Учреждение образования
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Н. В. Пацей, Д. В. Шиман,
И. Г. Сухорукова

ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Рекомендовано
учебно-методическим объединением по образованию
в области информатики и радиоэлектроники
в качестве учебно-методического пособия по курсовому
проектированию для студентов высших учебных заведений
по направлению специальности «Информационные системы
и технологии (издательско-полиграфический комплекс)»

Минск 2011

УДК 004.415.2.041(075.8)
ББК 32.973.2я73
П21

Р е ц е н з е н т ы :
кафедра электронных вычислительных средств Белорусского государственного университета информатики и радиоэлектроники (доктор технических наук, профессор, заведующий кафедрой *А. А. Петровский*);
доктор технических наук, профессор кафедры программного обеспечения информационных технологий Белорусского государственного университета информатики и радиоэлектроники *В. Н. Ярмолик*

Пацей, Н. В.

П21 Технология разработки программного обеспечения : учеб.-метод. пособие по курсовому проектированию для студентов по направлению специальности «Информационные системы и технологии (издательско-полиграфический комплекс)» / Н. В. Пацей, Д. В. Шиман, И. Г. Сухорукова. – Минск : БГТУ, 2011. – 130 с.

ISBN 978-985-530-081-7.

В учебно-методическом пособии последовательно рассматриваются основные этапы разработки программного обеспечения: анализ, построение архитектуры, планирование, разработка и тестирование. Освещается современное состояние и излагаются существующие подходы к технологии разработки программных продуктов. Кроме того, отдельное внимание уделяется составу и оформлению пояснительной записки.

Предназначено для студентов специальности «Информационные системы и технологии (издательско-полиграфический комплекс)» при выполнении курсового проекта по дисциплине «Объектно-ориентированное программирование».

УДК 004.415.2.041(075.8)
ББК 32.973.2я73

ISBN 978-985-530-081-7

ПРЕДИСЛОВИЕ

Представленное учебно-методическое пособие может быть использовано студентами специальности «Информационные системы и технологии» при выполнении курсового проекта по дисциплине «Объектно-ориентированное программирование», при проведении учебных семинаров и самостоятельном изучении.

Цель курсового проекта – закрепление знаний, полученных при изучении дисциплины «Объектно-ориентированное программирование», а также приобретение практических навыков разработки программы средней сложности с использованием современных технологий и инструментальных средств. В ходе выполнения курсового проекта студент должен научиться анализировать требования к программному обеспечению (ПО), уметь представлять исходные данные и результат, проектировать архитектуру программной системы, создавать дизайн клиентских приложений, кодировать на выбранном языке программирования с использованием объектно-ориентированной (компонент-ориентированной) технологии и библиотек, выполнять тестирование и отладку программ, оформлять необходимую документацию.

Данное пособие не является полным и исчерпывающим справочником по каждому аспекту разработки архитектуры, проектирования и дизайна приложений. Оно представляет практический обзор наиболее важных тем и содержит ссылки на подробные руководства или материалы, в которых соответствующие вопросы проработаны более детально.

Для успешного освоения материала и понимания концепций и задач, изложенных в учебно-методическом пособии, требуются:

1) навыки объектно-ориентированного программирования;
2) умение компилировать и выполнять отладку приложений;
3) знание реляционных баз данных;
4) понимание основ универсального языка моделирования UML (Unifed Modeling Language) и навыков работы с инструментальной средой Rational Rose.

В качестве инструментальной среды проектирования в пособии используется Rational Rose [1], для описания моделей – язык UML [1–4]. В соответствии с дисциплиной в издании рассматриваются методики создания приложений для выполнения преимущественно на клиентских компьютерах.

ВВЕДЕНИЕ

В современном мире всеобщей компьютеризации и информатизации требования, предъявляемые к программному обеспечению вообще и к программным продуктам (ПП), программным средствам (ПС) и приложениям в частности, весьма высоки. В связи с этим обеспечение удовлетворяющих пользователя потребительских качеств программы, таких как надежность, быстродействие, соответствие заявленным возможностям, полнота документации, возможности расширения, развития и т. д., без строгого соблюдения определенной технологии практически невозможно.

В соответствии с обычным значением слова «технология» под *технологией программирования* (programming technology) будем понимать технологию разработки программного средства как совокупность абсолютно всех технологических процессов его создания – от момента зарождения идеи о данном ПС до составления необходимой программной документации [5]. Каждый процесс указанной совокупности базируется на использовании неких методов и средств.

Процесс разработки ПС и методы оценки продукта, процессов жизненного цикла (ЖЦ) стандартизованы (ISO/IEC 12207, ISO/IEC 15504, ISO 912, СТБ ИСО/МЭК 12207-2003 и др.) [6, 7].

Можно выделить три класса ПО: системное; пакеты прикладных программ (прикладное ПО); инструментарий технологии программирования (инструментальные средства для разработки ПО).

Системное ПО направлено на создание операционной среды функционирования других программ; обеспечение надежной и эффективной работы самого компьютера и вычислительной сети; проведение диагностики и профилактики аппаратуры компьютера и вычислительных сетей; выполнение вспомогательных технологических процессов (копирование, архивация, восстановление файлов программ, баз данных (БД) и т. п.).

Прикладное ПО служит программным инструментарием решения функциональных задач и является самым многочисленным классом ПО. В данный класс входят программные продукты, выполняющие обработку информации различных предметных областей. Таким образом, можно сказать, что прикладное ПО – комплекс взаимосвязанных программ для решения задач определенного класса предметной области.

Глава 1. ПОРЯДОК ВЫПОЛНЕНИЯ КУРСОВОГО ПРОЕКТА

Перед началом курсового проектирования студент получает индивидуальное задание в виде темы курсового проекта и бланка задания. Темы курсовых проектов выдает преподаватель. В исключительных случаях возможен самостоятельный выбор темы по согласованию с преподавателем.

В процессе выполнения задания студент должен:

- провести поиск в литературных и Интернет-источниках информации по теме проекта;
- выполнить обзор существующих решений задач по данному направлению, перечислить возможные пути решения;
- провести анализ ПС по выданному заданию;
- выполнить проектирование ПС;
- разработать ПС, согласно спроектированной архитектуре;
- провести тестирование ПС;
- оформить пояснительную записку к курсовому проекту, в которой описать (задокументировать) результаты работы по каждому из вышеперечисленных пунктов, разработать руководство пользователя для реализованного ПС.

По окончании работы студент представляет отчет (пояснительную записку) и ПС, записанное на электронный носитель, с необходимыми для работы компонентами на проверку руководителю курсового проекта. После предварительной проверки студент допускается (или не допускается) к защите. На защите он докладывает преподавателю или комиссии о проделанной работе и демонстрирует разработанное ПС, отвечает на вопросы, поясняет и обосновывает сущность своих решений. По результатам защиты выставляется оценка.

Примерные темы курсового проекта

1. Автоматизация кадрового учета для полиграфического предприятия.
2. Автоматизация учета услуг и клиентов издательства.

3. ПС «Типография».
4. ПС «Управление ресурсами полиграфического предприятия».
5. ПС «Бухгалтерия типографии».
6. ПС «Рекламное агентство».
7. ПС «Расчет стоимости допечатных и послепечатных работ».
8. ПС «Аналитический модуль издательства».
9. ПС «Типография. Заказ печатной продукции».
10. ПС «Анализ эффективности работы подразделений полиграфического предприятия».
11. ПС «Калькулятор себестоимости издания».
12. ПС «Менеджер типографии».
13. ПС «Управление рекламным производством».
14. ПС «Управление упаковочным производством».
15. ПС «Склад издательства».
16. ПС «Органайзер».

Глава 2. МЕТОДОЛОГИИ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Существуют понятия технологии и методологии программирования. Несмотря на то, что в обоих случаях изучаются соответствующие методы, в технологии программирования методы рассматриваются «сверху», т. е. с точки зрения организации технологических процессов, а в методологии программирования – «снизу», т. е. с точки зрения основ их построения.

2.1. Rational Unified Process

Rational Unified Process (RUP) – рациональный унифицированный процесс, созданный в 1996 г. корпорацией Rational при участии Гради Буча, Айвара Якобсона и Джима Румбаха [8].

Продукты, поддерживающие методику RUP: Rose (используется практически на всех этапах разработки), SoDA (автодокументирование), RequisitePro (управление требованиями), ClearQuest (запросы на изменения), ClearCase (версионность), Administrator (управление репозиторием проекта), WorkBrench (настройка корпоративных процессов), Quantify (тестирование скорости кода), Purify (определение утечек памяти), PureCoverage (тест охвата кода), Robot (автоматизированное тестирование), SiteLoad (нагрузочное тестирование), SiteCheck (проверка «мертвых» Web-ссылок). В настоящее время доступен RUP, интегрированный в среду разработки Microsoft .NET (под названием Rational XDE).

Основная идея RUP – максимально четко распределить работу каждого участника процесса разработки. При разработке ПС проектировщики заранее продумывают все основные вопросы. При этом они не пишут программный код, поскольку не создают программный продукт, а только разрабатывают его дизайн, например, на UML. Как только проектный план готов, его можно использовать при создании системы. Поскольку проектировщики работают на некотором уровне абстракции, им удается избежать принятия ряда тактических решений, ведущих к энтропии программного продукта. Программисты руководствуются проектным планом и создают качественно выстроенную систему [8, 9].

2.2. Extreme Programming

Extreme Programming (XP) – экстремальное программирование. Идеологами методики можно считать Кента Бека, Уорда Каннингема и Рона Джеффриса.

Основные принципы: тесная коммуникация, постоянное тестирование, минимум документации и максимум гибкости [10, 11].

Понятно, что отрицание этапа «большого предварительного проектирования» допустимо только при разработке тривиальных систем (простейшие однопользовательские программы с минимумом бизнес-логики и т. п.), так как в середине проекта может обнаружиться прецедент, заставляющий переделать большую часть кода.

В отличие от RUP (или любой другой методики с иксированием результатов этапов) XP дает ощущение неуверенности в начале проекта. В середине архитектура стабилизируется, хотя получается просто набор из специфических решений, каждое из которых затрудняет дальнейшие изменения в программном коде. Конец же проекта характеризуется уверенностью в выполненной разработке. Инструментальных средств, поддерживающих методику XP, не существует (это может быть текстовый редактор или средство обеспечения версионности).

В методологии XP очень популярны два лозунга: «Do the Simplest Thing that Could Possibly Work» («Ищите самое простое решение, которое может сработать») и YAGNI («You Aren't Going to Need It» – «Это вам не понадобится»). Оба они олицетворяют собой одну из практик XP под названием «Простой дизайн» [12]. XP вообще не признает работы над архитектурой. Суть XP – сразу садиться за написание кода и надеяться на то, что рефакторинг решит все проблемы с проектированием.

Основная область применения XP – небольшие проекты с постоянно изменяющимися требованиями. Заказчик ПС может не иметь точного представления о том, что должно быть сделано. Поэтому функциональность разрабатываемого продукта может изменяться каждые несколько месяцев или чаще. Именно в этих случаях XP позволяет достичь максимального успеха.

2.3. Structured Analysis and Design Technique

Structured Analysis and Design Technique (SADT) – методология структурного анализа и проектирования. Известна как разработка компании SofTech, либо как только функциональный

вариант в версии IDEF0. Ее начали применять с 1973 г. во многих областях, таких как бизнес, производство, оборона, связь и организация проектирования.

SADT в полной мере реализует идею «большого предварительного проектирования в начале разработки» [13]. Из-за жесткой декомпозиции процесса разработки методику можно считать прародителем RUP.

Диаграммы в стандарте IDEF0 представляют собой модули системы в виде блоков с набором входов и выходов и набором управляющих воздействий на них (рис. 2.1).

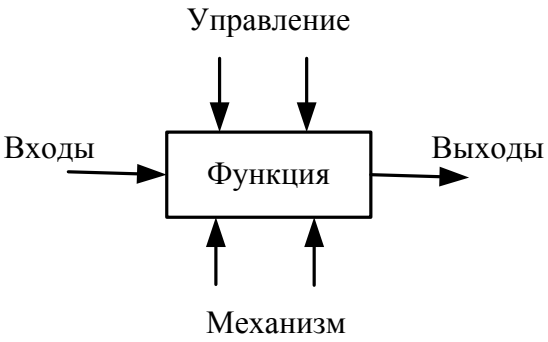


Рис. 2.1. Функциональный блок и интерфейсные дуги

Каждый компонент модели может быть декомпозирован на другой диаграмме. Каждый блок детальной диаграммы представляет собой подфункцию, границы которой определены интерфейсными дугами. Каждый из блоков детальной диаграммы может быть также детализирован на следующей в иерархии диаграмме. На каждом шаге декомпозиции более общая диаграмма называется родительской для более детальной диаграммы.

Структурный подход к проектированию программных продуктов предполагает разработку следующих моделей [14, 15]:

- ✔ диаграмм потоков данных IDEF3 (DFD – Data Flow Diagrams), описывающих взаимодействие источников и потребителей информации через процессы, которые должны быть реализованы в системе;
- ✔ диаграмм «сущность – связь» IDEF1 (ERD – Entity-Relationship Diagrams), описывающих базы данных разрабатываемой системы;
- ✔ диаграмм переходов состояний IDEF2 (STD – State Transition Diagrams), характеризующих поведение системы во времени;
- ✔ функциональных диаграмм;
- ✔ спецификаций процессов;
- ✔ словаря терминов.

Все диаграммы связывают друг с другом иерархической нумерацией блоков: первый уровень – A0 (рис. 2.2), второй – Al, A2 и т. д. (рис. 2.3), третий – A11, A12, A13 и т. п., где первая цифра означает номер родительского блока, а последняя – номер конкретного блока детальной диаграммы.

Рис. 2.2. Функциональная диаграмма первого уровня

Диаграмма, представленная на рис. 2.3, детализирует функции программы. На ней показаны три блока: *Меню*, *Сортировка выбранным методом*, *Вывод результата*. Для каждого блока определены исходные данные, управляющие воздействия и результаты.

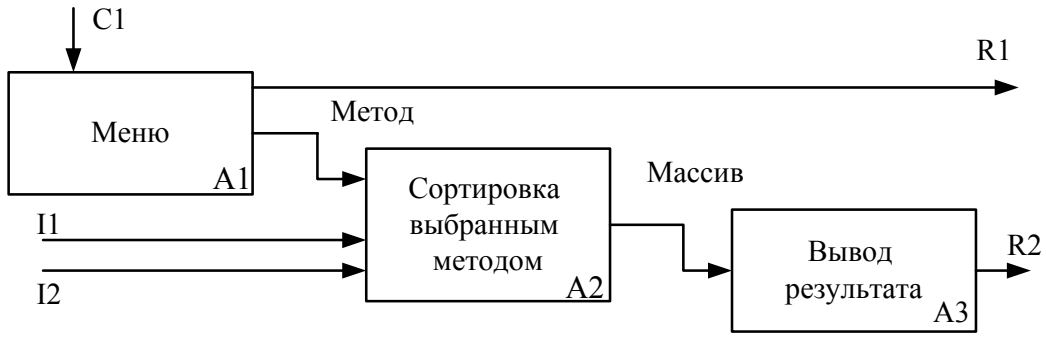


Рис. 2.3. Функциональная уточняющая диаграмма
для программы сортировки массива

На детализирующей диаграмме используются следующие обозначения: I1 – размер массива; I2 – массив; C1 – выбор метода; R1 – вывод описания метода; R2 – отсортированный массив.

Методика серьезно проигрывает остальным по охвату этапов жизненного цикла ПО, сконцентрировавшись только на сборе требований и бизнес-моделировании.

2.4. Microsoft Solutions Framework & Microsoft Operations Framework

Microsoft Solutions Framework (MSF) – набор решений от Microsoft (MS) и Microsoft Operations Framework (MOF) – набор операций от MS.

Ядро этой системы составляют шесть основных моделей [17]:

1) модель производственной архитектуры предлагает набор принципов, обеспечивающих быстрое создание производственной архитектуры посредством выпуска версий;

2) модель проектной группы описывает роли, обязанности каждого участника, распределение ответственности и порядок работы;

3) модель процесса разработки ПС описывает фазы, этапы, виды деятельности и результаты процесса разработки приложения и их связь с моделью проектной группы MSF;

4) модель управления рисками описывает порядок и условия реализации упреждающих решений и мер для постоянного выявления потенциальных проблем, позволяет обнаружить наиболее существенные риски и реализовать стратегии их устранения;

5) модель процесса проектирования описывает концептуальное, логическое и физическое проектирование, реализует точку зрения на проект трех аудиторий: конечных пользователей, проектной группы и разработчиков;

6) модель приложения реализует логичный, трехуровневый, ориентированный на сервисы метод проектирования и разработки программного обеспечения, применяет пользовательские сервисы, бизнес-сервисы и сервисы данных.

Модель производственной архитектуры MSF является структурной и состоит из четырех элементов (перспектив): бизнеса, приложения, информации и технологии.

Архитектура приложения представляет собой концептуальное описание структуры программного продукта. Как показано на рис. 2.4, каждое приложение разделено на три уровня: пользовательский, прикладной и уровень данных. Кроме того, во всех приложениях имеется презентационный код, код обработки бизнес-правил и данных, а также код, отвечающий за хранение информации.

Пользовательский уровень отображает данные и позволяет пользователю редактировать их. Существуют два основных типа

интерфейса: «родной» (реализуемый средствами подсистемы пользовательского интерфейса операционной системы; например, в Microsoft Windows применяются API Win32 и элементы управления Windows) и на основе Web (HTML или XML), в результате они могут отображаться любым обозревателем на любой платформе.

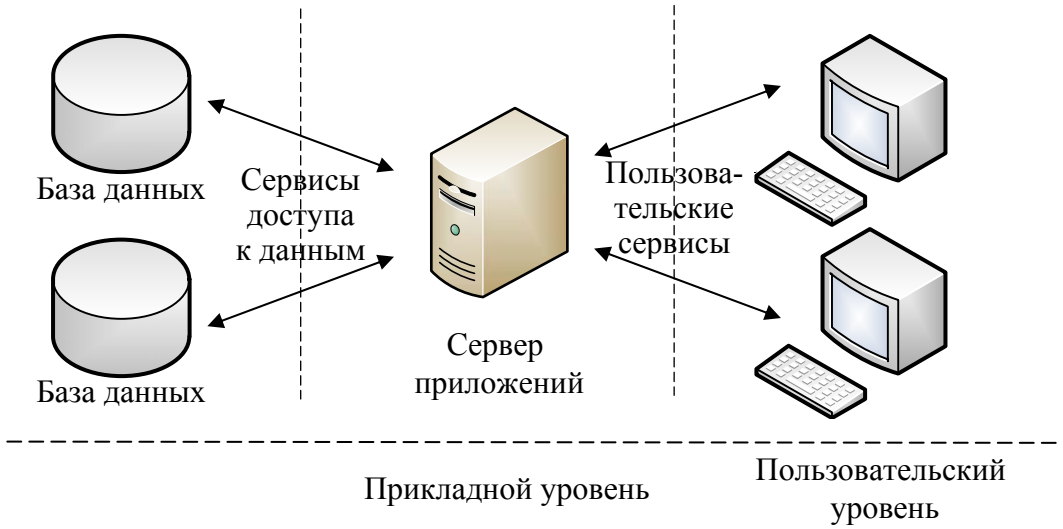


Рис. 2.4. Физическая модель и ее уровни

На прикладном уровне реализованы бизнес-правила и ограничения на данные. И хотя его сервисы используются презентационным уровнем, он не привязан к какому-либо клиенту – сервисы прикладного уровня доступны любому клиенту. Бизнес-правила выражаются в форме прикладных алгоритмов, корпоративных правил и т. д. Прикладной уровень не знает, как и где хранится обрабатываемая им информация. В этом вопросе он полагается на сервисы доступа к данным, выполняющие всю работу по получению и передаче данных. Сервисы доступа к данным также реализуются в виде изолированных модулей, «знающих» о месте хранения информации. Таким образом, если хранилище перемещено или изменен его формат, потребуется обновить только сервисы доступа к данным. Для многоуровневых приложений в качестве хранилища информации подходят простые системы управления базами данных (СУБД), необходимые для обслуживания данных в таблицах и быстрой выборки информации (например, с помощью индексов) и хранилища данных (в частности, SQL Server, Exchange Server, In Memory Database (IMDB) и Microsoft Access).

2.5. Rapid Application Development

В последнее время широкое распространение получила методология быстрой разработки приложений – Rapid Application Development (RAD). Жизненный цикл ПО по методологии RAD состоит из четырех фаз:

- анализа и планирования требований;
- проектирования;
- реализации;
- внедрения.

RAD хороша в первую очередь для относительно небольших проектов, разрабатываемых для конкретного заказчика, и неприменима для построения сложных расчетных программ, операционных систем или систем управления, т. е. программ, требующих написания большого объема (сотни тысяч строк) уникального кода.

Основные принципы методологии RAD:

1) итерационная разработка приложений;

2) необязательность полного завершения работ на каждом из этапов жизненного цикла (ЖЦ);

3) применение CASE-средств, обеспечивающих целостность данных;

4) участие конечных пользователей в процессе разработки систем;

5) разработка прототипов;

6) тестирование, производимое параллельно с разработкой;

7) разработка подсистем несколькими, немногочисленными, хорошо управляемыми командами;

8) четкое планирование и контроль выполнения работ.

2.6. Personal Software Process

Современные методики предусматривают также рекомендации для индивидуальных разработчиков – персональный процесс разработки программного обеспечения – Personal Software Process (PSP) [16]. Методика PSP предлагает семь этапов, проходя которые разработчик может улучшить собственный процесс. Software Engineering Institute (SEI) разработал набор из 76 форм, стандартов и инструкций. Одним из основных принципов является непрерывность внедрения и контроля за результатами внедрения этого

подхода, постоянное развитие и переход от одной рекомендации по повышению качества кода к другой.

В целом при работе по методике PSP программист проходит несколько стадий. Человек не осознает свои возможности, которые чаще всего бывают существенно ниже необходимого уровня. После первых отрицательных результатов человек начинает осознавать собственный уровень, т. е. становится сознательно некомпетентным. После долгих тренировок, решения уже практических задач человек начинает осознавать, в какой области он обладает знаниями, а в какой нет, т. е. становится сознательно компетентным. Через некоторое время человек не задумывается над тем, как и почему он принял то или иное решение. Это уровень бессознательной компетентности.

Этап PSP 0. На данном этапе программист учится составлять общий план проекта, оценивать время на каждый небольшой этап разработки, учится больше внимания обращать на тестирование разработанных модулей. При этом ведется непрерывное документирование всех действий – обнаруженных ошибок, времени, требующегося на разработку модулей, количества строк, написанных при работе с данным модулем. Этап PSP 0 предназначен для того, чтобы приучить программиста документировать результаты своей работы и составлять список работ, необходимых для реализации проекта.

Этап PSP 0.1. На данном этапе разработчик учится измерять объем работ – строк кода программы в целом и ее отдельных модулей, страниц документации и т. д. Для каждой выполняемой задачи программист описывает объем работ, который ему пришлось выполнить для ее реализации, и время, которое он потратил на выполнение этого объема работ.

Этап PSP 1. На данном этапе разработчик учится оценивать размер будущей программы. При этом следует различать новый код, повторно используемый код, базовый код прежней версии.

Этап PSP 1.1. На данном этапе программист на основе полученных количественных данных учится оценивать рабочее время, которое потребуется для выполнения некоторого объема работ. Результатом является умение составлять календарный план работ. Календарный план составляется в следующем порядке: оценка требований продукта, основные модули продукта, оценка объема модулей, оценка времени, требующегося на реализацию модулей, составление графика работ. В дальнейшем разработчик ведет

график реального выполнения работ, отслеживая отклонения запланированного графика от реального. Методика PSP не одобряет изменения плана работ, так как он может служить источником новых ошибок.

Этап PSP 1.2. Данный этап учит разработчика управлять качеством своего кода. Он пытается предсказать количество ошибок, которые могут появиться в его коде. В ходе разработки программист сверяется с плановыми числами и при нахождении расхождений пытается выявить их причину.

Этап PSP 2. Здесь программист учится качественно проектировать программу в целом. На предыдущих этапах при заданных требованиях следовало разделить программу на блоки и оценить их с точки зрения реализации. На основе имеющихся знаний разработчик получает возможность анализировать требования к программе, создавать спецификации проекта, выполнять высокоуровневое проектирование. Здесь программист приучает себя вначале писать спецификации на модуль, а лишь затем начинать его реализацию.

Этап PSP 3. Данный этап посвящен совершенствованию персонального процесса разработки. Программист ставит себе некоторую цель по повышению своей производительности или улучшению качества кода. Далее определяются методы достижения этой цели, проводятся замеры характеристик работы, полученные характеристики анализируются, и на их основе разработчик вырабатывает рекомендации по улучшению.

Менеджеры, которые возглавляют команду сотрудников, работающих в соответствии с PSP, должны производить постоянный контроль за качеством работы, а не только сверять их работу с календарным планом и констатировать количество ошибок постфактум.

2.7. Инструментарий технологии программирования

Инструментарий технологии программирования – это программные продукты, предназначенные для поддержки технологии программирования (рис. 2.5) [17].

Средства для создания приложений – совокупность языков и систем программирования, инструментальные среды пользователя,

а также различные программные компоненты для отладки и поддержки создаваемых программ.

Инструментальная среда пользователя – это специальные средства, встроенные в пакеты прикладных программ, такие как библиотека функций, процедур, объектов и методов обработки; макрокоманды; клавишные макросы; языковые макросы; конструкторы экранных форм и объектов; генераторы приложений; языки запросов высокого уровня; конструкторы меню и др.

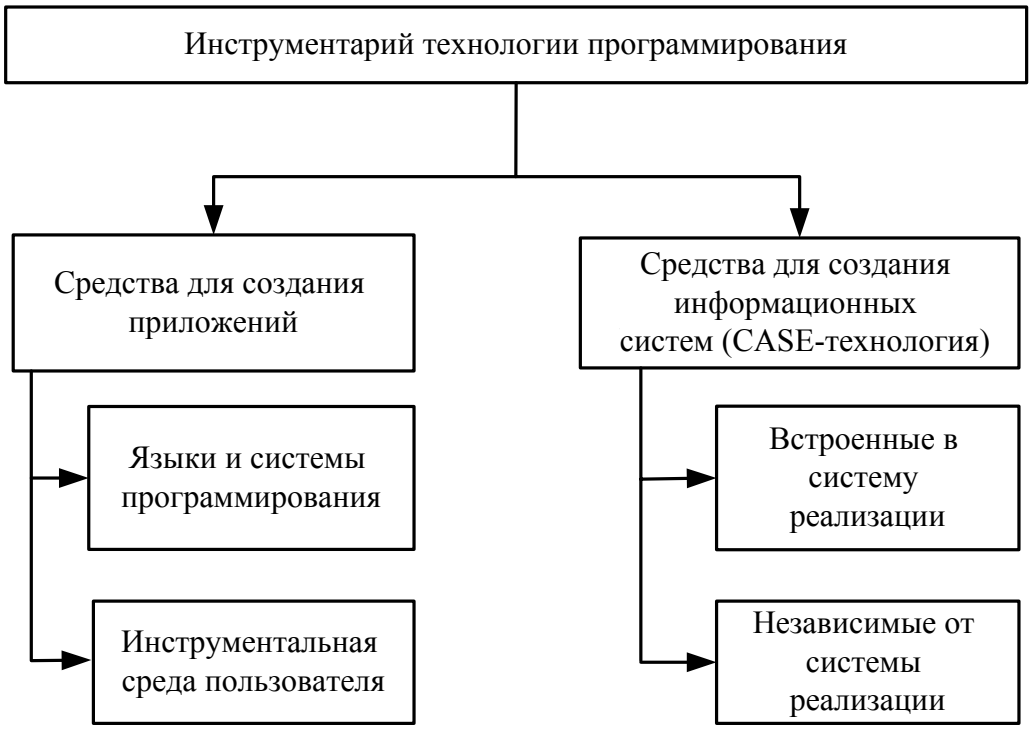


Рис. 2.5. Инструментарий технологии программирования

Интегрированные среды разработки программ объединяют набор средств для их комплексного применения на технологических этапах создания программы.

Одним из современных средств разработки информационных систем (ИС) является CASE-технология (CASE – Computer-Aided System Engineering) – программный комплекс, автоматизирующий весь технологический процесс анализа, проектирования, разработки и сопровождения сложных программных систем. В настоящее время CASE-технологии используются не только для производства ПС, но и как мощный инструмент решения исследовательских и проектных задач [15]. Хороший инструмент для рисования

диаграмм, сбора требований, построения моделей предметной области значительно облегчает работу системного аналитика.

Средства CASE-технологий делятся (рис. 2.5) на встроенные в систему реализации (все решения по проектированию и реализации привязки к выбранной СУБД); независимые от системы реализации (все решения по проектированию ориентированы на унификацию начальных этапов жизненного цикла программы и средств их документирования, обеспечивают большую гибкость в выборе средств реализации). В некоторых CASE-системах поддерживается кодогенерация программ – создание каркаса программ и создание полного продукта.

Существует мнение, согласно которому успех методики RUP в значительной степени обусловлен наличием качественных инструментов компании Rational, поддерживающих все этапы ЖЦ продукта [9].

Средства анализа и проектирования. Данные CASE-средства применяются для проектирования и создания спецификаций программной системы, поддерживают SE (Software Engineering) и IE (Information Engineering):

- CASE-аналитик (Эйтекс);
- POSE (Computer Systems Advisers);
- Design/IDEF (Meta Software);
- BPWin (Logic Works);
- SELECT (Select Software Tools);
- CASE/4/0 (micro TOOl GmbH) и ряд других средств.

Средства проектирования баз данных и файлов. Средства данной группы служат для логического моделирования данных, автоматического преобразования моделей в третью нормальную форму, автоматическую генерацию схем баз данных и описаний форматов файлов на уровне программного кода:

1) ERWin (Logic Works);
2) S-Designor (SPD);
3) Designtr/2000 (Oracle);
4) Sillverrun (Computer Systems Advisers).

Средства коллективной разработки ПО. Основное достоинство CASE-технологии – это поддержка коллективной работы над проектом за счет возможности работы в локальной сети разработчиков, экспорта (импорта) любых фрагментов проекта, организованного управления проектами. Рассмотрим наиболее применяемые пакеты прикладных программ для коллективной разработки ПО.

Microsoft Visual SourceSafe (VSS – Visual SourceSafe) – программный продукт компании Microsoft, файл-серверная система управления версиями, предназначенная для небольших команд разработчиков. VSS позволяет хранить в общем хранилище файлы, разделяемые несколькими пользователями, для каждого файла хранится история версий. VSS входит в состав пакета Microsoft Visual Studio и интегрирован с продуктами этого пакета. Там, где VSS недостаточно, ему на замену предлагается Visual Studio Team Foundation Server, входящий в состав Visual Studio Team System.

Subversion – свободно распространяемая система управления версиями с открытым кодом. Subversion часто называют «svn» – по названию клиентской программы, входящей в ее дистрибутив. Subversion – централизованная система. Данные хранятся в едином хранилище. При сохранении новых версий используется дельта-компрессия, т. е. система находит отличия новой версии от предыдущей и записывает только их, избегая ненужного дублирования данных. Хранилище может располагаться на локальном диске или на сетевом сервере.

2.8. Задание

Установите необходимые инструментальные CASE-средства: Rational Rose или MS Visio, MS VS 2010 или MS VS 2008.

Глава 3. МОДЕЛИ ЖИЗНЕННОГО ЦИКЛА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

В соответствии со стандартом СТБ ИСО/МЭК 12207-2003 под жизненным циклом (ЖЦ) программного средства или системы подразумевается совокупность процессов, работ и задач, включающая в себя разработку, эксплуатацию и сопровождение ПС или системы, охватывающая их жизнь от формулирования концепции до прекращения использования [18].

Согласно данному стандарту, жизненный цикл программных средств состоит из процессов. Каждый процесс ЖЦ разделен на набор работ. Каждая работа разделена на набор задач, а весь процесс разработки содержит тринадцать работ:

1) подготовка процесса разработки;
2) анализ требований к системе;
3) проектирование системной архитектуры;
4) анализ требований к программным средствам;
5) проектирование программной архитектуры;
6) техническое проектирование программных средств;
7) программирование и тестирование программных средств;
8) сборка программных средств;
9) квалификационные испытания программных средств;
10) сборка системы;
11) квалификационные испытания системы;
12) ввод в действие программных средств;
13) обеспечение приемки программных средств.

Существуют также еще международные нормативные документы, регламентирующие ЖЦ ПО. Например, стандарт ISO/IEC 12207 (ISO – International Standards Organization – Международная организация по стандартизации, IEC – International Electrotechnical Commission – Международная комиссия по электротехнике) определяет структуру ЖЦ, содержащую процессы, действия и задачи, которые должны быть выполнены во время создания ПО [19].

Исторически в ходе развития теории проектирования программного обеспечения и по мере его усложнения утвердились четыре основные модели ЖЦ.

3.1. Каскадная модель

Первой по времени появления и самой распространенной явилась каскадная модель (рис. 3.1).

Каскадная стратегия представляет собой однократный проход этапов разработки. Она характеризуется следующими основными особенностями:

- последовательным выполнением входящих в ее состав этапов;
- окончанием каждого предыдущего этапа до начала последующего;
- отсутствием временного перекрытия этапов (последующий этап не начнется, пока не завершится предыдущий);
- отсутствием (или определенным ограничением) возврата к предыдущим этапам;
- наличием результата только в конце разработки.

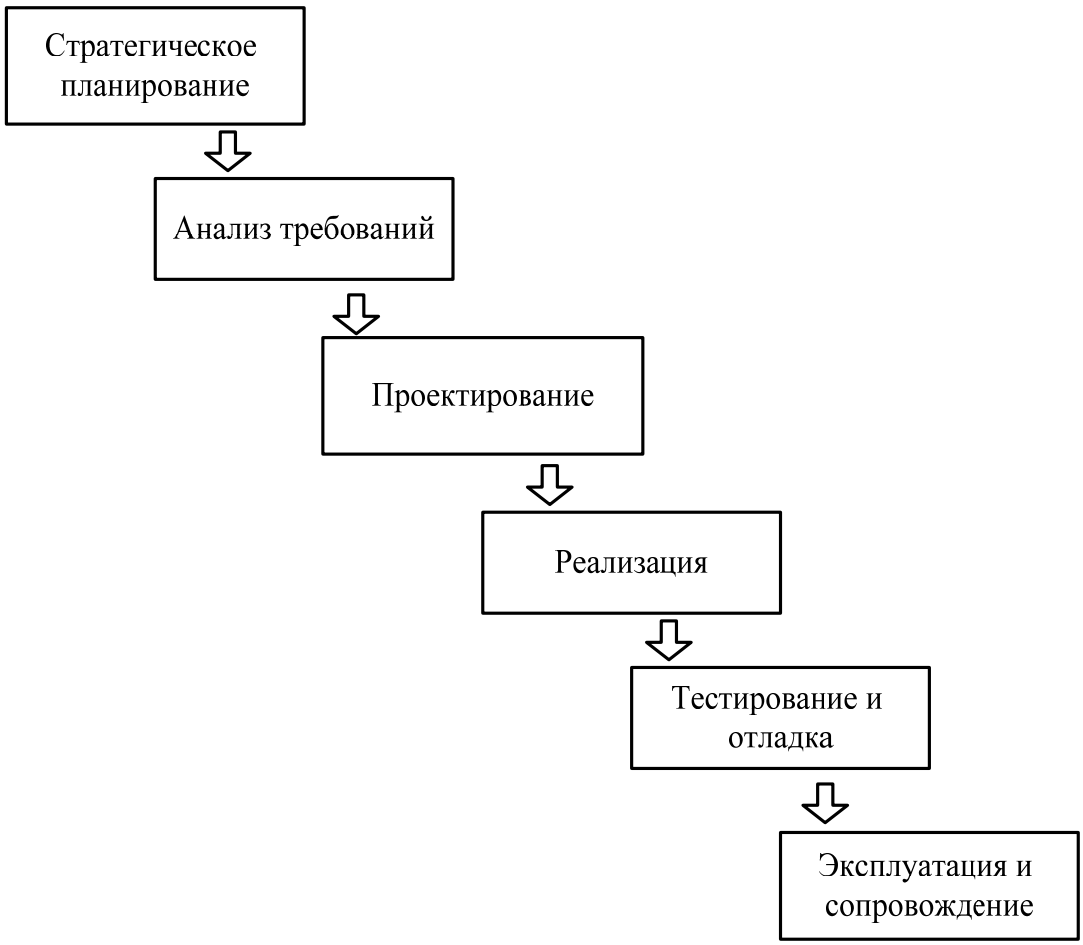


Рис. 3.1. Каскадная модель жизненного цикла ПО

Основными представителями моделей, реализующих каскадную стратегию, являются каскадная и V-образная модели.

Главными достоинствами каскадной стратегии являются: стабильность требований в течение всего жизненного цикла разработки; простота применения стратегии, необходимость только одного прохода этапов разработки; простота планирования, контроля и управления проектом; возможность достижения высоких требований к качеству проекта в случае отсутствия жестких ограничений затрат и графика работ; доступность для понимания заказчиками.

К недостаткам каскадной стратегии следует отнести: сложность четкого формулирования требований в начале жизненного цикла ПС и невозможность их динамического изменения на протяжении ЖЦ; последовательность линейной структуры процесса разработки; на практике разрабатываемые ПС или система обычно слишком велики, чтобы все работы по их созданию выполнить однократно; в результате возврат к предыдущим шагам для решения возникающих проблем приводит к увеличению затрат и нарушению графика работ; проблемность финансирования проекта, связанная со сложностью единовременного распределения денежных средств; непригодность промежуточного продукта для использования; недостаточное участие пользователя в разработке системы или ПС – только в самом начале (при разработке требований) и в конце (во время приемочных испытаний), что приводит к невозможности предварительной оценки пользователем качества ПС или системы.

Области применения каскадной стратегии ограничены недостатками. Ее использование наиболее эффективно в следующих случаях:

1) при разработке проектов с четкими, неизменяемыми в течение ЖЦ требованиями, понятными реализацией и техническими методиками;

2) при разработке проекта, ориентированного на построение системы или продукта такого же типа, как уже разрабатывались программистами ранее;

3) при разработке проекта, связанного с созданием и выпуском новой версии уже существующего продукта или системы;

4) при разработке проекта, связанного с переносом уже существующего продукта на новую платформу;

5) при выполнении больших проектов, в которых задействовано несколько больших команд разработчиков.

3.2. Итерационная модель

Следующей стадией развития теории проектирования ПО стала итерационная (инкрементная) модель ЖЦ, или так называемая поэтапная модель с промежуточным контролем (рис. 3.2). Основной ее особенностью является наличие обратных связей между этапами, вследствие этого появляется возможность проведения проверок и корректировок проектируемой системы на каждой стадии разработки. В результате трудоемкость отладки по сравнению с каскадной моделью существенно снижается.

Данная стратегия основана на полном определении всех требований к разрабатываемому ПС в начале процесса разработки. Однако полный набор требований реализуется постепенно в соответствии с планом в последовательных циклах разработки. Результат каждого цикла называется инкрементом. Первый инкремент реализует базовые функции ПС. В последующих инкрементах функции ПС постепенно расширяются, пока не будет реализован весь набор требований к ПС. Различия между инкрементами соседних циклов в ходе разработки постепенно уменьшаются.

Представителем модели, реализующей итерационную стратегию, является, например, RAD-модель. Следует отметить, что RAD-модель может использоваться как при итерационной, так и при эволюционной стратегиях разработки. Современной реализацией итерационной стратегии является XP.

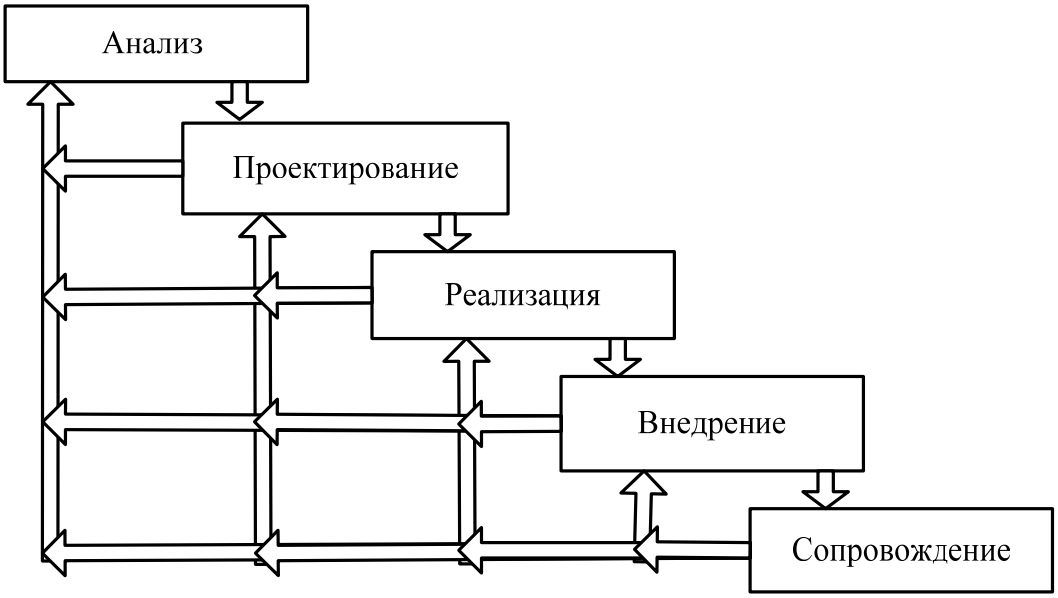


Рис. 3.2. Итерационная модель жизненного цикла ПО

3.3. Спиральная модель

Третья модель ЖЦ ПО – спиральная (spiral) модель (рис. 3.3) – поддерживает итерации поэтапной модели, но особое внимание уделяется начальным этапам проектирования: анализу требований, проектированию спецификаций, предварительному проектированию и детальному проектированию. Каждый виток спирали соответствует поэтапной модели создания фрагмента или версии ПО, уточняются цели и требования к программному обеспечению, оценивается качество разработанного фрагмента или версии и планируются работы следующей стадии разработки (витка). Таким образом, углубляются и конкретизируются все детали проектируемого ПО.

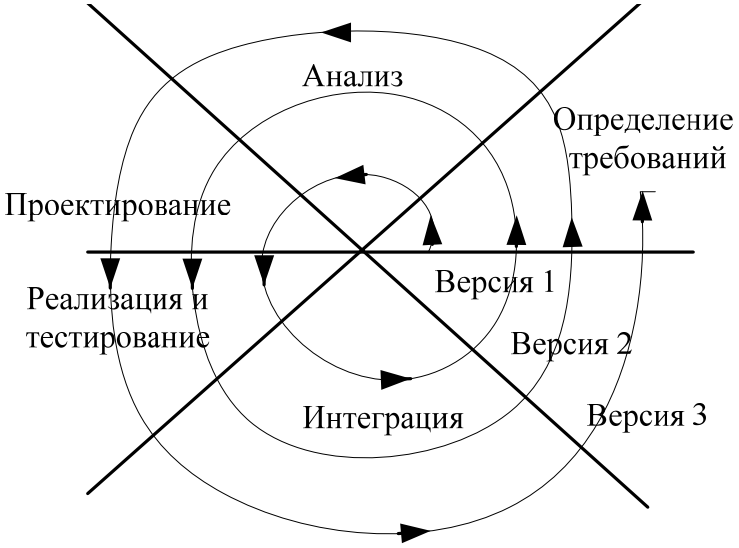


Рис. 3.3. Спиральная модель жизненного цикла ПО

Спиральная модель – классический пример эволюционной стратегии конструирования.

Достоинства спиральной модели: наиболее реально (в виде эволюции) отображает разработку ПО; позволяет явно учитывать риск на каждом витке эволюции разработки; включает шаг системного подхода в итерационную структуру разработки; использует моделирование для уменьшения риска и совершенствования программного изделия.

К недостаткам спиральной модели относятся: повышенные требования к заказчику; трудности контроля и управления временем разработки. Также основная проблема спирального цикла – определение

момента перехода на следующий этап. Для ее решения необходимо ввести временные ограничения на каждый из этапов жизненного цикла. Переход осуществляется в соответствии с планом, даже если не вся запланированная работа закончена. План составляется на основе статистических данных, полученных в предыдущих проектах, и личного опыта разработчиков.

3.4. Компонентно-ориентированная модель

Компонентно-ориентированная модель является развитием спиральной модели и тоже основывается на эволюционной стратегии конструирования. В этой модели конкретизируется содержание квадранта конструирования – оно отражает тот факт, что в современных условиях новая разработка должна основываться на повторном использовании существующих программных компонентов (рис. 3.4).

Программные компоненты, созданные в реализованных программных проектах, хранятся в библиотеке. В новом программном проекте, исходя из требований заказчика, выявляются кандидаты в компоненты. Далее проверяется наличие этих кандидатов в библиотеке. Если они найдены, то компоненты извлекаются из библиотеки и используются повторно. В противном случае создаются новые компоненты, они применяются в проекте и включаются в библиотеку.

Достоинствами компонентно-ориентированной модели являются: уменьшение на 30% времени разработки программного продукта; снижение стоимости программной разработки до 70%; увеличение в 1,5 раза производительности разработки.

Фирма Rational Software предложила свою модель ЖЦ, которая называется Rational Objectory Process (ROP). ROP – итеративный процесс, в течение которого происходит последовательное уточнение результатов. ROP разбит на циклы, каждый из которых в свою очередь состоит из четырех фаз:

- начальная стадия (Inception);
- разработка (Elaboration);
- конструирование (Construction);
- ввод в эксплуатацию (Transition).

Результатом работы каждого такого цикла является своя версия программной системы. Каждая стадия завершается в четко определенной контрольной точке (milestone).

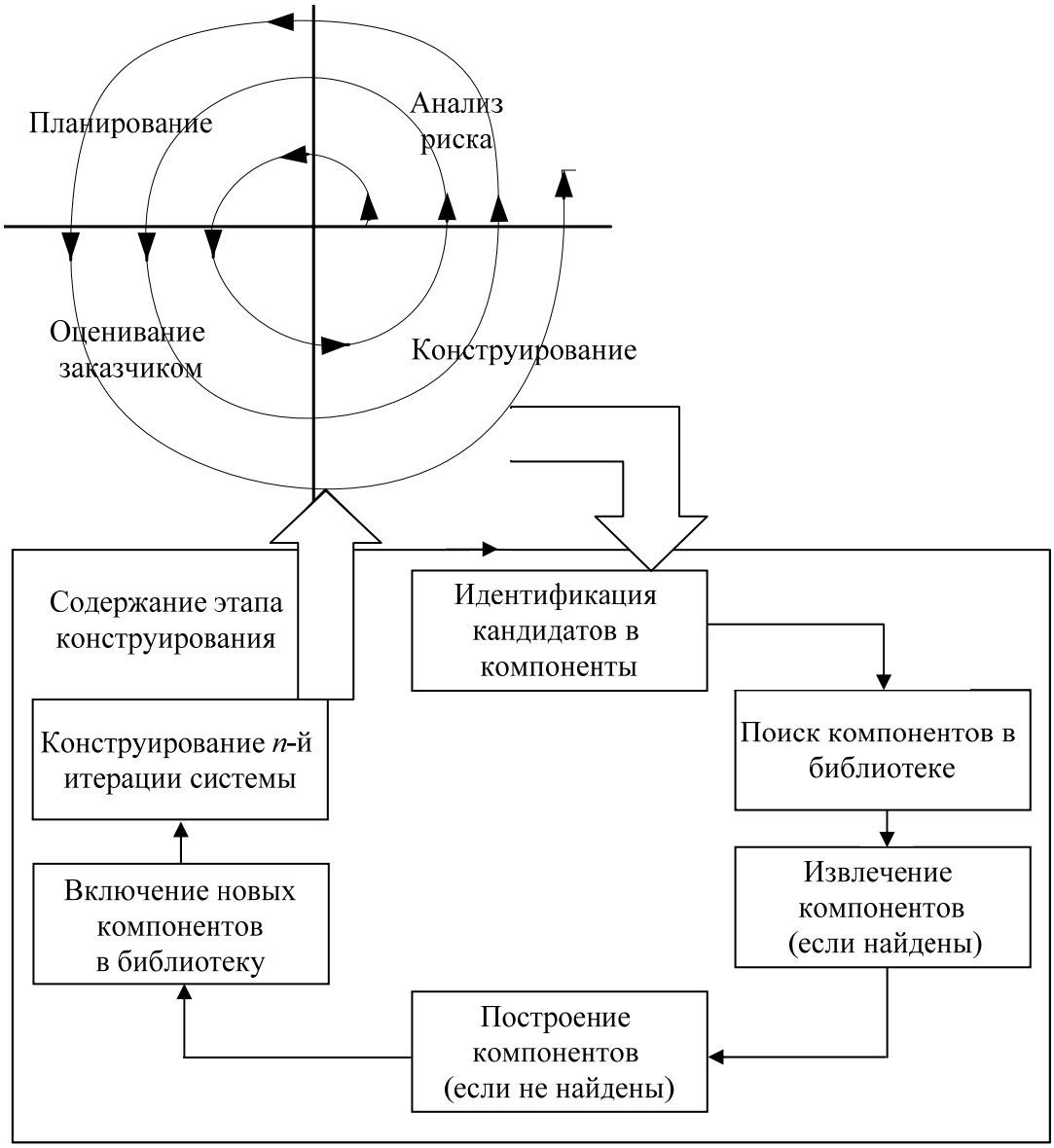


Рис. 3.4. Компонентно-ориентированная модель жизненного цикла ПО

Для выбора приемлемой модели жизненного цикла разработки программных средств и систем рекомендуется использовать специальную процедуру, разработанную в Институте качества программного обеспечения (SQI). Данная процедура базируется на применении таблиц вопросов. Затем необходимо ответить на вопросы и расположить по степени важности категории. Последний этап – выбрать ту модель, которая соответствует столбцу с наибольшим количеством отмеченных ответов с учетом их степени важности. Выбранная модель жизненного цикла является наиболее приемлемой для анализируемого проекта.

В данном курсовом проекте исходите из следующих временных соотношений между отдельными видами работ, согласно методологии RUP (таблица).

Распределение времени по этапам разработки ПО (в процентах к общему времени разработки)

Виды работ	Этапы разработки ПС				Всего
	анализ	проектирование	кодирование	отладка и тестирование	
Анализ требований и разработка спецификаций	13				13
Подготовка данных для отладки		2	2	4	8
Планирование отладки	2		2	4	8
Проектирование		13			13
Тестирование	5	5	4	11	25
Кодирование			8		8
Испытание ПО				17	17
Документирование			4	4	8
Всего	20	20	20	40	100

Как видно из таблицы, наиболее трудоемкие этапы – это тестирование и испытание (включает отладку) программного обеспечения.

Глава 4. ЭТАП ЖИЗНЕННОГО ЦИКЛА «УПРАВЛЕНИЕ ТРЕБОВАНИЯМИ»

Требования задают возможности, которые должна предоставлять система, так что соответствие или несоответствие некоторому множеству требований часто определяет успех или неудачу проекта. Поэтому имеет смысл узнать, что собой представляют требования, записать их, упорядочить и отслеживать их изменения.

Управление требованиями – это систематический подход к выявлению, организации и документированию требований к системе, а также процесс, в ходе которого вырабатывается и обеспечивается соглашение между заказчиком и выполняющей проект группой по поводу меняющихся требований к системе.

На данном этапе предъявляются требования к создаваемой системе по ее функциональности, качеству, пользовательскому интерфейсу, времени (и стоимости) разработки. Требования к системе должны быть четко оговорены, задокументированы и подписаны обеими сторонам. Делается это для того, чтобы ни одна из сторон не могла впоследствии отказаться от набора и характеристик требований (различное понимание требований, различное содержание каждой функции и правил взаимодействия разрабатываемой системы, недопонимание целей и задач разрабатываемой системы). Слишком общая постановка задачи приводит к множественности ее трактовки, слишком узкая постановка задачи лишает разработчиков маневра.

При составлении списка требований следует выполнить их ранжирование (оценка важности и необходимости). Так, например, в случае опасности невыполнения сроков разработки будет существовать возможность оценить, какое из требований может быть безболезненно исключено из данной версии ПО. Кроме того, следует помнить, что требования к системе растут в ходе проектирования и программирования примерно на 2–10%.

4.1. Типы требований

Спецификация требований может содержать (в зависимости от степени детализации – подробно или кратко):

- описание архитектуры системы;
- функциональные (и общие) требования;
- основные варианты использования;
- описание или скиншоты пользовательского интерфейса;
- требования безопасности и производительности;
- требования к программному и аппаратному обеспечению и ограничения.

На верхнем уровне типов располагаются бизнес-требования (business requirements).

Пример бизнес-требований: ПС должно сократить срок разработки заказов в 3 раза.

Следующий тип – требования пользователей (user requirements). *Пример требования пользователя*: система должна представлять диалоговые средства для ввода исчерпывающей информации о публикации, последующей фиксации информации в базе данных.

Требования пользователей часто бывают плохо структурированными, дублирующимися, противоречивыми. Поэтому для создания системы важен третий тип, в котором осуществляется формализация требований.

Следующий тип – функциональные требования (functional requirements), регламентирующие функционирование или поведение системы. Функциональные требования описывают сервисы, предоставляемые ПС, его поведение в определенных ситуациях, реакцию на те или иные входные данные и действия, которые система позволит выполнять пользователям. Другими словами, функциональные требования отвечают на вопрос, что должна делать система в тех или иных ситуациях. Иногда сюда добавляются сведения о том, чего система делать не должна.

Пример функциональных требований (или просто функций) по работе с электронной публикацией: публикация может быть создана, отредактирована, удалена и перемещена.

Нефункциональные требования, соответственно, регламентируют внутренние и внешние условия или атрибуты функционирования ПС. Можно выделить следующие основные группы нефункциональных требований: внешние интерфейсы (External Interfaces), атрибуты качества (Quality Attributes), ограничения (Constraints).

Среди внешних интерфейсов в большинстве современных систем наиболее важным является интерфейс пользователя (UI – User Interface). Кроме того, выделяются интерфейсы с внешними

устройствами (аппаратные интерфейсы), программные интерфейсы и интерфейсы передачи информации (коммуникационные интерфейсы).

Основными атрибутами качества являются применимость, надежность, производительность, эксплуатационная пригодность.

В спецификациях Rational Unified Process при классификации требований используется модель FURPS+. Акроним FURPS обозначает следующие категории требований: Functionality (функциональность), Usability (применимость), Reliability (надежность), Performance (производительность), Supportability (эксплуатационная пригодность).

Символ «+» расширяет FURPS-модель, добавляя к ней ограничения проекта, требования выполнения, требования к интерфейсу, физические требования.

Кроме того, в спецификациях RUP выделяются такие категории требований, как требования, указывающие на необходимость согласованности с некоторыми юридическими и нормативными актами, требования к лицензированию, требования к документированию.

Рассмотрим пример требований к ПС «Издатель».

I. Общие требования

1. Пользователь должен иметь возможность работать с неограниченным числом изданий.

2. Каждое издание должно содержать:

- ✔ файлы (контент – текст, звук, видео, графический материал);
- ✔ пункты (флаги, метки, описания, предыдущие версии, даты публикации, изменения и т. п.);
- ✔ структуру дерева.

3. Пользователь должен иметь полный доступ к пунктам и выполнять конфигурирование ПО.

4. Пользователь должен иметь возможность управлять категориями, файлами, атрибутами; загружать и выгружать файлы и т. д.

II. Требования к пользовательскому интерфейсу

UI ПС должно поддерживать русский и английский языки. Язык определяется в процессе инсталляции и т. д.

III. Функциональные требования

Требования по ведению log-файла. Все действия по управлению файлами должны записываться в файл со следующей структурой строки: YYYY-MM-DD hh-mm, имя дерева, имя файла, имя операции, результат. Log-файл должен быть доступен через пользовательский интерфейс.

ПС должно содержать следующие части: базу данных изданий, XML-дерево структуры базы данных, клиентский модуль.

Клиентский модуль должен поддерживать функциональность: парольный доступ, работа с файлами, атрибутами, структурой дерева, загрузка, выгрузка информации, публикация материала.

IV. Требования безопасности

ПС «Издатель» должно обеспечивать механизм шифрования выбранных файлов информации на основе алгоритма DES.

Все пользователи должны иметь одинаковые права.

V. Требования к программному и аппаратному обеспечению

Минимальные требования к аппаратному обеспечению:

- не менее чем 32-битовый процессор;
- не менее 128 RAM;
- не менее 1 G HDD.

Минимальные требования к программному обеспечению:

1) минимум Solaris 2.7;

2) Lib Xerces C++ 2.2.

4.2. Задание

1. Изучите предметную область и назначение разработки, выполните обзор существующих решений (аналогов) в данной области. Оформите проведенные исследования в виде раздела курсового проекта «Обзор…».

2. На основе выданного руководителем задания уточните требования, установите набор выполняемых функций, а также перечень и характеристики исходных данных, интерфейс. Затем определите перечень результатов, их характеристики и способы представления. Требования следует записать так, чтобы они были понятными, полными, последовательными. Это может быть документ, модель или листок с описанием.

Глава 5. ЭТАП ЖИЗНЕННОГО ЦИКЛА «АНАЛИЗ»

Существует эмпирический закон, который гласит, что в процессе создания ПО 30% времени тратится на анализ требований и проектирование, 40% времени – на кодирование и еще 30% – на доводку и тестирование. В конкретных случаях это отношение может меняться, однако для общего случая вполне верно.

Это один из наиболее ответственных этапов создания программного продукта. На основе требований составляется список функций, которые должна реализовывать система, формулируются спецификации программного обеспечения в виде текстовых описаний, структурных схем и диаграмм. Иными словами, выполняется анализ проекта.

В процессе определения спецификаций строится общая модель предметной области и конкретизируются основные функции ПС и его поведение при взаимодействии с окружающей средой [14]. Производится выбор архитектуры, среды разработки ПО, интерфейса пользователя и др. От этого выбора будет зависеть качество конечного программного продукта.

Окончанием данного этапа могут служить следующие результаты:

- начальный глоссарий терминов;
- общее описание системы – основные требования к проекту, его функциональные характеристики и ограничения;
- выбор и построение архитектуры проекта (см. прил. 1);
- начальная модель вариантов использования;
- план проекта, отражающий стадии и итерации;
- один или несколько прототипов.

5.1. Глоссарий терминов

Много недоразумений, которых можно было бы избежать, возникают из-за того, что одни и те же понятия называются по-разному. Например, одни оперирует понятиями бизнеса, другие (проектировщики) мыслят категориями классов и паттернов,

а третьи (разработчики баз данных) – таблицами и запросами. Поэтому важно, во-первых, выработать единую систему терминов и обозначений, согласованную всеми участниками разработки; во-вторых, «пронести» понятия из предметной области в архитектуру системы.

Словарь терминов представляет собой краткое описание основных понятий, используемых при составлении спецификаций [14].

Обычно описание термина в словаре выполняют по следующей схеме:

1) термин;

2) категория (понятие предметной области, элемент данных, условное обозначение и т. д.);

3) краткое описание.

Пример. Термин – Web-сайт. Категория – Интернет-программирование. Описание – совокупность Web-страниц с повторяющимся дизайном, объединенных по смыслу, навигационно и физически находящихся на одном сервере.

5.2. Определение функциональных требований

Если в управлении требованиями функции ПС описывают кратко ожидаемое поведение системы, то на данном этапе необходимо расширить их описание. Важно, чтобы функциональная спецификация ПС была математически точной. Желательно также, чтобы при ее разработке применялись математические методы и формализованные языки. Функциональная спецификация должна базироваться на четких понятиях и утверждениях, понимаемых однозначно.

Итак, функциональная спецификация состоит из трех частей:

- описание внешней информационной среды, с которой будет взаимодействовать разрабатываемое ПС. Должны быть определены все используемые каналы ввода и вывода и все информационные объекты, а также существенные связи между этими информационными объектами;
- задание функций ПС. Вводятся обозначения всех определяемых функций, специфицируются их входные данные и результаты выполнения с указанием типов данных и заданием всех ограничений, которым должны удовлетворять эти

данные и результаты. Определяется содержание каждой из этих функций;

✔ описание исключительных ситуаций, если таковые могут возникнуть при выполнении программ, и реакций на эти ситуации, которые должны обеспечить соответствующие программы.

5.3. Диаграммы вариантов использования

Описание функций системы по методологии RUP можно выполнить, используя диаграммы UML (Unified Modeling Language).

Определение прецедентов (вариантов использования – Use Cases) представляет собой краткое, лаконичное описание отдельной функции бизнес-процесса. Прецеденты позволяют описывать границы проектируемой системы, ее интерфейс, а затем выступают как основа для тестирования системы. После разделения системы на мелкие, четко определенные подзадачи становится значительно проще их описывать, реализовывать и оценивать прогресс их выполнения.

Выбор актеров

Перед построением диаграммы выделяют первичных актеров, использующих систему по прямому назначению. Каждый из первичных актеров участвует в выполнении одной или нескольких главных задач системы.

Кроме первичных, существуют и вторичные актеры. Они наблюдают и обслуживают систему. Вторичные актеры существуют только для того, чтобы первичные актеры могли использовать систему.

Определение элементов вариантов использования

Каждый элемент варианта использования задает некоторый путь использования системы, выполнение некоторой части функциональных возможностей. Полная совокупность элементов определяет все существующие пути использования системы.

Запускается элемент варианта использования актером, поэтому его удобно выявлять с помощью актеров. Рассматривая каждого актера, мы решаем, какие элементы Use Case он может выполнять. Для этого изучается описание системы.

Каждый элемент вариантов использования выделяет частный аспект функциональных возможностей системы. Поэтому можно независимо разрабатывать элементы вариантов использования для

разных функциональных областей, а позднее соединить их вместе (для формирования полной модели требований).

Пример. В соответствии с требованиями ПС «Издатель» (рис. 5.1) система имеет одну роль – издатель (пользователь).

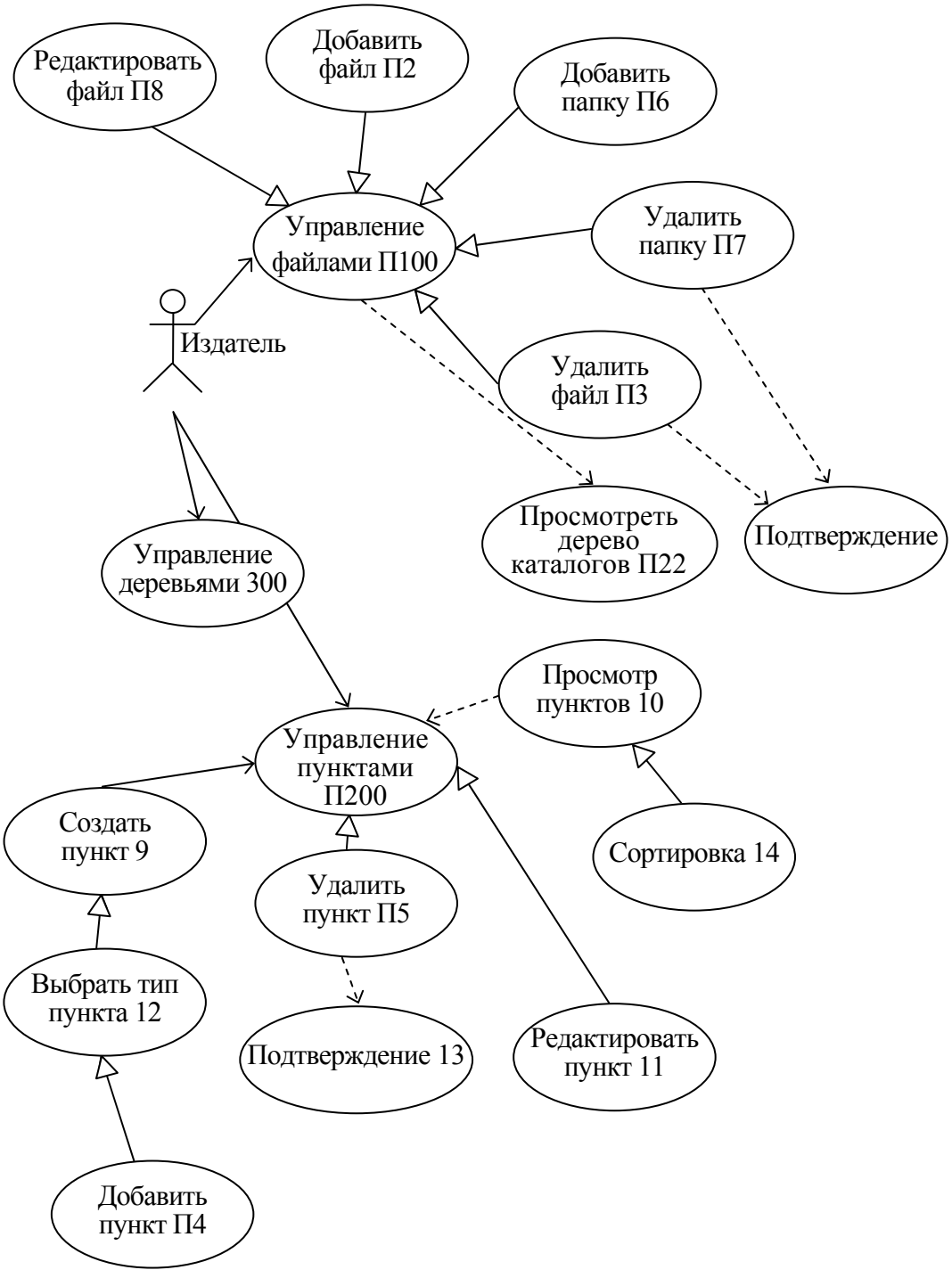


Рис. 5.1. Диаграмма вариантов использования ПС «Издатель»

Следовательно, основные прецеденты (варианты использования) для системы могут быть следующие:

1) П100 – управление файлами;
2) П200 – управление пунктами;
3) П300 – управление деревьями.

Вариант использования можно описать кратко или подробно. Краткая форма описания содержит название варианта использования, его цель, действующих лиц, тип варианта использования (основной, второстепенный или дополнительный) и его краткое описание [1, 2] (пример см. в 4.1).

Например, диаграмма прецедентов для ПС «Издатель» представлена на рис. 5.1.

Наибольшие трудности при построении диаграмм Use Case вызывает применение отношений включения и расширения.

Расширение функциональных возможностей

Для добавления в элемент Use Case новых действий удобно применять отношение расширения. С его помощью базовый элемент может быть расширен новым элементом (рис. 5.2).

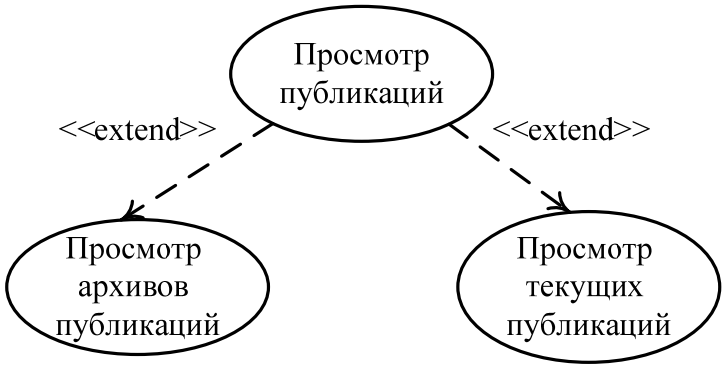


Рис. 5.2. Применение отношения расширения

Уточнение модели требований

Уточнение модели сводится к выявлению одинаковых частей в элементах прецедентов и извлечению этих частей. Любые изменения в такой части, выделенной в отдельный элемент, будут автоматически влиять на все элементы вариантов использования.

Извлеченные элементы называют абстрактными. Они не могут быть конкретизированы сами по себе, применяются для описания одинаковых частей в других, конкретных элементах.

На рис. 5.3 приведен пример выделения абстрактного элемента варианта использования *Печать*. Этот элемент будет специализироваться на выполнении распечаток.

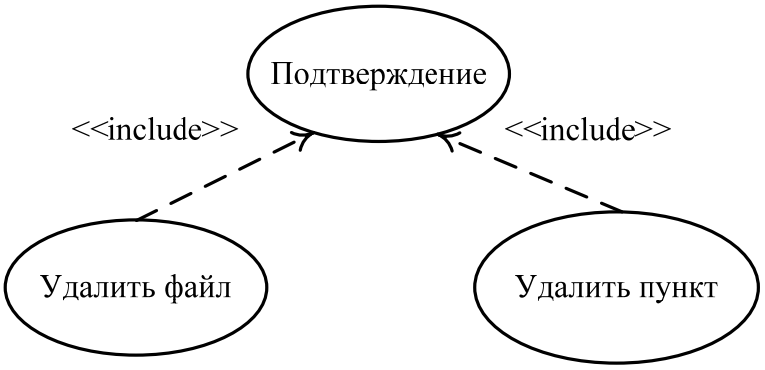


Рис. 5.3. Применение отношения включения

Пример детального описания вариантов использования, показанных на диаграмме, может быть представлен в виде табл. 5.1–5.3, которые приведены ниже. Каждая таблица соответствует одному варианту.

Таблица 5.1

Вход в систему П1

ПС	«Издатель»
Краткое описание	Пользователь проходит фазу аутентификации
Цель	Только авторизированные пользователи могут использовать функциональность ПС «Издатель»
ID/Приоритет	П1 – UC1
Предусловия	Нет
Постусловия	Пользователь получает доступ к главной странице приложения. Имя и пароль не требуются, за исключением случаев: 1) пользователь нажал кнопку «*Выход*»; 2) пользователь не использовал ПС в течение 30 мин (тайм-аут по умолчанию). Если вход выполнен не верно, сообщение об ошибке должно отображаться на текущей странице ПС. Количество повторных попыток равно трем
Актер	Издатель
Описание	Пользователь заполняет поля *Имя пользователя* и *Пароль*. Нажимает на кнопку «*Вход*»

Окончание табл. 5.1

ПС	«Издатель»
Ограничения	Имя пользователя и пароль не должны превышать 10 символов. Доступ должен быть осуществлен только при совпадении и *Имени пользователя*, и *Пароля*. Одновременно ПС может использовать только один пользователь
Расширения	Нет
Включения	Нет
Завершение	По истечении интервала тайм-аута должна быть установлена ошибка E_LOG_TIMEOUT. При неверном *Имени пользователя* и *Пароле* должна быть установлена ошибка E_LOG_ERROR

Таблица 5.2

Просмотр общей информации П22

ПС	«Издатель»
Краткое описание	Позволяет пользователю просматривать управляющую информацию
Цель	Информативная
ID/Приоритет	П22 – UC2
Предусловия	Нет
Постусловия	Пользователь видит следующую информацию: ✔ количество файлов; ✔ количество атрибутов; ✔ количество деревьев. Информация обновляется после нажатия на кнопку «*Обновить*»
Актер	Издатель
Описание	Нет
Ограничения	Нет
Расширения	П122
Включения	Нет

Таблица 5.3

Управление файлами П100

ПС	«Издатель»
Краткое описание	Позволяет пользователю управлять файлами
Цель	Управление файлами
ID/Приоритет	П100 – UC1
Предусловия	Нет
Постусловия	Нет
Событие	Пользователь переходит на вкладку *Файлы* ПС

Окончание табл. 5.3

ПС	«Издатель»
Актер	Издатель
Описание	Пользователь видит дерево каталогов и файлов. Пользователь может: 1) добавить файл (П2); 2) удалить файл (П3); 3) добавить пункт (П4); 4) удалить пункт (П5); 5) добавить папку (П6); 6) удалить папку (П7)
Ограничения	Нет
Расширения	Нет
Включения	Нет

Многие критикуют использование диаграмм из-за предполагаемой трудоемкости их составления и актуализации. Однако целесообразен любой способ эффективного описания элементов системы и способов их взаимодействия.

5.4. Задание

1. Составьте подробный план работы над курсовым проектом со сроками выполнения и сдачи, согласуйте с руководителем.

2. Обоснуйте выбор языка программирования, технологий, СУБД, которые планируется использовать для создания проекта.

3. В случае необходимости составьте глоссарий терминов (он должен включать определение основных понятий предметной области, описание структур элементов данных, их типов и форматов, а также всех сокращений и условных обозначений).

4. Опишите функциональные требования в виде диаграмм вариантов использования и таблиц.

5. Оформите результаты, используя MS Visio или Rational Rose.

Глава 6. ЭТАП ЖИЗНЕННОГО ЦИКЛА «ПРОЕКТИРОВАНИЕ»

Проектирование – итерационный процесс, при помощи которого требования к ПО транслируются в инженерные представления ПО. Вначале эти представления дают только концептуальную информацию (на высоком уровне абстракции – архитектуру), последующие уточнения приводят к формам, которые близки к текстам на языках программирования.

Обычно в проектировании выделяют две ступени: предварительное проектирование и детальное проектирование. Предварительное проектирование формирует абстракции архитектурного уровня, детальное проектирование уточняет эти абстракции, добавляет подробности алгоритмического уровня. Кроме того, во многих случаях выделяют интерфейсное проектирование, цель которого – сформировать GUI. Предварительное проектирование включает три типа деятельности:

- *структурирование системы.* Система структурируется на несколько подсистем, где под подсистемой понимается независимый программный компонент. Определяются взаимодействия подсистем;
- *моделирование управления.* Определяется модель связей управления между частями системы;
- *декомпозиция подсистем на модули.* Каждая подсистема разбивается на модули. Определяются типы модулей и межмодульные соединения.

6.1. Компонентный подход к проектированию

Концепция разработки систем с помощью изолированных модулей, выполняющих требуемую функциональность по определенному интерфейсу, давно завоевала популярность [20]. Компоненты значительно расширили круг разработчиков ПС и снизили требования к их квалификации. Большим достижением компонентов является возможность параллельной работы над отдельными частями системы.

Компонентный подход дополняет и расширяет существующие подходы в программировании, особенно ООП, как это видно из таблицы. Объекты рассматриваются на логическом уровне проектирования программной системы, а компоненты – это непосредственная физическая реализация объектов.

Один компонент может быть реализацией нескольких объектов или даже некоторой части объектной системы, полученной на уровне проектирования. Компоненты конструируются как некоторая абстракция, которая состоит из трех частей: информационной, внешней и внутренней.

Схема эволюции элементов компонентов

Элемент композиции	Описание элемента	Схема взаимодействия	Представление, хранение	Результат композиции
Процедура, подпрограмма, функция	Идентификатор	Непосредственное обращение, оператор вызова	Библиотеки подпрограмм и функций	Программа
Модуль	Паспорт модуля, связи	Вызов модулей, интеграция модулей	Банк, библиотеки модулей	Программа с модульной структурой
Объект	Описание класса	Создание экземпляров классов, вызов методов	Библиотеки классов	Объектно-ориентированная программа
Компонент	Описание логики (бизнес), интерфейсов (APL, IDL), схемы развертывания	Удаленный вызов в компонентных моделях (COM, CORBA, OSF и др.)	Репозитарий компонентов, серверы и контейнеры компонентов	Распределенное компонентно-ориентированное приложение
Сервис	Описание бизнес-логики и интерфейсов сервиса (XML, WSDL и др.)	Удаленный вызов (RPC, HTTP, SOAP и др.)	Индексация и каталогизация сервисов (XML, UDDI и др.)	Распределенное сервисно-ориентированное приложение

Объект всегда функционирует в составе сервера – динамической библиотеки или исполняемого файла, которые обеспечивают функционирование объекта.

Компонентный подход лежит в основе технологий, разработанных на базе COM (Component Object Model – компонентная модель объектов), и технологии создания распределенных приложений CORBA (Common Object Request Broker Architecture –

общая архитектура с посредником обработки запросов объектов). Эти технологии используют сходные принципы и различаются лишь особенностями их реализации.

Технология COM – фирмы Microsoft является развитием технологии OLE 1 (Object Linking and Embedding – связывание и внедрение объектов). Она определяет общую парадигму взаимодействия программ любых типов: библиотек, приложений, операционной системы, т. е. позволяет одной части программного обеспечения использовать функции (службы), предоставляемые другой, независимо от того, функционируют ли эти части в пределах одного процесса, в разных процессах на одном компьютере или на разных компьютерах (рис. 6.1) [21]. Модификация COM, обеспечивающая передачу вызовов между компьютерами, называется DCOM (Distributed COM – распределенная COM).

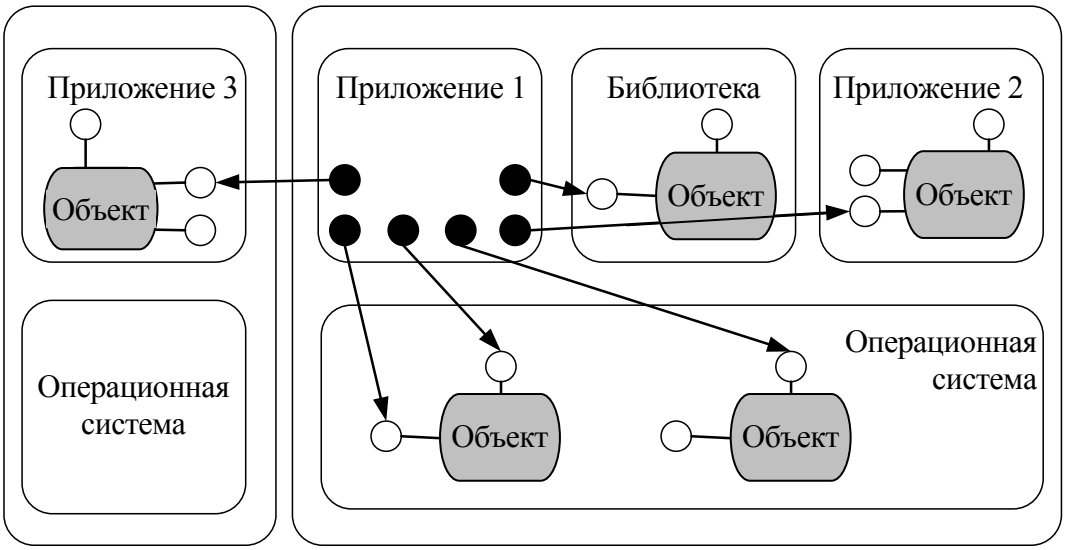


Рис. 6.1. Взаимодействие программных компонентов различных типов

По технологии COM-приложение предоставляет свои службы, используя специальные объекты – объекты COM, которые являются экземплярами классов COM. Объект COM так же, как и обычный объект, включает поля и методы, но в отличие от обычных объектов каждый объект COM может реализовывать несколько интерфейсов, обеспечивающих доступ к его полям и функциям. Это достигается за счет организации отдельной таблицы адресов методов для каждого интерфейса (по типу таблиц виртуальных методов). При этом интерфейс обычно объединяет несколько однотипных функций. Кроме того, классы COM поддерживают наследование интерфейсов, но не поддерживают

наследования реализации, т. е. не наследуют код методов, хотя при необходимости объект класса-потомка может вызвать метод родителя.

Каждый интерфейс имеет имя, начинающееся с символа «I», и глобальный уникальный идентификатор IID (Interface IDentifier). Любой объект COM обязательно реализует интерфейс IUnknown (на схемах этот интерфейс всегда располагают сверху (рис. 6.1)). Использование этого интерфейса позволяет получить доступ к остальным интерфейсам объекта [22, 23].

Расширением понятия компонента является паттерн – абстракция, которая содержит описание взаимодействия совокупности объектов в общей кооперативной деятельности, для которой определены роли участников и их ответственность.

6.2. Проектирование развертывания

Развертывание приобретает актуальность для больших систем. В простейшем случае программа не требует специальных действий для работы с ней. Однако для сложных систем для запуска комплекса может возникнуть необходимость в выполнении целого ряда работ. Например, может потребоваться развернуть аппаратное обеспечение или оборудование, инсталляция аппаратного и программного обеспечения для его работы, установка и настройка специального программного обеспечения: серверов приложений и БД, клиентских программ, заполнение баз начальными данными.

Компонентные диаграммы

Компонентная диаграмма UML (component diagram) – разновидность диаграмм реализации, моделирующих физические аспекты систем. Компонентная диаграмма показывает организацию набора компонентов и зависимости между компонентами [24].

Элементами компонентных диаграмм являются компоненты и интерфейсы, а также отношения зависимости и реализации. Как и другие диаграммы, компонентные диаграммы могут включать примечания и ограничения. Кроме того, компонентные диаграммы могут содержать пакеты или подсистемы, используемые для группировки элементов модели в крупные фрагменты.

Компонент – базисный строительный блок физического представления ПС. Графически компонент изображается как прямоугольник с вкладками, обычно включающий имя (рис. 6.2).

Рис. 6.2. Обозначение компонента

Моделирование реализации системы

Реализация системы может включать большое количество разнообразных компонентов: исполняемых элементов, динамических библиотек, файлов данных, справочных документов, файлов инициализации, файлов регистрации, сценариев, файлов установки.

Моделирование этих компонентов, отношений между ними – важная часть управления конфигурацией системы.

Например, на рис. 6.3 показана часть реализации системы, группируемая вокруг исполняемого элемента *издатель.exe*.

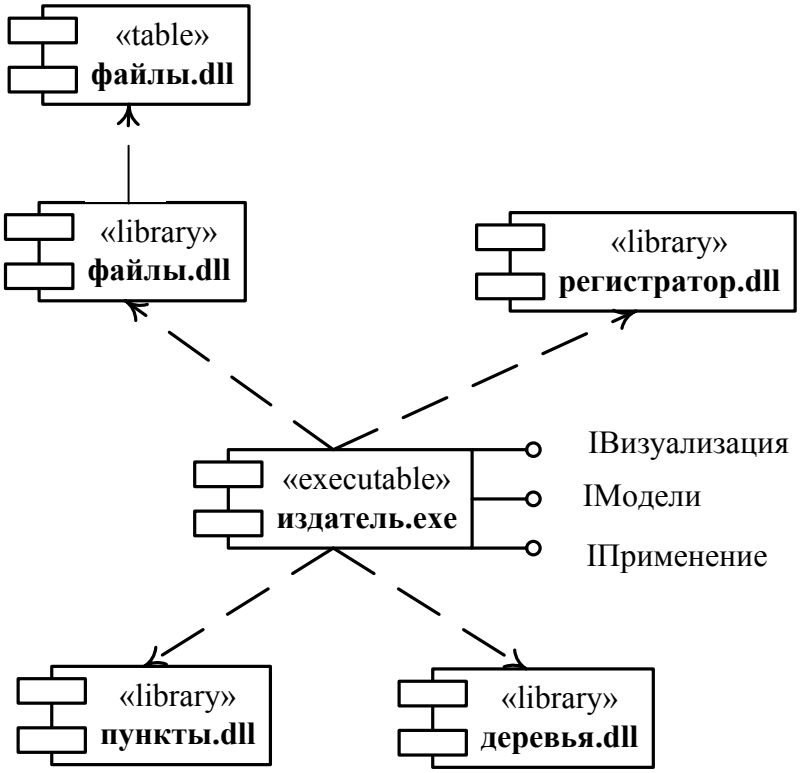


Рис. 6.3. Моделирование реализации системы

Здесь изображены четыре библиотеки (*регистратор.dll*, *файлы.dll*, *пункты.dll*, *деревья.dll*), а также таблица базы данных (*файлы.tbl*). В диаграмме указаны отношения зависимости, существующие между компонентами.

Диаграммы размещения

Диаграмма размещения (deployment diagram) – вторая из двух разновидностей диаграмм реализации UML, моделирующих физические аспекты систем. Диаграмма размещения показывает конфигурацию обрабатывающих узлов в период работы системы, а также компоненты (рис. 6.4).

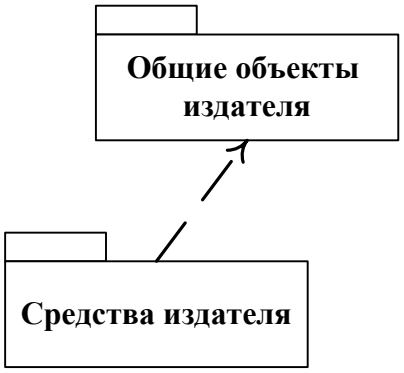


Рис. 6.4. Диаграмма размещения ПС «Издатель»

Элементами диаграмм размещения являются узлы (физический элемент, который существует в период работы системы и представляет компьютерный ресурс), а также отношения зависимости и ассоциации. Как и другие диаграммы, диаграммы размещения могут содержать примечания и ограничения. Кроме того, диаграммы размещения могут включать компоненты, каждый из которых должен быть размещен в некотором узле, а также содержать пакеты или подсистемы, используемые для группировки элементов модели в крупные фрагменты. При необходимости визуализации конкретного варианта аппаратной топологии в диаграммы размещения могут помещаться объекты.

6.3. Логическое разделение на слои

Независимо от типа проектируемого приложения его структуру можно разложить на логические группы программных компонентов. Эти логические группы называются слоями. Слои помогают разделить разные типы задач, которые осуществляются этими компонентами, что упрощает создание дизайна, поддерживающего возможность повторного использования компонентов. Каждый логический слой включает ряд отдельных типов

компонентов, сгруппированных в подслои, каждый из подслоев выполняет определенный тип задач.

Размещение нескольких слоев на одном компьютере (одном уровне) – довольно обычное явление. Термин «уровень» используется в применении к схемам физического распределения, например двухуровневое, трехуровневое, *n*-уровневое.

Типовой трехслойный дизайн, представленный на рис. 6.5, включает следующие слои [25]:

1) слой представления содержит ориентированную на пользователя функциональность, которая отвечает за реализацию взаимодействия пользователя с системой, и, как правило, включает компоненты, обеспечивающие общую связь с основной бизнес-логикой, инкапсулированной в бизнес-слое;

2) бизнес-слой (его еще называют слоем бизнес-логики) реализует основную функциональность системы и инкапсулирует связанную с ней бизнес-логику. Обычно он состоит из компонентов, некоторые из которых предоставляют интерфейсы сервисов, доступные для использования другими участниками взаимодействия;

3) слой доступа к данным обеспечивает доступ к данным, хранящимся в рамках системы, и данным, предоставляемым другими сетевыми системами. Доступ может осуществляться через сервисы. Слой данных предоставляет универсальные интерфейсы, которые могут использоваться компонентами бизнес-слоя.

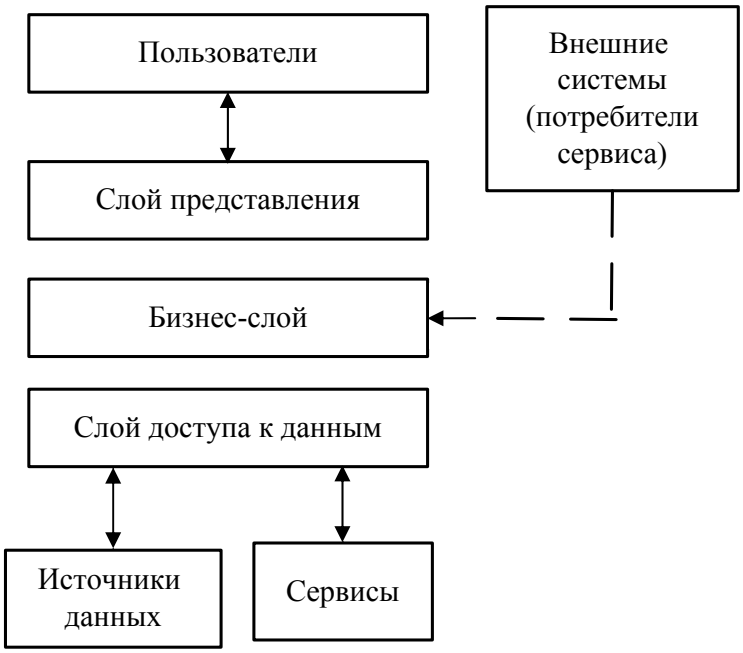


Рис. 6.5. Схема логического разделения на слои

Обычным подходом при создании ПС, которое должно обеспечивать сервисы для других ПС, а также реализовывать непосредственную поддержку клиентов, является использование слоя сервисов, который предоставляет доступ к бизнес-функциональности ПС. Слой сервисов обеспечивает альтернативное представление, позволяющее клиентам использовать другой механизм для доступа к ПС.

Если программа не предоставляет сервисы, возможно, отдельный слой сервисов не понадобится, тогда ПС будет включать только слой представления, бизнес-слой и слой доступа к данным.

Приступая к проектированию ПС, прежде всего, сосредоточьтесь на самом высоком уровне абстракции и начинайте с группировки функциональности в слои. Далее следует определить открытый интерфейс для каждого слоя, который зависит от типа создаваемого продукта. Определив слои и интерфейсы, необходимо принять решение о том, как будет развертываться ПС. Наконец, выбираются протоколы связи для обеспечения взаимодействия между слоями и уровнями ПС.

6.4. Проектирование слоя представления

Слой представления содержит компоненты, реализующие и отображающие пользовательский интерфейс, а также управляющие взаимодействием с пользователем:

- ✔ компоненты пользовательского интерфейса (UI – User Interface) – визуальные элементы ПС, используемые для отображения данных пользователю и приема пользовательского ввода;
- ✔ компоненты логики представления – это код ПС, определяющий его поведение и структуру и не зависящий от конкретной реализации пользовательского интерфейса.

Существует несколько общих принципов, которых следует придерживаться при проектировании слоя представления:

1) обеспечьте полезные и информативные сообщения об ошибках без предоставления в них конфиденциальных данных;

2) избегайте блокирования пользовательского интерфейса при выполнении действий, для завершения которых может потребоваться довольно длительное время. Как минимум, обеспечьте обратную связь о ходе выполнения действия и продумайте возможность отмены операции;

3) рассмотрите возможность предоставления пользователю дополнительных возможностей по конфигурированию и, где это возможно, персонализации UI, обеспечивая тем самым гибкий и настраиваемый интерфейс;

4) избегайте применения жестко кодированных строк и внешних ресурсов для текстовых данных или данных компоновки.

Валидация

Эффективная стратегия проверки пользовательского ввода и данных имеет критически важное значение для безопасности и корректной работы ПС. Определите правила валидации пользовательского ввода и бизнес-правила, существующие в слое представления. При проектировании стратегии проверки пользовательского ввода и данных руководствуйтесь следующими рекомендациями:

- ✔ проверка пользовательского ввода должна проводиться в слое представления, тогда как проверка на соответствие бизнес-правилам – в бизнес-слое. Однако если бизнес-слой и слой представления разнесены физически, логика проверки на соответствие бизнес-правилам должна дублироваться в слое представления для улучшения удобства использования и уменьшения времени отклика. Этого можно достичь с помощью метаданных или путем применения одинаковых компонентов правил проверки в обоих слоях;
- ✔ руководствуйтесь целью ограничить, предотвратить и очистить злонамеренный ввод;
- ✔ убедитесь, что правильно обрабатываете ошибки валидации, и избегайте предоставления конфиденциальных данных в сообщениях об ошибках. Кроме того, обеспечьте протоколирование сбоев при проверке.

Прототип пользовательского интерфейса

Прототип – это наглядная модель пользовательского интерфейса. В сущности, это функционирующий «черновик» интерфейса, созданный на основе представлений о потребностях пользователей. Прототип может принимать множество различных форм – от бумажных макетов до реальных программ, имитирующих работу пользовательского интерфейса. Однако независимо от формы прототип должен давать четкое представление о способе взаимодействия пользователя с программой.

На первом этапе создания пользовательского интерфейса нужно определить самые важные задачи, которые потребуется решать пользователям. Число задач может варьироваться в зависимости от сложности продукта. Попробуйте выделить хотя бы следующие категории:

1) *задачи, которые скорее всего придется решать новым пользователям программы.* Здесь надо понять потребности новичков и сделать так, чтобы они как можно скорее преуспели в решении своих проблем с помощью вашей программы;

2) *задачи, которые чаще всего решают постоянные пользователи программы.* Нужно постараться не разочаровать постоянных пользователей, безошибочно определив их ключевые задачи. Соответственно, следует сосредоточить основное внимание на этих задачах, которые нужно максимально обогатить полезными возможностями, качественно реализовать и хорошо описать.

Следующий шаг – выбрать инструмент, позволяющий создать прототип легко и быстро. К наиболее популярным методикам создания прототипов относятся:

✔ *прототипы на бумаге.* Для создания такого прототипа нужно просто нарисовать фрагменты пользовательского интерфейса на бумаге. Не обязательно добиваться точности воспроизведения – достаточно, чтобы эти кусочки бумаги давали четкое представление об элементах пользовательского интерфейса;

✔ *инструменты RAD.* Создание прототипов с помощью инструментов для быстрой разработки приложений – вероятно, самая популярная методика. Подходящим можно считать любой инструмент, позволяющий быстро создать наглядную функционирующую модель пользовательского интерфейса (рис. 6.6);

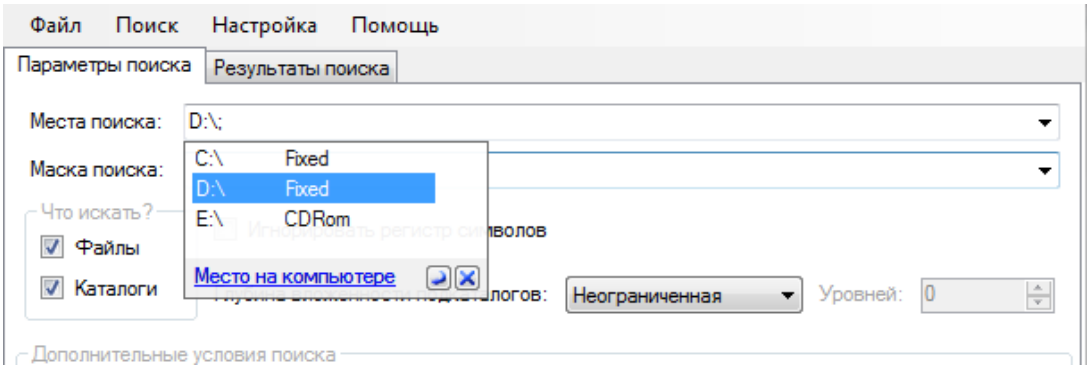


Рис. 6.6. Пример фрагмента интерфейса ПС «Издатель»

✔ *описания.* Создание прототипов пользовательского интерфейса с помощью описаний – наименее ценный из трех представленных здесь инструментов. Описания пользовательского интерфейса страдают от трех недостатков. Во-первых, их интерпретация часто неопределенна. Во-вторых, описание трудно протестировать и оценить. Применять описания в качестве эталонов также очень трудно. Наконец, даже при наличии отзывов, поддерживать точность и внутреннюю согласованность описаний зачастую нелегко.

Получив прототип, можно приступать к его проверке с помощью пользователей. Для этого нужно попросить пользователей выполнить определенные вами ключевые задачи с помощью только что созданного прототипа и после этого внести в прототип соответствующие изменения.

6.5. Проектирование бизнес-слоя

Бизнес-слой обычно включает следующие компоненты: фасад приложения и компоненты бизнес-логики [25].

Фасад приложения (необязательный компонент) обычно обеспечивает упрощенный интерфейс для компонентов бизнес-логики, часто сочетая множество бизнес-операций в одну, что упрощает использование бизнес-логики.

Бизнес-логика, как и любая логика приложения, занимается вопросами извлечения, обработки, преобразования и управления данными приложения; применением бизнес-правил и политик и обеспечением непротиворечивости и действительности данных. Чтобы создать наилучшие условия для повторного использования, компоненты бизнес-логики не должны содержать поведения или логики ПС конкретного варианта использования или пользовательской истории.

При проектировании бизнес-слоя необходимо учитывать технические требования для основных составляющих этого слоя: компонентов бизнес-слоя, бизнес-сущностей и компонентов бизнес-процесса. Рекомендуемый процесс проектирования бизнес-слоя содержит [23]:

1) создание дизайна бизнес-слоя в первом приближении. Определите, кто будет использовать бизнес-слой: слой представления, слой сервисов или другие приложения. Это задаст

тактику методов доступа к бизнес-слою. Далее определите требования безопасности для бизнес-слоя, требования и стратегию валидации;

2) проектирование компонент бизнес-слоя. При проектировании и реализации приложения могут использоваться несколько типов компонентов бизнес-слоя. Примерами таких компонентов являются компоненты бизнес-процесса, служебные компоненты и вспомогательные компоненты;

3) проектирование компонент бизнес-сущностей. Бизнес-сущности используются для размещения и управления бизнес-данными, используемыми приложением. Бизнес-сущности должны гарантировать проверку данных, размещаемых в сущности. Кроме того, бизнес-сущности обеспечивают свойства и операции для доступа и инициализации данных сущности;

4) проектирование компонент рабочего процесса. Существует масса сценариев, в которых задачи должны выполняться в установленном порядке в зависимости от завершения определенных этапов или координируются человеком. Эти требования можно отобразить в основных сценариях рабочего процесса.

6.6. Проектирование слоя доступа к данным

Общие принципы

Для проектирования реляционных БД могут быть использованы структурные методы проектирования, основанные на ER-диаграммах, и объектно-ориентированные методы, базирующиеся на использовании языка UML (рис. 6.7). Они различаются, главным образом, лишь терминологией. ER-модель концептуально проще UML, в ней меньше понятий, терминов, вариантов применения.

Инкапсулируйте функциональность доступа к хранилищу данных в слое доступа к данным. Слой доступа к данным должен скрывать детали доступа к источнику данных. Он должен обеспечивать управление подключениями, формирование запросов.

Выберите технологию доступа к данным (см. прил. 2). Выбор технологии доступа к данным зависит от типа данных, с которыми придется работать, и того, как предполагается обрабатывать данные в приложении [27].

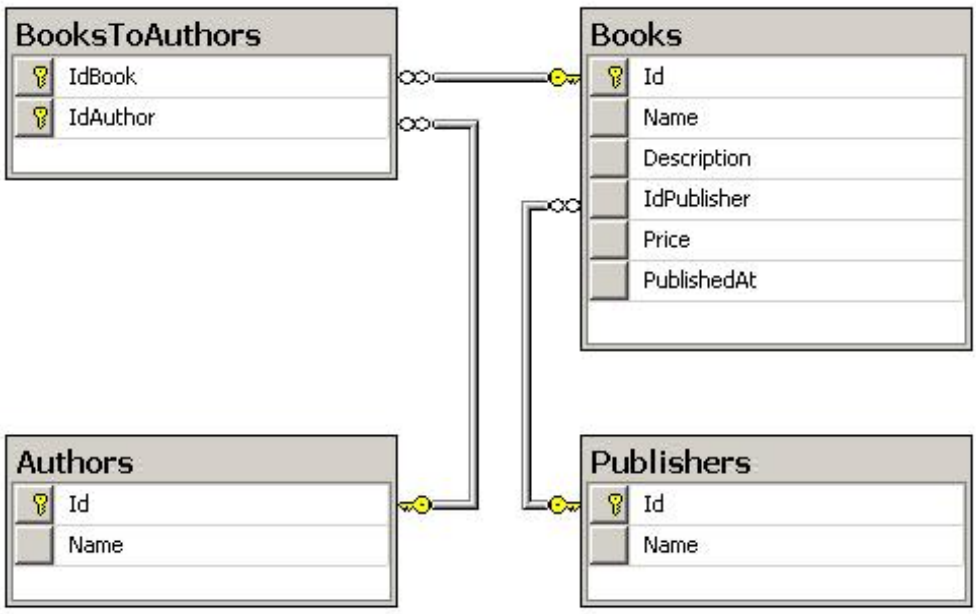


Рис. 6.7. Структура базы данных для издательства

Специальные вопросы проектирования

Существует ряд общих вопросов, на которые следует обратить внимание при проектировании.

Правильный выбор формата данных обеспечивает возможность взаимодействия с другими приложениями и способствует обмену сериализованными данными между разными процессами или компьютерами.

Выберите соответствующую стратегию распространения исключений. Например, разрешите исключениям распространяться к граничным слоям, где они могут быть зарегистрированы и в случае необходимости преобразованы перед передачей в следующий слой.

Определите, как будут обрабатываться исключения, возникающие при обработке данных. Слой доступа к данным должен перехватывать и обрабатывать (по крайней мере, обеспечивать начальный этап) все исключения, связанные с источниками данных и операциями CRUD (Create, Read, Update и Delete). Исключения, касающиеся самих данных и доступа к источникам данных, и ошибки истечения времени ожидания должны обрабатываться в этом слое и передаваться в другие слои, только если сбои оказывают влияние на скорость ответа и функциональность приложения.

Убедитесь, что перехватываете исключения, которые не будут перехвачены где-либо в другом месте (например, глобальном

обработчике ошибок), и очищайте ресурсы и состояние после возникновения исключения.

Правильно выберите стратегию протоколирования и уведомления для критических ошибок и исключений, чтобы регистрировать достаточный объем сведений об исключениях и не разглашать конфиденциальных данных.

Запросы

Запросы – это основные операции манипуляции данными в слое доступа к данным. Они являются механизмом преобразования запросов приложения в CRUD-действия базы данных. Поскольку запросы имеют такое большое значение, они должны быть оптимизированы для обеспечения максимальной производительности и пропускной способности базы данных. При использовании запросов в слое доступа к данным руководствуйтесь следующими рекомендациями:

- используйте параметризованные SQL-запросы и типизированные параметры, чтобы повысить безопасность и снизить шансы успеха атак с внедрением SQL-кода;
- рассмотрите возможность использования объектов для построения запросов. Оптимизируйте схему базы данных для выполнения запросов;
- при построении динамических SQL-запросов избегайте смешения бизнес-логики с логикой, используемой для формирования SQL-выражений, поскольку это может сильно усложнить обслуживание и отладку кода.

Хранимые процедуры

В прошлом хранимые процедуры обеспечивали лучшую производительность по сравнению с динамическими SQL-выражениями. Однако сейчас современные ядра СУБД практически уравняли производительность обработки хранимых процедур и динамических SQL-выражений (использующих параметризованные запросы). Основными факторами при принятии решения об использовании хранимых процедур являются абстракция, удобство обслуживания и среда выполнения.

С точки зрения безопасности и производительности основными рекомендациями для хранимых процедур является использование типизированных параметров и отсутсвие динамического SQL в хранимых процедурах [28].

Валидация

Создание эффективного решения валидации пользовательского ввода и данных имеет решающее значение для безопасности приложения. Определите правила валидации данных, поступающих с других слоев и от компонентов сторонних производителей, а также из базы данных или хранилища данных. При проектировании стратегии проверки руководствуйтесь следующими рекомендациями:

1) проверяйте все данные, получаемые слоем доступа к данным от всех вызывающих сторон. Гарантируйте правильную обработку значений NULL и фильтрацию недействительных символов;

2) при проектировании механизмов валидации учитывайте назначение данных. Например, пользовательский ввод, используемый для создания динамического SQL, должен проверяться на наличие символов или шаблонов, встречающихся в атаках внедрением SQL-кода;

3) возвращайте информативные сообщения об ошибках в случае сбоя валидации.

Определившись с данными, подлежащими проверке, выберите методики проверки для них. Самыми распространенными методиками проверки являются:

- прием заведомо допустимого. Принимаются только данные, удовлетворяющие заданным критериям, все остальные данные отклоняются;
- отклонение заведомо недопустимого. Принимаются данные, не содержащие известный набор символов или значений;
- очистка. Известные плохие символы или значения устраняются или преобразовываются с целью сделать ввод безопасным.

После разложения на слои переходите к статическому проектированию ПС.

6.7. Диаграммы классов

Основным средством для представления статических моделей являются диаграммы классов (class diagram) [2, 3, 9].

В статических моделях не показывается динамика изменений системы во времени. Они несут в себе не только структурные описания, но и описания операций, реализующих заданное поведение

системы. Вершины диаграмм классов содержат классы, а дуги (ребра) – отношениями между ними.

UML предлагает использовать три уровня диаграмм классов в зависимости от степени их детализации:

1) концептуальный уровень, на котором диаграммы классов отображают связи между основными понятиями предметной области;

2) уровень спецификаций, на котором диаграммы классов отображают связи объектов этих классов;

3) уровень реализации, на котором диаграммы классов непосредственно показывают поля и операции конкретных классов.

Каждую из перечисленных моделей используют на определенном этапе разработки программного обеспечения:

- концептуальную модель – на этапе анализа;
- диаграммы классов уровня спецификации – на этапе проектирования;
- диаграммы классов уровня реализации – на этапе реализации.

Диаграммы классов обычно содержат следующие сущности: классы, интерфейсы, кооперации, отношения зависимости, обобщения и ассоциации.

Отношения в диаграммах классов

Отношения, используемые в диаграммах классов, показаны на рис. 6.8.

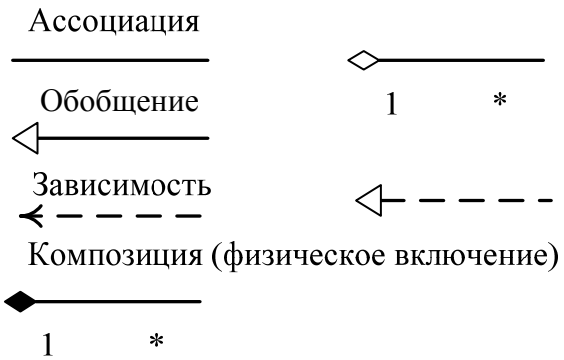


Рис. 6.8. Отношения в диаграммах классов

Ассоциация фиксирует структурные отношения между экземплярами разных классов. Ассоциации являются клеем, соединяющим воедино все элементы программной системы. Без ассоциаций система превращается в набор изолированных классов-одиночек.

Пример. В системе *Издатель* имеются две ключевые абстракции – *Файл* и *Папка*. Класс *Файл* играет роль элемента, хранимого в *Папке*. Класс *Папка* играет роль хранилища для *Файла* (рис. 6.9).

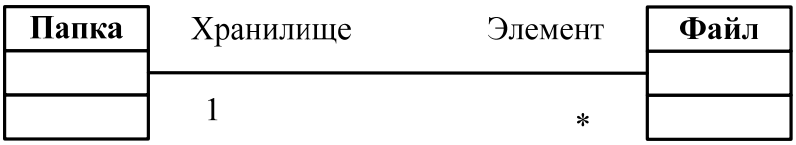


Рис. 6.9. Отношение ассоциации

Наследование – наиболее популярная разновидность отношения ***обобщение-специализация***. Альтернативой наследованию считается делегирование. При делегировании объекты делегируют свое поведение родственным объектам. При этом классы становятся не нужны.

Конкретизация выражает другую разновидность отношения ***обобщение-специализация*** (применяется в Ada 95, C++).

Зависимость отображает влияние одного класса на другой. Она часто представляется в виде частной формы – использования, которое фиксирует отношение между клиентом, запрашивающим услугу, и сервером, предоставляющим эту услугу.

Агрегация обеспечивает отношения целое-часть, объявляемые для экземпляров классов. Агрегация дает возможность перемещения от целого (агрегата) к его частям (свойствам). Например, файл и текст (рис. 6.10).

На рис. 6.11 приведен пример нефизического включения частей (*Пользователь*, *Администратор*) в агрегат *Издательство*. Очевидно, что *Пользователь* и *Администратор* являются элементами *Издательство*, но они не входят в него физически. В этом случае говорят, что части включены в агрегат по ссылке.

Реализация определяет отношение, при котором класс-приемник обеспечивает свою собственную реализацию интерфейса другого класса-источника. Иными словами, здесь идет речь о наследовании интерфейса. Семантически реализация – это «скрещивание» отношений зависимости и обобщения-специализации.

В качестве примера построения диаграммы классов на рис. 6.12 представлена диаграмма классов для ПС «Издатель». Эту систему образует шесть классов.

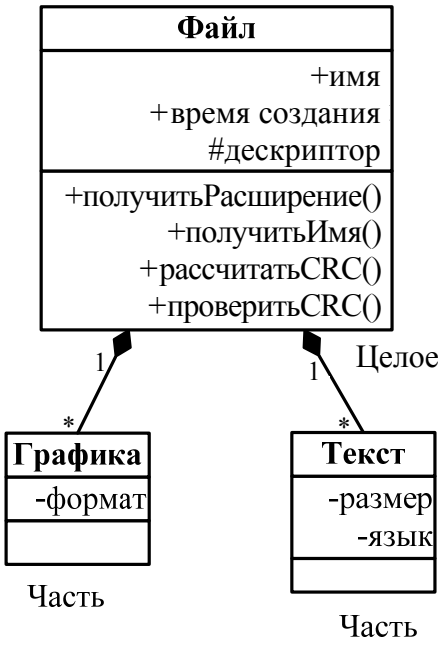


Рис. 6.10. Отношение агрегации

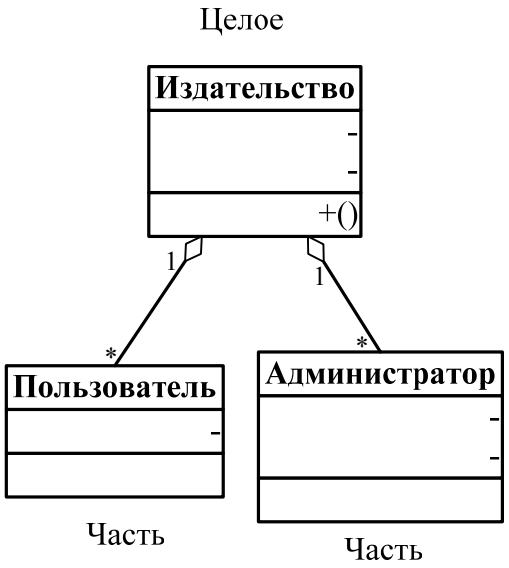


Рис. 6.11. Отношение нефизической агрегации

Пример

Классы-агрегаты *Издательство* и *Отдел* имеют операции добавления и удаления своих частей, которые включаются в агрегаты по ссылке (рис. 6.12). Частями *Издательства* являются *Пользователи* и *Отделы*, а частями *Отделов – Издатели*. Отношения агрегации между классом *Издательство* и классами *Отдел* и *Покупатель* слегка отличны. Издательство может состоять из одного или нескольких отделов, но каждый отдел находится в одном и только одном издательстве. С другой стороны, в издательство может ходить любое количество пользователей (включая нулевое количество), причем пользователь может посещать одно или несколько издательств.

Между классами *Издательство* и *Издатель* существуют два отношения – агрегация и ассоциация. Агрегация показывает, что каждый издатель работает в одном или нескольких отделах, а в каждом отделе должен быть хотя бы один издатель. Ассоциация отображает, что каждым отделом управляет только один издатель – менеджер, а некоторые издатели не являются менеджерами.

Ассоциация между классами *Публикация* и *Издатель* фиксирует, что в публикации должен быть занят хотя бы один издатель, впрочем издатель может создавать любое количество публикаций (или вообще может не работать с публикациями).

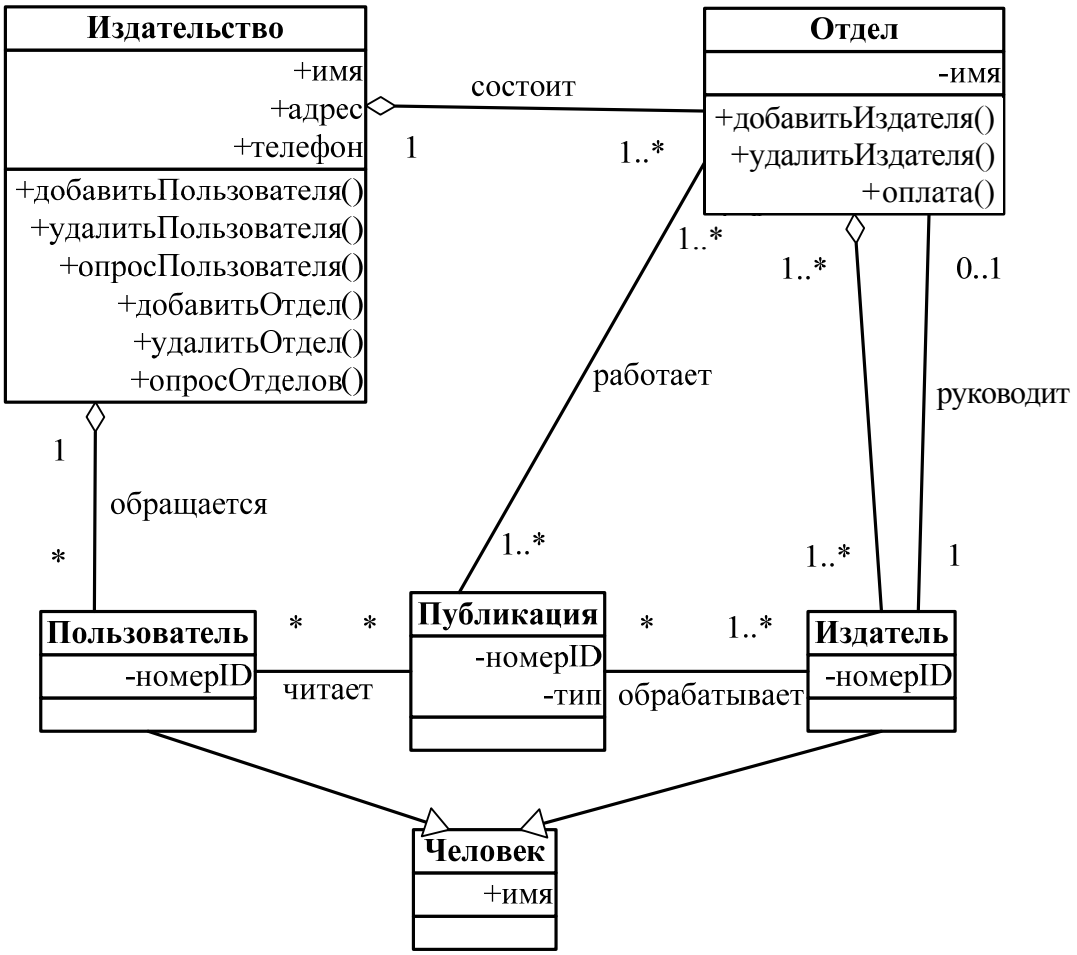


Рис. 6.12. Диаграмма классов ПС «Издательство»

Между классами *Публикация* и *Пользователь* тоже определена ассоциация. Она поясняет, что пользователь может использовать любое число публикаций, а каждой публикацией может пользоваться любое число пользователей.

И наконец, на диаграмме отображены два отношения наследования, утверждающие, что и в пользователях, и в издателях есть человеческое начало.

Описанная концептуальная модель характеризует статические свойства разрабатываемого ПО. Для описания особенностей его поведения, т. е. возможных действий системы, целесообразно использовать диаграммы последовательностей, диаграммы деятельности, а при необходимости и диаграммы состояний объектов.

На данном этапе может быть также представлено детальное описание всех классов. Например:

Public Methods

CFile ()

virtual **~CFile** ()

long **GetSize** ()

std::string **GetPath** ()

std::string **GetName** ()

std::string **GetExtension** ()

bool **GetCRC** (**FileCRC** &crc)

bool **CheckCRC** (**FileCRC** crc)

Public Attributes

char * **m_szName**

FILE * **m_pData**

bool CFile::CheckCRC (FileCRC ***crc***)

метод используется для проверки CRC

Parameters:

crc входной crc для сравнения

Returns:

TRUE в случае совпадения, в противном случае - FALSE

bool CFile::GetCRC (FileCRC & ***crc***)

метод вычисляет CRC файла

Parameters:

crc выходной crc файла

Returns:

TRUE в случае успеха, в противном случае - FALSE

std::string CFile::GetExtension ()

метод возвращает расширение файла

Returns:

Расширение файла

std::string CFile::GetName ()

возвращает имя файла без расширения и пути

Returns:

Имя файла

std::string CFile::GetPath ()

возвращает путь к файлу

Returns:

Путь к файлу

long CFile::GetSize ()

получает размер файла

Returns:

Размер файла

6.8. Диаграммы последовательностей

Диаграмма последовательностей системы (sequence diagram) относится к динамическим моделям системы и обеспечивает представление поведения систем. «Динамизм» этих моделей состоит в том, что в них отражается изменение состояний в процессе работы системы (в зависимости от времени).

Диаграмма последовательностей – графическая модель, которая для определенного сценария варианта использования показывает динамику взаимодействия объектов во времени (рис. 6.13) [1, 2].

6.9. Диаграммы состояний

Главное предназначение диаграммы состояний (statechart diagram) – описать возможные последовательности состояний и переходов, которые в совокупности характеризуют поведение элемента модели в течение его жизненного цикла. Диаграммы состояний используются также для моделирования динамических аспектов системы.

6.10. Диаграммы деятельности

Для моделирования процесса выполнения операций в языке UML используются диаграммы деятельности. На этапе анализа требований и уточнения спецификаций диаграммы деятельности позволяют конкретизировать основные функции разрабатываемого программного обеспечения.

Пример диаграммы деятельности приведен на рис. 6.14. Эта диаграмма описывает деятельность пользователя в издательстве. Здесь представлены две точки ветвления – для выбора способа публикации и для принятия решения о запросе. Присутствуют три линейки синхронизации: верхняя отражает разделение на два параллельных процесса, средняя показывает и разделение, и слияние процессов, а нижняя – только слияние процессов.

Дополнительно на этой диаграмме показаны две дорожки – дорожка пользователя и дорожка издательства, которые разделены вертикальной линией. Каждая дорожка имеет имя и фиксирует область деятельности конкретного лица, обозначая зону его ответственности.

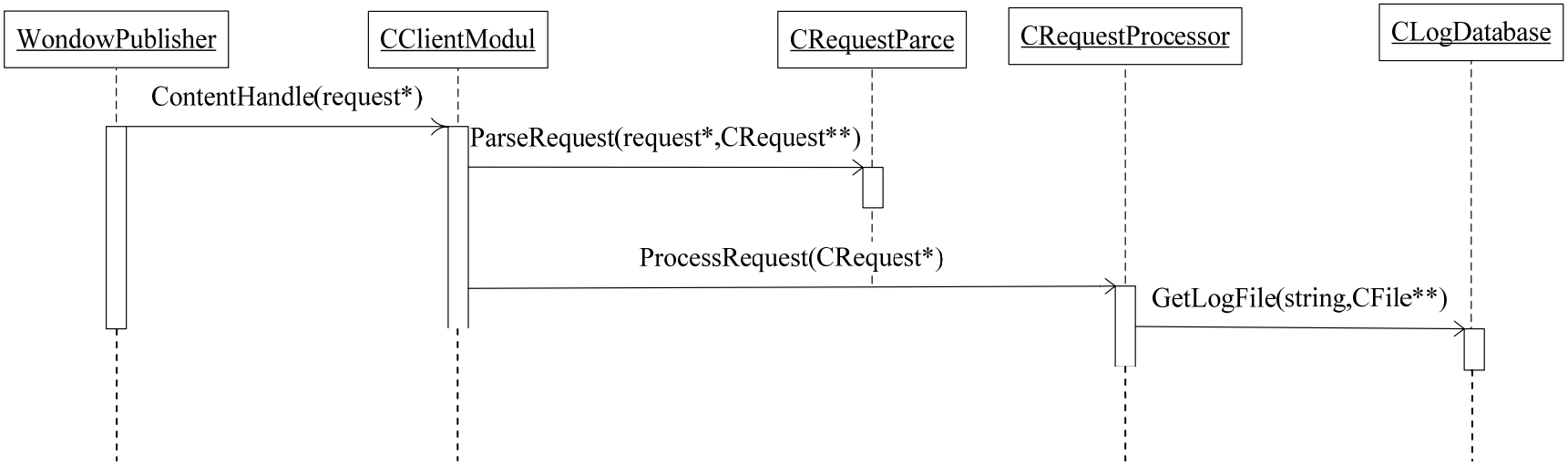


Рис. 6.13. Диаграмма последовательностей

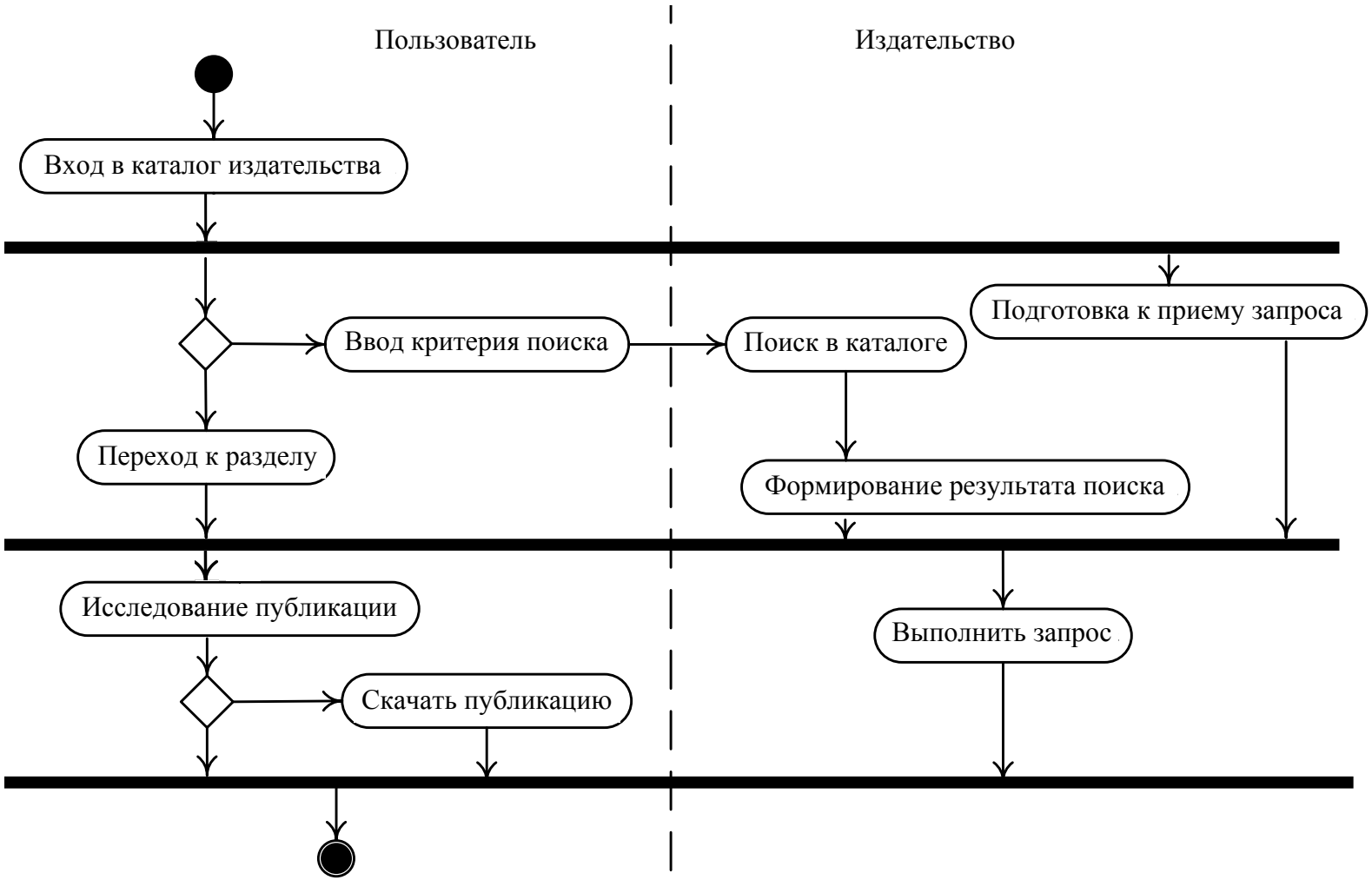


Рис. 6.14. Диаграмма деятельности пользователя в издательстве

6.11. Реинженеринг бизнес-процессов

Термин «реинженеринг» стал очень популярен в последнее время. Означает он, прежде всего, переосмысление, перестройку, повышение эффективности бизнес-процессов. Во многих случаях, когда ставится задача не просто построения новой системы «с нуля», а изменения существующих бизнес-процессов, полезно построение моделей системы AS IS (как есть) и TO BE (как будет) с целью выяснения различий (и своевременного их согласования).

К средствам реинженеринга относятся документаторы, анализаторы программ, средства реструктурирования:

- Adpac CASE Tools (Adpac);
- Scan/COBOL и Superstructure (Computer Data Systems);
- Inshtctor/Recoder (Language Tecnologe).

6.12. Задание

1. Постройте диаграмму размещения и развертывания ПС.
2. Выполните проектирование ПС. Выделите в нем компоненты (объекты) для трех слоев: слоя представления, бизнес-слоя и слоя доступа к данным.
3. Проведите декомпозицию и определите полную диаграмму классов.
4. Постройте диаграммы последовательностей (при необходимости диаграммы состояний и деятельности) для наиболее сложных процессов проекта.
5. Разработайте прототип графического пользовательского интерфейса. На основании диаграммы использования протестируйте функциональность UI.
6. Выберите и спроектируйте базу данных для разрабатываемого ПС.
7. Выберите стратегию валидации данных. Используйте методики проверки данных для защиты своей системы. Определите подходящую стратегию для обработки и протоколирования исключений.

Глава 7. ЭТАП ЖИЗНЕННОГО ЦИКЛА «КОДИРОВАНИЕ»

Обычно считается, что этап жизненного цикла «Кодирование» является основным с точки зрения реализации проекта в целом. Однако скорее можно сказать, что этап важен с точки зрения качества и функциональности продукта. На этапе проектирования была разработана подробная модель ПС. На основе представленной модели можно сгенерировать текст ПС автоматически на заданном языке программирования или последовательно и аккуратно выполнить кодирование всех классов и компонентов. Параллельно могут вестись работы по наполнению баз данных, дизайну интерфейса и другим задачам, напрямую не связанным с программированием. В связи с этим может потребоваться создать дополнительный инструментарий, осуществляющий тестирование модулей, ввод данных и т. д.

После кодирования выполните рефакторинг кода. Рефакторинг представляет собой перестройку кода без внешнего изменения функциональности. Часто программисты пишут не идеальный, а достаточно хороший код и только позже, когда видят дублирование кода, запутанную логику, длинные методы, начинают его улучшать.

К основным элементам рефакторинга относятся собственно каталог рефакторингов и принципы, которые можно сформулировать так:

1) вносить изменения небольшими порциями;

2) выполнять тестирование до рефакторинга и после;

3) если нужно внести новую функциональность, но это сложно (требуется рефакторинг), то лучше сначала сделать рефакторинг, а потом писать новый код.

Также следует отметить появление автоматизированных средств Р.

7.1. Обеспечение сопровождаемости программного средства

Обеспечение сопровождаемости ПС сводится к обеспечению изучаемости ПС и его модифицируемости [28].

Изучаемость (подкритерий качества) ПС определяется как С-документированность, информативность, структурированность,

понятность и удобочитаемость. При окончательном оформлении текста ПС целесообразно придерживаться следующих рекомендаций, определяющих практически оправданный стиль программирования [28]:

- ✔ добавляйте к тексту модуля комментарии, проясняющие и объясняющие особенности принимаемых решений; по возможности, включайте комментарии (хотя бы в краткой форме) на самой ранней стадии разработки кода;
- ✔ используйте осмысленные (мнемонические) и устойчиво различимые имена (оптимальная длина имени – 4–12 литер, цифры – в конце), не применяйте сходные имена;
- ✔ соблюдайте осторожность в использовании констант (уникальная константа должна иметь единственное вхождение в текст модуля: при ее объявлении или, в крайнем случае, при инициализации переменной в качестве константы);
- ✔ не бойтесь использовать необязательные скобки – они обходятся дешевле, чем ошибки;
- ✔ размещайте не больше одного оператора в строке; для прояснения структуры модуля используйте дополнительные пробелы (отступы) в начале каждой строки, этим обеспечивается удобочитаемость текста модуля.

Структурированность текста существенно упрощает его понимание. Удобочитаемость текста может быть обеспечена автоматически путем применения специального программного инструмента – форматера.

Расширяемость обеспечивается возможностями автоматически настраиваться на условия использования ПС по информации, задаваемой пользователем. К таким условиям относятся прежде всего конфигурация компьютера, на котором будет применяться ПС (в частности, объем и структура его памяти), а также требования конкретного пользователя к функциональным возможностям.

7.2. Задание

1. Сгенерируйте (или напишите) код программ для решения поставленной задачи.
2. Выполните отладку программных модулей и компонентов.
3. Получите результаты работы.
4. Добавьте комментарии к листингам программы.
5. Разработайте руководство пользователя.

Глава 8. ЭТАП ЖИЗНЕННОГО ЦИКЛА «ТЕСТИРОВАНИЕ»

8.1. Тестирование программного средства

При тестировании проводится всестороннее исследование функциональности разработанных модулей и системы в целом, степени, в которой они отвечают разработанным требованиям. Тестирование системы в целом показывает степень готовности программного продукта. В связи с этим имеет смысл накладывать окончание предыдущего этапа ЖЦ с данным [29].

Существует несколько видов тестирования. Начальное тестирование проводится непосредственно разработчиками для того, чтобы убедиться, что ПС работает в соответствии с документацией.

Функциональное тестирование выполняется с целью более глубокой проверки соответствия функциональности ПС. Основной задачей такого вида тестирования является проверка устойчивости, корректности и, возможно, безопасности системы.

Нагрузочное тестирование проверяет работоспособность ПС при повышенных нагрузках, в экстремальных условиях: нехватка оперативной или внешней памяти, потери канала связи или соединения с другим приложением, работа с поврежденными файлами и т. д. При этом программа должна проявлять устойчивость и безопасность.

Тестирование пользовательского интерфейса может быть разделено на две части: проверка функциональности интерфейса и проверка удобства интерфейса. Тестирование функциональности пользовательского интерфейса проводится с целью определения соответствия действий, привязанных к элементам управления, требованиям, заявленным в документации. Тестирование удобства пользовательского интерфейса выполняется скорее инженерными психологами, чем тестировщиками. Здесь психологи проверяют, насколько приятно и удобно пользоваться данной программой, очевиден ли ее интерфейс, нельзя ли его улучшить [30].

Наиболее общим является регрессионное тестирование, состоящее из всех тестов, которые программа проходила до сих

пор. Задачей регрессионного тестирования является проверка функциональности уже включенных модулей. Тест необходим в связи со сложным взаимовлиянием модулей – изменение поведения одного из них может повлечь изменение поведения других, причем не всегда в лучшую сторону. Вновь проведенные тесты также включаются в систему регрессионных. Модули, которые не прошли тестирование, отправляются на доработку.

Подобное тестирование необходимо, так как изменение функционирования некоторого модуля может повлиять на работу других, связанных с ним, модулей, зачастую – многочисленных.

Кроме того, тестирование делят на тестирование методом черного (функциональное тестирование), серого и белого ящиков (структурное тестирование). В первом случае тестировщик не имеет никакой информации о структуре тестируемой системы и описывает только симптомы неисправности, при этом он может высказывать предположения о причинах возникновения неисправности.

В последнем случае тестировщик обладает полным исходным кодом программы и может указывать конкретные строки кода, вызвавшие сбой. Это типично для юнит-тестирования (unit testing), при котором тестируются только отдельные части системы. Оно обеспечивает то, что компоненты конструкции работоспособны и устойчивы до определенной степени.

Тестирование методом серого ящика является промежуточным вариантом – тестировщик имеет представление о составе модулей тестируемой системы, однако не имеет информации об их внутренней структуре или особенностях реализации.

План тестирования

Для проведения тестирования создается план (test plan), в котором описываются стратегии, ресурсы и график тестирования отдельных компонент и системы в целом. В плане отмечаются работы, места теста в каждом процессе, степень покрытия программы тестами и процент тестов, которые выполняются со специальными результатами [31].

Тестовые инженеры создают множество тестовых сценариев (test cases), каждый из которых проверяет результат взаимодействия между актером и системой на основе определенных пред- и постусловий использования таких сценариев. Сценарии в основном относятся к тестированию по типу белого ящика

и ориентированы на проверку структуры и операций интеграции компонентов системы.

Документирование результатов тестирования включает описание:

- задач, назначения и содержания ПС, а также описание функций соответственно требованиям заказчика;
- технологии разработки системы;
- планов тестирования различных объектов, необходимых ресурсов, соответствующих специалистов для проведения тестирования и технологических способов;
- тестов, контрольных примеров, критериев и ограничений оценки результатов программного продукта, а также процесса тестирования;
- учета процесса тестирования, составление отчетов об аномальных событиях, отказах и дефектах в итоговом документе системы.

Отслеживание ошибок

Системы по defect-tracking получили широкое распространение в последнее время. Суть таких инструментов состоит в том, чтобы увеличить контроль и довести до автоматизма процесс обнаружения, исправления и тестирования ошибок в ПО.

Разработчик, обнаруживший ошибку (владелец), дает ей кодовое имя, описывает ее симптомы и ситуацию, которая привела к возникновению ошибки. Ошибке назначается приоритет, сроки и ответственный за исправление проблемы. Назначенный работник решает проблему и сообщает об этом владельцу ошибки. Тот удостоверяется, что вопрос решен и «закрывает» задачу. Системы, интегрирующие в себе весь процесс накопления данных об ошибках, уведомлениях участников процесса, получения отчетности, значительно увеличивают эффективность работы.

Последующий анализ ошибок, функции их распределения по времени, сортировка с учетом приоритета, критичности, частоты повторения, среднему времени исправления помогут в дальнейшем если не предотвратить их появление, то хотя бы понять причины их возникновения (и принять соответствующие меры).

В целом разработчики различают дефекты программного обеспечения и сбои. В случае сбоя программа ведет себя не так, как ожидает пользователь. Дефект – это ошибка/неточность, которая может быть (а может и не быть) следствием сбоя.

Набор данных

Для проведения качественного процесса тестирования важно обладать полным набором данных, представляющих различные варианты ситуации автоматизируемого процесса. Чаще всего такой набор данных предоставляет конечный пользователь системы, но во многих случаях разработчики сами могут дополнить предоставленный набор своими вариантами исключительных ситуаций.

Например, пользователю не придет в голову проверить реакцию системы на отключение электричества, обрыв связи или внезапную недоступность СУБД посередине бизнес-транзакции.

Следует отметить, что тестирование нужно проводить на «эталонных» наборах данных, а не на данных разработчиков. Иными словами, бизнес-сущности должны «по всем правилам» быть занесены в систему через соответствующую функциональность. Часто сгенерированные данные или занесенные вручную в БД – это не те же самые данные, которые позволяет создать система «официальным» способом. В отношении качества данных, в первую очередь, различия проявляются в значениях по умолчанию отдельных полей. Иногда есть различия в том, что позволяет ввести СУБД и программа.

Автоматизация тестов

Большим преимуществом при выполнении тестирования является наличие средств, позволяющих автоматизировать процесс тестирования. После добавления новых функций в систему очень важно убедиться, что не нарушена работа существующих функций, и для этого подходит система, самостоятельно «прогоняющая» большинство контрольных примеров без участия человека.

Среди средств автоматизации подготовки тестов и анализа их результатов можно выделить:

1) генераторы случайных тестов в заданных областях входных данных;

2) отладчики (для локализации ошибок);

3) анализаторы динамики (profilers). Обычно входят в состав отладчиков и применяются для проверки соответствия тестовых наборов структурным критериям тестирования;

4) средства автоматической генерации структурных тестов.

Тестирование компонентов

Компоненты тестируют по отдельности. Самый простой способ – написать программу, вызывающую все функции компонента [32]. Для этого используют любые языки сценариев и средства быстрой разработки приложений (RAD – Rapid Application Development) типа Microsoft VBScript и Microsoft Visual Basic.

Если компонент не работает, как должно, его нужно запустить в отладчике и изучить каждую строку его кода. Обязательно проверяют и все возвращаемые методами значения и уже по ним определяют тип ошибки.

Если компоненты, обращающиеся к данным, не могут получить доступ к источникам информации, для выявления причин нужно использовать средства СУБД и ODBC. Если же вы работаете с SQL Server, тестировать подключения к базам данных, выполнение запросов и т. п. следует с помощью утилиты SQL Enterprise Manager. Для изучения операций над базами данных применяют утилиту SQL Trace.

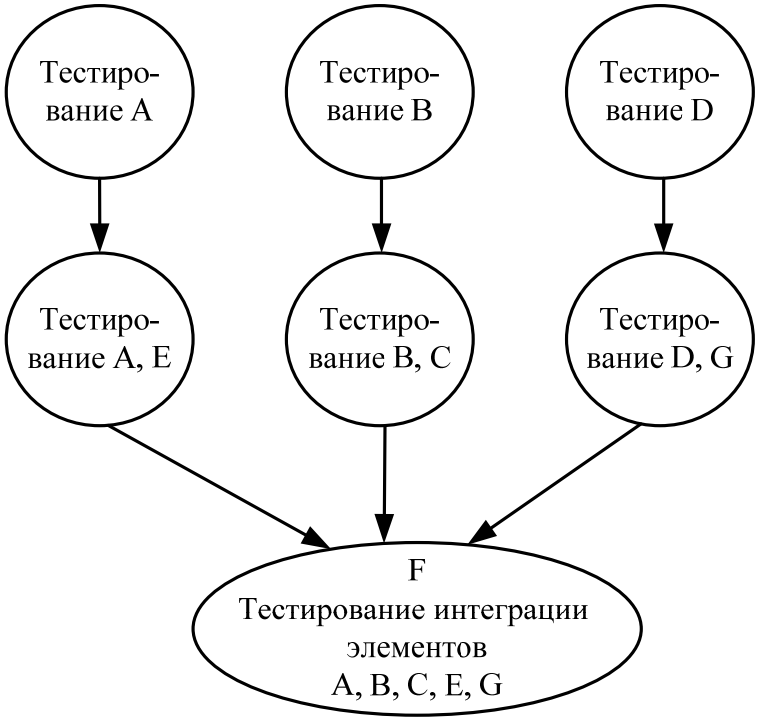
Следующий шаг – проверка интерфейсов для последующей их интеграции, суть которой заключается в анализе выполнения операторов вызова.

При этом могут возникать ошибки в случае неправильного задания параметров в операторах вызова или при вычислениях процедур или функций. Возникающие ошибки в связях устраняются, а затем повторно проверяется связь с компонентой в виде троек: компонента – интерфейс – компонента (рисунок).

Следующим шагом тестирования системы является проверка функционирования системы с помощью тестов проверки функций и требований к ним. После проверки системы на функциональных тестах идет проверка системы на исполнительных и испытательных тестах, подготовленных согласно требованиям к ПО, аппаратуре и выполняемым функциям. Испытательному тесту предшествует верификация и валидация ПО.

Тестирование, основанное на ошибках и сценариях

Цель тестирования, основанного на ошибках, – проектирование тестов, ориентированных на обнаружение предполагаемых ошибок [32]. Разработчик выдвигает гипотезу о предполагаемых ошибках. Для проверки его предположений разрабатываются тестовые варианты.



Интеграционное тестирование компонент

Тестирование, основанное на ошибках, оставляет в стороне два важных типа ошибок: некорректные спецификации, взаимодействия между подсистемами.

Ошибка из-за неправильной спецификации означает, что продукт не выполняет то, чего хочет заказчик. От этого страдает качество – соответствие требованиям заказчика. Ошибки, связанные с взаимодействием подсистем, происходят, когда поведение одной подсистемы создает предпосылки для отказа другой подсистемы.

Тестирование, основанное на сценариях, ориентировано на действия пользователя, а не на действия программной системы [32]. Это означает фиксацию задач, которые выполняет пользователь, а затем применение их в качестве тестовых вариантов. Задачи пользователя фиксируются с помощью элементов Use Case.

Сценарии элементов Use Case обнаруживают ошибки взаимодействия, каждая из которых может быть следствием многих причин. Поэтому соответствующие тестовые варианты более сложны и лучше отражают реальные проблемы, чем тесты, основанные на ошибках. С помощью одного теста, базирующегося на сценарии, проверяется множество подсистем.

Рассмотрим, например, проектирование тестов, основанных на сценариях, для текстового редактора. Рабочие сценарии опишем в виде спецификаций элементов Use Case.

Элемент Use Case: исправлять черновик.

Предпосылки: обычно печатают черновик, читают его и обнаруживают ошибки, которые не видны на экране. Данный элемент Use Case описывает события, которые при этом происходят:

1. Печатать весь документ.
2. Прочитать документ, изменить определенные страницы.
3. После внесения изменения страница перепечатывается.
4. Иногда перепечатываются несколько страниц.

Этот сценарий определяет как требования тестов, так и требования пользователя. Пользователю нужны метод для печати отдельной страницы, метод для печати диапазона страниц.

В ходе тестирования проверяется редакция текста как до печати, так и после печати. Разработчик теста может надеяться обнаружить, что функция печати вызывает ошибки в функции редактирования. Это будет означать, что в действительности две программные функции зависят друг от друга.

Тестирование содержания класса

Описываемые ниже способы ориентированы на отдельный класс и операции, которые инкапсулированы классом.

Стохастическое тестирование класса. При стохастическом тестировании исходные данные для тестовых вариантов генерируются случайным образом [32].

Например, класс *Файл*, который имеет следующие операции: *Открыть*, *Изменить*, *Создать*, *Закрыть*, *Удалить*.

Каждая из этих операций применяется при определенных ограничениях: файл должен быть открыт или создан перед применением других операций; файл должен быть закрыт после завершения всех операций.

Даже с этими ограничениями существует множество допустимых перестановок операций. Минимальная работа экземпляра *Файл* включает следующую последовательность операций:

Открыть → *Изменить* → *Закрыть*.

Здесь стрелка обозначает операцию следования. Эта последовательность является минимальным тестовым вариантом для *Файла*.

Тестирование разбиений на уровне классов. Тестирование разбиений уменьшает количество тестовых вариантов, требуемых для проверки классов (тем же способом, что и разбиение по эквивалентности для стандартного ПО). Области ввода и вывода разбивают на категории, а тестовые варианты разрабатываются для проверки каждой категории.

Обычно используют одну из трех категорий разбиения [32]. Категории образуются операциями класса.

Первый способ – ***разбиение на категории по состояниям***. Основывается на способности операций изменять состояние класса.

Например, для класса *Файл* операции *Изменить*, *Удалить* изменяют его состояние и образуют первую категорию. Операции *Проверить CRC*, *Копировать* не меняют состояние *Файла* и образуют вторую категорию. Проектируемые тесты отдельно проверяют операции, которые изменяют состояние, а также те операции, которые не изменяют состояние. Таким образом:

Тестовый вариант 1 (ТВ 1):
Открыть → *Вставить* → *Закрыть*.

Тестовый вариант 2 (ТВ 2):
Открыть → *Просмотреть* → *Закрыть*.

ТВ 1 изменяет состояние объекта, в то время как *ТВ 2* не меняет состояние.

Второй способ – ***разбиение на категории по свойствам***. Основывается на свойствах, которые используются операциями. В классе *Файл* для определения разбиений можно использовать свойства CRC и ограничение доступа. Например, на основе ограничения доступа операции подразделяются на три категории:

- операции, которые используют ограничение доступа;
- операции, которые изменяют ограничение доступа;
- операции, которые не используют и не изменяют ограничение доступа.

Для каждой категории создается тестовая последовательность.

Третий способ – ***разбиение на категории по функциональности***. Основывается на общности функций, которые выполняют операции. Например, операции в классе *Файл* могут быть разбиты на категории: операции инициализации (*Открыть*, *Создать*); запросы (*Свойства*, *Ограничить доступ*); операции завершения (*Закрыть*).

Тестирование на основе состояний

В качестве источника исходной информации используют диаграммы схем состояний, фиксирующие динамику поведения класса. Данный способ позволяет получить набор тестов, проверяющих поведение класса и тех классов, которые сотрудничают с ним [33].

Окружение программы

Под окружением понимается та среда, в которой работает приложение. Это, во-первых, аппаратное обеспечение – компьютер (разной степени мощности), монитор (разной диагонали и разрешения), периферийные устройства, комплектующие и вообще все, что имеет отношение к «железу». Во-вторых, это программное обеспечение – операционная система и патчи (сервис-паки к ней), драйвера, JVM, офисные пакеты, СУБД и доступ к данным, средства получения отчетности, коммуникации и обслуживания и т. п.

Необходимо обеспечить всестороннее тестирование системы во всевозможных программных и аппаратных конфигурациях. Если же программа предназначена для работы со строго определенной версией того или иного компонента, нужно максимально облегчить доступ пользователя к нему. Например, если приложение для работы требует установленного компонента *X* версии *Y*, то лучше включить его в инсталляционный пакет, чем ожидать, что пользователь установит его корректно самостоятельно.

Отладка программ

Локализация и исправление ошибок называется отладкой [34]. Отладка – это процесс обнаружения причин возникновения ошибки и ее последующего исправления (в отличие от тестирования, являющегося процессом обнаружения самого факта существования ошибки). Отладка включает в себя элементы тестирования и разработки. На некоторых проектах отладка может занимать до 50% всего времени разработки. Как и тестирование, отладка не является способом улучшения качества ПО. Отладка – это всего лишь способ исправления дефектов в программе. Качество программ должно обеспечиваться аккуратным анализом требований или прототипированием, грамотным проектированием и использованием лучших практик кодирования.

8.2. Задание

1. На основании спецификации отлаживаемого модуля подготовьте тесты для каждой возможности и каждой ситуации, для каждой границы областей допустимых значений всех входных данных, для каждой области изменения данных, для каждой области недопустимых значений всех входных данных и каждого недопустимого условия.
2. Проведите функциональное тестирование компонентов и модулей.
3. Выполните тестирование пользовательского интерфейса.
4. Проведите регрессионное тестирование.
5. Результаты оформите в виде таблиц.

Глава 9. ЖИЗНЕННЫЕ ЦИКЛЫ «ВНЕДРЕНИЕ» И «СОПРОВОЖДЕНИЕ»

9.1. Жизненный цикл «Внедрение»

Этап внедрения включает подготовку качественной документации, создание инсталляционного пакета и процесс обновления компонентов системы, своевременное реагирование на обнаруженные проблемы посредством патчей и хотфиксов, хранение и контроль версий успешных и тестовых билдов продукта, ведение истории внедрения.

Документация может содержать следующие компоненты.

Текст программы должен содержать необходимые комментарии.

Описание программы – сведения о логической структуре и функционировании программы.

Описание применения должно отражать сведения о назначении программного обеспечения, области использования, применяемых методах, классе решаемых задач, ограничениях для использования, минимальной конфигурации технических средств.

Руководство пользователя, как правило, состоит из следующих разделов: общие сведения о программном продукте; описание установки; описание запуска; инструкции по работе (или описание пользовательского интерфейса); сообщения пользователю.

Раздел «Установка» обычно содержит подробное описание действий по установке программного продукта и сообщений, которые при этом могут быть получены.

9.2. Жизненный цикл «Сопровождение»

Сопровождение программы – поддержка работоспособности программы, переход на ее новые версии, внесение изменений, исправление ошибок и т. д.

Сопровождение ПО также следует предусмотреть еще на этапе проектирования, так как его стоимость может существенно повлиять на оценку прибыльности проекта. В ходе использования системы

осуществляется информационная поддержка пользователей, их обучение, устранение обнаруженных неисправностей, изменение функциональности и состава системы в соответствии с вновь появившимися требованиями. Если оговаривается определенный сервис, создается служба технической поддержки, которая осуществляет обслуживание комплекса, отвечает на возникающие вопросы. При обнаружении неисправностей или несоответствия заявленной функциональности производится замена соответствующих модулей вновь разработанными. В случае, если заказчик решает изменить функциональность системы, выполняется ее модернизация. При этом работы осуществляются начиная с первого пункта.

9.3. Задание

1. Выполните подготовку раздела «Описание структуры программного средства». Он должен содержать информацию о структуре и конкретных компонентах программного обеспечения, в том числе краткое описание его функций, реализованных методов, алгоритмов, а также обоснование принятых технических и технико-экономических решений.

2. Выполните подготовку раздела «Руководство пользователя». В нем опишите режимы работы, форматы ввода-вывода информации и возможные настройки. Опишите последовательность выполнения работы, средства защиты, разработанные в данном ПС, реакцию программы на неверные действия пользователя.

Глава 10. ДИЗАЙН ПОЛЬЗОВАТЕЛЬСКОГО ИНТЕРФЕЙСА

10.1. Введение

Пользовательский интерфейс (UI) – это часть приложения, получающая информацию от пользователя и отображающая ее.

В силу большого разнообразия пользователей и видов ПС существует множество различных стилей пользовательских интерфейсов, при разработке которых могут использоваться разные принципы и подходы. Однако существуют важнейшие принципы, которые следует соблюдать всегда [30]:

- UI должен базироваться на терминах и понятиях, знакомых пользователю;
- UI должен быть единообразным;
- UI должен позволять пользователю исправлять собственные ошибки;
- UI должен позволять пользователю получать справочную информацию: как по его запросу, так и генерируемой ПС.

В настоящее время широко распространены командные и графические пользовательские интерфейсы.

Командный пользовательский интерфейс дает пользователю возможность обращаться к ПС с некоторым заданием (запросом), представляемым некоторым текстом (командой) на специальном командном языке (языке заданий). Командный пользовательский интерфейс обычно выбирают только опытные пользователи. Такой интерфейс позволяет им осуществлять быстрое взаимодействие с компьютером и объединять команды в процедуры и программы.

Графический пользовательский интерфейс (GUI) предоставляет пользователю возможности:

1) обращаться к ПС путем выбора на экране подходящего графического или текстового объекта;

2) получать от ПС информацию на экране в виде графических и текстовых объектов;

3) осуществлять прямые манипуляции с графическими и текстовыми объектами, представленными на экране;

4) размещать на экране множество различных окон, в которые можно выводить информацию независимо;

5) использовать экранный указатель для выбора объектов (или их элементов), размещенных на экране; экранный указатель перемещается с помощью клавиатуры или мыши.

Достоинством графического пользовательского интерфейса является возможность создания удобной и понятной пользователю модели взаимодействия с ПС (панель управления, рабочий стол и т. п.) без необходимости изучения какого-либо специального языка. Однако его разработка требует больших трудозатрат. Кроме того, возникает серьезная проблема по переносимости ПС на другие операционные системы, так как графический интерфейс существенно зависит от возможностей (графической пользовательской платформы), предоставляемых операционной системой для его создания.

В основе современного графического пользовательского интерфейса лежат две основные концепции.

Первой из них является понятие программы, управляемой данными. Как правило, эта концепция практически реализуется через механизм сообщений. Внешние устройства (клавиатура, мышь, таймер) посылают сообщения модулям программы о наступлении тех или иных событий (например, при нажатии клавиши или передвижении мыши). Поступающие сообщения попадают в очередь сообщений, откуда извлекаются прикладной программой.

Таким образом, программа не должна все время опрашивать мышь, клавиатуру и другие устройства в ожидании, не произошло ли чего-нибудь, заслуживающего внимания. Когда событие произойдет, программа получит извещение об этом с тем, чтобы надлежащим образом его обработать. Поэтому программы для таких сред обычно представляют собой цикл обработки сообщений: извлечь очередное сообщение, обработать его, если оно интересно, либо передать стандартному обработчику сообщений, обычно входящему в систему и представляющему собой стандартные действия системы в ответ на то или иное событие.

Сообщения могут посылаться не только устройствами, но и отдельными частями программы (в частности, возможна посылка сообщения себе). Так, один модуль может послать сообщение другому модулю, или меню посылает сообщение о выборе определенного пункта. При этом существует также способ прямой

посылки сообщения, минуя очередь, когда непосредственно вызывается обработчик сообщений адресата.

Второй основополагающей концепцией является понятие окна как объекта. Окно – это не просто прямоугольная область на экране, это и программа (процедура, функция), способная выполнять различные действия, присущие окну. Одним из важных действий является реагирование на поступающие сообщения и посылка сообщений другим объектам.

Одной из главных функций окна является перерисовка содержания окна. Любое окно должно уметь при получении соответствующего запроса перерисовать себя (или свою часть) на экране. Перерисовка может реализовываться или как реакция на специальное сообщение, или как виртуальная функция (при использовании объектно-ориентированных языков). В состав любой GUI обязательно входит достаточно мощный графический модуль, обеспечивающий выполнение всех основных графических операций и поддерживающий отсечение изображения по заданной (в том числе и довольно сложной) области отсечения. За счет этого реализуется возможность перерисовки фрагмента окна – устанавливается область отсечения, совпадающая с требуемым фрагментом, а затем выполняется запрос на перерисовку.

10.2. Дизайн пользовательского интерфейса

Большинство принципов проектирования пользовательского интерфейса совпадают с правилами создания художественного произведения. Ведь такие понятия, как композиция, цвет и т. д., одинаковы и для экрана компьютера, и для листа бумаги, и для холста.

Современные средства программирования позволяют создавать пользовательский интерфейс, просто перетаскивая элементы управления на форму или Web-страницу. Однако советуем вам не отказываться от предварительного планирования. Сначала лучше прорисуйте пользовательский интерфейс на бумаге, определите необходимые элементы, их относительную важность и взаимосвязи (как это указано в предыдущем разделе).

Стили интерфейса

Существует три основных стиля и один дополнительный [30]:

- однодокументный интерфейс (SDI – Single-Document Interface);

✔ многодокументный интерфейс (MDI – Multiple-Document Interface). В нем разрешается одновременно открывать несколько документов, каждый в своем окне. MDI-приложения характеризуются наличием в меню команды Window для переключения между окнами;

✔ интерфейс в стиле Explorer – это окно, состоящее из двух панелей, в одной из которых отображается дерево (слева), а другая представляет собой область отображения текущего элемента дерева (справа). Таков интерфейс *Проводника* (приложения Explorer) из состава Microsoft Windows;

✔ отчет. Этот стиль обычно считается дополнительным. Информация в таком интерфейсе отображается в любом формате – графическом, табличном, текстовом или комбинированном.

При выборе интерфейса нужно принять во внимание как назначение приложения, так и работу пользователей. Например, для простой программы «часы» лучше всего выбрать стиль SDI, так как маловероятно, что кому-то потребуется сразу несколько часов одновременно. Для приложения, обрабатывающего страховки, лучше применить MDI-интерфейс, поскольку возможна работа сразу с несколькими документами, например для их сравнения [36]. Интерфейс в стиле Explorer используется во многих новых приложениях, так как позволяет искать и просматривать множество документов, изображений или файлов. Отчеты представляют собой снимок состояния информации в определенный момент времени и обычно не предполагают непосредственного взаимодействия пользователя с отображаемой информацией.

Диалоговые окна

Большинство приложений взаимодействуют с пользователем. Для запроса данных, необходимых для работы программы, служат диалоговые окна. Это форма специального типа, которая отображает информацию и, как правило, требует в ответ каких-либо действий со стороны пользователя. Обычно, чтобы продолжить работу с приложением, диалоговое окно нужно закрыть [31].

Существует принцип – чем больше слов в диалоговом окне, тем меньшее число пользователей их читают.

Элементы управления

В таблице приведены часто используемые элементы управления и их назначение.

Элементы управления

Элемент управления	Описание
Метка (label)	Только отображает текст
Текстовое поле (text box)	Текст, который пользователь может редактировать. Обычно применяется для ввода информации, например пароля пользователя
Кнопка (command button)	Кнопка (обычно прямоугольной формы), выполняющая команду пользователя по ее щелчку
Флажок (check box)	Набор вариантов, из которых можно выбрать один или несколько. Показывает состояние какого-либо условия – включено/выключено, правда/ложь, да/нет. Если флажки сгруппированы, они независимы друг от друга – пользователь может выбрать любое число вариантов
Переключатель (option button)	Набор вариантов, из которых можно указать только один. Переключатели работают только в группе; выбор одного из них автоматически отключает остальные
Список (list box)	Прокручиваемый список вариантов. Обычно он отображается вертикально в одну колонку, но можно применять списки и из нескольких колонок. Если число элементов списка превышает число отображаемых элементов, должны быть предусмотрены полосы прокрутки. Из списка можно выбрать несколько элементов, удерживая клавишу *Ctrl*. Кроме того, список бывает раскрывающимся
Поле со списком (combo box)	Прокручиваемый список, объединенный с текстовым полем. Похож на обычный список, но пользователю разрешается как вводить информацию в текстовое поле, так и выбирать элемент из списка
Ползунок (slider control)	Позволяет указать значение на шкале. Такими элементами управления обычно задают громкость звука или регулируют цвета изображения
Индикатор (progress indicator)	Показывает, какая часть процесса выполнена к настоящему моменту. Появление на экране этого элемента управления говорит о том, что приложение осуществляет какую-то работу. Если для этого требуется много времени, следует предусмотреть вывод времени, необходимого для завершения работы

Командные кнопки

Это самый простой элемент управления, который имеет больше всего тонкостей. Как известно, чем больше кнопка, тем легче попасть на нее курсором. Однако, помимо простоты нажатия на кнопку, есть другая составляющая проблемы: пользователю

должно быть трудно нажать не на ту кнопку. Добиться этого можно либо изменением состояния кнопки при наведении на нее курсором, либо установлением пустого промежутка между кнопками.

Кнопка должна (или не должна) быть пользователем нажата. Соответственно, пользователю нужно как-то сигнализировать, что кнопка нажимаема. Лучшим способом такой индикации является придание кнопке псевдообъема, т. е. визуальной высоты.

Кнопка должна как-то показывать пользователям свои возможные и текущие состояния. Не должно быть дублирования состояний. Также очень важно делать заблокированные состояния действительно заблокированными.

Командные кнопки необходимо снабжать названиями, выраженными в виде глаголов в форме инфинитива. Таким образом, следует избегать создания кнопок с ничего не говорящим текстом, поскольку такой текст не сообщает пользователям, что именно произойдет после нажатия кнопки.

Подписи к стоящим параллельно кнопкам лучше стараться делать примерно одинаковой длины. Все подписи обязаны быть позитивными (т. е. не должны содержать отрицания). Если подпись не помещается в одну строку, выравнивают индикатор кнопки (кружок или квадрат) по первой строке подписи (рис. 10.1).

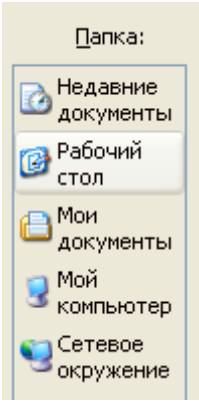


Рис. 10.1. Пример
интерфейса кнопок

Списки

Ширина списка как минимум должна быть достаточной для того, чтобы пользователь мог определить различия между элементами. В идеале, конечно, ширина всех элементов должна быть меньше ширины списка, но иногда это невозможно. В таких

случаях не стоит добавлять к списку горизонтальную полосу прокрутки, лучше урезать текст элементов.

Для этого нужно определить самые важные фрагменты текста, после чего все остальное заменить отточием (...). Поскольку важно максимально ускорить работу пользователей, необходимо сортировать элементы (рис. 10.2). Идеальным вариантом является сортировка по типу элементов. Если же элементы однотипны, их следует сортировать по алфавиту, причем списки с большим количеством элементов полезно снабжать дополнительными элементами управления, влияющими на сортировку или способ фильтрации элементов. Если можно определить наиболее популярные значения, их можно сразу расположить в начале списка, но при этом придется вставлять в список разделитель, а в систему – обработчик этого разделителя.

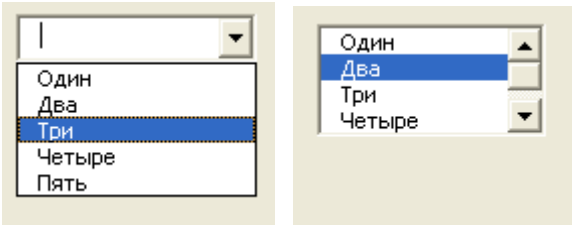


Рис. 10.2. Пример списков

Если используется раскрывающийся список, в котором помимо собственно элементов есть «метаэлемент», включающий все элементы из списка, то ему часто не дают названия, оставляя строку списка пустой, что неправильно. Такой элемент нужно снабжать названием, например «*Все значения*» или «*Ничего*».

Другим, более сложным вариантом списка является пролистываемый список (рис. 10.3). По вертикали в список должно помещаться как минимум четыре строки, а лучше восемь. Напротив, список по высоте больший, нежели высота входящих в него элементов, и, соответственно, содержащий пустое место в конце, смотрится неряшливо.

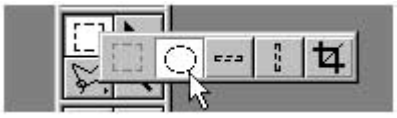


Рис. 10.3. Пример
пролистываемого списка

Список единственного выбора является промежуточным вариантом между группой радиокнопок и раскрывающимся списком. Он меньше группы радиокнопок с аналогичным числом элементов, но больше раскрывающегося списка. Следовательно, использовать его стоит только в условиях «ленивой экономии» пространства экрана.

Меню

Существуют несколько различных таксономий меню. Первая таксономия делит меню на два типа: статические меню (постоянно присутствующие на экране) и динамические меню (пользователь должен вызвать меню, чтобы выбрать какой-либо элемент). На эффективность меню наибольшее влияние оказывают устройство отдельных элементов и их группировка. Несколько менее важны другие факторы, такие как выделение элементов и стандартность меню.

Самым важным свойством хорошего элемента меню является название его элементов. Название должно быть самым эффективным из возможного. Так, например, главное слово в элементе должно стоять первым. Короткий текст элемента, без сомнения, быстро читаясь, совершенно необязательно быстро распознается. Поэтому не стоит сильно сокращать текст элемента: выкидывать нужно все лишнее, но не более.

Второй составляющей качества меню является группировка его элементов. В большинстве меню группировка оказывает не меньшее значение при поиске нужного элемента, нежели само название элемента, просто потому, что даже идеальное название не сработает, если элемент просто нельзя найти. Чтобы уметь эффективно группировать элементы в меню, нужно знать ответы на три вопроса: зачем элементы в меню нужно группировать, как группировать элементы и как разделять группы между собой. Наличие явных разделителей многократно облегчает построение ментальной модели, поскольку не приходится гадать, как связаны между собой элементы. Для разграничения групп традиционно используют полоски. Это надежное, простое решение, однако с дизайнерской точки зрения полоски плохи, поскольку представляют собой визуальный шум. Гораздо правильнее, но и труднее, использовать только визуальные паузы между группами.

Третье, наличие многих уровней вложенности в меню вызывает так называемые «каскадные ошибки»: выбор неправильного

элемента верхнего уровня неизбежно приводит к тому, что все следующие элементы также выбираются неправильно. При этом широкие меню больше нравятся пользователям. Поэтому большинство разработчиков интерфейсов стараются создавать широкие, а не глубокие меню.

Раскрывающиеся меню довольно удобны для использования. Однако с уровнем вложенности элементов, большим трех, не применяются (рис. 10.4) [35].

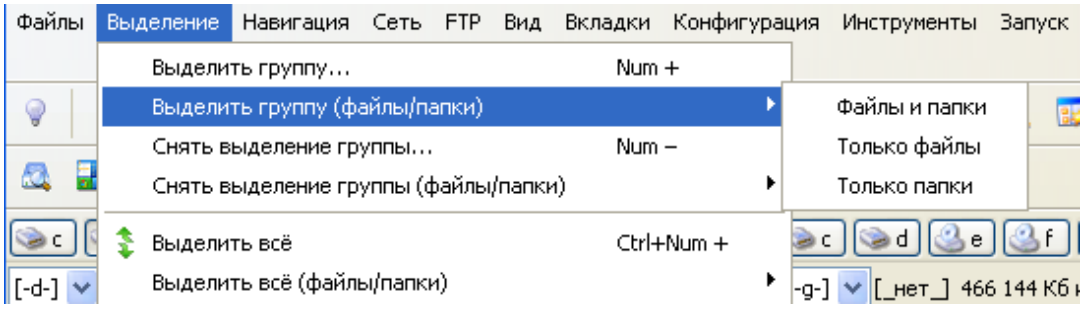


Рис. 10.4. Пример каскадного меню

Контекстные меню

Поскольку основной причиной появления контекстных меню является стремление максимально повысить скорость работы пользователей, на их размер и степень иерархичности накладываются определенные ограничения. Если меню будет длинным, пользователям придется сравнительно долго возвращать курсор на прежнее место, так что привлекательность нижних элементов окажется под вопросом. Поэтому лучше сокращать размер контекстных меню до разумного минимума (порядка семи элементов).

К тому же не надо забывать, что главное меню не всегда перекрывает выделенный (актуальный) объект, а контекстное меню – почти всегда. В большинстве же случаев перекрытие актуального объекта нежелательно (сбивается контекст). В этой ситуации можно уменьшить размер меню в расчете, что маленькое меню будет перекрывать малое количество информации.

Делать иерархические контекстные меню можно, но необходимо сознавать, что вложенными элементами почти никто не будет пользоваться.

Последнее отличие контекстных меню от обычных заключается в том, что в них очень важен порядок следования элементов

(рис. 10.5). В главном меню не обязательно стремиться к тому, чтобы наиболее часто используемые элементы были самыми первыми – все равно курсор придется возвращать к рабочему объекту, так что разницы в дистанции перемещения курсора практически нет. В контекстном же меню ситуация обратная – чем дальше нужный элемент от верха меню, тем больше придется двигать курсор.

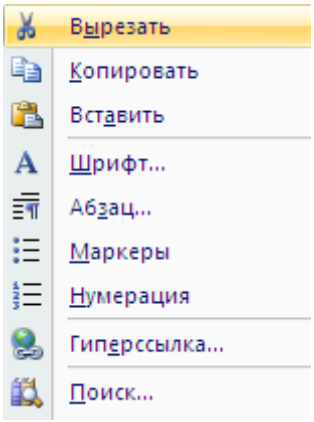


Рис. 10.5. Пример контекстного меню

Полосы прокрутки

Полосы прокрутки не столько помогают перемещаться по документу, сколько показывают то, что не весь документ помещается на экране. Один из шагов по улучшению полос прокрутки заключается в следующем – размер ползунка может быть сделан пропорциональным отношению видимой части документа ко всему его объему.

Таким образом, полосы прокрутки стали еще более бесполезны, поэтому относиться к ним надо не как к данности, но как к еще одному элементу управления, который можно использовать, а можно и не использовать. Если все-таки приходится оставлять полосы прокрутки, крайне желательно добиться нескольких свойств:

1) размер ползунка должен показывать общий объем пролистываемого документа;

2) стрелки на полосах должны быть спаренными, т. е. обе стрелки должны находиться рядом, а не на разных сторонах полоски;

3) если невозможно сделать динамическое изменение области просмотра при пролистывании, необходимо показывать текущее местоположение пользователя во всплывающей подсказке.

Следует обеспечить обработку погрешности перемещения курсора. Когда пользователь курсором перемещает ползунок, а смотрит в это время на документ, курсор может сойти с полосы. До определенного момента (смещение на 30–70 пикселей) система должна такое смещение игнорировать.

Вкладки

Теоретически число вкладок может быть сколь угодно большим (рис. 10.6). На практике оно ограничивается двумя факторами: во-первых, объемом памяти, а во-вторых, размером области, в которую ярлыки вкладок могут помещаться. Дело в том, что если ширина всех ярлыков будет больше ширины окна, то придется либо делать несколько строк ярлыков, либо скрывать часть из них, пока пользователь не нажмет специальную кнопку. И то и другое плохо. Во-первых, из-за большого количества мелких деталей (границ ярлыков) вся конструкция довольно медленно сканируется, т. е. трудно найти нужную вкладку. Во-вторых, при последовательном обращении к нескольким вкладкам из разных рядов эти ряды меняются местами, т. е. каждый раз нужно заново искать нужную вкладку. И то и другое крайне негативно сказывается на субъективном удовлетворении и скорости работы.

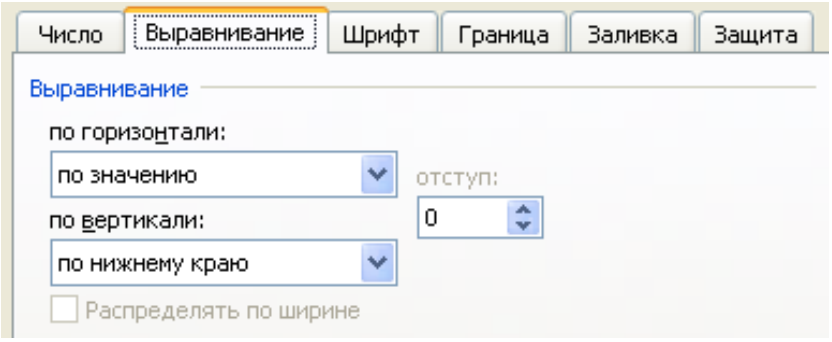


Рис. 10.6. Пример вкладок

Скрывать часть ярлыков тоже нехорошо. Предположим, что пользователь нажал на стрелку вправо, показывающую следующую часть ярлыков. Если при этом значительно пролистывать строку с ярлыками, пользователь будет полностью терять контекст. Если же пролистывать строку по одному элементу, контекст не потеряется, но перемещение между вкладками будет очень медленным.

Существует и третий способ решения проблемы – можно просто убрать вкладки, заменив их раскрывающимся списком. Этот способ тоже не совсем хорош, поскольку не слишком визуален и к тому же сравнительно медлителен.

Самым эффективным решением является комбинация второго и третьего способов: основные экраны реализуются в форме вкладок, а дополнительные вызываются через раскрывающийся список. Это позволяет обеспечить максимальное количество наглядности и скорости работы.

Курсоры

Правило использования курсоров звучит следующим образом: используйте индикацию видом курсора во всех случаях, когда вы можете сделать это эстетически привлекательно (рис. 10.7); в остальных случаях о курсорах даже и не думайте.

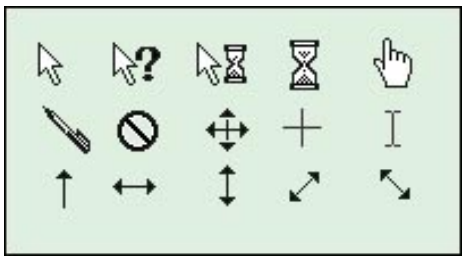


Рис. 10.7. Виды курсоров

Можно нарисовать и собственные курсоры, при этом следует придерживаться двух правил: избегать цвета и делать курсоры такими, чтобы активная точка курсора (которая указывает) была как можно ближе к верхнему левому краю курсора.

Звук

Звук, как и цвет, в интерфейсе не есть вещь абсолютно необходимая. Он практически не может делать что-либо полезное, помимо индицирования конца какого-либо процесса (благодаря которому пользователь освобождается от необходимости поминутно поглядывать на экран).

Во-первых, восприятие звуков у каждого человека уникально еще более, чем даже восприятие цвета. Если пользователям полезно давать возможность изменять цвета системы, то применительно к звукам правило звучит более жестко: обязательно позволяйте пользователю самому выбирать звуки (и уж как минимум отключать их вовсе).

Во-вторых, надо всегда помнить, что качественный и приятный фоновый звук чаще всего не воспринимается пользователями как благо, но зато при этом звуке субъективно улучшается изображение (на 20%). Это и есть главный аргумент за хороший звук [34].

Изображения и значки

Картинки и значки добавят привлекательности приложению, но обращаться с ними нужно осторожно. Изображения могут передать информацию вместо текста, но люди относятся к ним поразному. Панели инструментов со значками, изображающими различные функции, очень полезны, но если пользователи не смогут сразу же понять, что на них нарисовано, работа только замедлится. При выборе значков для панелей инструментов стоит придерживаться уже действующих стандартов. Например, во многих приложениях значок New File (Создать файл) – это лист бумаги с загнутым уголком. Хотя можно придумать и лучшую метафору для этой функции, другое изображение только запутает пользователя.

Важно учитывать и национальные традиции. При создании значков и изображений лучшее правило – простота. Сложные многоцветные картинки плохо отображаются на значке панели инструментов размером 16×16 пикселей.

Желательно готовить иконки трех стандартных размеров: 16×16 пикселей (16 цветов), 32×32 пикселей (16 цветов) и 48×48 пикселей (256 цветов).

Шрифты

Поскольку для передачи информации в основном применяется текст, выбор легко читаемого при разных разрешениях и на разных мониторах шрифта – важная часть проектирования пользовательского интерфейса. Лучше всего задействовать простые шрифты везде, где возможно. Декоративные шрифты, например Script, хорошо выглядят на бумаге, но не на экране. К тому же текст, написанный таким шрифтом малого размера, трудно читать.

Обычно в интерфейсе используют стандартные шрифты Windows: Arial, Times New Roman или System. В противном случае вместе с приложением придется распространять и применяемый шрифт, так как на компьютерах, где он не установлен, система заменит его на другой, что иногда сильно ухудшает внешний вид приложения. При выборе шрифта, как и многих других

элементов интерфейса, важна согласованность. Общая рекомендация – не использовать более двух шрифтов двух или трех размеров, иначе ваше приложение будет похоже на текст, который составлен из букв, вырезанных из разных газет.

Размещение элементов управления

В большинстве интерфейсов не все элементы равнозначны. Интерфейс нужно проектировать так, чтобы пользователь сразу же понимал, какие элементы важнее других. Поэтому расположение должно подчеркивать иерархию: важные элементы следует помещать на видное место, а маловажные или редко используемые – на менее заметное.

В большинстве языков мира текст читается слева направо и сверху вниз. Это верно и для экрана компьютера. Поэтому почти все пользователи первым делом обращают внимание на верхний левый угол экрана, следовательно, именно там стоит расположить самые важные элементы интерфейса.

Необходимо также объединить элементы управления в группы. Группировать элементы следует в соответствии с назначением. Например, кнопки перемещения по базе данных надо объединить, а не разбрасывать по всей форме, так как их функции взаимосвязаны. То же касается и сведений о клиенте – например, поля с именем и адресом обычно объединяют в одну группу. Элементы одной группы, как правило, ограничивают рамкой.

Часто пользователи переходят от одного элемента управления к другому с помощью клавиши *Tab*, поэтому в таких формах ввода, как диалоговые панели, важен осмысленный порядок обхода.

Согласованность элементов пользовательского интерфейса – важнейшее качество интерфейса, гарантирующее гармонию.

Для достижения согласованности следует на ранних фазах проектирования выработать стратегию и соглашения о стиле. Типы элементов управления, их размеры, способы группирования и шрифты надо определять заранее. Для решения этих вопросов можно использовать пробную версию системы или ее прототипы [35]. Итак, какие же вопросы следует согласовать?

1. Тип элемента управления – не стоит использовать все доступные элементы управления. Выбирайте только те, которые наилучшим образом подходят для приложения.

2. Свойства – важно согласовать свойства элементов управления. Например, если в одном месте у текстового поля белый фон, то в другом месте не стоит без веских причин использовать серый.

3. Тип формы – для удобства работы с приложением важно согласовать формы. Две формы – одна с серым фоном и с трехмерными эффектами, а другая с белым фоном и плоскими элементами – внесут непонятность и несогласованность.

Понятность элементов

Назначение интуитивно понятных элементов можно определить по их внешнему виду. Интуитивно понятные элементы стоит применять и в пользовательском интерфейсе.

Например, пользователи ожидают, что белое поле, окруженное рамкой, предназначено для редактирования текста. Однако можно отобразить его и без рамки, так что оно будет выглядеть как нередактируемая метка.

Использование разделителей

Разделители подчеркивают важность некоторых элементов, в результате чего работа с приложением становится более удобной. Обычно разделителями служит пустое пространство вокруг или между данными формы. Элементы управления и данные загромождают интерфейс, и становится трудно найти нужные поля и информацию. Разделители же позволяют подчеркнуть важность элементов формы. Заранее определенные расстояния между элементами и их выравнивание по вертикали и горизонтали делают интерфейс удобнее.

Простота дизайна

Вероятно, самое важное в дизайне интерфейса – его простота. Если интерфейс приложения сложен, то, скорее всего, и самим приложением трудно пользоваться. И с эстетической точки зрения ясный и простой дизайн всегда предпочтительнее.

Лучший тест на простоту интерфейса: если пользователь не может выполнить какую-либо задачу самостоятельно, вероятно, придется переделать интерфейс.

Цвет и изображения

Поскольку большинство современных мониторов способны отображать миллионы цветов, возникает соблазн использовать их как можно больше. Появляются проблемы, особенно если все вопросы, касающиеся цвета, не решены на начальных стадиях проектирования интерфейса.

Цветовые предпочтения у всех разные, вкусы пользователей и разработчиков далеко не всегда совпадают. Цвета могут вызывать сильные эмоции, а значения отдельных цветов часто несут конкретную смысловую нагрузку. Поэтому лучше остаться консерватором и применить приятные, нейтральные тона.

Выбор цвета зависит от предполагаемых пользователей приложения и от настроения, которое хотят передать дизайнеры. Ярко-красные, зеленые и желтые цвета годятся для детских программ, но банковские служащие вряд ли оценят их по достоинству.

Яркие цвета в небольших количествах позволяют привлечь внимание к важным элементам интерфейса. Однако их число следует ограничить, а цветовая схема должна соответствовать стилю приложения. По возможности применяйте 16-цветную палитру.

Еще одна проблема – цветовая слепота пользователей. Многие люди не различают комбинации некоторых основных цветов, например красного и зеленого. Поэтому они не увидят красный текст на зеленом фоне.

Удобство использования (usability)

Юзабилити – это степень эффективности, продуктивности и удовлетворенности, с которыми продукт может быть использован определенными пользователями в определенном контексте использования для достижения заданных целей [36].

Существует несколько общепризнанных стандартов по юзабилити и доступности, которые применяются в проектировании интерфейсов. Некоторые из них носят рекомендательный характер, в то время как другие являются обязательными. При этом характер требований может зависеть от страны, для которой создается продукт, или от типа системы.

Существует правило, по которому удобство использования важнее «стильности» и элегантности. Удобство приложения определяют пользователи.

Начиная проектировать пользовательский интерфейс, изучите другие приложения. Чтобы сделать их удобными, затрачено много ресурсов, проведено множество исследований. У интерфейсов таких приложений много общего: панели инструментов, всплывающие подсказки, строки состояния, контекстно-зависимые меню и диалоговые панели с вкладками. При проектировании стоит учесть и собственный опыт разработчиков как пользователей ПС. Однако все свои идеи надо проверять на прототипах.

Кроме того, большинство успешных приложений предоставляет пользователю возможность выбрать способ выполнения того или иного действия в зависимости от его предпочтений. Например, скопировать файл в программе Explorer можно средствами меню и клавиатурных команд или перетащив файл мышью. Разнообразие вариантов только добавляет привлекательности приложению, поэтому стоит предусмотреть возможность выполнения всех функций как с помощью клавиатуры, так и мыши.

Один из важнейших параметров, который проверяется при тестировании удобства использования программы, – доступность различных функций. Если пользователь не может понять, как применять функцию (или не может ее найти), эта функция бесполезна. Чтобы проверить доступность функции, попросите пользователей выполнить определенную задачу, не объясняя, как это сделать. Если им не удастся справиться с первой попытки, над доступностью этой функции придется еще поработать.

Все пользователи знают, как перемещаться по приложению средствами меню. Очень важно правильно использовать эту систему, как, собственно, и другие элементы интерфейса. Система навигации приложения должна быть простой и логичной.

Как бы ни был хорош интерфейс, возникают ситуации, когда пользователю не обойтись без помощи. Поэтому модель помощи пользователям должна включать в себя как встроенную справочную систему, так и печатную документацию. Кроме того, можно добавить всплывающие подсказки, статусные строки, подсказки и мастера. Модель помощи пользователям, как и другие части приложения, надо проектировать на ранних стадиях процесса разработки. Включенные в нее функции зависят от сложности приложения и от опыта пользователей.

Нормативные документы

Существует несколько стандартов, разработанных в ISO и относящихся к сфере юзабилити и человеко-компьютерного взаимодействия:

- ISO 9241 – содержит требования к эргономике визуальных дисплейных терминалов для офисной работы. Основной акцент ISO 9241 сделан на требования к офисному оборудованию, которые должны выполняться всеми производителями, например требования к дисплеям, клавиатурам, отражению, цвету, компоновке элементов на экране, диалогам и сообщениям об ошибках. Этот

стандарт не применим к проектированию интерфейсов для мобильных устройств. В нем дается определение юзабилити – «эффективность, продуктивность и удовлетворение пользователя», а также приводятся различные метрики, которые помогают проектировать удобные пользовательские интерфейсы;

- ISO 13407 – описан процесс проектирования интерактивных систем, ориентированных на пользователей. Этот стандарт содержит рекомендации по организации процесса проектирования интерфейсов и органичному встраиванию этого процесса в общий процесс производства ПО. В стандарте описаны методы юзабилити, необходимые для определения контекста использования продукта, выявления требований пользователей и заказчиков к системе, прототипирования и юзабилити-тестирования продукта;
- ISO 18529 – эргономика человеко-компьютерного взаимодействия – описание процесса проектирования интерефейсов, ориентированных на пользователей;

В стандарте детально рассмотрена модель зрелости организации с точки зрения уровня использования в ней UCD-процесса. Даются рекомендации по переходу на более высокие уровни зрелости;

- ISO 14915 – эргономика программного обеспечения мультимедийных пользовательских интерфейсов. В стандарте приведены рекомендации по созданию элементов управления для мультимедийных продуктов, таких, например, как обучающих систем, справочных киосков, электронных справок;
- ISO 16071– эргономика взаимодействия «человек – система». Руководящие указания по доступу к интерфейсам «человек – машина»;
- ISO 16982 – эргономика взаимодействия «человек – система». Методы, основанные на удобстве применения, для обеспечения проектирования, ориентированного на человека;
- ISO 20282 – юзабилити повседневных вещей. В первой части стандарта рассказывается о методе определения свойств контекста, в котором будет использоваться разрабатываемый продукт. Во второй части описывается методика измерения юзабилити для повседневных вещей.

Глава 11. ОФОРМЛЕНИЕ КУРСОВОГО ПРОЕКТА

Курсовой проект должен включать оттестированное программное средство и пояснительную записку.

11.1. Структура расчетно-пояснительной записки

Пояснительная записка курсового проекта состоит из следующих структурных элементов:

- титульный лист установленного образца (см. прил. 3);
- задание на курсовой проект (см. прил. 4);
- реферат;
- содержание;
- введение;
- главы основной части;
- заключение;
- список использованных источников;
- приложения.

Рассмотрим перечень подлежащих разработке вопросов.

Введение

Введение (0,5–1,5 страниц) должно содержать оценку современного состояния решаемой проблемы, ее актуальность и практическую значимость, описание основной цели работы и подчиненные ей более частные задачи.

Основная часть

Основную часть следует делить на разделы, подразделы, пункты и подпункты.

В структуре основной части должно быть выделено не менее трех разделов, а в их составе – не менее двух подразделов и т. д.

В основной части можно выделить проектный и программный разделы.

Проектный раздел должен включать:

1) обзор и анализ существующих аналогов и прототипов разработки и обоснование необходимости разработки, новизны решаемой задачи, ее связь с аналогичными разработками;

2) информационные инструменты и технологии, использованные при выполнении курсового проекта;

3) глоссарий терминов;

4) построение модели решаемой задачи (формирование структуры ПС и разработка алгоритмов, задаваемых спецификациями);

5) определение состава модулей с разделением их на иерархические уровни, фиксация межмодульных интерфейсов;

6) описание архитектуры ПС в виде диаграмм UML:

- ✔ диаграмма вариантов использования – функциональная структура ПС, показывающая функциональное назначение всего ПС и его отдельных частей;
- ✔ модульная (иерархическая) структура ПС, фиксирующая результаты проектирования ПС;
- ✔ диаграммы компонентов;
- ✔ диаграммы последовательностей и кооперации;
- ✔ диаграммы состояний;
- ✔ диаграммы наследования, зависимостей, классов и ПС, фиксирующие результаты объектно-ориентированного проектирования ПС;
- ✔ схемы алгоритмов, иллюстрирующих основные методы и алгоритмы, реализованные в ПС;

7) выбор структур информации в базе данных, проектирование базы данных;

8) описание методов по защите информации.

Программный раздел должен содержать:

- ✔ внешнее описание ПО;
- ✔ описание и содержание интерфейса программы (руководство пользователя);
- ✔ экспериментальный раздел;
- ✔ тестирование (проверка в нормальных и экстремальных условиях, например, граничные объемы исходных данных, реакция на нулевые примеры и исключительные ситуации).

Заключение

В заключении приводятся:

1) краткие выводы по результатам работы;

2) разработка рекомендаций и исходных данных по конкретному использованию;

3) перспективы дальнейшего развития.

Список использованных источников

Список использованных источников включает все источники, записанные в порядке появления ссылок на них в тексте пояснительной записки. Ссылки в тексте на литературные источники обязательны. Сведения об источниках, включенных в список, следует приводить в соответствии с требованиями ГОСТ 7.1.

Приложения

Приложения содержат:

- листинги программ;
- примеры работы;
- результаты тестирования;
- копии экрана;
- концептуальную модель, логическую модель и т. д.

11.2. Оформление расчетно-пояснительной записки

Примерный объем пояснительной записки должен составлять 25–40 страниц печатного текста, включая приложения [37].

К оформлению основного текста пояснительной записки предъявляются следующие требования.

Титульный лист

Титульный лист следует оформлять в соответствии с приложением 3. Титульный лист не нумеруется, но в общую нумерацию входит.

Содержание

Содержание размещается на новой странице и включает введение, наименования разделов, подразделов и пунктов, заключение, список использованных источников и приложения с указанием номеров страниц, на которых они расположены. Слово «Содержание» записывают в виде заголовка, размещенного по центру.

Заголовки элементов пояснительной записки в содержании соединяют отточием с номером страницы, на которой расположен заголовок. Номера страниц проставляют арабскими цифрами вплотную к правому полю без знаков препинания. Содержание должно быть сформировано автоматически.

Параметры страницы

Пояснительная записка оформляется на стандартных листах формата А4 (297×210 мм) на одной стороне листа. Поля страницы: правое – 10 мм; верхнее – 20 мм; левое – 23 мм; нижнее – 15 мм. Цвет шрифта – черный. Текст пояснительной записки следует печатать шрифтом Times New Roman, размером 14 пт, через одинарный междустрочный интервал. В случае вставки в строку формул допускается увеличение междустрочного интервала. Размер шрифта символов в формулах и уравнениях, заголовках разделов, заголовках и подрисуночных надписях иллюстраций, заголовках и тексте таблиц должен соответствовать размерам основного шрифта текста.

Допускается использование сокращений и аббревиатур.

Текст пояснительной записки набирается с абзацного отступа, равного 15 мм. Размеры полей и абзацных отступов должны быть одинаковыми на протяжении всего текста пояснительной записки.

Страницы пояснительной записки следует нумеровать арабскими цифрами, соблюдая сквозную нумерацию по всему тексту пояснительной записки. Номера ставятся в правом нижнем углу без точки.

В общую нумерацию страниц включают титульный лист, бланк задания, все листы работы, иллюстрации и таблицы, расположенные на отдельных листах, а также приложения. На титульном листе номер страницы не проставляют.

Оформление заголовков

Каждый раздел начинается с новой страницы. Заголовки элементов записки «Содержание», «Введение», «Заключение», «Список использованных источников», «Приложение» следует записывать в начале соответствующих страниц строчными буквами, кроме первой прописной, полужирным шрифтом с выравниванием по центру.

Заголовки структурных элементов и разделов основной части следует располагать с абзацного отступа, набирать полужирным шрифтом, строчными буквами, кроме первой прописной буквы.

В конце заголовков точку не ставят. Перенос слов в заголовках запрещен. Если заголовок занимает более одной строки, то последующие его строки должны быть записаны без абзацного отступа. Если заголовок состоит из двух предложений, то их разделяют точкой.

Нумерация заголовков выполняется арабскими цифрами. Подразделы должны иметь порядковую нумерацию внутри раздела, нумерация подраздела состоит из номеров раздела и подраздела,

разделенных точкой. В конце точка не ставится (например, 11.1). Номер пункта состоит из номеров раздела, подраздела и пункта, разделенных точками (например, 11.1.1).

Заголовки разделов должны быть отделены от текста интервалом 18 пт, заголовки подразделов и пунктов: сверху – интервалом 18 пт, снизу – интервалом 12 пт. Соседние, последовательно записанные заголовки раздела и подраздела следует отделять друг от друга интервалом 12 пт, а подраздела и пункта – интервалом 6 пт.

Запрещено переносить заголовки подразделов и пунктов с листа на лист, а также записывать их в конце текста, если после указанных заголовков на листе размещается меньше двух строк текста.

Изложение текста

В тексте не допускается употреблять обороты разговорной речи, применять для одного и того же понятия различные термины, сокращения слов, кроме установленных правилами орфографии. Перечень допускаемых сокращений русских слов оговорен в ГОСТ 2.316 и ГОСТ 7.12, белорусских – в СТБ 7.12.

Наименование команд, режимов следует выделять кавычками и курсивом, например «*Включить*».

В тексте перед обозначением параметра необходимо дать пояснение, например «…среднеквадратическое отклонение £…».

Перечисления

Пункты перечисления записывают после двоеточия с абзацного отступа, после каждого пункта ставится точка с запятой, после последнего – точка. Перед каждым пунктом перечисления следует ставить тире. Ниже приведен пример простого перечисления.

В цикле необходимо обрабатывать исключения нескольких типов:

– отсутствие прав доступа к файлу/каталогу;

– принудительное завершение потока извне;

– слишком длинный путь (более 255 символов).

В сложных перечислениях для детализации используют строчные буквы и арабские цифры.

Таблицы

Таблицу следует располагать в записке непосредственно после текста, в котором она упоминается. На все таблицы документа в тексте должны быть приведены ссылки, например «…в таблице 11.1».

Таблица при необходимости может иметь заголовок, который выполняется строчными буквами (кроме первой прописной) и помещается на одной строке через тире после слова «Таблица».

При переносе части таблицы на другую станицу слово «Таблица» и ее название помещают только над первой частью таблицы, над последующими частями таблицы пишут слева «Продолжение таблицы» с указанием ее номера, например «Продолжение таблицы 11.1».

Таблицы нумеруются арабскими цифрами сквозной нумерацией или в пределах раздела. Ниже приводится пример выполнения таблицы.

Таблица 11.1 – Соответствие классов-форм и реализованных модулей

Форма	Модуль	Назначение
Form_TMain	Unit1	Главный модуль
Form_LogOn	Unit2	Окно авторизации

В таблице допускается применять шрифт на 1–2 пт меньший, чем в тексте.

Формулы и уравнения

В формулах и уравнениях в качестве символов следует применять обозначения, установленные стандартами или принятые в данной отрасли. Пояснения символов и числовых коэффициентов, входящих в формулу, если они не пояснены ранее в тексте, должны быть приведены непосредственно под формулой. Первая строка пояснения должна начинаться со слова «где» без двоеточия после него.

Например. Математическое ожидание непрерывной случайной величины рассчитывается по следующей формуле:

$$M[X] = \int_{-\infty}^{+\infty} x f_X(x) dx, \qquad (11.1)$$

где X – непрерывная случайная величина;

$f_X(x)$ – плотность распределения величины X.

Формулы и уравнения выравниваются по центру.

Все формулы и уравнения нумеруются арабскими цифрами сквозной нумерацией или в пределах раздела. В случае нумерации в пределах раздела номер формулы состоит из номера раздела и порядкового номера формулы, разделенных точкой. Номер указывают в круглых скобках с правой стороны листа на уровне формулы. Ссылки в тексте на порядковые номера формул дают в скобках, например «…в формуле (11.1)…».

Иллюстрации

Иллюстрации (фотографии, рисунки, чертежи, схемы, диаграммы, графики, карты и др.) служат для наглядного представления характеристик объектов исследования, полученных теоретических и (или) экспериментальных данных и выявленных закономерностей. Не допускается одни и те же результаты представлять в виде иллюстрации и таблицы.

Иллюстрации следует располагать непосредственно после абзаца, содержащего ссылку на них. Если для размещения иллюстрации недостаточно места на соответствующей странице, необходимо разместить ее в начале следующей страницы. На все иллюстрации должны быть даны ссылки в тексте.

Иллюстрации следует нумеровать сквозной нумерацией или в пределах каждого раздела арабскими цифрами. При нумерации в пределах раздела номер рисунка включает в свой состав номер раздела и порядковый номер рисунка по разделу, разделенных точкой, например «Рисунок 11.1». Иллюстрации отделяют от текста интервалом 14 пт. Ниже дается пример оформления рисунка.

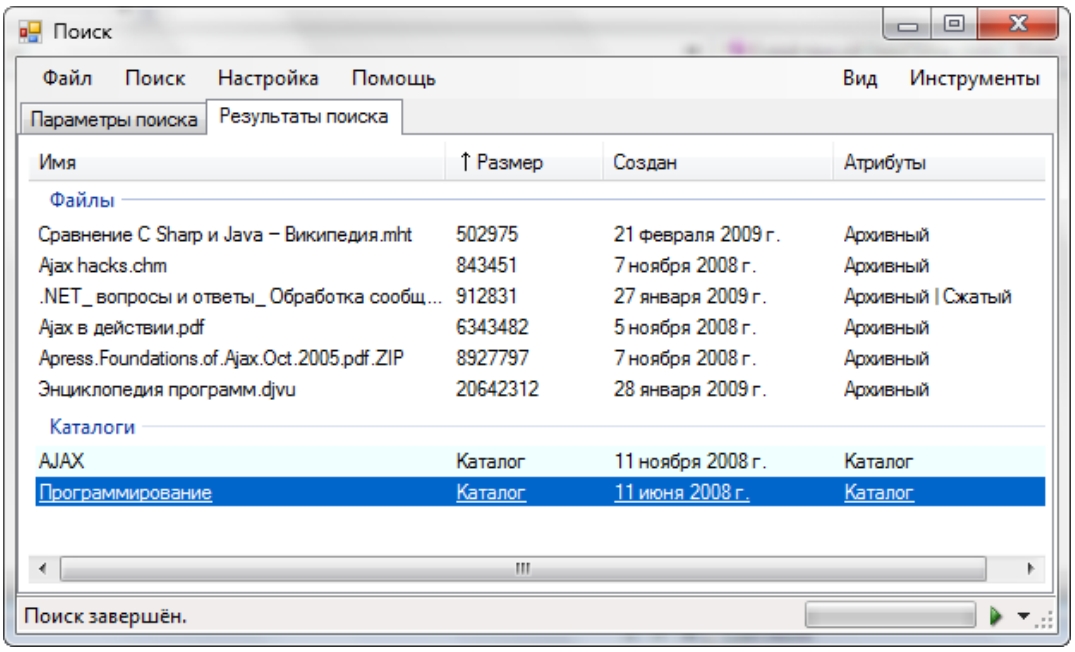


Рисунок 11.1 – Схема базы данных (модуль тестирования)

Ссылки

Ссылки на источники в тексте осуществляются путем приведения порядкового номера в соответствии с библиографическим

списком. Номер источника по списку заключается в квадратные скобки, например [4].

Приложения

Приложения должны иметь общую с остальной частью записки сквозную нумерацию страниц. В тексте документа на все приложения должны быть ссылки. Приложения располагают в порядке ссылок на них. Все приложения должны быть перечислены в содержании документа с указанием их номера и заголовка.

Каждое приложение следует начинать с нового листа с указанием в правом верхнем углу слова «ПРИЛОЖЕНИЕ», напечатанного прописными буквами. Приложение должно иметь содержательный заголовок, который размещается с новой строки по центру листа с прописной буквы.

Приложения обозначают прописными буквами русского алфавита, начиная с А (за исключением Ё, З, Й, О, Ч, Ь, Ы, Ъ); например ПРИЛОЖЕНИЕ А, ПРИЛОЖЕНИЕ Б. Допускается обозначать приложения буквами латинского алфавита, за исключением I и O.

Пример оформления списка использованных источников

1. Брауде, Э. Дж. Технология разработки программного обеспечения / Э. Дж. Брауде. – СПб.: Питер, 2004. – 656 с.

2. Сергиевский, Г. М. Функциональное и логическое программирование / Г. М. Сергиевский, Н. Г. Волченков. – М.: Академия, 2010. – 320 с.

3. Компьютерная геометрия / А. О. Иванов [и др.]. – М.: Интернет-университет информационных технологий, 2010. – 392 с.

4. Способ устройства кодирования: пат. 432848 Респ. Беларусь, МПК7 C08/H 8/90 / Л. М. Урбанович, Н. Г. Волочкова; заявитель Полоц. гос. ун-т. – № a9897644; заявл. 09.07.2006; опубл. 23.09.2010 // Афіцыйны бюл. / Нац. цэнтр інтэлектуал. уласнасці. – 2010. – № 3. – С. 156.

5. Библиотека электронных ресурсов компании DATA+ [Электронный ресурс] / Электронная газета ArcReview 2007. – № 1. – Минск, 2010. – Режим доступа: http://www.dataplus.ru/Arcrev/Number_40/7_GISbb.html. – Дата доступа: 25.01.2010.

6. Proceeding of minisymposium on biological nomenclature in the 21st centry [Electronic resource] / ed. J. L. Reveal. – College Park M.D., 1996. – Mode of access: http://www.inform.ind.edu/PBIO/brum.html. – Date of access: 14.09.2005.

ПРИЛОЖЕНИЕ 1

АРХИТЕКТУРА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Архитектура ПО – это его строение, как оно видно (или должно быть видно) извне (некоторая совокупность взаимодействующих подсистем) [5].

Архитектурный стиль, иногда называемый архитектурным шаблоном, – это набор принципов (схема), обеспечивающий абстрактную инфраструктуру для семейства систем. Архитектурный стиль улучшает секционирование и способствует повторному использованию дизайна благодаря обеспечению решений часто встречающихся проблем [25, 30].

Обзор основных архитектурных стилей

В таблице приводится список типовых архитектурных стилей, которые определяются в главе 5, и дается краткое описание каждого из них [25, 30, 38–40].

Архитектурные стили

Архитектурный стиль/парадигма	Описание
Клиент/сервер	Система разделяется на два приложения, где клиент выполняет запросы к серверу. Во многих случаях в роли сервера выступает база данных, а логика приложения представлена процедурами хранения
Компонентная архитектура	Дизайн приложения разлагается на функциональные или логические компоненты с возможностью повторного использования, предоставляющие тщательно проработанные интерфейсы связи
Дизайн на основе предметной области	Объектно-ориентированный архитектурный стиль, ориентированный на моделирование сферы деловой активности и определяющий бизнес-объекты на основании сущностей этой сферы
Многослойная архитектура	Функциональные области приложения разделяются на многослойные группы (уровни)

Окончание таблицы

Архитектурный стиль/парадигма	Описание
Шина сообщений	Архитектурный стиль, предписывающий использование программной системы, которая может принимать и отправлять сообщения по одному или более каналам связи, так что приложения получают возможность взаимодействовать, не располагая сведениями друг о друге
N-уровневая/ 3-уровневая архитектура	Функциональность выделяется в отдельные сегменты, во многом аналогично многослойному стилю, но в данном случае сегменты физически располагаются на разных компьютерах
Объектно-ориентированная архитектура	Парадигма проектирования, основанная на распределении ответственности приложения или системы между отдельными многократно используемыми и самостоятельными объектами, содержащими данные и поведение
Сервисно-ориентированная архитектура (SOA)	Описывает приложения, предоставляющие и потребляющие функциональность в виде сервисов с помощью контрактов и сообщений

Архитектура программной системы практически никогда не ограничена лишь одним архитектурным стилем, часто она является сочетанием архитектурных стилей, образующих полную систему. Например, может существовать SOA-дизайн, состоящий из сервисов, при разработке которых использовалась многослойная архитектура и объектно-ориентированный архитектурный стиль.

Создавая приложение курсового проекта, можно реализовать клиента, который будет отправлять запросы к программе на сервере. В этом случае развертывание клиента и сервера будет выполнено с помощью архитектурного стиля клиент/сервер и использовать компонентную архитектуру для дальнейшего разложения дизайна на независимые компоненты, предоставляющие соответствующие интерфейсы. Применение объектно-ориентированного подхода к этим компонентам повысит возможности повторного использования, тестирование и гибкость.

С точки зрения количества пользователей, работающих с одной копией ПО, различают однопользовательскую и многопользовательскую (сетевую) архитектуру.

Кроме того, в рамках однопользовательской архитектуры выделяют:

- программы;
- пакеты программ (несколько отдельных программ, решающих задачи определенной прикладной области);
- программные комплексы (совокупность программ, совместно обеспечивающих решение небольшого класса сложных задач одной прикладной области);
- программные системы (организованная совокупность программ (подсистем), позволяющая решать широкий класс задач из некоторой прикладной области).

Методика построения архитектуры

Рассмотрим итеративную технику, которая может использоваться при продумывании и создании прототипа будущей архитектуры.

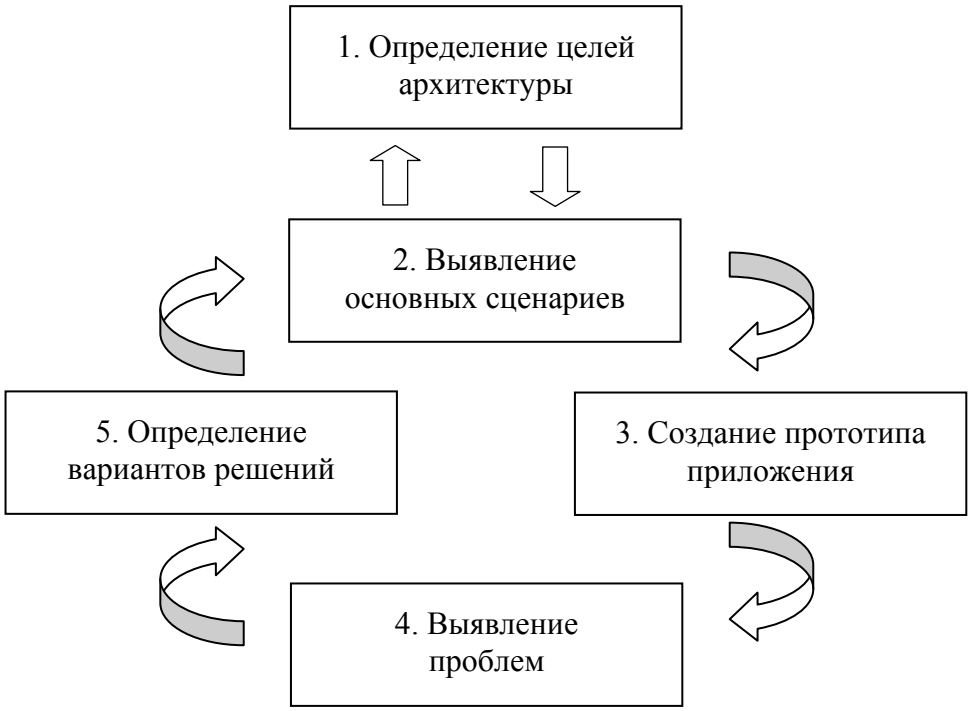
Исходные данные, выходные данные и этапы проектирования

Обычно исходными данными являются варианты использования и сценарии поведения пользователя, функциональные и нефункциональные требования (включая параметры качества, такие как производительность, безопасность, надежность и др.).

В ходе процесса разработки создается список значимых с точки зрения архитектуры вариантов использования, аспектов архитектуры, требующих специального внимания, и возможных архитектурных решений, которые удовлетворяют требованиям и ограничениям, выявленным в процессе проектирования. Общей техникой постепенной доработки дизайна до тех пор, пока он не будет удовлетворять всем требованиям и ограничениям, является итеративная методика, включающая пять основных этапов (рисунок).

Такой процесс создания архитектуры предполагает итеративный и инкрементный подход. Сначала создается возможный вариант архитектуры – обобщенный дизайн, который может тестироваться по основным сценариям, требованиям, известным ограничениям и параметрам качества. В ходе доработки варианта архитектуры выявляются дополнительные детали и сведения о дизайне, результатом чего становится расширение основных сценариев, корректировка общего представления приложения и подхода к решению проблем.

На основании общих целей архитектуры можно планировать время, затрачиваемое на каждую из работ по проектированию. Например, на разработку прототипа может уйти лишь несколько дней, тогда как для создания полной и детально проработанной архитектуры сложного приложения потребуется месяц и множество итераций. Исходя из своего понимания целей, оценивайте необходимое количество времени и сил для каждого этапа, это поможет прийти к видению результата и четкому определению целей и приоритетов архитектуры.



Основные этапы итеративного процесса проектирования архитектуры

Ключевые сценарии

В контексте архитектуры и дизайна вариант использования (use case) – это описание ряда взаимодействий между системой и одним или более действующими лицами (либо пользователем, либо другой системой). Сценарий – это более широкое и всеобъемлющее описание взаимодействия пользователя с системой, чем ветвь варианта использования. Основной целью при продумывании архитектуры системы должно быть выявление нескольких ключевых сценариев, что поможет при принятии решения об архитектуре.

Применяйте сценарии и варианты использования для тестирования своего дизайна и выявления возможных проблем.

Общее представление приложения

Создание общего представления приложения включает следующие действия:

1) выбор типа приложения (определите, приложение какого типа создается);
2) задание ограничений развертывания;
3) определение значащих архитектурных стилей проектирования;
4) выбор подходящих технологий.

Выбор технологии представления

Рассмотрим рекомендации, которые помогут выбрать технологии представления, реализации и связи на платформе Microsoft [41].

Для разработки приложения для мобильных устройств могут использоваться технологии слоя представления, такие как .NET Compact Framework, ASP.NET и Silverlight.

Для разработки приложений с насыщенными интерфейсами пользователя (UI – User Interface), развертываемыми и выполняемыми на клиенте, могут применяться сочетания технологий слоя представления Windows Presentation Foundation (WPF), Windows Forms и XAML Browser Application (XBAP).

Для развертывания насыщенных клиентских Интернет-приложений (RIA) может использоваться подключаемый модуль Silverlight™ или Silverlight в сочетании с AJAX.

Для создания Web-приложений могут применяться ASP.NET WebForms, AJAX, Silverlight, ASP.NET MVC и ASP.NET Dynamic Data.

Для создания сервисов, предоставляющих функциональность внешним потребителям систем и сервисов, могут использоваться Windows Communication Foundation (WCF) и ASP.NET Web services (ASMX).

Анализ архитектуры приложения

Это критически важная задача, поскольку позволяет сократить затраты на исправление ошибок, как можно раньше выявить и исправить возможные проблемы. Анализ архитектуры следует выполнять часто: по завершении основных этапов проекта и в ответ на существенные изменения в архитектуре.

Основная цель анализа архитектуры – подтверждение применимости базовой архитектуры и ее возможных вариантов, а также проверка соответствия предлагаемых технических решений функциональным требованиям и параметрам качества [20, 41].

ПРИЛОЖЕНИЕ 2

ТЕХНОЛОГИИ УРОВНЯ ДАННЫХ UDA

Разработанная Microsoft универсальная архитектура данных (UDA – Universal Data Architecture) предназначена для организации высокопроизводительного доступа к хранящимся на предприятии данным любого типа – структурированным и неструктурированным, связанным и несвязанным [30]. UDA – набор COM-интерфейсов, реализующих концепцию доступа к данным. Она основана на интерфейсах OLE DB для создания компонентов, работающих с базами данных. OLE DB позволяет хранилищам информации открывать свои функции без необходимости имитировать реляционный источник данных. Кроме того, в рамках этой технологии универсальным сервисным компонентам (например, специализированным обработчикам запросов) разрешено расширять функции более простых поставщиков данных. Поскольку основная цель OLE DB – эффективный доступ к информации, а не простота использования, в UDA добавлен интерфейс прикладного уровня Microsoft ActiveX Data Objects (ADO).

Microsoft Data Access Components (MDAC) – это реализация UDA, включающая как ADO, так и компонент доступа OLE DB для ODBC. Это позволяет ADO обращаться к любой базе данных, снабженной драйвером ODBC (фактически ко всем основным базам данных). ADO поддерживает двойные интерфейсы, которые можно применять в языках сценариев, в C++, в Microsoft Visual Basic и других средствах разработки. Как показано на рисунке, ADO как механизм доступа к информации дает возможность написать приложения для уже существующих и новых структурированных и неструктурированных данных независимо от их местонахождения.

UDA создана на основе ODBC, Remote Data Objects (RDO) и Data Access Objects (DAO), но по сравнению с ними ее возможности значительно расширены.

Основные этапы моделирования данных таковы:

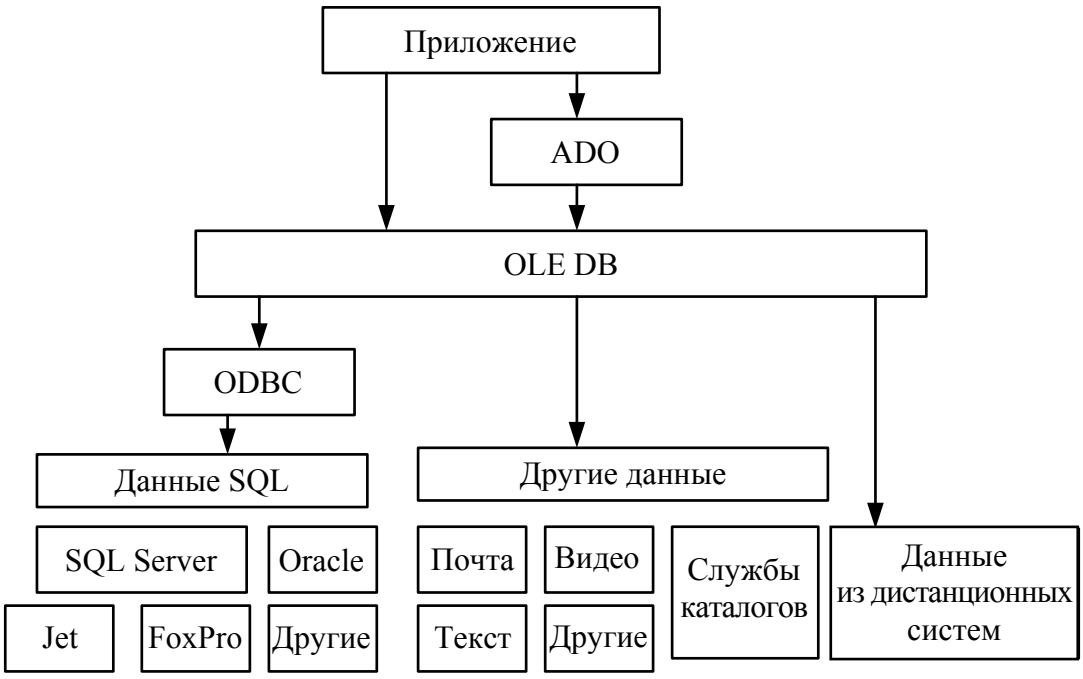
- ✔ определение структуры данных и связанных с ними процессов (например, логическая организация данных);

✔ определение типов данных, их размеров и значений по умолчанию;

✔ обеспечение целостности данных с помощью бизнес-правил и ограничений;

✔ подготовка операционных процессов, например проверок защиты и резервного копирования;

✔ выбор технологии хранения данных (реляционной, иерархической, индексируемой и т. д.).



Структура UDA

Определение характеристик данных

По мере исследования структур данных можно группировать некоторые их элементы, выявляя характеристики и связи:

1) определять таблицы, строки и столбцы;
2) выбирать ключевые поля;
3) создавать связи между таблицами;
4) определять типы данных.

Обеспечение целостности данных

Для обеспечения целостности данных, хранимых и используемых в структурах приложения, следует контролировать все процессы, обращающиеся к информации. Реализовать это позволяют следующие основные концепции:

- нормализация данных;
- определение бизнес-правил доступа к данным;
- обеспечение целостности ссылок;
- проверка данных.

Выбор технологии доступа к данным

При выборе технологии доступа к данным необходимо учесть тип данных, с которыми предполагается работать, и то, как эти данные будут обрабатываться в приложении.

Используйте ADO.NET Entity Framework (EF), если хотите создать модель данных и соотнести ее с реляционной базой данных; соотнести один класс с множеством таблиц, используя наследование, или выполнять запросы к реляционным хранилищам, не входящим в семейство продуктов Microsoft SQL Server.

Используйте LINQ to XML, если приложение работает с XML-данными, запросы к которым необходимо выполнять, применяя синтаксис LINQ.

Выбор способа подключения к источнику данных

Подключения к источникам данных – это фундаментальная часть слоя доступа к данным. Слой доступа к данным должен координировать все подключения к источнику данных, используя для этого инфраструктуру доступа к данным.

Открывайте подключения к источнику данных как можно позже и закрывайте их как можно раньше. Это обеспечит блокировку ресурсов лишь на короткие промежутки времени и сделает их более доступными для других процессов.

Там, где это возможно, осуществляйте транзакции через одно подключение.

По соображениям безопасности избегайте использования системных или пользовательских DSN для хранения данных подключения.

Предусмотрите логику повторного подключения для случаев разрыва соединения с источником данных или его закрытия по истечении времени ожидания.

По возможности используйте пакетные команды, что позволит сократить количество обращений к серверу базы данных.

Другой важный аспект, который необходимо учесть, – требования безопасности в связи с доступом к источнику данных.

Предпочтительнее использовать аутентификацию Windows, а не аутентификацию SQL Server.

При использовании аутентификации SQL применяйте учетные записи с надежными паролями; с помощью ролей базы данных ограничьте права доступа каждой учетной записи в рамках SQL Server; шифруйте строки подключения в конфигурационных файлах.

Используйте учетные записи с наименьшими правами доступа к базе данных и требуйте от вызывающей стороны предоставлять слою данных идентификационные данные для целей аудита.

Не храните пароли для проверки пользователей в базе данных, ни в виде открытого текста, ни в зашифрованном виде. Храните хеши паролей с шумом (случайные разряды, используемые как один из параметров в функции хеширования).

При использовании SQL-выражений для доступа к источнику данных четко обозначьте границы доверия и применяйте параметризованные запросы, а не конкатенацию строк. Это обеспечит защиту от атак через внедрение SQL-кода.

Выработка стратегий обработки ошибок источника данных

На данном этапе должна быть выработана общая стратегия обработки ошибок источников данных. Все исключения, связанные с источниками данных, должны перехватываться слоем доступа к данным.

Стратегия централизованного управления исключениями обеспечит единообразие при обработке исключений.

Предусмотрите логику повтора попыток для обработки ошибок, возникающих при переходе на другой ресурс в случае сбоя сервера или базы данных.

Форматы данных

Для передачи данных между уровнями чаще всего используются скалярные значения, XML, объекты DataSets и собственные объекты. В таблице представлены ключевые факторы, оказывающие влияние на выбор типа данных.

Если клиентское приложение не может обрабатывать данные в предоставляемом формате, должен быть реализован механизм трансляции, обеспечивающий их преобразование. Однако это может негативно сказаться на производительности. При проектировании стратегии доступа к данным руководствуйтесь следующими рекомендациями:

1) по возможности загружайте данные асинхронно, так чтобы UI не блокировался при их загрузке. Однако необходимо также учесть конфликты, которые могут возникать при попытках пользователя взаимодействовать с данными до того, как они будут полностью загружены, и при проектировании интерфейса предусмотреть механизмы защиты от возникающих в результате этого ошибок;

2) если клиент предполагает потреблять большие объемы данных, разделение данных на фрагменты и их асинхронная загрузка в локальный кэш позволят улучшить производительность. Необходимо спланировать, как будет обрабатываться несогласованность локальных копий и исходных данных. Для этого могут использоваться временные метки или события;

Форматы данных

Тип	Факторы
Скалярные значения	Требуется обеспечить встроенную поддержку сериализации. Позволяют обрабатывать возможные изменения схемы. Скалярные значения обусловливают тесное связывание, которое потребует изменения сигнатур методов, таким образом, оказывая влияние на вызывающий код
XML	Необходимо слабое связывание, при котором вызывающая сторона должна знать только о данных, определяющих бизнес-сущность, и схеме, обеспечивающей метаданные для бизнес-сущности. Возможно поддерживать разные типы вызывающих сторон, включая клиентов сторонних производителей. Требуется обеспечить встроенную поддержку сериализации
DataSet	Необходимо обеспечить поддержку сложных структур данных. Следует обрабатывать наборы и сложные отношения. Важно отслеживать изменения данных DataSet. Требуется обеспечить встроенную поддержку сериализации
Собственные объекты	Необходимо обеспечить поддержку сложных структур данных. Взаимодействие осуществляется с компонентами, которым известен тип объекта. Желательна поддержка бинарной сериализации для обеспечения лучшей производительности

3) в сценариях без постоянного подключения отслеживайте подключение и реализуйте механизм диспетчеризации сервисов для поддержки пакетной обработки, чтобы обеспечить пользователям возможность выполнять множество обновлений данных;

4) выработайте стратегию выявления и управления конфликтами параллельной обработки, которые возникают при попытке обновления централизованно хранящихся данных многими пользователями.

ПРИЛОЖЕНИЕ З

ПРИМЕР ОФОРМЛЕНИЯ ТИТУЛЬНОГО ЛИСТА ПОЯСНИТЕЛЬНОЙ ЗАПИСКИ КУРСОВОГО ПРОЕКТА

Учреждение образования
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет издательского дела и полиграфии
Кафедра информационных систем и технологий
Специальность 1-40-01 02 «Информационные системы и технологии»
Специализация «Издательско-полиграфический комплекс»

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
КУРСОВОГО ПРОЕКТА
по дисциплине «Объектно-ориентированное программирование»
Тема _____
_____

Исполнитель
студент(ка) 2 курса группы 8 _____ И. А. Иванов
подпись, дата

Руководитель
доцент, канд. техн. наук _____ Н. В. Пацей
подпись, дата

Курсовой проект защищен с оценкой _____
Руководитель _____ Н. В. Пацей
подпись

Минск 2012

ПРИЛОЖЕНИЕ 4

ПРИМЕР ЗАДАНИЯ НА КУРСОВОЙ ПРОЕКТ

Учреждение образования
«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет издательского дела и полиграфии
Кафедра информационных систем и технологий
Специальность 1-40-01 02 «Информационные системы и технологии»
Специализация «Издательско-полиграфический комплекс»

«УТВЕРЖДАЮ»
Заведующий кафедрой ИСиТ
_____ П. П. Урбанович
подпись

«__»_____201__ г.

ЗАДАНИЕ
на курсовой проект

студенту(ке) _____
1. Тема
_____
_____
2. Срок защиты _____
3. Исходные данные
_____
_____
_____
_____
_____
_____
_____

4. Содержание расчетно-пояснительной записки курсового проекта (перечень вопросов, подлежащих разработке)

_____

_____

_____

_____

_____

_____

_____

5. Перечень графического, иллюстрационного материала

_____

_____

_____

_____

6. Консультанты (с указанием разделов)

_____

_____

_____

_____

7. Календарный график работы

_____

_____

_____

8. Дата выдачи задания _____

Руководитель_____ Н. В. Пацей
подпись

Задание принял(а) к исполнению _____
дата и подпись студента(ки)

ПРИЛОЖЕНИЕ 5

СПИСОК АББРЕВИАТУР И ТЕРМИНОВ

ActiveX	ActiveX Data Objects Технология, построенная на базе OLE-automation, предназначена для создания программного обеспечения, предполагает использование визуального программирования для создания компонентов.
ADO	ActiveX Data Objects Интерфейс программирования приложений для доступа к данным.
AJAX	Asynchronous Javascript and XML Подход к построению интерактивных пользовательских интерфейсов Web-приложений.
API	Application Program Interface Интерфейс программирования приложений.
CASE-технология	Computer-Aided Software/System Engineering Автоматизированная технология разработки и сопровождения программного обеспечения.
CLR	Common Language Runtime Общая среда выполнения.
COM	Component Object Model Компонентная модель объектов.
COP	Component-Oriented Programming Компонент-ориентированное программирование.
CORBA	Common Object Request Broker Architecture Общая архитектура с посредником обработки запросов объектов.

CRC-карточки, CRC-cards	Class/Responsibilities/Collaborators Класс/Ответственности/Сотрудники Простое, но достаточно эффективное средство мозгового штурма при выявлении ключевых абстракций и механизмов.
DCOM	Distributed COM Распределенная COM.
DFD	Data Flow Diagrams Диаграммы потоков данных.
DLL	Dynamic-link Library Динамически подключаемая библиотека.
ER	Entity – Relationship Сущность – связь.
ERD	Entity-Relationship Diagrams Диаграммы «сущность – связь».
freeware	Бесплатные программы, свободно распространяемые, поддерживаются самим пользователем, который правомочен вносить в них необходимые изменения.
GUI	Grafic User Interface Графический пользовательский интерфейс, который обеспечивает отображение данных с помощью набора элементов управления.
HTML	HyperText Markup Language Язык разметки гипертекста.
IDEF0	Icam DEFinition Методология функционального моделирования и графическая нотация, предназначенная для формализации и описания бизнес-процессов.

IDL	Interface Definition Language Язык определения интерфейсов.
IEC	International Electrotechnical Commission Международная комиссия по электротехнике.
IL	Intermediate Language Низкоуровневый код, в который преобразуются все .Net языки.
IMDB	In Memory Database База данных в оперативной памяти.
ISO	International Standards Organization Международная организация по стандартизации.
MDAC	Microsoft Data Access Components Совокупность технологий компании Microsoft для унифицированного доступа к любым видам данных.
MDI	Multiple-Document Interface Многодокументный интерфейс.
MOF	Microsoft Operations Framework Набор операций от Microsoft.
MSF	Microsoft Solutions Framework Набор решений от Microsoft.
MVC	Model – view – controller Модель – представление – поведение (контроллер) Архитектура программного обеспечения, в которой модель данных приложения, пользовательский интерфейс и управляющая логика разделены на три отдельных компонента.

OBA	Microsoft Office Business Application Бизнес-приложения Microsoft Office.
ODBC	Open Database Connectivity Программный интерфейс (API) доступа к базам данных.
OLE DB	Object Linking and Embedding Связывание и внедрение объектов.
OLE-automation, или просто Automation	Технология создания программируемых приложений, обеспечивающая программируемый доступ к внутренним службам этих приложений.
PSP	Personal Software Process Персональный процесс разработки ПО.
RAD	Rapid Application Development Быстрая разработка приложений.
RIA	Rich Internet Application Насыщенные клиентские Интернет-приложения.
ROP	Rational Objectory Process Рациональный объектный процесс.
RUP	Rational Unified Process Рациональный унифицированный процесс.
SDI	Single-Document Interface Однодокументный интерфейс.
shareware	Некоммерческие (условно-бесплатные) программы, которые могут использоваться, как правило, бесплатно.
SOA	Service Oriented Architecture Сервисно-ориентированная архитектура.

STD	State Transition Diagrams Диаграммы переходов состояний.
UDA	Universal Data Architecture Универсальная архитектура данных.
UML	Unified Modeling Language Диаграммный способ (язык) для спецификации, визуализации, конструирования и документирования продуктов на процессах ЖЦ.
VSS	Visual SourceSafe Программный продукт компании Microsoft, файл-серверная система управления версиями.
WPF	Windows Presentation Foundation (кодовое название – Avalon) Графическая (презентационная) подсистема в составе .NET Framework.
XAML	Extensible Application Markup Language Расширяемый язык разметки приложений.
XML	Extensible Markup Language Расширяемый язык разметки.
XP	Extreme Programming Экстремальное программирование.
Абстрактный класс, abstract class	Класс, который не может иметь экземпляров. Абстрактный класс пишется в предположении, что его конкретные подклассы дополнят его структуру и поведение, скорее всего, реализовав абстрактные операции.
Актер, actor	Объект, воздействующий на другие объекты, но сам не подвергающийся воздействию с их стороны. В некоторых контекстах то же самое, что активный объект – действующие лица, для которых создается система.

Алгоритм	Система точно сформулированных правил, определяющая процесс преобразования допустимых исходных данных (выходной информации) в желаемый результат (выходную информацию) за конечное число шагов.
Архитектура, architecture	Логическая и физическая структура системы, сформированная всеми стратегическими и тактическими проектными решениями.
Ассоциация, association	Отношение, означающее некоторую смысловую связь между классами.
Делегирование, delegation	При делегировании один объект, ответственный за операцию, передает выполнение этой операции другому объекту.
Диспинтерфейс, dispinterface	Специальный интерфейс, облегчающий вызов функций объекта.
ЖЦ, жизненный цикл системы	Непрерывный процесс, который начинается с момента принятия решения о необходимости ее создания и заканчивается в момент ее полного изъятия из эксплуатации.
Задача, problem, task	Проблема, подлежащая решению.
Интерфейс, interface	Именованное множество операций, характеризующих поведение отдельного элемента модели извне без указания их внутренней структуры.
Метакласс, metaclass	Класс класса; класс, экземпляры которого сами являются классами.
Модуль, module	Единица кода, служащая строительным блоком физической структуры системы; программный блок, который содержит объявления, выраженные в соответствии с требованиями языка

	и образующие физическую реализацию части или всех классов и объектов логического проекта системы. Как правило, модуль состоит из интерфейсной части и реализации.
Надежность программной системы	Способность системы сохранять свои свойства (безотказность, устойчивость и др.) в процессе преобразования исходных данных в результаты в течение определенного промежутка времени при определенных условиях эксплуатации.
Объект, object	Отдельный экземпляр класса, который создается на этапе выполнения программы.
Объектно-ориентированная декомпозиция, object-oriented decomposition	Процесс разбиения системы на части, соответствующие классам и объектам предметной области. Практическое применение методов объектно-ориентированного проектирования приводит к объектно-ориентированной декомпозиции, при которой мы рассматриваем мир как совокупность объектов, согласованно действующих для обеспечения требуемого поведения.
Объектно-ориентированное программирование, object-oriented programming (OOP)	Методология реализации, при которой программа организуется как совокупность сотрудничающих объектов, каждый из которых является экземпляром какого-либо класса, а классы образуют иерархию наследования. При этом классы обычно статичны, а объекты очень динамичны, что поощряется динамическим связыванием и полиморфизмом.
Объектно-ориентированное проектирование, object-oriented design (OOD)	Методология проектирования, соединяющая процесс объектно-ориентированной декомпозиции и систему обозначений для представления логической и физической, статической и динамической моделей проектируемой системы. Система обозначений состоит из диаграмм классов, объектов, модулей и процессов.

Объектно-ориентированный анализ, object-oriented analysis	Метод анализа, согласно которому требования рассматриваются с точки зрения классов и объектов, составляющих словарь предметной области.
Отладка ПО	Деятельность, направленная на обнаружение и исправление ошибок в ПО с использованием процессов выполнения его программ. Отладка = Тестирование + Поиск ошибок + + Редактирование.
ПИ	Пользовательский интерфейс Система правил и средств, регламентирующая и обеспечивающая взаимодействие программы с пользователем.
ПО	Программное обеспечение Совокупность программ системы обработки информации и программных документов, необходимых для эксплуатации этих программ.
ПП	Программный продукт Это программное обеспечение, предназначенное для неопределенного круга покупателей.
Прецедент	Последовательность действий, выполняемых системой, которая выдает результат, ценный для конкретного субъекта.
Приложение, application	Программная реализация на компьютере решения задачи.
Программирование, programming	Теоретическая и практическая деятельность, связанная с созданием программ.
Рефакторинг, refactoring	Постоянное обновление, удаление лишних частей, ненужной функциональности программного кода.
СУБД	Системы управления базами данных.

Тестирование ПС	Процесс выполнения его программ на некотором наборе данных, для которого заранее известен результат применения или известны правила поведения этих программ.
Технология (от греч. techne)	Искусство, мастерство, умение.
Технология программирования	Совокупность методов и средств, применяемых в процессе разработки программного обеспечения.
Юзабилити, usability	Степень, в которой продукт может быть использован определенными пользователями для достижения поставленных целей эффективно, экономично и с удовольствием.

ЛИТЕРАТУРА

1. Rational Rose, методика RUP и язык UML [Электронный ресурс]. – Режим доступа: www.rational.com. – Дата доступа: 14.06.2011.

2. Кобайло, А. С. Моделирование информационных систем на базе Rational Rose: учеб.-метод. пособие для вузов / А. С. Кобайло, Н. А. Жиляк. – Минск: БГТУ, 2008. – 265 с.

3. Рамбо, Д. UML 2.0. Объектно-ориентированное моделирование и разработка / Д. Рамбо. – М.: БИНОМ, 2007. – 544 с.

4. Боггс, У. UML и Rational Rose / У. Боггс, М. Боггс. – М.: Лори, 2008. – 600 с.

5. Жоголев, Е. А. Технология программирования / Е. А. Жоголев. – М.: Научный мир, 2004. – 216 с.

6. Standard for Information Technoiogy – Software Life Cycle Processes: ISO/IEC 12207 [Electronic resource]. – Mode of access: www.iso.org/iso/. – Date of access: 14.06.2011.

7. Данные жизненного цикла промышленного объекта: ИСО 15926 [Электронный ресурс]. – Режим доступа: www.belgiss.org.by. – Дата доступа: 13.02.2011.

8. Крачтен, Ф. Введение в Rational Unified Process / Ф. Крачтен. – М.: Вильямс, 2002. – 240 с.

9. Поллис, Г. Разработка программных проектов на основе Rational Unified Process (RUP) / Г. Поллис, Л. Огастин, К. Лоу. – М.: Бином-Пресс, 2009. – 346 с.

10. Методика экстремального программирования XP [Электронный ресурс]. – Режим доступа: www.xprogramming.ru. – Дата доступа: 13.02.2011.

11. Коберн, А. Быстрая разработка программного обеспечения / А. Коберн. – М.: Лори, 2003. – 324 с.

12. Буч, Г. UML руководство пользователя / Г. Буч, Д. Рамбо, А. Джекобсон. – М.: ДМК, 2000. – 608 с.

13.Марка, Д. А. Методология структурного анализа и проектирования SADT / Д. А. Марка, К. Макгоуэн. – М.: Лори, 1993. – 239 с.

14. Иванова, Г. С. Технология программирования / Г. С. Иванова. – М.: Изд-во МГТУ им. Баумана, 2002. – 203 с.

15. Вендров, А. М. CASE-технологии. Современные методы и средства проектирования информационных систем / А. М. Вендров. – М.: Финансы и статистика, 1998. – 237 с.

16. Брауде, Э. Д. Технология разработки программного обеспечения / Э. Д. Брауде. – М.: Питер, 2004. – 655 с.

17. Все о CASE [Электронный ресурс]. – Режим доступа: www.caseclub.ru. – Дата доступа: 16.03.2011.

18. Информационные технологии. Процессы жизненного цикла программных средств: СТБ ИСО/МЭК 12207-2003 [Электронный ресурс]. – Режим доступа: www.nbrb.by/payment/TechCodePract/pdf/tcp_135_2008.pdf. – Дата доступа: 26.06.2011.

19. International standard ISO/IEC 12207 software life cycle processes [Electronic resource]. – Mode of access: www.abelia.com/docs/12207cpt.pdf. – Date of access: 26.06.2011.

20. Crnkovik, I. Component-Based Software Engineering: building systems from Components at 9th Conference and Workshops on Engineering of Computer-Based Systems / I. Crnkovik, S. Larsson, J. Stafford // Software Engineering Notes. – 2002. – Vol. 27, № 3. – P. 47–50.

21. Design Patterns, Elements of Reusable Object-oriented Software / E. Gamma [et al.]. – New-York: Addison-Wesley, 1995. – 345 p.

22. Component Object Model [Electronic resource]. – Mode of access: www.microsoft.com/tech/COM. asp. – Date of access: 26.06.2011.

23. Грищенко, В. Н. Методы и средства компонентного программирования / В. Н. Грищенко, Е. М. Лаврищева. – М.: Кибернетика и системный анализ, 2003. – С. 39–55.

24. Принципы проектирования и разработки программного обеспечения. Учебный курс MCSD / С. Ф. Уилсон, Б. Мэйплс. – 2-е изд., испр. – М.: Издательско-торговый дом «Русская Редакция», 2002. – 736 с.

25. Басс, Л. Архитектура программного обеспечения на практике / Л. Басс, П. Клементс, Р. Кацман. – 2-е изд. – СПб.: Питер, 2006. – 576 с.

26. Фаулер, М. Архитектура корпоративных программных приложений / М. Фаулер. – М.: Вильямс, 2007. – 236 с.

27. NET Data Access Architecture Guide [Electronic resource]. – Mode of access: http://msdn.microsoft.com/en-us/library/ms978510.aspx. – Date of access: 26.06.2011.

28. Найгард, М. Проектирование и развертывание готового к производственной эксплуатации ПО / М. Найгард. – New-York: Pragmatic Bookshelf, 2007. – 350 с.

29. Тестирование ПО [Электронный ресурс]. – Режим доступа: www.tester.com.ua. – Дата доступа: 26.06.2011.

30. Методология Microsoft Solutions Framework [Электронный ресурс]. – Режим доступа: http://www.microsoft.com/rus/msdn/msf/. – Дата доступа: 26.06.2011.

31. Липаев, В. В. Методы обеспечения качества крупномасштабных программных средств / В. В. Липаев. – М.: СИНТЕГ, 2003. – 520 с.

32. Гагарина, Л. Г. Технология разработки программного обеспечения: учеб. пособие / Л. Г. Гагарина; под ред. Л. Г. Гагариной. – М.: ИД «ФОРУМ»: ИНФРА-М, 2008. – 400 с.

33. Макгрегор, Д. Тестирование объектно-ориентированного программного обеспечения: практ. пособие / Д. Макгрегор, Д. Сайкс. – М.: ОООТИД «ДС», 2002. – 432 с.

34. Калбертсон, Р. Быстрое тестирование / Р. Калбертсон, К. Браун, Г. Кобб. – М.: Вильямс, 2008. – 374 с.

35. Головач, В. Дизайн пользовательского интерфейса / В. Головач. – М.: Символ-Плюс, 2009. – 94 с.

36. Разработка управляемого интерфейса / А. Ажеронок [и др.]. – М.: 1С-Паблишинг, 2010. – 728 с.

37. Проекты (работы) курсовые. Требования и порядок подготовки, представление к защите и защита: СТП БГТУ 002-2007. – Введ. 02.05.2007. – Минск: БГТУ, 2007. – 40 с.

38. Evaluating Software Architectures: Methods and Case Studies (SEI Series in Software Engineering) / P. Clements, R. Kazman, M. Klein. – Addison-Wesley Professional, ISBN-10: 020170482X, ISBN-13: 978-0201704822. 41Evans, E. Domain-Driven Design: Tackling Complexity in the Heart of Software [Electronic resource]. – Addison-Wesley, 2004. – Mode of access: http://www.microsoft.com/architectureguide. – Date of access: 26.06.2011.

39. Nilsson, J. Applying Domain-Driven Design and Patterns: With Examples in C# and NET / J. Nilsson. – New-York: Addison-Wesley, 2006. – 576 p.

40. Руководство Microsoft по проектированию архитектуры приложений [Электронный ресурс]. – Режим доступа: download.microsoft.com/documents/. – Дата доступа: 26.06.2011.

41. Architectural Blueprints – The “4+1” View Model of Software Architecture [Electronic resource]. – Mode of access: http://www.cs.ubc.ca/~gregor/teaching/papers/4+1view-architecture.pdf. – Date of access: 26.06.2011.

СОДЕРЖАНИЕ

Учебное издание

Пацей Наталья Владимировна
Шиман Дмитрий Васильевич
Сухорукова Ирина Геннадьевна

ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Учебно-методическое пособие

Редактор *Е. С. Ватеичкина*
Компьютерная верстка *Е. В. Ильченко*
Корректор *Е. С. Ватеичкина*

Подписано в печать 11.08.2011. Формат $60\times84^{1}/_{16}$.
Бумага офсетная. Гарнитура Таймс. Печать офсетная.
Усл. печ. л. 7,6. Уч.-изд. л. 7,8.
Тираж 100 экз. Заказ .

Издатель и полиграфическое исполнение:
УО «Белорусский государственный технологический университет».
ЛИ № 02330/0549423 от 08.04.2009.
ЛП № 02330/0150477 от 16.01.2009.
Ул. Свердлова, 13а, 220006, г. Минск.