UML Основы

Краткое руководство
по стандартному языку
объектного моделирования

Третье издание

Мартин Фаулер

Санкт-Петербург
2005

7

Диаграммы пакетов

Классы составляют структурный костяк объектно-ориентированной системы. Хотя они исключительно полезны, но нужно нечто большее для структурирования больших систем, которые могут состоять из сотен классов.

Пакет (package) – это инструмент группирования, который позволяет взять любую конструкцию UML и объединить ее элементы в единицы высокого уровня. В основном пакеты служат для объединения классов в группы, и именно этот способ их применения я здесь описываю, но помните, что пакеты могут применяться для любой другой конструкции языка UML.

В модели UML каждый класс может включаться только в один пакет. Пакеты могут также входить в состав других пакетов, поэтому мы остаемся в иерархической структуре, в которой пакеты верхнего уровня распадаются на подпакеты со своими собственными подпакетами, и так далее, до самого низа иерархии классов. Пакет может содержать и подпакеты, и классы.

В терминах программирования пакеты в UML соответствуют таким группирующим конструкциям, как пакеты в Java и пространства имен в C++ и .NET.

Каждый пакет представляет пространство имен (namespace), а это означает, что каждый класс внутри собственного пакета должен иметь уникальное имя. Если я хочу создать пакет с именем `Date`, а класс `Date` уже существует в пакете `System`, то я обязан поместить его в отдельный пакет. Чтобы отличить один класс от другого, я могу использовать полностью определенное имя (fully qualified name), то есть имя, которое указывает на структуру, владеющую пакетом. В языке UML в именах пакетов используются двойные двоеточия, поэтому классы дат могут иметь имена `System::Date` и `MartinFowler::Util::Date`.

На диаграммах пакеты изображаются в виде папок с закладками (рис. 7.1). Можно показывать только имя пакета или имя вместе с его

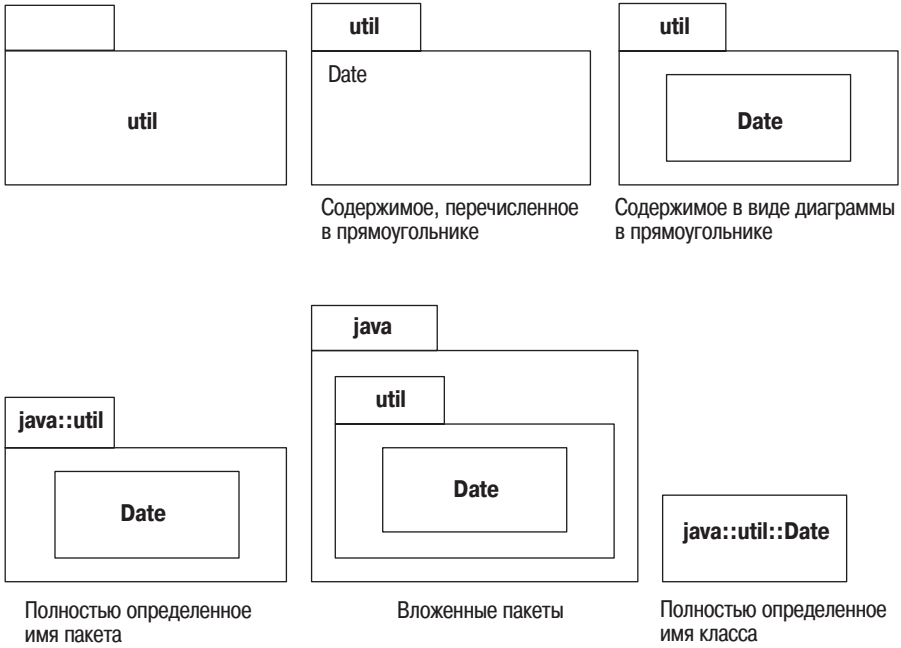

Рис. 7.1. Способы изображения пакетов на диаграммах

содержимым. В любом случае можно использовать либо полностью определенные имена, либо обычные имена. Изображение содержимого с помощью значков классов позволяет показать все особенности класса, вплоть до изображения диаграммы классов внутри пакета. Простое перечисление имен имеет смысл, если вы хотите лишь показать, какие классы входят в той или иной пакет.

Вполне можно встретить класс, например с именем `Date` (как в java.util), а не с полностью определенным именем. Этот стиль является соглашением, в основном принятым в Rational Rose, которое не входит в стандарт языка.

UML разрешает классам в пакетах быть открытыми (public) или закрытыми (private). Открытые классы представляют часть интерфейса пакета и могут быть использованы классами из других пакетов; закрытые классы недоступны извне. В различных средах программирования действуют различные правила в отношении видимости их группирующими конструкциями; необходимо придерживаться правил своего программного окружения, даже если это идет вразрез с правилами UML.

В таких случаях полезно сократить интерфейс пакета, экспортируя только небольшое подмножество операций, связанных с открытыми классами пакета. Можно сделать это, присвоив всем классам модификатор видимости private (закрытый), так чтобы они были доступны

только классам данного пакета, а также создав дополнительные открытые классы для внешнего использования. Эти дополнительные классы, называемые *Facades* (Фасады) [21], делегируют открытые операции своим более застенчивым соратникам по пакету.

Как распределить классы по пакетам? Это действительно сложный вопрос, на который может ответить только специалист с большим опытом работы в области проектирования. В этом деле могут помочь два правила: общий принцип замыкания (Common Closure Principle) и общий принцип повторного использования (Common Reuse Principle) [30]. Общий принцип замыкания гласит, что причины изменения классов пакета должны быть одинаковые. Общий принцип повторного использования утверждает, что классы должны использоваться повторно все вместе. Большинство причин, по которым классы должны объединяться в пакет, проистекают из зависимостей между классами, к которым я сейчас и перехожу.

Пакеты и зависимости

Диаграмма пакетов (package diagram) показывает пакеты и зависимости между ними. Я ввел понятие зависимости на *стр. 74*. При наличии пакетов для классов представления и пакетов для классов предметной области пакет представления зависит от пакета предметной области, если любой класс пакета представления зависит от какого-либо класса пакета предметной области. Таким образом, межпакетная зависимость обобщает зависимости между их содержимым.

В языке UML имеются разнообразные виды зависимостей, каждая из которых обладает самостоятельной семантикой и стереотипом. По моему мнению, намного проще начинать с зависимости без стереотипа и использовать более конкретные виды зависимостей только по мере необходимости. В средних и больших по размеру системах рисование диаграммы пакетов может быть одним из самых ценных приемов, позволяющих управлять их многомерной структурой. В идеале такая диаграмма должна быть сгенерирована на основании собственно исходного кода, так чтобы она реально отражала происходящее в системе.

Хорошая структура пакетов имеет прозрачный поток (clear flow) зависимостей – концепцию, которую трудно определить, но легко распознать. На рис. 7.2 показана простая диаграмма пакетов для промышленного приложения, которая хорошо структурирована и имеет прозрачный поток зависимостей.

Часто можно идентифицировать прозрачный поток, поскольку все зависимости идут в одном направлении. Это хороший индикатор правильно структурированной системы, но пакеты `data mapper` (преобразователь данных) на рис. 7.2 представляют исключение из этого эмпирического правила. Пакеты преобразователей данных действуют в качестве изолирующего уровня между пакетами предметной области

(domain) и пакетами базы данных (database) и служат примером шаблона Mapper (Преобразователь) [19].

Многие авторы утверждают, что в зависимостях не должно быть циклов (Acyclic Dependency Principle – принцип ацикличности зависимостей, [30]). Я не считаю это правило абсолютным, но думаю, что циклы необходимо локализовать, в частности не должно быть циклов, пересекающих уровни.

Чем больше зависимостей входит в пакет, тем более стабильным должен быть его интерфейс, поскольку любые изменения интерфейса отразятся на всех пакетах, зависящих от него (Stable Dependencies Principle – принцип стабильных зависимостей, [30]). Поэтому на рис. 7.2 пакету asset domain (предметная область собственности) требуется более стабильный интерфейс, чем пакету leasing data mapper (преобразователь данных аренды). Вы увидите, что зачастую более стабильные пакеты содержат относительно больше интерфейсов и абстрактных классов (Stable Abstractions Principle – принцип стабильных абстракций, [30]).

Отношения зависимостей не транзитивны (*стр. 75*). Чтобы убедиться в важности этого свойства для зависимостей, взгляните снова на рис. 7.2. Если изменяется класс пакета предметной области собственности, то может измениться и класс из пакета предметной области аренды (leasing domain). Но эти изменения не обязательно пройдут через представление аренды (leasing presentation). (Они проходят, толь-

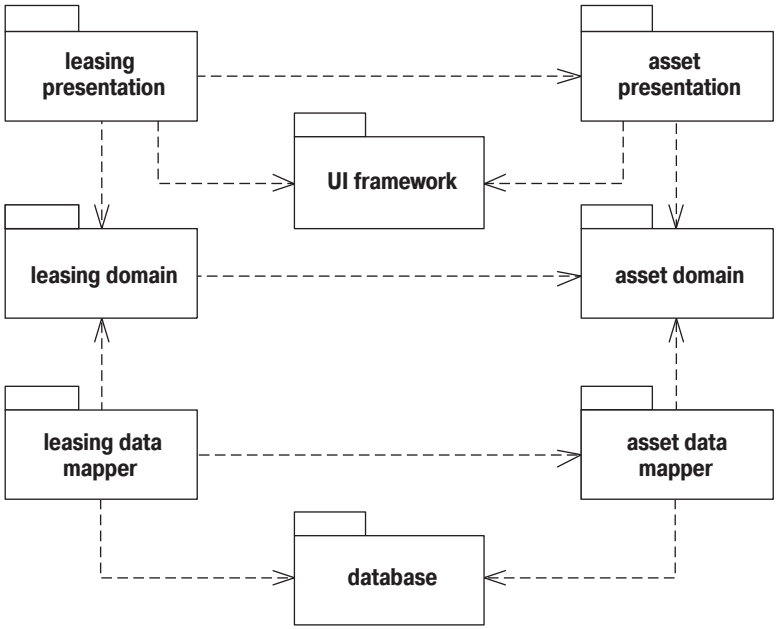

Рис. 7.2. Диаграмма пакетов для промышленного предприятия

ко если предметная область аренды изменяет свой интерфейс.) Некоторые пакеты используются в таком количестве мест, что можно запутаться в линиях зависимостей, идущих к ним. В таком случае действует соглашение, в соответствии с которым на пакете указывается ключевое слово «global» (глобальный).

В пакетах UML определены также конструкции, позволяющие пакетам импортировать классы из одного пакета в другой и объединять их, используя зависимости с ключевыми словами для их обозначения. Однако правила для такого типа элементов существенно отличаются в различных языках программирования. В целом, я думаю, что на практике значительно более полезна основная нотация зависимостей.

Аспекты пакетов

Рассматривая рис. 7.2, вы понимаете, что на диаграмме присутствуют структуры двух видов. Одна – структура уровня приложения: представление (presentation), предметная область (domain), преобразователь данных (data mapper) и база данных (database). Другая – структура предметных областей: аренда (leasing) и собственность (asset).

Это можно сделать более прозрачным, если разделить два данных аспекта так, как это сделано на рис. 7.3. На представленной диаграмме можно ясно увидеть каждый аспект. Однако эти два аспекта не явля-

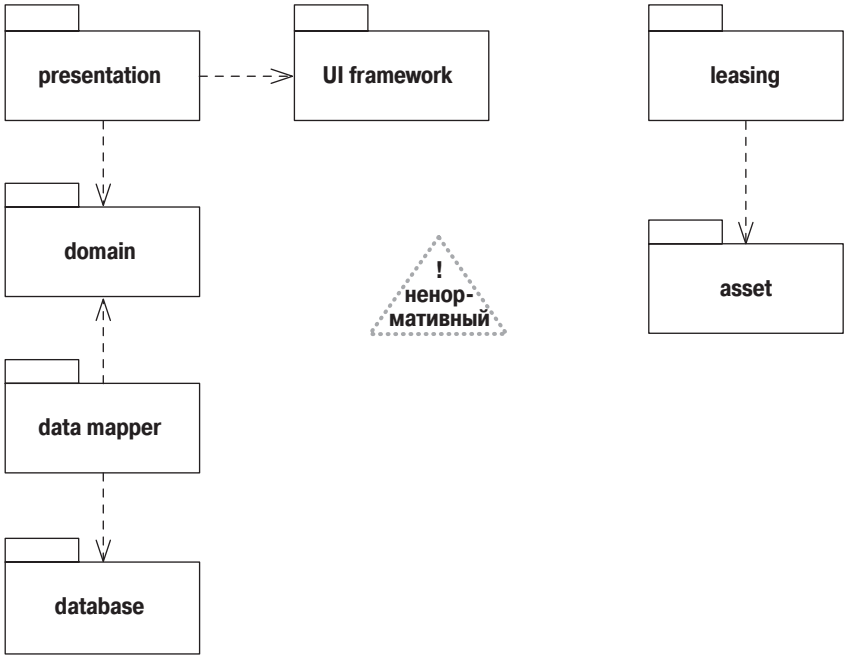

Рис. 7.3. Разделение рис. 7.2 на два аспекта

ются истинными пакетами, поскольку рассматриваемые классы можно объединить в один пакет. (Возможно, вам придется извлечь по одному классу из каждого аспекта.) Эта проблема является отражением проблемы иерархических пространств имен в языках программирования. Хотя диаграммы, подобные представленным на рис 7.3, не входят в стандарт языка UML, они зачастую очень удобны для объяснения структуры сложных приложений.

Реализация пакетов

Часто встречается ситуация, когда один пакет определяет интерфейс, который может быть реализован многими другими пакетами, как это показано на рис. 7.4. В данном случае отношение реализации означает, что шлюз базы данных (`Database Gateway`) определяет интерфейс, а другие классы шлюзов обеспечивают реализацию. На практике это может означать, что пакет шлюза базы данных (`Database Gateway`) содержит интерфейсы и абстрактные классы, которые полностью реализуются в других пакетах.

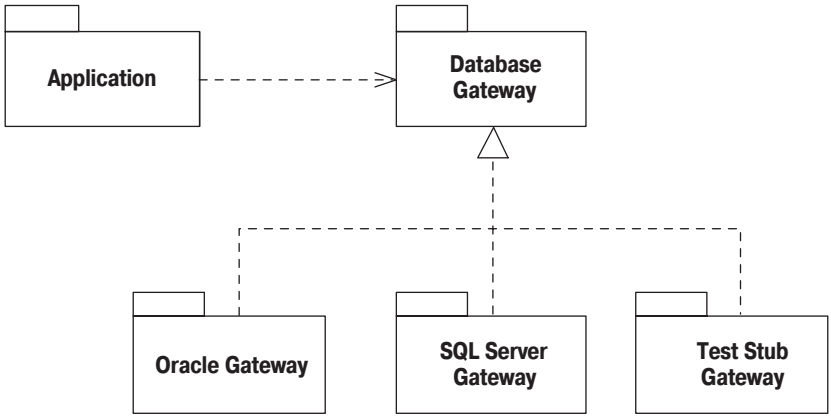

Рис. 7.4. Пакет, реализованный другими пакетами

Общепринято размещать интерфейс и его реализацию в разных пакетах. Действительно, клиентский пакет часто содержит интерфейс, который должен быть реализован другим пакетом (такую же нотацию затребованного интерфейса я обсуждал на *стр. 97*).

Допустим, что вы хотите предоставить некоторый пользовательский интерфейс (UI) выключателей (controls) для включения и выключения некоторого объекта. Мы хотим, чтобы он работал с различными устройствами, такими как обогреватели (heaters) или лампы (lights). Выключатели UI должны вызывать методы обогревателя, но мы не хотим, чтобы выключатели зависели от обогревателя. Мы можем избежать этой зависимости, определяя в пакете выключателей интерфейс,

который затем реализуется каждым классом, работающим с этими выключателями, как показано на рис. 7.5. Здесь представлен пример шаблона Separated Interface (Разделенный интерфейс) [19].

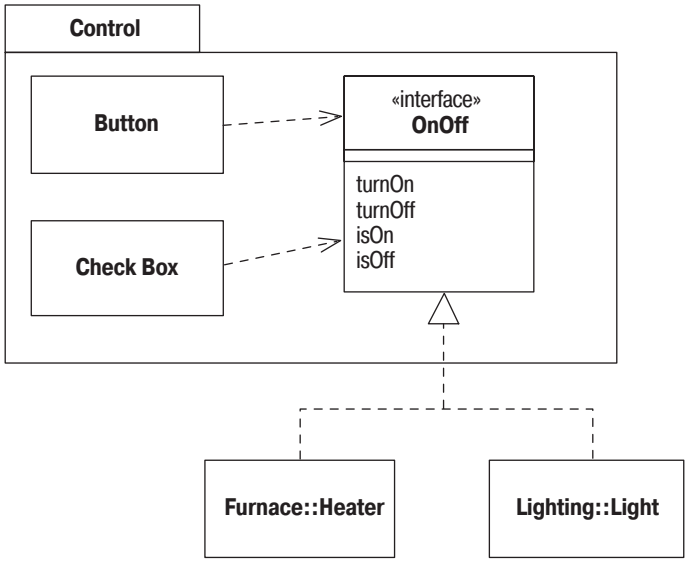

Рис. 7.5. *Определение затребованного интерфейса в клиентском пакете*

Когда применяются диаграммы пакетов

Я считаю, что **диаграммы пакетов исключительно удобны в больших по размерам системах для представления картины зависимостей между основными элементами системы.** Такие диаграммы хорошо соответствуют общепринятым программным структурам. Рисование диаграмм пакетов и зависимостей помогает держать под контролем зависимости приложения. Диаграммы пакетов представляют группирующий механизм времени компиляции. Для представления компоновки объектов во время выполнения применяются диаграммы составных структур (composite structure) (*стр. 155*).

8

Диаграммы развертывания

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-документы и т. д. Перечень артефактов внутри узла указывает на то, что на дан- ном 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 по-

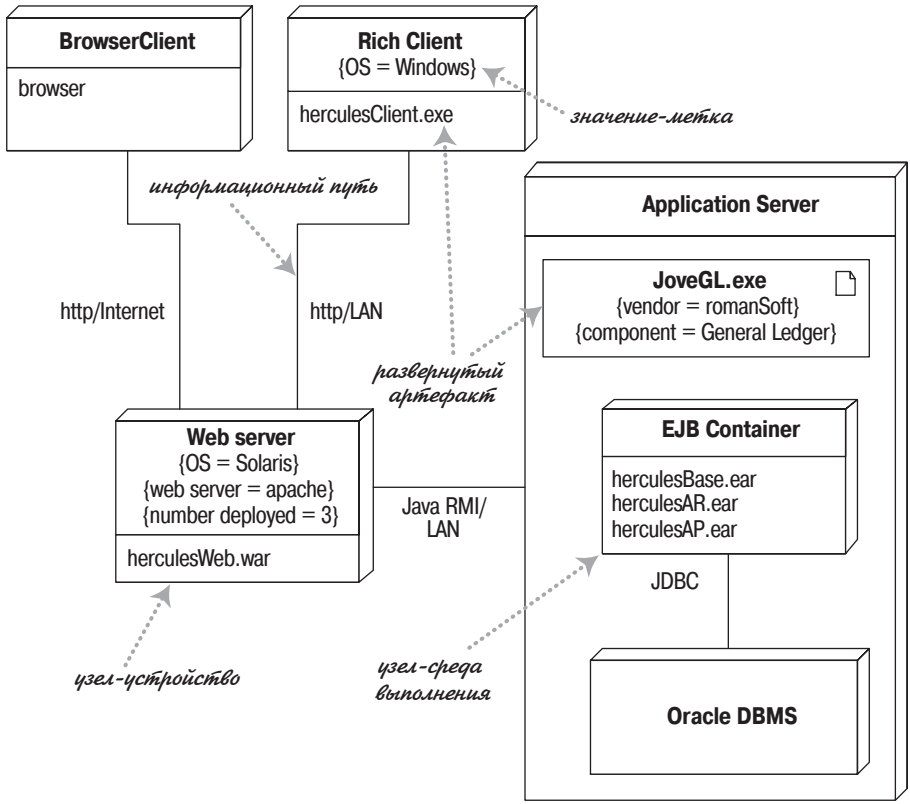

Рис. 8.1. Пример диаграммы развертывания

мощью 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.