Моделирование в жизни человека

Часто бывает очень трудно, а иногда и невозможно проследить за поведением реальных систем в разных условиях или изменять эти системы. Решить данную проблему помогают модели -- упрощённые подобия реальных объектов или явлений, обеспечивающий изучение некоторых их свойств. Человек сталкивается с моделями и моделированием в своей жизни постоянно: когда мы прогнозируем исход какого-то действия, объясняем кому-то, как пройти куда-то, или пытаемся формулировать законы природы.

Моделирование позволяет значительно упростить изучение сложных объектов и систем, особенно если их непосредственное исследование затруднено, а то и просто невозможно. Например, модели создают в случаях, когда объект слишком велик или слишком мал, когда изучаемый процесс протекает слишком быстро или слишком медленно, когда исследование реального объекта опасно для окружающих или исследование может повлечь разрушение объекта. В соответствии с этим бывают ментальные модели (когда строится некое описание объекта или явления в сознании человека) и натурные модели (которые воспроизводят внешний вид, структуру и поведение объекта). Вообще моделирование -- это отдельная область знаний со своими методами и правилами, и мы не будем особо вдаваться в классификации. Нас будут интересовать информационные или знаковые модели, которые описывают изучаемый объект или явление с помощью специальных формализмов.

Основные свойства познавательных моделей отражены в следующем определении: «мысленно представленная или материально реализованная система, которая адекватно отражает объект исследования или аналогично воспроизводит специфические свойства и соотношения. Модель должна быть в состоянии представлять объект так, чтобы облегчить его изучение, обеспечить получение об этом объекте новых знаний, составление прогнозов, лучшее управление определенными явлениями или оптимизацию определенных объектов или процессов». Кроме этих основных свойств модели обладают еще и некоторыми специфическими свойствами:

- *редукция или сжатие информации*, заключающиеся в абстрагировании от большого количества параметров оригинала;
- *целенаправленность,* которая позволяет отделить существенные детали от несущественных, то есть то, какие черты сущности или явления будут отражены в моделях;
- *объективность и субъективность,* потому что модели строятся всегда с

определённой целью и проецируют явления объективной реальности под определенным углом зрения;

- *относительность:* при изучении сложных объектов и систем, состоящих из множества взаимосвязанных элементов, создание результирующей модели, которая объединяла бы все аспекты исследуемого явления, зачастую просто невозможно;
- *способность к расширению,* то есть возможность ввода новых знаний в уже построенную модель, что обеспечивает все большее итерационное приближение к моделируемому оригиналу;
- *наглядность или логичность,* поскольку в одном случае модель отображает некоторые признаки оригинала, а в другом она носит абстрактно-логический характер и выражена в абстрактных понятиях и суждениях.

Визуальное моделирование ПО

Как и в других областях человеческой жизни, в разработке ПО модели также очень активно используются, причём самые разные – от ментальных до формальных и даже исполняемых. Если при создании сложной программной системы просто сесть и начать писать код, ничего хорошего из этого не выйдет. Для реализации программ есть удобные формализмы — различные языки программирования, методологии и технологии. Для анализа и проектирования языки программирования, как правило, не годятся, поскольку требуют слишком детального изложения решения задачи.

Под моделью ПО в общем случае понимается формализованное описание системы ПО на определенном уровне абстракции. Каждая модель определяет конкретный аспект системы, а также отражает точку зрения и является объектом деятельности различных людей с конкретными интересами, ролями и задачами.

Наиболее удобными и наглядными являются визуальные модели, где информация об архитектуре системы представляется в виде диаграмм. Наиболее близким аналогом визуальных моделей ПО в "обычной" инженерии являются чертежи — так же, как здание строится по нарисованным архитектором чертежам, так и крупные приложения программируются по нарисованным архитектором визуальным моделям. Кроме того, визуальные модели применяются не только при проектировании, но и при общении между программистами (гораздо проще показать картинку, по которой всё понятно, чем кучу кода), и при общении с заказчиком (заказчик вообще программировать может не уметь, а если ему показать картинку, ему после краткого объяснения нотации всё сразу станет понятно). Ну и по тем же причинам диаграммы — хорошее средство для документации. Ещё есть мнение, что диаграммы должны быть такими, чтобы по ним можно было сгенерировать исполняемый код: не пропадать же уже нарисованным диаграммам. Это спорно, потому что для генерации кода диаграммы должны содержать все необходимые детали, что противоречит самой сути моделирования — упрощению системы. Модель всегда должна быть проще того, что она моделирует, иначе в ней нет никакого смысла. Впрочем, программирование диаграммами (или визуальное программирование) вполне имеет право на жизнь и даже местами применяется, но слово "моделирование" при таком подходе неуместно.

Состав моделей, используемых в каждом конкретном проекте, и степень их

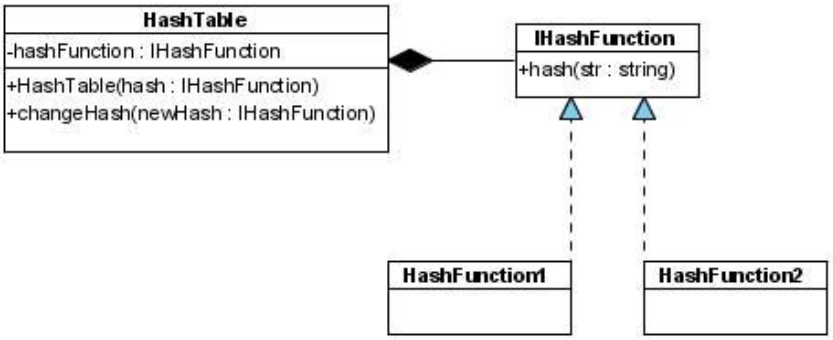
детальности в общем случае зависят от следующих факторов:

- сложности проектируемой системы;
- необходимой полноты ее описания;
- знаний и навыков участников проекта;
- времени, отведенного на проектирование.

Визуальные модели и чертежи имеют одно принципиальное отличие, вытекающее из принципиального отличия объектов физического мира и программ: программное обеспечение невидимо. Здание, деталь, подводную лодку можно нарисовать, так, как мы увидим её глазами (или похоже), программное обеспечение мы не видим. Мы видим текст программы, видим внешние проявления работы программы, но это не сама программа. Как выглядит программа, вообще говоря, непонятно. Поэтому рисовать программные системы можно весьма по-разному. Любая визуализация программного обеспечения представляет собой некоторую метафору — некоторые абстрактные и невидимые сущности программной системы сопоставляются видимым человеческому глазу объектам, всяким квадратикам и кружочкам. В общем-то, точно так же, эти абстрактные сущности сопоставляются тексту на конкретном языке программирования, или даже словесному описанию — это лишь разные способы передать информацию об одном объекте. Из этих соображений, и ещё из того, что модель должна представлять только существенную информацию, следует ещё одно важное для визуального моделирования понятие: точка зрения моделирования. Визуальная модель всегда направлена на конкретную группу читателей, и служит конкретно для чего-то, не бывает просто визуальных моделей (понятно почему — легко заблудиться среди миллионов возможностей визуализации). Визуальные модели бывают трёх типов:

- одноразовые, служащие исключительно для коммуникации (нарисовали диаграмму, потыкали в неё пальцами, изложили мысль и выкинули);
- документация — такая модель, которую можно положить в git или даже повесить на стенку (последний вариант бывает полезен при разработке больших систем, вешаем на стенку диаграмму с высокоуровневой архитектурой системы и стараемся ей следовать при разработке и сопровождении);
- диаграммы, по которым генерируется код, фактически, графические исходники.

Когда вы рисуете диаграмму, вы должны чётко понимать, зачем вы её рисуете. Если вы хотите объяснить кому-то идею решения задачи, вы рисуете только то, что существенно для объяснения, причём для объяснения именно того, что вы хотите объяснить, и именно тому, кому вы объясняете (картинки для директора и для вашего коллеги-программиста могут быть совсем разные). Например, идея решения задачи “реализовать хеш-таблицу с возможностью смены хеш-функции в run-time” может быть выражена такой моделью:

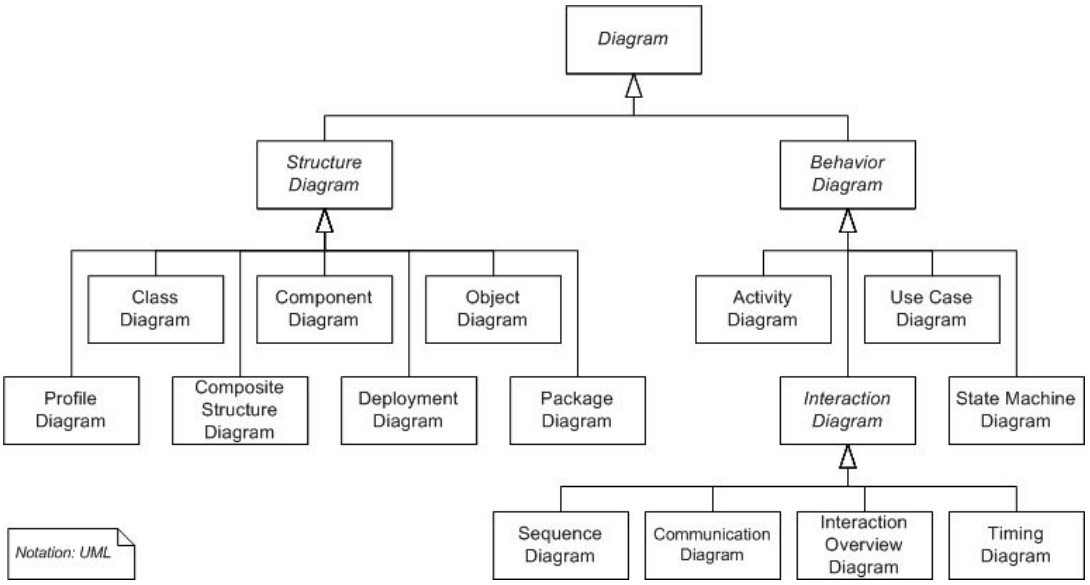


Понятно, что у хеш-таблицы ещё куча методов, что она содержит ещё массив списков и всё такое, но к делу это имеет слабое отношение, а идея решения по этой диаграмме более-менее понятна: мы заводим интерфейс для хеш-функции, реализуем его в классах-настоящих хеш-функциях, и передаём объекты этих классов конструктору хеш-таблицы, или специальному методу смены хеш-функции.

Unified Modeling Language

Понятно, что если рисовать диаграммы уж совсем абы как, от них будет мало пользы в деле передачи информации, потому что у каждого свои представления о том, как выглядит ПО. Требуется некая общепризнанная унифицированная нотация, чтобы незнакомые люди могли обмениваться диаграммами и однозначно понимать, что именно нарисовано. Так вот такой нотации до 1996 года не было, была куча конкурирующих визуальных нотаций. В 1995-1996 годах авторов трёх наиболее популярных из них (Гради Буч, автор метода Буча, Джеймс Рамбо, автор OMT, и Ивар Якобсон, автор OOSE) собрали вместе и велели ругаться, пока они не выработают единую нотацию. В результате они смогли договориться, так появился язык визуального моделирования UML. Так UML находится в процессе стандартизации, проводимом OMG (Object Management Group) -- организацией по стандартизации в области объектно-ориентированных методов и технологий, в настоящее время принят в качестве стандартного языка моделирования и получил широкую поддержку в индустрии ПО.

Та диаграмма, которая показана выше — это пример диаграммы классов UML. Язык UML описывает довольно большое количество различных видов диаграмм (сейчас их 13 штук), которые служат для описания программной системы с различных точек зрения — высокоуровневая структура системы (из каких компонентов она состоит), более низкоуровневая структура (из каких классов состоит система и как эти классы взаимосвязаны), поведение системы, что делает система, как система располагается на физическом оборудовании, как организован обмен сообщениями в различных частях системы, как меняется состояние частей системы в зависимости от времени, и т.д. Диаграммы в UML делятся на две группы — структуры (описывающие статическую структуру программы) и поведения (описывающие различные аспекты поведения системы во время выполнения). Иерархия диаграмм UML:



Мы поговорим про наиболее часто используемые на практике. Начнём с диаграмм случаев использования и активности, которые используются при извлечении и дальнейшей работе с требованиями. Затем обсудим структурные диаграммы классов, пакетов, компонентов, используемые при проектировании архитектуры и для документирования проектных решений в процессе разработки. Ну а под конец рассмотрим диаграммы, которые описывают систему в динамике: диаграммы объектов, последовательностей, состояний, активностей, временные диаграммы. Более подробно используемые на разных этапах разработки диаграммы показаны в таблице ниже.

Уровень разработки	Типы диаграмм
Бизнес-процесс Общее представление о контексте, в котором будет использоваться система, помогает понять потребности пользователей	● Диаграммы активности описывают поток работ между людьми и системами для достижения бизнес-целей. ● Принципиальные диаграммы классов описывают бизнес-понятия, используемые в рамках бизнес-процесса.
Требования пользователей Определение того, что нужно пользователям от вашей системы	● Диаграммы сценариев использования обобщают взаимодействия пользователей и других внешних систем с той системой, которую вы разрабатываете. К каждому сценарию использования можно присоединить другие документы, чтобы подробно его описать. ● Диаграммы классов описывают типы данных, о которые используются при взаимодействии

	пользователей с системой. ● Бизнес-правила и требования к качеству обслуживания могут быть описаны в отдельных документах.
Высокоуровневая архитектура Общая архитектура системы: основные компоненты и их взаимосвязи	● Диаграммы компонентов описывают разделение системы на независимые части. С ними потом можно сверяться, чтобы проверить код программы на его соответствие архитектуре. Они показывают интерфейсы частей, указывая сообщения и службы, которые предоставляются каждым компонентом и необходимы ему. ● Диаграммы последовательностей показывают, как компоненты взаимодействуют для реализации каждого сценария использования. ● Диаграммы классов описывают интерфейсы компонентов и типы данных, передаваемых между этими компонентами.
Шаблоны проектирования Соглашения и методы решения проблем проектирования, которые используются во всех частях процесса разработки	● Диаграммы классов UML описывают структуры шаблона. ● Диаграммы последовательностей или активностей фиксируют взаимодействия компонент и алгоритмы
Анализ кода Из кода можно автоматизированно создать несколько типов диаграмм	● Диаграммы последовательностей показывают взаимодействие между объектами в коде. ● Диаграммы пакетов или компонентов показывают зависимости между классами. ● Диаграммы классов показывают классы в коде.

Моделирование требований

Целями анализа и моделирования требований являются:

- достижение соглашения между разработчиками, заказчиками и пользователями о том, что должна делать разрабатываемая система;
- достижение лучшего понимания разработчиками того, что должна делать система;
- ограничение системной функциональности;
- создание базиса для планирования разработки проекта;
- определение пользовательского интерфейса;
- составление спецификации требований.

В процессе этой деятельности происходит создание следующих документов.

Предварительное соглашение – текстовый документ, который описывает, что будет включено в систему и что решено исключить, то есть, он ограничивает системную функциональность. В нем отражаются пожелания заинтересованных лиц, а также указываются основные нефункциональные требования (например, описывается платформа реализации, точность вычислений, время отклика).

Модель требований служит для достижения соглашения между заказчиком и разработчиками, давая возможность заказчику убедиться в том, что система будет делать то, что они ожидают, а разработчикам создать то, что требуется. Модель требований позволяет, во-первых, установить иерархию требований, что способствует лучшему пониманию человеком, во-вторых, дает наглядное графическое представление требований и зависимостей между ними, в третьих позволяет связать графическую форму представления с текстовой. Модель требований помогает:

- сосредоточиться на внешнем поведении системы независимо от ее внутреннего строения;
- описать потребности пользователей и заинтересованных лиц более однозначно, чем с помощью естественного языка;
- уменьшить число пропусков и несоответствий в требованиях;
- упростить реагирование на изменения требований;
- планировать последовательность разработки функций;
- использовать модели в качестве основы для системных тестов, устанавливая четкое отношение между тестами и требованиями. При изменении требований это отношение помогает правильно обновлять тесты. Это позволяет обеспечить соответствие системы новым требованиям.

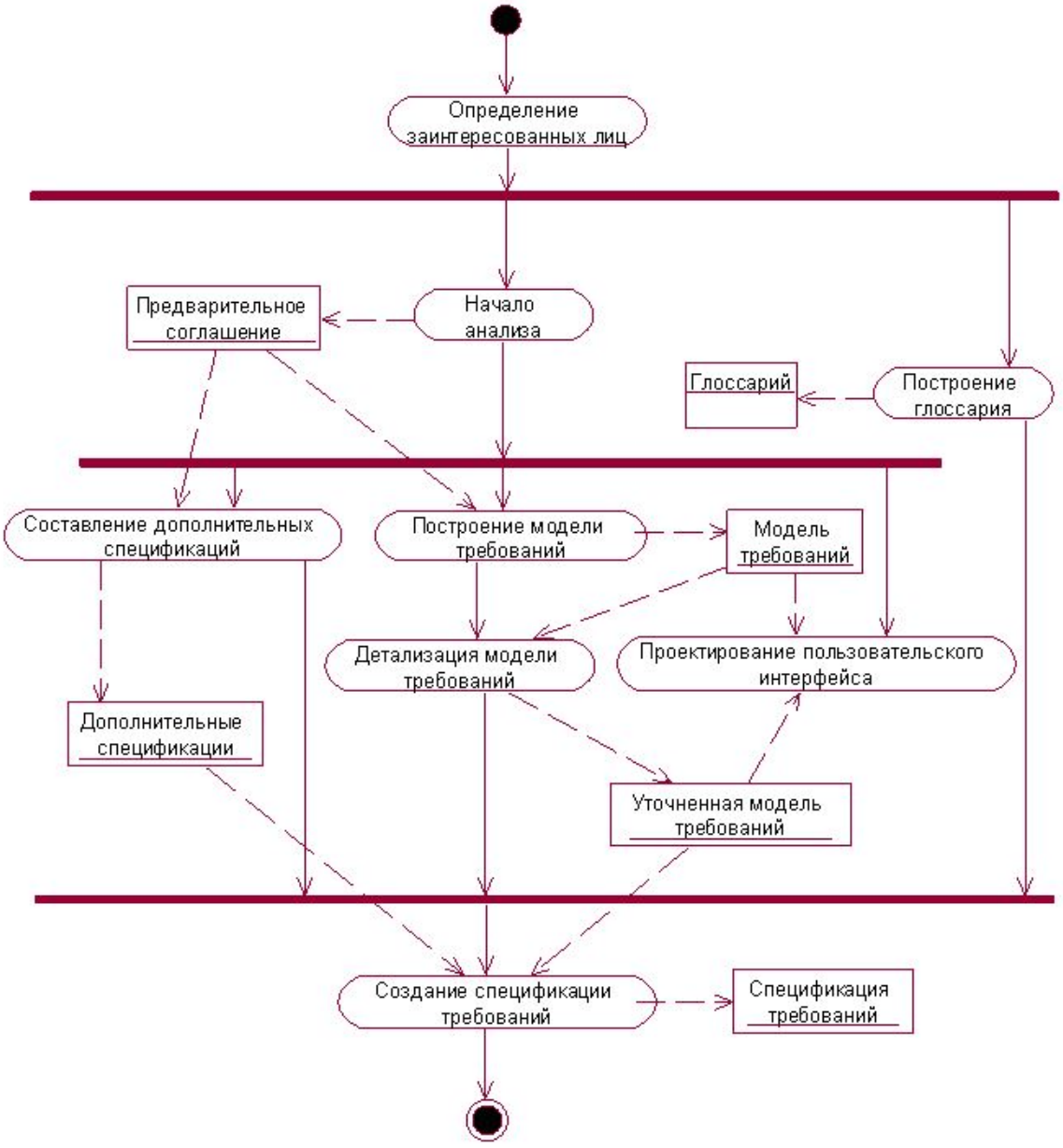
Такого рода модель может создаваться один раз и навсегда, а может разрабатываться итеративно и регулярно пересматриваться. В таком случае её удобно использовать как источник тем для разговоров с заказчиком или пользователями и как инструмент эффективного свода воедино результатов обсуждений. Ещё стоит отметить, что для небольших проектов совершенно необязательно оформлять модель требований в подробностях до создания кода. Частично работающее приложение, даже если оно сильно упрощено, может быть достаточно хорошей основой для плодотворного обсуждения требований с пользователями.

Прототип пользовательского интерфейса обеспечивает визуальное представление интерфейса пользователя.

Глоссарий – текстовый документ, содержащий определения основных понятий и терминов, которые должны одинаково пониматься заказчиком и разработчиком.

Спецификация требований (Software Requirements Specification) – основной документ, используемый при проектировании и разработке системы. Она включает модель требований и дополнительные спецификации, которые представляют собой текстовое описание требований к конечному продукту, но не к процессу его разработки и не содержат деталей реализации требований.

Деятельность по анализу и моделированию требований можно разбить на следующие этапы.



Сначала следует определить всех возможных заинтересованных лиц и разбить их на группы. По каждой группе далее собираются пожелания к будущей системе. Для этого проводят анкетирование, интервью, наблюдения, изучают внутренние и нормативные документы, проводят семинары и многое другое. Эти пожелания анализируются, определяются основные свойства и границы системы, достигаются соглашения о том, какие проблемы должны быть решены.

По завершении этапа должен быть составлен документ «Предварительное соглашение», который будет являться отправной точкой для выполнения всех последующих работ. На этом этапе начинается создание глоссария. Поскольку здесь речь идет о выявлении требований к программной системе, в глоссарий могут включаться термины, относящиеся к реализации. Определения этих терминов должны способствовать лучшему пониманию системы заинтересованными лицами.

Построение модели требований

Этот этап предполагает выявление актёров, сценариев использования и взаимодействий между ними, отражающих функциональные требования к системе с точки зрения пользователя.

Актёр -- это кто-то или что-то вне системы, влияющий на систему или находящийся под её влиянием. Актер может быть человеком, устройством, другой системой или подсистемой. Человек в реальном мире может быть представлен несколькими актерами, если у них есть несколько различных ролей и целей по отношению к системе. Они взаимодействуют с системой и производят над ней некоторые действия. Типовые примеры актёров:

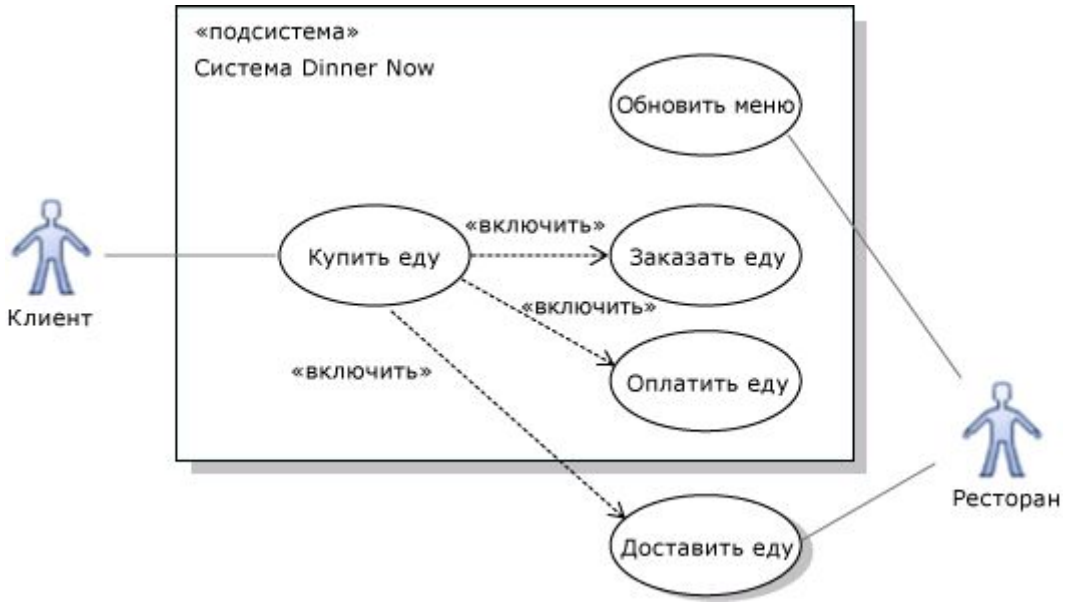
- люди или программные системы, которым проектируемая система требуется для выполнения их задач;
- люди или программные системы, которые необходимы для осуществления проектируемой системой своих функций;
- люди или программные системы, взаимодействующие с внешними программными и аппаратными интерфейсами проектируемой системы;
- люди или программные системы, выполняющие вспомогательные функции администрирования и поддержки.

Сценарий или случай использования представляет собой действие или последовательность действий, выполняемых системой в ответ на некоторое внешнее событие (инициируемое актёром). Или, более концептуально, сценарий использования представляет цель пользователя системы и процедуру, выполняемую пользователем для достижения этой цели.

Диаграмма сценариев использования является самым общим представлением функциональных требований к системе и не показывает порядок выполнения шагов для достижения цели каждого из сценариев использования. В дальнейшем эти диаграммы могут раскрываться в описание алгоритмов или потока событий при помощи других диаграмм (например, диаграмм активностей) или в других документах.

Например, система продажи продуктов питания через Интернет должна позволять клиентам выбирать элементы меню, а ресторанам-поставщикам -- обновлять меню. Это можно объединить в схеме сценариев использования.

Также можно показать, как сценарий использования составляется из более мелких сценариев. Например, заказ продуктов питания -- часть процесса покупки, который также включает оплату и доставку.



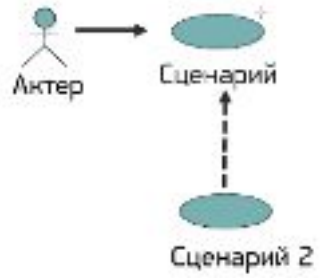
Кроме того, можно показать, какие сценарии использования включены в область разрабатываемых подсистем. Например, изображенная на рисунке подсистема не входит в состав сценария использования "Доставка еды". Это помогает задать контекст для разработки. Это также можно использовать при обсуждении, что будет включено в последующие релизы. Например, можно обсудить, будет ли компонент "Оплата еды" в первоначальном релизе системы функционировать непосредственно между рестораном и клиентом (а не обрабатываться в системе). В этом случае в начальном выпуске можно переместить компонент "Оплата еды" за пределы прямоугольника системы Dinner Now.

Если число сценариев использования слишком велико, то для упрощения читаемости и поддержки модели целесообразно разделить их по пакетам. Это также упрощает понимание модели и распределение ответственности путем назначения разработчиков, ответственных за пакеты сценариев использования. Пакеты позволяют организовать иерархию требований. Верхний уровень иерархии обычно определяется, исходя из основных видов деятельности организации. Пакетов верхнего уровня не должно быть много. Целесообразно выделить пакет администрирования и пакет вспомогательных действий, и 2-4 пакета, основных видов деятельности. В свою очередь, пакеты верхнего уровня могут включать пакеты следующего уровня и т. д.

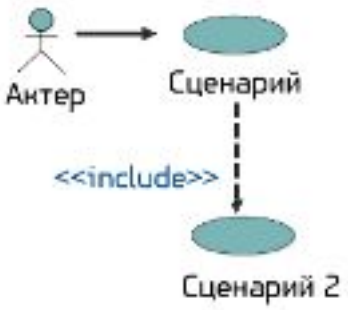
Сценариям могут быть назначены приоритет: критичный, важный и вспомогательный. Критичные сценарии представляют основную системную функциональность или имеют существенное архитектурное значение. Важные сценарии определяют такие системные функции, как сбор статистики, составление отчетов, контроль и функциональное тестирование. Если они не реализованы, то система может выполнять свое предназначение, но со значительно худшим качеством сервиса. Вспомогательные сценарии определяют дополнительную функциональность системы.

Между сценариями использования возможны следующие отношения:

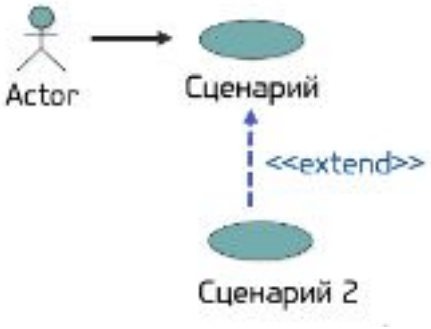
- зависимость: реализация сценария 2 зависит от реализации сценария 1;



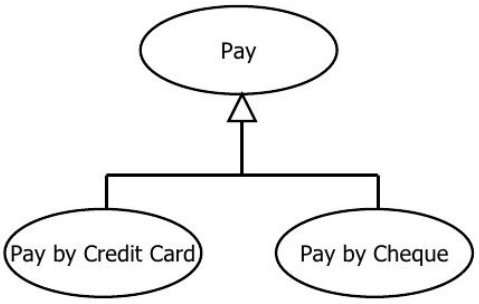
- включение: сценарий 2 полностью включён в сценарий 1, позволяя достичь дополнительную цель. С помощью этого отношения выделяют часто повторяющиеся сценарии;



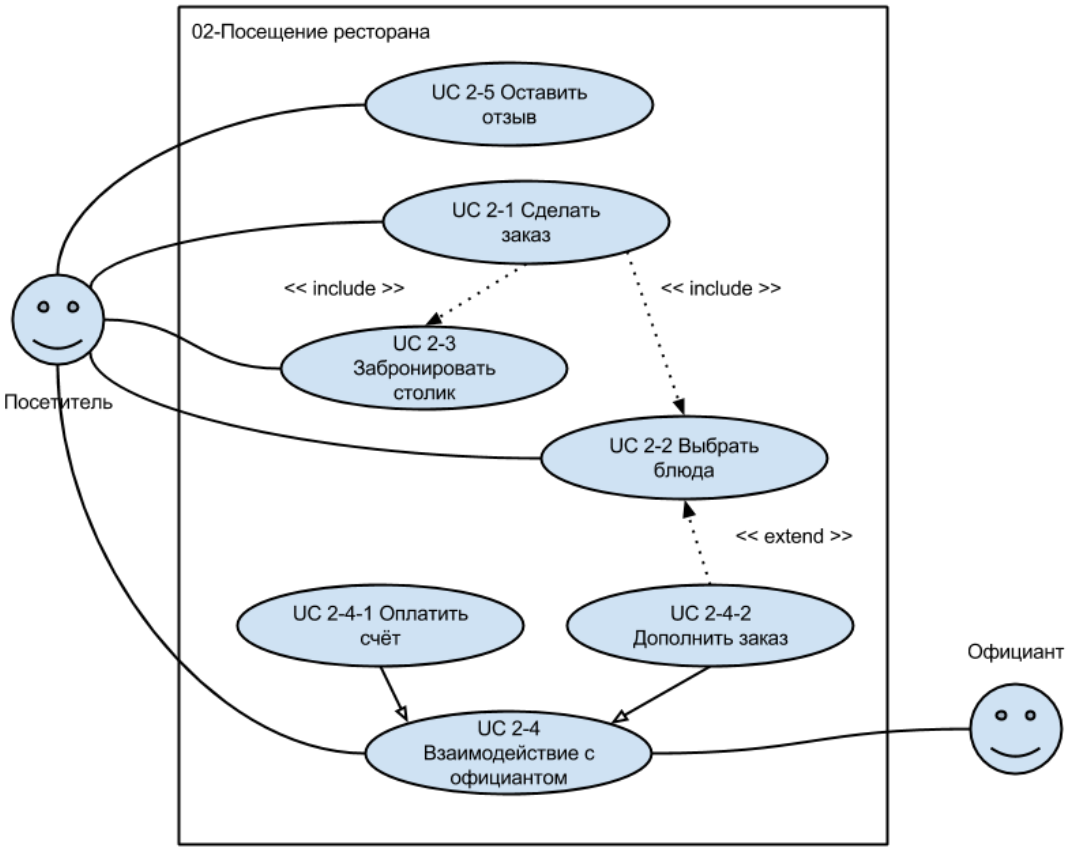
- расширение: сценарий 2 дополняет сценарий 1. Расширяющий сценарий использования часто создаёт какую-то дополнительную ценность.



- наследование: сценарий-наследник реализует какой-то способ достижения цели, определяемой сценарием-родителем.



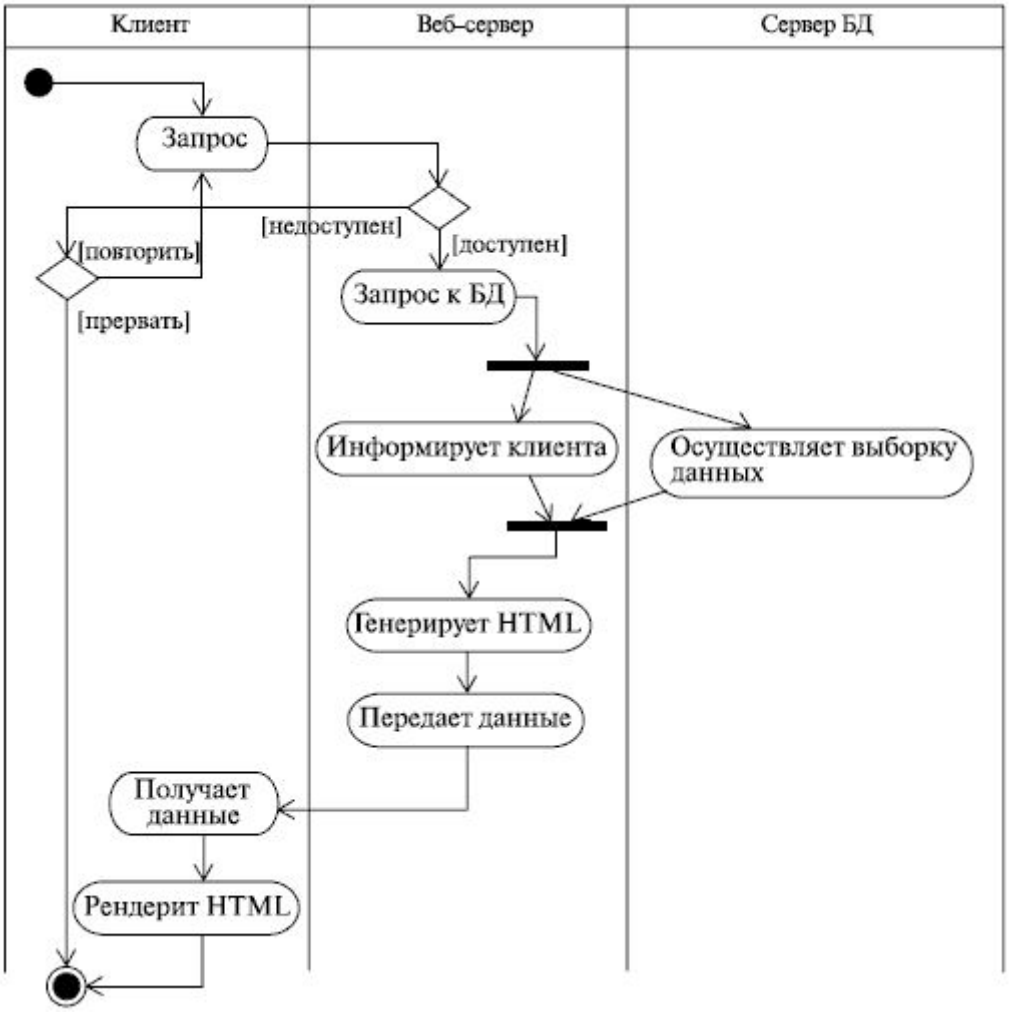
Например, диаграмма сценариев использования для функционального блока “Посещение ресторана” системы бронирования мест в ресторане могла бы выглядеть так:



Разработка этой модели требований тоже может происходить итерационно, уточняя и детализируя модель на каждом шаге.

Когда модель требований будет готова, нужно проверить, что она покрывает все заявки заинтересованных лиц, представленные в документе «Предварительное соглашение».

Диаграмма случаев использования даёт нам представление о том, **что** должна делать система, а на вопрос **как** мы можем ответить, используя диаграммы активности. То есть если сценарии использования ставят перед нами цель проектируемой системы, то диаграммы активности описывают последовательность действий, необходимых для ее достижения. Пример диаграммы показан ниже.



Более подробно про синтаксис диаграмм активности можно почитать, например, тут.

Спецификация требований

Помимо модели требований в ходе анализа требований создаются и дополнительные спецификации -- текстовые описания требований. Они дополняют модель требований и наряду с ней включаются в итоговый документ – спецификацию требований к ПС. Следует четко понимать, что каждый сценарий использования есть некоторое иерархическое требование к системе. Очень важно, чтобы между моделью требований и описанием требований в спецификации было точное соответствие. Если сценарий простой, то в спецификации он описывается как отдельное функциональное требование. В более сложных случаях сценарий может быть разбит на несколько более простых (на диаграммах это можно сделать с помощью пакетов, введя еще один уровень иерархии, в спецификации – ввести следующий уровень нумерации).

Также в спецификацию требований включается описание пользовательского интерфейса системы. Это нужно для того, чтобы заказчик мог более точно представить себе работу и возможности будущей системы и выдать свои замечания и уточнения требований. В зависимости от сложности проекта и уровня подготовленности заказчика результаты этих работ могут быть представлены в

различных формах:

- программная реализация, воспроизводящая точный вид экранных окон;
- набор экранных форм;
- модель навигации экранов в виде диаграмм классов с указанием атрибутов-полей и операций-кнопок.

Финальная спецификация требований утверждается руководством заказчика и разработчика и служит основным отправным документом для проектирования и разработки. В частности, модель требований, входящая в нее, в дальнейшем будет развита в модель анализа и дизайна.

CASE-инструменты

С начала 1980-х годов в активное обращение входит термин CASE-инструмент (Computer Aided Software Engineering), изначально используемый для описания любых инструментов, используемых для автоматизации разработки ПО: редакторов текста и документации, компиляторов и кодогенераторов, отладчиков, хранилищ данных и репозиториев и т.п. С развитием возможностей вычислительной техники и развитием подходов и методологий разработки ПО появляются инструменты поддержки сбора требований, анализа и дизайна, отладки и эмуляции создаваемых систем, поддержки итеративности процесса разработки и многое другое. Со временем в дополнение к текстовым инструментам появляются графические инструменты и методологии их использования, возросшая сложность и многочисленность которых приводит к тому, что с 1990-х годов под CASE-инструментами уже понимают не просто любые инструменты, используемые для разработки компьютерных программ, а инструменты (и даже целые инструментальные системы), автоматизирующие определенные этапы какой-либо методологии создания ПО. Появляется понятие CASE-инструментария или CASE-пакета (CASE workbench), т.е. программного средства, интегрирующего в себе несколько CASE-инструментов: например, в случае текстовых языков, это может быть среда разработки, в состав которой входят текстовый редактор с подсветкой синтаксиса, компилятор и отладчик. В целом считалось, что использование CASE-инструментариев помогает упорядочить и стандартизировать процесс разработки, тем самым снизить время и затраты, затрачиваемые на создание ПО, повысив его качество, иметь единый источник знаний о проекте, автоматически получать документацию о системе по создаваемым моделям, повышать тестируемость и сопровождаемость системы и т.п. Однако также активно обсуждались и недостатки существующих инструментариев, большей частью ориентированность на определенные этапы разработки, сложность интеграции друг с другом для формирования единого процесса, сложность в обучении и общее неудобство использования. С появлением инструментальных сред, поддерживающих полный цикл процесса разработки ПО в соответствии с выбранной методологией, в оборот входит термин CASE-среда или CASE-окружение (CASE environment). Соответственно, подходы к разработке, подразумевающие активное использование подобных инструментов, называют CASE-подходами. Подобная терминология сохраняется и по сегодняшний день.

В 1990-х годах в попытке упорядочить и взять под контроль ход разработки ПО популярность набирают идеи о моделировании деятельности по созданию

программных систем в виде процесса, вовлекающего работников компании в деятельность по достижению определенных бизнес-целей компании. При этом CASE-инструменты видятся как подходящее средство для автоматизации и формализации отдельных этапов процессов. В отсутствии единого понимания набора и состава таких этапов многие компании пытаются выстраивать процессы, исходя из собственных особенностей и потребностей, что приводит к существенному росту количества создаваемых CASE-инструментов, большей частью несовместимых друг с другом. Ситуация меняется с появлением языка UML, разработанного консорциумом OMG для унификации представления моделей ПО. Появляются подходы, использующие UML в качестве основного языка моделирования (например, RUP или MDA), и инструментальные средства для реализации данных подходов в рамках производственного процесса (первым инструментарием, поддержавшим UML, был Rational Rose, ставший весьма успешным коммерчески).

Состав типовой CASE-среды

В настоящее время существует более сотни различных CASE-инструментариев, различающихся как лицензией распространения, границами своей применимости, так и качеством реализации и удобством использования, однако набор инструментов, входящих в их состав, в большинстве случаев совпадает.

- Набор редакторов, с которыми работает пользователь среды, создавая модели разрабатываемого ПО. Чаще всего используются графические языки, модели на которых представляются набором диаграмм, однако нередко в состав CASE-среды входит редактор табличных данных или текстов (для задания документации, правки скриптов, конфигурационных файлов и т.п.). Для ряда инструментов также характерно наличие редактора форм, позволяющего задавать пользовательский интерфейс разрабатываемого приложения.
- Репозиторий, являющийся центральным хранилищем создаваемых моделей и предоставляющий доступ к ним всем остальным компонентам среды.
- Набор инструментальных средств, осуществляющих генерацию целевых артефактов разработки по создаваемым моделям. Тип и форма создаваемых сущностей зависят напрямую от назначения каждого конкретного CASE-инструментария. Это могут быть файлы с кодом на одном из текстовых языков программирования (например, “скелеты” отдельных частей приложения в виде функций-заглушек или полноценный код, готовый к компиляции, если в модели достаточно информации для его генерации), автоматические тесты или техническая документация для создаваемого кода, различные отчеты и документация по созданным моделям, скрипты сборки и размещения, инсталляторы готовых версий создаваемого ПО и многое другое, для автоматизированного создания чего могут использоваться CASE-среды. В эту же категорию можно отнести инструменты автоматических или автоматизированных преобразований моделей (например, для осуществления рефакторингов моделей или выполнения над ними часто повторяющихся рутинных операций).
- Механизмы циклической разработки ПО (round-trip engineering), позволяющие

дополнять “вручную” код, автоматически сгенерированный по моделям, так, чтобы изменения остались в силе при последующих перегенерациях. Стоит отметить, что для CASE-средств, основанных на UML, наличие подобных инструментов является крайне важным, поскольку UML является языком проектирования, и сгенерировать полноценный код возможно далеко не по всем UML-моделям.

- Средства обратного проектирования (reverse engineering), позволяющие строить модели по целевым артефактам (документации, коду и т.п.).
- Средства эмуляции и отладки создаваемых систем или их частей. Для ряда программных систем отладка может происходить посредством генерации исходного кода и отладки его с помощью соответствующих средств (например, запуска программ на языке C в отладчике gdb), однако часто процесс отладки целевой системы может происходить и на уровне моделей (например, при дороговизне или требований особых условий для реального запуска системы). В этом случае инструментальная среда может выдавать пользователю наглядную информацию о ходе процесса исполнения (например, подсвечивая исполняемую в текущий момент часть модели).
- Инструменты поддержки процесса командной разработки. Может быть поддержана как синхронная командная работа (например, одновременное изменение одной и той же диаграммы проектировщиками с разных компьютеров посредством сетевого доступа к центральному репозиторию), так и асинхронная (например, посредством обмена изменениями в моделях между CASE-средствами с помощью системы контроля версий).
- Набор верификаторов и анализаторов создаваемых моделей, обеспечивающих корректность целевого ПО и предоставляющих статистику по моделям в соответствии с используемыми в проекте метриками и т.п.
- Библиотеки шаблонов и готовых решений для набора типовых ситуаций, возникающих при проектировании выбранного типа ПО.
- Средства экспорта и импорта моделей для обеспечения возможности обмена данными между CASE-инструментами.
- Набор пользовательской документации к данной CASE-среде — руководства пользователя, учебные пособия, механизм всплывающих подсказок во время работы с системой и т.п.

Полезные источники

- Видео-курс про работу с требованиями
- Курс лекций по анализу требований