Глава 1

Процесс разработки программного обеспечения

Данная глава посвящена изложению — на уровне обзора — некоторых стратегических вопросов, касающихся процесса разработки ПО. Поскольку предлагаемые темы рассматриваются лишь на общем уровне, а некоторые из вопросов носят спорный характер, читателям вовсе не обязательно соглашаться с автором, чтобы извлечь пользу из оставшейся части книги (а, может быть, и переменить свое мнение по завершении чтения книги).

Образовательное значение данной главы состоит в том, чтобы ввести читателя в процессы и подходы, которые лежат в основе современной разработки ПО. Многие идеи и вопросы, рассматриваемые в данной главе, могут быть уже знакомы читателям из опыта, повседневного использования компьютеров или соответствующей литературы. При желании такие читатели могут просто бегло просмотреть данную главу и перейти к главе 2.

1.1. Характер процесса разработки ПО

Литература по управлению информационными системами (ИС) изобилует примерами провалившихся проектов, превышения сроков и бюджетов, ошибочных решений, систем, не пригодных для сопровождения и т.д. В исследовательском отчете организации Standish Group за 1998 год по проекту CHAOS (который так любят цитировать!) утверждается, что три из четырех программных проектов терпят неудачу в одной из приведенных выше областей. В свете подобных утверждений уместно задаться вопросами: “В чем причина подобных провалов? Какие симптомы свидетельствуют о возникновении проблем в проекте и как их можно нейтрализовать?”

Прежде, чем обратиться к этим вопросам, необходимо понять характер процесса разработки ПО. В своей ставшей уже классической статье [10] Брукс (Brooks) определяет как сущность, так и случайные свойства, характерные для программной инженерии (software engineering). *Сущность* программной инженерии кроется в собственных свойствах программного обеспечения, которые вызывают трудности при его создании. Эти трудности можно только осознать — их нельзя ни преодолеть за счет какого-либо технологического прорыва, ни сразить “серебряной пулей”. Согласно Бруксу

сущность программной инженерии проистекает из таких свойств ПО, как сложность, податливость, изменчивость и неосязаемость.

Четыре "существенных трудности" создания ПО определяют инвариант или неизменную составляющую процесса его разработки. Инвариант просто констатирует тот факт, что ПО является продуктом творческого акта разработки — ремесла или искусства в том смысле, что эта деятельность совершается скорее ремесленником, чем художником. При обычном положении дел ПО не является результатом повторяющегося акта производства.

После того, как мы разобрались, что собой представляет инвариант разработки ПО, можно обратиться к понятию *случайных свойств* программной инженерии. Эти свойства не присущи ПО изначально, они скорее привнесены извне и обусловлены вмешательством "человеческого фактора" в практику создания ПО. Свойства программной инженерии, которые носят случайный характер, также порождают определенные трудности при создании ПО. В свою очередь мы разделяем эти "случайные трудности" на три категории.

1. Участники проекта.
2. Процесс.
3. Язык и средства моделирования.

1.1.1. Инвариант разработки ПО

Как мы уже отмечали, ПО скорее *разрабатывается*, а не *производится* [65]. Конечно, никто не может отрицать, что достижения программной инженерии привнесли в практику разработки бо́льшую определенность, однако (в отличие от традиционных инженерных дисциплин), успех программного проекта гарантировать нельзя.

Алгоритмы, библиотеки программ, повторно используемые классы, программные компоненты и т.д. представляют собой неполные, фрагментарные решения для того, что нам требуется описать при разработке информационных систем. Проблема состоит в том, чтобы собрать воедино небольшие фрагменты проблемы в виде гармоничной *корпоративной системы*, которая удовлетворяет требования сложных бизнес-процессов.

Практика создания ПО способствует разработке систем из настраиваемых программных пакетов — решений на основе COTS-продуктов (*commercial-of-the-shelf* software — готовое к использованию ПО) или ERP-систем (*Enterprise Resource Planning* System— система планирования корпоративных ресурсов). Программный пакет может обеспечить функции стандартного бухгалтерского учета, производственной системы или системы управления кадрами. Акцент смещается от разработки "с чистого листа" к "настройке" ПО, однако, процесс производства все равно занимает значительное место.

Для всякой системы, разрабатываемой "с чистого листа", необходимо сначала создать *концептуальные конструкции* (модели) для конечного решения, которые бы удовлетворяли специфические потребности организации. После их создания функциональные возможности программного пакета настраиваются в соответствии с концептуальными конструкциями. Задачи программирования могут быть разными, но характер деятельности по анализу требований и системному проектированию, связанной с этими конструкциями, при разработке "с нуля" аналогичен. В общем, концептуальная конструкция (модель) сохраняется, несмотря на всевозможные изменения ее представления (реализации).

Что еще более важно, маловероятно, чтобы организация могла найти программный пакет для автоматизации ее ключевых бизнес-процессов. Ключевым бизнес-процессом для телефонной компании выступает телефония, а не управление кадрами или бухгалтерский учет. То, что заставляет компанию работать (и конкурировать), должно разрабатываться "с нуля" (или переделываться из *унаследованной системы*). Как верно было подмечено Шиперским (Szyperski) [88]: "Стандартные [программные] пакеты создают лишь гладкое игровое поле, а умение играть приходит совсем с другой стороны".

Конечно, в любом случае процесс разработки должен использовать преимущества *компонентной технологии* [1], [88]. Компонента — это исполняемый программный модуль, реализующий четко определенные функции (сервисы) и коммуникационные протоколы (интерфейсы) взаимодействия с другими компонентами. Компоненты можно сконфигурировать таким образом, чтобы удовлетворить требования к приложению. В настоящее время наибольшую популярность завоевали следующие компонентные технологии.

- CORBA (*Common Object Request Broker Architecture — Общая архитектура брокера объектных запросов*) от OMG (Object Management Group).
- DCOM (*Distributed Component Object Model — Распределенная модель компонентных объектов*) от Microsoft.
- EJB (*Enterprise JavaBeans – Корпоративная технология для Java-приложений*) от Sun.

Пакеты, компоненты и другие подобные методы не изменяют сущности создания ПО. В частности, принципы и задачи анализа требований и системного проектирования остаются неизменными. Конечный программный продукт может собираться из стандартных или разрабатываемых под заказ компонент, но сам процесс "сборки" попрежнему остается искусством. Откровенно говоря, как заметил Прессман (Pressman) [65], у нас нет даже "запасных частей", чтобы заменить отказавшие по ходу дела компоненты системы.

1.1.2. Участники проекта

Участниками проекта (*stakeholders*) мы называем людей, которые заинтересованы в программном проекте. Любой человек, интересы которого так или иначе затрагиваются системой либо сам оказывающий влияние на разработку системы, является участником проекта. (Проект здесь понимается в широком смысле, как некая деловая активность, предприятие, в нашем случае — это программный проект. *Прим. ред.*) Участники проекта делятся на две основные группы.

- Заказчики (пользователи и владельцы системы).
- Разработчики (аналитики, проектировщики, программисты и т.д.).

Мы предпочитаем использовать термин *заказчик* (*customer*), а не *пользователь* (*user*). Это различение особенно безусловно обоснованно с точки зрения разработки системы. Во-первых, заказчик — это лицо, которое платит за разработку и отвечает за принятие решений. Во-вторых, даже если заказчик не во всем прав, требования заказчика не могут быть произвольным образом изменены или отвергнуты разработчиками — любое противоречивое, невыполнимое или неправомерное требование должно быть оговорено с заказчиками.

В свете последнего утверждения мы отдаем себе отчет в том, что термин *пользователь* (в значении *заказчик*) значительно более распространен и мы не будем в дальнейшем избегать его. Однако, мы будем воздерживаться от использования неудобного термина конечный пользователь (*end-user*), под которым исторически подразумевался *уполномоченный пользователь*, выбранный для общения с разработчиками (вместо *реального пользователя*).

Информационные системы — системы *социальные*. Они разрабатываются людьми (разработчиками) для людей (заказчиков). Успех программного проекта определяется социальными факторами — роль технологии второстепенна. Есть немало примеров технически отсталых систем, которые работают и полезны заказчикам. Обратное, как правило, неверно. От системы, не приносящей пользы (ожидаемой или реальной), заказчик рано или поздно откажется независимо от того, насколько блестящей она является в техническом отношении.

В типичной ситуации основные причины провала программных проектов могут быть отнесены на счет участников проекта. Если рассматривать проблему провала проектов со стороны *заказчиков*, то эти причины выглядят следующим образом [62]:

- потребности заказчиков не поняты или не полностью зафиксированы;
- требования заказчиков изменяются слишком часто;
- заказчики не готовы выделить достаточно ресурсов под проект;
- заказчики не стремятся к сотрудничеству с разработчиками;
- ожидания заказчиков нереалистичны;
- система оказывается бесполезной для заказчиков.

Проекты также заканчиваются неудачей потому, что *разработчики* оказываются не на высоте поставленных задач. В связи с увеличением сложности ПО растет понимание того, что критическим фактором разработки становятся опыт и знания разработчиков. Хорошие разработчики могут дать решение. Высококлассные разработчики могут дать значительно лучшее решение, намного быстрее и дешевле. На эту тему существует довольно известная шутка Брукса: “Великие проекты — удел великих разработчиков” [10].

Мастерство и ответственность разработчиков являются факторами, вклад которых в достижение качества и продуктивности ПО трудно переоценить. Чтобы гарантировать успешную поставку ПО заказчику и, что более важно, обеспечить выгоды от его использования, организация-поставщик ПО должна в отношении разработчиков следовать некоторым простым правилам [10], [96]. Итак, организация-разработчик ПО должна добиваться следующего.

- Нанимать лучших разработчиков.
- Обеспечивать непрерывное обучение и повышать уровень образования своих разработчиков.
- Поощрять обмен информацией и общение между разработчиками так, чтобы они взаимно стимулировали друг друга.
- Стимулировать разработчиков, устраняя препятствия и направляя их усилия на продуктивную работу.
- Создавать исключительно благоприятную рабочую атмосферу (зачастую это оказывается намного более важным, чем редкие прибавки к жалованью).

- Увязывать личные цели разработчиков со стратегией и задачами организации.
- Придавать особое значение коллективной работе.

1.1.3. Процесс

Процесс разработки ПО определяет действия и организационные процедуры, направленные на усиление совместной работы в бригаде разработчиков с целью поставки заказчикам высококачественных программных продуктов. На модель процесса возлагаются следующие функции.

- Установление порядка выполнения действий.
- Определение состава и времени поставки артефактов, создаваемых в процессе разработки.
- Закрепление действий и артефактов за разработчиками.
- Введение критериев отслеживания хода проекта, измерение результатов и планирование будущих проектов.

Процесс разработки невозможно стандартизировать или систематизировать таким образом, чтобы любая организация могла использовать его автоматически. Каждая организация должна разработать свою собственную модель процесса или приспособить некоторый настраиваемый шаблон процесса под свои нужды. К последним относится шаблон, предлагаемый компанией *Rational Software Corporation*, известный как *Rational Unified Process* [47].

Процесс, который берет на вооружение организация, должен быть увязан с ее развитием, культурой, социальной динамикой, знаниями и опытом разработчиков, практикой руководства, ожиданиями заказчиков, масштабами проектов и даже характером проблемных областей. Поскольку все перечисленные факторы подвержены изменениям, организации может потребоваться ввести разнообразие в ее модель процесса и предусмотреть варианты модели для каждого отдельного программного проекта. Например, в зависимости от того, насколько разработчики хорошо знакомы с методами и средствами моделирования, в процесс может потребоваться включить специальные курсы обучения. Наверное самое большое влияние на процесс оказывает *масштаб проекта*. Для небольших проектов (порядка десяти, или около того, разработчиков) формальный процесс может вообще не потребоваться. Такая небольшая бригада скорее всего способна общаться и реагировать на изменения неформально. Для более крупных проектов неформальной коммуникативной сети может оказаться недостаточно, в этом случае требуется строго определенный процесс для управляемой разработки.

1.1.3.1. Итеративный процесс разработки с пошаговым наращиванием возможностей

Современные процессы разработки ПО непременно являются *итеративными* (*iterative*) процессами *с пошаговым наращиванием возможностей* (*incremental*) системы.

Модели системы уточняются и преобразуются на этапах анализа, проектирования и реализации — в результате успешных *итераций* добавляются новые детали, при необходимости вводятся изменения и усовершенствования, а *выпуски* программных модулей с *наращенными возможностями* поддерживают высокий уровень удовлетворенности пользователей и обеспечивают обратную связь, необходимую для продолжения разработки модулей.

Одно из положений методологии *Rational Unified Process* звучит следующим образом: "Итеративный процесс — это процесс, направленный на управление потоком исполняемых версий ПО. Процесс с наращиванием возможностей — это процесс, направленный на непрерывную интеграцию системной архитектуры для производства этих версий, при этом каждая новая версия заключает в себе усовершенствованные возможности в сравнении с предыдущими" [8].

Успех итеративного процесса с наращиванием возможностей основывается на раннем выявлении *архитектурных модулей системы*. Модули должны быть примерно одинаковыми по размерам, обладать сильной внутренней связностью и иметь минимум взаимных пересечений (внешних связей). Важен также порядок реализации модулей. Некоторые модули невозможно реализовать в виде исполняемой версии, если они зависят от информации или результатов вычислений, производимых другими модулями, которые еще должны быть разработаны. Если только в основе итеративной наращиваемой разработки не лежит планирование и контроль, то в отсутствие контроля реального хода проекта процесс может превратиться в "хакерство".

1.1.3.2. Модель технологической зрелости

Главной проблемой для любой организации, занятой производством программного обеспечения, является усовершенствование процесса разработки. Вполне естественно, что, для того, чтобы ввести в процесс усовершенствования, организация должна знать, какие проблемы связаны с существующим процессом. Одним из наиболее известных методов оценки и усовершенствования процессов является так называемая модель технологической зрелости (*Capability Maturity Model — CMM*) [12].

Модель CMM была разработана Институтом программной инженерии (Software Engineering Institute (SEI)) при Питтсбургском университете Карнеги-Меллона (Carnegie Mellon University) в США. Первоначально эта модель использовалась Министерством обороны США для оценки возможностей организаций в области ИТ, которые участвовали в конкурсах на получение оборонных заказов, сегодня она находит широкое применение в ИТ-индустрии как в Америке, так и за ее пределами.

По существу CMM — это *анкета*, которую заполняет организация, работающая в области ИТ. После анкетирования следует верификация и аттестация процесса, в результате которых организацию относят к одному из пяти уровней модели CMM. Чем выше уровень, тем более зрелым является процесс разработки ПО в организации.

На рис. 1.1 определены уровни, даны краткие описания основных особенностей каждого из уровней и приведены основные сферы улучшения процесса, необходимые для достижения организацией более высокого уровня технологической зрелости.

Артур (Arthur) [3] называет уровни зрелости "лестницей, ведущей к совершенству в области ПО". Пять ступенек на этой лестнице соответствуют хаосу, управляемому проекту, применению методов и средств организации процесса, измерению процесса и непрерывному повышению качества.

Как показывает опыт, на переход с одного уровня шкалы технологической зрелости на другой уходит несколько лет. Большинство организаций находятся на уровне 1, некоторые — на уровне 2; известно очень немного организаций, находящихся на 5-м уровне. Приводимые ниже вопросы показывают, насколько трудновыполнима эта задача. Организация, которая стремится достичь 2-го уровня, должна добиться положительных ответов на все приведенные ниже вопросы (а также на некоторые другие) [62].

Рис. 1.1. Уровни зрелости процесса по модели CMM

- Имеется ли в организации канал управления и отчетности по функции обеспечения качества ПО, отдельный от управления проектами по разработке ПО?
- Введена ли для каждого проекта функция конфигурационного управления ПО, которая затрагивает его разработку?
- Использует ли руководство формальный процесс для анализа каждой программной разработки прежде, чем взять на себя договорные обязательства?
- Используется ли для составления календарного плана разработки ПО формальная процедура?
- Применяются ли формальные процедуры для оценки затрат на разработку ПО?
- Собирается ли статистика в отношении программного кода и ошибок, выявленных при тестировании?
- Имеется ли в распоряжении высшего руководства механизм для регулярного анализа состояния проектов по разработке ПО?
- Используется ли определенный механизм для контроля изменений требований к ПО?

1.1.3.3. Стандарт ISO 9000

Помимо CMM существуют и другие модели усовершенствования процесса создания ПО. Особый интерес представляет серия стандартов по качеству ISO 9000, разработанная Международной организацией по стандартизации (International Organization for Standardization). Стандарты ISO этой серии применяются для управления *качеством* и определения *процесса* производства качественной продукции. Стандарты

носят общий характер — они применимы для любой отрасли и всех видов бизнеса, включая разработку ПО.

В основе серии стандартов ISO 9000 лежит предположение о том, что если процесс организован надлежащим образом, то и результат процесса (товар или услуга) также будет обладать надлежащим качеством. "Цель управления качеством заключается в производстве качественных товаров за счет того, чтобы сделать качество неотъемлемым свойством товара, а не проверки того, насколько оно присуще товару" [78].

Возвращаясь к нашему предыдущему рассмотрению особенностей процесса разработки ПО, заметим, что стандарты ISO не навязывают конкретного процесса и не уточняют его характеристик. Стандарты дают модель того, *что* должно быть достигнуто, однако не говорят о том, *как* именно должна осуществляться та или иная деятельность. Организация, которая стремится получить сертификат ISO (это также называется регистрацией) должна рассказать, что она делает; доказать, что слова у нее не расходятся с делом; и показать, чего она уже достигла [78].

Своеобразной "лакмусовой бумагой", позволяющей проверить, насколько организация заслуживает сертификата ISO, может служить ее способность создать качественный товар или обеспечить качественное обслуживание даже в том случае, если заменить весь персонал. С этой целью необходимо *оформить документально и зафиксировать* все виды своей деятельности. Для каждого вида деятельности должна быть определена письменная процедура, включая действия, которые необходимо выполнить в случае нарушения процесса или жалоб клиента.

Так же, как в случае модели CMM, сертификат ISO может быть официально предоставлен только после *аудита, проведенного на месте* регистрационным бюро ISO. Подобные аудиты затем повторяются через определенное время на регулярной основе. Организации вынуждены проходить через подобную систему из-за давления конкурентных факторов, обусловленных требованиями со стороны потребителей, чтобы поставщики товаров и услуг были сертифицированы.

1.1.4. Язык и средства моделирования

Участники проекта и процесс составляют два элемента "треугольника", обеспечивающего успех программного проекта. Третий элемент состоит из языка и средств моделирования. Подлежащие моделированию артефакты необходимо выразить (язык) и документально зафиксировать (средства).

Разработчикам необходим *язык* для построения как визуальных моделей, так и моделей других типов, и обсуждения их с заказчиками и коллегами. Язык должен позволять строить модели на различных уровнях абстракции для представления предлагаемых решений на различных уровнях детализации.

Язык должен обладать мощной *визуальной* составляющей согласно известной поговорке о том, что "лучше один раз увидеть, чем сто раз услышать". Язык моделирования должен также обладать мощной *декларативной семантикой*, т.е. должен позволять зафиксировать "процедурное" значение в форме "декларативного" предложения. Иначе говоря, мы должны иметь возможность сказать, "что" именно необходимо сделать, а не распространяться о том, "как " это должно делаться.

Кроме того, разработчикам необходимы средства для автоматизированного проектирования и создания программ или так называемые CASE-средства (Computer-Assisted Software Engineering — CASE). CASE-средства позволяют хранить и получать доступ к моделям через центральный репозиторий, а также манипулировать этими

моделями на экране компьютера в графическом и текстовом режимах. В идеале репозиторий должен обеспечивать одновременный доступ многих пользователей (многих разработчиков) к моделям. Ниже приводится перечень типичных функций CASE-репозитория.

- Координация доступа к моделям.
- Помощь в организации взаимодействия между разработчиками.
- Хранение нескольких версий моделей.
- Идентификация различий между версиями.
- Возможность совместного использования одних и тех же концептов в различных моделях.
- Проверка непротиворечивости и целостности моделей.
- Генерация проектных отчетов и документов.
- Генерация структур данных и программного кода (конструирование ПО).
- Генерация моделей по существующей реализации (реконструкция ПО) и т.д.

Следует заметить, что зачастую программа, сгенерированная с помощью CASE-средств, представляет собой на самом деле всего лишь скелет программы — вычислительный алгоритм, который необходимо дорабатывать программисту как при обычном программировании.

1.1.4.1. UML

"UML (Unified Modeling Language — унифицированный язык моделирования) — это визуальный язык моделирования общего назначения, который используется для спецификации, визуализации, конструирования и документирования артефактов программной системы" [76]. Язык UML был разработан компанией Rational Software Corporation для унификации лучших свойств, которыми обладали более ранние методы и способы нотации. В 1997 году организация OMG (Object Management Group — группа управления объектами) признала его в качестве стандартного языка моделирования. С тех пор UML получил дальнейшее развитие и широкое признание в отрасли ИТ.

Язык UML не зависит от применяемого процесса разработки ПО, хотя позже компания Rational Rose предложила процесс, соответствующий этому языку, под названием *Rational Unified Process* (Унифицированный процесс [компании] Rational. Заметим, впрочем, что название методологии можно перевести как "рациональный унифицированный процесс". *Прим. ред.*) [47]. Совершенно очевидно, что процесс, в котором в качестве базового языка принят UML, должен поддерживать *объектно-ориентированный подход* к созданию ПО. Язык UML не подходит для несовременных структурных подходов, результатом которых являются системы, реализованные с помощью процедурных языков программирования, наподобие языка COBOL.

Язык UML также не зависит от технологий реализации (поскольку они являются объектно-ориентированными). По нашему мнению это делает UML ограниченным в отношении поддержки этапа детализированного проектирования жизненного цикла (ЖЦ) ПО. В то же время это делает UML более устойчивым к частой смене платформ реализации.

Конструкции языка UML позволяют моделировать статику (структуру) и динамику (поведение) системы. Система представляется в виде взаимодействующих *объектов* (программных модулей), которые реагируют на внешние события. Действия объектов позволяют выполнить определенные задачи или получить клиентам (пользователям)

системы некоторый полезный результат. Отдельные модели отображают определенные стороны системы и пренебрегают другими сторонами, которые охватывают другие модели. Взятые в комплексе модели обеспечивают полное описание системы.

Модели, создаваемые с помощью языка UML, можно разделить на три группы.

- *Статические модели* (описывают статические структуры данных).
- *Модели поведения* (описывают взаимодействие объектов).
- *Модели изменения состояний* (описывают допустимые состояния системы, которые она принимает с течением времени).

UML также содержит несколько *архитектурных конструкций*, которые позволяют придать системе модульную структуру, используемую в процессе итеративной и наращиваемой разработки.

1.1.4.2. CASE-средства и совершенствование процесса

Совершенствование процесса — это нечто большее, чем просто введение новых методов и средств. В действительности, введение новых методов и средств в организации, находящейся на низком уровне зрелости процесса разработки, может принести больше вреда, чем пользы.

Подходящий пример — CASE-технологии. *Интегрированные* CASE-средства позволяют нескольким разработчикам взаимодействовать и совместно использовать проектную информацию для выработки новых проектных артефактов. Чтобы воспользоваться преимуществами этой технологии, бригада разработчиков должна подчиняться определенным правилам, поскольку CASE-средства налагают на процессы некоторые ограничения. Но если бригада разработчиков не настолько квалифицированна, чтобы усовершенствовать процесс разработки, чрезвычайно маловероятно, чтобы она смогла воспринять процесс, диктуемый CASE-средствами. В результате потенциальные возможности роста продуктивности и качества, которые обещает новая технология, так и не будут реализованы.

Рассмотренные выше особенности применения CASE-средств не должны натолкнуть вас на мысль, что CASE-технология — "рискованное дело". Она может и не дать вам ожидаемых выгод, если вы пытаетесь использовать ее для того, чтобы направлять работу всей бригады разработчиков, а бригада не готова следовать нужному процессу. Однако, те же методы и CASE-средства безусловно могут обеспечить повышение личной продуктивности и качества работы отдельных разработчиков, которые используют технологию на своих локальных рабочих станциях. Моделировать программные артефакты с помощью карандаша и бумаги уместно только в аудитории, но никак ни при работе над реальным проектом.

1.2. Планирование разработки системы

Проекты по разработке информационных систем (ИС) должны быть заранее спланированы. ИС необходимо идентифицировать, классифицировать, ранжировать и выбирать для первоначальной разработки, для усовершенствования или, возможно, для ликвидации. Вопрос состоит в следующем: "Какие технологии ИС и приложения принесут наибольшие деловые выгоды?" В идеале решение, которому необходимо следовать, должно базироваться на *бизнес-стратегии* и тщательном и методичном планировании [5],[37], [51].

Бизнес-стратегию можно определить посредством различных процессов, известных как *стратегическое планирование*, *бизнес-моделирование*, *реинжиниринг бизнес-процессов*, *стратегическое увязывание*, *управление информационными ресурсами* и т.п. Мы не ставим целью пояснять различия между названными подходами. Достаточно сказать, что все эти подходы связаны с изучением фундаментальных бизнес-процессов в организации, целью которых является определение долгосрочного в*и*дения бизнеса с последующим назначением приоритетов различным проблемам ведения бизнеса, которые могут быть разрешены с помощью информационной технологии (ИТ).

При этом существует немало организаций — в особенности это относится к небольшим организациям, которые не имеют отчетливой стратегии ведения бизнеса. Подобные организации скорее всего принимают решение о развитии ИС, просто выявляя наиболее насущные деловые проблемы, которые требуют немедленного решения. При изменении внешней бизнес-среды или внутренних условий ведения бизнеса существующая ИС снова подлежит модификации. Хотя этот подход обладает очевидными недостатками, он позволяет небольшим организациям быстро перестроиться в соответствии с текущей ситуацией либо извлечь выгоду из представившихся возможностей, либо дать отпор новым угрозам.

Крупные организации не могут позволить себе постоянные изменения в направлении ведения бизнеса. В действительности они зачастую диктуют направление деятельности другим организациям, работающим в той же сфере бизнеса, а также в определенной мере могут формировать среду для своих *повседневных* нужд. Однако, крупные организации должны внимательно вглядываться в будущее: использовать плановый подход при выборе проектов, связанных с разработкой. Как правило, это масштабные проекты, на выполнение которых требуется много времени. Они слишком громоздки, чтобы их можно было легко изменить или заменить, и должны быть способны адаптироваться или даже быть устремлены навстречу будущим возможностям и угрозам.

Существует много способов планирования разработки систем. Один из традиционных подходов получил название *SWOT* (Strengths, Weaknesses, Opportunities, Threats — сильные стороны, слабые стороны, благоприятные возможности, угрозы). Еще одна популярная стратегия базируется на модели *VCM* (Value Chain Model — модель цепочек ценности). Более современные варианты подходов к разработке бизнес-стратегий известны как *BPR* (Business Process Reengineering — реинжиниринг бизнес-процессов). Информацию, необходимую для деятельности организации, также оценивают с использованием так называемых проектных шаблонов для *ISA* (Information System Architecture — архитектура информационных систем). Подобные проектные шаблоны можно получить по аналогии из описательных схем, успешно зарекомендовавших себя в дисциплинах, отличных от ИТ (например, в строительной промышленности).

Все подходы к планированию разработки систем обладают одним общим знаменателем — они направлены скорее на достижение *эффективности* (делать то, что нужно), чем *продуктивности* (делать так, как нужно). Наиболее рациональное решение неверной проблемы немногого стоит!

1.2.1. Подход SWOT

Подход SWOT позволяет идентифицировать, классифицировать, ранжировать и выбирать проекты по разработке ИС таким образом, чтобы они были увязаны с силь-

ными и слабыми сторонами организации, а также возможностями и угрозами. Это подход сверху-вниз, применение которого начинается с определения миссии организации.

Формулировка миссии фиксирует уникальный характер организации и определяет ее видение того, где она хотела бы оказаться в будущем. Верно сформулированная миссия отводит главное место потребностям клиентов, а не товарам или услугам, которые предоставляет организация.

Формулировка миссии и разрабатываемая на ее основе бизнес-стратегия учитывают то, какими *внутренними сильными и слабыми сторонами* обладает компания в таких областях, как управление, производство, кадровое обеспечение, финансы, маркетинг, исследования и разработка и т.д. Эти сильные и слабые стороны должны быть детально обсуждены и согласованы, после чего им назначаются приоритеты. Преуспевающие организации способны в любой момент идентифицировать текущий набор сильных и слабых сторон, которые направляют разработку их бизнес-стратегий.

Выявление внутренних сильных и слабых сторон компании — необходимое, но недостаточное условие для успешного делового планирования. Организация функционирует не в вакууме — ее деятельность зависит от внешних экономических, социальных, политических и технологических факторов. Она должна знать о *внешних благоприятных возможностях*, из которых необходимо извлечь выгоду, а также о *внешних угрозах*, которых следует избегать. Этими факторами организация не в состоянии управлять, однако осведомленность в отношении их играет существенную роль при определении целей и задач организации.

На любом заданном отрезке времени организация преследует одну или очень небольшое количество *целей*. Цели обычно имеют долговременный характер (три-пять лет), а иногда относятся к "вечным проблемам". К типичным примерам целей относятся повышение уровня удовлетворенности потребителей, введение новых видов услуг, преодоление конкурентных угроз, усиление контроля над поставщиками и т.д. Каждая стратегическая цель должна быть связана с определенными задачами, которые обычно принимают форму годовых заданий. Например, цель "повысить уровень удовлетворенности потребителей" может поддерживаться задачей более быстрого выполнения заказов клиентов — скажем, в течение двух недель.

Цели и задачи требуют выработки *стратегий* управления и определенной *политики* в отношении реализации этой стратегии. Подобные методы управления позволяют привести в соответствие организационные структуры, распределить ресурсы и определить проекты, связанные с разработкой, в том числе, информационных систем.

1.2.2. Подход VCM

Метод ценностных цепочек — VCM — позволяет оценить конкурентные преимущества с помощью анализа всей цепочки видов деятельности в организации, начиная от получения сырья до конечной продукции, продаваемой и доставляемой потребителям. Метафора цепочки усиливает то положение, что единственное слабое звено приводит к разрыву всей цепи.

Модель служит цели уяснения того, какая конфигурация цепочки ценности сулит наибольшие конкурентные преимущества. Проекты по разработке ИС могут впоследствии указать на те сегменты, операции, каналы распределения, маркетинговые подходы и т.д., которые позволяют завоевать наибольшее конкурентное преимущество.

В первоначальном варианте метода VCM [63] организационные функции были разделены на *основные виды деятельности* и *вспомогательные виды деятельности*. Основ-

ные виды деятельности создают или добавляют ценность конечному продукту. Они разделяются на пять последовательных этапов: (1) входящее снабжение, (2) обработка, (3) исходящее снабжение, (4) сбыт и маркетинг, (5) обслуживание.

Поддерживающие виды деятельности не добавляют ценности, по крайней мере, прямо. Они также играют существенную роль, но не "обогащают" продукт. Поддерживающие виды деятельности включают: (1) администрацию и инфраструктуру, (2) управление кадрами, (3) исследования и разработку и — наверное, это не удивительно — разработку ИС.

Хотя модель VCM представляет собой полезный инструмент для планирования разработки ИС, вездесущая компьютеризация может способствовать переменам в способах ведения бизнеса, что, в свою очередь, создает конкурентные преимущества. Другими словами, *ИТ может преобразовать ценностные цепочки организации.* Таким образом, между ИТ и моделью VCM может быть установлена положительная обратная связь.

Портер (Porter) и Миллар (Millar) [64] выделяют пять шагов, которые должна предпринять организация, чтобы воспользоваться преимуществами, предоставляемыми ИТ.

1. Оценить информационную емкость продуктов и процессов.
2. Оценить роль ИТ в отраслевой структуре.
3. Выявить и ранжировать способы, с помощью которых ИТ создает конкурентное преимущество.
4. Рассмотреть, каким образом ИТ может создать новое направление в бизнесе.
5. Разработать план, направленный на извлечение выгод от использования ИТ.

1.2.3. Подход BPR

Подход к планированию разработки ИС с использованием методов реинжиниринга бизнес-процессов (BPR-методов) основан на допущении, что современные организации должны реконструировать себя и отказаться от функциональной декомпозиции, иерархических структур и принципов приоритетности повседневных нужд, которые они сегодня используют.

Рассматриваемая концепция была введена в 1990 году Хаммером (Hammer) и Девенпортом (Davenport) [30], и Шортом (Short) [19] и сразу обратила на себя внимание, вызвав полемику. Расширенное описание метода BPR можно найти в книгах его основателей [32], [18].

Сегодня большинство организаций имеют структуру с *вертикальным подчинением подразделений*, сосредоточенных на функциях, товарах или регионах. Эти структуры и методы работы прослеживаются вплоть до восемнадцатого века и берут свое начало в принципе разделения труда и последующей фрагментации работы, сформулированном еще Адамом Смитом. Никто из работников или подразделений не отвечает за *бизнес-процесс*, который определяется как "...совокупность видов деятельности, получающих на входе один или несколько типов ресурсов и создающих на выходе продукцию, представляющую ценность для потребителя" [32].

Методология реинжиниринга бизнес-процессов ставит под сомнение индустриальные принципы Смита разделения труда, иерархического управления и экономии на масштабах. В современном мире организация должна быть способна адаптироваться к быстрым изменениям рынка, новым технологиям, конкурентным факторам, требованиям потребителей и т.д.

Жесткие организационные структуры, в которых бизнес-процессы разорваны между многими подразделениями, устарели. Организации должны сосредоточиваться на бизнес-процессах, а не на отдельных задачах, заданиях, специалистах и функциях подразделений. Эти процессы разделены *по горизонтали* между видами деятельности и завершаются в точках контакта с потребителями. "Наиболее видимое различие между предприятием, ориентированным на бизнес-процессы, и традиционной организацией состоит в существовании у каждого из процессов "хозяина" [33].

Основная цель реинжиниринга бизнес-процессов состоит в радикальной реконструкции бизнес-процессов в организации (поэтому подход BPR зачастую называется *реконструкцией* или *перепроектированием процессов*). Бизнес-процессы необходимо идентифицировать, хорошо наладить и усовершенствовать. Поведение процессов фиксируется в виде *диаграмм потоков работ* (*workflow diagrams*) и изучается в рамках дисциплины "*Анализ потоков работ*". Потоки работ охватывают потоки событий, документов и информации в рамках бизнес-процессов и могут использоваться для подсчета времени, ресурсов и финансовых средств, необходимых для этих видов деятельности.

Основное препятствие на пути реализации BPR-подхода в организации лежит в необходимости внедрения горизонтального процесса в традиционную вертикальную структуру управления. Серьезная инициатива по внедрению BPR-подхода требует переключения организации на проектные бригады как основные организационные единицы. Эти бригады отвечают за один или более сквозных бизнес-процессов.

Иногда радикальные изменения неприемлемы. Традиционные структуры не могут быть изменены в одночасье. Радикальные шаги могут встретить сопротивление, и потенциальные выгоды от внедрения BPR-подхода могут быть подвергнуты риску. В данных обстоятельствах организация все же может выиграть от моделирования бизнес-процессов и попыток просто усовершенствовать их, а не подвергать полной переделке. Термин "усовершенствование бизнес-процессов" (Business Process Improvement — BPI) используется для характеристики подобного начинания [1].

После того, как бизнес-процесс определен, "хозяева" процесса могут потребовать поддержки со стороны ИТ с целью дальнейшего повышения продуктивности этих процессов. Результирующий проект по разработке ИС должен сосредоточиваться на реализации выявленных потоков работ. Сочетание *эффективности*, получаемой от применения BPR-подхода, с *продуктивностью*, являющейся результатом применения ИТ, может привести к поразительному улучшению всех современных показателей деятельности организации, таких как уровень качества и обслуживания, скорость, затраты, цена, конкурентные преимущества, гибкость и т.д.

1.2.4. Подход ISA

В отличие от уже описанных подходов подход с использованием унифицированной схемы для архитектуры информационных систем (Information Systems Architecture — ISA) основан на проектировании снизу-вверх. Этот подход предлагает нейтральную архитектурную концептуальную схему для проектных решений по созданию ИС, которая может подходить для различных бизнес-стратегий. По существу, подход с использованием ISA не включает методологии планирования разработки систем. Он просто предлагает схему, которая может служить в качестве рычага для достижения целей большинства бизнес-стратегий.

Подход на основе схемы ISA был впервые введен Захманом (Zachman) в его плодотворной статье [97] и впоследствии расширен Сова (Sowa) и Захманом [82]. Незначительно видоизмененный вариант оригинальной статьи вновь был опубликован Захманом [98].

Схема ISA представляет собой таблицу из тридцати ячеек, организованных в виде пяти строк (помеченных от 1 до 5) и шести столбцов (помеченных от A до F). Строки представляют различные *точки зрения* (или *ракурсы*), используемые при конструировании сложных инженерных продуктов, таких как информационные системы. Эти точки зрения принадлежат пяти основным "игрокам" — пяти участникам разработки ИС.

1. Планировщик (определяет границы системы).
2. Владелец (определяет концептуальную модель предприятия).
3. Проектировщик (задает физическую модель системы).
4. Конструктор (обеспечивает детализированное технологическое решение).
5. Субподрядчик (поставляет компоненты системы).

Шесть столбцов представляют шесть различных *описаний* или *архитектурных моделей*, с которыми "играет" каждый участник. Подобно точкам зрения описания значительно отличаются между собой, но в то же время они внутренне связаны между собой. Описания призваны дать ответ на шесть вопросов в отношении моделируемых сущностей, которыми задается каждый из участников.

A. *Что* составляет сущность (т.е., в случае ИС, данные).
B. *Как* сущность функционирует (т.е. бизнес-процессы).
C. *Где* сущность расположена (т.е. расположение обрабатывающих компонентов).
D. *Кто* работает с сущностью (т.е. пользователи).
E. *Когда* с сущностью что-то происходит (т.е. распределения событий и состояний во времени).
F. *Почему* сущность существует (т.е. мотивация предприятия).

Комбинация точек зрения и описаний, представленных в тридцати ячейках таблицы Захмана, представляет собой мощную систематизацию, на основе которой можно выстроить полную архитектуру для разработки ИС. Расположенные по вертикали ракурсы могут отличаться степенью детализации, но, что более важно, они отличаются по существу и используют различные представления модели. Различные модели отражают различные взгляды участников. Аналогично расположенные по горизонтали описания подготовлены, исходя из различных соображений. Каждое из этих описаний призвано ответить на один из шести вопросов.

Наиболее привлекательной стороной подхода ISA является то, что он предлагает схему, которая, вполне вероятно, может оказаться достаточно гибкой для адаптации к будущим изменениям в условиях ведения бизнеса и в ресурсах, которыми располагает предприятие. Это является следствием того, что решения на основе ISA не базируются на какой-либо определенной бизнес-стратегии. Это просто схема для полного описания ИС. Сама схема получена на основании опыта, накопленного более устоявшимися дисциплинами, некоторые из них насчитывают тысячи лет (например, классическая архитектура).

1.2.5. Системы для трех уровней управления

Планирование разработки системы связано с представлением о наличии в организации трех уровней управления.

1. Стратегического.
2. Тактического.
3. Оперативного.

Эти три уровня организационного управления характеризуются собственным уровнем принимаемых решений, различным набором необходимых приложений ИС и специфическими требованиями к поддержке со стороны ИТ-подразделений. Одна из задач, возникающих при планировании разработки системы, состоит в определении комплекса приложений ИС и ИТ-решений, который является наиболее эффективным для организации в определенный момент времени. В табл. 1.1 определены связанные с установлением соответствия приложений ИС и ИТ-решений уровню принимаемых решений [40], [72].

Таблица 1.1. Поддержка различных уровней принятия решений со стороны ИС и ИТ

Уровень принятия решений	*Направленность принимаемых решений*	*Типичные приложения ИС*	*Типичные ИТ-решения*
Стратегический	Стратегии, поддерживающие долгосрочные цели организации	Анализ рынка и сбыта; планирование разработки товаров; оценка эффективности	Добыча данных; управление знаниями
Тактический	Стратегии, поддерживающие краткосрочные задачи и распределение ресурсов	Анализ бюджета; прогнозирование фонда зарплаты; планирование запасов; обслуживание потребителей	Хранилища данных; анализ данных; электронные таблицы
Оперативный	Поддержка повседневной деятельности персонала и производство	Выплата жалованья; выписка счетов; закупки; бухгалтерский учет	Базы данных; обработка транзакций; генераторы приложений

Реализуемые ИС-приложения и ИТ-решения, которые способны обеспечить наибольшие выгоды организации, относятся к *стратегическому уровню*. Однако эти решения наиболее трудно реализовать — они используют "пограничные" технологии и требуют очень квалифицированного и специализированного проектирования. Тем не менее, именно эти системы способны обеспечить организации конкурентное преимущество на рынке.

На противоположном конце шкалы располагаются системы, поддерживающие *оперативный уровень управления*. Эти системы отличаются однообразием действий и процедур, используют традиционные технологии баз данных и зачастую собираются из готовых к применению пакетных программных решений. Данные системы неперспективны с точки зрения обеспечения конкурентного преимущества, однако без них организация не способна функционировать надлежащим образом.

Любая современная организация имеет в своем распоряжении полный комплект систем оперативного управления, но только организации, которые достигли больших высот в искусстве управления, обладают интегрированным набором приложений ИС стратегического уровня. Основная технология, которая используется для хранения и обработки данных в процессе принятия высокоуровневых стратегических и тактических решений, известна как технология *хранилищ данных* [47].

1.3. Этапы жизненного цикла программного обеспечения

Разработка программного обеспечения подчиняется определенному *жизненному циклу* (*lifecycle*). Жизненный цикл (ЖЦ) — это упорядоченный набор видов деятельности, осуществляемый и управляемый в рамках каждого проекта по разработке ПО. Процессы и методы — это механизмы реализации жизненного цикла. Жизненный цикл определяет *этапы*, так что программный продукт переходит с одного этапа на другой, начиная с зарождения концепции продукта и заканчивая этапом его сворачивания.

Жизненный цикл разработки ПО может быть представлен с различной степенью детализации этапов. На *укрупненном уровне* ЖЦ может включать только три этапа.

1. Анализ.
2. Проектирование.
3. Реализация.

Этап анализа (*analysis phase*) концентрируется на системных требованиях. Требования *определяются* и *специфицируются*. Осуществляется разработка и интеграция функциональных моделей и моделей данных для системы. Кроме того, фиксируются нефункциональные требования и другие системные ограничения.

Этап проектирования (*design phase*) разделяется на два основных подэтапа: архитектурное и детализированное проектирование. В частности, проводится уточнение конструкции программы для архитектуры клиент/сервер, которая интегрирует объекты пользовательского интерфейса и базы данных. Поднимаются и фиксируются вопросы проектирования, которые влияют на понятность, приспособленность к сопровождению и масштабируемость системы.

Этап реализации (*implementation phase*) включает написание программ клиентских приложений и серверов баз данных. Акцент делается на итеративных процессах реализации с наращиванием возможностей системы. Успех поставки программного продукта не в последнюю очередь определяется циклической разработкой. *Циклическая разработка* (*round-trip engineering*) характеризуется периодическим возвратом от реализации клиентских приложений и серверов баз данных к проектным моделям и обратно.

Коротко говоря, анализ указывает на то, что делать, проектирование — на то, как с помощью имеющейся технологии сделать это “что”, а реализация воплощает задуманное на предыдущих этапах в виде осязаемого программного продукта, поставляемого заказчику.

На *детализированном уровне* ЖЦ можно разделить на следующие семь этапов.

1. Установление требований.
2. Спецификация требований.
3. Проектирование архитектуры.

4. Детализированное проектирование.
5. Реализация.
6. Интеграция.
7. Сопровождение (и окончательное сворачивание).

Некоторые авторы также рассматривают в качестве двух дополнительных этапов *планирование* и *тестирование*. По нашему мнению, эти два важных вида деятельности *не* являются отдельными этапами ЖЦ, поскольку охватывают весь ЖЦ в целом. План управления проектом по разработке ПО составляется в самом начале процесса, существенно уточняется после этапа спецификации и продолжает развиваться в течение всего оставшегося ЖЦ. Аналогично тестирование отличается наивысшей интенсивностью после этапа реализации, однако оно также применимо к программным артефактам, вырабатываемым на всех остальных этапах.

1.3.1. Этап установления требований

Котонья (Kotonya) и Соммервилль (Sommerville) [46] определяют *требование* (*requirement*) как "формулировку сути системного сервиса или ограничения". *Формулировка сути сервиса* характеризует поведение системы по отношению к отдельным пользователям или ко всему контингенту пользователей. В последнем случае описание сервиса фактически служит определением *бизнес-правила*, которое должно выполняться всегда (например, "двухнедельная зарплата выплачивается в среду"). Формулировка сути сервиса может быть связана с некоторым вычислением, которое должна произвести система (например, "вычислить комиссионные продавца на основе объема продаж на прошлую среду с использованием конкретной формулы").

Формулировка ограничения выражает ограничивающее условие на поведение системы или на разработку системы. Примером первого ограничения может быть ограничение на безопасность: "только непосредственные руководители могут обращаться к информации о зарплате их персонала". Примером последнего типа может быть формулировка: "мы должны использовать средства разработки компании Sybase."

Обратите внимание, что иногда различие между формулировкой ограничения на поведение системы и формулировкой услуги на основе бизнес-правила размыто. Это не составляет проблемы в том случае, если все требования идентифицированы и дублирование исключено.

Задачей этапа определения требований является определение, анализ и обсуждение требований с заказчиками. На этом этапе применяются различные методы сбора информации от заказчиков. Это и исследование концепции с помощью структурированных и неструктурированных интервью пользователей, анкеты, изучение документов и форм, видеозаписи и т.д. Последним методом, применяемым на этапе определения требований, является *быстрая разработка прототипа* (*rapid prototyping*) решения, так что требования, вызывающие затруднения, могут быть прояснены, что позволяет избежать недоразумений.

Анализ требований включает переговоры между разработчиками и заказчиками. Этот шаг необходим для исключения противоречивых и дублирующихся требований, а также согласования проектного бюджета и сроков.

Результатом этапа установления требований является документ, содержащий *изложение требований* (*requirements document*). Это большей частью текстовый документ с некоторыми неформальными диаграммами и таблицами. В этот документ, как правило, не

включаются формальные модели за исключением, может быть, некоторых простых и широко известных видов нотации, которые легко могут быть восприняты заказчиками и могут облегчить взаимопонимание между разработчиками и заказчиками.

1.3.2. Этап спецификации требований

Этап спецификации требований начинается с того момента, когда разработчики приступают к моделированию требований с использованием определенного метода (например, такого как UML). CASE-средства используются для ввода, анализа и документирования модели. В результате документ описания требований дополняется графическими моделями и отчетами, сгенерированными с помощью CASE-средств. По существу, документ, излагающий требования, заменяется документом, содержащим *спецификацию требований* (*specifications document*, иногда он обозначается жаргонным словечком *specs*).

В рамках объектно-ориентированного анализа в качестве основных методов спецификации требований используются два типа диаграмм: *диаграммы классов* (*class diagrams*) и *диаграммы прецедентов* (*use case diagrams*). Эти методы позволяют разрабатывать спецификации данных и функций. Обычно документ, содержащий спецификацию требований, включает также описание других видов требований. Среди атрибутов системы, требования к которым могут быть приведены в спецификации, относятся, например, производительность, пригодность к использованию (или практичность), пригодность к сопровождению, безопасность, политические и юридические требования, и даже "впечатления и ощущения от использования программы" ("look and feel").

Модели спецификации могут и должны перекрываться. Это позволяет рассмотреть предлагаемое решение под разными углами, выделяя и анализируя различные аспекты решения. Кроме того, это дает возможность проверить непротиворечивость и полноту требований.

В идеале модели спецификаций должны быть независимы от программной и аппаратной платформ, на которых должна разворачиваться система. Учет особенностей программной и аппаратной платформ накладывает жесткие ограничения на словарь (а следовательно — и на выразительность) языка моделирования. Более того, словарь может быть труден для понимания заказчиками и, таким образом, препятствовать общению разработчиков и заказчиков.

Тем не менее, некоторые формулировки ограничений могут фактически навязывать разработчикам необходимость рассмотрения особенностей программного и аппаратного обеспечения. Более того, сами заказчики могут быть выразителями требований относительно определенной технологии или даже требовать применения определенной технологии. Отсюда вывод: при возможности следует избегать рассмотрения особенностей программного и аппаратного обеспечения на этапе составления спецификации системы.

1.3.3. Этап проектирования архитектуры

Документально оформленная спецификация похожа на контракт между разработчиками и заказчиками на поставку программного продукта. В ней перечисляются все требования, которым должен удовлетворять программный продукт. Теперь спецификации передаются в руки системных архитекторов и проектировщиков для разработки детализированных моделей системной архитектуры и ее внутренних механизмов.

Проект выполняется в терминах программных и аппаратных платформ, на которых предстоит реализовать систему.

Описание системы в терминах составляющих ее модулей называется *архитектурным проектированием* (*architectural design*). Проект архитектуры включает выбор стратегических решений по клиентской и серверной частям системы.

Описание внутренних механизмов каждого модуля (прецедентов) называется *детализированным проектированием* (*detailed design*). Детализированный проект включает подробные алгоритмы и структуры данных для каждого модуля. Такие алгоритмы и структуры данных приспосабливаются ко всем ограничениям, связанным с базовой платформой реализации. Эти ограничения могут как усиливать основную архитектурную концепцию, так и препятствовать ее воплощению.

Архитектурное проектирование связано с выбором стратегии решения и разбивкой системы на модули. *Стратегия решения* (*solution strategy*) требует разрешения вопросов, касающихся клиентской (пользовательский интерфейс) и серверной (база данных) частей системы, а также всевозможного ПО промежуточного уровня (*middleware*), необходимого для связывания клиента и сервера. Выбор основных строительных блоков (модулей) относительно независим от стратегии решения, однако детализированный проект модулей должен соответствовать выбранному решению по архитектуре клиент/сервер.

Зачастую модели архитектуры клиент/сервер расширяются до так называемой *трехзвенной архитектуры* (*three-tier architecture*), в которой логика приложения составляет отдельный слой. Среднее звено представляет собой слой логики и в этом качестве может поддерживаться или не поддерживаться отдельным аппаратным обеспечением. Логика приложения — это процесс, который может выполняться на клиентской машине или на сервере, т.е. он скомпилирован в виде клиентского или серверного процесса и реализован как библиотека DLL (Dynamic Link Library — динамически подключаемая библиотека), интерфейс API (Application Programming Interface — интерфейс прикладного программирования), RPC-вызовов (Remote Procedure Calls— удаленный вызов процедуры) и т.д. [57].

1.3.4. Этап детализированного проектирования

Архитектурный проект описывает программный продукт с точки зрения составляющих его модулей. Детализированный проект описывает каждый модуль. При разработке типичной ИС модули реализуются либо в виде клиентской компоненты, либо серверной компоненты. За первые отвечают проектировщики прикладной части, вторую должны разрабатывать проектировщики баз данных.

Проект *пользовательского интерфейса* (клиентского приложения) должен соответствовать принципам проектирования GUI-интерфейса, установленным разработчиком конкретного GUI-интерфейса (Windows, Motif, Macintosh). Подобные принципы обычно доступны в WWW как часть электронной документации GUI-интерфейса (см. например, [92]).

Основной принцип объектно-ориентированного проектирования GUI-интерфейса состоит в том, что управление приложением *является прерогативой пользователя*, а не программы. Программа реагирует на случайные события, источником которых является пользователь, и предоставляет необходимый программный сервис. Остальные принципы проектирования GUI-интерфейса являются следствием этого факта. (Конечно, принцип “контроль является прерогативой пользователя” не следует при-

нимать буквально — программа по-прежнему может проверять права пользователя и запретить некоторые пользовательские действия.)

Проект базы данных определяет объекты сервера базы данных — скорее всего, реляционной (или, возможно, объектно-реляционной). Часть этих объектов представляет собой контейнеры данных (таблицы, взгляды и т.д.). Другие объекты являются процедурами (хранимые процедуры, триггеры и т.д.)

1.3.5. Этап реализации

Реализация информационной системы включает *инсталляцию* приобретенного ПО и *программирование* ПО, разрабатываемого под заказ. Кроме того, реализация подразумевает осуществление некоторых других важных мероприятий, таких как загрузка тестовых и производственных баз данных, тестирование, обучение пользователей, вопросы, связанные с аппаратным обеспечением, и т.д.

Типичная организация бригады по реализации проекта ПО предполагает разделение программистов на две группы: одну, ответственную за программирование клиентских приложений, и другую, которая должна отвечать за программирование сервера баз данных. Клиентские программы реализуют оконный интерфейс, логику приложений и при необходимости вызывают программы баз данных (хранимые процедуры). Ответственность за непротиворечивость баз данных и корректность транзакций лежит на серверных программах.

Как раз в духе итеративной и наращиваемой разработки проект *пользовательских интерфейсов* иногда подвергается значительным изменениям на этапе реализации. Прикладные программисты могут предпочитать — в зависимости от вида реализуемых диалоговых окон — следовать принципам разработки, предлагаемым поставщиком GUI-интерфейса, продвигать программирование или повышать продуктивность работы пользователей.

Аналогично реализация сервера *баз данных* может вызвать изменения в проектных документах. Непредвиденные проблемы, связанные с базами данных, трудности при программировании хранимых процедур и триггеров, вопросы параллелизма, интеграция с клиентскими процессами, настройка производительности и т.д. — вот перечень только некоторых причин, которые могут повлечь за собой необходимость модификации проекта.

1.3.6. Этап интеграции

Наращиваемая разработка предполагает *наращиваемую интеграцию* программных модулей. Эта задача не так проста, как может показаться на первый взгляд. Для больших систем интеграция отдельных модулей может потребовать больше времени и усилий, чем любой из более ранних этапов ЖЦ, включая реализацию. Еще Аристотель заметил, что целое больше простой суммы частей.

Интеграция модулей должна быть тщательно спланирована в самом начале жизненного цикла ПО. Программные компоненты, подлежащие отдельной реализации, должны быть идентифицированы на ранних стадиях анализа системы. К этому вопросу необходимо постоянно возвращаться, уточняя детали во время архитектурного проектирования. Порядок реализации должен позволять как можно более плавную наращиваемую интеграцию.

Основная трудность, связанная с наращиваемой интеграцией, заключается в существовании взаимных обратных зависимостей между модулями. Хороший проект сис-

темы отличается минимальной *связностью* (*coupling*) модулей. Тем не менее, время от времени два модуля оказываются зависимы друг от друга, так что ни один из них не может функционировать изолированно.

Что делать, если необходимо поставить один из модулей еще до того, как другой готов к применению? Ответ состоит в написании специальной программы для временного "заполнения бреши" так, чтобы все модули оказались интегрированными. Программная процедура, предназначенная для имитации работы отсутствующего модуля, называется *заглушкой* (*stubs*).

В отличие от традиционных программных систем, основанных на понятии главной программы, в современных объектно-ориентированных системах, управляемых событиями, отсутствует центральная интеллектуальная (главная) программа. Более того, в современных системах отсутствует четко определенная интеграционная структура. Традиционные стратегии интегрирования сверху-вниз или снизу-вверх не применимы к современным системам.

Объектно-ориентированные системы должны быть *спроектированы под интеграцию.* Каждый модуль должен быть как можно более независимым. Зависимости между модулями необходимо идентифицировать и минимизировать на этапах анализа и проектирования. В идеале, каждый модуль должен образовывать один поток обработки, который запускается в ответ на определенное требование клиента. Использования заглушек как операций замещения следует, по возможности, избегать. Если система спроектирована недостаточно качественно, этап интеграции приведет к хаосу и поставит под угрозу весь проект по разработке системы.

1.3.7. Этап сопровождения

Этап сопровождения наступает после успешной передачи заказчику каждого последующего программного модуля и, в конечном счете, всего программного продукта. Сопровождение — не только неотъемлемая часть жизненного цикла ПО; оно составляет его большую часть, если речь идет о времени и усилиях персонала ИТ-подразделений, приходящихся на сопровождение. Скеч (Schach) приводит оценку, по которой 67% времени ЖЦ приходится на сопровождение ПО [77].

Сопровождение состоит из трех различных стадий [51].

1. Поддержка эксплуатации.
2. Адаптивное сопровождение.
3. Улучшающее сопровождение.

Поддержка эксплуатации (*housekeeping*) связана с рутинными задачами сопровождения, необходимыми для поддержания системы в состоянии готовности к применению пользователями и эксплуатационным персоналом. Адаптивное сопровождение (*adaptive maintenance*) связано с отслеживанием и анализом работы системы, настройкой ее функциональных возможностей применительно к изменениям внешней среды и адаптацией системы для достижения заданной производительности и пропускной способности. Под *улучшающим сопровождением* (*perfective maintenance*) понимают перепроектирование и модификацию системы для удовлетворения новых или существенно изменившихся требований.

В конечном итоге продолжение сопровождения системы становится нецелесообразным, и ее следует свернуть. *Сворачивание* (*phasing out*) обычно осуществляется по причинам, которые имеют мало общего с утратой ПО своей *полезности*: оно, возможно, по-

прежнему остается вполне пригодным для *использования*, однако становится непригодным для *сопровождения*. Скеч приводит четыре причины сворачивания ПО [77].

1. Предлагаемые изменения выходят далеко за рамки ближайших возможностей улучшающего сопровождения.
2. Система выходит из-под контроля служб сопровождения, и последствия изменений невозможно предвидеть.
3. Расширение ПО в будущем невозможно из-за отсутствия надлежащей документации.
4. Аппаратная и/или программная платформы, на которых реализована система, подлежат замене, а видимых путей для миграции нет.

1.3.8. Планирование проекта в течение жизненного цикла ПО

Известное изречение гласит: то, что нельзя спланировать, нельзя и осуществить. Планирование охватывает весь жизненный цикл программного проекта. Оно начинается после того, как в результате работ по *системному планированию* определена бизнес-стратегия организации, и программный проект обозначен. *Проектное планирование* — это деятельность, направленная на оценку комплекта поставки, затрат, рисков, этапов и требуемых ресурсов. Оно также включает выбор методов разработки, процессов, средств, стандартов, бригадной организации и т.д.

Проектные планы подобны ускользающей цели. Они меньше всего напоминают что-то раз навсегда заданное и неизменное. Проектные планы подвержены изменениям на протяжении всего ЖЦ. При этом данные изменения не выходят за рамки, устанавливаемые несколькими *постоянными ограничениями*.

В качестве типичных ограничений выступают *время* и *деньги* — каждый проект имеет четкий конечный срок и строго ограниченный бюджет. Одна из первых задач проектного планирования состоит в оценке осуществимости проекта в условиях временных, бюджетных и прочих ограничений. Если проект осуществим, то ограничения фиксируются документально и могут быть изменены только в рамках формальной процедуры утверждения.

Осуществимость проекта оценивается с учетом нескольких факторов [37], [91].

- *Практическая осуществимость* связана с возвратом к вопросам, впервые поднятым при *системном планировании*, когда проект был обозначен; она связана с изучением того, как предлагаемая система повлияет на организационные структуры, процедуры и людей.
- *Экономическая осуществимость* связана с оценкой затрат на проект и приносимых им выгод (известной также как анализ затрат и результатов).
- *Техническая осуществимость* связана с оценкой практичности предлагаемых технических решений и наличия необходимых навыков, опыта и ресурсов.
- *Осуществимость по срокам* связана с оценкой обоснованности план-графика выполнения проекта.

Не все ограничения известны или могут быть оценены во время открытия проекта. На этапе выработки требований выявляются дополнительные ограничения, которые должны подвергаться изучению с точки зрения их влияния на осуществимость проекта. К подобным ограничениям можно отнести юридические, договорные, политические ограничения и ограничения, связанные с безопасностью.

С учетом оценки осуществимости составляется *проектный план* и устанавливаются правила управления проектом и процессом. В проектном плане находят отражение следующие вопросы [91].

- Рамки проекта.
- Проектные задания.
- Управление и контроль проекта.
- Управление качеством.
- Метрики и измерения.
- План-график проекта.
- Распределение ресурсов (людских, материальных, инструментальных).
- Руководство людьми.

1.3.9. Измерения в течение жизненного цикла ПО

Измерение времени и усилий, затраченных на проект, а также принятие на вооружение других *метрик* (*metrics*) для проектных артефактов в действительности является важной частью *управления проектом и процессом*. Несмотря на важность, в организациях с низким уровнем технологической зрелости этой частью часто пренебрегают. Цена здесь высока. Не "измеряя" прошлого, организация не в состоянии точно планировать будущее.

Метрики обычно рассматриваются в контексте *качества* и *сложности* ПО — они применяются в отношении качества и сложности *программного продукта* [36], [65]. Используются для измерения таких характеристик качества, как корректность, надежность, продуктивность, целостность, практичность, пригодность к сопровождению, гибкость и тестопригодность. Например, надежность ПО можно оценить с помощью измерения частоты и серьезности отказов, среднего времени между отказами, точности выходных результатов, восстанавливаемости после отказов и т.д.

Другим важным применением метрик является измерение моделей разработки (*продуктов разработки*) на различных этапах ЖЦ ПО. Затем метрики используются для оценки эффективности процесса и повышения качества работы на различных этапах ЖЦ.

К типичным метрикам, которые применяются в отношении *процесса создания* ПО и могут быть приняты к использованию на различных этапах ЖЦ, относятся следующие метрики.

- Изменчивость требований (процент требований, которые претерпевают изменения до завершения этапа спецификации требований). Эта метрика может отражать трудность получения требований от заказчиков.
- Изменчивость требований после этапа спецификации требований. Эта метрика может указывать на низкое качество документального оформления требований.
- Прогнозирование "мест перегрева" и "узкого горла" в системе (частота, с которой пользователи пытаются выполнить различные функции на прототипе программного продукта).
- Объем документов, содержащих спецификации, которые генерируются CASE-средствами, и другие более детализированные метрики, взятые из репозитория CASE-системы, например, такие как количество классов в модели классов. Если их

применить к нескольким прошлым проектам, затраты и время выполнения которых известны, эти метрики способны обеспечить идеальную плановую "базу данных" для прогнозирования времени и усилий, необходимых для будущих проектов.

- Фиксирование статистики отказов, времени их появления, обнаружения и устранения. Подобная статистика может отражать доскональности системы обеспечения качества, процессов пересмотра и деятельности по тестированию ПО.
- Среднее число тестов, после проведения которых тестируемая компонента считается готовой для интеграции поставки заказчику. Эта метрика может отражать уровень процедур отладки, используемых программистами.

1.3.10. Тестирование в течение жизненного цикла ПО

Подобно проектному планированию или измерению характеристик ПО, *тестирование* (*testing*) — это деятельность, охватывающая весь жизненный цикл ПО. Тестирование — это не просто отдельный этап ЖЦ, следующий за реализацией. После того, как дело зашло уже довольно далеко и программный продукт реализован, приступать к тестированию слишком поздно. Исправление ошибок, допущенных на ранних этапах ЖЦ, обходится безмерно дорого [77].

Действия по тестированию должны быть тщательно спланированы. Планирование процесса тестирования начинается с установления перечня *тестовых прецедентов* или *тест-прецедентов* (*test case*). Тест-прецеденты (или тест-планы) определяют шаги тестирования, которые необходимо предпринять, чтобы попытаться "сломать" программную модель или продукт.

Тест-прецеденты должны быть определены для каждого функционального модуля (*прецедента*), описанного в виде документально оформленных требований. Соотнесение тест-прецедентов с прецедентами устанавливает траекторию *прослеживаемости* (*traceability*) между тестами и требованиями пользователей. Тестопригодность программного артефакта определяется, в частности, его прослеживаемостью.

Вполне естественно, что каждый разработчик тестирует продукт своего труда. Однако у разработчиков-авторов программных артефактов их "детище" всегда на первом месте, так что от них трудно ожидать непредвзятого отношения к результатам своей работы.

Для более эффективного тестирования требуется привлечение независимых (по крайней мере — относительно) тестировщиков, которые могут провести методическое тестирование. Эту задачу можно поручить SQA-группе организации (*Software Quality Assurance* – обеспечение качества ПО). Группа должна включать лучших разработчиков организации. Их задача заключается в тестировании, а не в разработке. Затем на SQA-группу (но не на разработчиков-авторов) возлагается ответственность за качество программного продукта.

Чем больший объем тестирования выполнен на ранних этапах разработки, тем больше отдача. Требования, спецификации и любые другие документы (включая тексты программ) должны быть протестированы с использованием *формальных пересмотров* (*formal review*) (так называемого *сквозного контроля* (*walkthrough*) и *инспекции* (*inspection*)).

Формальные пересмотры — это тщательно подготовленные заседания, целью которых является анализ определенной части документации или системы. Специально назначенный рецензент заранее изучает документ и формулирует различные вопросы. Заседание решает, действительно ли поднятый вопрос вскрыл недочет, но не де-

лает при этом никаких попыток предложить немедленное решение проблемы. Позже разработчик-автор обращается к выявленному дефекту. При условии, что заседание проходит в дружественном духе, а участники избегают "указывать пальцем" на виновников ошибок, совместные усилия приводят к раннему обнаружению и исправлению многих ошибок.

После того, как программные прототипы и первые версии программного продукта становятся доступны, предпринимается *тестирование, основанное* на *выполнении программы* (*execution-based testing*). Существует два основных вида тестирования, основанного на прогонах.

- Тестирование по спецификации (тестирование по методу "черного ящика").
- Тестирование по программному коду (тестирование по методу "прозрачного ящика").

При *тестировании по спецификации* (*testing to specification*) сама программа рассматривается как "черный ящик", о котором ничего не известно, за исключением того, что он получает некоторую информацию на входе и вырабатывает некоторую информацию на выходе. На вход программы подаются некоторые входные данные, а результирующие данные анализируются на предмет наличия ошибок. Тестирование по спецификации особенно полезно для выявления некорректных или упущенных требований.

При *тестировании по программному коду* (*testing to code*) программная логика подвергается "сквозному просмотру" с целью установить, какие данные необходимо подать на вход, чтобы испытать различные выполняемые ветви программы. Тестирование по коду дополняет тестирование по спецификации — эти два вида тестирования направлены на выявление различных категорий ошибок.

Наращиваемая разработка предполагает не только наращиваемую интеграцию программных модулей, но и наращиваемое или *регрессионное тестирование* (*regression testing*). Регрессионное тестирование представляет собой повторное выполнение предыдущих тест-прецедентов на том же *базисном наборе данных* (*baseline data set*) после расширения — за счет наращивания возможностей — ранее выпущенных программных модулей. Тестирование проводится в предположении, что прежние функциональные возможности должны остаться неизменными и не должны быть нарушены за счет расширения.

Неплохим инструментом поддержки регрессионного тестирования могут служить *средства записи-воспроизведения* (*capture-playback tools*), которые позволяют зафиксировать взаимодействие пользователя с программой, а затем воспроизвести их без дальнейшего вмешательства пользователя. Основная трудность регрессионного тестирования заключается в нестабильности базисного набора данных. Дело в том, что наращиваемая разработка не только расширяет процедурную логику программы, но также расширяет и (модифицирует) основные структуры данных. Программный продукт с расширенными возможностями может заставить изменить базисный набор данных, таким образом лишая смысла сравнение результатов.

1.4. Подходы к разработке программного обеспечения

"Революция в программном обеспечении" вызвала ряд значительных изменений в способах работы программных продуктов. В частности, значительно увеличились *интерактивные возможности* программ. Задачи, выполняемые программами, и их поведение могут динамически адаптироваться к запросам пользователей.

Процедурная логика программ, написанных в прошлом на языках подобных языку COBOL, не отличалась гибкостью и не очень-то откликалась на неожиданные события. После запуска программа выполнялась до завершения в более или менее детерминированном режиме. Иногда программа могла запрашивать некоторую информацию от пользователя и следовать по различным выполняемым ветвям. Однако, в общем случае взаимодействие с пользователем было ограничено, а количество различных выполняемых ветвей заранее фиксировано. Контроль оставался за программой, а не пользователем.

С появлением современного графического пользовательского интерфейса (GUI) все кардинально изменилось. Программы на основе GUI-интерфейса управляются событиями, ход их выполнения носит случайный и непредсказуемый характер и диктуется инициируемыми пользователем событиями, источником которых являются клавиатура, мышь и другие устройства ввода.

В среде GUI-интерфейса пользователь контролирует (по большей части) выполнение программы, а не наоборот. За каждым событием стоит программный *объект*, который знает, как обслужить данное событие при текущем состоянии хода выполнения программы. После завершения обслуживания управление вновь возвращается к пользователю.

Различные стили программирования требуют разных подходов к разработке ПО. При разработке традиционного ПО хорошую службу сослужил *структурный подход*. Современные системы на основе GUI-интерфейса требуют объектного программирования, и объектный подход является наилучшим способом проектирования таких систем.

1.4.1. Структурный подход

Структурный подход (*structured approach*) к разработке систем получил широкое распространение (и был признан стандартом де-факто) в 1980-х годах. Этот подход основан на двух методах: диаграммах потоков данных ((data flow diagrams — DFD) для моделирования процессов и диаграммах сущность-связь (entity relationship diagrams — ERD) для моделирования данных.

Структурный подход является *функционально-ориентированным* и рассматривает DFD-диаграммы в качестве движущей силы разработки ПО. Позднее, в качестве одного из непосредственных результатов широкого распространения моделей реляционных баз данных, значение DFD-диаграмм в структурной разработке снизилось, и подход стал более *ориентированным на данные*, и, соответственно, акцент в разработке сместился на ERD-диаграммы.

Сочетание DFD- и ERD-диаграмм дает относительно полные модели анализа, которые фиксируют все функции и данные системы на требуемом уровне абстракции независимо от особенностей аппаратного и программного обеспечения. Затем модель анализа преобразуется в проектную модель, которая обычно выражается в понятиях реляционных баз данных. После этого следует этап реализации.

Структурный подход к анализу и проектированию отличается рядом особенностей, некоторые из них не очень хорошо увязываются с современными методами конструирования ПО.

- Этот подход скорее является *последовательным и трансформационным*, чем итеративным подходом с наращиванием возможностей (т.е. этот подход не способствует непрерывному процессу разработки, осуществляемому посредством итеративной детализации и пошаговой поставки ПО с наращенными возможностями).

- Этот подход направлен на поставку негибких решений, которые способны удовлетворить набор определенных бизнес-функций, но которые может быть трудно масштабировать и расширять в будущем.
- Этот подход предполагает разработку "с чистого листа" и не поддерживает повторное использование уже существующих компонент.

Трансформационный характер структурного подхода является источником повышенного риска неправильно истолковать исходные требования пользователей по ходу разработки. Такой риск усиливается за счет необходимости постепенной замены относительно декларативной семантики моделей анализа на процедурные решения для проектных моделей и программного кода реализации (это является следствием того, что модели анализа семантически богаче, чем базовые проектные и реализационные модели).

1.4.2. Объектно-ориентированный подход

Объектно-ориентированный подход (*object-oriented approach*) к разработке систем получил распространение в 1990-х годах. Ассоциация производителей ПО Object Management Group утвердила в качестве стандартного средства моделирования для этого подхода язык UML (Unified Modeling Language — Унифицированный язык моделирования).

По сравнению со структурным подходом объектно-ориентированный подход в большей степени *ориентирован на данные* — он развивается вокруг моделей классов. На этапе анализа для классов не требуется определять операции — только атрибуты. Возрастающее значение использования в языке UML *прецедентов* способствует незначительному смещению акцентов от данных к функциям.

Существует понимание того, что разработчики используют объектный подход вследствие технических преимуществ объектной парадигмы, таких как абстракция, инкапсуляция, повторное использование, наследование, передача сообщений, полиморфизм и т.д. Эти технические свойства могут привести к более высокому уровню повторного использования программного кода и данных, сокращению времени разработки, росту продуктивности труда программистов, повышению качества ПО, большей понятности программ и т.д.

При всей свой привлекательности данные преимущества объектной технологии до сих пор не нашли своего практического воплощения. Тем не менее мы продолжаем использовать сегодня объекты и будем использовать их завтра. Одной из причин этого является возможность работы с новым стилем программирования — программированием, *управляемым событиями* (*event-driven*), — который поддерживается современными GUI-интерфейсами.

Другой причиной популярности объектного подхода является возможность удовлетворить требования вновь *возникающих типов приложений* и находить наилучшие способы борьбы с *ростом количества невыполненных заказов на приложения*. К двум наиболее важным новым категориям приложений, для которых требуется объектная технология, относятся *вычислительная обработка для рабочих групп* и *системы мультимедиа*. Идея остановить рост числа невыполненных заказов посредством концепции "помещения объектов в оболочку" доказала как свою привлекательность, так и работоспособность.

Объектный подход к разработке систем следует *итеративному* процессу *с наращиванием возможностей*. Единая модель (и один проектный документ) конкретизируется на этапах анализа, проектирования и реализации — в результате успешных итераций

добавляются новые детали, при необходимости вводятся изменения и усовершенствования, а выпуски программных модулей с наращенными возможностями поддерживают высокий уровень удовлетворенности пользователей и обеспечивают обратную связь, необходимую для продолжения разработки модулей.

Разработка с помощью последовательной детализации становится возможной благодаря тому, что все создаваемые в ходе разработки модели (анализа, проектирования и реализации) обладают семантическим богатством и базируются на одном и том же "языке" — базовый словарь этих моделей существенно не отличается (классы, атрибуты, методы, наследование, полиморфизм и т.д.). Следует, однако, заметить, что если в основу реализации положена реляционная база данных, необходимость в сложной и связанной с риском трансформации по-прежнему сохраняется (поскольку базовая семантика реляционной модели сравнительно беднее).

Объектный подход устраняет большинство из наиболее значительных недостатков структурного подхода, однако, служит источником некоторых новых проблем.

- Этап анализа проводится на еще более высоком уровне абстракции, и — если серверная часть решения по реализации предполагает использование реляционной базы данных — *семантический разрыв* между концепцией и ее реализацией может быть значительным. Хотя анализ и проектирование могут проводиться итеративно с наращиванием возможностей, в конце концов разработка достигает этапа реализации, которая требует трансформации решения применительно к реляционной базе данных. Если в качестве платформы реализации используется объектная или объектно-реляционная база данных, трансформация проекта проходит значительно легче.
- *Управление проектом* сложно осуществлять. Менеджеры измеряют степень продвижения разработки с помощью четко определенной декомпозиции работ, элементов комплекта поставки и ключевых этапов. При объектной разработке с помощью "детализации" не существует четких границ между этапами, а проектная документация непрерывно развивается. Приемлемое решение в такой ситуации лежит в делении проекта на небольшие модули и управлении ходом разработки за счет частого выпуска выполняемых версий этих модулей (некоторые из этих выпусков могут быть для внутреннего применения, а другие — поставляться заказчику).
- Еще одна большая проблема применения объектного подхода связана с возрастающей сложностью решения, что, в свою очередь, сказывается на таких характеристиках ПО, как приспособленность к сопровождению и масштабируемость.

Сложности, связанные с объектным подходом, не влияют на тот факт, что по словам Артура Кларка "будущее нельзя повернуть вспять". Возврата к вызывающему ностальгию процедурному стилю пакетных приложений на языке COBOL нет. Все участники проектов по созданию ИС хорошо знакомы с Internet, электронной коммерцией, компьютерными играми и другими интерактивными приложениями.

Новые программные приложения отличаются значительно большей сложностью построения, и структурный подход совершенно не соответствует этой задаче. На сегодняшний день объектно-ориентированный подход — единственный известный метод, позволяющий совладать с разработкой нового управляемого событиями отличающегося высоким уровнем интерактивности ПО.

Резюме

В этой главе мы рассмотрели стратегические вопросы, касающиеся процесса разработки программного обеспечения. Для некоторых читателей содержимое этой главы значит не больше "родительской" нотации. Читателям, обладающим некоторым опытом в области разработки ПО, эта глава могла дать дополнительный интеллектуальный заряд. По отношению ко всем читателям данная глава была задумана как введение к предстоящему более всестороннему обсуждению.

По своей *природе* разработка ПО ближе к *ремеслу* или даже *искусству*. Результаты программного проекта нельзя полностью предвидеть при его начале. Основная *непредсказуемая трудность* при разработке ПО связана с участниками проекта — программный продукт должен дать участникам проекта ощутимые выгоды; в противном случае его ждет провал. В "треугольник" факторов, обеспечивающий успех проекта, помимо человеческого фактора входят стабильный *процесс* и поддержка *языка и средств моделирования*.

Цель разработки ПО — предоставить заказчику *продуктивную* систему. Разработке ПО предшествует *системное планирование*, в ходе которого определяется, какие продукты будут наиболее *эффективными* для организации. Существуют разные способы осуществить системное планирование. В данной главе были рассмотрены четыре известных подхода: SWOT, VCM, BPR и ISA. Системы, поддерживающие стратегический уровень принятия решений, приносят наибольшую выгоду. Эти же системы создают наибольшие проблемы для разработчиков.

Разработка программного обеспечения подчиняется определенному *жизненному циклу*. В этой книге наибольшее внимание уделяется двум этапам жизненного цикла: *анализу* и *проектированию*. Другие этапы включают реализацию, интеграцию и сопровождение. Разработку ПО составляют и некоторые другие виды деятельности, такие как проектное планирование, сбор данных измерений, тестирование и управление изменениями. Мы не относим эти виды деятельности к отдельным этапам, поскольку они регулярно повторяются на протяжении всего ЖЦ.

В прошлом программные продукты отличались *процедурным* характером — запрограммированная процедура выполняла свою задачу более-менее последовательным и предсказуемым образом, после чего завершалась. *Структурный подход к разработке* успешно использовался для производства подобных систем.

Современные программные продукты являются *объектно-ориентированными* — программа состоит из программных объектов, которые выполняются случайным, непредсказуемым образом, и программа не завершается до тех пор, пока пользователь не прекратит ее выполнение. Объекты "бездействуют", ожидая наступления инициированного пользователем события, чтобы начать вычисление; для выполнения задачи они могут запросить от других объектов предоставить им услуги, после чего снова впадают в "спячку", но сразу "поднимаются по тревоге" — стоит только пользователю инициировать другое событие. Современные клиент/серверные приложения ИС, разработанные с использованием GUI-интерфейса, являются объектно-ориентированными, и объектно-ориентированный подход к разработке наилучшим образом пригоден для производства таких приложений. Оставшаяся часть книги посвящена объектно-ориентированному подходу.

Вопросы

B1. Исходя из своего опыта в отношении программных продуктов, как бы вы могли проинтерпретировать замечание Фреда Брукса о том, что сущность программной инженерии проистекает из таких свойств ПО, как сложность, податливость, изменчивость и неосязаемость? Какое объяснение вы могли бы дать этим четырем факторам? В чем программная инженерия отличается от традиционных инженерных дисциплин, таких как строительство или машиностроение?

B2. Мы пытались доказать, что *производство ПО* — это *искусство* или *ремесло*. В качестве подтверждения этого тезиса можно привести высказывание о том, что "Искусство — это союз между Богом и художником, и чем меньше вкладывает в него художник, тем лучше" (Андре Жид). Какой урок могут вынести из этого высказывания разработчики ПО? Согласны ли вы с ним?

B3. Объясните разницу между программными *пакетами* и *компонентами*. Какое будущее, по-вашему, ожидает эти две технологии?

B4. Вспомните определение *участника предприятия*. Относятся ли продавец ПО и специалист по технической поддержке к участникам предприятия? Объясните свою точку зрения.

B5. Какой уровень технологической зрелости требуется для организации, чтобы овладеть кризисной ситуацией? Объясните свою точку зрения.

B6. В ходе объяснения SWOT-подхода к системному планированию мы заметили, что "верно сформулированная миссия отводит главное место потребностям клиентов, а не товарам или услугам, которые предоставляет организация". Пожалуйста, объясните и проиллюстрируйте, каким образом нацеливание *формулировки миссии* на определенные товары или услуги может привести к утрате основной цели системного планирования — достижения *эффективности*.

B7. Реинжиниринг бизнес-процессов (BPR) проводит ясное различие между *бизнес-процессом* и *бизнес-функцией*. В чем заключается это различие? Приведите пример бизнес-процесса, который разорван по горизонтали по всей организации.

B8. Сравните концепции *цепочек ценности* и *бизнес-процесса*.

B9. Почему понимание метода *ISA* (архитектура информационной системы) важно для системной разработки?

B10. Назовите три *уровня управления* организацией. Рассмотрите банковское приложение, которое предназначено для отслеживания стереотипов поведения владельцев кредитных карточек, чтобы автоматически блокировать карточку, если банк заподозрит злоупотребление (кража, подделка и т.д.). Какой уровень управления поддерживает подобное приложение? Обоснуйте свое заключение.

B11. Объясните разницу между этапами определения *требований* и разработки *спецификации*.

B12. Объясните взаимосвязь двух этапов проектирования (*архитектурное проектирование* и *детализированное проектирование*) с первыми двумя этапами жизненного цикла — этапом определения *требований* и этапом разработки *спецификации*.

B13. Что вы понимаете под соглашением, объектно-ориентированные системы должны *проектироваться под интеграцию*?

B14. Системное планирование и измерения ПО существенно связаны. Объясните этот тезис.

B15. Объясните взаимосвязь между прослеживаемостью и тестопригодностью.

B16. Какой основной метод моделирования применяется при структурном подходе к разработке?

B17. Каковы основные причины сдвига от структурного подхода к проектированию объектно-ориентированному?