Глава 1

Введение

В этой главе:

- Основные концепции UML для моделирования Обзор UML. Что такое паттерн проектирования?
- Модели Классов и Объектов
  Что такое классы и объекты. Как классы и объекты совместно работают в кооперации. Кооперации являются реализацией вариантов использования. Группирую логические элементы модели.
- Модели Компонентов и Размещения
  Определяют артефакты времени исполнения и их размещение на вычислительных узлах.
- Диаграммы состояний и Поведенческие Модели
  Что такое диаграммы состояний и как они используются для моделирования поведения?
- Модели Вариантов использования и Требований
  Описание поведения системы "черного ящика" без показа ее внутренней структуры

1.1 Основные концепции UML для моделирования

Унифицированный язык моделирования (UML) -- является стандартом языка объектного моделирования 3ro поколения и принадлежит Object Management Group (OMG). Первоначальная версия стандарта OMG UML 1.1 была выпущена в ноябре 1997 г. С тех пор были выпущены несколько младших ревизий и одна старшая. На момент, когда автор работал над этой книгой, текущей версией стандарта являлась версия 2.0. Она доступна с сайта www.omg.org.

Язык UML обладает богатыми возможностями для моделирования программного обеспечения и систем. Для моделирования прогаммного обеспечения UML является стандартом *де факто*. Есть несколько причин, которые, как я считаю, объясняют его феноменальный успех. Во-первых, UML достаточно прост в изучении, а после изучения достаточно прост в использовании. Во-вторых, UML является формальным языком и поэтому модели на языке UML могут быть проверены (если для достижения точности приложены достаточные усилия). Полученные модели могут быть непосредственно исполнены (с использованием соответствующих инструментальных средств, таких как Rhapsody™), а на их основе может быть сгенерирован программный код, который может быть

использован в конечной системе. И, в-третьих, существует *огоромное* количество инструментальных средств, поддерживающих UML. Есть не просто много вендоров на рынке, а вендоров отличающихся друга от друга благодаря специализации на различных аспектах моделирования и разработки.

UML использует графическую нотацию, простую и удобную в применении и в большей части простую для понимания[1]. Несмотря на то, что некоторые люди утверждают что в UML *слишком много диаграмм*, на самом деле есть только четыре основных типа (см. Рис. 1-1) структурные диаграммы, к которым относятся диаграммы классов, структуры, объектов, пакетов, компонентов и диаграммы размещения. Эти диаграммы отличаются друг от друга не сколько отображаемыми элементами, сколько своим *предназначением.* Функциональные диаграммы подчеркивают функциональность, а не структуру и поведение; к функциональным диаграммам относятся диаграммы вариантов использования и диаграммы информационных потоков. Диаграммы взаимодействия предназначены для отображения каким образом элементы взаимодействуют друг с другом во времени для реализации своей функциональности. К диаграммам взаимодействия относятся диаграммы последовательности, диаграммы коммуникаций (ранее известные как "диаграммы кооперации") и временные диаграммы. И, наконец, диаграммы поведения предназначены для на определения поведения индивидуальных элементов. К ним относятся диаграммы состояний и диаграммы деятельности. Хотя широкие возможности нотации UML могут несколько озадачить новичков, на практике сложные системы могут быть разработаны с использованием трех ключевых диаграмм: диаграмм классов, диаграмм состояний и диаграмм последовательности. Остальные диаграммы могут использоваться для моделирования дополнительных аспектов системы (например, для сбора требований, или как программное обеспечение размещается на аппаратуре). Дополнительные диаграммы несомненно привносят ценность модели, но наиболее часто вам будут необходимы только три базовых типа диаграмм для разработки систем и программного обеспечения.

[1]Хотя, как и у всех языков, у UML имеются свои недостатки.

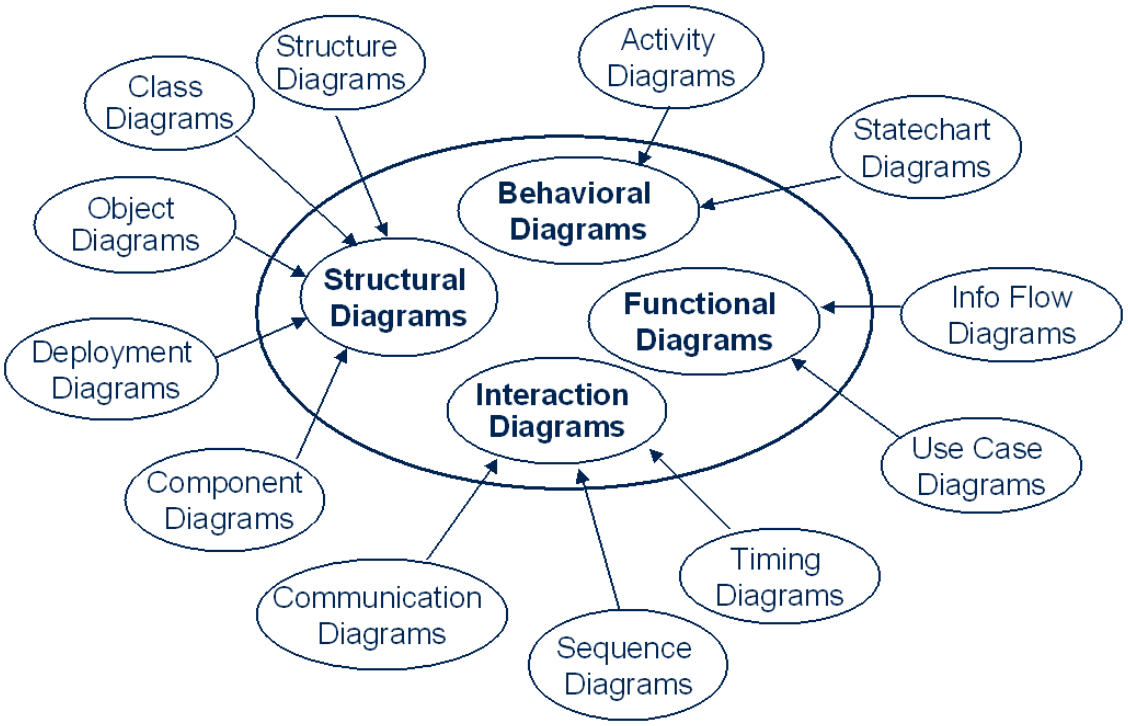


Рис. 1-1: Виды диграмм UML

В основе UML лежит формально определеная семантическая модель, которя называется метамоделью UML. Данная семантическая модель является одновременно широкой (охватывает большинство аспектов, необходимых для спецификации и проектирования систем) и глубокой (что позволяет создавать модели, которые являются одновременно *точными и исполняемыми*, на основе которых может генерироваться программный код, готовый к компиляции. В заключении необходимо отметить, что при помощи UML разработчик может смоделировать любые аспекты системы, которые ему необходимо осознать и отобразить. Рис. 1-2 содержит снимок экрана с результатами исполнения модели. На диаграммах в Rhapsody для отображения различных аспектов, таких как текущее состояние, используется выделение цветом. К сожалению, черно-белые иллюстрации не позволяют хорошо отобразить такое выделение. На рисунке также показаны элементы управления исполнением: шаг внутрь, шаг вперёд, установка точки прерывания, генерация событий и т.п. Rhapsody позволяет динамически создавать диаграммы последовательности, на которых показана история взаимодействия выбранных объектов во время их исполнения [2].

[2]Все модели в данной книге созданы при помощи Rhapsody. На прилагаемом к книге компакт-диске содержится пробная версия Rhapsody, которую вы можете использовать для выполнения заданий, составляющих основу этой книги. Более подробные инструкции вы можете получить, обратившись к файлу README.TXT на компакт-диске, а также ознакомившись с учебными пособиями, доступными из меню Help->List of Books.

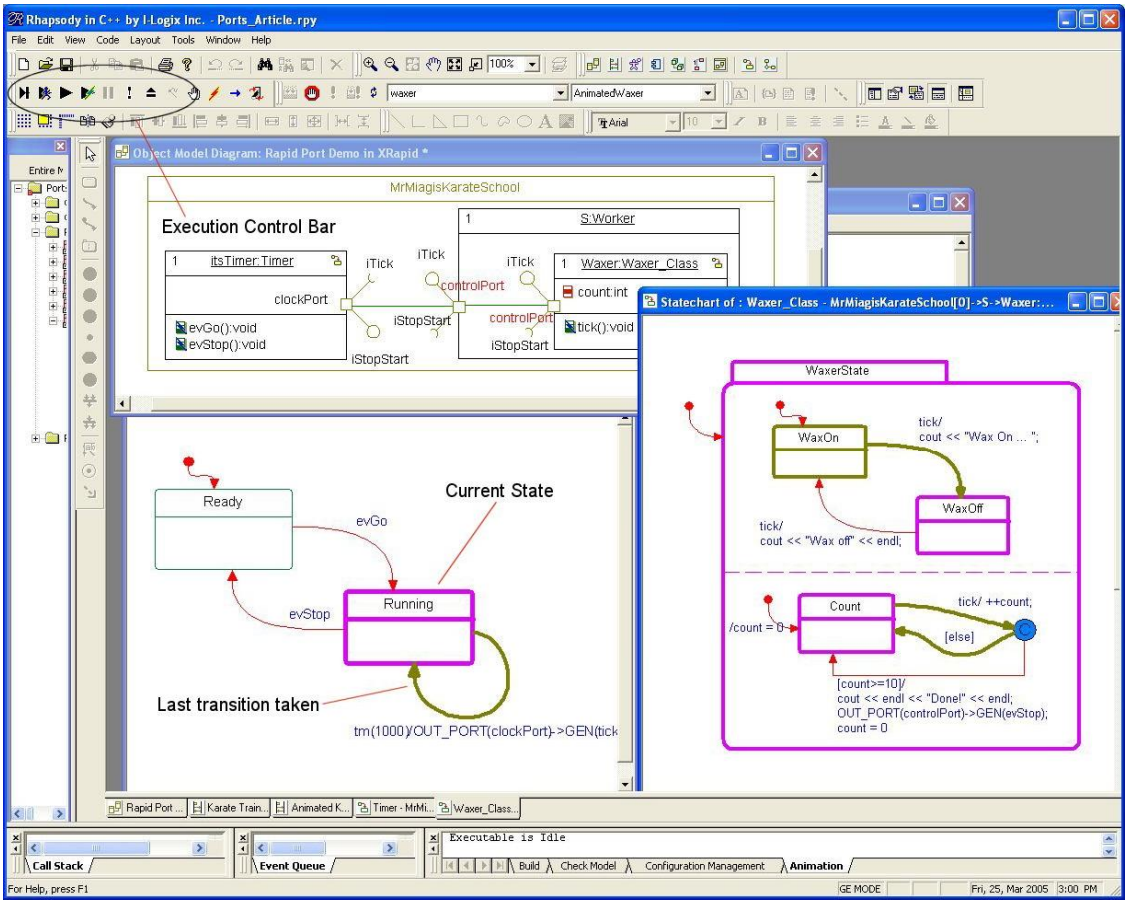


Рис. 1-2: Исполнение модели

UML является *стандартом*, в отличие от большинства языков моделирования, которые являются собственностью одной компании и поддерживаются одним инструментом. Использование стандартизованного языка моделирования означает возможность выбора разработчиком инструментальных средств и услуг из множества источников. Так, в настоящее время доступно более десятка различных инструментов моделирования на основе UML. Поскольку все эти инструментальные средства сфокусированы на различных аспектах моделирования и значительно различаются по стоимости, разработчики могут выбрать из них наиболее удовлетворяющий их потребностям и потребностям их проекта. Например, в Rhapsody компании IBM -- большое внимание уделяется глубокой семантики UML, что предоставляет возможность валидации и тестирования моделей пользователей путем их исполнения и отладки с отображением результатов на анимированных диаграммах UML. Исполнение может осуществляться как на машине разработчика, так и на целевой аппаратуре, а сгенерированный код может использоваться в финальной поставляемой системе. В других инструментальных средствах большее внимание уделяется иным вопросам. Например, могут быть реализованы возможности для рисования диаграмм, предоставляющие большую гибкость за меньшую цену. Такое разнообразие

предоставляет разработчикам большую свободу при выборе инструментального средства. Это также является дополнительным стимулом для инноваций и улучшений инструментальных средств. Поскольку UML является широко используемым стандартом, многие компании проводят тренинги по UML и его практическому использованию. Я сам провожу достаточно много времени, консультируя и обучая разработчиков по всему миру, специализируясь на использовании UML при разработке встраиваемых систем и программного обеспечения реального времени[3].

Язык UML *применим* для разработки систем и программного обеспечения в самых разных прикладных областях. За годы применения UML, он использовался для разработки буквально всех видов систем, содержащих программное обеспечение -- начиная с систем учета и заканчивая программным обеспечением управления полетом. Как и всякий поддерживаемый стандарт, UML со временем изменяется: в нем исправляются ошибки, реализуются хорошие идеи и исключаются неудачные. В настоящее время UML используется для моделирования и построения систем значительно отличающихся по своему размеру, разрабатываемых в проектах с участием одного-двух человек до проектов с участием тысяч разработчиков. UML поддерживает *все* сущности, необходимые для моделирования временных ограничений и управления ресурсами, которые характеризуют системы реального времени и встраиваемые системы. Это означает, что разработчику не приходится выходить за границы UML для определения различных аспектов разрабатываемых систем, независимо от их сложности.

В этой главе мы рассмотрим основы UML. *Ее цель -- стоит скорее в том, чтобы освежить, а не научить*. Тем, кому интересно более основательное рассмотрение самого UML, я рекомендую прочитать книгу дополняющую эту, *Real-Time UML 3rd Edition: Advances in the UML for Real-Time Systems* (Addison-Wesley, 2004) by Bruce Powel Douglass. Кроме того, на сайте компании IBM (www.ibm.com) опубликовано много вводных статей по этой тематике.

1.2 Структурные элементы и диаграммы

UML определяет достаточно богатый набор структурных элементов и предоставляет визуальные представления для связанных наборов элементов. Мы начнем обсуждение с моделирования малых элементов, затем рассмотрим моделирование крупных элементов, после чего рассмотрим способы организации моделей.

1.2.1 Малые сущности: объекты, классы и интерфейсы

[3]Читатели, нуждающиеся в профессиональных услугах, могут связаться со мной по адресу: bruce.douglass@telelogic.com.

В UML есть несколько элементарных структурных сущностей, которые используются в моделях пользователей: объекты, классы, типы данных и интерфейсы. Эти структурные элементы составляют основу структурного проектирования в моделях пользователей. Упрощенно, каждый *объект* представляет собой структуру данных, которая также предоставляет сервисы, для обработки этих данных. Объект существует только во время исполнения; это означает, что в определенное время при работе системы объект может занимать некоторое место в памяти компьютера. Данные, известные объекту, хранятся в *атрибутах* -- простых, элементарных переменных, локальных для данного объекта. Сервисы, выполняющие действия с этими данными, называются *методами*; эти сервисы запускаются клиентами данного объекта (как правило, другими объектами) или другими методами данного объекта. Диаграммы состояний и диаграммы деятельности могут определять допустимые последовательности вызовов этих сервисов, задавая пред- и постусловия которые должны при этом выполняться

Класс является спецификацией множества объектов, имеющих общую структуру и общее поведение. Объекты -- это *экземпляры* класса. Во время работы системы могут создаваться несколько экземпляров класса, однако каждый объект является экземпляром только одного класса. Для класса может быть определена диаграмма состояний, которая координирует и управляет исполнением его примитивных поведенческих элементов (называемых *действиями*, которые часто являются просто вызовами методов класса), определяя множество допустимых последовательностей вызовов.

На Рис. 1-3 показан пример типичной диаграммы классов. Заметьте, что объекты на ней не показаны. Это потому, что когды вы отображаете объекты, вы показываете "моментальный снимок" работающей системы в определенный момент времени. Диаграммы классов представляют множество возможных структур объектов. В большинстве случаев, для создания структурных представлений, вы будете использовать диаграммы классов, а не диаграммы объектов.

Элемент DeliveryController на рисунке -- это пример класса. Он содержит атрибуты, например, commandedConcentration (имеющий тип double) и selectedAgent (имеющий тип AgentType). И предоставляет методы, например, для выбора нужного лекарственного препарата (selectAgent), или получения количества оставшегося лекарственного препарата (getVolume), а также для задания концентрации лекарственного препарата (setConcentration). Класс DeliveryController изображен в виде обычного прямоугольника, разделенного на три сегмента. Верхний сегмент прямоугольника содержит имя класса. Среднии сегмент содержит список атрибутов. Следует отметить, что этот список не обязательно полон – не все атрибуты класса должны обязательно показываться на диаграмме. В нижнем сегменте прямоугольника показаны методы, определенные в данном классе.

На Рис. 1-3 также показаны другие классы и линии (называемые *ассоциациями* - о них будет рассказываться далее), связывающие их. Некоторые из классов отображаются в виде простых, не разделенных на сегменты прямоугольников – как, например, класс TextView. Атрибуты и методы, которые вместе называются *составляющими* класса, не обязательно должны отображаться на диаграмме. Они содержатся в модели, и их всегда можно просмотреть в окне обозревателя модели. При желании их можно отобразить и на диаграмме. Вы можете управлять тем, какие составляющие отображаются на разных диаграммах.

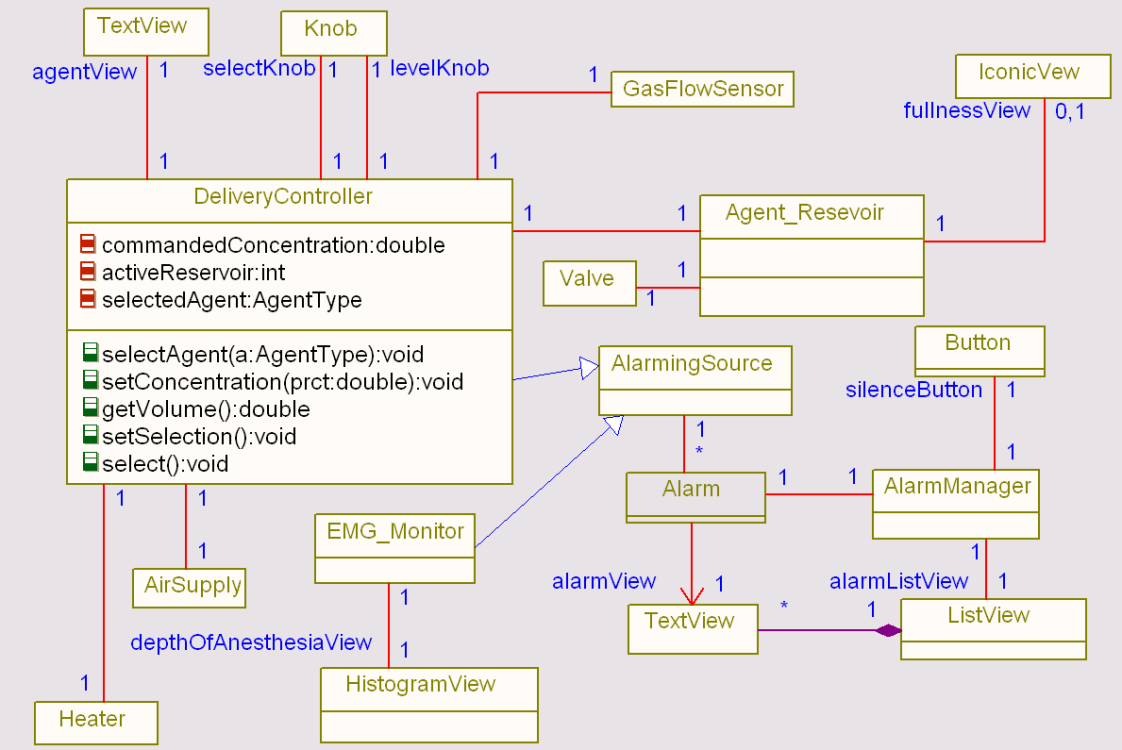


Рис. 1-3: Типичная диаграмма классов

Интерфейс представляет собой именованный набор сервисов. Сервисы бывают двух видов. Операции являются синхронными сервисами, которые запускаются (или *вызываются*) клиентами класса. Реакции на события -- это асинхронные сервисы, которые запускаются путем посылки асинхронного сигнала о событии объекту, обрабатывающему данное событие. Интерфейс может предоставлять возможности для запуска операций и посылки асинхронных сигналов. Как при запуске операций, так и при посылке асинхронных сигналов могут передаваться данные, называемые параметрами. Имя сервиса, в совокупности с набор параметров, называют *сигнатурой* сервиса. Класс, совместимый с интерфейсом, предоставляет метод для каждой операции и реакцию на событие для каждого асинхронного сигнала, определенных в интерфейсе.

Интерфейсы позволяют отделить определение сервисов, которые могут быть запущены для класса, от реализации этих сервисов. Класс определяет *методы* (для операций) и *реакции на события* (которые определяются на диаграмме состояний для соответсвующих сигналов о событиях). Как для методов, так и для реакций на события определяется программный код, реализующий данные сервисы. Операции и сигналы – это лишь спецификации, которые не содержат реализаций.

Интерфейсы не могут иметь реализации (т.е. не могут иметь атрибутов или методов) и для них не могут быть созданы экземпляры. Говорят, что класс *реализует* интерфейс, если он предоставляет метод для каждой операции, определенной в этом интерфейсе, и эти методы имеют те же имена, параметры, возвращаемые значения, предусловия и постусловия, что и соответствующие операции интерфейса.

Интерфейсы могут быть отображены на диаграммах двумя способами. В первом способе интерфейс выглядит на диаграмме так же, как и класс, за исключением ключевого слова interface в двойных кавычках над его именем («interface»). Этот способ называется в UML "с использованием *стереотипа*". Он применяется в тех случаях, когда необходимо показать на диаграмме сервисы этого интерфейса. Во втором способе, который часто называют нотацией чупа-чупс, интерфейс отображается в виде маленького именованного кружка сбоку от класса. Оба эти способа показаны на Рис. 1-4 Когда используется нотация чупа-чупс, отображается только имя интерфейса. Когда используется способ со стереотипом, на диаграмме может быть показан список сервисов интерфейса.

На Рис. 1-4 показаны два интерфейса. В данном случае оба интерфейса содержат асинхронные сигналы о событии. Стрелка обобщения определяет класс, который реализует данный интерфейс (т.е. предоставляет сервисы). Здесь также показаны порты для классов (более подробно о портах будет рассказываться далее). Порты типизируется интерфейсами, которые или предоставляются или используются через этот порт. В этом примере clockPort, составляющая класса Timer, предоставляет сервисы, определенные интерфейсом iStopStart, и использует сервисы, определенные интерфейсом iTick. Это означает, что для подключения к этому порту другой класс должен предоставлять интерфейс iTick и использовать интерфейс iStopStart. Для обозначения предоставляемого (также называемого "предлагаемого") интерфейса используется нотация чупа-чупс, а для используемого интерфейса - нотация сокета. Это позволяет разработчикам определять операционные контракты, которые разрешают связывать друг с другом только совместимые элемент, что особенно важно при разбиении системы на подсистемы и компоненты.

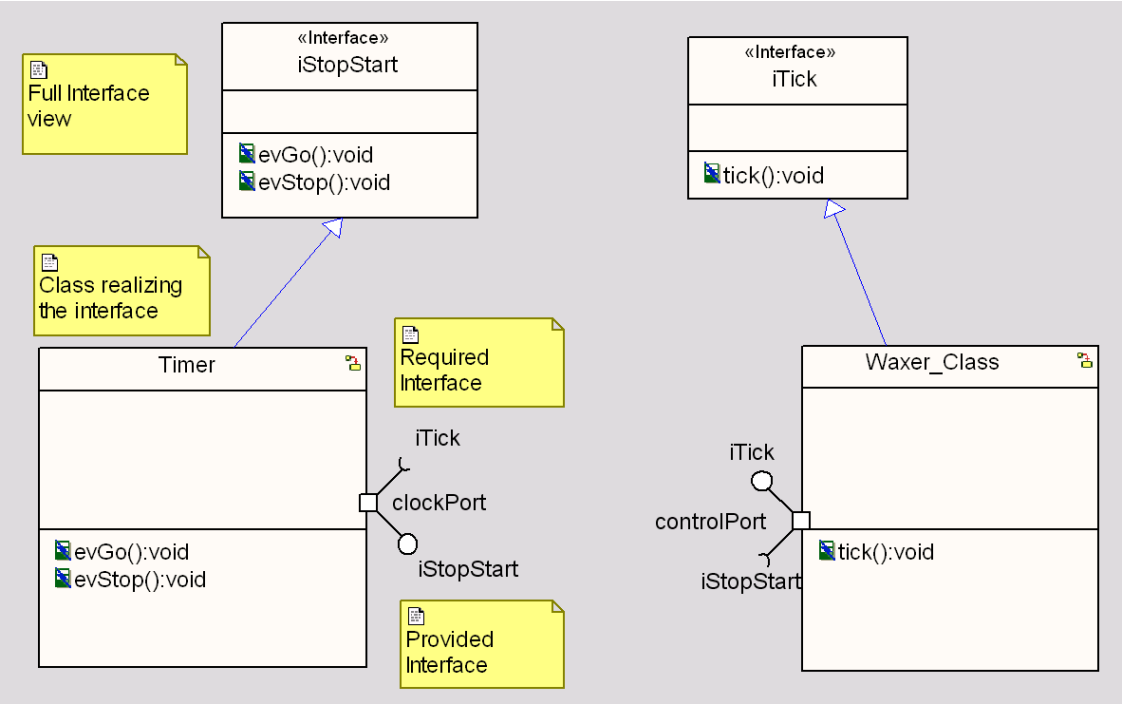


Рис. 1-4: Интерфейсы

Если говорить кратко, объект является одним из многих *экземпляров* класса. У класса имеется две важных составляющих – атрибуты (в которых хранятся значения данных) и методы (которые предоставляют сервисы клиентам класса). Интерфейсы представляют собой именованные наборы операций, которые *реализуются* классами. Использование интерфейсов совсем не обязательно. Большое количество полезных систем были спроектированы исключительно с использованием классов. Однако существуют случаи, когда дополнительный уровень абстракции бывает необходим, особенно когда создается несколько реализаций одного интерфейса.

Структурные классы

Одним из наиболее значимых нововведений в UML 2.0 стали *структурные классы*. Структурный класс – это класс, который состоит из нескольких составных *частей* (ролей объектов), которые сами типизированны с помощью классов. Структурные классы имеют такие же составляющие и свойства (такие как атрибуты и методы), как и обычные классы. Но кроме этого они содержат и управляют своими внутренними частями. Структурные классы являются очень существенным улучшением стандарта UML [4], потому что с помощью их стало возможным явным образом определять вложенные архитектурные структуры. Например, системы содержат подсистемы как части; в свою очередь, подсистемы могут содержать

[4] В Rhapsody эта возможность была реализована, начиная с самой первой версии.

подсистемы следующего уровня, и так далее вплоть до простых (не являющихся структурными) частей. Кроме того, поскольку для представления параллельных элементов (т.е. потоков или задач) используются активные объекты, то эти активные объекты могут содержать пассивные (работающие не параллельно) объекты в виде частей, позволяя ясным образом определять архитектуру параллелизма.

На Рис. 1-5 показан обычный структурный класс SensoryAssembly. Этот класс содержит внутренние части: такие как positioner (типизированный классом Gimbal), MissionTasker (его класс не показан на диаграмме) и theSensor (типизированный классом Sensor). Некоторые из этих частей, в свою очередь, также являются структурными классами и содержат составные части. Таким образом, мы можем конструировать иерархии вложенности любой сложности.

Необходимо сделать еще несколько замечаний о том, что изображено на Рис. 1-5. Во-первых, отношение между классом и его частями не является отношением между классами. Оно подразумевает отношение композиции между классами, но сами части не являются классами. Класс, типизирующий часть, может использоваться для определения и других частей, в том числе в различных структурных классах. Части также не являются объектами. Все экземпляры SensorAssembly имеют такую структуру, таким образом для своей составной части positioner класс не ссылается на конкретный объект, существующий в единственном числе. Когда мы создаем экземпляр класса SensorAssembly, некий экземпляр класса Gimbal будет играть роль, которую определяет данная часть. Однако сама эта часть не является экземпляром. По сути, это роль, которую некоторый экземпляр будет играть во время работы системы, и эта роль типизируется определением структурного класса. UML использует термин *часть* исключительно для обозначения такой роли объекта.

Во-вторых, необходимо отметить присутствие небольших квадратиков на границах прямоугольников различных классов и частей. Эти квадратики называются портами и будут рассматриваться далее.

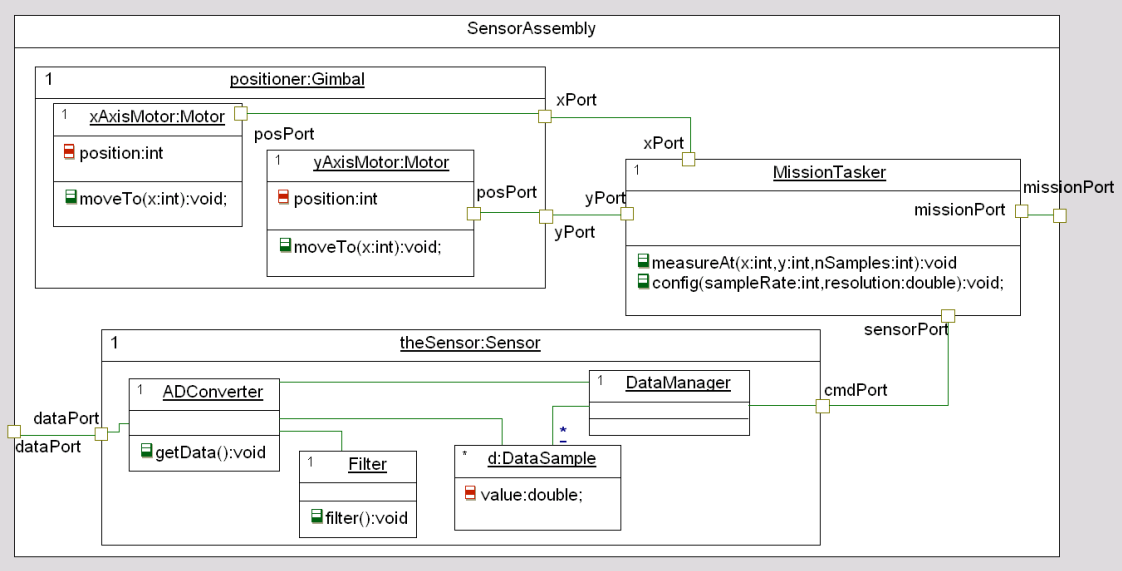


Рис. 1-5: Структурные классы

Порты

Классы могут содержать составляющие, называемые *портами* . Порты являются именованными точками соединения классов и, как уже упоминалось ранее, типизируются сервисами, которые используются или предоставляются через них.

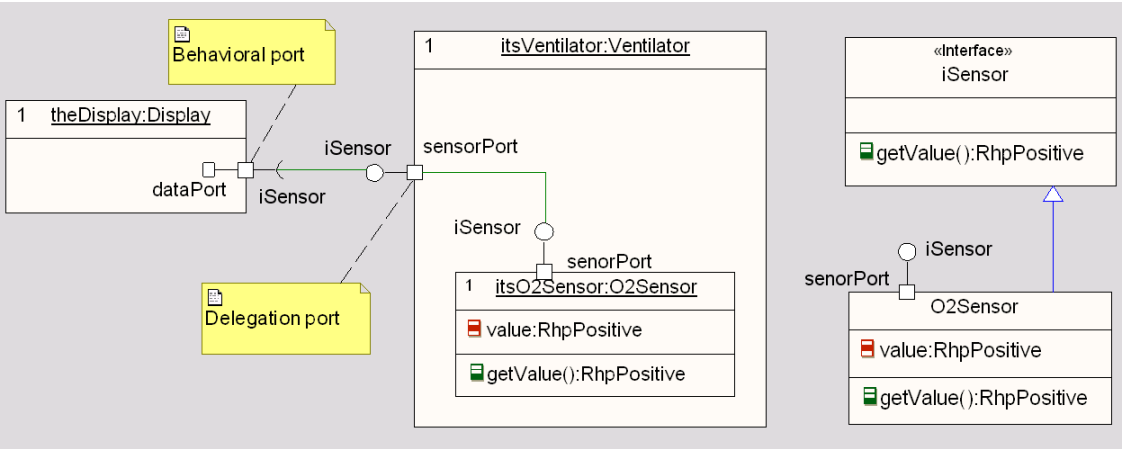


Рис. 1-6: Порты

1.2.2 Отношения

Для реализации поведения уровня системы многие объекты должны работать совместно, а для совместной работы они должны быть каким-то образом связаны. Как часть структурного моделирования, UML определяет различные виды

отношений между структурными элементами. Далее мы в основном сосредоточим свое внимание на отношениях между классами, но эти отношения времени проектирования между спецификациями формально определяют отношения между экземплярами этих классов.

1.2.2.1 Ассоциации

UML определяет несколько видов отношений между классами. Важнейшими из них являются ассоциации, обобщения и зависимости. Наиболее простое отношение называется *ассоциацией*. Ассоциация -- это отношение времени проектирования между классами, которое определяет что во время исполнения между экземплярами этих классов может существовать *связь* (отношение между объектами с возможностью доступа, благодаря которому они могут запускать сервисы друг друга).

В UML определены три различных типа ассоциаций: просто ассоциация, агрегирование и композиция. Ассоциация между классами означает, что в некоторый момент работы системы между экземплярами этих классов может существовать связь, позволяющая им запускать сервисы друг друга. При этом ничего не говорится о том, *как* это происходит, и даже является ли это синхронным вызовом метода (это наиболее распространенный случай) или посылкой асинхронного сигнала о событии. Ассоциации являются определением средства взаимодействия, позволяющего объектам во время исполнения посылать сообщения друг другу. На диаграммах классов ассоциации изображаются линиями, соединяющими классы.

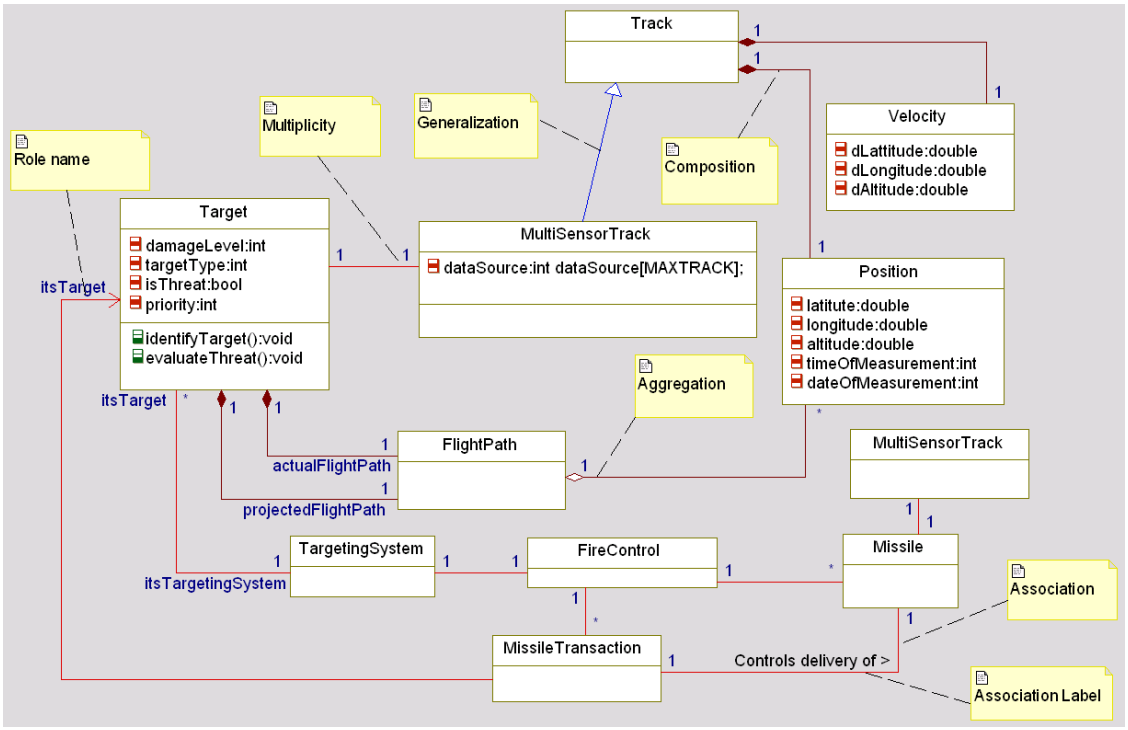


Рис. 1-7: Отношения между классами

Для ассоциаций между двумя классами могут быть определены несколько дополнительных аспектов. Например, для обоих концов ассоциации может быть определено *имя роли*. Они определяет имена для экземпляров классов, под которыми они видны из противоположного класса в ассоциации. Наиболее частой реализацией ассоциации является указатель, с именем совпадающим с именем роли противоположного конца ассоциации. Например, см. Рис. 1-7, в классе *Target* может быть опеределен указатель с именем *itsTargetingSystem*, который будет использоваться во время исполнения. Аналогично в классе TargetingSystem может быть определен указатель *itsTarget*, позволяющий TargetingSystem посылать сообщения для Target. Я использую слово "может", потому что это всего лишь наиболее частый способ реализации ассоциации, однако существуют и другие способы.

Менее распространенным является использование *меток ассоциаций*, как например, между *MissileTransaction* и *Missile*. Обычно метка используется для пояснения, почему между двумя классами определена ассоциация. В данном случае метка "Контролирует доставку >" описывает для каких целей данное отношения используется. С помощью значка ">" можно определять направление для метки, в данном случае, направление от *MissileTransaction* к *Missile*.

Множественность – это, пожалуй, наиболее важная характеристика для конца ассоциации. Множественность для конца ассоциации определяет число

экземпляров класса, которые во время исполнения могут в данной роли участвовать в ассоциации. Множественность может быть представлена в виде:

- определенного числа, например "1" или "MAX_DATA_SOURCES"
- списка значений, разделяемых запятыми, например: "0,1" или "3,5,7"
- диапазона значений, например "1..10"
- комбинации списка и диапазона значений, например: "1..10, 25", что означает "от одного до 10 включительно, или 25"
- звездочки, что означает "ноль или более"
- диапазона значений со звездочкой на месте верхнего предела, например: "1..*", что означает "один или более".

На рисунке мы видим, что множественность определена для всех концов ассоциаций, за исключением направленной ассоциации между *MissileTransaction* и *Target.* Линия без стрелок означает, что ассоциация является двунаправленной, т.е. объекты на обоих концах ассоциации могут посылать сообщения объекту на противоположном конце ассоциации. В том случае, когда только один из объектов может посылать сообщение другому, но не наоборот, мы добавляем стрелку с *не закрашенным концом* (далее мы увидим, что тип стрелки имеет значение), указывающую направление посылки сообщений. На конце ассоциации, не продоставляющего возможность доступа (на конце без стрелки), множественность обычно не указывается. Это делается потому, что для конца ассоциации, предоставляющем возможность доступа, множественность влияет на реализацию ассоциации в классе. Множественность 1 может быть реализована при помощи простого указателя, а множественность * (ноль или более) потребуется массив указателей. В тоже время, для конца ассоциации без возможности доступа (как в случае "противоположного" конца направленной ассоциации), реализация *остается* неизменной, поэтому его множественность просто опускается.

Все эти дополнительные средства, используемые в обозначениях ассоциаций, за исключением наверное множественности, не являются обязательными и могут добавляться при желании с целью более точного описания отношения между соответствующими классами.

Ассоциация между классами означает, что в определенный момент жизненного цикла экземпляров классов, для которых определено отношение ассоциации, *может* существовать связь, позволяющая этим экземплярам обмениваться сообщениями. При этом не делается никаких утверждений или предположений о том, какой из этих объектов создается первым, какой объект их создает и как устанавливается связь между объектами. Как мы увыдим чуть позже, композиция, усиленная форма агрегации, *налагает* ответственность по созданию и уничтожению объектов.

1.2.2.2 Агрегация

Агрегация – это особый вид ассоциации, указывающий на то, что между двумя объектами существует отношение "целое-часть". Конец ассоциации со стороны объекта-целого, обозначается белым ромбом, как на Рис. 1-7. Например, рассмотрим классы *FlightPath* и *Position*. Класс *FlightPath* является "целым", включающим в себя множество элементов *Position*. Ромб на отношении агрегации указывает на то, что класс *FlightPath* является "целым". Знак "*" на конце ассоциации указывает на то, что список может содержать ноль и более элементов Position. Если бы мы хотели ограничить число путевых точек количеством не более 100, то могли бы задать множественность "0..100".

Поскольку агрегация является специальной формой ассоциации, все вышеописанные средства, которые применимы для ассоциаций, применимы и для отношения агрегации, включая направленность, множественность, имена ролей и имя ассоциации.

Как мы вскоре увидим, агрегация является относительно слабой формой отношения "целое-часть". Она ничего не утверждает о зависимостях жизненных циклов объектов или ответственности по их созданию/уничтожению. На самом, деле на этапе проектирования и реализации агрегация часто воспринимается как ассоциация. Тем не менее, использование агрегации может быть полезно для понимания структуры модели и отношений между концептуальными элементами предметной области. Иногда бывает неясно, что нужно использовать – ассоциацию или агрегирование. В таких случаях не следует беспокоиться. Агрегация обычно реализуется так же, как ассоциация, поэтому это никак не меняет реализацию. Если вы не можете выбрать, просто выберите один из вариантов, при необходимости, вы сможете его изменить впоследствии.

1.2.2.3 Композиция

Композиция является сильной формой агрегации, в которой элемент "целое" (также называемый "композитом") отвечает за создание и уничтожение объектов-частей. В предыдущем разделе мы познакомились с нотацией, используемую для структурного класса. Так вот композиция является отношением между структурным классом и классами его частей. Существует ключевое различие между агрегацией и композицией. В композиции вы явным образом назначаете на композит ответственность за создание и уничтожение объекта. Поэтому, на конце для «целого» множественность отношения композиции *всегда* равна 1. Объект может иметь несколько владельцев-агрегатов, но только одного владельца-композита.

Из-за ответственности за жизненный цикл частей, композит существует до того как части начнут свое существование и существует после их уничтожения. Если части

имеют фиксированную множественность по отношению к композиту, то обычно такие части создаются в его конструкторе (специальной операции, создающей объект) и уничтожаются в его деструкторе. Если же множественность не является фиксированной (т.е. "*"), то композит динамически создает и уничтожает объекты-части во время исполнения. Поскольку композит отвечает за создание и уничтожение, каждый из объектов-частей может принадлежать *только одному* композиту, хотя может участвовать в других отношениях ассоциации и агрегации. Композиция также является разновидностью ассоциации, поэтому для нее могут использоваться все средства, используемые для обычных ассоциаций.

На Рис. 1-7 композиция показана в виде закрашенного ромба, а на Рис. 1-5 в виде вложенных частей (структурного класса). В качестве упражнения, перерисуйте Рис. 1-5 так, чтобы использовались заштрихованные ромбы, а Рис. 1-7, чтобы использовалась вложенные части. Для того чтобы диаграммы были семантически правильными, закрашенный ромб должен определять только отношение между *классами*, а вложенные части -- отношение между классом и его частями (ролями объекта).

1.2.2.4 Обобщение

Отношение обобщения в UML означает, что один класс определяет множество составляющих, которые либо специализированы или расширены в другом классе. Обобщение можно понимать как отношение вида "этот класс является разновидностью другого", и поэтому оно относится ко времени проектирования, а не времени исполнения.

Обобщение активно используется при моделировании классов. Обобщение используется как средство, обеспечивающее совместимость интерфейсов классов, практически также как это делается для элементов интерфейс. На самом деле, использование обобщения позволяет реализовать интерфейсы на языках, в которых интерфейсы не являются встроенными концепциями языка – как, например, в C++. Также, обобщение может существенно упростить ваши модели классов, поскольку множество составляющих, общих для нескольких классов, могут быть вынесены вместе в единый суперкласс, вместо того чтобы повторно определять их во многих производных классах. И, наконец, обобщение позволяет подменять различные реализации. Например, в реализации одного производного класса может оптимизировать ся производительность в наихудшем случае, в реализации другого – оптимизироваться размер используемой памяти, а в реализации третьего – оптимизироваться надежность системы за счет внутренней избыточности. Экземпляры различных производных классов могут быть подставлены в качестве экземпляров суперкласса и при этом кооперация будет продолжать работать.

Обобщение в UML означает две вещи: наследование и возможность подстановки. Прежде всего, оно означает *наследование* -- т.е. производные классы (как минимум)

имеют те же атрибуты, операции, методы и отношения, что и суперклассы, с которыми они связаны отношениями обобщения. Конечно, если бы производные классы были *идентичны* своим суперклассам, это было бы не столь интересно, поэтому производные классы могут отличаться от своих суперклассов следующими способами: путем специализации и расширения.

Производные классы могут *специализировать* операции или конечные автоматы своих суперклассов. Специализация означает, что одна и та же операция (или список действий на диаграмме состояний) реализуется отличным от суперкласса образом. Этот способ обычно называется *полиморфизмом*. Он позволяет реализовать следующее поведение. Допустим некоторый класс связан отношением ассоциации с другим классом, для которого определены производные классы. Тогда во время исполнения экземпляр первого класса может запустить операцию, определенную во втором классе. Если при этом связь установлена с экземпляром производного класса, запускается операция производного класса, а не суперкласса.

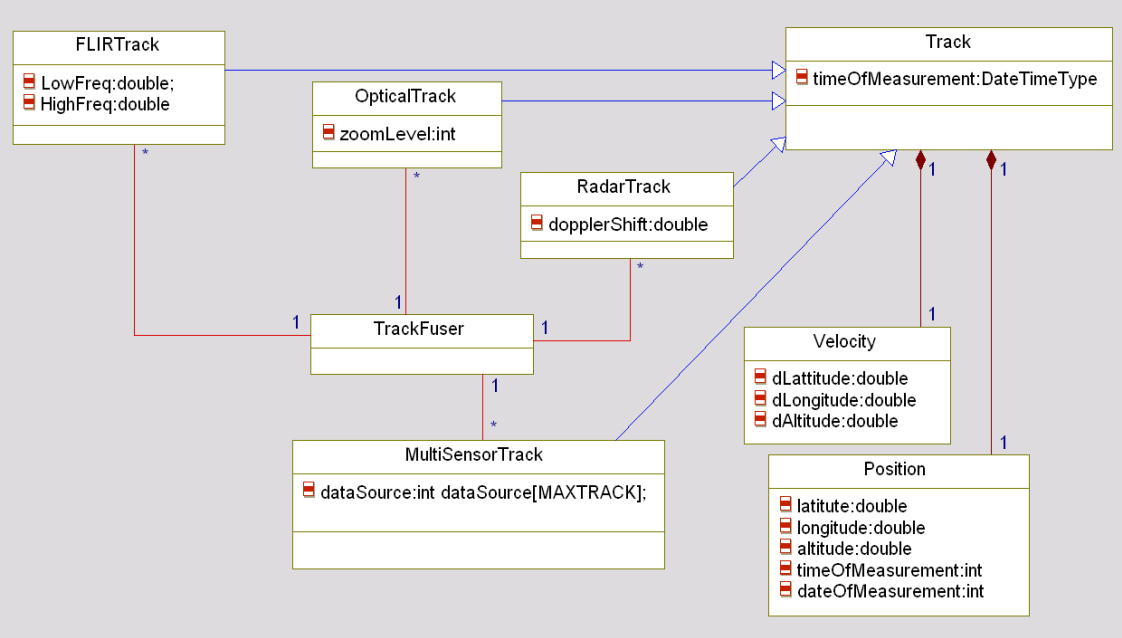


Рис. 1-8: Обобщение для классов

На Рис. 1-8 показан простой пример. Для класса *Track* определен атрибут timeOfMeasurement, содержащий информацию о времени измерения трека, и отношения композиции с классами Velocity и Position. Этот класс является базовым классом для классов специализированных треков, измеряемых тепловизионной системой переднего обзора, оптическим и радарным датчиками. Каждый из этих классов имеет дополнительные данные, специфичные для используемой технологии измерения. Класс *TrackFuser* объединяет данные от трех датчиков,

отслеживающих полет одного и того же самолета в мультисенсорный трек. Поскольку эти производные классы наследуют составлящие базового класса, для них также определены отношения композиции с классами Velocity и Position.

Второе момент, подразумеваемый под обобщением -- это возможность *подстановки*. Это означает, что экземпляр производного класса может везде использоваться вместо экземпляра базового класса. На Рис. 1-8 класс Track может предоставлять своему клиенту некоторый набор сервисов (например, возвращать значение координат, скорости или идентификационного номера самолета). Поскольку MultisensorTrack является производным классом от класса Track, во время исполнения некий класс, использующий класс Track, может в действительности иметь связь и взаимодействовать с экземпляром класса MultisensorTrack. Клиент не знает и не заботиться об этом, поскольку MultisensorTrack наследует все возможности своего базового класса.

1.2.2.5 Зависимость

Все виды ассоциаций и отношения обобщения являются базовыми отношениями, определенными в UML. Однако есть еще несколько типов полезных отношений. Все они имеют название *зависимость*. В UML определены четыре основных типа зависимостей: абстрагирование, связывание, использование и доступа, значение каждой из которых может быть еще более уточнено с помощью определения стереотипов. Например, «refine» (детализует) и «realize» (реализует) являются стереотипами зависимости абстрагирования, а «friend» (друг) – стереотип для зависимости доступа. Все эти специализированные формы зависимостей отображаются отоношением зависимости со стереотипом (пунктирная линия с не закращенной стрелкой).

Вероятно, наиболее часто используемыми стереотипами для зависимостей являются «bind» (связывание), «usage » (использование) и «friend». Несомненно, именно они чаще всего встречаются на диаграммах, однако существуют и другие стереотипы. Для знакомства с полным списком "официальных" стереотипов можно обратиться к спецификации стандарта UML[5].

Стереотип «bind» используется для связывания списка фактических параметров с формальным списком параметров. Он используется для описания параметризованных классов (шаблонов в C++ или родовых сегментов в Ada). Это особенно важно для паттернов проектирования, поскольку сами паттерны проектирования описывают параметризованные взаимодействия, и поэтому они часто определяются с использованием параметризованных классов.

Параметризованный класс – это класс, определенный в терминах более примитивных элементов, с использованием символьных имен и без использования имен фактических элементов, которые будут использоваться в классе. Символьное

[5] Доступен на. www.omg.org

имя называется *формальным параметром*, а фактический элемент после связывания называется *фактическим параметром*. На Рис. 1-9 *NumericParameter* является параметризованным классом, атрибуты и методы которого определены в терминах неопределенного типа *BaseType*. Если мы связываем фактический параметр (например, *double*) для использования вместо *BaseType*, то получаем класс (*NumericParameter_double*) , экземпляры которого мы можем создавать.Таким образом связанный параметризованный класс называется *инстанцируемым классом*, поскольку у нас имеется вся необходимая информация для создания его экземпляров. Зависимость «bind» связывает фактический параметр, *double*, с формальным параметром, *BaseType*.

Для параметризованных классов могут определяться производные классы, как показано на рисунке. Класс *SafeNumericParameter* сохраняет данные дважды, один раз в обычном формате, а другой раз -- с битовой инверсией. Он не является инстанцируемым классом, поскольку для него не определен класс, который будет использоваться вместо *BaseType*. Это делаем это, когда определяем класс *SafeNumericParameter_double*.

На этой же диаграмме показано использование зависимости «friend». Это означает, что у класса *SafetyMonitor* имеется доступ ко всем составляющим класса SafeNumericParameter вне зависимости от их области видимости: private (закрытый), protected (защищенный) или public (открытый). Это как в старой поговорке: "Вы открываете душу только друзьям".

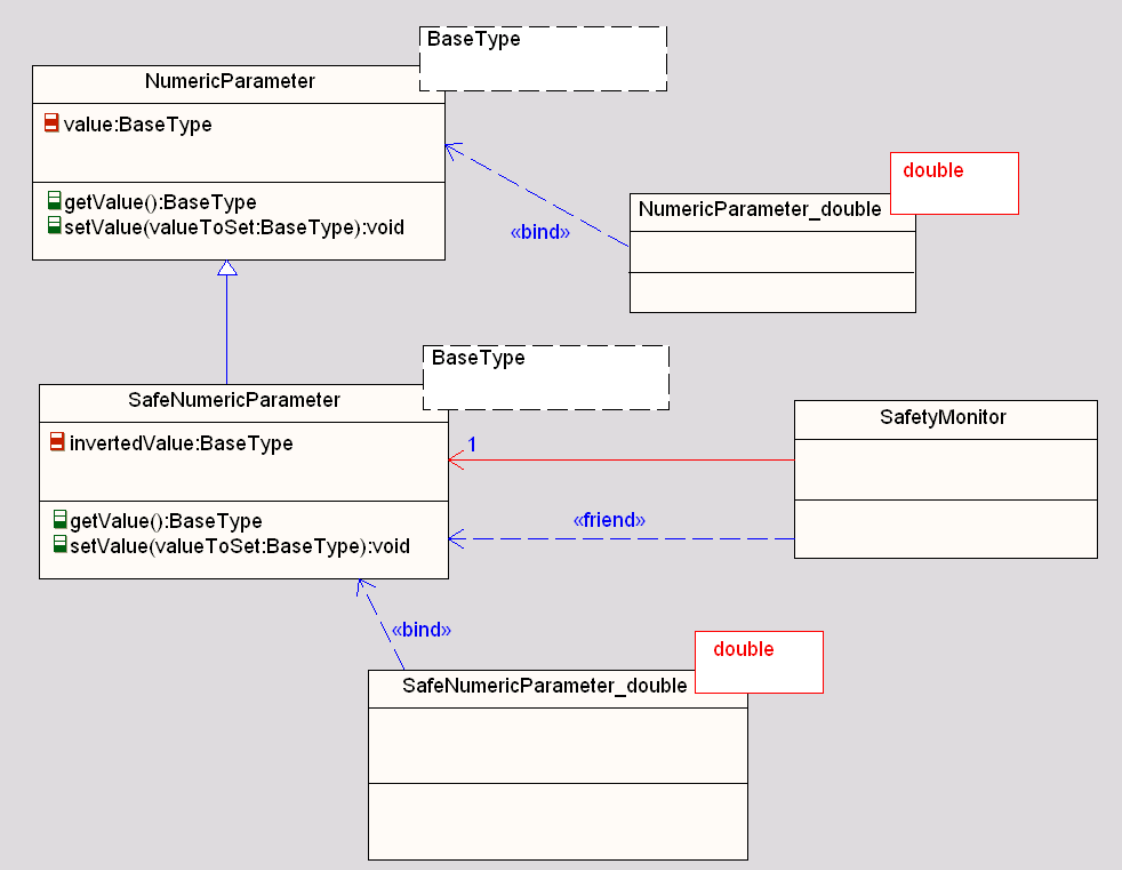


Рис. 1-9: Отношение зависимости

1.2.3 Крупные структурные элементы: подсистемы, компоненты и пакеты

Классы, объекты и интерфейсы являются, по большей части, небольшими элементами. Поведение всей системы в целом определяется кооперациями большого количества таких элементов. В сложных современных системах важно рассматривать более крупномасштабные структурные элементы. UML предоставляет целый ряд сущностей для управления разработкой системы на высоком уровне. Несмотря на это в большинстве литературных источников отсутствуют четкие объяснения и примеры использования этих возможностей. Если быть честным, даже в спецификации UML не приводится доходчивого объяснения данных возможностей и их взаимосвязей.

Одним из таких элементов является пакет. Пакеты используются *исключительно* для организации UML-модели. Они не являются инстанциируемыми элементами и не присутствуют в работающей системе. Пакеты являются элементами моделей, которые могут содержать другие элементы моделей, в том числе и другие пакеты.

Пакеты используются для организации моделей и позволяют коллективу разработчиков эффективно ими манипулировать при совместной работе. Пакеты определяют пространство имен для элементов модели, которые они содержат, и не несут более никакой семантики. Пакеты являются основным организационными единицами для конфигурационного управления – и, как правило, являются основными элементами конфигурации. Когда вы берете или обновляете пакет в системе конфигурационного управления, вы обновляете все элементы, содержащиеся в данном пакете. Rhapsody позволяет уменьшить масштаб элемента конфигурации до размера отдельного класса. Однако большинство разработчиков считают это слишком детальным, так как усложняет процесс изъятия связанных элементов из репозитория.

Как правило, пакет содержит элементы, существующие только во время проектирования – классы и типы данных. Они также содержат варианты использования и диаграммы, такие как диаграммы последовательности и диаграммы классов. Эти элементы проекта затем используются для конструирования кооперацией, реализующих функциональные возможности уровня системы. Рис. 1-10 показывает, что пакеты отображаются в виде папки с ярлыком. При желании для пакетом могут быть отображены элементы, которые они содержат. На этом рисунке для большинства пакетов отображено по одному классу, но обычно пакеты содержат десятки и даже более элементов. Отношения зависимости не являются обязательными, но могут использоваться для отображения заивисимостей компиляции.

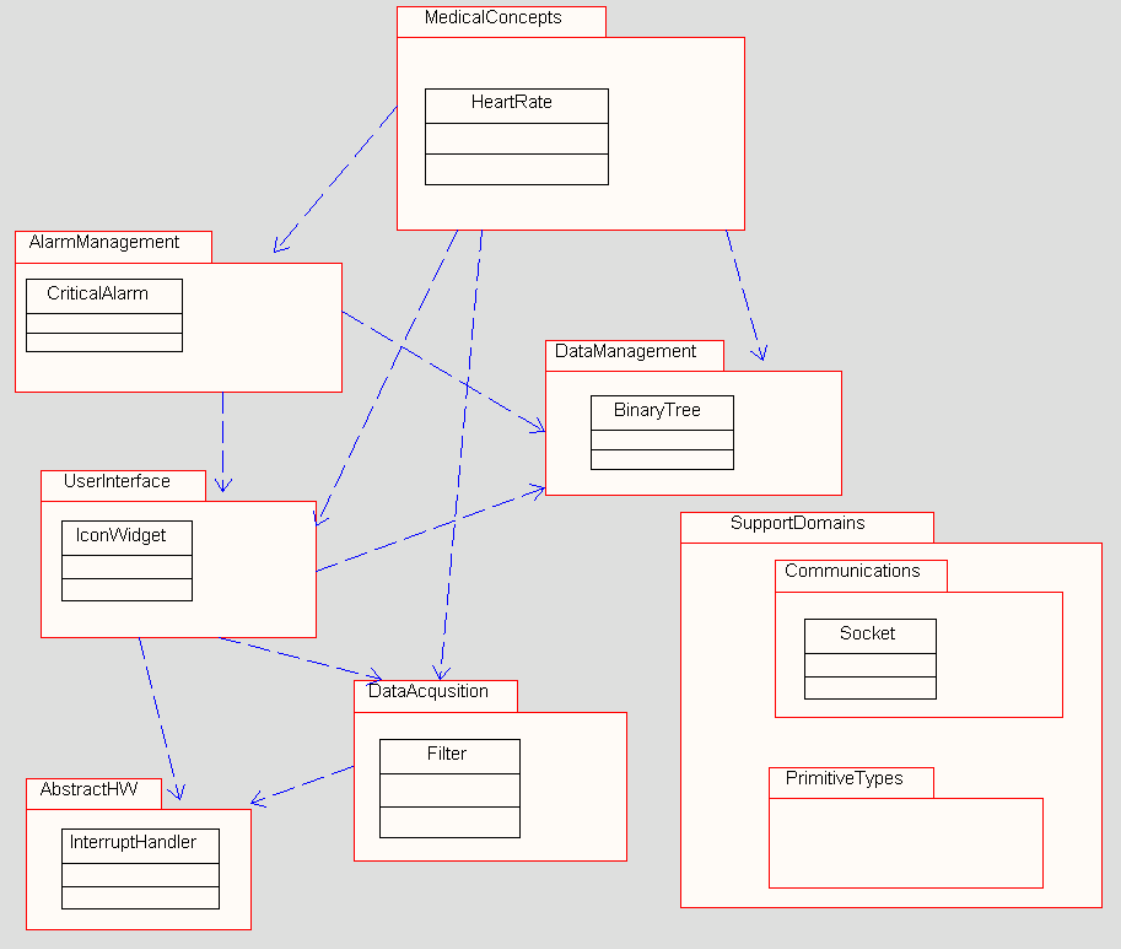


Рис. 1-10: Пакеты

Подсистемы – это не что иное, как классы, точнее, структурные классы. Они являются архитектурными строительными блоками большого масштаба. В отличие от пакетов, подсистемы являются инстанциируемыми. Это означает, что вы можете создать экземпляр типа, который будет занимать память во время исполнения. Подсистемы используются для организации элементов исполняемой системы и состоят из экземпляров. Критерием объединения элементов в подсистему являются "общее функциональное назначение". Реальная работа подсистемы реализуется ее частями; сама же подсистема предоставляет вовне совместное поведение входящих в нее частей. Подсистемы организуют и управляют поведением своих внутренних частей, но вся "реальная" работа выполняется элементами, которые иногда называются *семантическими объектами* системы -- элементарными объектами, реализующими функциональность нижнего уровня. Подсистема находится на более высоком уровне абстракции для системы, чем эти элементарные семантические объекты, и данный уровень абстракции позволяет нам намного более легко представлять и манипулировать структурой и поведением сложных систем.

В UML 2.0 компоненты также являются структурными классами, предоставляющими внешние интерфейсы, независимые от языка программирования. Между компонентами и подсистемами существуют незначительные различия на уровне метамодели, но при использовании компоненты и подсистемы практически идентичны друг другу.

1.3 Элементы и диаграммы для описания поведения системы

До сих пор мы рассматривали структурные элементы системы: классы и объекты (микроуровень), а также системы, подсистемы и компоненты (макроуровень). Однако нас, как разработчиков, в еще большей степени интересует динамическое поведение этих структурных элементов во время работы системы. Мы можем рассматривать поведение с двух различных точек зрения – как структурные элементы ведут себя по отдельности и как они ведут себя в кооперации.

В метамодели UML *ModelElements* являются минимальными структурными элементами, для которых может быть определено поведение. Для *классификаторов* (которые принадлежат *ModelElements*), также определены *поведенческие совтавляющие*, а именно *операции* и реализующие эти операции *методы*. На практике, нас в основном интересует определение реактивного поведения только для некоторых *классификаторов* (классов, объектов, подсистем, компонентов и вариантов использования) и некоторых *элементов модели (действий*, *операций* и *методов*).

1.3.1 Действия и деятельности

Действие – это определение исполняемого выражения, которая является абстракцией вычислительной процедуры, результатом которой является изменение состояния модели, и которая может быть вызвана путем посылки сообщения объекту, либо путем изменения связи или значения аттрибута. Это означает, что действие – это элементарный элемент модели, эквивалентный отдельному выражению в стандартном языке программирования, такму как: "++X" или "a=b+sin(c*PI)". В UML определено несколько разных типов действий, например, создание экземпляра, вызов метода или генерация события.

Хотя UML определяет *семантику действий*, он не определяет синтаксис языка, который должен использоваться для их определения. Некоторые инструментальные средства используют собственный абстрактный язык определения действий, который преобразуются в конструкции целевого языка программирования в процессе трансформации модели. Это предоставляет (как

утверждается[6]) преимущество по переносимости приложений с одного языка программирования на другой. Обратной стороной такого подхода является сложность отладки на уровне исходного кода, и если вы вдруг измените исходный код, связь между моделью и программным кодом безвозвратно теряется. До сих пор большинство разработчиков предпочитает определять действия на целевом языке программирования, таком как C, C++, Java, Ada или каком-либо другом. Это происходит потому, что во-первых, *они уже знают* этот язык. А во-вторых, для этих языков определены стандарты (в отличие от собственных языков определения действий), и их поддерживает множество различных компиляторов на разных платформах. Отладка в этом случае существенно проще, а изменения в коде могут быть синхронизированы с моделью сравнительно легко при использовании соответсвующего инструментального средства.

Для действий определена семантика "выполнения до завершения", что означает следующее: если действие было начато, оно будет выполняться до тех пор, пока не будет завершено. Разумеется, это не означает, что одно действие не может быть прервано другим действием, исполняемым в более приоритетном потоке. При обратном переключении контекста исполнения на исходный поток, выполнение данного действия будет продолжено до полного завершения. Это означает, что если объект выполняет какое-либо действие, это действие будет выполняться до его завершения, даже если этот объект получит событие, указывающее ему выполнить что-то еще. Объект не будет обрабатывать входящие события до тех пор, пока действие не завершится.

Деятельность -- это действие, которое выполняется в то время, когда классификатор находится в некотором состоянии, и прекращается либо после его завершения, либо когда классификатор изменяет свое состояние. Таким образом, для деятельности не определена семантика "выполнение до завершения". Объект, выполняющий некоторую деятельность, может получить событие, приводящее к переходу, выходу из состояния и прекращению деятельности. Таким образом, UML позволяет моделировать на примитивном уровне как прерываемое, так и непрерываемое поведение.

1.3.2 Операции и методы

Операция представляет собой определение поведения классификатора, которое может быть запущено, в то время как метод является реализацией операции. Другими словами, операция является определением метода. Операции запускаются синхронно и соответствуют событиям вызова в метамодели UML. Как вы, наверное, уже догадались, для операций могут быть определены типизированные

[6]Я говорю "как утверждается" потому что на практике генерация кода на выбранном языке с использованием этих инструментальных средств не является такой простой и очевидной задачей, какой она может показаться на первый взгляд; большинство инструментальных средств, работающие таким образом, генерируют нечитабельный и неэффективный код на выбранном вами языке.

параметры и они могут возвращать типизированные значения. Как правило, вызовы операций используются в качестве действий при описании реактивного поведения.

Существует два основных способа моделирования поведения метода. Первый и наиболее распространенный способ состоит в определении всех действий, необходимых для реализации метода на некотором языке определения действий. Очевидно, что этот подход лучше всего подходит для простых методов. Второй подход, о котором будет рассказано далее, заключается в моделировании операции при помощи диаграммы деятельности.

1.3.3 Диаграммы деятельности

В UML 1.x диаграммы деятельности представляли собой не что иное, как конечные автоматы, только с отличным синтаксисом. Другими словами, диаграммы деятельности и конечные автоматы имели одинаковую метамодель поведения. Однако, в UML 2.0 концепция изменилась.

В UML 2.0 исполнение диаграмм деятельности основывается на *семантике исполнения на основе маркеров*. Это означает, что выполнение определенной деятельности начинается тогда, когда эта деятельность получает маркер исполнения от предшествующего ему действия. Когда эта деятельность оказывается выполненной, маркер исполнения передается следующей деятельности в последовательности. Переход от деятельности к деятельности (или от действия к действию) выполняется после завершения предшествующей деятельности – события для этого не используются. Прямоугольники с закругленными краями на Рис. 1-11 отображают действия или деятельности. Стрелки указывают направление потока исполнения. Переход, с жирной точкой на конце, называется *начальным псевдосостоянием*. Он определяет состояние, с которого начинается исполнение при начале работы.

Диаграмма деятельности содержит операторы. Ромбы на диаграмме обозначают выбор на основе проверки сторожевых условий (отображаемых в квадратных скобках). Эти условия представляют собой булевы выражения, без каких либо сторонних эффектов. Переход выполняется, когда сторожевое условия принимает значение «истина». *Альтернативный переход* выполняется только в том случае, если сторожевые условия на всех остальных переходах принимают значение «ложь». Жирная линия с одним входящим переходом и несколькими исходящими называется развилкой. Развилка отличается от выбора тем, что для выбора выполняется только один переход; в то время как, для развилки активизируются *все* исходящие из нее переходы. Таким образом, развилка означает наличие

нескольких логических потоков исполнения, выполняемых одновременно[7]. Слияние (жирная линия с несколькими входящими переходами) соединяет несколько логических потоков в один. Конечное состояние означает окончание выполнения действий.

Действия и деятельности могут принадлежать различным объектам. Объекты могут вызывать методы других объектов, с которыми у них имеются связи. Для распределения действий и деятельностей между объектами обычно используются элементы UML, называемые *разделами*. На рисунке разделы показаны в виде именованных прямоугольников, содержащих действия и деятельности. Имя каждого раздела определяет класс, который предоставляет вызываемый метод.

Еще один новый элемент, появившийся в UML 2.0, – это контакт. Так же как активность (или действие) соответствует функции, контакт соответствует параметру функции. Таким образом, контакты могут отображаться либо для начальных точек переходов (что обозначает передачу данных параметру, определяемому контактом), либо для конечных точек переходов (что обозначает получение данных параметром). Контакты для начальных точкек переходов называются *выходными контактами*, а контакты для конечных точках переходов называются *входными контактами*. На Рис. 1-11 показаны входные и выходные контакты. Для обозначения данных на диаграмме также может использоваться нотация *объект в состоянии*, унаследованная из UML 1.4. Например, таким элементом является "validatedFlightPlan"; его состояние показано внутри квадратных скобок. В UML 2.0, как правило, используются контакты.

Диаграммы деятельности похожи на блок-схемы. Как правило, они используются для моделирования алгоритмов. Поэтому в большинстве случаев они применяются для определения поведения метода класса или поведения варианта использования, реализующего алгоритм.

[7]Я использую термин "логический", поскольку они не обязательно реализуются как потоки ОС. Они могут выполняться последовательно, однако с логической точки зрения они являются параллельными, и разработчик не знает, какова очередность их исполнения. Поэтому с точки зрения очередности исполнения они независимы друг от друга.

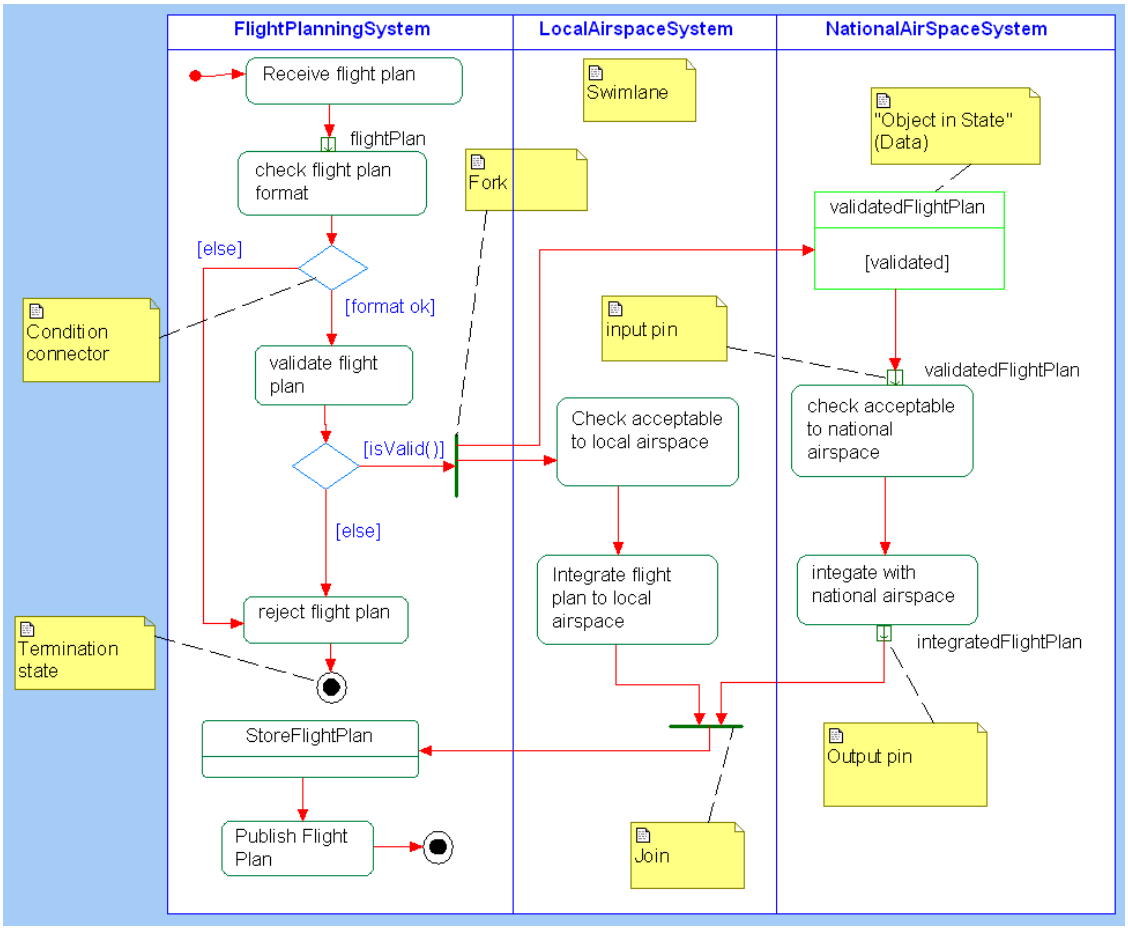


Рис. 1-11: Диаграмма деятельности

1.3.4 Диаграммы состояний

Конечный автомат – это автомат, определяемый конечным набором условий его существования (называемых "состояниями"), а также, конечным набором переходов между состояниями, инициируемыми событиями. Конечный автомат отличается от диаграммы деятельности тем, что переходы инициируются событиями (в основном), а не по заверщению работы в предыдущем состоянии. Диаграммы состояний в основном используются для моделирования поведения *реактивных* элементов, таких как классы и варианты использования, которые находятся в определенном состоянии до тех пор, пока не произойдет интересующее их событие. После этого событие обрабатывается, выполняются соответствующие действия и элемент переходит в новое состояние.

Для моментов входа в состяние, выхода из состояния и при совершении переходов могут быть определены действия для выполнения, такие как запуск операции. Порядок выполнения действий таков: сначала выполняются выходные действия для предшествующего состояния, затем действия при переходе, после чего входные действия для следующего состояния.

В UML для формального представление конечных автоматов используются диаграммы состояний, которые являются более выразительными и лучше масштабируются, нежели "классические" конечные автоматы Мили-Мура. Диаграммы состояний UML, основанные на семантике и нотации[8] диаграмм состояний Дэвида Харела, имеют целый ряд расширений по отношению к классическим диаграммам состояний Мили-Мура, такие как :

- Вложенные состояния для спецификации иерархии состояний
- "И"-состояния для определения логической независимости и параллелизма
- Псевдосостояния для описания семантики поведения общего назначения.

На Рис. 1-12 показаны некоторые из базовых элементов диаграммы состояний, описывающей набор телефонного номера с телефонного аппарата. Она включает базовые "ИЛИ"-состояния и переходы, а также менее простые концепции, в том числе вложенные состояния и начальные псевдосостояния.

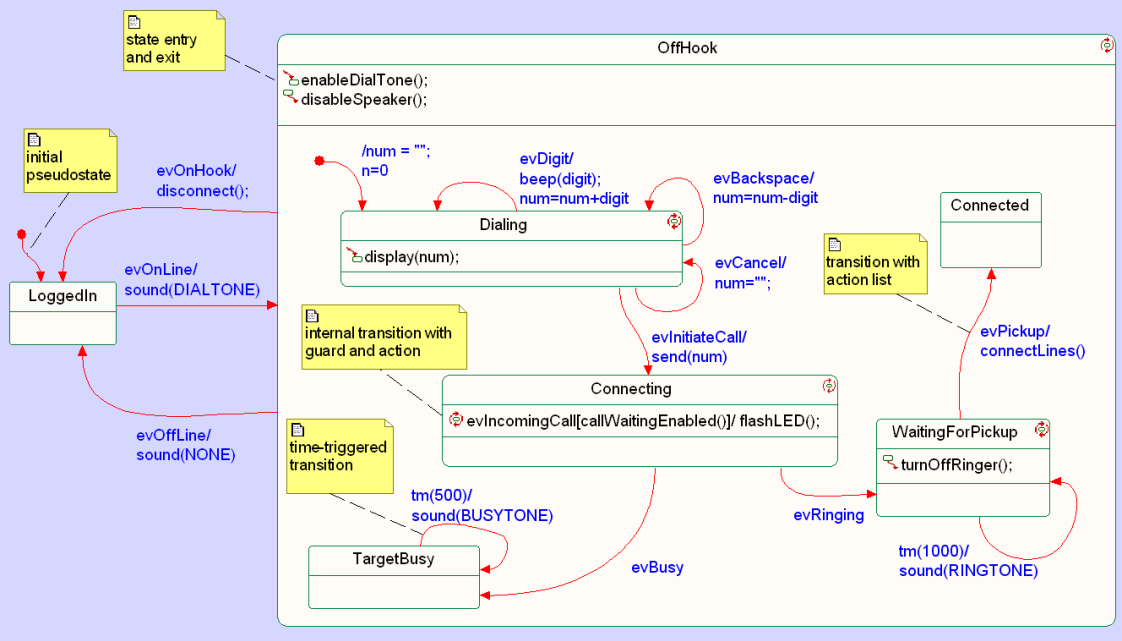


Рис. 1-12: Простой конечный автомат

Переходы изображаются в виде стрелок, выходящих из одного состояния и заканчивающихся в другом. Эти состояния называются соответственно предыдущее и последующее. Для переходов обычно определены необязательные *сигнатура события* и *список действий*. Базовая форма их определения выглядит следующим образом:

[8]Дэвид Харел (Dr. David Harel), предложивший диаграммы состояний, был одним из основателей I-Logix (в настоящее время I-Logix является частью IBM) и внес весомый вклад в создание инструментальных средств Statemate и Rhapsody.

имя события '(' список параметров ')' '['сторожевое условие']' '/' список действий

Имя события -- это просто имя класса определенного для события, которое может быть послано экземпляру классификатора во время исполнения, например, 'evOnHook' или 'tm' на Рис. 1-12. В UML определяются четыре основных типа событий, которые могут быть переданы или обработаны.

- Событие сигнала – событие, передаваемое асинхронно
- Событие вызова - событие, передаваемое синхронно
- Событие времени - событие прошествия заданного интервала времени (чаще всего) или наступления заданного момента времени.
- Событие изменения - событие изменения значения состояния или атрибута классификатора.

Асинхронная передача событий всегда реализуется с использованием очереди событий, в которой событие находится до тех пор, пока элемент не будет готов его обработать. Это означает, что отправитель события "забывает" о нем сразу после отправки, продолжая выполнение своей работы. Ему нет никакого дела, будет ли это событие обработано принимающей стороной. При синхронной передаче обработка события производится в соответствии с текущим состоянием принимающей стороны и выполняется в потоке его отправителя, который при этом блокируется и не может продолжать работу до окончания обработки события. Как правило, данная обработка реализуется путем вызова метода класса, отвечающего за обработку событий, который исполняет соответствующую часть конечного автомата, возвращая управление отправителю только после завершения обработки события. В Rhapsody события вызова называются "триггерными операциями". Они похожи на обычные операции тем что выполняются синхронно, но для них обычным образом не определяется тело метода. Для них реализацией является список действий, определенных в конечном автомате.

Вместе с событиями могут передаваться параметры, которые являются типизированными значениями доступными для конечного автомата. Эти параметры могут использоваться при обработке события в условиях и действиях конечного автомата. Диаграмма состояний определяет лишь список формальных параметров. Объект, отправляющий событие, должен предоставить фактические параметры, соответсвующие списку формальных параметров. Rhapsody использует специальный синтаксис для передачи параметров внутри событий. Для каждого события, содержащего данные, Rhapsody автоматически создает структуру с именем *params*, которая содержит именованные параметры. Например, если у события *e* имеется два параметра -- x и y, то для того чтобы использовать их в сторожевом условии, необходимо воспользоваться указателем params, чтобы получить доступ к значениям этих параметров. Для перехода, инициируемого событием *e* и сторожевым условием определяющим, что x должно быть больше 0 и y не должно превышать 10, синтаксис будет следующий:

```
e[params->x > 0 && params->y <= 10]
```

События времени почти всегда определяются относительно момента входа в состояние. Часто используется синтаксис 'tm(интервал)' для именования таких событий (мы будем использовать данный синтаксис), где 'интервал' - это параметр, определяющий длительность отрезка времени[9]. Если тайм-аут произойдет раньше, чем другие обрабатываемые события, тогда будет выполнен переход, инициируемый событием времени. Если некоторое другое событие будет передано объекту раньше, чем произойдет тайм-аут, то отсчет времени будет остановлен. Если при этом происходит повторный вход в то же состояние, отсчет времени начинается заново.

Если для перехода не определено именованное инициирующее событие, то этот переход инициируется событием "завершения" (или "пустым" событием). Это событие происходит либо после входа в состояние (после выполнения входных действий для состояния), либо когда все деятельности в состоянии завершены.

Сторожевое условие представляет собой булево выражение, указываемое в квадратных скобках сразу после инициирующего события. Сторожевое условие должно возвращать только значения ИСТИНА или ЛОЖЬ и не должно давать никаких сторонних эффектов. Если для перехода определено сторожевое условие и приходит инициирующее событие (если определено), то переход выполняется тогда и только тогда, когда сторожевое условие истинно. Если сторожевое условие ложно, то инициирующее событие просто игнорируется и никаких действий не выполняется.

Действия при переходе, выполняются непосредственно в момент перехода – то есть после того, как объект, находясь в предыдущем состоянии, получает именованное событие и сторожевое условие (если определено) принимает значение ИСТИНА. Действия выполняются в следующем порядке: выходные действия, действия при переходе и затем входные действия, и выполняются в соответствии с семантиков "выполнения до завершения", как уже говорилось ранее. На Рис. 1-12 показаны действия при переходах, а также входные действия для состояния Dialing и выходные действия для состояния WaitingForPickup, а также входные и выходные действия для состояния OffHook.

Кроме входных и выходных действий, для состояний могут быть определены реакции в состоянии, также называемые *внутренними переходами*. Эти действия выполняются когда объект находится в определенном состоянии и приходит инициирующее событие. При этом состояние объекта не изменяется. На Рис. 1-12 с помощью комментариев UML отмечено несколько входных действий, выходных действий и внутренних переходов. В Rhapsody используются небольшие иконки рядом с действиями для обозначения когда данные действия выполняются (на входе, на выходе или как реакция в состоянии).

[9]Часто также используется синтаксис "after(интервал)".

На Рис. 1-12 прямоугольники с закругленными углами отображают состояния. Данные состояния называются *"ИЛИ"-состояниями*, поскольку на любом уровне абстракции для конечного автомата объект может находиться только в одном таком состоянии. Объект Telephone, изображенный на этом рисунке, может находиться либо в состоянии LoggedIn, либо в состоянии Offhook. Это "ИЛИ"-состояния одного уровня абстракции. Для обозначения состояния, в котором объект оказывается непосредственно после его создания, используется начальное псевдосостояние. В Rhapsody во всех конечных автоматах может использоваться макрос IS_IN(имя состояния). IS_IN(LoggedIn) вернет значение ИСТИНА, если объект Telephone находится в состоянии LoggedIn.

Состояние OffHook является *составным состоянием*, содержащим вложенные состояния, такие как: Dialing, Connecting и WaitingForPickup. Внутри состояния OffHook вложенные в него состояния являются "ИЛИ"-состояниями. Это означает, что если объект находится в состоянии OffHook, то он может находиться *только в одном* из этих вложенных состояний. Если объект Telephone находится в состоянии WaitingForPickup, то не только IS_IN(WaitingForPickup) возвращает ИСТИНА, но и IS_IN(OffHook). Это основное правило для вложенных состояний. Например, если вы находитесь в ванной, то, определенно, вы ТАКЖЕ находитесь в доме. В каждый момент времени вы можете находиться только в одной комнате или в одном доме.

Событие evOnLine приводит к переходу объекта в состояние OffHook, что должно также приводить ко входу в некоторое вложенное состояние. Но в какое именно? На этот вопрос отвечает начальное псевдосостояние, определенное внутри состояния OffHook. Оно указывает, в какое из вложенных состояний объект переходит по умолчанию. При желании начальное состояние можно обойти, явным образом определив переход к другому вложенному состоянию. Обратите внимание на то, что для состояния OffHook определены два перехода обратно в состояние LoggedIn. Они определены для составного состояния и происходят всегда, когда OffHook является текущим состоянием, независимо от того, в каком из вложенных состояний (внутри OffHook) находится объект в настоящий момент. В конечных автоматах Мили-Мура, в которых отсутствуют вложенные состояния, для того, чтобы обеспечить аналогичное поведение такие два перехода должны были бы связывать каждое из вложенных состояний с состоянием LoggedIn.

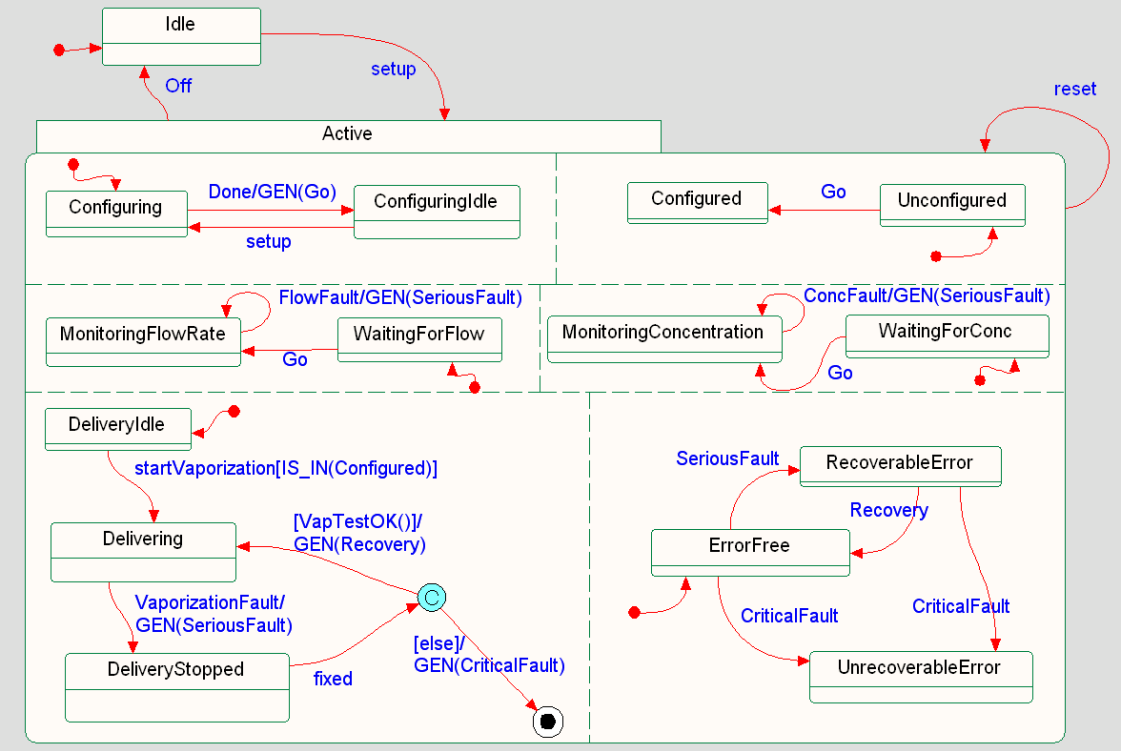


Рис. 1-13: И-состояния

На Рис. 1-13 показаны "И"-состояния - одна из важнейших расширенных концепций для диаграмм состояний. В то время как "ИЛИ"-состояния являются отличимыми и эксклюзивными, "И"-состояния являются отличимыми, но не эксклюзивными. Эти состояния аналогичны логическим потокам на диаграммах деятельности. Как правило, они не реализуются с помощью отдельных потоков операционной системы, тем не менее порядок обработки "И"-состояний совершенно не определен. Приемники событий определяются для всего объекта в целом. Это означает, что если у объект находится в нескольких активных "И"-состояний и этот объект получает событие, то *все* активные "И"-состояния получают копию этого события и могут отреагировать на это событие, либо проигнорировать его, в зависимости от ситуации. Порядок, в котором "И"-состояния обрабатывают копии этого события, не определен. Если необходимо задать определенную последовательность, используйте "ИЛИ"-состояния, а не "И"-состояния. Правила по синхронизации действий между "И"- состояниями, а также конечные автоматы UML, более подробно рассматриваются в третьей главе книги *Real-Time UML 3rd Edition*[10] и главах 7 и 12 книги *Doing Hard Time)*[11].

[10] Douglass, Bruce Powel *Real-Time UML 3rd Edition: Advances in the UML for Real-Time Systems* (Addison-Wesley, 2004)
[11] Douglass, Bruce Powel *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns* (Addison-Wesley, 1999)

1.3.5 Взаимодействия

Совместное поведение многих элементов, работающих совместно, моделируется в UML с использованием *взаимодействий*. UML предоставляет три основных диаграммы для представления взаимодействий: диаграммы коммуникации (в UML 1.x эти диаграммы назывались "диаграммами кооперации"), диаграммы последовательности и временные диаграммы. Диаграммы коммуникации, во многом похожие на диаграммы объектов, дополнительно отображают сообщения между объектами, упорядоченные с использованием нумерации. Диаграммы последовательности отображают роли, исполняемые объектами, в виде вертикальных линий жизни и последовательности обмена сообщениями, упорядоченные на диаграмме сверху вниз. Временные диаграммы отображают изменения состояний или значений с течением времени. Диаграммы коммуникации и временные диаграммы используются сравнительно редко, поэтому далее мы ограничимся только рассмотрением диаграмм последовательности.

Диаграммы последовательности отображают роли объектов, которые могут представлять объекты, подсистемы, системы или даже варианты использования, взаимодействующие во времени. Вертикальные линии на Рис. 1-14, называемые "линиями жизни", отображают объекты (или роли объектов). Объекты (или роли объектов) могут посылать и получать сообщения. Сообщения изображаются на диаграммах в виде линий со стрелками, направленных от одной линии жизни к другой.

Диаграмма последовательности отображает *некий экземпляр* или "пример исполнения" определенной части системы при определенных условиях. Данный экземпляр обычно называется *сценарием*. Сообщения могут быть синхронными (отображаются с помощью закрашенной стрелки) или асинхронными (отображаются с помощью незакрашенной стрелки). Последовательность разворачивается от верхней части диаграммы к нижней ее части. На диаграмме также могут быть отображены состояния и условия для линии жизни, а также ограничения. На Рис. 1-14 справа показано временое ограничение. Более общее ограничение показано прикрепленным к сообщению alarm(Gas_Supply_ Fault). Оно сужает типы отказов, определяемыех этим значением.

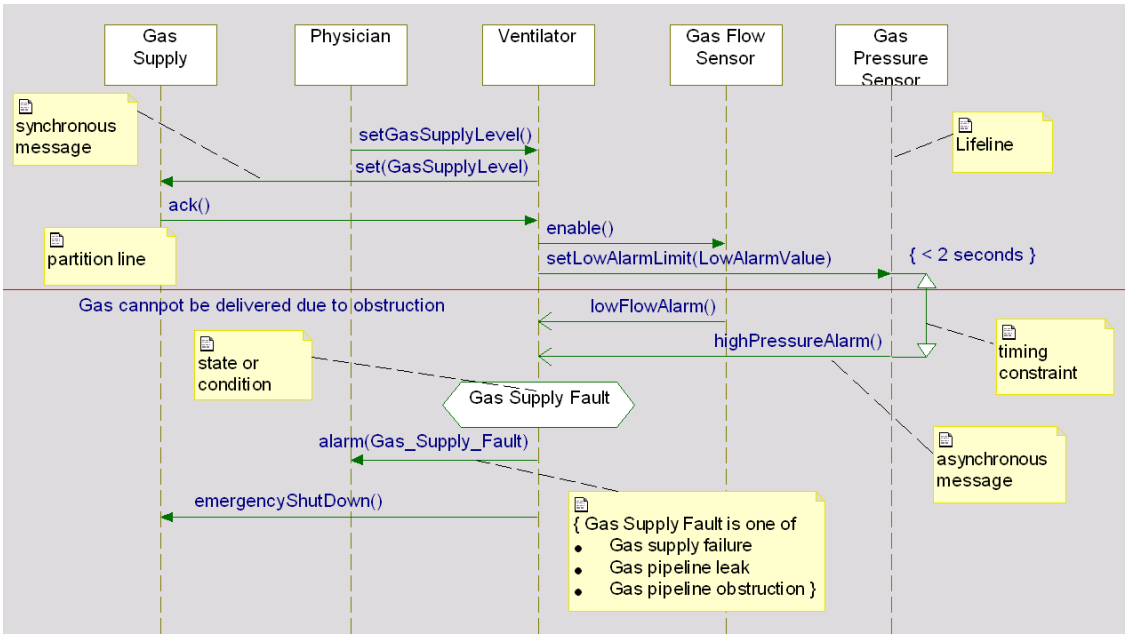


Рис. 1-14: Диаграмма последовательности

В UML 2.0 диаграммы последовательностей существенно расширены по сравнению с более ранними версиями UML. Одно из важных расширений позволяет определять на диаграммах специальные разделы, называемые *фрагментами взаимодействия*. Для каждого фрагмента взаимодействия может быть определен оператор, такой как: loop (цикл), opt (выбор), alt (альтернатива), ref (ссылка), par (параллельный) и т.д. Эти фрагменты взаимодействия и операторы существенно расширяют возможности диаграмм последовательности как инструментов для спецификации взаимодействий. На Рис. 1-15 изображены три фрагмента взаимодействия. В одном из них используется оператор par (параллельный), означающий, что в нем содержатся фрагменты, исполняемые параллельно. Внутри данного фрагмента взаимодействия содержатся еще два вложенных фрагмента с операторами цикла, которые означают, что данные фрагменты повторяются до тех пор, пока не будет достигнуто условие окончания цикла.

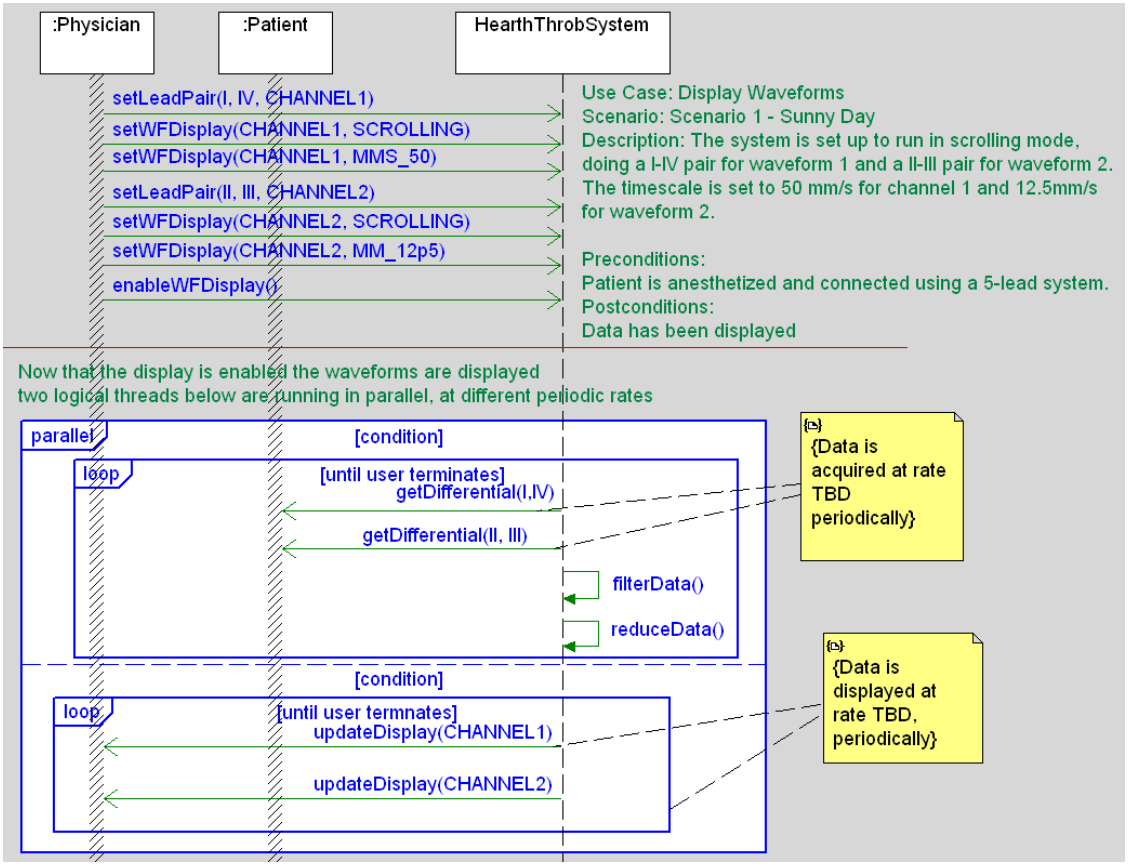


Рис. 1-15: Фрагменты взаимодействия

Я считаю, что наиболее важным расширением для диаграмм последовательности в UML 2.0 является возможность их формальной декомпозиции. Декомпозиция может производиться "горизонтально" при помощи фрагмента взаимодействия с оператором "ref", либо "вертикально" путем определения ссылки от линии жизни к отдельной диаграмме последовательности, отображающей тот же сценарий на более детальном уровне абстракции. На следующих трех рисунках показано, как данный механизм декомпозиции может использоваться.

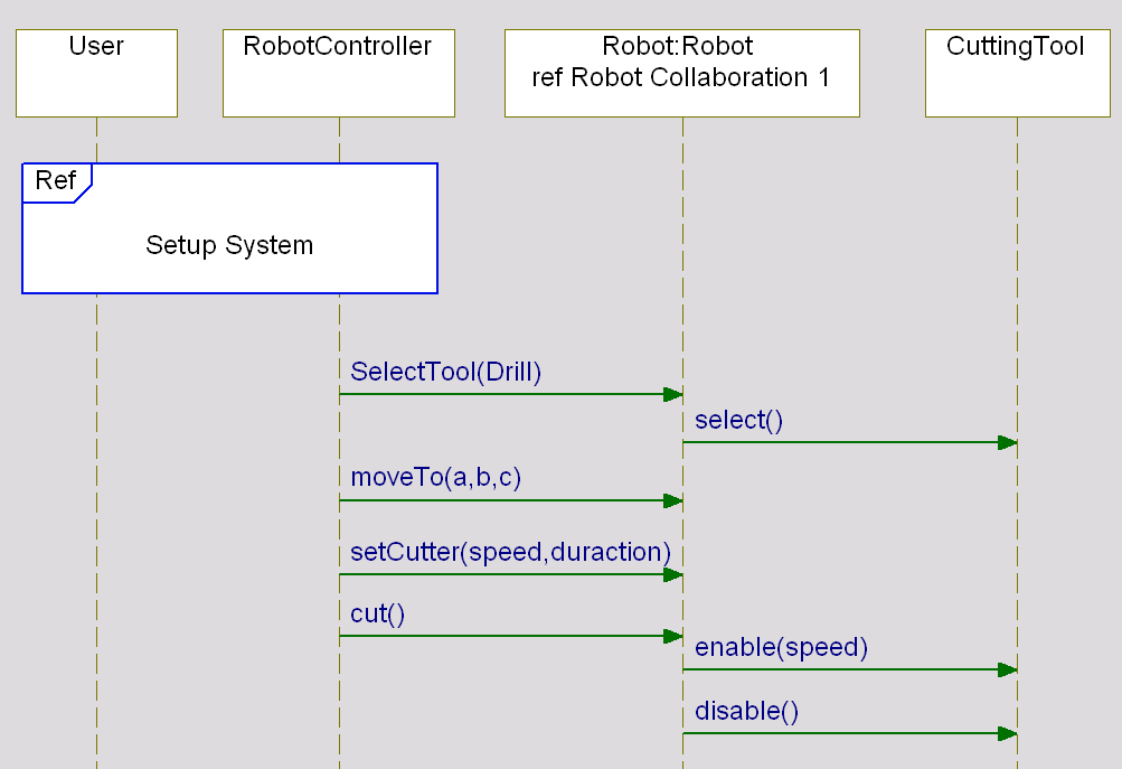


Рис. 1-16: Высокоуровневые диаграммы последовательности

На Рис. 1-16 показана высокоуровневая диаграмма последовательности для промышленной робототехнической системы. Пользователь настраивает систему путем определения плана задач, и затем контроллер отдает роботу команды на выполнение задач, определенных в плане. Робот содержит внутренние части - два поворотных сочленения (которые называются "колено" и "локоть") и вращающийся манипулятор, который может захватывать инструменты и управлять ими. Эта высокоуровневая диаграмма последовательности содержит два фрагмента, ссылающиеся на более детальные взаимодействий. Первый из них является ссылкой на фрагмент взаимодействия "Настройка системы", определенным на другой диаграмме. Если открыть эту диаграмму (в Rhapsody это можно сделать из контекстного меню), то мы увидим диаграмму, показанную на Рис. 1-17.

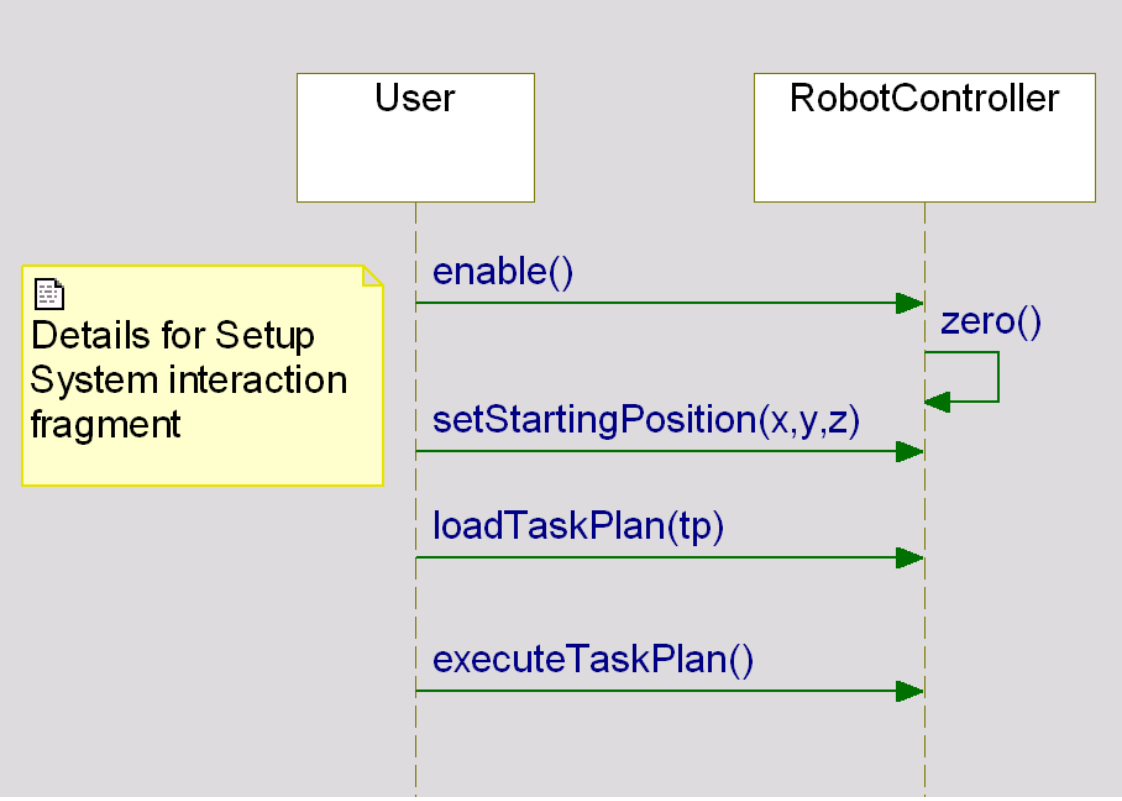


Рис. 1-17: Фрагмент взаимодействия, на который указывает ссылка

Еще более полезной является возможность декомпозиции линии жизни. Данный механизм позволяет увидеть один и тот же сценарий на разных уровнях абстракции, тем самым избавляя нас от необходимости создания единственной диаграммы, перегруженной подробностями. На Рис. 1-18 показано, как взаимодействуют внутренние части робота для исполнения своих ролей в данном сценарии. Линия жизни ENV используется для связи между уровнями взаимодействия. На диаграмме верхнего уровня сообщение, направленное к линии жизни Robot, выходит *из* линии жизни ENV на детальной диаграмме. И наоборот, сообщение, направленное *к* линии жизни ENV, на детальной диаграмме, выходит из линии жизни Robot на высокоуровневой диаграмме последовательности.

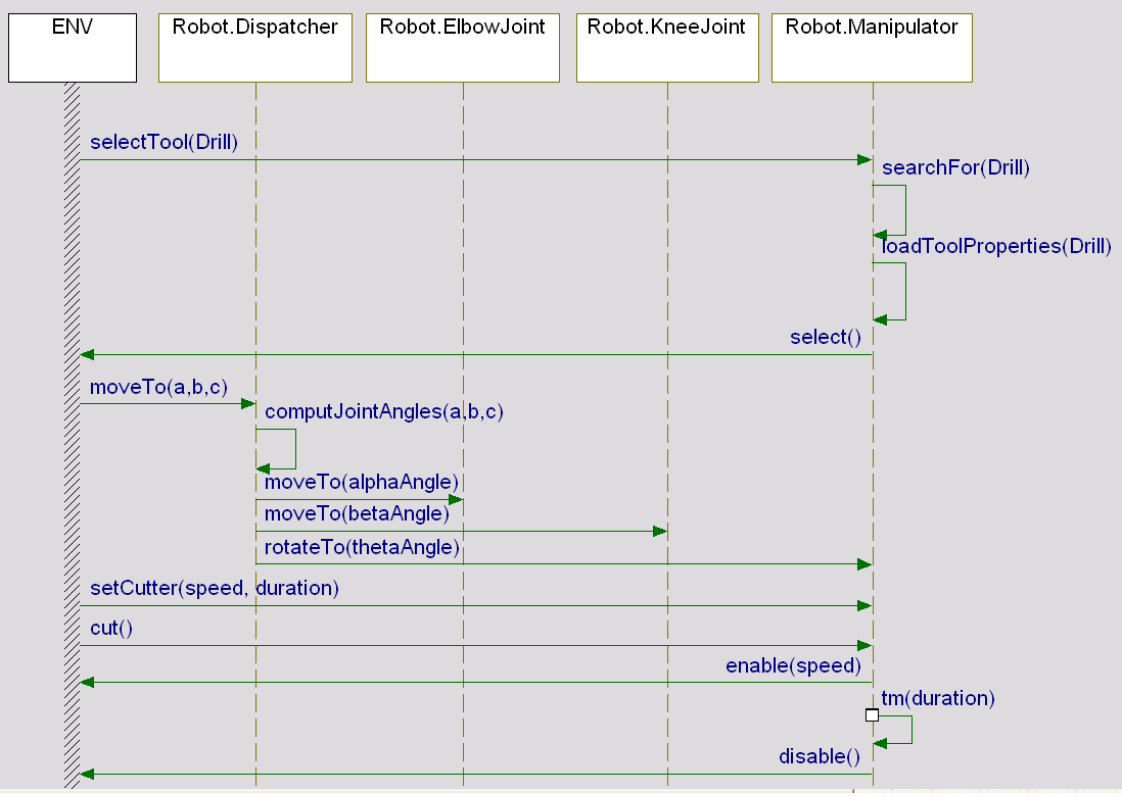


Рис. 1-18: Результат декомпозиции линии жизни

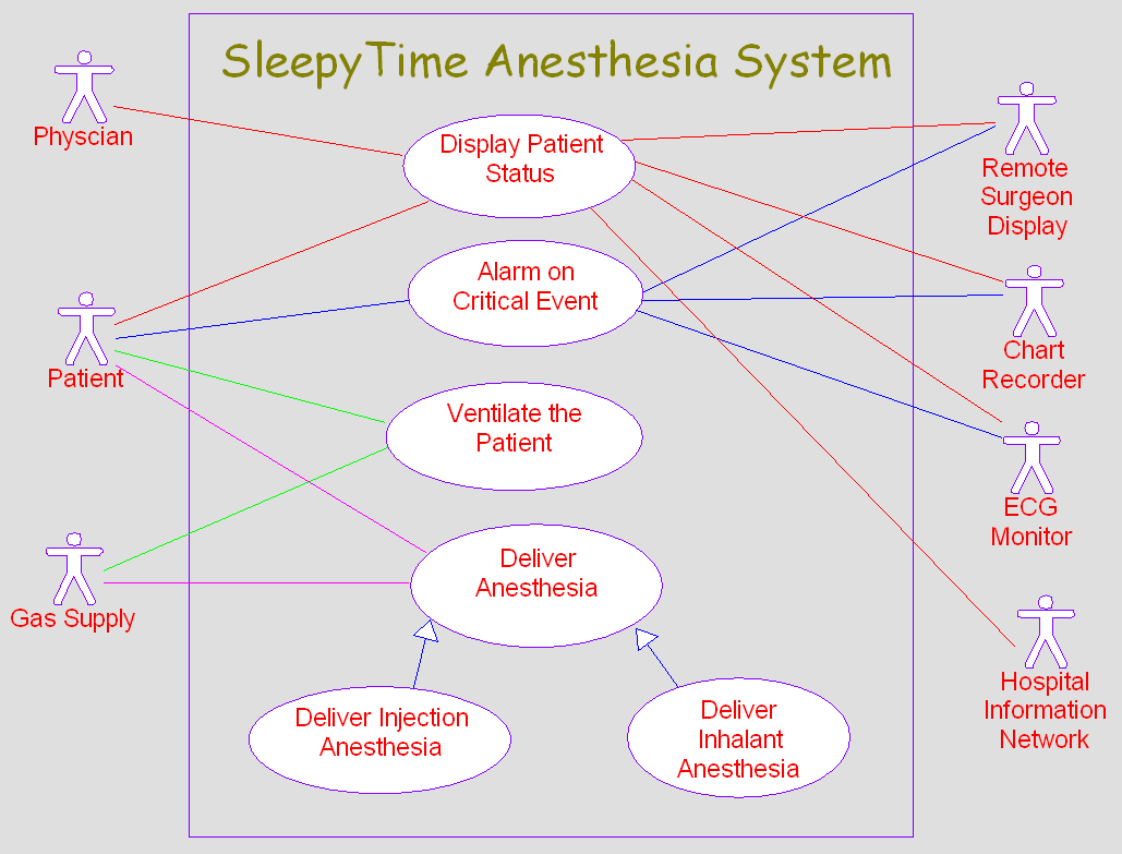
1.4 Модели вариантов использования и требований

Вариант использования – это именованная возможность системы или элемента системы большого масштаба. Он содержит операционные требования и связанных с ними требования к качеству для некоторого функционального аспекта системы. Вариант использования должен предоставлять полезный результат одному или нескольким действующим лицам и не должен раскрывать или подразумевать способов внутренней реализации этой функциональности. В системах среднего масштаба один вариант использования может соответствовать 8 – 20 страницам подробного описания требований к системе.

Существуют два типа требований, обычно определяемых для системы и ее состаных частей: это функциональные требования и так называемые нефункциональные требования, или требования к качеству. Функциональные требования определяют, что система должна делать, например: "Система должна подавать анестезирующий препарат галотан в виде, пригодном для вдыхания". Требования к качеству определяют, *насколько хорошо* должна быть реализована данная функциональность. Например: "Система должна подавать анестезирующий препарат галотан в форме, пригодной для вдыхания, *и поддерживать заданную концентрацию газа данного вещества в пределах 0.5% от объема. Объемная концентрация анестезирующего газа от 0% до максимальной концентрации 10%*

должна достигаться не дольше чем за 10 минут при частоте дыхания пациента 10 вдохов в минуту и разница объема между вдохом и выдохом 600 мл."

Варианты использования изображаются в виде овалов, соединенных отношениями ассоциации с действующими лицами. Что означает, что кооперации, реализующие варианты использования, взаимодействуют значимым образом с данными действующими лицами. Между вариантами использования также могут быть определены отношения, хотя новичкам в моделировании можно посоветовать не злоупотреблять этим видом отношений.[12] Между вариантами использования могут быть определены три вида отношений: обобщение (когда один из вариантов использования является специализированной формой другого), включения (когда один из вариантов использования включает другой для реализации своего назначения) и расширения (когда один из вариантов использования добавляет дополнительные функциональные аспекты другому). На Рис. 1-19 изображена диаграмма использования медицинской системы для выполнения анестезии.



[12]Исходя из моего опыта, достаточно легко начать неправильно использовать отношения между вариантами использования, пытаясь функционально декомпозировать систему, для чего они *не предназначены* . Назначение вариантов использования состоит в том, чтобы определить функциональное поведение системы, подсистемы или другого крупного классификатора способом независмимым от реализации.

Рис. 1-19: Диаграмма использования

Поскольку вариант использования предоставляет только информацию об имени и определенных отношениях с действующими лицами и другими вариантами использования, то где-то должны быть собраны и детальные требования . В процессе Harmony последовательность работ по сбору детальных требований к варианту использования называется "детализация варианта использования". Возможные два подхода к детализации вариантов использования, дополняющих друг друга: на основе примера и на основе спецификации. Независимо от используемого подхода при детализации нельзя ссылаться на элементы внутренней структуры системы, поскольку требования должны быть определены способом не зависящим от реализации.

Варианты использования могут рассматриваться как "контейнеры", содержащие связанные с ними детализированные требования. Как упоминалось выше, эти требования могут быть комбинацией функциональных требований и требований к качеству. Для детализации могут использоваться следующие способы: моделирование сценариев на основе диаграмм последовательности или спецификация вариантов использования на основе конечных автоматов.

Моделирование сценариев для вариантов использования подразумевает создание диаграмм последовательности, которые отображают различные сценарии варианта использования. Каждый из сценариев описывает очень конкретное взаимодействие между системой и действующими лицами. Сценарии определяют сообщения, передаваемыми от действующих лиц системе и от системы к действующим лицам, а также допустимые последовательности этих сообщений.

При моделировании сценариев важно помнить, что целью является сбор требований, а не функциональная декомпозиция системы. Поэтому в данных сценариях могут использоваться только действующие лица и сама система или вариант использования, но не могут внутренние части системы. При детализации вариантов использования для подсистемы, взаимодействующие с ней подсистемы выступают как действующие лица.

Другим подходом по сбору требований является их *спецификация*. Спецификация может содержать сотни и даже тысячи текстовых утверждений, однако текст лучше использовать в сочетании с более формальным способом, таким как диаграммы состояний или диаграммы деятельности, используемые для определения всех возможных сценариев. При использовании диаграмм состояний сообщения, передаваемые от действующих лиц системе, отображаются в виде событий на диаграмме состояний, в то время как сообщения, передаваемые от системы к действующим лицам отображаются в виде действий на диаграмме состояний.

Заключение

Итак, мы закончили краткий обзор UML. Когда речь заходит о UML, то первое, о чем вспоминают люди - это классы и объекты, но возможности этого языка не ограничиваются только этим. UML обладает гораздо более широкими и мощными возможностями. UML также включает описание примитивных элементов поведения – действий и деятельностей, а также средства для спецификации допустимой последовательности их выполнения с использованием диаграмм деятельности и диаграмм состояний. UML предоставляет средства для описания примеров взаимодействия, преимущественно с использованием диаграмм последовательности. И, наконец, варианты использования позволяют сгруппировать требования в удобные для использования, внутренне согласованные функциональные модули. Эта глава не предполагалась быть вашим *единственным* введением в UML. Хотя все основные элементы языка были рассмотрены, мы не вдавались при этом в подробности, а многие полезные возможности UML вообще не обсуждались. Как уже упоминалось ранее, данная книга планировалась как дополняющая книгу *Real-Time UML 3rd Edition*, к которой заинтересованный читатель может обратиться за более детальным рассмотрением UML.

UML – это язык, независящий от используемого процесса. Любой имеющий смысл процесс может быть эффективно использован вместе с UML. Однако не все процессы одинаково эффективны. В следующей главе мы рассмотрим процесс, называемый *процессом Harmony*, который разрабатывался автором, при взаимодействии со многими другими людьми, в течение многих лет. Этот процесс очень эффективен при разработке систем реального времени и встраиваемых систем. Задачи и ответы к ним, которые вы найдете в этой книге далее, будут следовать (в той или иной степени) в соответствии с процессом Harmony, а следующая глава будет посвящена краткому обзору этого процесса.

Содержимое компакт-диска

К книге прилагается компакт-диск, содержащий модели, описываемые в книге, а также дистрибутив Rhapsody для пробного использования. Все модели, рассматриваемые в этой книге, создавались с помощью Rhapsody. Инструкции по установке Rhapsody для пробного использования, а также как получить необходимую лицензию содержатся в файле README.TXT. Вы не обязательно должны использовать Rhapsody для работы над задачами в книге, однако я рекомендую вам делать это.

Rhapsody – это очень мощный инструмент с уникальным набором возможностей. После того, как вы установите его на свой компьютер, я *настоятельно рекомендую* вам, прежде чем начинать работать с моделями, рассматриваемыми в этой книге, потратить некоторое время на изучение этого инструмента и изучить *до конца*

учебное пособие по его использованию, чтобы научиться обращаться с ним, познакомиться с его интерфейсом, научиться запускать и отлаживать приложения и т.п. Чтобы открыть учебное пособие, запустите Rhapsody, затем в меню Help (Помощь) выберите List of Books (Список книг). В списке документов вы увидете документ Rhapsody in C++ Tutorial. Выберите данный документ (либо документ для того языка, который вы собираетесь использовать) и проделайте все задания данного учебного пособия. Это облегчит для вас выполнение упражнений, рассматриваемых в книге. Не волнуйтесь – я подожду, пока вы справитесь с этим и вернетесь обратно! ☺