

3. Процесс объектно-ориентированной разработки программного обеспечения

Процесс объектно-ориентированной разработки ПО не сводится к сумме рецептов, однако он определен достаточно хорошо, чтобы быть предсказуемым и воспроизводимым. В этом разделе мы подробно рассмотрим его как итеративно развивающийся процесс, описав цели, виды деятельности, результаты и меры прогресса, характерные для его различных фаз.

3.1. Основные принципы

Удачным проектом назовем тот, который удовлетворил (по возможности, превзошел) ожидания заказчика, уложился во временные и финансовые рамки, легко поддается изменению и адаптации. Выделяют две черты, которые являются общими для удачных проектов и отсутствуют у неудачных. Это: ясное представление об архитектуре создаваемой системы и хорошо организованный, итеративно развивающийся процесс работы над проектом [3]. Рассмотрим эти черты подробнее.

3.1.1. Архитектура

Архитектура объектно-ориентированной программной системы содержит структуры классов и объектов, имеющие горизонтальное и вертикальное слоение. Обычно конечного пользователя не интересует архитектура системы. Однако именно ясная внутренняя структура ПО играет важную роль в построении системы, которая будет понятна, тестируема, устойчива и сможет развиваться и перестраиваться [3]. Более того, ясность архитектуры позволяет уточнить абстракции и механизмы, тем самым, делая систему проще, меньше и надежнее.

Как же отличить хорошую архитектуру от плохой? Как правило, хорошая архитектура тяготеет к объектной ориентированности. Это не означает, что любая объектно-ориентированная архитектура оказывается хорошей, или что хороша только объектно-ориентированная архитектура. Однако применение принципов объектно-ориентированной декомпозиции приводит к архитектуре, обладающей свойствами организованной сложности.

Перечислим свойства присущие хорошей архитектуре.

1. Архитектура представляет собой многоуровневую систему абстракций. На каждом уровне абстракции, взаимодействуя друг с другом,

обеспечивают требуемое поведение системы, имеют четкий интерфейс с внешним миром и основываются на хорошо продуманных средствах нижнего уровня.

2. Интерфейс абстракций строго отграничен от реализации. Реализацию можно изменять, не затрагивая при этом интерфейс. Изменяясь внутренне, абстракции продолжают соответствовать ожиданиям внешних клиентов.

3. Архитектура проста, то есть не содержит ничего лишнего: общее поведение системы обеспечивается выявленными абстракциями и механизмами.

Описывая процесс объектно-ориентированной разработки ПО, будем различать стратегические и тактические решения. **Стратегическое решение** имеет важное архитектурное значение и связано с высоким уровнем системы. Механизмы обнаружения и обработки ошибок, парадигмы интерфейса пользователя, политика управления памятью, устойчивость объектов, синхронизация процессов, работающих в реальном масштабе времени, - все это стратегические архитектурные решения. В противоположность этому, **тактическое решение** имеет только локальное архитектурное значение и поэтому обычно связано с деталями интерфейса и реализации абстракций. Протокол класса, сигнатура метода, выбор алгоритма - все это тактические архитектурные решения.

Хорошая архитектура всегда демонстрирует баланс между стратегическими и тактическими решениями. При слабой стратегии даже очень изящно задуманный класс не сможет вполне соответствовать своей роли. Самые прозорливые стратегические решения будут разрушены, если не уделить должного внимания разработке отдельных классов. В обоих случаях пренебрежение архитектурой рождает программные эквиваленты анархии и неразберихи.

3.1.2. Организация итеративного процесса разработки ПО

Процесс создания ПО должен быть хорошо организованным. При этом важно соблюдать баланс между управляемостью и неформальностью творческой работы. Рассмотрим две крайности – полное отсутствие организации процесса разработки ПО и очень жесткие, строго соблюдаемые правила разработки. В первом случае мы имеем анархию. В результате тяжелого труда (преимущественно нескольких своих членов) команда разработчиков может создать что-то стоящее, но состояние проекта всегда будет неизмеримо и непредсказуемо. Следует ожидать, что команда работает весьма неэффективно, а, может быть, и вообще не создаст ничего пригодного для передачи заказчику. Это - пример проекта в свободном падении. Во втором случае, мы будем иметь диктатуру, в которой инициати-

вы наказуемы, экспериментирование, которое могло бы привнести больше элегантности в архитектуру, не поощряется, и действительные требования заказчика никогда корректно не доходят до разработчиков нижнего уровня, скрытых за бумажной стеной, воздвигнутой бюрократией.

Как примерить творчество с управляемостью? Каждый проект уникален, и, следовательно, разработчик сам должен поддерживать баланс между ними. Для исследовательских приложений, разрабатываемых тесно сплоченной командой высококвалифицированных разработчиков, чрезмерная формальность негативно отразится на новациях. Для очень сложных проектов, разрабатываемых большим коллективом разработчиков, отделенных друг от друга пространством и временем, недостаток формальности приводит к хаосу.

Реальный процесс создания ПО сочетает элементы как каскадной, так и итеративной моделей ЖЦПО. Существуют различные концепции относительно такого сочетания, однако все они так или иначе разграничивают макро- и микроэлементы процесса разработки ПО. Микропроцесс родственен спиральной модели развития и служит каркасом для итеративного подхода к развитию. Макропроцесс близок к традиционному «водопаду» и задает направляющие рамки для микропроцесса. Концепция, разработанная Гради Бучем, считается классическим подходом к объектно-ориентированной разработке ПО, подробно с ней можно ознакомиться в его книге «Объектно-ориентированный анализ и проектирование с примерами приложений на C++» [3]. В данном пособии будет рассмотрена концепция, которая задействует весь арсенал средств, предоставляемых UML, и поддерживается многими Case-средствами (Rational Rose, System Architect, Paradigm Plus и др.) [2,6].

3.2. Процесс объектно-ориентированной разработки программного обеспечения и UML

Процесс объектно-ориентированной разработки согласно рассматриваемой концепции включает в себя четыре фазы [2, 6]:

- начальная фаза (inception);
- уточнение (elaboration);
- конструирование (construction);
- внедрение и сопровождение (transition).

Рассмотрим перечисленные фазы подробнее.

3.2.1. Начальная фаза

Начальная фаза - это начало проекта. На этой фазе собирается информация и разрабатываются базовые концепции, определяется сколько

приблизительно будет стоить проект и какой доход принесет, а также выполняется начальный анализ для выявления границ проекта и оценки его размера. В это же время составляется план итераций, описывающий, какие варианты использования на каких итерациях должны быть реализованы. В конце этой фазы принимается решение продолжать или не продолжать проект.

Начальная фаза проекта в основном последовательна, а не итеративна. Так как проект может быть начат только один раз, начальная фаза также выполняется лишь однажды. В отличие от нее, другие фазы повторяются несколько.

Начальная фаза завершается, когда предварительные исследования закончены и для работы над проектом выделены необходимые ресурсы.

Начальная фаза может принимать множество разных форм. Для некоторых проектов это беседа за чашкой кофе: «Давайте подумаем насчет размещения каталога наших услуг в Интернете». Для более крупных проектов начальная фаза может вылиться во всестороннее изучение всех возможностей реализации проекта, которое займет месяцы.

3.2.2. Уточнение

В фазе уточнения выполняется планирование, а также анализ и проектирование для построения базовой архитектуры системы.

Фаза уточнения завершается, когда варианты использования полностью детализированы и одобрены пользователями, прототипы завершены настолько, чтобы уменьшить риски, разработаны диаграммы Классов. Иными словами, эта фаза пройдена, когда система спроектирована, рассмотрена и готова для передачи разработчикам.

Рассмотрим вопросы, связанные с построением *базовой архитектуры* системы. В фазе уточнения следует попытаться лучше уяснить свои задачи, ответив на следующие вопросы.

1. Что Вы на самом деле собираетесь создать?
2. Как Вы собираетесь это сделать?
3. Какую технологию Вы собираетесь использовать?

Основная задача фазы уточнения – детализация вариантов использования. (В разделе «Диаграммы Вариантов Использования» рассмотрено, что входит в детали вариантов использования.) Предъявляемые к варианту использования требования низкого уровня предусматривают описание потока событий внутри него, выявление действующих лиц, разработку диаграмм Взаимодействия для графического отображения потока событий, а также определение всех переходов состояний, которые могут иметь место в рамках варианта использования. На основе требований определенных в форме детализированных вариантов использования, составляется документ

под названием «Спецификация требований к программному обеспечению» (Software Requirement Specification, SRS). SRS содержит детальное описание всех требований к системе. Кроме того, в фазе уточнения проектируются диаграммы Классов, диаграммы Пакетов и Диаграммы Размещения. (Подробно эти вопросы были рассмотрены в разделе «Визуальное моделирование».)

Уточнение также включает в себя кодирование прототипов. Чтобы лучше понять требования к системе, следует построить прототип любых более или менее сложных вариантов использования. Прототипирование – это хороший способ для достижения наилучшего понимания динамики функционирования системы. Иногда вполне достаточно диаграмм, чтобы разобраться в ситуации, однако бывают и другие случаи, когда для правильного понимания происходящего не обойтись без прототипа. Обычно не следует заниматься прототипированием всего подряд, нужно используя общую модель предметной области, выделить те ее части, которые нуждаются в прототипировании.

Если Вы пользуетесь прототипом, не ограничивайтесь только той средой, в которой выполняете разработку конечного продукта. Для прототипирования нередко используют язык программирования отличный от того, на котором программируется сама система.

В фазе уточнения выполняются и такие задачи, как уточнение предварительных оценок, изучение SRS и модели вариантов использования с точки зрения качества проектируемой системы, анализ рисков. Решая, какие вопросы рассматривать во время этой фазы, нужно исходить в первую очередь из тех рисков, которые присущи проекту. Что может привести к провалу проекта? Чем больше риск, тем большее внимание ему следует уделить. (Подробно виды рисков описаны в разделе «Организация процесса объектно-ориентированной разработки программного обеспечения».)

Рассмотрим вопросы, связанные с осуществлением **планирования**.

Сущность планирования заключается в определении последовательности итераций конструирования и вариантов использования, реализуемых на каждой итерации. Планирование завершается, когда для всех вариантов использования определена своя итерация и назначена дата начала каждой итерации. В фазе уточнения более детальное планирование не делается.

Рассмотрим этапы составления планов.

Первый шаг заключается в отнесении вариантов использования к различным категориям. Для этого нужно выполнить следующие действия.

1. Пользователи должны указать уровень приоритетности для каждого варианта использования. Обычно используют три уровня:

- 1) эта функция абсолютно необходима для любой реальной системы;
- 2) какое-то небольшое время я смогу прожить без этой функции;
- 3) эта функция важна, но пока я могу обойтись и без нее.

2. Разработчики должны принимать во внимание «архитектурный» риск, связанный с каждым вариантом использования, который заключается в следующем: если реализацию данного варианта использования отложить на слишком долгое время, то вся выполненная до этого работа может потребовать значительной переделки. Для этой ситуации также используют три категории оценки:

- 1) высокая степень риска;
- 2) возможно, но маловероятно;
- 3) маленький риск.

3. Разработчики должны оценить степень своей уверенности относительно объема работы, требуемого для реализации каждого варианта использования. Это называют риском планирования. Здесь также полезно использовать три уровня:

- 1) я точно знаю, сколько времени потребуется на реализацию;
- 2) я могу оценить время только с точностью до человеко-месяца;
- 3) у меня нет никаких соображений по этому поводу.

После выполнения первого шага нужно с точностью до человеко-месяца оценить время, требуемое для реализации каждого варианта использования. Делая такую оценку, следует исходить из предположения, что придется выполнить анализ, проектирование, кодирование, автономное тестирование, интеграцию и документирование. Предполагается также, что разработчики будут полностью, без каких-либо отвлечений, заняты в данном проекте. Оценку должны выполнять разработчики, а не менеджеры.

Следующий шаг заключается в определении длительности итерации. Желательно, чтобы длина итерации была фиксированной на протяжении всего проекта, тогда можно будет получать результаты с заданной регулярностью. Итерация должна быть достаточно длительной для того, чтобы успеть реализовать некоторое количество вариантов использования.

После этого следует рассмотреть трудоемкость каждой итерации. Допустим, что эффективность работы разработчиков составляет в среднем 50%, т.е. на реализацию вариантов использования они расходуют половину своего рабочего времени. Умножьте длительность итерации на количество разработчиков и на 0,5. В результате получится трудоемкость каждой итерации. Например, если у Вас восемь разработчиков и продолжительность итерации составляет три недели, то на каждую итерацию потребуется 12 человеко-недель ($8 \times 3 \times 0,5$).

Просуммируйте время на реализацию всех вариантов использования, разделите на трудоемкость одной итерации и добавьте на всякий случай единицу. В результате получится начальная оценка количества итераций, которое потребуется для Вашего проекта.

Следующий шаг заключается в распределении вариантов использования по итерациям. Варианты использования, обладающие высоким приоритетом, «архитектурным» риском и/или риском планирования, следует

реализовывать в первую очередь. Не откладывайте риск до самого конца, Вам может потребоваться разделить слишком большие варианты использования на менее крупные или пересмотреть некоторые начальные оценки вариантов использования в соответствии с порядком их реализации.

Для приемки, отведите от 10% до 35% времени конструирования на тонкую настройку и компоновку конечного продукта. Если у вас нет опыта выполнения этих операций в данной среде, то отведите на них еще больше времени. Затем добавьте фактор случайности: от 10% до 20% времени конструирования, в зависимости от степени риска. Учтите воздействие этого фактора в конце фазы приемки. Для себя самого следует планировать выход конечного продукта без учета этой случайности, однако внешний план для пользователей должен ее учитывать.

После выполнения всех перечисленных рекомендаций у Вас будет план, в котором для каждой итерации указаны реализуемые варианты использования. Этот план должен отражать согласованное мнение разработчиков и пользователей.

3.2.3. Конструирование

На стадии конструирования построение системы выполняется путем серии итераций. Каждая итерация является своего рода мини-проектом. На каждой итерации для конкретных вариантов использования выполняются анализ, проектирование, кодирование, тестирование и интеграция. Итерация завершается демонстрацией результатов пользователям и выполнением системных тестов с целью контроля корректности реализации вариантов использования. Тестирование и интеграция - это достаточно крупные задачи, они всегда занимают больше времени, чем ожидается. Поэтому не следует откладывать их выполнение на самый конец проекта. (Подробно вопросы, связанные с тестированием и интеграцией, рассмотрены в разделе «Организация процесса объектно-ориентированной разработки программного обеспечения».)

Итерации на стадии конструирования являются одновременно инкрементными (наращиваемыми) и повторяющимися. Итерации являются **инкрементными** по отношению к той функции, которую они выполняют. Каждая итерация добавляет очередные конструкции к вариантам использования, реализованным во время предыдущих итераций. Итерации являются **повторяющимися** по отношению к разрабатываемому коду. На каждой итерации некоторая часть существующего кода заново переписывается с целью сделать его более гибким. В этом процессе очень часто используется метод реорганизации (refactoring). Стоит внимательно понаблюдать за тем, какой объем кода оказывается ненужным после каждой итерации. Если каждый раз выбрасывается менее 10% предыдущего кода, то это долж-

но вызывать подозрение. (Подробно вопросы, связанные с выполнением реорганизации, рассмотрены в разделе «Организация процесса объектно-ориентированной разработки программного обеспечения».)

В рамках каждой итерации следует также заниматься *детальным планированием*. Самой важной частью любого плана являются меры, которые нужно предпринимать, если что-то происходит не так, как было запланировано. Нужно отметить, что такое случается всегда.

Главная особенность итеративной разработки заключается в том, что она жестко ограничена временными рамками, и сдвигать сроки недопустимо. Исключением может быть перенос реализации каких-либо вариантов использования на более позднюю итерацию по соглашению с заказчиком. Смысл таких ограничений - поддерживать строгую дисциплину разработки и не допускать переноса сроков.

Если Вы все же перенесли много вариантов использования на более поздний срок, то пора корректировать план, пересмотрев при этом оценку трудоемкости реализации вариантов использования.

Так как система в основном проектируется в фазе уточнения, конструирование не предполагает большого количества решений по проекту, что позволяет команде работать параллельно. Это означает, что разные группы программистов могут одновременно работать над различными объектами ПО, зная, что после завершения фазы система «сойдется».

Визуальное моделирование обладает тем преимуществом, что некоторые Case-средства (например, Rational Rose) способны генерировать «скелетный код» системы. Для использования этой возможности следует разработать компоненты и диаграмму компонентов. Генерацию кода можно начать сразу после создания компонентов и нанесения на диаграмму зависимостей между ними. Это не означает, что с помощью Case-средств можно получить любой код, реализующий бизнес-логику приложений, но в общем случае можно сгенерировать определение классов, атрибутов, областей действия (общих (public), закрытых (private) и защищенных (protected)), прототипов функций и операторов наследования. Это позволяет сэкономить время, так как написание кода вручную – довольно кропотливая и утомительная работа. Получив код, программисты могут сконцентрироваться на специфических аспектах проекта, связанных с бизнес-логикой. Еще одна группа разработчиков должна выполнить экспертную оценку кода, чтобы убедиться в его функциональности и соответствии стандартам и соглашениям по проекту. Затем объекты должны быть подвергнуты оценке качества. Если в фазе конструирования были добавлены новые атрибуты или функции или если были изменены взаимодействия между объектами, код следует преобразовать в модель Case-средства с помощью обратного проектирования.

Конструирование можно считать завершенным, когда программное обеспечение готово и протестировано. Важно убедиться в адекватности

модели и программного обеспечения; модель будет чрезвычайно полезна в процессе сопровождения ПО.

3.2.4. Ввод в действие

Фаза ввода в действие наступает, когда готовый программный продукт передают пользователям. Задачи в этой фазе предполагают завершение работы над финальной версией продукта, завершение приемочного тестирования, завершение составления документации и подготовку к обучению пользователей. Чтобы отразить последние внесенные изменения, следует обновить Спецификацию требований к программному обеспечению, диаграммы Вариантов Использования, Классов, Компонентов и Размещения. Важно, чтобы модели были синхронизированы с готовым продуктом, поскольку они будут использоваться при его сопровождении. Кроме того, модели будут неоценимы при внесении усовершенствований в созданную систему через несколько месяцев после завершения проекта.

На стадии приемки продукт не дополняется никакой новой функциональностью (кроме, разве что, самой минимальной и абсолютно необходимой). Здесь только вылавливаются ошибки, и проводится оптимизация. (Подробно вопросы, связанные с оптимизацией, рассмотрены в разделе «Организация процесса объектно-ориентированной разработки программного обеспечения».) Хорошим примером для фазы приемки может служить период времени между выпуском бета-версии и появлением окончательной версии продукта.

3.3. Организация процесса объектно-ориентированной разработки программного обеспечения

Рассмотрим некоторые вопросы, связанные с организацией объектно-ориентированной разработки программного обеспечения, которые не были включены в другие разделы.

1. Роли разработчиков. Разработчики программного обеспечения - не взаимозаменяемые части. Успешное создание любой сложной системы требует уникальных и разнообразных навыков всех членов коллектива. Опыт показывает, что объектно-ориентированная разработка требует иного разделения труда по сравнению с традиционными методами. Важнейшими в объектно-ориентированном подходе считаются следующие роли разработчиков: архитектор проекта, ответственные за подсистемы, прикладные программисты. Рассмотрим их подробнее.

Архитектор проекта отвечает за эволюцию и сопровождение архитектуры системы. Для малых или средних систем архитектурное проекти-

рование обычно выполняется одним или двумя архитекторами. Для больших проектов эта обязанность может быть распределена в большом коллективе. Архитектор проекта - не обязательно самый главный разработчик, но непременно такой, который может квалифицированно принимать стратегические решения (как правило, благодаря большому опыту в построении систем такого типа). Благодаря опыту, разработчики интуитивно знают, какие общие архитектурные шаблоны уместны в данной предметной области и какие проблемы эффективности встают в определенных архитектурных вариантах. Архитекторы - не обязательно лучшие программисты, хотя они должны уметь программировать. Архитекторы проекта должны также быть сведущи в обозначениях и организации процесса объектно-ориентированной разработки, потому что они должны в конечном счете выразить свое архитектурное видение в терминах классов и объектов. Лучше привлекать архитектора к активной работе уже при проведении анализа и оставлять его на как можно более длительный срок, даже на все время эволюции системы. Тогда он освоится с действительными потребностями системы и со временем испытает на себе последствия своих решений. Кроме того, сохраняя в руках одного человека или небольшой команды разработчиков ответственность за архитектурную целостность, мы повышаем шансы получить гибкую и простую архитектуру.

Ответственные за подсистемы - главные создатели абстракций проекта. Они отвечают за проектирование целых категорий классов или подсистем. Каждый ответственный в сотрудничестве с архитектором проекта разрабатывает, обосновывает и согласует с другими разработчиками интерфейс своей категории классов или подсистемы, а потом возглавляет ее реализацию, тестирование и выпуск релизов в течение всей эволюции системы. Ответственные за подсистемы должны хорошо знать систему обозначений и организацию процесса объектно-ориентированной разработки. Обычно они программируют лучше, чем архитекторы проекта, но не располагают обширным опытом последних. Ответственные за подсистемы составляют от трети до половины численности команды.

Прикладные программисты (инженеры) - младшие по рангу участники проекта. Эта деятельность может включать в себя проектирование некоторых классов, но в основном связана с реализацией и последующим тестированием классов и механизмов, разработанных проектировщиками команды. Инженеры могут не слишком хорошо разбираться в системе обозначений и в организации процесса разработки, но они являются очень хорошими программистами, знающими основные идиомы и слабые места выбранных языков программирования. Инженеры составляют половину или более команды.

В больших проектах могут потребоваться и другие роли (табл. 3.1). Большинство из них (например, роль специалиста в средствах разработки) явно не связаны с объектно-ориентированной технологией, но некоторые

непосредственно вытекают из нее (такие, как инженер, отвечающей за повторное использование).

Таблица 3.1. Роли разработчиков

Роли	Обязанности
Менеджер проекта	Отвечает за управление материалами проекта, заданиями, ресурсами и графиком работ
Аналитик	Отвечает за развитие и интерпретацию требований конечных пользователей; должен быть экспертом в предметной области
Инженер по повторному использованию	Управляет хранилищем (репозитарием) материалов проекта, активно ищет общее и добивается его использования; находит, разрабатывает или приспосабливает компоненты для общего использования
Контролер качества	Измеряет результаты процесса разработки; задает общее направление тестирования всех прототипов и релизов
Менеджер интеграции	Отвечает за сборку совместимых друг с другом версий категорий и подсистем в релизы; следит за их конфигурированием
Ответственный за документацию	Готовит документацию по выпускаемому продукту и его архитектуре для конечного пользователя
Инструментальщик	Отвечает за создание и адаптацию инструментов программирования, которые облегчают производство программ и (особенно) генерацию кода
Системный администратор	Управляет физическими компьютерными ресурсами в проекте

Для небольших проектов обязанности могут совмещаться. С другой стороны, для очень больших проектов каждой из ролей может заниматься целая организация. Объектно-ориентированная разработка может обойтись меньшим числом занятых в ней людей по сравнению с традиционными методами. Чтобы за один год произвести высококачественную программу объемом несколько сот тысяч строк, достаточно 30 - 40 разработчиков. Однако лучшие результаты достигаются в том случае, когда задействовано меньшее количество разработчиков, но с более высокой квалификацией.

Кадровая политика должна состоять в том, чтобы молодые разработчики работали под руководством более опытных. Не обязательно каждому разработчику быть экспертом по абстракциям, но каждый разработчик может со временем этому научиться.

2. Интеграция и управление версиями. В процессе объектно-ориентированной разработки происходит множество мелких интеграций,

каждая из которых соответствует созданию нового прототипа или архитектурного релиза. Каждый новый релиз поступательно развивает предыдущие, удовлетворяя все большему числу требований, и в конечном счете развивается в готовую систему. Такая стратегия постепенного развития позволяет своевременного выявлять и устранять возникающие проблемы.

Общий принцип таков, что процесс интеграции должен быть непрерывным. В конце каждой итерации должна выполняться полная интеграция. Интеграция может и должна выполняться даже еще чаще. Разработчику следует интегрировать приложения после выполнения любой сколько-нибудь значительной части работы. Во время каждой интеграции должен выполняться полный набор автономных тестов, чтобы обеспечить полное регрессионное тестирование.

Рассмотрим поток релизов с точки зрения разработчика, занятого созданием некоторой подсистемы. Подсистема - это, как правило, связанная группа классов. Когда стабилизируется рабочая версия подсистемы, разработчик передает ее команде, отвечающей за интеграцию. Эта команда собирает совместимые подсистемы в единую систему. В конце концов набор подсистем «замораживается», берется за точку отсчета и входит во внутренний релиз. Этот релиз становится доступен всем разработчикам, которые проводят дальнейшую доработку своих частей. Таким образом, благодаря четко определенным и защищенным интерфейсам подсистем, объектно-ориентированная разработка может вестись параллельно. Каждая подсистема располагает интерфейсами тех подсистем, которые с ней взаимодействуют.

Внутренних релизов выпускается гораздо больше, чем релизов для конечного пользователя. В больших проектах внутренние релизы готовятся через несколько недель, релизы для пользователя раз в несколько месяцев. Под релизом понимают не только текст программы, но и все другие продукты объектно-ориентированной разработки: технические требования, диаграммы и т.п.

В любой момент разработки системы могут существовать несколько версий ее подсистем: версия для текущего разрабатываемого релиза, версия для текущего внутреннего релиза, версия для последующего релиза, предназначенного для заказчика и т.д.

3. Тестирование. Принцип непрерывной интеграции приложим и к тестированию, которое также производится в течение всего процесса разработки. Чем позже выполняются тестирование и интеграция, тем более трудными задачами они становятся и тем большую дезорганизацию могут внести в проект. Все это ведет к высокой степени риска. При итеративной разработке на каждой итерации выполняется весь процесс, что позволяет оперативно справляться со всеми возникающими проблемами.

Во времена создания OS/360 Фред Брукс определил, что тестирование занимает половину проектного времени (включая устранение неиз-

бежных ошибок) [6]. М. Фаулер и К. Скотт в своей книге «UML Distilled. Applying the Standard Object Modeling Language» отмечают относительно тестирования следующее: «С возрастом я все более серьезно отношусь к тестированию. Я предпочитаю следовать практическому правилу Кента Бека, согласно которому объем написанного разработчиком кода тестов должен быть по меньшей мере равен объему кода самого программного продукта. Тестирование должно быть непрерывным процессом. Не следует писать программный код до тех пор, пока Вы не знаете, как его тестировать. Как только Вы написали код, сразу же пишите для него тесты. Пока все тесты не отработают, нельзя утверждать, что Вы завершили написание кода» [6].

Однажды написанный тестовый код должен использоваться постоянно. Тестовый код должен быть организован таким образом, чтобы можно было запускать каждый тест с помощью простой командной строки или нажатия кнопки на графическом экране. Результатом выполнения теста должно быть «ОК» или список ошибок. Кроме того, все тесты должны проверять свои собственные результаты. Ничто не приводит к столь неоправданной трате времени, как получение на выходе теста числового значения, с интерпретацией которого нужно разбираться отдельно.

Разделите все тесты на автономные и системные. Автономные тесты нужно писать самим разработчикам. Они должны быть организованы в пакеты и тестировать интерфейсы всех классов. Системные тесты должны разрабатываться отдельной небольшой командой, которая занимается исключительно тестированием, рассматривая всю систему как черный ящик.

В контексте объектно-ориентированной разработки тестирование должно охватывать как минимум три направления:

1) тестирование модулей, которое предполагает тестирование отдельных классов и механизмов и является обязанностью инженера, который их реализовал;

2) тестирование подсистем, которое предполагает тестирование целых категорий или подсистем и является обязанностью ответственного за подсистему; тесты подсистем могут использоваться регрессивно для каждой вновь выпускаемой версии подсистемы;

3) тестирование системы, которое предполагает тестирование системы как целого и является обязанностью контролеров качества; тестирование системы, как правило, тоже происходит регрессивно.

Тестирование должно фокусироваться на внешнем поведении системы; его побочная цель - определить границы системы, чтобы понять, как она может выходить из строя при определенных условиях.

4. Повторное использование. Любой программный продукт (тексты программ, архитектура, сценарии или документация) может быть использован повторно. При объектно-ориентированной разработке можно использовать повторно: классы, объекты, механизмы и другие элементы про-

ектирования, а также их шаблоны. Повторное использование шаблонов элементов приносит больший эффект, чем использование конкретных элементов.

В удачных проектах количество повторно использованных элементов может достигать до 70 % (то есть почти три четверти программного обеспечения системы было взято без изменений из некоторого другого источника), но может быть и нулевым. Это зависит от многих факторов. В любом случае повторное использование позволяет экономить ресурсы, которые иначе пришлось бы потратить еще раз. Повторное использование может и не принести краткосрочных выгод, но окупается в долгосрочной перспективе. Этим имеет смысл заниматься в организации, которая имеет обширные, далеко идущие планы разработки программного обеспечения и смотрит дальше интересов текущего проекта.

Повторное использование не должно протекать по воле случая. Им нужно заниматься специально, лучше всего поручить его конкретным работникам.

5. Реорганизация. Рассмотрим подробнее вопросы, связанные с *реорганизацией*.

В программном обеспечении существует принцип энтропии. Он предполагает, что программы начинают свой жизненный цикл достаточно хорошо спроектированными, но по мере добавления хотя бы самой малой функциональности теряют строгость своей структуры, превращаясь в конечном счете в бесформенную массу «спагетти».

Отчасти это происходит благодаря масштабу. Предположим, Вы разработали небольшую программу, которая вполне удовлетворительно выполняет конкретную работу. Пользователи все время просят Вас расширить ее функции, в результате чего она становится все более и более сложной. Как бы тщательно Вы ни следили за проектом, это все равно может произойти.

Одной из причин возрастания энтропии является дописывание нового кода поверх существующего, не обращая при этом особого внимания на прежнюю программу. Хотя при добавлении новой функции программу лучше перепроектировать, обычно это выливается в большую дополнительную работу, поскольку переписывание существующей программы порождает новые ошибки и проблемы. У инженеров есть поговорка: «Если не сломалось - не переделывай». Однако, если Вы не будете перепроектировать программу, дополнения могут оказаться более сложными, чем могли бы быть в противном случае.

Со временем за такую сверхсложность придется платить все более высокую цену. Вследствие этого приходится идти на компромисс: перепроектирование влечет за собой серьезные трудности в краткосрочном плане, зато приносит выгоду в перспективе. Находясь под прессом жест-

ких плановых сроков, большинство разработчиков предпочитает отодвигать эти трудности на будущее.

Термин «реорганизация» используется для описания методов, которые позволяют уменьшить краткосрочные трудности, связанные с пере проектированием. Реорганизация не изменяет функциональность программы; однако при этом Вы изменяете ее внутреннюю структуру для того, чтобы сделать ее более понятной и модифицируемой.

Реорганизационные изменения обычно довольно невелики: переименование метода, перемещение атрибута из одного класса в другой, консолидация двух подобных методов в суперкласс. Каждое отдельное изменение очень мало, однако даже пара часов, потраченных на внесение таких небольших изменений, могут существенно улучшить программу.

Реорганизацию можно облегчить, если следовать следующим принципам.

1. Не следует одновременно заниматься реорганизацией программы и расширением ее функциональности. Необходимо проводить четкую границу между этими действиями. В процессе работы можно переключаться от одного действия к другому - например, в течение получаса заниматься реорганизацией, затем потратить час на добавление новой функции и полчаса на реорганизацию только что добавленного кода.

2. Перед началом реорганизации убедитесь, что у Вас имеются хорошие тесты. Запускайте их как можно чаще. Таким образом, Вы быстро узнаете, не нарушили ли чего-нибудь Ваши изменения.

3. Вносите изменения минимальными и тщательно обдуманными шагами. Переместите атрибут из одного класса в другой. Объедините два подобных метода в один суперкласс. Реорганизация часто включает выполнение множества небольших локальных изменений, которые в конечном счете выливаются в крупномасштабное изменение. Если каждый шаг будет небольшим и будет завершаться обязательным тестированием, то Вы сможете избежать в будущем длительной отладки.

Реорганизацию следует выполнять в следующих случаях.

1. Вы расширяете функциональность своей программы и обнаружили, что с существующим кодом возникают некоторые проблемы. Тогда процесс расширения следует приостановить до тех пор, пока не будет реорганизован старый код.

2. Код становится трудным для понимания. В этом случае реорганизация служит хорошим способом облегчить понимание кода и сохранить это понимание на будущее.

Потребность в реорганизации часто возникает, когда приходится иметь дело с кодом, написанным каким-либо другим разработчиком. Если Вы занимаетесь такой реорганизацией, то делайте это вместе с автором кода. Не так просто написать код, который другие могли бы легко понимать.

К сожалению, пока реорганизация применяется гораздо реже, чем следовало бы. Тем не менее она является одним из ключевых методов совершенствования разработки ПО, независимо от того, в какой среде вы работаете.

6. Оптимизация. Главная идея итеративной разработки - поставить весь процесс разработки на регулярную основу с тем, чтобы команда разработчиков смогла получить конечный продукт. Однако есть некоторые вещи, которые не следует выполнять слишком рано, например оптимизация. *Оптимизация* снижает прозрачность и расширяемость системы, однако повышает ее производительность. В этой ситуации Вы оказываетесь перед необходимостью принятия компромиссного решения - в конце концов, система должна быть достаточно производительной, чтобы удовлетворять пользовательским требованиям. Слишком ранняя оптимизация затруднит последующую разработку, поэтому ее следует выполнять в последнюю очередь.

7. Риски. Исходя из опыта, риски полезно разделить на четыре категории.

Риски, связанные с требованиями. Каковы требования к системе? Самая большая опасность заключается в том, что построенная вами система не будет удовлетворять требованиям пользователей. Во время фазы уточнения вам необходимо как следует разобраться с требованиями и их относительными приоритетами.

Технологические риски. С какими технологическими рисками вам придется столкнуться? Задайте себе следующие вопросы.

Если Вы собираетесь использовать объекты, есть ли у вас достаточный опыт объектно-ориентированного проектирования?

Если Вы собираетесь использовать язык Java и Web, насколько хорошо эта технология работает? Сможете ли вы действительно реализовать те функции, которых пользователи ждут от Web-браузера, связанного с базой данных?

Риски, связанные с квалификацией персонала. Сможете ли вы получить сотрудников с необходимым опытом и квалификацией?

Политические риски. Существуют ли политические силы, которые могут оказаться на вашем пути и серьезно повлиять на проект?

Рисков может быть и больше, но перечисленные категории рисков присутствуют почти всегда.

8. Документация. Разработка программной системы включает в себя гораздо больше, чем просто написание кода. Основные аспекты проекта должны быть постоянно доступны всем участникам разработки и внешним пользователям. Также нужно сохранить сведения о решениях, принятых при анализе и проектировании, для тех, кто будет заниматься сопровождением системы.

Что должно быть документировано? Документация, представляемая конечному пользователю, должна включать инструкции по установке и использованию системы. Кроме того, должны быть документированы результаты анализа. Результатом анализа является описание назначения системы, сопровождаемое характеристиками производительности и перечислением требуемых ресурсов. Часто эти результаты анализа объединяют в один формальный документ, который формулирует требования анализа к поведению системы, иллюстрируя их диаграммами, и показывает такие неповеденческие аспекты системы, как эффективность, надежность, защищенность и переносимость. Должна также вестись документация по архитектуре и реализации. Архитектура объектно-ориентированной системы выражает структуру классов и объектов (логическая архитектура), а также структуру модулей (физическая архитектура). Эти сведения можно объединить в формальный документ, описывающий архитектуру, который должен быть доступен всем членам коллектива для согласования в команде разработчиков общего видения системы и деталей архитектуры, а также для того, чтобы сохранить информацию о решениях, принимаемых по данным вопросам.

Основные результаты объектно-ориентированной разработки отображаются на диаграммах. Как уже отмечалось в разделе «Визуальное моделирование», люди легче понимают сложную информацию, если она представлена визуально (графически), нежели описана в тексте. Важно также чтобы использовалась единая система обозначений, которая способствовала бы пониманию деталей проекта всеми его участниками. Язык UML предусматривает следующие виды диаграмм: диаграммы Вариантов Исполнения, диаграммы Классов, диаграммы Взаимодействия, диаграммы Состояний, диаграммы Компонентов и диаграммы Размещения. В совокупности эти диаграммы предоставляют возможность проследить их появление непосредственно из начальных требований к системе. Диаграммы Размещения показывают физическое расположение различных компонентов системы в сети. Они содержат процессоры, устройства, процессы и связи между процессорами и устройствами. На диаграммах Компонентов показываются основные модули системы. Каждый модуль представляет реализацию классов, которые можно найти на диаграммах Классов. Классы определяются из диаграмм Взаимодействия. На диаграммах взаимодействия показаны сценарии для требований представленных диаграммами Вариантов исполнения. В конечном счете на диаграммах Вариантов исполнения отражены требования предъявляемые к системе.

Подразумевается, что документация системы эволюционирует вместе с архитектурными релизами. Чтобы не относиться к ведению документации как к основному занятию, лучше всего получать ее как естественный, полуавтоматически генерируемый побочный продукт эволюционного процесса.

9. Методы оценки качества программного продукта. Качество программы не должно быть делом случая. Качество должно гарантироваться процессом разработки. На самом деле, объектно-ориентированная технология не порождает качества автоматически: можно написать сколь угодно плохие программы на любом объектно-ориентированном языке программирования.

В процессе объектно-ориентированной разработки основное значение придается архитектуре программной системы. Качество закладывается благодаря простой, гибкой архитектуре и осуществляется естественными и последовательными тактическими решениями.

Контроль качества программного продукта - это «систематические действия, подтверждающие пригодность к использованию программного продукта в целом» [3]. Цель контроля качества - дать нам количественные меры качества программной системы.

Одним из количественных критериев качества является *количество обнаруженных ошибок*. Во время эволюции системы ошибки учитываются в соответствии с их весом и расположением. График обнаружения ошибок отображает зависимость количества обнаруженных ошибок от времени. Не так важно действительное число ошибок, как наклон этого графика. Для управляемого процесса этот график имеет форму горба с вершиной примерно в середине периода тестирования, а дальше эта кривая падает до нуля. Неуправляемому процессу соответствует неубывающая со временем или медленно убывающая кривая.

Достоинством итеративного развития является возможность вести непрерывный сбор данных о количестве обнаруженных ошибок уже на ранних стадиях разработки. Для каждого нового релиза можно провести тестирование системы и нарисовать график зависимости количества ошибок от времени. У «здорового» проекта горбовидная форма этого графика наблюдается для каждого релиза, начиная с самых ранних.

Другая количественная мера - *плотность ошибок*. Количество обнаруженных ошибок на килостроку программного текста (KSLOC - Kilo Source Lines Of Code) является традиционным показателем, применимым, в частности, к объектно-ориентированным системам. В «здоровых» проектах плотность ошибок «имеет тенденцию достигать стабильного значения при просмотре примерно 10 KSLOC. При дальнейшем просмотре кода, мы не должны наблюдать увеличения этого показателя» [3].

В объектно-ориентированных системах число ошибок также измеряют *на категорию классов или на класс*. При этом правило 80/20 считается приемлемым: 80 % выявленных ошибок в программе сосредоточено в 20 % классов системы [3].

В дополнение к этим более формальным подходам к накоплению получаемой при тестировании информации об ошибках, считается полезным устраивать «охоту за ошибками». В которой все желающие могут экспе-

риментировать с релизом в течение ограниченного промежутка времени, после чего награждается призом тот, кто обнаружил наибольшее количество ошибок, или тот, кто отыскал самую незаметную ошибку.

10. Методы оценки сложности и завершенности программного обеспечения. Число строк, попавших во фрагмент кода, абсолютно никак не связано с его завершенностью и сложностью. Этот подход устарел, в нем слишком просто играть с цифрами, что приводит к оценкам производительности, отличающимся друг от друга более чем на два порядка. Этот подход зависти от понятия строка программы. Считаются ли физические строки или точки с запятой? Как учесть несколько операторов на одной строке и операторы, которые занимают более одной строки? Традиционные меры подходят для первых поколений языков программирования, однако они не являются показателями завершенности и сложности объектно-ориентированной системы.

Прогресс объектно-ориентированной разработки измеряют *числом готовых и работающих классов* (логический аспект) или *количеством функционирующих модулей* (физический аспект). Другой мерой прогресса является *стабильность ключевых интерфейсов*. Сначала интерфейсы всех ключевых абстракций изменяются ежедневно, если не ежечасно. Через некоторое время стабилизируются наиболее важные из ключевых интерфейсов, следом - вторые по важности и т.д. К концу жизненного цикла разработки только несколько несущественных интерфейсов нуждаются в доработке. Безусловно, может потребоваться внести некоторые изменения в ключевые интерфейсы, но они обычно остаются совместимыми снизу вверх.

Предлагаются несколько мер, которые непосредственно применимы к объектно-ориентированным системам для оценки их сложности [3]:

- 1) взвешенная насыщенность класса методами;
- 2) глубина дерева наследования;
- 3) число потомков;
- 4) сцепление объектов;
- 5) отклик на класс;
- 6) недостаток связности в методах.

Взвешенная насыщенность класса дает относительную меру его сложности. Если считать, что все методы имеют одинаковую сложность, то это будет просто число методов в классе. Класс, который имеет большее количество методов среди классов одного с ним уровня, является более сложным.

Глубина дерева наследования и число потомков - количественные характеристики формы и размера структуры классов. Хорошо структурированная объектно-ориентированная система чаще бывает организована как лес классов, чем как одно высокое дерево. Советуется строить сбалан-

сированные по глубине и ширине структуры наследования: обычно - не глубже, чем 7 ± 2 уровня, и не шире, чем 7 ± 2 ветви.

Сцепление объектов - мера их взаимозависимости. Как и при традиционном программировании, следует стремиться проектировать независимые объекты, поскольку их можно использовать повторно.

Отклик на класс - количество методов, которые могут вызываться экземплярами класса.

Связность методов - мера насыщенности абстракции. Класс, который может вызывать больше методов, чем равные ему по уровню классы, является более сложным. Класс с низкой связностью является случайной или неподходящей абстракцией. Его нужно переабстрагировать в несколько классов, или передать его обязанности другим классам.

3.4. Преимущества и недостатки объектно-ориентированной разработки программного обеспечения

1. Преимущества. Объектная модель принципиально отличается от моделей, которые связаны с традиционными методами структурного анализа, проектирования и программирования. Это не означает, что объектная модель требует отказа от всех ранее найденных и испытанных временем методов и приемов. Скорее, она вносит некоторые новые элементы, которые добавляются к предшествующему опыту. Объектный подход обеспечивает ряд существенных удобств, которые другими моделями не предусматривались. Наиболее важно, что объектный подход позволяет создавать системы, которые удовлетворяют пяти признакам хорошо структурированных сложных систем. Кроме этого есть еще пять преимуществ объектной модели.

Во-первых, объектная модель позволяет в полной мере использовать выразительные возможности объектных и объектно-ориентированных языков программирования. Опыт показал, что при использовании таких языков, как Object Pascal, C++ и т.п. вне объектной модели, их наиболее сильные стороны либо игнорируются, либо применяются неправильно.

Во-вторых, использование объектного подхода существенно повышает уровень унификации разработки и пригодность для повторного использования не только программ, но и проектов, что в конце концов ведет к созданию среды разработки [3]. Объектно-ориентированные системы часто получаются более компактными, чем их не объектно-ориентированные эквиваленты. А это означает не только уменьшение объема кода программ, но и удешевление проекта за счет использования предыдущих разработок, что дает выигрыш в стоимости и времени.

В-третьих, использование объектной модели приводит к построению систем на основе стабильных промежуточных описаний, что упрощает

процесс внесения изменений. Это дает системе возможность развиваться постепенно и не приводит к полной ее переработке даже в случае существенных изменений исходных требований.

В-четвертых, объектная модель уменьшает риск разработки сложных систем, прежде всего потому, что процесс интеграции растягивается на все время разработки, а не превращается в единовременное событие. Объектный подход состоит из ряда хорошо продуманных этапов проектирования, что также уменьшает степень риска и повышает уверенность в правильности принимаемых решений.

Наконец, объектная модель ориентирована на человеческое восприятие мира. По словам Робсона, «многие люди, не имеющие понятия о том, как работает компьютер, находят вполне естественным объектно-ориентированный подход к системам» [3].

В настоящее время объектно-ориентированное проектирование - единственная методология, позволяющая справиться со сложностью, присущей очень большим системам. Однако следует заметить, что иногда применение OOD может оказаться нецелесообразным, например, из-за неподготовленности персонала или отсутствия подходящих средств разработки.

Кроме перечисленных, объектно-ориентированная технология имеет следующие достоинства:

- предсказуемость процесса разработки;
- сокращение времени на разработку;
- сокращение кода программы;
- создание более гибких, легко изменяемых систем;
- возможность разработки сложных систем, для которых нет альтернативного решения разработки.

2. Недостатки. Недостатки объектно-ориентированной технологии рассматривают в двух аспектах: производительность и начальные затраты.

В объектно-ориентированных языках, по сравнению с процедурными языками, возрастает время на пересылку сообщения от одного объекта другому. При вызове методов, которые не найдены и не связаны статически во время компиляции, выполняемая программа должна динамически искать нужный метод по классу объекта-получателя. Исследования показывают, что в худшем случае на вызов метода тратится в 1.75...2.5 раза больше времени, чем на обычный вызов процедуры. С другой стороны, наблюдения показывают, что при вызове методов динамический поиск действительно необходим примерно в 20 % случаев. Кроме того, компилятор часто может определять, какие вызовы могут быть связаны статически, и сгенерировать для них вызов процедуры вместо динамического поиска.

Другая причина снижения производительности кроется в том, что объектно-ориентированная технология порождает многоуровневые системы абстракций. Вследствие этого расслоения, каждый метод оказывается

очень маленьким, так как он строится на методах нижнего уровня. Другим следствием расслоения является то, что иногда методы служат лишь для получения доступа к защищенным атрибутам объекта. В результате происходит слишком много вызовов. Вызов метода на высшем уровне абстракции обычно влечет каскад других вызовов; методы верхних уровней вызывают методы нижних уровней и т.д. Для приложений, в которых время - ограниченный ресурс, недопустимо слишком большое количество вызовов методов. С другой стороны, такое слоение способствует пониманию системы. Рекомендуется сначала проектировать систему с желаемыми функциональными свойствами, а потом определять узкие места. Впоследствии методы можно вынести в подпрограмму, выигрывая тем самым время.

Потеря производительности происходит также из-за большого количества наследуемого кода. Класс, лежащий глубоко в структуре наследования, может иметь много суперклассов; при компоновке программы должно быть подгружено описание их всех. Для маленьких приложений это практически может означать, что нужно избегать глубоких иерархий классов, потому что они требуют чрезмерного количества объектного кода.

Еще один источник проблем для производительности объектно-ориентированных программ - их поведение в системе со страничной организацией памяти. Большинство компиляторов выделяет память сегментами, размещая каждый компилируемый модуль (обычно файл) в одном или более сегментах. Если программа работает слишком медленно из-за чрезмерно частого переключения страниц, можно попробовать изменить физическое расположение классов в модулях. Это проектное решение, касающееся физической модели системы, на логику программы оно не повлияет.

Еще одна составляющая риска - динамическое размещение и уничтожение объектов. Оно сопряжено с дополнительными вычислительными расходами по сравнению со статическим резервированием. В этом случае можно отказаться от динамического создания объектов и разместить их в памяти заранее или заменить стандартный алгоритм выделения памяти на специально приспособленный для нужд программы.

Требуются определенные начальные затраты на переход к объектно-ориентированной технологии. Объектная ориентация - это не просто язык программирования, который можно выучить на трехдневных курсах или по книжке. Требуется время, чтобы освоить объектно-ориентированное проектирование как новое мировоззрение. Мировоззрение, которое должно быть усвоено как разработчиками, так и менеджерами.

В заключении нужно отметить, что положительные свойства объектно-ориентированных систем, как правило, окупают перечисленные недостатки.