Преимущества UML

UML объектно-ориентирован, в результате чего методы описания результатов анализа и проектирования семантически близки к методам программирования на современных ОО-языках;

UML позволяет описать систему практически со всех возможных точек зрения и разные аспекты поведения системы;

Диаграммы UML сравнительно просты для чтения после достаточно быстрого ознакомления с его синтаксисом;

UML расширяет и позволяет вводить собственные текстовые и графические стереотипы, что способствует его применению не только в сфере программной инженерии;

UML получил широкое распространение и динамично развивается.

Критика UML

Избыточность языка. UML часто критикуется, как неоправданно большой и сложный. Он включает много избыточных или практически неиспользуемых диаграмм и конструкций.

Неточная семантика. Так как UML определён комбинацией себя (абстрактный синтаксис), OCL (языком описания ограничений — формальной проверки правильности) и Английского (подробная семантика), то он лишен скованности присущей языкам, точно определённым техниками формального описания. В некоторых случаях абстрактный синтаксис UML, OCL и Английский противоречат друг другу, в других случаях они неполные. Неточность описания самого UML одинаково отражается на пользователях и поставщиках инструментов, приводя к несовместимости инструментов из-за уникального трактования спецификаций.

Проблемы при изучении и внедрении. Вышеописанные проблемы делают проблематичным изучение и внедрение UML, особенно когда руководство насильно заставляет использовать UML инженеров при отсутствии у них предварительных навыков.

Только код отражает код. Ещё одно мнение — что важны рабочие системы, а не красивые модели. Как лаконично выразился Джек Ривс, «The code is the design» («Код и есть проект»).

Кумулятивная нагрузка/Рассогласование нагрузки (Cumulative Impedance/Impedance mismatch). Рассогласование нагрузки — термин из теории системного анализа для обозначения неспособности входа системы воспринять выход другой. Как в любой системе обозначений UML может представить одни системы более кратко и эффективно, чем другие. Таким образом, разработчик склоняется к решениям, которые более комфортно подходят к переплетению сильных сторон UML и языков программирования. Проблема становится более очевидной, если язык разработки не придерживается принципов ортодоксальной объектно-ориентированной доктрины (не старается соответствовать традиционным принципам ООП).

Пытается быть всем для всех. UML — это язык моделирования общего назначения, который пытается достигнуть совместимости со всеми возможными языками разработки. В контексте конкретного проекта, для достижения командой проектировщиков определённой цели, должны быть выбраны применимые возможности UML. Кроме того, пути ограничения области применения UML в конкретной области проходят через формализм, который не полностью сформулирован, и который сам является объектом критики.

Типы диаграмм представлены на слайде

1. Деятельности
   1. диаграмма, на которой показано разложение некоторой деятельности на её составные части.
2. Классов
   1. Более подробно позднее. Одна из основных диаграм
3. Взаимодействия
   1. Взаимодействие между объектами; акцент на связях
4. Компонентов
   1. показывает разбиение программной системы на структурные компоненты и связи (зависимости) между компонентами. В качестве физических компонент могут выступать файлы, библиотеки, модули, исполняемые файлы, пакеты и т. п.
5. Составных структур
   1. демонстрирует внутреннюю структуру классов и, по возможности, взаимодействие элементов (частей) внутренней структуры класса.
6. Развертывания
   1. служит для моделирования работающих узлов (аппаратных средств, англ. node) и артефактов, развёрнутых на них.
7. Объектов
   1. демонстрирует полный или частичный снимок моделируемой системы в заданный момент времени. На диаграмме объектов отображаются экземпляры классов (объекты) системы с указанием текущих значений их атрибутов и связей между объектами.
8. Пакетов
   1. Иерархическая структура времени компиляции
9. Последовательности
   1. Взаимодействие между объектами; акцент на последовательности
10. Конечных автоматов
    1. Как события изменяют объект в течение его жизни
11. Временная
    1. Взаимодействие между объектами; акцент на синхронизации
12. Прецедентов
    1. Как пользователи взаимодействуют с системой

Наиболее часто используются диаграммы классов.

Диаграмма классов описывает типы объектов системы и различного рода статические отношения, которые существуют между ними. На диаграммах классов отображаются также свойства классов, операции классов и ограничения, которые накладываются на связи между объектами.

Пример диаграммы классов

Прямоугольники на диаграмме представляют классы и разделены на три части: имя класса (жирный шрифт), его атрибуты и его операции.

Метка **видимость** обозначает, относится ли атрибут к открытым (**+**) (public) или к закрытым (**-**) private).

**• Имя** атрибута – способ ссылки класса на атрибут – приблизительно соответствует имени поля в языке программирования.

**• Тип** атрибута накладывает ограничение на вид объекта, который может быть размещен в атрибуте. Можно считать его аналогом типа поля в языке программирования.

**• Кратность**

**Кратность** свойства обозначает количество объектов, которые могут

заполнять данное свойство. Чаще всего встречаются следующие кратности:

**•** 1 (Заказ может представить только один клиент.)

**•** 0..1 (Корпоративный клиент может иметь, а может и не иметь единственного торгового представителя.)

**•** \* (Клиент не обязан размещать заказ, и количество заказов не ограничено. Он может разместить ноль или более заказов.)

**• Значение по умолчанию** представляет собой значение для вновь создаваемых объектов, если атрибут не определен в процессе создания.

**•** Элемент **{строка свойств}** позволяет указывать дополнительные свойства атрибута. В примере он равен **{readOnly}**, то есть клиенты не могут изменять атрибут. Если он пропущен, то, как правило,атрибут можно модифицировать. Остальные строки свойств будут

описаны позже.

Направление обозначает, является ли параметр входным (**in**), выходным (**out**) или тем и другим (**inout**). Если направление не указано, то предполагается **in**.

Например, в счете операция может выглядеть так:

**+ balanceOn (date: Date) : Money**

**Читаем слайд**

Считается, что между двумя элементами существует **зависимость** (dependency), если изменения в определении одного элемента (**сервера**) могут вызвать изменения в другом элементе (**клиенте**). В случае классов зависимости появляются по разным причинам: один класс посылает сообщение другому классу; один класс владеет другим классом как частью своих данных; один класс использует другой класс в качестве параметра операции. Если класс изменяет свой интерфейс, то сообщения, посылаемые этому классу, могут стать недействительными.

Здесь важно то, что зависимость имеет только одно направление и идет от класса представления (клиент) к классу предметной области (сервер). Таким образом, мы знаем, что имеем возможность свободно изменять класс **Класс А**, не оказывая влияния на объект Класс Били другие объекты предметной области.

Зависимость — это слабая форма отношения использования, при котором изменение в спецификации одного влечёт за собой изменение другого, причем обратное не обязательно. Возникает когда объект выступает например в форме параметра или локальной переменной.

Графически представляется пунктирной стрелкой, идущей от зависимого элемента к тому, от которого он зависит.

Существует несколько именованных вариантов.

Зависимость может быть между экземплярами, классами или экземпляром и классом.

Ассоциация показывает, что объекты одной сущности (класса) связаны с объектами другой сущности. Существует пять различных типов ассоциации. Наиболее распространёнными являются двунаправленная и однонаправленная. Например, классы «рейс» и «самолёт» связаны двунаправленной ассоциацией, а классы «человек» и «кофейный автомат» связаны однонаправленной.

Двойные ассоциации (с двумя концами) представляются линией, соединяющей два классовых блока. Ассоциации более высокой степени имеют более двух концов и представляются линиями, один конец которых идет к классовому блоку, а другой к общему ромбику. В представлении однонаправленной ассоциации добавляется стрелка, указывающая на направление ассоциации.

Стрелка – это порядок следования классов

Агрегация — это разновидность ассоциации при отношении между целым и его частями. Как тип ассоциации агрегация может быть именованной. Одно отношение агрегации не может включать более двух классов (контейнер и содержимое).

Агрегация встречается, когда один класс является коллекцией или контейнером других. Причём по умолчанию, агрегацией называют агрегацию по ссылке, то есть когда время существования содержащихся классов не зависит от времени существования содержащего их класса. Если контейнер будет уничтожен, то его содержимое — нет.

Графически агрегация представляется пустым ромбиком на блоке класса и линией, идущей от этого ромбика к содержащемуся классу.

Композиция — более строгий вариант агрегации. Известна также как агрегация по значению.

Композиция имеет жёсткую зависимость времени существования экземпляров класса контейнера и экземпляров содержащихся классов. Если контейнер будет уничтожен, то всё его содержимое будет также уничтожено.

Графически представляется как и агрегация, но с закрашенным ромбиком.

К одним из наиболее частых источников недоразумений в UML –можно отнести агрегацию и композицию. В нескольких словах это можно объяснить так: Агрегация (aggregation) – это отношение типа «часть целого». Точно так же можно сказать, что двигатель и колеса представляют собой части автомобиля. Звучит вроде бы просто, однако при рассмотрении разницы между агрегацией и композицией возникают определенные трудности.

До появления языка UML вопрос о различии между агрегацией и композицией у аналитиков просто не возникал. Осознавалась подобная неопределенность или нет, но свои работы в этом вопросе аналитики совсем не согласовывали между собой. В результате многие разработчики считают агрегацию важной, но по совершенно другой причине.

Язык UML включает агрегацию (рис. 5.3) но семантика ее очень расплывчата. Как говорит Джим Рамбо (Jim Rumbaugh): «Можно представить себе агрегацию как плацебо для моделирования»

Наряду с агрегацией в языке UML есть более определенное свойство – композиция (composition). На рис. 5.4 экземпляр класса Point (Точка) может быть частью многоугольника, а может представлять центр окружности, но он не может быть и тем и другим одновременно. Главное правило состоит в том, что хотя класс может быть частью нескольких других классов, но любой экземпляр может принадлежать только одному владельцу. На диаграмме классов можно показать несколько классов потенциальных владельцев, но у любого экземпляра класса

есть только один объект владелец.

Вы заметите, что на рис 5.4 я не показываю обратные кратности. В большинстве случаев, как и здесь, они равны 0..1. Единственной альтернативой является значение 1, когда класскомпонент разработан такимобразом, что у него только один класс владелец.

Правило «нет совместного владения» является ключевым в композиции. Другое допущение состоит в том, что если удаляется многоугольник (Polygon), то автоматически должны удалиться все точки (Points),которыми он владеет.

Композиция – это хороший способ показать свойства, которыми владеют по значению, свойства объектовзначений (стр. 100) или свойства, которые имеют определенные и до некоторой степени исключительные права владения другими компонентами. Агрегация совершенно не

имеет смысла; поэтому я не рекомендовал бы применять ее в диаграммах. Если вы встретите ее в диаграммах других разработчиков, то вам придется покопаться, чтобы понять их значение. Разные авторы и команды разработчиков используют их в совершенно разных целях.

Типичный пример **обобщения** (generalization) включает индивидуального и корпоративного клиентов некоторой бизнес\_системы. Несмотря на определенные различия, у них много общего. Одинаковые свойства можно поместить в базовый класс **Customer** (Клиент, супертип), при этом класс **Personal Customer** (Индивидуальный клиент) и класс **Corporate Customer** (Корпоративный клиент) будут выступать как подтипы.

**Реализация** — отношение между двумя элементами модели, в котором один элемент (*клиент*) реализует поведение, заданное другим (*поставщиком*).

**Абстрактный класс** (abstract class) – это класс, который нельзя реализовать непосредственно. Вместо этого создается экземпляр подкласса. Обычно абстрактный класс имеет одну или более абстрактных операций. У **абстрактной операции** (abstract operation) нет реализации; это чистое объявление, которое клиенты могут привязать к абстрактному классу.

Наиболее распространенным способом обозначения абстрактного класса или операции в языке UML является написание их имен курсивом. Можно также сделать свойства абстрактными, определяя абстрактное свойство или методы доступа. Курсив сложно изобразить на доске, поэтому можно прибегнуть к метке: **{abstract}**. Интерфейс – это класс, не имеющий реализации, то есть вся его функциональность абстрактна. Интерфейсы прямо соответствуют интерфейсам в C# и Java и являются общей идиомой в других типизированных языках. Интерфейс обозначается ключевым словом **«interface»**.

Классы обладают двумя типами отношений с интерфейсами: предоставление или требование. Класс **предоставляет интерфейс**, если его можно заменить на интерфейс. В Java и .NET класс может сделать это, реализуя интерфейс или подтип интерфейса. В C++ создается подкласс класса, являющегося интерфейсом.

Класс требует интерфейс, если для работы ему нужен экземпляр данного интерфейса. По сути дела, это зависимость от интерфейса. На рис. 5.6 эти отношения демонстрируются в действии на базе небольшого набора классов, заимствованных из Java. Я мог бы написать класс Order (Заказ), содержащий список позиций заказа (Line Items). Поскольку я использую список, то класс Order зависит от интерфейса List (Список). Предположим, что он вызывает методы equals, add и get.

При выполнении связывания объект Order действительно будет использовать экземпляр класса ArrayList, но ему не нужно знать, что необходимо вызывать эти три метода, поскольку они входят в состав интерфейса List.

Класс ArrayList – это подкласс класса AbstractList. Класс AbstractList предоставляет некоторую, но не всю реализацию поведения интерфейса List. В частности, метод get – абстрактный. В результате ArrayList реализует метод get, а также переопределяет некоторые другие операции класса AbstractList. В данном случае он переопределяет метод add,но вполне удовлетворен наследованием реализации метода equals.

Почему бы мне просто не отказаться от этого и не заставить Order прямо использовать ArrayList? Применение интерфейса позволяет мне получить преимущество при последующем изменении реализации, если потребуется. Другой способ реализации может оказаться более производительным – он может предоставить функции работы с базой данных или другие возможности. Программируя интерфейс, а не реализацию, я избегаю необходимости переделывать весь код, когда достаточно изменить реализацию класса List. Следует всегда стараться программировать интерфейс так, как показано выше, то есть всегда использовать наиболее общий тип.

Диаграммы последовательности

**Диаграммы взаимодействия** (interaction diagrams) описывают взаимодействие групп объектов в различных условиях их поведения. UML определяет диаграммы взаимодействия нескольких типов, из которых наиболее употребительными являются диаграммы последовательности. Обычно диаграмма последовательности описывает один сценарий. На диаграмме показаны экземпляры объектов и сообщения, которыми обмениваются объекты в рамках одного прецедента (use case).

Для того чтобы начать обсуждение, рассмотрим простой сценарий. Предположим, что у нас есть заказ, и мы собираемся вызвать команду для определения его стоимости. При этом объекту заказа (**Order**) необходимо просмотреть все позиции заказа (**Line Items**) и определить их цены, основанные на правилах построения цены продукции в строке заказа (**Order Line**). Проделав это для всех позиций заказа, объект заказа должен вычислить общую скидку, которая определяется индивидуально для каждого клиента.

На рис. 4.1 приведена диаграмма, представляющая реализацию данного сценария. Диаграммы последовательности показывают взаимодействие, представляя каждого участника вместе с его линией жизни (lifeline), которая идет вертикально вниз и упорядочивает сообщения на странице; сообщения также следует читать сверху вниз. Одно из преимуществ диаграммы последовательности заключается в том, что мне почти не придется объяснять ее нотацию. Можно видеть, что экземпляр заказа посылает строке заказа сообщения getQuantity и getProduct. Можно также видеть, как заказ применяет метод к самому себе и как этот метод посылает сообщение getDiscountInfo экземпляру клиента.

Однако диаграмма не все показывает так хорошо. Последовательность сообщений getQuantity, getProduct, getPricingDetails и calculateBasePrice должна быть реализована для каждой строки заказа, тогда как метод calculateDiscounts вызывается лишь однажды. Такое заключение нельзя сделать на основе этой диаграммы, но позднее я введу дополнительное обозначение, которое поможет в этом.

Каждая линия жизни имеет полосу активности, которая показывает интервал активности участника при взаимодействии. Она соответствует времени нахождения в стеке одного из методов участника. В языке UML полосы активности не обязательны, но я считаю их исключительно удобными при пояснении поведения. Единственным исключением является стадия проработки дизайна, поскольку их неудобно рисовать на белых досках.

Именование бывает часто полезным для установления связей между участниками на диаграмме. Как видно на диаграмме, вызов метода **getProduct** возвращает **aProduct**, имеющего то же самое имя и, следовательно, означающего того же самого участника, **aProduct**, которому посылается вызов **getPricingDetails**. Обратите внимание, что обратной стрелкой я обозначил только этот вызов с целью показать соответствие. Многие разработчики используют возвраты для всех вызовов, но я предпочитаю применять их, только когда это дает дополнительную информацию; в противном случае они просто вносят неразбериху. Не исключено, что даже в данном случае можно было опустить возврат, не запутав читателя.

В диаграммах последовательности для создания и удаления участников применяются некоторые дополнительные обозначения (рис. 4.3).

В случае создания участника надо нарисовать стрелку сообщения, направленную к прямоугольнику участника. Если применяется конструктор, то имя сообщения не обязательно, но я обычно маркирую его словом «new» в любом случае. Если участник выполняет что\_нибудь

непосредственно после создания, например команду запроса, то надо начать активацию сразу после прямоугольника участника.

Удаление участника обозначается большим крестом (X). Стрелка сообщения, идущая в X, означает, что один участник явным образом удаляет другого; X в конце линии жизни показывает, что участник удаляет сам себя.

Если в системе работает сборщик мусора, то объекты не удаляются

вручную, тем не менее следует при помощи X показать, что объект

больше не нужен и готов к удалению. Так следует поступать и в случае

операций закрытия, показывая, что объект больше не используется.

Общая проблема диаграмм последовательности заключается в том, как отображать циклы и условные конструкции. Прежде всего надо усвоить, что диаграммы последовательности для этого не предназначены. Подобные управляющие структуры лучше показывать с помощью диаграммы деятельности или собственно кода. Диаграммы последовательности применяются для визуализации процесса взаимодействия объектов, а не как средство моделирования алгоритма управления. Как было сказано, существуют дополнительные обозначения. И для циклов, и для условий используются **фреймы взаимодействий** (interaction frames), представляющие собой средство разметки диаграммы взаимодействия. На рис. 4.4 показан простой алгоритм, основанный

на следующем псевдокоде.

**Оператор Значение**

**alt** Несколько альтернативных фрагментов (alternative); выполняет\_

ся только тот фрагмент, условие которого истинно (рис. 4.4)

**opt** Необязательный (optional) фрагмент; выполняется, только если

условие истинно. Эквивалентно **alt** с одной веткой (рис. 4.4)

**par** Параллельный (parallel); все фрагменты выполняются парал\_

лельно

**loop** Цикл (loop); фрагмент может выполняться несколько раз, а за\_

щита обозначает тело итерации (рис. 4.4)

**region** Критическая область (critical region); фрагмент может иметь

только один поток, выполняющийся за один прием

**neg** Отрицательный (negative) фрагмент; обозначает неверное вза\_

имодействие

**ref** Ссылка (reference); ссылается на взаимодействие, определенное

на другой диаграмме. Фрейм рисуется, чтобы охватить линии

жизни, вовлеченные во взаимодействие. Можно определять па\_

раметры и возвращать значение

**sd** Диаграмма последовательности (sequence diagram); используется

для очерчивания всей диаграммы последовательности, если это

необходимо

**foreach (lineitem)**

**if (product.value > $10K)**

**careful.dispatch**

**else**

**regular.dispatch**

**end if**

**end for**

**if (needsConfirmation)**

**messenger.confirm**

**end procedure**

**Диаграмма объектов** (object diagram) – это снимок объектов системы в какой\_то момент времени. Поскольку она показывает экземпляры, а не классы, то диаграмму объектов часто называют диаграммой экземпляров.

Диаграмму объектов можно использовать для отображения одного из вариантов конфигурации объектов. (На рис. 6.1 показано множество классов, а на рис. 6.2 представлено множество связанных объектов.)

Последний вариант очень полезен, когда допустимые связи между объектами могут быть сложными.

Можно определить, что элементы, показанные на рис. 6.2, являются экземплярами, поскольку их имена подчеркнуты. Каждое имя представляется в виде: **имя экземпляра : имя класса**. Обе части имени не являются обязательными, поэтому имена **John**, **:Person** и **aPerson** являются допустимыми. Если указано только имя класса, то необходимо поставить двоеточие. Можно также задать значения и атрибуты, как показано на рис. 6.2.

Строго говоря, элементы диаграммы объектов – это спецификации экземпляров, а не сами экземпляры. Причина в том, что разрешается оставлять обязательные атрибуты пустыми или показывать спецификации экземпляров абстрактных классов. Можно рассматривать **спецификации экземпляров** (instance specifications) как частично определенные экземпляры.

С другой стороны, диаграмму объектов можно считать коммуникационной диаграммой (*стр. 152*) без сообщений

Диаграммы развертывания представляют физическое расположение системы, показывая, на каком физическом оборудовании запускается та или иная составляющая программного обеспечения. Диаграммы развертывания очень просты, поэтому будем кратки. На рис. 8.1 показан пример простой диаграммы развертывания. Главными элементами диаграммы являются узлы, связанные информационными путями. **Узел** (node) – это то, что может содержать программное обеспечение. Узлы бывают двух типов. **Устройство** (device) – это физическое оборудование: компьютер или устройство, связанное с системой. **Среда выполнения** (execution environment) – это программное обеспечение, которое само может включать другое программное обеспечение, например операционную систему или процесс\_контейнер.

Узлы могут содержать **артефакты** (artifacts), которые являются физическим олицетворением программного обеспечения; обычно это файлы.

Такими файлами могут быть исполняемые файлы (такие как файлы *.exe*, двоичные файлы, файлы DLL, файлы JAR, сборки или сценарии) или файлы данных, конфигурационные файлы, HTML\_документыи т. д. Перечень артефактов внутри узла указывает на то, что на данном узле артефакт разворачивается в запускаемую систему.

Артефакты можно изображать в виде прямоугольников классов или перечислять их имена внутри узла. Если вы показываете эти элементы в виде прямоугольников классов, то можете добавить значок документа или ключевое слово **«artifact»**. Можно сопровождать узлы или артефакты значениями в виде меток, чтобы указать различную интересную информацию об узле, например поставщика, операционную систему, местоположение – в общем, все, что придет вам в голову.

Часто у вас будет множество физических узлов для решения одной и той же логической задачи. Можно отобразить этот факт, нарисовав множество прямоугольников узлов или поставив число в виде значения\_метки. На рис. 8.1 я обозначил три физических веб\_сервера с помощью метки **number deployed** (количество развернутых), но это не стандартная метка.

Артефакты часто являются реализацией компонентов. Это можно показать, задав значения\_метки внутри прямоугольников артефактов. Информационные пути между узлами представляют обмен информацией в системе. Можно сопровождать эти пути информацией об используемых информационных протоколах.

Прецеденты – это технология определения функциональных требований к системе. Работа прецедентов заключается в описании типичных взаимодействий между пользователями системы и самой системой и предоставлении описания процесса ее функционирования. Вместо того чтобы описывать прецеденты в лоб, я предпочитаю подкрасться к ним сзади и начать с описания сценариев. **Сценарий** (scenario) – это последовательность шагов, описывающих взаимодействие пользователя и системы. Поэтому при наличии онлайнового магазина, основанного на веб\_сайте, мы можем использовать сценарий «Покупка товара» (Buy a Product), в котором происходит следующее.

*Покупатель просматривает каталог и помещает выбранные товары в корзину. При желании оплатить покупку он вводит информацию о кредитной карте и производит платеж. Система проверяет авторизацию кредитной карты и подтверждает оплату товара тотчас же и по электронной почте.*

Подобный сценарий описывает только одну ситуацию, которая может иметь место. Однако если авторизация кредитной карты окажется неудачной, то подобная ситуация может послужить предметом уже другого сценария. В другом случае у вас может быть постоянный клиент, для которого проверка информации о покупке и кредитной карте необязательна, и это будет третий сценарий.

Так или иначе, но все эти сценарии похожи. Суть в том, что во всех трех сценариях у пользователя одна и та же цель: купить товар. Пользователь не всегда может это сделать, но цель остается. Именно цель пользователя является ключом к прецедентам: прецедент представляет собой множество сценариев, объединенных некоторой общей целью пользователя.

В терминах прецедента пользователи называются актерами. **Актер** (actor) представляет собой некую роль, которую пользователь играет по отношению к системе. Актерами могут быть пользователь, торговый представитель пользователя, менеджер по продажам и товаровед.

Актеры действуют в рамках прецедентов. Один актер может выполнять несколько прецедентов; и наоборот, в соответствии с одним прецедентом могут действовать несколько актеров. Обычно клиентов много, поэтому роль клиента могут играть многие люди. К тому же один человек может играть несколько ролей, например менеджер по продажам, выполняющий роль торгового представителя клиента. Актер не обязательно должен быть человеком. Если система предоставляет некоторый сервис другой компьютерной системе, то другая система является актером.

На самом деле *актер* – не совсем верный термин; возможно, термин *роль* (*role*) подошел бы лучше. Очевидно, имел место неправильный перевод со шведского языка, и в результате сообщество пользователей прецедентов теперь употребляет термин *актер*.

Прецеденты считаются важной частью языка UML. Однако удивительно то, что определение прецедентов в UML довольно скудное. В UML ничего не говорится о том, как определять содержимое прецедента.

Все, что описано в UML, – это диаграмма прецедентов, которая показывает, как прецеденты связаны друг с другом. Но почти вся ценность прецедентов как раз в их содержании, а диаграмма имеет ограниченное значение.

**Покупка товара**

Целевой уровень: уровень моря

Главный успешный сценарий:

1. Покупатель просматривает каталог и выбирает товары для покупки.

2. Покупатель оценивает стоимость всех товаров.

3. Покупатель вводит информацию, необходимую для доставки товара

(адрес, доставка на следующий день или в течение трех дней).

4. Система предоставляет полную информацию о цене товара и его доставке.

5. Покупатель вводит информацию о кредитной карточке.

6. Система осуществляет авторизацию счета покупателя.

7. Система подтверждает оплату товаров немедленно.

8. Система посылает подтверждение оплаты товаров по адресу электронной почты покупателя.

Расширения:

3а. Клиент является постоянным покупателем.

.1: Система предоставляет информацию о текущей покупке и ее цене,

а также информацию о счете.

.2: Покупатель может согласиться или изменить значения по умолчанию, затем возвращаемся к шагу 6 главного успешного сценария.

6a. Система не подтверждает авторизацию счета.

.1: Пользователь может повторить ввод информации о кредитной карте

или закончить сеанс.

включение прецедента, но я думаю, что подчеркивание, которое предполагает гиперссылку, работает прекрасно, а во многих инструментах действительно будет гиперссылкой. Так, на рис. 9.1 первый шаг включает шаблон «просматривает каталог и выбирает товары для покупки».

Включенные прецеденты могут быть полезными в случае сложных шагов, которые иначе загромождали бы главный сценарий, или когда одни и те же шаги присутствуют в нескольких сценариях. Однако не пытайтесь разбивать прецеденты на подпрецеденты и использовать их

для функциональной декомпозиции. Такая декомпозиция – хороший способ потерять много времени.

Наряду с шагами сценария можно вставить в прецедент дополнительную общую информацию.

**• Предусловие** (pre\_condition) описывает действия, обязательно выполняемые системой перед тем, как она разрешит начать работу прецедента. Это полезная информация, позволяющая разработчикам не проверять некоторые условия в их программе.

**• Гарантия** (guarantee) описывает обязательные действия системы по окончании работы шаблона ответа. Успешные гарантии выполняются после успешного сценария; минимальные гарантии выполняются после любого сценария.

**• Триггер** (trigger) определяет событие, инициирующее выполнение прецедента.

При рассмотрении дополнительных элементов относитесь к этому скептически. Лучше сделать слишком мало, чем слишком много. Кроме того, приложите максимум усилий, чтобы сделать прецедент кратким и легким для чтения. Я убедился, что излишне подробный прецедент,

который трудно читать, скорее приведет к провалу, чем к достижению цели. Не обязательно записывать все детали; устное общение часто бывает очень эффективным, особенно во время итеративного цикла, когда необходимые условия быстро выполняются запущенной программой.

Степень детализации, необходимая в прецеденте, зависит от уровня риска этого прецедента. Часто детали нужны в начале только немногих ключевых прецедентов, другие можно конкретизировать непосредственно перед их реализацией.

Я уже говорил, что язык UML умалчивает о содержимом прецедента, но предоставляет формат диаграммы, позволяющий его отображать (рис. 9.2). Хотя диаграмма иногда оказывается полезной, без нее можно обойтись. При разработке прецедента не стоит прилагать много усилий для создания диаграммы. Вместо этого лучше сконцентрироваться на текстовом содержании прецедентов Лучше всего обдумывать диаграмму прецедентов с помощью графической таблицы, показывающей их содержимое. Она напоминает диаграмму контекста, используемую в структурных методах, поскольку она показывает границы системы и ее взаимодействие с внешним миром. Диаграмма прецедентов показывает актеров, прецеденты и отношения между ними:

**•** Какие актеры выполняют тот или иной прецедент

**•** Какие прецеденты включают другие прецеденты

В языке UML помимо отношения **«include»** (включает) есть и другие типы отношений между прецедентами, например отношение **«extend»** (расширяет). Я настоятельно рекомендую его избегать. Слишком часто на моих глазах разработчики целыми командами надолго погружались в рассмотрение различных отношений между прецедентами, понапрасну растрачивая силы. Лучше уделяйте больше внимания текстовому описанию прецедента; именно в этом заключается истинная ценность этой технологии.