которое оказывается более значимо, чем сумма его составляющих; спецификация того, как элемент наподобие прецедента или операции реализуется посредством набора классификаторов и ассоциаций, играющих конкретные роли и используемых конкретным способом.

Кратность - спецификация диапазона возможных значений мощности множества.

Линия жизни объекта - линия на диаграмме последовательностей, которая

описывает существование объекта на протяжении некоторого промежутка времени.

Метакласс - класс, экземплярами которого являются классы.

Метод - реализация операции.

Механизм - образец (паттерн) проектирования, применимый к сообществу классов.

Механизм расширения - один из трех механизмов (стереотипы, помеченные значения и ограничения), с помощью которых можно контролируемым способом расширять язык UML.

Множественная классификация - семантическая разновидность обобщения, в которой объект может непосредственно принадлежать более чем одному классу.

Множественное наследование - семантическая разновидность обобщения, в которой потомок может иметь более чем одного родителя.

Модель - упрощение реальности, создаваемое для лучшего понимания разрабатываемой системы; семантически замкнутая абстракция системы,

Мощность множества - число элементов в множестве.

Наследование - механизм, с помощью которого более специализированные элементы заимствуют структуру и поведение более общих элементов.

Наследование интерфейса - наследование интерфейса более специализированного элемента; не включает наследование реализации.

Наследование реализации - наследование реализации более специализированного элемента; включает также наследование интерфейса.

Начальная фаза - первая фаза цикла разработки программного обеспечения, в которой исходная идея становится достаточно обоснованной, чтобы можно было принять решение о переходе к фазе исследования.

Неполнота - моделирование элемента, некоторые части которого отсутствуют.

Несовместимое подсостояние - подсостояние, в котором система не может находиться, одновременно находясь в других подсостояниях внутри одного и того же состояния.

Несогласованность - моделирование элемента, для которого не гарантируется

логическая непротиворечивость модели.

Нить - облегченный поток управления, который может выполняться параллельно с другими (вычислительными) нитями в пределах одного и того же процесса.

Область действия - контекст, в котором употребление некоего имени является осмысленным.

Обобщение - отношение специализации/обобщения, в котором объекты специализированного элемента (потомка) могут быть подставлены вместо объектов обобщенного элемента (родителя, или предка).

Образец (паттерн) - типичное решение типичной проблемы в данном контексте.

Обратное проектирование - процесс преобразования кода на конкретном языке программирования в модель.

Объект - конкретная материализация абстракции; сущность с хорошо определенными границами, в которой инкапсулированы состояние и поведение; экземпляр класса.

Обязанность - контракт или обязательство, принимаемое на себя типом или классом.

Ограничение - расширение семантики элемента UML, позволяющее добавлять новые или модифицировать существующие правила.

Одиночное наследование - семантическая разновидность обобщения, когда потомок может иметь только одного родителя.

Операция - реализация услуги, которая может быть запрошена у любого объекта класса.

Особенность - свойство, например операция или атрибут, которое инкапсулировано внутри другой сущности, такой как интерфейс, класс или тип данных.

Особенность поведения - динамическая характеристика элемента, такого как операция или метод.

Отметка времени - обозначение для момента "наступления события.

Отношение - семантическая связь между элементами. .

Отправитель - объект, передающий экземпляр сообщения объекту-получателю.

Отправка - передача экземпляра сообщения от объекта-отправителя объекту - получателю.

Пакет - универсальный механизм организации элементов в группы.

Параллельное подсостояние - подсостояние, в котором система может находиться одновременно с нахождением в других подсостояниях внутри одного и того же составного состояния.

Параллельность - выполнение двух или более работ в течение одного и того же промежутка времени. Параллельность может быть достигнута путём перемежающегося или истинного одновременного выполнения двух или более нитей.

Параметр - спецификация переменной, которая может быть изменена, передана или возвращена.

Параметризованный элемент - дескриптор элемента с одним или более несвязанными параметрами.

Паттерн (образец) - типичное решение типичной проблемы в данном контексте.

Переход - отношение между двумя состояниями, показывающее, что объект, находящийся в первом состоянии, должен выполнить некоторые действия и перейти во второе состояние, как только наступит некоторое событие и при этом будут выполнены определенные условия.

Перечислимый тип - список поименованных величин, образующих область значений некоторого атрибута.

Поведение - наблюдаемый эффект события, в том числе его результаты.

Поведенческое свойство - динамическое свойство элемента, такое как операция или метод.

Подкласс: в отношении обобщения - специализация другого класса, родителя.

Подсистема - группирование элементов, часть из которых составляет спецификацию поведения, предлагаемого другими содержащимися в нем элементами.

Подсостояние - состояние, являющееся частью другого состояния.

Положение - размещение компонента в узле.

Получатель - объект, обрабатывающий экземпляр сообщения, переданного объектом - отправителем.

Получение - обработка экземпляра сообщения, переданного объектом-отправителем.

Помеченное значение - расширение свойств элемента UML, которое позволяет включать новую информацию в его спецификацию.

Поставщик - тип, класс или компонент, предоставляющий услуги, которые могут быть востребованы другими элементами.

Построение - третья фаза цикла разработки программного обеспечения, в ходе которой исполняемый архитектурный прототип доводится до состояния, когда он может быть передан пользователям.

Постусловие - ограничение, которое должно быть выполнено по завершении операции.

Потомок - подкласс.

Предметная область -.область знаний или деятельности, характеризуемая концепциями и терминами, понятными тем, кто работает в данной области.

Предусловие - ограничение, которое должно быть выполнено, когда вызывается операция.

Прецедент - описание множества последовательных событий (включая варианты), выполняемых системой, которые приводят к наблюдаемому актером результату.

Примечание - графический символ для изображения ограничений или комментариев, присоединяемый к элементу или множеству элементов.

Примитивный тип - базовый тип, например "целое" или "строка".

Продукт - артефакт процесса разработки, такой как модель, код, документация и рабочий план.

Проекция - отображение множества на его подмножество.

Производный элемент - элемент модели, который можно вычислить по другим

элементам, но который тем не менее включен в нее для ясности или для удобства проектирования, несмотря на то что он не привносит новой семантики.

Пространство имен - область действия, в которой могут быть определены и использованы имена; внутри пространства имен каждое имя идентифицирует уникальный элемент.

Процесс - ресурсоемкий поток управления, который может выполняться параллельно с другими процессами.

Прямое проектирование - процесс преобразования модели в код путем отображения на конкретный язык программирования.

Псевдосостояние - вершина автомата, которая выглядит как состояние, но не ведет себя как таковое. К числу псевдосостояний относятся начальное, конечное и историческое состояния.

Реализация (Implementation) - конкретное воплощение контракта, объявленного интерфейсом; определение того, как что-либо конструируется или вычисляется.

Реализация (Realization) - семантическое отношение между классификаторами, в котором одна сторона формулирует условия контракта, а другая обязуется его выполнить.

Родитель - суперкласс, или "надкласс".

Роль - поведение сущности, участвующей в конкретном контексте.

Свертывание - моделирование элемента, некоторые части которого скрыты для упрощения восприятия.

Свойство - поименованное значение, обозначающее некоторую характеристику элемента.

Связывание - создание элемента по шаблону путем подстановки фактических аргументов вместо формальных параметров шаблона.

Связь - семантическое соединение между объектами; экземпляр ассоциации.

Сигнал - спецификация асинхронного стимула, передаваемого от одного экземпляра другому.

Сигнатура - совокупность имени и параметров операции.

Синхронное действие - запрос, послав который, объект-отправитель ожидает результат.

Система - множество элементов, организованных для достижения конкретной цели, иногда разложенное на несколько подсистем и описываемое набором моделей, возможно с различных точек зрения.

Событие - спецификация существенного факта, имеющего положение в пространстве и во времени. В контексте автоматов событие - это возникновение стимула, который может активизировать переход из одного состояния в другое.

Событие времени - событие, обозначающее истечение промежутка времени с момента входа в текущее состояние.

Сообщение - спецификация передачи информации между объектами в расчете на то, что за этим последует некоторая деятельность; прием сообщения обычно трактуется как возникновение события.

Составное состояние - состояние, составленное из параллельных или несовместимых подсостояний.

Состояние - ситуация в жизненном цикле объекта, во время которой он удовлетворяет некоторому условию, выполняет определенную деятельность или ожидает какого-то события.

Состояние действия - состояние, которое представляет вычисление атомарного действия, как правило - вызов операции.

Спецификация - текстовое объявление синтаксиса и семантики некоторого строительного блока; декларативное описание того, чем является или что делает некая сущность.

Срабатывание - выполнение перехода состояний.

Статическая классификация - семантическая разновидность обобщения, в которой объект не может изменять свой тип или роль.

Статический вид - аспект системы, в котором основное внимание уделяется ее структуре.

Стереотип - расширение словаря UML, позволяющее создавать новые виды строительных блоков, производные от существующих, но специфичные для

конкретной задачи.

Стимул - операция или сигнал.

Сторожевое условие - условие, которое должно быть выполнено для того, чтобы сработал переход, с которым оно ассоциировано.

Строка - последовательность символов, имеющих графическое начертание.

Структурное свойство - статическое свойство элемента.

Суперкласс: в отношении обобщения - обобщение другого класса, потомка.

Сценарий - конкретная последовательность действий, иллюстрирующая поведение.

Тип - стереотип класса, используемый для специфицирования семейства объектов, а также операций (но не методов), применимых к этим объектам.

Тип данных - тип, значения которого никак не идентифицированы. К типам данных относятся примитивные встроенные типы (например, числа и строки), а также перечислимые типы (например, булевский).

Трассировка - зависимость, которая показывает историческое или процессуальное отношение между двумя элементами, представляющими одну и ту же концепцию, без указания правил вывода одного элемента из другого.

Требование - желаемая функциональность, свойство или поведение системы.

Узел - физический элемент, существующий во время выполнения системы и представляющий вычислительный ресурс, который обладает по меньшей мере памятью, а зачастую также и процессором.

Управляемый прецедентами: в контексте цикла разработки программного обеспечения - процесс, в котором прецеденты служат основным артефактом для формулирования желаемого поведения системы, для верификации и контроля системной архитектуры, для тестирования и для обмена информацией между участниками проекта.

Управляемый рисками: в контексте цикла разработки программного обеспечения - процесс, в котором при выпуске каждой новой версии основное внимание обращается на выявление и уменьшение факторов, представляющих наибольший риск для успешного завершения проекта.

Уровень абстракции - точка в иерархии абстракций, нисходящей от верхних (очень абстрактных) до нижних (очень конкретных) уровней.

Устойчивый объект - объект, который продолжает существовать после завершения создавшего его процесса или потомка.

Уточнение - отношение, которое представляет более полную спецификацию того, что ранее уже было специфицировано на определенном уровне детализации.

Фаза - промежуток времени между двумя опорными точками в процессе разработки, в течение которого должны быть достигнуты заранее поставленные хорошо определенные цели, артефакты доведены до готовности и принято решение о том, следует ли переходить к следующей фазе.

Фактический параметр - аргумент функции или процедуры.

Фокус управления - символ на диаграмме последовательностей, показывающий промежуток времени, в течение которого объект выполняет некоторое действие непосредственно или путем вызова подчиненной операции.

Целостность - правильность и согласованность взаимодействия различных сущностей.

Шаблон - параметризованный элемент.

Экземпляр - конкретная материализация абстракции. К этой сущности могут быть применены операции; она обладает состоянием, в котором запоминаются результаты операций.

Экспортировать: в контексте пакетов - делать элемент видимым вне объемлющего пространства имен.

Элемент - атомарная составляющая модели.

Элемент распределения - множество объектов или компонентов, размещенных в некотором узле как единая группа.