UML Основы

Краткое руководство
по стандартному языку
объектного моделирования

Третье издание

Мартин Фаулер

Санкт-Петербург
2005

1

Введение

Что такое UML?

Унифицированный язык моделирования (UML) – это семейство графических нотаций, в основе которого лежит единая метамодель. Он помогает в описании и проектировании программных систем, в особенности систем, построенных с использованием объектно-ориентированных (ОО) технологий. Это определение в чем-то упрощенное. В действительности разные люди могут видеть в UML разные вещи. Это является следствием как собственной истории развития языка, так и различных точек зрения специалистов на то, что делает процесс разработки программного обеспечения эффективным. Поэтому моя задача в этой главе во многом заключается в построении общей картины книги и в объяснении различного видения и разнообразных способов применения UML разработчиками.

Графические языки моделирования уже продолжительное время широко используются в программной индустрии. Основная причина их появления состоит в том, что языки программирования не обеспечивают нужный уровень абстракции, способный облегчить процесс проектирования.

Несмотря на то что графические языки моделирования существуют уже достаточно давно, в среде разработчиков программного обеспечения очень много спорят об их роли. Эти споры оказывают непосредственное влияние на восприятие разработчиками самого языка UML.

UML представляет собой относительно открытый стандарт, находящийся под управлением группы OMG (Object Management Group – группа управления объектами), открытого консорциума компаний. Группа OMG была сформирована для создания стандартов, поддерживающих межсистемное взаимодействие, в частности взаимодействие объектно-ориентированных систем. Возможно, группа OMG более известна по стандартам CORBA (Common Object Request Broker Architecture – общая архитектура посредников запросов к объектам).

UML появился в результате процесса унификации множества объектно-ориентированных языков графического моделирования, процветавших в конце 80-х и в начале 90-х годов. Появившись в 1997 году, он отправил эту Вавилонскую башню в вечность, за что я и многие другие разработчики испытываем по отношению к нему глубокую благодарность.

Способы применения UML

Основу роли UML в разработке программного обеспечения составляют разнообразные способы использования языка, те различия, которые были перенесены из других языков графического моделирования. Эти отличия вызывают долгие и трудные дискуссии о том, как следует применять UML.

Чтобы разрешить эту сложную ситуацию, Стив Меллор (Steve Mellor) и я независимо пришли к определению трех режимов использования UML разработчиками: режим эскиза, режим проектирования и режим языка программирования. Безусловно, самый главный из трех, по крайней мере, на мой пристрастный взгляд, – это режим использования *UML для эскизирования*. В этом режиме разработчики используют UML для обмена информацией о различных аспектах системы. В режиме проектирования можно использовать эскизы при прямой и обратной разработке. При *прямой разработке* (*forward-engineering*) диаграммы рисуются до написания кода, а при *обратной разработке* (*reverse-engineering*) диаграммы строятся на основании исходного кода, чтобы лучше понять его.

Сущность эскизирования, или эскизного моделирования, в избирательности. В процессе прямой разработки вы делаете наброски отдельных элементов программы, которую собираетесь написать, и обычно обсуждаете их с некоторыми разработчиками из вашей команды. При этом с помощью эскизов вы хотите облегчить обмен идеями и вариантами того, что вы собираетесь делать. Вы обсуждаете не всю программу, над которой намереваетесь работать, а только самые важные ее моменты, которые вы хотите донести до коллег в первую очередь, или разделы проекта, которые вы хотите визуализировать до начала программирования. Такие совещания могут быть очень короткими: 10-минутное совещание по нескольким часам программирования или однодневное совещание, посвященное обсуждению двухнедельной итерации.

При обратной разработке вы используете эскизы, чтобы объяснить, как работает некоторая часть системы. Вы показываете не все классы, а только те, которые представляют интерес и которые стоит обсудить перед тем, как погрузиться в код. Поскольку эскизирование носит неформальный и динамичный характер и вам нужно делать это быстро и совместно, то наилучшим средством отображения является доска. Эскизы полезны также и в документации, при этом главную роль играет процесс передачи информации, а не полнота. Инструментами эскизного моделирования служат облегченные средства рисования, и

часто разработчики не очень придерживаются всех строгих правил UML. Большинство диаграмм UML, показанных в этой книге, как и в других моих книгах, представляют собой эскизы. Их сила в избирательности передачи информации, а не в полноте описания.

Напротив, язык *UML как* средство *проектирования* нацелен на полноту. В процессе прямой разработки идея состоит в том, что проект разрабатывается дизайнером, чья работа заключается в построении детальной модели для программиста, который будет выполнять кодирование. Такая модель должна быть достаточно полной в части заложенных проектных решений, а программист должен иметь возможность следовать им прямо и не особо задумываясь. Дизайнером модели может быть тот же самый программист, но, как правило, в качестве дизайнера выступает старший программист, который разрабатывает модели для команды программистов. Причина такого подхода лежит в аналогии с другими видами инженерной деятельности, когда профессиональные инженеры создают чертежи, которые затем передаются строительным компаниям.

Проектирование может быть использовано для всех деталей системы либо дизайнер может нарисовать модель какой-то конкретной части. Общий подход состоит в том, чтобы дизайнер разработал модели проектного уровня в виде интерфейсов подсистем, а затем дал возможность разработчикам поработать над реализацией подробностей.

При обратной разработке цель моделей состоит в представлении подробной информации о программе или в виде бумажных документов, или в виде, пригодном для интерактивного просмотра с помощью графического броузера. В такой модели можно показать все детали класса в графическом виде, который разработчикам проще понять.

При разработке моделей требуется более сложный инструментарий, чем при составлении эскизов, так как необходимо поддерживать детальность, соответствующую требованиям поставленной задачи. Специализированные CASE-средства (computer-aided software engineering – автоматизированная разработка программного обеспечения) попадают в эту категорию, хотя сам этот термин стал почти ругательным, и поставщики стараются его избегать. Инструменты прямой разработки поддерживают рисование диаграмм и копирование их в репозиторий с целью сохранения информации. Инструменты обратного проектирования читают исходный код, записывают его интерпретацию в репозиторий и генерируют диаграммы. Инструменты, позволяющие выполнять как прямую, так и обратную разработку, называются *двухсторонними* (*round-trip*).

Некоторые средства используют исходный код в качестве репозитория, а диаграммы используют его для графического представления. Такие инструменты более тесно связаны с программированием и часто встраиваются прямо в средства редактирования исходного кода. Мне нравится называть их «тройными» инструментами.

Граница между моделями и эскизами довольно размыта, но я думаю, что различия остаются в том, что эскизы сознательно выполняются неполными, подчеркивая важную информацию, в то время как модели нацелены на полноту, часто имея целью свести программирование к простым и до некоторой степени механическим действиям. Короче говоря, я бы определил эскизы как пробные элементы, а модели – как окончательные.

Чем дольше вы работаете с UML, а программирование становится все более механическим, тем очевиднее становится необходимость перехода к автоматизированному созданию программ. Действительно многие CASE-средства так или иначе генерируют код, что позволяет автоматизировать построение значительной части системы. В конце концов, вы достигнете такой точки, когда сможете описать с помощью UML всю систему и перейдете в режим использования *UML в качестве языка программирования*. В такой среде разработчики рисуют диаграммы, которые компилируются прямо в исполняемый код, а UML становится исходным кодом. Очевидно, что такое применение UML требует особенно сложных инструментов. (Кроме того, нотации прямой и обратной разработки теряют всякий смысл, поскольку UML и исходный код становятся одним и тем же.)

Один из интересных вопросов, касающихся UML как языка программирования, – это вопрос о моделировании логики поведения. UML 2 предлагает три способа моделирования поведения: диаграммы взаимодействия, диаграммы состояний и диаграммы деятельности. Все они имеют своих сторонников в сфере программирования. Если UML добьется популярности как язык программирования, то будет интересно посмотреть, какой из этих способов будет иметь успех.

Другая точка зрения разработчиков на UML находится где-то между его применением для концептуального моделирования и его применением для моделирования программного обеспечения. Большинство разработчиков используют UML для моделирования программного обеспечения. С *точки зрения программного обеспечения* элементы UML практически непосредственно отображаются в элементы программной системы. Как мы увидим впоследствии, отображение отнюдь не означает следование инструкциям, но когда мы используем UML, мы говорим об элементах программного обеспечения.

С *концептуальной точки зрения* UML представляет описание концепций предметной области. Здесь мы не столько говорим об элементах программного обеспечения, сколько занимаемся созданием словаря для обсуждения конкретной предметной области.

Нет строгих правил выбора точки зрения. Поскольку проблему можно рассматривать под разными углами зрения, то и способов применения существует довольно много. Некоторые инструменты автоматически преобразуют исходный код в диаграммы, трактуя UML как альтернативный вид исходного кода. Это в большей степени программный ракурс. Если же диаграммы UML применяются для того, чтобы проверить

Архитектура, управляемая моделью, и исполняемый UML

Когда говорят о UML, часто упоминают об *MDA* (Model Driven Architecture – архитектура, управляемая моделью) [27]. По сути дела, MDA представляет собой стандартный подход к использованию UML в качестве языка программирования; этот стандарт находится под управлением группы OMG, как и сам UML. Создавая систему моделирования, соответствующую MDA, поставщики могут разработать модели, способные работать и в MDA-совместимом окружении.

Говоря об MDA, часто подразумевают и UML, поскольку MDA использует UML в качестве основного языка моделирования. Но, конечно, вы не обязаны следовать MDA, применяя UML.

MDA разделяет процесс разработки на две основные части. Разработчики моделей представляют конкретное приложение, создавая *PIM* (Platform Independent Model – модель, не зависящая от платформы). PIM – это модель UML, не зависящая от какой-то конкретной технологии. Затем инструменты могут превратить PIM в PSM (Platform Specific Model – модель, зависящая от платформы). PSM – это модель системы, нацеленная на определенную среду выполнения. Другие инструменты берут PSM и генерируют код для этой платформы. В качестве PSM можно использовать UML, но это необязательно.

Поэтому если вы хотите создать систему складского учета с использованием MDA, вам придется начать с единой модели PIM вашей системы. Затем при желании запустить эту систему складского учета в J2EE или .NET вы должны будете использовать инструменты каких-либо производителей для создания двух моделей PSM – по одной на каждую из этих двух платформ.

Если процесс перехода от модели PIM к модели PSM и окончательной программе полностью автоматизирован, то мы используем UML в качестве языка программирования. Если на каком-нибудь этапе присутствует ручная обработка, то мы используем UML в режиме проектирования.

Стив Меллор (Steve Mellor) долгое время активно работал в этой области и с недавнего времени стал употреблять термин *исполняемый UML* (Executable UML) [32]. Исполняемый UML похож на MDA, но использует немного другую терминологию. Точно так же вы начинаете с модели, не зависящей от платформы, которая эквивалентна MDA-модели PIM. Однако на следующем этапе применяется компилятор модели (Model Compiler), для того чтобы за один прием превратить UML-модель в готовую к развертыванию систему; поэтому модель PSM не нужна. Как и предполагает термин *компилятор*, этот этап полностью автоматизирован.

Компиляторы модели основаны на повторно используемых прототипах. *Прототип* (*archetype*) описывает способ превращения исполняемого UML в соответствующую программную платформу. Поэтому в случае с системой складского учета придется купить компилятор модели и два прототипа (J2EE и .NET). Примените эти прототипы к вашему исполняемому UML, и у вас будет две версии системы складского учета.

Исполняемый UML не использует полный стандарт UML; многие конструкции языка считаются ненужными и не применяются. Поэтому исполняемый UML проще, чем полный.

Все это звучит хорошо, но насколько это реалистично? На мой взгляд, здесь есть два аспекта. Во-первых, вопрос об инструментах: достаточно ли они развиты, чтобы выполнить поставленную задачу. Этот фактор со временем меняется; определенно, как я уже писал, они не слишком широко применяются, и я не многие из них видел в действии.

Более фундаментальным аспектом является сама идея применения UML в качестве языка программирования. С моей точки зрения, использовать UML как язык программирования стоит, только если в результате получается нечто более продуктивное, чем в случае применения другого языка программирования. Однако исходя из своего опыта работы в различных графических средах разработки, я не стал бы это утверждать. Даже если UML и более продуктивен, надо еще накопить критическую массу пользователей, чтобы принять его в качестве основного направления. UML сам по себе представляет большое препятствие. Подобно многим пользователям, имеющим опыт работы с языком Smalltalk, я считаю его более продуктивным, чем многие современные основные языки. Но в настоящее время Smalltalk представляет только небольшую нишу в пространстве языков, и я не наблюдаю большого количества проектов, написанных на нем. Чтобы избежать судьбы языка Smalltalk, UML должен быть счастливчиком, даже если он самый лучший.

и понять различные значения терминов «пул активов» (asset pool) с группой бухгалтеров, то следует принять точку зрения значительно более близкую к концептуальной.

В предыдущем издании этой книги я разделил программную точку зрения на спецификацию (specification) и реализацию (implementation). Сейчас я понял, что на практике довольно трудно провести между ними точную границу, поэтому чувствую, что не нужно больше беспокоиться о различиях между ними. Однако я всегда был склонен в своих диаграммах делать ударение на интерфейсе, а не на реализации.

Следствием различных способов применения UML является масса споров о том, что означают диаграммы UML и как они связаны с осталь-

ным миром. Особенно это влияет на отношение между UML и исходным кодом. Некоторые разработчики считают, что UML нужно применять для создания модели, не зависящей от языка программирования, который используется для реализации проекта. Другие убеждены в том, что модель, не зависимая от языка, – это оксюморон с выраженным ударением на слово «морон» (moron – идиот).

Другое различие во взглядах относится к вопросу о сущности UML. По-моему, большинство пользователей UML, особенно создателей эскизов, видят сущность UML в его диаграммах. Однако авторы UML считают диаграммы вторичным, а первичным признают метамодель UML. Диаграммы же – лишь представление метамодели. Такая точка зрения имеет смысл также для разработчиков, использующих UML в режиме проектирования и в режиме языка программирования.

Итак, всякий раз когда вы читаете что-нибудь, относящееся к UML, важно понимать точку зрения автора. Только тогда вы сможете правильно оценить эти часто горячие дискуссии, которые вызывает UML.

Написав это, я должен пояснить свои пристрастия. Практически вся моя работа с UML посвящена созданию эскизов. Я считаю, что эскизы UML полезны и в процессе прямой, и в процессе обратной разработки, а также и с концептуальной точки зрения, и в программном ракурсе.

Я не любитель детальных моделей, созданных в процессе прямого проектирования, поскольку убежден, что они слишком трудно реализуются и замедляют процесс разработки. Создание моделей уровня интерфейсов подсистем вполне разумно, но даже в этом случае вы должны быть готовы к изменению этих интерфейсов, когда разработчики будут реализовывать взаимодействие интерфейсов. Значение моделей в процессе обратной разработки зависит от того, как работает инструментарий. Если он применяется в качестве динамического броузера, то он может быть очень полезным; если же он генерирует большой документ, то вся его работа сводится к пустому переводу бумаги.

Я считаю, что применение UML в качестве языка программирования – удачная идея, но сомневаюсь, что он когда-нибудь будет широко использоваться в этом качестве. Я не уверен, что для большинства программных задач графическое представление более продуктивно, чем текстовое, и даже если это так, то вряд ли такой язык получит широкое распространение.

В соответствии с моими пристрастиями эта книга посвящена, главным образом, использованию UML для создания эскизов. К счастью, это имеет смысл при написании краткого руководства. Я не в состоянии показать все возможности других режимов работы UML в книге такого объема, но эта небольшая книга является хорошим введением в другие книги, которые могут решить эту задачу. Поэтому если вас интересуют другие режимы использования UML, я предлагаю считать эту книгу введением и обратиться к другим книгам, когда это будет необ-

ходимо. Для тех же, кого интересуют только эскизы, данное издание может оказаться именно тем, что нужно.

Как мы пришли к UML

Надо признать, что я люблю историю. Мое любимое легкое чтение – это хорошая историческая книга. Но я понимаю, что это нравится не всем. Я говорю здесь об истории, поскольку считаю, что во многих отношениях трудно оценить место, занимаемое UML, не зная истории его развития.

В 80-х годах объекты начали выходить из исследовательских лабораторий и делать свои первые шаги в направлении «реального» мира. Язык Smalltalk был реализован на платформе, пригодной для практического использования; появился на свет и C++. В то же время разработчики начали задумываться об объектно-ориентированных языках графического моделирования.

Основные книги по объектно-ориентированным языкам графического моделирования появились между 1988 и 1992 годами. Ведущими фигурами в этом деле были Гради Буч (Grady Booch) [5]; Питер Коуд (Peter Coad) [7], [8]; Айвар Джекобсон (Ivar Jacobson) (Objectory) [24]; Джим Оделл (Jim Odell) [34]; Джим Рамбо (Jim Rumbaugh) (OMT) [38], [39]; Салли Шлаер (Sally Shlaer) и Стив Меллор (Steve Mellor) [41], [42]; Ребекка Вирфс-Брок (Rebecca Wirfs-Brock) (Responsibility Driven Design) [44].

Каждый из перечисленных авторов в то время был неформальным лидером группы профессионалов, приверженцев этих идей. Все эти методы были очень похожи, хотя между ними и существовали небольшие, но нередко раздражающие отличия. Идентичные базовые концепции обязательно проявлялись в самых разнообразных нотациях, которые вызывали путаницу в головах моих клиентов.

В это горячее время о стандартизации говорили в той же степени, в какой ее игнорировали. Команда из группы OMG пыталась разбираться со стандартизацией, но в результате получала только письма с протестами от всех ведущих методологов. (Есть такой старый анекдот. Вопрос: В чем разница между методологом и террористом? Ответ: С террористом можно вести переговоры.)

Катастрофическим событием, инициировавшим появление UML, стали уход Джима Рамбо из GE и его встреча с Гради Бучем в Rational (теперь это подразделение IBM). С самого начала было ясно, что союз Буч/Рамбо может создать критическую массу[1] этого бизнеса. Гради и Джим провозгласили, что «война методов завершена – мы победили»,

[1] Критическая масса (в маркетинге) – обязательный набор новшеств, которые должны быть присущи товару, чтобы он считался современным. – *Примеч. ред.*

по существу заявляя, что они собираются достигнуть стандартизации, пойдя по «пути Microsoft». Некоторые методологи проектирования предложили создать анти-Бучевскую коалицию.

К конференции OOPSLA '95 Гради и Джим подготовили свое первое публичное описание их совместного метода – версию 0.8 документации по *Унифицированному методу* (*Unified Method*). И что более существенно, они объявили, что фирма Rational Software купила Objectory и что Айвар Джекобсон должен присоединиться к Унифицированной команде. Rational устроила шикарный прием, празднуя выход версии 0.8. (Гвоздем программы стало первое публичное выступление поющего Джима Рамбо; мы все надеемся, что оно также было и последним.)

На следующий год процесс стал более открытым. Группа OMG, которая раньше в основном стояла в стороне, теперь стала играть активную роль. Rational вынуждена была принять идеи Айвара, а также работать с другими партнерами. Более существенно то, что группа OMG решила взять на себя ведущую роль.

Важно понять, почему вступила в дело группа OMG. Методологи проектирования, как и авторы книг, любят думать, что они важные персоны. Но я не думаю, что вопли авторов книг могут быть хотя бы услышаны группой OMG. Причиной участия группы OMG стали крики поставщиков программных средств. Поставщики испугались, что стандарт, контролируемый фирмой Rational, даст продуктам фирмы неоправданное конкурентное преимущество. В результате производители побудили группу OMG к действиям под флагом необходимости взаимодействия CASE-систем. Это был очень важный аргумент, поскольку для группы OMG взаимодействие значило очень много. Появилась идея создать язык UML, который бы позволил CASE-средствам свободно обмениваться моделями.

Мэри Лумис (Mary Loomis) и Джим Оделл возглавили инициативную группу, созданную для решения поставленной задачи. Оделл дал ясно понять, что готов предоставить свой метод для стандарта, но не желает поддерживать стандарт, продиктованный фирмой Rational. В январе 1997 года различные организации представили на рассмотрение свои предложения по стандартизации методов, которые бы способствовали обмену моделями. Фирма Rational сотрудничала с несколькими организациями и в соответствии со своим планом представила версию 1.0 документации по UML – первое создание, соответствующее имени Унифицированный язык моделирования (Unified Modeling Language).

Затем последовал короткий период выкручивания рук, когда объединялись различные предложения. Группа OMG приняла окончательную версию 1.1 в качестве официального стандарта OMG. Позднее были созданы другие версии. Версия 1.2 была целиком косметическая, версия 1.3 – более значимой. В версию 1.4 было введено множество детализированных понятий о компонентах и профилях, а в версию 1.5 добавили семантику действия.

Когда специалисты говорят о UML, они называют его создателями главным образом Гради Буча, Айвара Джекобсона и Джима Рамбо. Чаще всего их называют «Трое друзей» (Three Amigos), хотя шутники любят опускать первый слог второго слова. Несмотря на то что они заработали свое доброе имя в основном в связи с UML, я думаю, что не совсем справедливо отдавать им всю славу. Нотация UML впервые была сформирована в Унифицированном методе Буча/Рамбо. С тех пор была проведена большая работа в комитетах группы OMG. На этой стадии Джим Рамбо был лишь одним из тройки, которая выполнила свои тяжелые обязательства. Я считаю, что основную благодарность за создание UML заслужили именно члены комитета UML.

Нотации и метамодели

Язык UML в своем нынешнем состоянии определяет нотацию и метамодель. **Нотация** представляет собой совокупность графических элементов, которые применяются в моделях; она и есть синтаксис данного языка моделирования. Например, нотация диаграммы классов определяет способ представления таких элементов и понятий, как класс, ассоциация и кратность.

Конечно, при этом возникает вопрос точного определения смысла ассоциации, кратности и даже класса. Общепринятое употребление этих понятий предполагает некоторые неформальные определения, однако многие разработчики испытывают потребность в более строгом определении.

Идея строгой спецификации и языков проектирования наиболее распространена в области формальных методов. В таких методах модели и спецификации представляются с помощью некоторых производных средств исчисления предикатов. Соответствующие определения математически строги и исключают неоднозначность. Однако эти определения нельзя считать универсальными. Даже если вы сможете доказать, что программа соответствует математической спецификации, не существует способа доказать, что эта математическая спецификация действительно удовлетворяет реальным требованиям системы.

Большинство графических языков моделирования являются отнюдь не строгими; их нотация в большей степени апеллирует к интуиции, чем к формальному определению. В целом это не выглядит таким уж большим недостатком. Хотя подобные методы могут быть неформальными, многие разработчики по-прежнему считают их полезными, и это нельзя не принимать во внимание.

Однако методологи ищут способы добиться большей строгости методов, не жертвуя при этом их практической полезностью. Один из способов заключается в определении некоторой **метамодели** – диаграммы (как правило, диаграммы классов), определяющей понятия языка.

На рис. 1.1 изображена небольшая часть метамодели языка UML, которая показывает отношение между свойствами. (Этот фрагмент приведен с единственной целью – дать лишь общее представление о том, что такое метамодель. Я даже не буду пытаться давать здесь какие-либо дополнительные пояснения.)

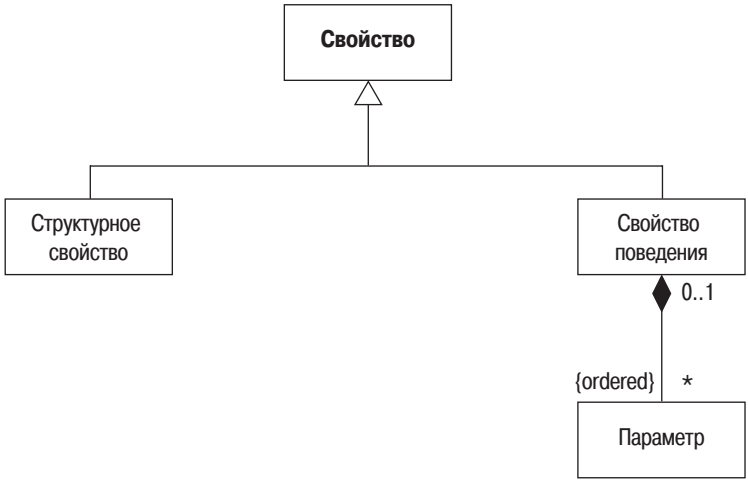

Рис. 1.1. *Фрагмент метамодели UML*

Насколько велико влияние метамодели на того, кто применяет соответствующую нотацию при моделировании? Ответ зависит, главным образом, от режима работы с языком. Создателя эскизов обычно это не слишком волнует; проектировщика это должно беспокоить значительно больше. И это жизнено важно для тех, кто использует UML в качестве языка программирования, поскольку метамодель определяет абстрактный синтаксис данного языка.

В настоящее время многие люди, вовлеченные в разработку UML, интересуются в основном метамоделью, в частности потому, что она важна при использовании UML и языка программирования. Вопросы нотации часто стоят на втором месте, что важно помнить, если вы собираетесь поближе познакомиться с документацией по стандарту.

Когда вы глубже погрузитесь в изучение UML, то поймете, что вам требуется значительно больше, чем просто графическая нотация. Вот почему инструменты UML так сложны.

В этой книге я не слишком строг. Я предпочитаю традиционные пути и обращаюсь к вашей интуиции. Это естественно для такой маленькой книжки, написанной автором, склонным, в основном, работать в режиме эскизного моделирования. Приверженцам большей строгости следует обратиться к более обстоятельным трудам.

Диаграммы UML

UML 2 описывает 13 официальных типов диаграмм, перечисленных в табл. 1.1, классификация которых приведена на рис. 1.2. Хотя эти виды диаграмм отражают различные подходы многих специалистов к UML и способ организации моей книги, авторы UML не считают диаграммы центральной составляющей языка. Поэтому диаграммы определены не очень строго. Часто вполне допустимо присутствие элементов диаграммы одного типа в другой диаграмме. Стандарт UML указывает, что определенные элементы обычно рисуются в диаграммах соответствующего типа, но это не догма.

Таблица 1.1. Официальные типы диаграмм UML

Диаграмма	Глава книги	Цель	Происхождение
Деятельности	11	Процедурное и параллельное поведение	В UML 1
Классов	3, 5	Классы, свойства и отношения	В UML 1
Взаимодействия	12	Взаимодействие между объектами; акцент на связях	Диаграмма кооперации в UML 1
Компонентов	14	Структура и взаимосвязи между компонентами	В UML 1
Составных структур	13	Декомпозиция класса во время выполнения	Новое в UML 2
Развертывания	8	Развертывание артефактов в узлы	В UML 1
Обзора взаимодействий	16	Комбинация диаграммы последовательности и диаграммы деятельности	Новое в UML 2
Объектов	6	Вариант конфигурации экземпляров	Неофициально в UML 1
Пакетов	7	Иерархическая структура времени компиляции	Неофициально в UML 1
Последовательности	4	Взаимодействие между объектами; акцент на последовательности	В UML 1
Конечных автоматов	10	Как события изменяют объект в течение его жизни	В UML 1
Временная	17	Взаимодействие между объектами; акцент на синхронизации	Новое в UML 2
Прецедентов	9	Как пользователи взаимодействуют с системой	В UML 1

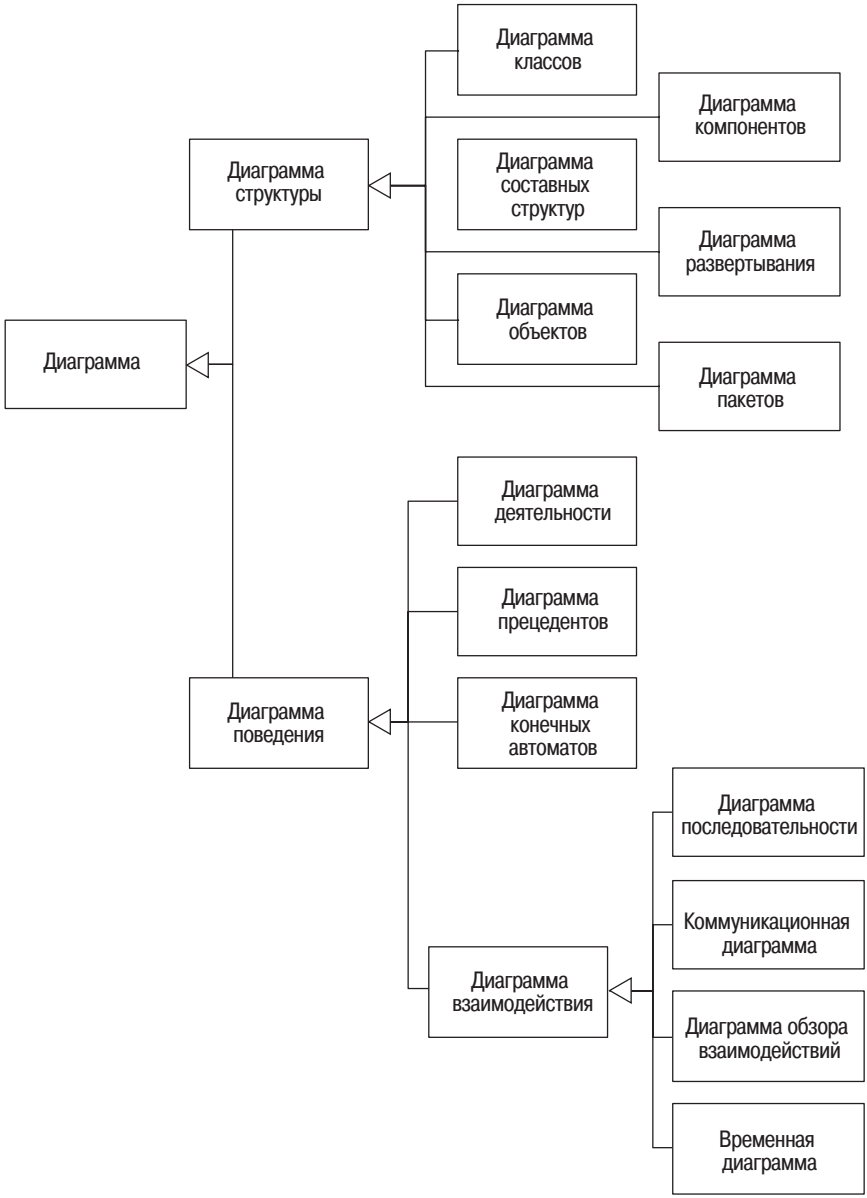

***Рис. 1.2.** Классификация типов диаграмм UML*

Что такое допустимый UML?

На первый взгляд, ответить на этот вопрос легко: допустимый UML – это язык, определенный в соответствии со спецификацией. Однако на практике ответ несколько сложнее.

Существенным в вопросе является то, на каких правилах базируется UML: описательных или предписывающих. Язык с **предписывающими правилами** (prescriptive rules) управляется официальной основой, которая устанавливает, что является, а что не является допустимым языком, и какое значение вкладывается в понятие высказывания языка. Язык с **описательными правилами** (descriptive rules) – это язык, правила которого распознаются по тому, как люди применяют его на практике. Языки программирования в основном имеют предписывающие правила, установленные комитетом по стандартам или основными поставщиками, тогда как естественные языки, такие как английский, в основном имеют описательные правила, смысл которых устанавливается по соглашению.

UML – точный язык, поэтому можно было бы ожидать, что он основан на предписывающих правилах. Но UML часто рассматривают как программный эквивалент чертежей из других инженерных дисциплин, а эти чертежи основаны не на предписывающих нотациях. Никакой комитет не говорит, какие символы являются законными в строительной технической документации; эти нотации были приняты по соглашению, как и в естественном языке. Стандарты сами по себе еще ничего не решают, поскольку те, кто работает в этой области, не смогут следовать всему, что указывают стандарты; это то же самое, что спрашивать французов о французской академии наук. К тому же язык UML настолько сложен, что стандарты часто можно трактовать по-разному. Даже ведущие специалисты по UML, которые рецензировали эту книгу, не согласились бы интерпретировать стандарты.

Этот вопрос важен и для меня, пишущего эту книгу, и для вас, применяющих язык UML. Если вы хотите понять диаграммы UML, важно уяснить, что понимание стандартов – это еще не вся картина. Люди принимают соглашения и в индустрии в целом, и в каких-то конкретных проектах. Поэтому, хотя стандарт UML и может быть первичным источником информации по UML, он не должен быть единственным.

Моя позиция состоит в том, что для большинства людей UML имеет описательные правила. Стандарт UML оказывает наибольшее влияние на содержание UML, но это делает не только он. Я думаю, что особенно верным это станет для UML 2, который вводит некоторые соглашения по обозначениям, конфликтующие или с определениями UML 1, или с использованием по соглашению, а также еще больше усложняет язык. Поэтому в данной книге я стараюсь обобщить UML так, как я его вижу: и стандарты, и применение по соглашению. Когда мне приходится указывать на некоторое отличие в этой книге, я буду употреблять термин **применение по соглашению** (conventional use), чтобы обозначить то, чего нет в стандарте, но, как я думаю, широко применяется. В случае если что-то соответствует стандарту, я буду употреблять термин **стандартный** (standard) или **нормативный** (normative). (Нормативный – это термин, посредством которого люди обозначают утвер-

ждение, которое вы должны подтвердить, чтобы оно соответствовало стандарту. Поэтому выражение ненормативный UML – это своеобразный способ сказать, что нечто совершенно неприемлемо с точки зрения стандарта UML.)

Рассматривая диаграмму UML, необходимо помнить, что основной принцип UML заключается в том, что любая информация на конкретной диаграмме может быть подавлена. Это подавление может носить глобальный характер – скрыть все атрибуты – или локальный – не показывать какие-нибудь конкретные классы. Поэтому по диаграмме вы никогда не можете судить о чем-нибудь по его отсутствию. Даже если метамодель UML имеет поведение по умолчанию, например [1] для атрибутов, когда вы не видите эту информацию на диаграмме, это может быть обусловлено либо поведением по умолчанию, либо тем, что она просто подавлена.

Говоря это, следует упомянуть, что существуют основные соглашения, например о том, что многозначные свойства должны быть множествами.

Не надо слишком зацикливаться на допустимом UML, если вы занимаетесь эскизами или моделями. Важнее составить хороший проект системы, и я предпочитаю иметь хороший дизайн в недопустимом UML, чем допустимый UML, но плохой дизайн. Очевидно, хороший и допустимый предпочтительнее, но лучше направить свою энергию на разработку хорошего проекта, чем беспокоиться о секретах UML. (Конечно, в случае применения UML в качестве языка программирования необходимо соблюдать стандарты, иначе программа будет работать неправильно!)

Смысл UML

Одним из затруднений в UML является то, что хотя спецификация подробно описывает определение правильно сформированного UML, но этого недостаточно, чтобы определить значение UML вне сферы изысканного выражения «метамодель UML». Не существует формальных описаний того, как UML отображается на конкретные языки программирования. Вы не можете посмотреть на диаграмму UML и *точно* сказать, как будет выглядеть соответствующий код. Однако у вас может быть *приблизительное представление* о виде программы. На практике этого достаточно. Команды разработчиков часто формируют собственные локальные соглашения, и, чтобы их использовать, вам придется с ними познакомиться.

UML не достаточно

UML предоставляет довольно большое количество различных диаграмм, помогающих описать приложение, но это отнюдь не полный

список всех полезных диаграмм, с которыми вам, возможно, придется работать. Во многих случаях полезными могут оказаться различные диаграммы, и не надо избегать диаграмм, не имеющих отношения к UML, если не нашлось диаграмм UML, подходящих для ваших целей.

На диаграмме потока экранов (рис. 1.3) показаны различные экраны интерфейса пользователя и способы перемещения по ним. Я изучал и использовал диаграммы потока экранов многие годы и не встречал ничего, кроме очень приблизительных определений того, что они означают. В UML нет ничего подобного этим диаграммам, но я по-прежнему считаю их очень полезными.

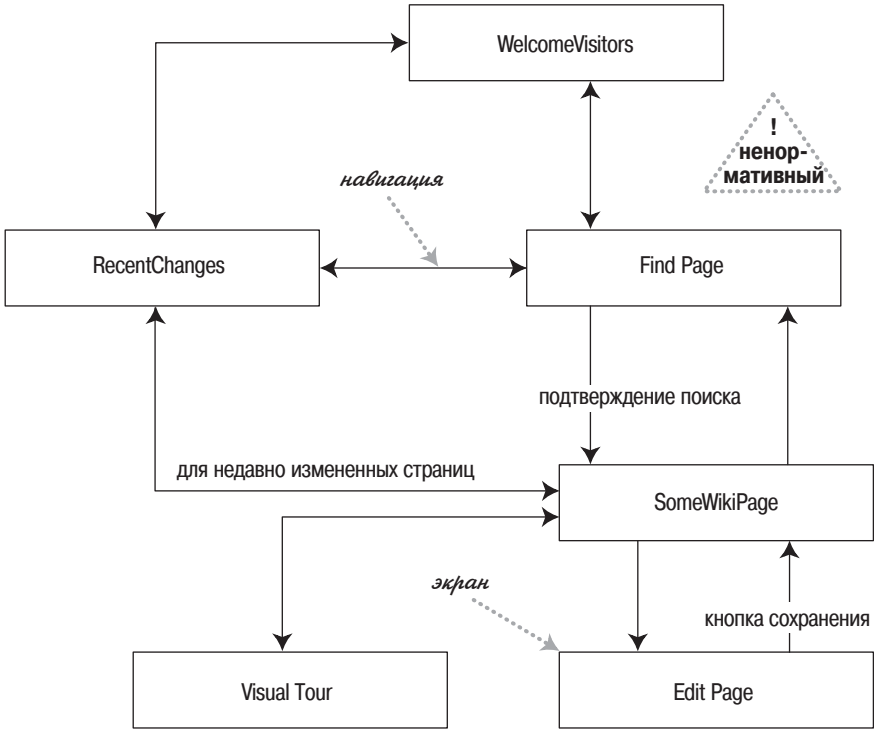

Рис. 1.3. *Неформальная диаграмма потока экранов для части wiki (http://c2.com/cgi/wiki)*

В табл. 1.2 представлен другой мой любимец – таблица решений. Таблица решений – это хороший способ показать сложные логические условия. Это можно реализовать с помощью диаграммы деятельности, но как только вы выходите за рамки простых случаев, таблица решений становится компактнее и проще для понимания. Как и диаграммы потока экранов, многие виды таблиц решений не представлены в языке. Таблица 1.2 разделена на две части: логические условия, расположенные выше двойной черты, и их результаты внизу таблицы.

Каждый столбец показывает, как конкретная комбинация условий приводит к определенному множеству результатов.

Таблица 1.2. Таблица решений

Специальный клиент	X	X	Y	Y	N	N
Приоритетный заказ	Y	N	Y	N	Y	N
Международный заказ	Y	Y	N	N	N	N
Плата	$150	$100	$70	$50	$80	$60
Предупредительный сигнал	•	•	•			

В разных книгах вы встретите различные варианты таких вещей. Не стесняйтесь пробовать приемы, которые кажутся вам подходящими для вашего проекта. Если они работают, пользуйтесь ими. Если нет – забудьте о них. (Этот же совет, конечно, относится и к диаграммам UML.)

С чего начать

Никто, даже создатели UML, не понимают всего UML и не используют все его возможности. Большинство разработчиков при работе задействуют лишь небольшое подмножество UML. Вы должны найти свое подмножество, которое бы подходило для решения задач, стоящих перед вами и вашими коллегами.

Тем, кто только начинает, я советую на первых порах сконцентрироваться на основных формах диаграмм классов и диаграмм последовательности. На мой взгляд, это наиболее общие и самые полезные типы диаграмм.

Постигнув их суть, вы сможете приступить к применению более продвинутых нотаций диаграмм классов и взглянуть на другие типы диаграмм. Экспериментируйте с диаграммами и выясните, насколько они полезны для вас. Не бойтесь отбросить любые из них, которые не кажутся вам полезными для вашей работы.

2

Процесс разработки

Как я уже говорил, UML вырос из группы методов объектно-ориентированного анализа и дизайна. До некоторой степени все они представляют собой комбинацию графического языка моделирования и процесса, в котором определяются подходы к разработке программного обеспечения.

Интересно, что когда UML сформировался, многие участники обнаружили, что хотя они и могут принять язык моделирования, но определенно не могут согласиться с процессом. В результате они согласились перенести все соглашения по процессу на более позднее время и ограничить применение UML областью языка моделирования.

Эта книга называется «UML. Основы», поэтому я могу оставить в покое проигнорированный процесс. Однако я не думаю, что приемы моделирования имеют смысл без понимания того, как они соответствуют процессу. Способы применения UML в значительной степени зависят от типа процесса, с которым вы работаете.

Таким образом, я считаю, что необходимо сначала поговорить о процессе, чтобы вы смогли увидеть контекст использования UML. Я не собираюсь подробно описывать какой-либо конкретный процесс; я просто хочу дать вам достаточно информации, чтобы вы смогли увидеть этот контекст, и показать, где вы можете узнать больше.

В беседах о UML можно часто услышать упоминание о RUP (Rational Unified Process – унифицированный процесс, созданный компанией Rational). RUP – это один из процессов, точнее говоря, процесс-структура, который вы можете использовать с UML. Но кроме общего участия различных сотрудников фирмы Rational и названия «унифицированный», он не имеет особенного отношения к UML. Язык UML можно использовать с любым процессом. RUP является популярным подходом; он обсуждается на *стр. 52*.

Процессы итеративные и водопадные

Самая бурная дискуссия о процессах разворачивается вокруг выбора между итеративной и водопадной моделями. Часто эти термины употребляются неправильно, в частности потому, что применение итеративного процесса вошло в моду, а водопадный процесс считается чем-то вроде брюк в клеточку. В результате процесс разработки многих проектов объявляется итеративным, хотя в действительности он является процессом водопадного типа.

Существенная разница между двумя этими типами проявляется в том, каким образом проект делится на более мелкие части. Если предполагается, что разработка проекта займет год, то мало кто сможет с легким сердцем сказать людям, что им надо уйти на год и вернуться, когда все будет сделано. Необходимо некоторое разделение, чтобы разработчики смогли найти подход к решению проблемы и наладить работу.

При организации работы в стиле **водопада** проект делится на основании вида работ. Чтобы создать программное обеспечение, необходимо предпринять определенные действия: проанализировать требования, создать проект, выполнить кодирование и тестирование. Наш годичный проект может включать двухмесячную фазу анализа, за которой следует четырехмесячная фаза дизайна, а затем трехмесячная фаза кодирования и, наконец, трехмесячная фаза тестирования.

Итеративный стиль делит проект по принципу функциональности продукта. Можно взять год и разделить его на трехмесячные итерации. В первой итерации берется четверть требований и выполняется полный цикл разработки программного обеспечения для этой четверти: анализ, дизайн, кодирование и тестирование. К концу первой итерации у вас есть система, обладающая четвертью необходимой функциональности. Затем вы приступаете ко второй итерации и через шесть месяцев получаете систему, делающую половину того, что ей положено.

Естественно, приведенное выше описание упрощено, но в этом состоит суть различия. Конечно, на практике к течению процесса добавляются непредвиденные вредные ручейки.

При разработке способом водопада после каждого этапа обычно в каком-либо виде выполняется формальная сдача, но часто имеет место возвращение назад. В процессе кодирования могут выясниться обстоятельства, вынуждающие снова вернуться к этапам анализа и дизайна. Конечно, в начале кодирования не следует думать, что анализ завершен. И решения, принятые на стадии анализа и дизайна, неизбежно будут пересматриваться позднее. Однако эти обратные потоки представляют собой исключения и должны быть по возможности сведены к минимуму.

При итеративном процессе разработки перед началом реальной итерации обычно наблюдается некоторая исследовательская активность.

Как минимум на требования будет брошен поверхностный взгляд, достаточный, по крайней мере, для разделения требований на итерации для последующего выполнения. В процессе такого исследования могут быть приняты некоторые решения по дизайну самого высшего уровня. С другой стороны, несмотря на то что в результате каждой итерации должно появиться интегрированное программное обеспечение, готовое к поставке, часто бывает, что оно еще не готово и нужен некоторый стабилизационный период для исправления последних ошибок. Кроме того, некоторые виды работ, такие как тренировка пользователей, оставляются на конец.

Конечно, вы вполне можете не передавать систему на реализацию в конце каждой итерации, но она должна находиться в состоянии производственной готовности. Однако часто бывает, что система сдается на регулярной основе; это хорошо, поскольку вы оцениваете работоспособность системы и получаете более качественную обратную реакцию. В этой ситуации часто говорят о проекте, имеющем несколько версий, каждая из которых делится на несколько **итераций**.

Итеративную разработку называют по-разному: инкрементной, спиральной, эволюционной и постепенной. Разные люди вкладывают в эти термины разный смысл, но эти различия не имеют широкого признания и не так важны, как противостояние итеративного метода и метода водопада.

Возможен и смешанный подход. В книге Мак-Коннелла [31] описывается жизненный цикл **поэтапной доставки** (staged delivery), в соответствии с которым сначала выполняются анализ и проектирование верхнего уровня в стиле водопада, а затем кодирование и тестирование, разделенные на итерации. В таком проекте может быть четырехмесячный этап анализа и дизайна, а затем четыре двухмесячные итерации построения системы.

Большинство авторов публикаций по процессу создания программного обеспечения, особенно принадлежащие к объектно-ориентированному сообществу, последние пять лет испытывают неприязнь к подходу в стиле водопада. Из всего множества причин этого явления самая главная заключается в том, что при использовании метода водопада очень трудно утверждать, что разработка какого-то проекта действительно идет в верном направлении. Слишком легко объявить победу на раннем этапе и скрыть ошибки планирования. Обычно единственный способ, которым вы действительно можете показать, что следуете по такому пути, состоит в том, чтобы получить протестированное, интегрированное программное обеспечение. В случае итеративного процесса это повторяется многократно, и в результате, если что-то идет не так, как надо, мы своевременно получаем соответствующий сигнал.

Хотя бы только по этой причине я настоятельно рекомендую избегать метода водопада в чистом виде. По крайней мере, необходимо приме-

нять поэтапную доставку, если невозможно использовать итеративный метод в полном объеме.

Объектно-ориентированное сообщество долгое время было привержено итеративному способу разработки, и, не боясь ошибиться, можно сказать, что значительная часть разработчиков, участвующих в создании UML, предпочитают в том или ином виде итеративную разработку. Однако, по моим ощущениям, в практике программной индустрии метод водопада все еще занимает ведущее положение. Одну из причин этого явления я называю псевдоитеративной разработкой: исполнители объявляют, что ведут итеративное проектирование, а на самом деле действуют в стиле водопада. Основные симптомы этого следующие:

- «Мы выполняем одну итерацию анализа, а затем две итерации проектирования...»
- «Код данной итерации содержит много ошибок, но мы исправим их в конце...»

Особенно важно, чтобы каждая итерация завершалась созданием протестированного, интегрированного программного продукта, который бы имел качество, как можно более близкое к качеству серийной продукции. Тестирование и интеграцию оценить труднее всего, поэтому в конце разработки проекта лучше эти этапы не проводить. Процесс тестирования следует организовать так, чтобы любая итерация, не объявленная в плане как сдаточная, могла бы быть переведена в такой статус без серьезных дополнительных усилий разработчиков.

Общим приемом при итеративной разработке является **упаковка по времени** (time boxing). Таким образом, итерация будет занимать фиксированный промежуток времени. Если обнаружилось, что вы не в состоянии выполнить все, что планировали сделать за время итерации, то необходимо выбросить некоторую функциональность из данной итерации, но не следует изменять дату исполнения. В большинстве проектов, основанных на итеративном процессе, протяженность итераций одинакова, и это позволяет вести разработку в постоянном ритме.

Мне нравится упаковка по времени, поскольку люди обычно испытывают трудности при сокращении функциональности. Регулярно практикуясь в сокращении функциональности, они учатся делать осмысленный выбор между изменением времени разработки и изменением функциональности. Сокращение функциональности в ходе итерации позволяет также людям научиться расставлять приоритеты между требованиями к проекту.

Одним из наиболее общих аспектов итеративной разработки является вопрос переделки. Итеративная разработка недвусмысленно предполагает, что вы будете перерабатывать и удалять существующий код на последней итерации проекта. Во многих областях человеческой деятельности, например в промышленном производстве, переделка считается ущербом. Но создание программного обеспечения не похоже на

промышленное производство – часто бывает выгоднее переработать существующий код, чем латать код неудачно спроектированной программы. Некоторые технические приемы способны оказать существенную помощь, чтобы процесс переделки был более эффективным.

- **Автоматизированные регрессивные тесты** (automated regression tests) позволяют быстро найти любые ошибки, внесенные в процессе изменений. Семейство оболочек тестирования xUnit представляет наиболее подходящий инструмент для создания автоматизированных тестов для модульной проверки (unit tests). Начиная с исходного JUnit *http://junit.org*, здесь есть порты для почти всех возможных языков (*http://www.xprogramming.com/software.htm*). Хорошим правилом является создание модульных тестов примерно такого же размера, что и тестируемая программа.
- **Рефакторинг** (refactoring) – это способ изменения существующего программного продукта [20]. Механизм рефакторинга основан на применении серии небольших, сохраняющих поведение трансформаций основного кода. Многие из этих трансформаций могут быть автоматизированы (*http://www.refactoring.com*).
- **Последовательная интеграция** (continuous integration) сохраняет синхронизацию действий разработчиков, объединенных в команду, что позволяет избежать болезненных циклов интеграции [18]. В ее основе лежит полностью автоматизированный процесс построения, который может быть автоматически прекращен, как только любой член команды начнет записывать код в базу. Предполагается, что разработчики записывают код ежедневно, поэтому автоматические сборки выполняются несколько раз в день. Процесс построения включает запуск большого блока автоматических регрессионных тестов, что позволяет быстро обнаружить и без труда исправить любую несогласованность.

Все эти технические приемы пропагандировались недавно в книге «Extreme Programming» [2], хотя они применялись и раньше и могли (и должны были) применяться, независимо от того, использовался ли XP (eXtreme Programming) или какой-либо другой гибкий процесс.

Прогнозирующее и адаптивное планирование

Одна из причин, по которым метод водопада еще жив, заключается в желании обеспечить предсказуемость при создании программного обеспечения. Ничто так не раздражает, как отсутствие точной оценки стоимости создания программного продукта и сроков его разработки.

Прогнозирующий подход направлен на выполнение работы на начальном этапе проекта, для того чтобы лучше понять, что нужно делать в дальнейшем. Таким образом, наступает момент, когда оставшуюся часть проекта можно оценить с достаточной степенью точности. В процессе **прогнозирующего планирования** (predictive planning) проект

разделяется на две стадии. На первой стадии составляются планы, и тут предсказывать трудно, но вторая стадия более предсказуема, поскольку планы уже готовы.

При этом не надо все делить на белое и черное. В процессе выполнения проекта вы постепенно добиваетесь большей предсказуемости. И даже если у вас есть план, все может пойти не так, как вы спрогнозировали. Вы просто ожидаете, что при наличии четкого плана отклонения будут менее значительными.

Однако все еще идут острые дискуссии о том, много ли проектов могут быть предсказуемыми. Сущность данного вопроса состоит в анализе требований. Одна из самых существенных причин сложности программных проектов заключается в трудности понимания требований к программным системам. Большинство программных проектов подвергаются существенному **пересмотру требований** (requirements churn): изменению требований на поздней стадии выполнения проекта. Такой пересмотр вдребезги разбивает основу прогнозов. Последствия пересмотра можно предотвратить, заморозив требования на ранней стадии проекта и не позволяя изменениям появляться, но это приводит к риску поставить клиенту систему, которая больше не удовлетворяет требованиям пользователей.

Эта проблема приводит к двум различным вариантам действий. Один путь – это направить больше усилий собственно на проработку требований. Другой путь состоит в получении более определенного множества требований, чтобы сократить возможные изменения.

Приверженцы другой школы утверждают, что пересмотр требований неизбежен, что во многих проектах трудно стабилизировать требования в такой степени, чтобы имелась возможность использовать прогнозирующее планирование. Это может быть либо следствием того, что исключительно трудно представить, что может делать программный продукт, либо следствием того, что условия рынка диктуют непредсказуемые изменения. Эта школа поддерживает **адаптивное планирование** (adaptive planning) в соответствии с утверждением, что прогнозируемость – это иллюзия. Вместо того чтобы дурачить себя иллюзорной предсказуемостью, мы должны повернуться лицом к реальности постоянных изменений и использовать такой подход в планировании, при котором изменение в проекте считается величиной постоянной. Это изменение контролируется таким образом, чтобы в результате выполнения проекта поставлялось как можно лучшее программное обеспечение; но хотя проект и контролируем, предсказать его нельзя.

Различие между прогнозирующими и адаптивными проектами проявляется разными путями, когда люди говорят о состоянии проекта. Когда утверждается, что выполнение проекта идет хорошо, поскольку работа ведется в соответствии с планом, то имеется в виду метод прогнозирования. При адаптивной разработке нельзя сказать «в соответствии с планом», поскольку план все время меняется. Это не означает, что

адаптивные проекты не планируются; обычно планирование занимает значительное время, но план трактуется как основная линия проведения последовательных изменений, а не как предсказание будущего.

На основе прогнозирующего плана можно разработать контракт с фиксированной функциональностью по фиксированной цене. В таком контракте точно указывается, что должно быть создано, сколько это стоит и когда продукт будет поставлен. В адаптивном плане такое фиксирование невозможно. Вы можете обозначить бюджет и сроки поставки, но вы не можете точно зафиксировать функциональность поставляемого продукта. Адаптивный контракт предполагает, что пользователи будут сотрудничать с командой разработчиков, чтобы регулярно пересматривать требуемую функциональность и прерывать проект, если прогресс слишком незначителен. Как таковой процесс адаптивного планирования может определять проект с переменными границами функциональности по фиксированной цене.

Естественно, адаптивный подход менее желателен, поскольку все предпочитают большую предсказуемость программных проектов. Однако предсказуемость зависит от точности, корректности и стабильности множества требований. Если вы не в состоянии стабилизировать свои требования, то прогнозирующий план базируется на песке, а изменения настолько значительны, что проект сбивается с курса. Отсюда вытекают два важных совета.

1. Не составляйте прогнозирующий план до тех пор, пока не получите точные и корректные требования и не будете уверены, что они не подвергнутся существенным изменениям.
2. Если вы не можете получить точные, корректные и стабильные требования, то используйте метод адаптивного планирования.

Предсказуемость и адаптивность предоставляют выбор жизненного цикла. Адаптивное планирование совершенно определенно подразумевает итеративный процесс. Прогнозирующий план может быть выполнен любым из двух способов, хотя за его выполнением легче наблюдать в случае применения метода водопада, или метода поэтапной поставки.

Гибкие процессы

За последние несколько лет вырос интерес к гибким процессам разработки программного обеспечения. *Гибкий* (*agile*) – это широкий термин, охватывающий большое количество процессов, имеющих общее множество величин и понятий, определенных Манифестом гибкой разработки программного обеспечения (Manifesto of Agile Software Development) (*http://agileManifesto.org*). Примерами таких процессов являются XP (Extreme Programming – экстремальное программирование), Scrum (столкновение), FDD (Feature Driven Development – разработка, управляемая возможностями), Crystal (кристалл) и DSDM (Dynamic Systems Development Method – метод разработки динамических систем).

В терминах нашего обсуждения гибкие процессы исключительно адаптивны по своей природе. Они также имеют четкую ориентацию на человека. Гибкие подходы предполагают, что наиболее важным фактором успешного завершения проекта является квалификация исполнителей и их хорошая совместная работа с человеческой точки зрения. Значимость процессов или инструментов, ими используемых, определенно стоит на втором месте.

Гибкие методы в основном направлены на использование коротких, ограниченных по времени итераций, чаще всего заканчивающихся через месяц или раньше. Поскольку их вклад в документацию невелик, то в гибком подходе не предполагается применение UML в режиме проектирования. Чаще всего UML используется в режиме эскизирования и реже в качестве языка программирования.

В большинстве своем гибкие процессы не слишком **формализованы**. Сильно формализованные или тяжеловесные процессы имеют много документации и постоянный контроль во время выполнения проекта. Гибкий подход предполагает, что формализм мешает проведению изменений и противоречит природе талантливых личностей. Поэтому гибкие процессы часто называют **облегченными** (lightweight). Важно понимать, что недостаточная формализованность является следствием адаптивности и ориентации специалистов, а не фундаментальным свойством.

Унифицированный процесс от Rational

Хотя унифицированный процесс, разработанный компанией Rational (Rational Unified Process, RUP), не зависит от UML, их часто упоминают вместе. Поэтому я думаю, что будет уместно сказать здесь об этом несколько слов.

Хотя RUP называется процессом, в действительности это оболочка процессов, предоставляющая словарь и свободную структуру для обсуждения процессов. В случае применения RUP в первую очередь необходимо выбрать **шаблон разработки** (development case) – процесс, который вы собираетесь использовать в проекте. Шаблоны разработки могут очень значительно варьироваться, поэтому не думайте, что ваш шаблон разработки будет сильно похож на другие шаблоны. При выборе шаблона разработки сразу требуется человек, хорошо знакомый с RUP, – тот, кто сможет приспособить RUP к определенным требованиям проекта. В качестве альтернативы существует постоянно увеличивающийся набор распределенных по пакетам шаблонов разработки, с которых можно начать.

Независимо от шаблона разработки RUP по существу является итеративным процессом. Метод водопада не совместим с философией RUP, хотя с прискорбием должен отметить, что проекты, в которых применяются процессы в стиле водопада, нередко обряжают в одежды RUP.

Все RUP-проекты должны иметь четыре фазы.

1. **Начало** (inception). На этой стадии осуществляется первичная оценка проекта. Обычно именно здесь вы решаете, стоит ли вкладывать средства в фазу уточнения.
2. **Уточнение (**elaboration**).** На этой стадии идентифицируются основные прецеденты проекта и в итеративном процессе создается программное обеспечение, для того чтобы развернуть архитектуру системы. В конце фазы уточнения у вас должно быть достаточно полное понимание требований и скелет работающей системы, которую можно взять за основу разработки. В частности, необходимо обнаружить и разрешить основные риски проекта.
3. На стадии **построения** (construction) продолжается процесс создания и разрабатывается функциональность, достаточная для выпуска продукта.
4. **Внедрение** (transition) состоит из различных стадий работы, выполняемых в конце и в неитеративном режиме. Они могут включать развертывание в информационном центре, обучение пользователей и тому подобное.

Между фазами существует полная неопределенность, особенно между уточнением и построением. Для кого-то переход к построению – это момент, когда можно переключиться в режим прогнозирующего планирования. А для кого-то это просто точка, в которой появляется ясное понимание требований и архитектуры, определение которой, как вам кажется, движется к завершению.

Иногда RUP называют просто унифицированным процессом (Unified Process, UP). Так обычно поступают организации, которые хотят применить терминологию и общий подход RUP, но не хотят пользоваться лицензионными продуктами фирмы Rational Software. Можно думать о RUP как о продукте фирмы Rational, основанном на UP, а можно считать RUP и UP одним и тем же. В обоих случаях вы найдете людей, которые с вами согласятся.

Настройка процесса под проект

Программные проекты значительно отличаются друг от друга. Способ, которым ведется разработка программного обеспечения, зависит от многих факторов: типа создаваемой системы, используемой технологии, размера и распределенности команды, характера рисков, последствий неудач, стиля работы в команде и культуры организации. Поэтому не следует ожидать, что найдется процесс, подходящий для всех проектов.

Следовательно, необходимо приспособить процесс, чтобы он соответствовал вашему конкретному окружению. Один из первых шагов, которые необходимо сделать, – это взглянуть на проект и выбрать наиболее

подходящие процессы. Таким образом, вы получите короткий список процессов.

Затем необходимо определить, что следует предпринять, чтобы настроить процесс под конкретный проект. При этом надо соблюдать осторожность. Некоторые процессы трудно оценить, не поработав с ними. В таких случаях лучше провести с новым процессом пару итераций, чтобы понять, как он функционирует. Затем можно начинать модифицировать процесс. Если вы с самого начала знаете, как работает процесс, то можно модифицировать его без предварительной подготовки. Помните, что, как правило, легче начинать с малого и понемногу добавлять, чем сделать слишком много, а потом что-то выбрасывать.

Шаблоны

UML говорит, как изобразить объектно-ориентированный дизайн. Напротив, шаблоны представляют результат: примеры дизайна.

Многие утверждают, что в процессе выполнения проектов появляются трудности, поскольку исполнители не так сведущи в дизайне, который хорошо известен более подготовленным разработчикам. Шаблоны описывают общие подходы к работе, и они собираются людьми, которые обнаруживают повторяющиеся темы в процессе дизайна. Эти люди берут каждую такую тему и описывают ее так, чтобы другие разработчики смогли прочитать шаблон и узнать способ его применения.

Рассмотрим пример. Предположим, что у вас есть некоторые объекты, запускаемые в процессе на рабочем столе, и они должны взаимодействовать с другими объектами, запускаемыми во втором процессе. Возможно, второй процесс располагается также на вашем рабочем столе; возможно, он располагается в другом месте. Объекты вашей системы не должны искать другие процессы в сети или выполнять удаленные вызовы процедур.

Для удаленного объекта можно создать объект-посредник внутри локального процесса. Посредник имеет тот же самый интерфейс, что и удаленный объект. Локальный объект общается с посредником при помощи рассылки обычных сообщений внутри процесса. При этом посредник отвечает за передачу сообщений реальному объекту независимо от его местоположения.

Создание посредника – это обычная практика в сетевом взаимодействии и в других областях. Специалисты имеют большой опыт работы с посредниками, знают, как их применять, какие это дает преимущества и как их реализовать, какие им свойственны ограничения. Эти навыки обсуждаются в методических изданиях, подобных этому; все они рассказывают, как можно схематически представить посредника, и хотя это полезно, но не так, как было бы полезно рассмотрение опыта использования посредника.

В начале 90-х годов специалисты начали приобретать такой опыт. Они образовали сообщество людей, заинтересованных в написании шаблонов. Эти специалисты спонсировали ряд конференций и выпустили несколько книг.

Наиболее известной публикацией по шаблонам, выпущенной этой группой, является книга «банды четырех» [21], в которой подробно рассмотрены 23 шаблона дизайна. Посредникам в этой книге посвящен десяток страниц, дающих детальное представление о том, как объекты работают друг с другом; кроме того, обсуждаются преимущества и ограничения шаблонов, общие варианты использования и советы по реализации шаблонов.

Шаблон – это больше, чем модель. Шаблон должен также включать обоснование выбранного пути. Часто говорят, что шаблон – это ключ к решению проблемы. Шаблон должен четко идентифицировать проблему, объяснить, почему он решает проблему, а также объяснить, при каких условиях шаблон работает, а при каких нет.

Шаблоны важны, поскольку они являются следующим этапом в понимании основ языка или технологии моделирования. Шаблоны предоставляют набор решений, а также показывают, что помогает создать хорошую модель и какова последовательность действий при разработке модели. Шаблоны учат на примерах.

Когда я начинал, меня удивляло, почему я должен все изобретать с нуля. Почему у меня нет руководства, в котором бы рассказывалось о том, как делать общие вещи? Сообщество пытается создать такую книгу.

В настоящее время издано множество книг по шаблонам, и они очень отличаются по качеству. Мои любимые – это [21], [36], [37], [13], [35] и, извините за нескромность, [16] и [19]. Кроме того, вы можете зайти на домашнюю страничку шаблонов: *http://www.hillside.net/patterns*.

Как бы ни были вы вначале уверены, что знаете свой процесс, очень важно, чтобы по мере продвижения вперед вы учились. Действительно, одно из больших преимуществ итеративной разработки состоит в возможности часто совершенствовать процесс.

В конце каждой итерации следует проводить ее **ретроспективный анализ** (iteration retrospective), собирая команду на совещания, чтобы рассмотреть, как идут дела и что можно улучшить. Если итерация короткая, то достаточно двух часов. При этом хорошо составить список из трех категорий:

1. *Сохранить*: все, что работало правильно, и вы хотите это продолжить.
2. *Проблемы*: разделы, которые работали неправильно.
3. *Испытать*: изменения в процессе с целью его улучшения.

Вы можете начинать ретроспективный анализ каждой итерации, рассматривая элементы предыдущей сессии и определяя их изменения. Не забудьте про список того, что нужно сохранить; важно отслеживать элементы, которые работают правильно. Если вы этого не делаете, то можете потерять ощущение перспективы проекта и, возможно, перестать обращать внимание на успешные приемы.

В конце разработки проекта или его основной версии можно провести более формальный **ретроспективный анализ проекта** (project retrospective), который может занять пару дней; более подробную информацию можно найти на *http://www.retrospectives.com/* и в книге [26]. Больше всего меня раздражает, когда организации упорно игнорируют собственный опыт и постоянно совершают одни и те же ошибки.

Настройка UML под процесс

При рассмотрении графических языков моделирования обычно о них думают в контексте водопадного процесса. Водопадный процесс, как правило, сопровождается документами, выступающими в качестве пресс-релизов между стадиями анализа, дизайна и кодирования. Часто графические модели могут занимать основную часть этих документов. Действительно, многие из структурных методов 70-х и 80-х годов часто говорят о моделях анализа и дизайна, подобных этой.

Независимо от того, применяете вы метод водопада или нет, так или иначе вы проводите анализ, дизайн, кодирование и тестирование. Можно запустить итеративный проект с недельными итерациями, когда каждую неделю работает метод водопада.

Использование UML не подразумевает обязательную разработку документов или загрузку сложных CASE-систем.

Анализ требований

В процессе анализа требований необходимо понять, что клиенты и пользователи программного обеспечения ожидают от системы. В вашем распоряжении имеется множество приемов UML:

- Прецеденты, которые описывают, как люди взаимодействуют с системой.
- Диаграмма классов, которая строится с точки зрения концептуальной перспективы и может служить хорошим инструментом для построения точного словаря предметной области.
- Диаграмма деятельности, которая показывает рабочий поток организации, способы взаимодействия программного обеспечения и пользователей. Диаграмма деятельности может показать контекст для использования прецедентов, а также детали работы сложных прецедентов.

- Диаграмма состояний, которая может оказаться полезной, если концепция имеет своеобразный жизненный цикл с различными состояниями и событиями, которые изменяют эти состояния.

Анализируя состояния, помните, что самое важное – это взаимодействие с вашими пользователями и клиентами. Обычно это непрограммисты, и они не знакомы с UML и другими подобными технологиями. Несмотря на это я успешно применял перечисленные приемы при общении с людьми, не имеющими инженерной подготовки. Чтобы добиться этого, надо свести количество нотаций к минимуму. Не следует вводить элементы, специфичные для программной реализации.

Будьте готовы в любой момент отойти от правил UML, если это поможет улучшить взаимопонимание. Наибольший риск в случае применения UML для анализа состоит в том, что вы строите диаграммы, не совсем понятные специалистам в конкретной предметной области. Такая диаграмма хуже, чем бесполезная; она лишь способна вселить в разработчиков ложное чувство уверенности.

Проектирование

При разработке модели вы можете широко применять диаграммы. Можно использовать больше нотаций и при этом быть более точным. Вот некоторые полезные приемы:

- Диаграммы классов с точки зрения программного обеспечения. Они показывают классы программы и их взаимосвязи.
- Диаграммы последовательности для общих сценариев. Правильный подход состоит в извлечении наиболее важных и интересных сценариев из прецедента, а также в использовании CRC-карточек и диаграмм последовательности с целью понять, что происходит в программе.
- Диаграммы пакетов, показывающие высокоуровневую организацию программного продукта.
- Диаграммы состояний для классов со сложным жизненным циклом.
- Диаграммы развертывания, показывающие физическую конфигурацию программного обеспечения.

Многие из этих приемов позволяют документировать программное обеспечение после его создания. Кроме того, это может помочь людям сориентироваться в программе, если они ее не создавали и не знакомы с кодом.

В рамках жизненного цикла метода водопада необходимо создавать эти диаграммы и выполнять различного рода действия (активности) как часть конкретных фаз. Документы, создаваемые по окончании какой-либо фазы, обычно включают диаграммы для проведенных действий. Стиль водопада подразумевает применение UML в качестве инструмента проектирования.

При итеративном подходе диаграммы UML могут выступать или как модели, или как эскизы. В режиме проектирования аналитические диаграммы обычно создаются в рамках итерации, предшествующей итерации построения функциональности. Каждая итерация не начинается с самого начала. Напротив, она модифицирует существующую документацию, подчеркивая изменения, произошедшие в новой итерации.

Создание моделей обычно происходит на ранней стадии итерации и может быть сделано по частям для различных разделов функциональности, назначенной для данной итерации. Кроме того, итерация подразумевает изменение существующей модели, а не построение каждый раз новой модели.

Применение UML в режиме эскизирования – более подвижный процесс. Один из подходов заключается в выделении пары дней в начале итерации на создание эскизного дизайна для данной итерации. Можно также проводить короткие сессии проектирования в любой момент в ходе итерации, устраивая короткие получасовые совещания всякий раз, когда разработчики начинают спорить по поводу нетривиальной функции.

В режиме проектирования вы ожидаете, что программная реализация будет строиться в соответствии с диаграммами. Изменение модели следует считать отклонением, которое требует, чтобы проектировщики, создавшие эту модель, ее пересмотрели. Эскиз обычно трактуется как первый срез дизайна. Если в ходе кодирования обнаруживается, что эскиз не совсем точен, то разработчики должны иметь возможность изменить дизайн. Разработчики, внедряющие дизайн, должны сами решать, стоит ли устраивать широкую дискуссию, чтобы понять все возможные варианты.

Относительно моделей у меня есть свое собственное мнение, которое заключается в том, что даже хорошему проектировщику очень трудно составить правильную модель. Я часто обнаруживаю, что моя собственная модель не остается нетронутой по завершении кодирования. И все же я считаю эскизы UML полезными, хотя не думаю, что их следует возводить в абсолют.

В обоих режимах имеет смысл исследовать несколько вариантов дизайна. Как правило, лучше просматривать варианты в режиме эскизного моделирования, чтобы иметь возможность быстро создавать и изменять варианты. Выбрав дизайн, вы можете либо использовать эскиз, либо детально проработать его до уровня модели.

Документация

После создания программного обеспечения можно написать документацию на готовый продукт с помощью UML. Я считаю, что диаграммы UML очень помогают понять систему в целом. Однако я должен под-

черкнуть, что не верю в возможность создания подробных диаграмм всей системы. Процитирую Уорда Каннингема [14]:

> *Тщательно отобранные и хорошо написанные заметки могут легко заменить традиционную обширную документацию. Последняя редко проясняет суть вещей, за исключением отдельных аспектов. Выделите эти аспекты … и забудьте обо всем остальном.*

Я полагаю, что подробная документация должна генерироваться на основе программного кода – так, как это делает, например JavaDoc. Для того чтобы подчеркнуть важные концепции, необходимо написать дополнительную документацию. Можно считать ее составной частью первого этапа знакомства с программой, предшествующего детальному изучению кода. Я предпочитаю представлять документацию в виде текста, достаточно краткого, чтобы его можно было прочитать за чашкой кофе, и снабженного диаграммами UML, придающими обсуждению наглядный характер. Очевидно, что составитель документации должен решать, что важно, а что нет, поскольку разработчик вооружен для этого значительно лучше, чем читатель.

Диаграмма пакетов представляет хорошую логическую маршрутную карту системы. Эта диаграмма помогает понять логические блоки системы, а также обнаружить их взаимозависимости и держать их под контролем. Диаграмма развертывания (см. главу 8), которая показывает физическую картину системы на верхнем уровне, также может оказаться полезной на этой стадии.

Для каждого пакета я предпочитаю строить диаграмму классов. При этом я не указываю каждую операцию в том или ином классе, а показываю только важные свойства, которые помогают мне понять общую картину. Такая диаграмма классов служит своего рода оглавлением в виде графической таблицы.

Диаграмму классов следует сопроводить несколькими диаграммами взаимодействий системы, которые показывают наиболее важные из них. Повторюсь: правильный отбор очень важен; помните, что в документации такого типа полнота – враг понятности.

Если некоторый класс в течение своего жизненного цикла имеет сложное поведение, то для его описания я строю диаграмму конечного автомата (глава 10). Но делаю это только в случае достаточно сложного поведения, что, на мой взгляд, бывает очень редко.

В книге также часто встречаются фрагменты программного кода, важные для понимания системы и профессионально написанные. Кроме того, я использую диаграмму деятельности (глава 11), но только если она обеспечивает мне лучшее понимание, чем сам код.

Сталкиваясь с повторяющимися понятиями, я описываю их при помощи шаблонов (*стр. 54*).

В документации важно отразить неиспользованные варианты дизайна и пояснить, почему они были отброшены. Об этом часто забывают, но

такая информация может оказаться самой важной для ваших пользователей.

Понимание унаследованного кода

Язык UML помогает вникнуть за пару дней в труднодоступную для понимания ветвь незнакомой программы. Построение эскиза ключевых аспектов системы может действовать как графический запоминающий механизм, который помогает зафиксировать важную информацию о системе в процессе ее изучения. Эскизы ключевых классов и их наиболее важных взаимосвязей помогают пролить свет на происходящее.

Современные инструменты позволяют генерировать подробные диаграммы ключевых разделов системы. Не следует применять эти инструменты для создания больших бумажных отчетов; лучше с их помощью вскрывать наиболее важные пласты исследуемого программного кода. Особенно радует, что есть возможность генерации диаграмм последовательности, позволяющих проследить взаимодействие множества объектов при реализации сложного метода.

Выбор процесса разработки

Я твердый сторонник итеративного процесса разработки. Как я уже говорил в этой книге: «Применяйте итеративный метод разработки только в проектах, которым вы желаете успеха».

Может быть, кому-то покажется, что это болтовня, но с годами я становлюсь все более агрессивным сторонником итеративной разработки. При грамотном применении она является весьма важным методом, способным помочь в раннем выявлении возможных рисков и в улучшении управляемости процессом разработки. Однако это не означает, что можно вовсе обойтись без руководства проектом (хотя, если быть справедливым, я должен отметить, что некоторые используют ее именно для этой цели). Итеративная разработка требует тщательного планирования. Но это весьма надежный подход, и поэтому любая книга по объектно-ориентированной разработке рекомендует его применять – и не без основания.

Вы не должны удивляться, услышав, что я – как один из авторов Манифеста по гибкой разработке программного обеспечения (Manifesto for Agile Software Development) – большой любитель гибких подходов. У меня также накоплен большой положительный опыт в экстремальном программировании (Extreme Programming), и я рекомендую вам основательно познакомиться с этими технологиями.