Некоммерческое акционерное общество

АЛМАТИНСКИЙ УНИВЕРСИТЕТ ЭНЕРГЕТИКИ И СВЯЗИ

Кафедра компьютерных технологий

УТВЕРЖДАЮ
Декан ФАИТ
_____ Табултаев С.С.
«_____»_____ 2017 г.

TRPOSRV 5301- ТЕХНОЛОГИИ РАЗРАБОТКИ ПРОГРАММНОГО ОБСПЕЧЕНИЯ ДЛЯ СИСТЕМ РЕАЛЬНОГО ВРЕМЕНИ

Конспекты лекций для магистрантов специальности 6М070400 - «Вычислительная техника и программное обеспечение» (научное и педагогическое направление)

Алматы 2017

СОСТАВИТЕЛЬ: к.ф.-м.н. А.А. Аманбаев « Технологии разработки программного обеспечения для систем реального времени». Конспекты лекций для магистрантов специальности 6М070400 - «Вычислительная техника и программное обеспечение» (научное и педагогическое направление). – Алматы: АУЭС, 2017. - 59 с.

Протокол № _6_ заседания кафедры от «_ 6 »__06___2017 г.

Заведующий кафедрой ММиПО _____ М.Ж.Байсалова

Конспекты лекций рассмотрен и утвержден на заседании учебно-методической комиссии факультета Аэрокосмических и информационных технологий протокол № __5__ от «___20.06__» 2017 г.

Конспекты лекций составлены согласно рабочей программе дисциплины «Технологии разработки программного обеспечения для систем реального времени» и является ознакомление магистрантов с вопросами проектирования сложных программных систем, обучение их методологии структурного анализа и проектирования SADT, освоение ими основ объектно- ориентированного подхода к проектированию программных систем и приобретение практических навыков применения современных технологии проектирования (CASE-технологии).

Конспекты лекций предназначены для магистрантов специальности 6М070400 - «Вычислительная техника и программное обеспечение» .

Библиогр. – 18 наимен.

Рецензент:

Содержание

Лекция 1. Введение. Определение, классификации систем реального времени. Краткая история развития проектирования. 4

Лекция 2. Жизненный цикл программного обеспечения. Управление проектом 6

Лекция 3. Организация разработки программного обеспечения. Анализ требований. 10

Лекция 4. Организация разработки программного обеспечения. Определение образа и границ проекта. Управление изменениями требований 17

Лекция 5. Анализ и моделирование функциональной области внедрения. Архитектурное проектирование. 20

Лекция 6. Спецификация функциональных требований к ПО ИС. Архитектура распределенных систем. 24

Лекция 7. Методологии моделирования предметной области. Детальное проектирование. 27

Лекция 8. Моделирование бизнес-процессов средствами BPwm. Объектно-ориентированное проектирование 31

Лекция 9. Информационное обеспечение ИС. Анализ пригодности 37

Лекция 10. Моделирование информационного обеспечения. Модели реализации объектно-ориентированных программных систем. 40

Лекция 11. Унифицированный язык визуального моделирования Unified Modeling Language (UML). Проектирование пользовательского интерфейса. 43

Лекция 12. Этапы проектирования ИС с применением UML и Rational Rose. Тестирование объектно-ориентированных систем. 47

Список литературы 55

Лекция 1. Введение. Основные понятия технологии проектирования программных систем.

Проектирование программного обеспечения (ПО) представляет собой процесс построения приложений реальных размеров и практической значимости, удовлетворяющих заданным требованиям функциональности и производительности. *Программирование*-это один из видов деятельности, входящих в цикл разработки программного обеспечения. Другим видом деятельности является разработка программной документации, в которой находят отображения все виды работ по спецификации артефактов процесса разработки приложения.

Артефакты - это все, что сопровождает процесс разработки ПО. Набор артефактов включает: спецификации требований к ПО; проектные модели и диаграммы; исходный и объектный код; тестовые процедуры и тестовые варианты; результаты измерений продуктивности; объектные модули, планы, отчеты и т.д.

Технология разработки программного обеспечения охватывает разнообразные типы программ: автономные, встроенные, реального времени и сетевые.

Типичная схема разработки ПО состоит из следующей последовательности шагов:

- уяснить природу и среду применения предлагаемого продукта;
- выбрать процесс разработки и создать план;
- собрать требования;
- спроектировать и собрать продукт;
- выполнить тестирование продукта;
- выпустить продукт и обеспечить его сопровождение.

На первом шаге необходимо, прежде всего, уяснить суть проекта. Нужно составить представление о масштабах проекта и с этой целью оценить, какими сроками, финансами и персоналом мы располагаем.

Второй шаг связан с определением тех средств, при помощи которых будут отслеживаться вносимые в документацию и в программный код изменения. Этот процесс называется *управлением конфигурациями*. Далее, нужно определиться с самим процессом разработки, то есть выбрать одну из разновидностей: водопадная модель процесса, спиральная модель процесса, инкрементная модель процесса либо USDP - унифицированный процесс разработки ПО.

После этого обычно составляется общий план проекта, включающий в себя план- график (расписание проекта).

Третий шаг состоит в сборе требований к приложению. Он включает в себя, прежде всего, обсуждение проекта с заказчиками и другими участниками, заинтересованными в его выполнении.

На следующем шаге наступает черед проектирования и реализации самого продукта.

Программный продукт, как окончательный, так и промежуточный, подлежит тщательному тестированию.

И, наконец, когда продукт выпущен, наступает фаза его сопровождения, включающая внесение в него исправлений и улучшений.

В современной практике разработки ПО существует пять ключевых требований, сформулированных Хэмфри [13]. Первое из них - заранее выбрать свою шкалу измерения качества для проекта и продукта. Второе требование состоит в сборе информации по всем проектам с целью создания базы для оценки будущих проектов. Третье положение гласит, что все требования, схемы, программные коды и материалы тестирования должны быть легко доступны всем членам команды. Суть четвертого условия состоит в том, что все участники команды должны следовать избранному процессу разработки. Пятое требование состоит в том, что выбранные показатели качества должны постоянно измеряться и эти измерения должны протоколироваться.

Цель любого программного *проекта* состоит в производстве некоторого программного *продукта*. То, как в рамках проекта производится продукт, представляет собой *процесс*. Поскольку критичным для успеха дела является взаимодействие членов команды, поэтому необходимо включать в рассмотрение *персонал* (четыре «П» разработки программ).

В понятие продукта разработки включается не только программный код-сюда относятся различные артефакты. Кроме того, участвующие в процессе разработчики играют разнообразные роли, которые в USDP называются работниками .

В технологии разработки ПО различают *методы, средства и процедуры*.

Методы обеспечивают решение следующих задач:

- планирование и оценка проекта;
- анализ пользовательских и системных требований;
- проектирование сценариев, структур и потоков данных;
- кодирование;
- тестирование;
- сопровождение.

Средства (утилиты)- обеспечивают компьютерную поддержку методов.

Процедуры являются «клеем», который соединяет методы и утилиты так, что они обеспечивают непрерывную технологическую цепочку разработки. Процедуры определяют:

- порядок применения методов и утилит;
- формирование отчетов, форм по соответствующим требованиям;
- контроль, который помогает обеспечивать качество и координировать изменения;
- формирование «вех», по которым руководители оценивают прогресс.

Последовательность шагов, использующих методы, утилиты и процедуры, называют *парадигмами инженерии ПО.*

Применение парадигм гарантирует систематический, упорядоченный подход к промышленной разработке, использованию и сопровождению ПО.

Фактически, парадигмы вносят в процесс создания ПО организующее инженерное начало, необходимость которого трудно переоценить.

Основная литература – 2,5, 7.

Контрольные вопросы:

1. Что собой представляет процесс проектирования ПО?
2. В чем состоит различие между просто программированием и проектированием программного обеспечения ?
3. Какие типы программ и типичная схема разработки ПО?
4. Разновидности процесса разработки?
5. Содержание этапов классического жизненного цикла?
6. Какова основная цель макетирования?
7. Каковы стратегии конструирования программ: достоинства и недостатки соответствующих моделей?

Лекция 2. Жизненный цикл программного обеспечения. Управление проектом

Управление проектом заключается в управлении производством продукта в рамках отведённых средств и времени.

Управление проектом охватывает:

- инфраструктуру (организационные моменты);
- управляющий процесс (права и ответственности участников);
- процесс разработки (методы, инструменты, процедуры, языки, документация и поддержка);
- расписание (моменты времени, к которым должны быть представлены выполненные фрагменты работы).

Планировщики проекта могут варьировать стоимость, возможности, качество и дату завершения проекта. Руководитель проекта может управлять следующими факторами, которые, как правило, не являются фиксированными.

1. Общая стоимость проекта.
2. Возможности продукта.
3. Качество продукта.
4. Длительность проекта.

Например, показатели качества могут варьироваться. Если целевые значения показателей качества установлены слишком низко, это может привести к увеличению расходов на переделки из-за неудовлетворённости заказчика. Если же целевые значения показателей качества установлены слишком высоко, то затраты на поиски самой последней незначительной ошибки могут оказаться недопустимо высокими.

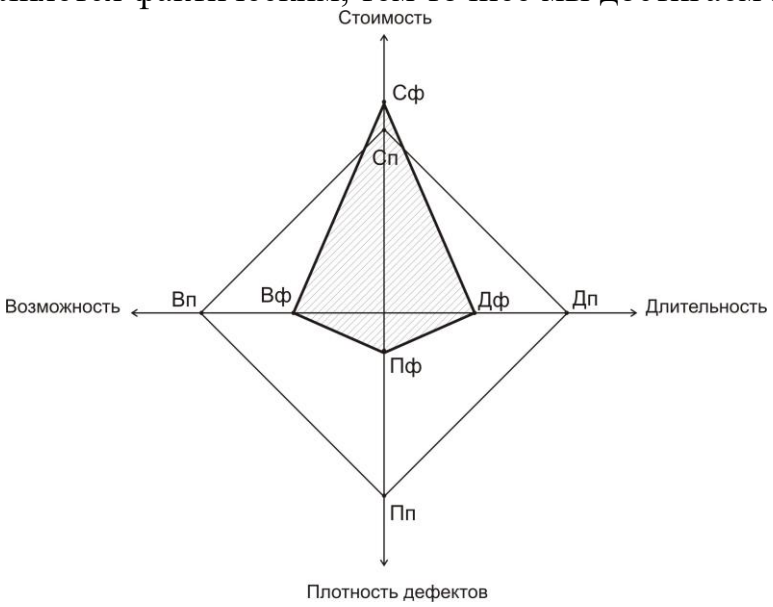
Возможности продукта также не являются фиксированными. Например,заказчик может отказаться от некоторого требования, если это сократит длительность проекта на 15%. Дата завершения проекта также может иногда сдвигаться. Например, руководитель может пожелать сдвинуть

срок разработки, если продукт обладает столь многообещающими возможностями, что есть шанс завоевать рынок.

Визуализация параметров проекта с помощью лепестковых диаграмм, позволяет менеджеру проекта правильно расставить приоритеты при принятии решений.

В лепестковых диаграммах значение каждой переменой (фактора) откладывается на оси, исходящей от центра, которому соответствуют наименее желательные показатели (факторы), а целевые значения откладываются на некотором планируемом расстоянии от центра.

Таким образом, получается четырехугольник. Например, на оси «Возможности» центр соответствует значению «0 % требований выполнено», а единица - «100 % требований выполнено». Фактические значения обычно находятся где-то в промежутке, хотя могут и выходить за пределы многоугольника, если удаётся перекрыть плановые значения показателей. Состояние проекта оценивается путем закрашивания многоугольника фактических значений. Чем больше целевой (плановый) многоугольник заполняется фактическим, тем точнее мы достигаем поставленных целей.



Вп – цель = 100% требований выполнено

Вф – 60% требований выполнено

Пф- цель = 4 дефекта на тысячу строк комментированного кода

Пп – 1 дефект на тысячу строк

Сп- цель = S 70 тыс.

Сф – S 90 тыс.

Дп- цель = 30 недель

Дф- 20 недель

Процессы управления

Типичная схема процесса управления проектом состоит из следующих действий, объединяемых общим названием - *планирование проекта*:

1. Написание предложений по созданию ПО.

Содержат описание целей проекта и способов их достижения. Они также включают в себя оценки финансовых и временных затрат на выполнение проекта.

2. Планирование и составление графика работ по созданию ПО.

На этапе планирования проекта определяются процессы, этапы и получаемые на каждом из этапов результаты, которые должны привести к выполнению проекта.

3. Оценивание стоимости проекта.

Определение стоимости проекта напрямую связано с его планированием, поскольку здесь оцениваются ресурсы, требующиеся для выполнения плана.

4. Контроль за ходом выполнения работ.

Мониторинг проекта- это непрерывный процесс, когда менеджер проекта постоянно отслеживает ход реализации проекта, сравнивая фактические и плановые показатели выполнения работ с их стоимостью.

5. Подбор персонала

Менеджеры проекта обычно обязаны сами подбирать исполнителей для своих проектов, в соответствии с их профессиональным уровнем.

Такой подбор имеет определенные ограничения и не является свободным.

6. Написание отчетов и предложений.

Заключается в посылке отчетов о ходе выполнения проекта, как заказчику, так и подрядным организациям. В этих отчётах должна быть та информация, которая позволяет четко оценить степень готовности создаваемого ПО.

Эффективное управление программным проектом напрямую зависит не только от плана проекта, но и от таких планов, как: план качества, план аттестации, план управления конфигурацией, план сопровождения ПО и, наконец, плана по управлению персоналом.

Процесс планирования проекта можно сформулировать с помощью следующего псевдокода:

Определение проектных ограничений.
Первоначальная оценка параметров проекта.
Определение этапов выполнения проекта и контрольных отметок.
ЦИКЛ (пока проект не завершится или не будет остановлен)
Составление графика работ.
Начало выполнения работ.
Ожидание окончания очередного этапа работ.
Отслеживание хода выполнения работ.
Пересмотр оценок параметров проекта.
Изменение графика работ.
Пересмотр проектных ограничений.
ЕСЛИ (возникла проблема)
ТО *Пересмотр технических или организационных параметров проекта.*

ВСЕ-ЕСЛИ

ВСЕ-ЦИКЛ

План проекта

План проекта должен четко показать ресурсы, необходимые для реализации проекта, разделение работ на этапы и временной график выполнения этих этапов. Можно представить план проекта как описание технологического процесса создания ПО. В таком плане обязательно присутствуют ссылки на планы других, разрабатываемых отдельно, видов.

Большинство планов последнего типа, содержат следующие разделы:

1. Введение
2. Организация выполнения проекта.
3. Анализ рисков.
4. Аппаратные и программные ресурсы, необходимые для реализации проекта.
5. Разбиение работ на этапы.
6. График работ.
7. Механизмы мониторинга и контроля за ходом выполнения проекта.

График работ и сетевые диаграммы. Составление графика работ-одна из самых ответственных работ, выполняемых менеджером проекта. Здесь менеджер оценивает длительность проекта, определяет ресурсы для реализации этапов и представляет эти этапы в виде согласованной последовательности. Процесс составления графика работ имеет следующий вид:

Создание графиков работ реализуется в виде временных и сетевых диаграмм, показывающие зависимости между различными этапами проекта. Разновидностями графиков работ могут быть: этапы проекта, сетевая диаграмма этапов, временная диаграмма деятельности этапов и распределение исполнителей по этапам.

Управление рисками. Важной частью работы менеджера проекта является оценка рисков, которые могут повлиять на график работ или на качество программного продукта, и разработка мероприятий по предотвращению риска.

Выделяются три типа рисков.

1. *Риски для проекта*, которые влияют на график работ или ресурсы.
2. *Риски для разрабатываемого продукта*, влияющие на качество или производительность разрабатываемого программного продукта.
3. *Бизнес-риски*, относящиеся к организации- разработчику или поставщикам.

Процесс управления рисками состоит из четырех стадий.

1. Отделение рисков.

2. Анализ рисков.

3. Планирование рисков.

4.Мониторинг рисков.

Основная литература - 5, 7.

Контрольные вопросы и упражнения:

1. Что такое управление проектом? Составляющие управления проектом.

2. Объясните, почему нематериальность програмных систем порождает особые проблемы в процессе управления программным проектом?

3. Объясните, почему процесс планирования проекта является операционным и почему план должен постоянно пересматриваться в течение всего срока выполнения проекта?

4. Определите, в дополнение к приведенным рискам, еще на ваш взгляд заслуживающих внимания.

5. Объясните суть и назначение графических способов представления графиков работ.

Лекция 3. Организация разработки программного обеспечения. Анализ требований

Чтобы построить что-либо, мы, прежде всего, должны понять, чем *это* должно быть. Процесс понимания и документирования этого и называется *анализом требований*.

Анализ требований (reguirements analysis) - процесс получения законченного письменного утверждения, которое определяет, какими должны быть функциональность, внешний вид, производительность и поведение приложения.

Анализ требований служит мостом между неформальным описанием требований, выполняемым заказчиком, и проектированием системы. Методы анализа призваны формализовать обязанности системы, фактически их применение дает ответ на вопрос: что должна делать будущая система?

Структурный анализ - один из формализованных методов анализа требований к ПО. Автор этого метода - Том Де Марко (1979) [14]. В этом методе программное изделие рассматривается как преобразователь информационного потока данных. Основной элемент структурного анализа - диаграмма потоков данных. Диаграмма потоков данных - графическое средство для изображения информационного потока и преобразований, которым подвергаются данные при движении от входа к выходу системы.

Как известно, элементами проблемной области для любой системы являются потоки, процессы и структуры данных. При структурном анализе активно работают только с потоками данных и процессами.

Методы анализа, ориентированные на структуры данных, обеспечивают:

1. определение ключевых информационных объектов и операций;

2. определение иерархической структуры данных;

3. компоновку структур данных из типовых конструкций -

последовательности, выбора, повторения;

4. последовательность шагов для превращения иерархической структуры данных в структуру программы.

Наиболее известны два метода: метод Варнье-Орра и метод Джексона [14].

Анализ требований к приложению является процессом осмысления и последующего выражения концепций в конкретной форме. Большинство недостатков, найденных в произведенном ПО, возникло на стадии анализа требований и такие недостатки труднее всего исправить.

Обычно требования выражают, что приложение должно делать: зачастую здесь не пытаются сформулировать, *как* добиться выполнения этих функций. Например, следующее выражение является требованием для бухгалтерского приложений.

Система должна предоставлять пользователю доступ к балансу его банковского счета.

Вообще говоря, следующее выражение не является требованием для приложения.

Балансы счетов клиентов будут храниться в таблице под названием "Балансы" в базе данных Access.

Второе выражение касается того, *как* должно быть построено приложение, а не того, *что* это приложение должно делать.

Результатом анализа требований является документ, который обычно называют спецификацией требований, или спецификацией требований к ПО (SRS - Software Reguarements Specification).

C - требования и D - требования

Анализ требований разделяется на два уровня [2]. Первый уровень документирует желания и потребности заказчика и пишется на языке, понятном заказчику. Результаты иногда называют требованиями заказчика, или C-требованиями. Первичной аудиторией для C-требований будет сообщество заказчиков, а уже вторичной - сообщество разработчиков. Второй уровень документирует требования в специальной, структурированной форме. Эти документы называются требованиями разработчика, или D-требованиями. Первичной аудиторией для D-требований будет сообщество разработчиков, а уже вторичной - сообщество заказчиков. Хотя целевые аудитории для C- и D-требований различны, заказчики и разработчики тесно сотрудничают при создании успешных продуктов.

Теория разработки программ и мировой опыт настаивают на аккуратном документировании требований. Без таких документов команда практически не знает, каких целей она пытается достичь, не может корректно проверить свою работу, проследить свою производительность, получить адекватные данные по своей работе, предсказать объем и усилия в своей следующей работе и удовлетворить своих заказчиков. Короче говоря, не может быть

профессиональной разработки без письменных требований, поэтому каждое требование должно:

- быть четко выражено;
- быть легко доступно;
- быть пронумеровано;
- сопровождаться подтверждающими тестами;
- предусматриваться проектом;
- быть учтено кодом;
- быть протестировано отдельно;
- быть протестировано вкупе с другими требованиями;
- быть подтверждено тестированием после того, как выполнена сборка приложения.

Типичная схема процесса анализа требований

1.Идентифицировать "заказчика", что значит определить заинтересованных лиц, то есть лиц, имеющих долю в результирующем продукте.

2. Провести интервью с представителями заказчика для того, чтобы

- определить желания и потребности;
- использовать инструмент поддержки;
- набросать графический интерфейс пользователя;
- определить конфигурацию оборудования.

3. Написать С-требования в форме стандартного документа.

4. Проверить С-требования.

5. Построить С-требования.

Для всех этапов необходимо отслеживать следующие метрики:

- затраченное время;
- количество страниц С-требований;
- количество времени общения с заказчиком на страницу;
- самооценка качества;
- оценка дефектов по проверкам.

Существует несколько способов организации SRS. Приведем оглавление стандарта IEEE 830-1993.

1. Введение.
1.1. Цель.
1.2. Область применения.
1.3. Определения, термины и сокращения.
1.4. Ссылка.
1.5. Обзор.
2. Общее описание.
2.1. Перспективы продукта.
2.1.1. Системные интерфейсы.
2.1.2. Пользовательские интерфейсы.
2.1.3. Аппаратные интерфейсы.
2.1.4. Программные интерфейсы.

2.1.5. Коммуникационные интерфейсы.
2.1.6. Ограничения по памяти.
2.1.7. Операции.
2.1.8. Требования по адаптации.
2.2. Функции продукта.
2.3. Пользовательские характеристики.
2.4. Ограничения.
2.5. Предположения и зависимости.
2.6. Распределение требований.
3. Детальные требования.
4. Сопровождающая информация.

Преимущество стандарта IEEE в том, что он однозначно решает большинство вопросов, которые можно было бы истолковать по-разному.

Анализ требований нужно рассматривать в контексте того, что:

- именно люди, бесспорно являются источниками возникновения требований;
- заказчики субъективны в своих требованиях;
- разработчики несут профессиональную ответственность, что может основательно влиять на требования;
- нужды заказчика несколько труднее определить, чем его желания;
- большая часть анализа требований является коммуникационной деятельностью, тщательно организованной для получения наилучших результатов.

Каким образом организовать коммуникационную деятельность при анализе требований подробно рассказано в [5,20]. В этих источниках показывается, как можно эффективно формулировать требования с помощью методов прецедентов и более традиционных форм выражения требований. В книгах описываются методы определения, реализации, верификации и проверки достоверности требований, а также ряд важнейших процессов разработки, которыми профессионально должна овладеть команда для успешного управления требованиями на протяжении жизненного цикла проекта.

Описание требований заказчика (С-требований)

Заказчики разрабатывают концепцию того, как их приложение будет работать. Эту концепцию иногда называют *моделью приложения*, или *концепцией работы*. Поскольку обычно заказчики не владеют технологией выражения таких концепций, инженеры могут предложить подходящие технологии, такие, как варианты использования, потоки данных или переходы состояний. Перечисленные техники также используются и в проектировании. Варианты использования - концепция, которую изобрел Д.Якобсон, является очень полезным способом выражения требований заказчика в форме взаимодействия приложения с внешним пользователем.

Вариант использования (Use Case) - некий целостный набор функций, имеющий определенную ценность для пользователя.

Варианты использования можно вывести в результате идентификации

задач для пользователя. Для этого следует задаться вопросом: "Каковы обязанности пользователя по отношению к системе и чего он ожидает от системы?". Варианты использования можно определить в результате непосредственного анализа функциональных требований. Во многих случаях функциональное требование отображается непосредственно в вариант использования.

Согласно Лешека Мацяшек [4], анализ требований может быть совместим с определением и спецификацией вариантов использования и выявлением объектов предметной области с помощью следующих таблиц:

- распределения требований по субъектам и вариантам использования;
- описательной спецификации варианта использования;
- соответствия функциональных требований и классов - сущностей. Все классы предметной области могут быть получены в результате анализа требований.

Описания требований заказчика могут быть получены также с помощью диаграмм потоков данных и диаграмм переходов состояний. Последние являются эффективным способом достижения соглашения разработчика и заказчика для приложений, управляемых событиями.

Дизайн пользовательского интерфейса входит в фазу проектирования ПО, однако его также можно считать и частью фазы анализа требований. Заказчики часто представляют себе приложение в форме графического пользовательского интерфейса (GUI), и хорошим способом помочь им описать программу является разработка эскиза GUI.

Основные руководящие принципы проектирования интерфейса, ориентированного на пользователя приведены в [4, Гл.7].

Отметим следующие шаги разработки пользовательских интерфейсов:

1. Узнайте своего пользователя (C) (обработка C-требований).
2. Поймите назначение проектируемой системы (C).
3. Примените принципы хорошего экранного дизайна (C,D).
4. Подберите подходящий тип окон (C,D).
5. Разработайте системные меню (C,D).
6. Выберите соответствующие аппаратные устройства управления (C).
7. Выберите соответствующие экранные элементы управления (C).
8. Организуйте и создайте раскладку окон (C,D).
9. Выберите подходящие цвета (D).
10. Создайте осмысленные значки (C,D).
11. Предоставьте эффективные сообщения, обратную связь и руководство (D).

Существует четыре способа формулирования требований заказчика.

1. Если требование простое и стоит само по себе, выразите его четкими предложениями.
2. Если требование представляет взаимодействие пользователя и приложения, выразите его с помощью варианта использования.
3. Если требование затрагивает элементы обработки, каждый из которых получает и выдает данные, используйте диаграмму потоков данных.

4. Если требование затрагивает состояния, в которых может находиться программа (или части программы), выполните следующие действия:

- определите все состояния программы;
- укажите исходное состояние;
- определите события (происходящие вне системы), которые приводят к переходу состояний;
- определите вложенные состояния.

Быстрое прототипирование и исследование осуществимости

Быстрое прототипирование - это частичная реализация целевой программы, в том числе обычно значительной части пользовательского интерфейса.

Быстрое построение прототипа - полезный способ установить требования заказчика, а также определить и упразднить рискованные части проекта. Студенческие проекты часто остаются в выигрыше от прототипа, при условии что прототип довольно скромен. Он дает возможность команде опробовать свой процесс разработки до начала работы над самим проектом. Построение недорогого прототипа поможет заказчику лучше понять, продукт какого типа будет выпущен, помогает программисту лучше понять, какой продукт они должны выпустить.

И, наконец, в анализе требований возможна ситуация перед разработчиком, заключающаяся в том, что, а могут ли вообще быть реализованы предложенные требования! В этом случае можно построить доказательство концепции. Это частичная реализация приложения или полномасштабная система с соответствующим ПО и оборудованием, похожая на предлагаемую к разработке.

Анализ требований: добавление детальных требований

Детальные требования (D-требования) - единственный документ, в котором определяется конкретная природа программы, Уровень детализации должен быть полным, но не чрезмерным.

Для разработчиков ПО D- требования являются базой для проектирования и разработки.

Типичная схема процесса анализа D- требований состоит:

1. Выбрать систему организации D- требований.
2. Создать диаграммы последовательности для вариантов использования.
3. Получить D- требования из C- требований и от заказчика.
4. Сделать черновик планов тестирования.
5. Провести инспектирование.
6. Утвердить с заказчиком (обратная связь с п.3).
7. Выпустить, когда компонент утвержден заказчиком.

Типы D- требований.

Существуют несколько типов требований:

1. Функциональные требования:

- отражают функциональность приложения

2. Нефункциональные требования:

1) Производительность: скорость, пропускная способность, использование памяти;

2) Надежность и доступность;

3) Обработка и доступность;

5) Ограничения: на точность, на инструменты и язык, технологию проектирования, используемые стандарты и платформы.

3. Обратные требования - перечисление того, чего программа не будет делать. Эта классификация применима как к С-, так и к D- требованиям. Однако она становится более важной при написании D- требований, поскольку она руководит процессами разработки и тестирования в разных аспектах.

Свойства D- требований.

Для того, чтобы все детали проекта были охвачены и не пропущены, вводятся следующие *свойства,* которыми должны обладать D- требования:

1) Прослеживаемость:

- прослеживаемость функциональных требований;
- контроль нефункциональных требований.

2) Пригодность к тестированию и однозначность.

3) Приоритет, то есть назначение приоритетов требованиям путем классификации требований по категориям: важные, желательные и необязательные.

4) Полнота, то есть самодостаточность требования.

5) Состояние ошибки- проверка требования на событие ошибка.

6) Согласованность – отсутствие противоречий между требованиями.

Способы организации D- требований.

D- требования можно организовать с помощью нескольких схем:

-*по основным свойствам*, то есть когда требования сгруппированы по различным свойствам программы;

-*по режиму работы программы*;

-*по вариантам использования*, или по сценариям. Здесь идея заключается в том, что большинство детальных требований являются частью варианта использования;

-по *классу;*

-по *иерархии функций*, то есть путем функциональной декомпозиции программы;

-*по состояниям*, то есть путём указания детальных требований, применимых к каждому состоянию, в которых может находиться программа. Внутри классификации каждого состояния перечислены события, влияющие на программу, находящуюся в конкретном состоянии.

Можно посоветовать организовывать D- требования в комбинацию классификаций.

Способ организации D- требований часто связан с возможной архитектурой приложения.

Метрики для анализа D- требования.

Метрики контроля качества детальных требований включают в себя:

-метрики того, насколько хорошо написаны требования;

-метрики эффективности проверки требований;

-метрики эффективности процесса анализа требований;

-метрики полноты требований.

Основная литература- 2 3,5.

Контрольные вопросы:

1.В чем разница между C- и D- требованиями?

2. В чем достоинства и недостатки разделения требований на категории C и D?

3. Какие задачи решает анализ требований?

4. Что такое диаграмма потоков данных?

5. Каковы особенности диаграммы управляющих потоков?

6. Какова организация диаграммы переходов – состояний?

7. Какие задачи решают методы анализа, ориентированные на структуры данных?

8.Что такое вариант использования в контексте формулирования требований заказчика?

9. У типичной программы есть много требований. Назовите важную проблему, сопутствующую созданию требований и работе с ними.

10.Назовите несколько категорий, желаемых свойств и способов организации детальных требований.

Лекция 4. Организация разработки программного обеспечения. Определение образа и границ проекта. Управление изменениями требований

Бизнес-требования - составляют высший уровень абстракции в цепи требований: они определяют образ и границы системы. Бизнес-требования – высокоуровневая бизнес-цель организации, которая строит продукт, или клиента, который приобретает продукт.

Пользовательские и функциональные требования к ПО должны находиться в соответствии с контекстом и целями, устанавливаемыми бизнес-требованиями. Четкое представление образа и границ проекта особенно важно при разработке сложного, распределенного ПО. Положение о рамках и ограничениях проекта необычайно полезно при обсуждении предлагаемых функций и целевых выпусков, при принятии решений об изменении и расширении требований.

Образ продукта (product vision) - выстраивает работу всех заинтересованных лиц в одном направлении. Он описывает, *что* продукт представляет собой сейчас и *каким он станет* впоследствии. *Границы проекта* (project scope) - показывают, к какой области конечного долгосрочного образа продукта будет направлен текущий проект.

Границы более динамичны, чем образ. Задача планирования заключается в управлении границами определенного проекта, как определенным подмножеством большого стратегического образа.

Бизнес-требования и варианты использования

Бизнес-требования определяют как набор бизнес-задач (вариантов использования), которые позволяют выполнять приложение (ширина приложения), так и глубину уровня, до которого реализуется каждый вариант использования. Глубина простирается от простой реализации до полной автоматизации с множеством вспомогательных средств.

Бизнес-требования влияют на приоритеты реализации вариантов использования и связанные с ними функциональные требования. Бизнес-требования также существенно влияют на способ реализации требований.

Документ об образе и границах собирает бизнес-требования в единый документ, который подготавливает основу для последующей разработки продукта. Таким документом может быть следующий шаблон документа об образе и границах проекта.

1. Бизнес-требования
1.1 Исходные данные
1.2 Возможности бизнеса
1.3 Бизнес-цели и критерии успеха
1.4 Потребности клиента или рынка
1.5 Бизнес-риски
2. Образ решения
2.1 Положение об образе проекта
2.2 Основные функции
2.3 Предположения и зависимости
3. Масштабы и ограничения проекта
3.1 Объем первоначально запланированной версии
3.2 Объем последующих версий
3.3 Ограничения и исключения
4. Бизнес-контекст
4.1 Профили заинтересованных лиц
4.2 Приоритеты проекта
4.3 Операционная среда

Раздел Бизнес-требования описывает: основные преимущества, которые новая система даст ее заказчикам, покупателям и пользователям; содержит общее описание предыстории или ситуации, в результате которых было принято решение о создании продукта; существующие рыночные возможности и рынок, на котором продукту придется конкурировать с другими продуктами; суммирует важные преимущества бизнеса, предоставляемые продуктом в количественном и измеряемом виде; потребности типичных покупателей или целевых сегментов рынка, которые решит новая система.

Раздел Образ решения – определяет стратегический образ системы, позволяющей выполнять бизнес- задачи; сжатое положение об образе проекта, обобщающее долгосрочные цели и назначение нового продукта, а также предположения и зависимости.

Раздел Масштабы и ограничения проекта определяют рамки и ограничения, то есть *что* может делать, а *что не может делать* система; обобщает основные запланированные функции первоначальной версии продукта и последующих версий.

Раздел Бизнес-контекст обобщает некоторые бизнес-проблемы проекта, включая профили основных категорий заинтересованных лиц и приоритеты управления.

Большинство разработчиков сталкиваются с необходимостью включения и (или) изменения требований к ПО. Определенные изменения требований абсолютно приемлемы, неизбежны и даже благоприятны. Бизнес-процессы, рыночные возможности, конкурирующие продукты и технологии – все они могут меняться в ходе разработки продукта. Первый шаг при управлении незапланированным ростом объема – документирование образа, границ и ограничений новой системы, как части бизнес-требований. Необходимо оценивать каждое предложенное требование или функцию, соотносясь с бизнес-целями, образом продукта и границами проекта. Приемами управления могут быть: задействование клиентов в сборе информации, составление прототипов реализации продукта, короткие циклы разработки при инкрементальной разработке системы, умение сказать: «Нет» на предложенное требование.

Политика контроля изменений содержит следующие основные положения: все изменения должны вноситься в соответствии с процессом; запрос на изменение еще не гарантирует выполнение этого изменения; содержимое базы данных изменений должно быть видимым для всех заинтересованных в проекте лиц; каждое включенное изменение должно отслеживаться вплоть до утверждения запроса на это изменение и т.д.

Основная литература – 5, 9,12

Контрольные вопросы:

1. Что является признаком того, что бизнес-требования недостаточно ясно определены?

2 В чем заключается задача планирования?

3. Что определяют возможности бизнеса для коммерческого продукта?

4. Что определяют возможности планирования для корпоративной информационной системы?

5. Укажите шаги и приемы при управлении незапланированным ростом объема проекта.

Лекция 5. Анализ и моделирование функциональной области внедрения. **Архитектурное проектирование**

В настоящее время в IT-индустрии интенсивно развивается дисциплина программных архитектур и проектирования. Теперь мы можем говорить об архитектуре высокого уровня и архитектуре низкого уровня в терминах, понятных всем профессиональным разработчикам.

Любое приложение имеет аппаратные и программные компоненты. *Системная разработка* - это процесс анализа и проектирования, который разделяет приложение на аппаратные и программные компоненты. Некоторые аспекты этой декомпозиции диктуются требованиями заказчика, другие определяются разработчиками.

Процесс системной разработки начинается с определения общих системных требований. Затем делается выбор оптимального соотношения между аппаратным и программным обеспечением. Затем к программой части применяется технология разработки, начиная с анализа требований и т.д.

Создание *архитектуры программы*- это проектирование на *самом высоком уровне*. Оставшуюся часть процесса проектирования называют *детальным проектированием*.

Принципиальная проблема системы ПО – это их сложность. Хороший способ борьбы со сложностью – модуляризация или декомпозиция задачи на подзадачи. Критериями успешности декомпозиции являются такие характеристики, как связь и сцепление. *Связность внутри модуля* –это сила взаимосвязей между элементами модуля. *Сцепление* характеризует степень взаимодействия между модулями. Эффективная модульность достигается максимизацией связности и минимизацией сцепления.

Малое сцепление в совокупности с большой связностью очень важны при проектировании приложений, ввиду постоянного процесса внесения изменений в проекты. Архитектуры с малым сцеплением и большой связностью более приспособлены для модификации, поскольку изменения в таких архитектурах имеют наиболее локальный эффект. Однако создавать такие архитектуры достаточно сложно.

В качестве примера рассмотрим декомпозицию персонального финансового приложения.

- Счета (проверка, сохранение и т.д.);
- Оплата счетов (электронная, чеком и т.д.);
- Глобальные отчеты (общие активы, задолженность и т.д.);
- Займы (автомобиль, образование, дом и т.д.);
- Инвестиции (акции, долговые обязательства и т.д.);

Хотя такая декомпозиция привлекательна с точки зрения пользователя, она имеет большие недостатки как архитектурная декомпозиция. Например, Счета имеют малую связность, поскольку они слабо взаимосвязаны. А связность модулей здесь довольно велика. Например, при выплате задолженности задействуются модули Счета, Оплаты счетов, Займы и, возможно, Отчеты.

Существует следующая альтернатива этой декомпозиции:

- Интерфейс (пользовательский и коммуникационный интерфейсы, отчетность и т.д.);

- Поставщики (арендодатель, ссуды, коммунальные услуги и т.д.);
- Активы (проверка счетов, акции, долговые обязательства и т.д.);

Модели, каркасы, образцы проектирования и компоненты

Декомпозиция проекта на компоненты (модули) является существенным шагом, но предстоит гораздо большая работа по созданию архитектуры. Для начала необходимо согласовать декомпозицию с такими *моделями*, как: *варианты использования, классы и переходы состояний*.

Обычно, любое приложение описывается с нескольких точек зрения, то есть проекций. В мире программирования проекции называются моделями.

Существуют:

Модель вариантов использования – коллекция вариантов использования, поясняющая, что именно приложение должно делать.

Модель классов – объясняет построение блоков, из которых будет сформировано приложение. Внутри модели классов (объектной модели) можно показать методы и атрибуты.

Модель компонентов – коллекция диаграмм потоков данных, которая объясняет, каким образом приложение будет работать в терминах перемещения данных.

Модель переходов состояний – коллекция диаграмм переходов состояний, которая определяет момент времени, в который приложение осуществляет свою работу.

Архитектура приложения зачастую выражается в терминах одной из моделей и поддерживается остальными моделями. В каждой архитектуре есть, по меньшей мере, одна модель классов, способная реализовать эту архитектуру.

Каркасы – это *коллекция классов*, используемых в нескольких различных приложениях. Часто классы внутри каркаса взаимосвязаны. Они могут быть абстрактными и использоваться через наследование.

Модель классов приложения, таким образом, получается из классов предметной области – в результате анализа требований. Классы каркаса получаются из разработки архитектуры для приложений, подобных тому, которые мы проектируем. Таким образом, классы каркаса либо являются частью ранее существовавшего пакета, либо разработаны, как в процессе анализа архитектуры. *Проектные классы* – оставшиеся классы, добавленные для завершения проектирования.

Образец проектирования – это найденная опытным путем комбинация компонентов, обычно классов или объектов, которая решает определенные общие проектировочные задачи. Все образцы проектирования (паттерны - ***pattern***) разделяются на структурную, креационную и поведенческую категории. Структурные образцы имеют дело со способами представления объектов, креационные – способами создания сложных объектов, поведенческие – позволяют следить за поведением объектов.

Компоненты – повторно используемые объекты, которые не требуют знания ПО, использующего их. Например, COM и Java Beans. Одни

компоненты используются другими с помощью агрегирования и взаимодействуют в основном с помощью событий.

Типы архитектур и их модели

Рассмотрим некоторые типы архитектур и укажем паттерны, которые могут помочь в реализации этих архитектур.

Архитектуры, основанные на потоках данных. Потоки данных являются наиболее общим способом отображения архитектур.

Архитектура независимых компонентов и паттерн Facade состоят из компонентов, работающих параллельно и время от времени общающихся друг с другом. Например, в WWW, где сервера и браузеры все время работают параллельно и иногда общаются между собой.

Архитектура параллельных взаимодействующих процессоров. Такая архитектура характеризуется тем, что в ней одновременно запускаются несколько процессов.

Архитектуры событийно – управляемых систем и паттерн State. Архитектура виртуальных машин рассматривает приложение как программу, написанную на специальном языке.

Репозиторная архитектура – архитектура, построенная главным образом вокруг данных, и предназначена для обработки транзакций по отношению к базам данных.

Уровневые архитектуры – это логически связанная коллекция артефактов ПО, обычно это *пакеты классов*.

В UML пакет трактуется как коллекция классов. В общем случае UML позволяет помещать в этот архитектурный компонент любые материалы, связанные с приложением, в том числе код, результаты проектирования, документацию и пр., например, содержимым пакетов могут быть классы, реализующие: объекты интерфейса с пользователем; сценарии вариантов использования; объекты предметной области; интерфейс с базой данных; внутренние структуры данных, такие как деревья, списки и т.д.; исключения , для обработки нештатных ситуаций и т.п.

Процедура выбора архитектуры

1. Разбейте систему на замкнутые модули.

2. Сравните со стандартными типами архитектур. Улучшите декомпозицию.

• Есть поток данных в пакетах между обрабатывающими станциями?

- Архитектура последовательных данных.

• Обрабатывающие станции ожидают получения входных данных, чтобы начать свою работу?

-Архитектура каналов и фильтров.

• Процессы выполняются параллельно?

-Архитектура параллельных взаимодействующих процессов.

• Процесс обеспечивает обслуживание пользовательских процессов?

-Клиент – серверная архитектура.

• Процесс реагирует только на происходящие события?

-Системы, управляемые событиями.

• Приложение состоит из процессов, которые выполняются по сценарию?

-Образец проектирования Interpreter.

• Приложение строится для хранилища данных?

-Репозиторные архитектуры.

• Существует упорядочение по уровням?

-Уровневые архитектуры.

Можно предложить еще один способ выбора архитектуры, как продолжение первого.

3. Сделайте выбор среди представленных альтернативных архитектур.

4. Добавьте к классам, полученным на основе анализа требований, классы, обеспечивающие согласование с выбранной архитектурой.

Например, в системе, управляемой событиями, это могут быть классы, контролирующие переходы между состояниями.

5. Примените существующий каркас и (или) образец проектирования, если найдете полезный.

6. Распределите классы по пакетам (4-8 пакетов). Каждый должен иметь смысл в контексте приложения.

7. Удостоверьтесь, что связность между частями высока. Низкое сцепление будет подтверждением правильного выбора.

8. Рассмотрите возможность добавления façade – класса (объекта) для управления интерфейсами пакетов.

Основная литература -1,4,5

Контрольные вопросы:

1. Поясните понятие модуля и модульности. Зачем используют модули?
2. Что такое связность модуля?
3. Какие существуют типы связности?
4. Что такое сцепление модуля?
5. Какие существуют типы сцепления?
6. Объясните, почему архитектурное проектирование системы должно предшествовать разработке формальной спецификации?
7. Для чего необходимо архитектурное проектирование ПО?
8. Образцы проектирования, категории образцов и что они позволяют на этапе разработки архитектуры ПО?
9. Модельные точки зрения и модели МДА?
10. В чем заключается концепция МДА?
11. Какие модели существуют в МДА?
12. На каких этапах МДА используется язык UML?

Лекция 6. Спецификация функциональных требований к ПО ИС. Архитектура распределенных систем

Целью темы является изучение архитектуры распределенных программных систем. Освоив, эту тему студент должен:

-знать основные преимущества и недостатки распределенных систем;

-иметь представление о различных подходах, используемых при разработке архитектур клиент/сервер;

-понимать различия между архитектурой клиент/сервер и архитектурой распределенных объектов;

-знать концепцию брокера запросов к объектам и принципы, реализованные в стандартах CORBA.

В настоящее время практически все большие программные системы являются распределенными.

Распределенной называется система, в которой обработка информации сосредоточена не на одной вычислительной системе, а распределена между несколькими компьютерами.

Модель архитектуры клиент/сервер – это модель распределенной системы(РС), в которой показано распределение данных и процессов между несколькими процессорами.

Все современные программные системы делятся на три класса.

- *Прикладные программные системы* (ПС), для работы на одном компьютере или рабочей станции (текстовые процессоры, электронные таблицы, графические системы и т.п.).
- *Встроенные системы* (системы управления бытовыми устройствами, приборами и т.п.) работающие на одном, либо группе процессоров.
- *Распределенные системы* на слабо интегрированной группе параллельно работающих процессоров, связанных через сеть (банкоматы, издательские системы, системы ПО коллективного пользования и т.п.).

Существует шесть основных характеристик распределенных систем.

1. Совместное использование ресурсов.
2. Открытость.
3. Параллельность.
4. Масштабируемость.
5. Отказоустойчивость.
6. Прозрачность.

Проблемы проектирования распределенных систем связаны с : идентификацией ресурсов; коммуникацией, качеством системного сервиса и архитектурой ПО.

Задача разработчиков РС – спроектировать программное или аппаратное обеспечение так, чтобы предоставить все необходимые характеристики РС.

Выделяются несколько родственных типа архитектур РС:

1.*Многопроцессорная архитектура* – самая простая РС, состоящая из множества различных процессов, которые могут (но необязательно) выполняться на разных процессорах. Данная модель часто используется в больших системах реального времени.

2. *Архитектура клиент/сервер*. В этой модели система представляется как набор сервисов, предоставляемых серверами клиентам. Такая

архитектура должна отражать логическую структуру, разрабатываемого приложения.

В рамках данной архитектуры приложение может иметь трехуровневую структуру: *уровень представления*, обеспечивающий информацию для пользователя и взаимодействие с ними; *уровень выполнения* – реализующий логику работы приложения и *уровень управления данными*, на котором выполняются все операции с базами данных.

Самой простой архитектурой клиент/сервер является двухуровневая, в которой приложение состоит из сервера (или множества идентичных серверов) и группы клиентов. Существует два вида такой архитектуры: *модель тонкого клиента и модель толстого клиента*.

Приведем рекомендации, по применению разных типов архитектуры клиент/ сервер.

Двухуровневая архитектура тонкого клиента - в наследуемых системах, в которых нецелесообразно разделять выполнение приложения и управление данными, а также в приложениях, в которых обрабатываются большие массивы данных (запросов), но с небольшим объемом вычислений в самом приложении.

Двухуровневая архитектура толстого клиента – где требуется интенсивная обработка данных, а также в приложениях с постоянным набором функций, на стороне пользователя, применяемых в среде с хорошо отлаженным системным управлением.

Трехуровневая и многоуровневая архитектуры клиент/сервер – в больших приложениях с сотнями и тысячами клиентов, в которых часто меняются данные и методы обработки, а также в приложениях, в которых выполняется интеграция данных из многих источников.

3. Архитектура распределенных объектов.

В модели клиент/сервер распределенной системы клиент запрашивает сервисы только у сервера, но не у других клиентов; серверы могут функционировать как клиенты и запрашивать сервисы у других серверов, но не у клиентов, и наконец, клиенты должны знать о сервисах, предоставляемых определенными серверами, и о том, как взаимодействуют эти серверы. Такая модель не обеспечивает поддержку масштабируемости и не предоставляет средства включения клиентов в систему на распределенных серверах.

Более общим подходом является проектирование архитектуры системы как архитектуры распределенных объектов. В этой архитектуре основными компонентами системы являются объекты, предоставляющие набор сервисов через свои интерфейсы. Такие объекты могут располагаться на разных компьютерах в сети и взаимодействовать посредством *промежуточного слоя* (программной шины), называемого *брокером запросов к объектам*.

В процессе проектирования систем архитектуру распределенных объектов можно использовать либо в виде логической модели, позволяющей структурировать и спланировать систему, либо как гибкий подход к реализации систем клиент/сервер. Примером системы с такой архитектурой

может служить система обработки данных, хранящихся в разных базах данных.

Главным недостатком архитектуры распределенных объектов является сложность их проектирования, по сравнению с системами клиент/сервер.

4. CORBA

В архитектуре распределенных объектов существуют проблемы, связанные с тем, что объекты могут быть реализованы на разных языках программирования, могут запускаться на разных платформах и их имена не должны быть известны всем другим объектам системы. Поэтому промежуточное ПО должно выполнять большую работу для того, чтобы поддерживалось постоянное взаимодействие объектов.

В настоящее время для поддержки распределенных объектных вычислений существует два основных стандарта промежуточного ПО:

- CORBA(Common Object Reguest Broker Architekture- архитектура брокеров запросов к общим объектам);
- DCOM (Distributed Component Object Model- объектная модель распределенных компонентов).

Стандарты CORBA описывают четыре основных элемента: модель объектов; брокер запросов к объектам, который управляет запросами к сервисам объектов; совокупность сервисов объектов и совокупность общих компонентов, построенных на верхнем уровне основных сервисов.

В модели CORBA объект инкапсулирует атрибуты и сервисы как обычный объект. Вместе с тем в объектах CORBA еще должно содержаться определение различных интерфейсов, описывающих глобальные атрибуты и операции объекта. Интерфейсы объектов CORBA определяются на стандартном универсальном языке описания интерфейсов - IDL. Интерфейс IDL отделяет объекты от брокера, поэтому реализацию объектов можно изменять не затрагивая другие компоненты системы.

Основная литература – 7

Контрольные вопросы:

1. Объясните, почему распределенные системы всегда более масштабируемы, чем централизованные. Какой вероятный предел масштабируемости программных систем?
2. В чем основное отличие между моделями толстого и тонкого клиента в разработке систем клиент/ сервер?
3. Объясните, почему использование распределенных объектов совместно с брокером запросов к объектам упрощает реализацию масштабируемых систем клиент/ сервер.
4. Какие базовые средства должен предоставлять брокер запросов к объектам?

Лекция 7. Методологии моделирования предметной области. Детальное проектирование

Детальное проектирование - это полный объем работ по проектированию, исключая архитектуру и реализацию. Оно содержит определение классов, связывающих классы предметной области и классы архитектуры.

Детальное проектирование – это техническая деятельность, которая следует за выбором архитектуры. Основной целью этой деятельности является как можно более полная подготовка проекта к его реализации, то есть к созданию программного кода.

Существует определенная взаимосвязь вариантов использования, архитектуры и детального проектирования. Варианты использования являются частью требований, отталкиваясь от которых разработчики выбирают архитектуру, после чего они разрабатывают детальный проект для реализации требуемых вариантов использования, с учетом выбранной архитектуры.

На первом шаге проектирования ПО варианты использования фиксируются как часть требований. На втором шаге проектирования варианты использования используются для определения классов предметной области. На третьем шаге – разрабатывается программная архитектура и добавляются соответствующие классы проекта. Последний шаг заключается в проверке того факта, что архитектура и детальный проект удовлетворяют требуемым вариантам использования.

Типичная схема детального проектирования .

- Начинайте с *архитектурных моделей*, то есть, моделей классов предметной области и архитектуры; общей модели переходов состояний, потоков данных и модели вариантов использования.
- Представьте классы и образцы проектирования, которые включают в себя классы архитектуры и предметной области.
- Совершенствуйте модели, обеспечивая их непротиворечивость
- Для каждого класса определите их инварианты.
- Для каждого метода используйте пред- и пост- условия, блок схемы и псевдокод.
- Набросайте план модульного тестирования.
- Проинспектируйте планы тестирования и проектирования.
- Запускайте в реализацию.

В USDP детальное проектирование имеет место в основном во время итераций проектирования и конструирования. USDP поддерживает три стереотипа классов на уровне анализа, не связанных с классами проектирования: *классы сущности, граничные классы и классы управления*.

Проектирование с использованием интерфейсов можно сравнить с заключением контракта, причем элементы приложения, использующие эту основу функциональности, следует проектировать так, чтобы они «не знали» подробности реализации функциональности.

Следующим приемом детального проектирования является повторно используемые компоненты, развитию которого способствовало широкое

распространение объектно – ориентированных, объектно-подобных и других парадигм компонентов. Примером повторного использования программного кода является применение библиотеки Microsoft MFC, элементов управления Visual Basic, объектов COM, Java Beans и других классов Java API. Стандартом для распределенного повторного использования является архитектура CORBA консорциума OMG. Веб – приложения (не компоненты), также часто бывают повторно используемыми, и, наконец, STL – библиотека стандартных шаблонов, применяется к различным структурам данных и к объектам практически любых классов.

В детальном проектировании, в качестве инструментов для проектирования, используются диаграммы последовательности и диаграммы потоков данных. Покажем соответствующие способы усовершенствования моделей для детального проектирования

Диаграммы последовательности

1. Начните с диаграмм последовательности, созданных для детальных требований или архитектуры, относящихся к вариантам использования.
2. Если это необходимо, представьте дополнительные варианты использования для показа взаимодействия проектируемых частей с остальным приложением.
3. Обеспечьте полную детализацию диаграмм последовательности.

- убедитесь, что все объекты и их классы специфицированы;
- выберите конкретные имена функций, вместо естественного языка для представления операций.

Диаграммы потоков данных

1. Соберите диаграммы потоков данных(ДПД), созданные для детальных требований и(или) архитектуры.
2. При необходимости, представьте дополнительные диаграммы потоков данных, чтобы пояснить потоки данных и процесс.
3. Выясните, каким частям других моделей соответствуют диаграммы потоков данных
4. Обеспечьте полную детализацию диаграмм потоков данных:

- Выясните природу процесса, протекающего в каждом узле;
- Выясните тип передаваемых данных;
- Если описание процесса требует большей детализации, раскройте узлы процесса в диаграмме.

Диаграммы последовательности, построенные по варианту использования, представляют классы этого варианта использования и взаимодействие объектов этих классов согласно сценария, по системе методов, которые представляют собой словесные описания функциональности моделируемого варианта использования. Поскольку все методы, необходимые для реализации этого варианта использования, теперь известны, можно представить их на объектной модели. Продолжая процесс анализа работы объектов в сценарии реализации варианта использования, мы получим детальные модель классов и модель вариантов использования.

Для детального проектирования модели данных должны детально описываться, а затем отображаться на функции или классы и методы.

Следующий шаг детального проектирования – специфицирование классов, методов и атрибутов.

Синтаксис представления атрибута класса имеет вид:

Видимость имя [множественность]: тип = начальное значение {характеристики}

где , *Видимость* – public, protected либо private;

м*ножественность* – количество экземпляров свойства;

характеристики – дополнительные данные об атрибуте, например, характеристика **frozen** говорит о том, что после инициализации объекта значение этого свойства не изменяется.

Синтаксис представления *операции* класса следующий:

видимость имя (список параметров) :тип результата{характеристики},

где *характеристики* – дополнительные данные об операции, например, **leaf** означает неполиморфную операцию;

В свою очередь, спецификация списка параметров имеет вид:

направление имя : тип = значение по умолчанию,

где *направление* **– in** означает входной параметр, **out** – выходной, **inout** – входной параметр, который может модифицироваться.

Одним из полезных средств создания спецификации классов является CORBA IDL(Interface Definition Language – язык определения интерфейса). Это стандартный текстовый формат для описания интерфейсов, обеспечиваемых наборами классов, их атрибутов и функций.

Спецификация алгоритмов

После определения функций, которые необходимо реализовать, могут оказаться полезным описать используемый алгоритм, кратко останавливаясь на исходном коде. Инструментами таких описаний могут быть: блок-схемы, псевдокод, диаграммы деятельности.

Особое место в детальном проектировании занимают образцы проектирования, которые должны использоваться только в благоприятных условиях, поскольку в противном случае, они могут только усложнить обычное проектирование.

Каждый образец проектирования обычно имеет два аспекта. Первый аспект – это обычная модель классов, описывающая классы и их взаимосвязь. Второй аспект представляет способ действия образца, для описания которого может быть применена модель вариантов использования.

Приведем один из способов применения образцов проектирования в детальном проектировании.

1. Ознакомьтесь с проблемами проектирования, которые можно решить с помощью образцов проектирования.

- уясните различия между креационными(К), структурными(С) и поведенческими(П) образцами проектирования;

- рассмотрите по порядку каждый из этапов детального проектирования.

2. Выясните, к чему относится данная задача : к созданию чего либо сложного (К), к представлению сложной структуры комбинации объектов (С) или она связана с поведением приложения (П), то есть с фиксацией выбранного режима состояния объектов.

3. Выясните, существует ли образец проектирования, пригодный для решения этой задачи.

4. Решите, что перевешивает – преимущества или недостатки

- преимущества обычно включают в себя высокую гибкость;

- недостатки обычно заключаются в повышенной сложности и низкой эффективности.

Креационные образцы проектирования – позволяют разнообразить создание объектов и управлять ими, в том числе координировать группы объектов, создаваемые в ходе работы приложения. К креационным образцам относятся: Singleton,Factory, Prototype, Abstract Factory, Builder.

Структурные образцы проектирования – представляют комплексные объекты, такие как списки, коллекции и деревья, в виде объектов с удобными интерфейсами. К структурным образцам относятся: Composite, Decorator,Adapter,Façade,Flyweight и Proxy.

Образцы проектирования, основанные на поведении – позволяют варьировать поведение приложения в ходе его работы, иначе пришлось бы проектировать и реализовывать каждый вариант поведения по отдельности. К поведенческим образцам относятся : Interpriter, Observer, State, Iterator, Mediator

Таким образом, в результате проведения детального проектирования, план проекта становится более конкретным во многих отношениях. В частности, теперь с большей точностью может быть проведена оценка стоимости проекта, графики работ могут быть разбиты вплоть до конкретных задач, которые, в свою очередь, могут быть распределены между разработчиками.

Основная литература: 1, 2, 3, 5

Контрольные вопросы:

1.Что такое детальное проектирование?

2.Какое место в детальном проектировании занимают варианты использования и архитектура?

3.Какова типичная схема процесса детального проектирования?

4.В чем суть проектирования по схеме USDP?

5.В чем суть проектирования с использованием интерфейсов? Каким образом следует проектировать элементы приложения, использующие требуемую функциональность?

6.Какое место занимают в детальном проектировании детальные диаграммы последовательности и диаграммы потоков данных?

7.Что такое инварианты класса? Инварианты, пред-и постусловия функций?

8.Когда следует использовать блок- схемы и псевдокоды?

9. Три категории образцов проектирования и что они позволяют реализовать на этапе детального проектирования?

10. Библиотеки стандартных шаблонов (STL) C++ и детальное проектирование.

Лекция 8. Моделирование бизнес-процессов средствами BPwm. Объектно-ориентированное проектирование

Объектно –ориентированное проектирование представляет собой стратегию, в рамках которой разработчики системы вместо операций и функций мыслят в понятиях *объекты*. Программная система состоит из взаимодействующих объектов, которые имеют собственное локальное состояние и могут выполнять определенный набор операций, определяемый состоянием объекта.

Под процессом объектно – ориентированного проектирования подразумевается проектирование классов объектов и взаимоотношений между этими классами. Когда проект реализован в виде исполняемой программы, все необходимые объекты создаются динамически с помощью определений классов.

Объектно – ориентированное проектирование - только часть объектно-ориентированного процесса разработки системы, где на протяжении всего процесса создания ПО используется объектно- ориентированный подход.

Этот подход подразумевает выполнение трех этапов.

- *Объектно ориентированный анализ* – метод анализа, согласно которому требования рассматриваются с точки зрения классов и объектов, составляющих словарь предметной области. Здесь объекты анализа отражают реальные объекты – сущности, а также определяются операции, выполняемые объектами.
- *Объектно – ориентированное проектирование* - 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 – ориентированное программирование* – 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-ориентированная декомпозиция – 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-ориентированные системы – совокупность автономных и в определенной мере независимых объектов.

Объекты и классы объектов

Объект – это нечто, способное пребывать в различных состояниях и имеющее определенное множество операций. Состояние определяется как набор значений атрибутов объекта. Операции представляют сервисы другим объектам (клиентам) для выполнения определенных вычислений. Объекты создаются в соответствии с определением класса объектов, которое служит шаблоном для создания объектов.

Взаимодействие между объектами осуществляется посредством запросов к сервисам (вызов методов) из других объектов и, при необходимости, путем обмена данными, требующимися для поддержки сервиса. Копии данных, необходимых для работы сервиса, и результаты работы сервиса передаются как параметры. Взаимодействие между объектами могут быть синхронным и асинхронным и моделируется с помощью различных типов ассоциаций [16]

Общая модель взаимодействия объектов позволяет их одновременное выполнение в виде параллельных процессов. Такие объекты могут выполняться на одном или разных компьютерах как распределенные объекты, в отличие от модели последовательного выполнения, в которой запросы к сервисам объектов и вызовы функций реализованы одним и тем же способом.

За счет механизма потоков (threads), который позволяет создавать параллельно выполняющиеся объекты, объектно-ориентированную архитектуру программной системы можно преобразовать так, чтобы объекты стали параллельными процессами.

Существуют два типа параллельных объектов.

1.*Серверы,* в которых объект реализован как параллельный процесс с методами, соответствующими определенным операциям объекта. Методы запускаются в ответ на внешнее сообщение (событие) и могут выполняться параллельно с методами, связанными с другими объектами.

2. *Активные объекты*, у которых состояние может изменяться посредством операций, выполняющихся внутри самого объекта. Процесс, представляющий объект, постоянно выполняет эти операции, а следовательно, никогда не останавливается.

Серверы наиболее полезны в распределенных средах, где вызывающий и вызываемый объекты, постоянно выполняются на разных компьютерах. Активные объекты используются там, где объектам необходимо обновлять свое состояние через определенные интервалы времени.

Процесс объектно-ориентированного проектирования

Первый этап в любом процессе проектирования состоит в выявлении взаимоотношений между проектируемым программным обеспечением и его

окружением, чтобы обеспечить необходимую функциональность системы и понять, как структурировать систему, чтобы она могла эффективно взаимодействовать со своим окружением. Таким образом, мы должны построить две дополняющие друг друга модели:

1. Модель *окружения системы* – это статическая модель , которая описывает другие системы из окружения разрабатываемого ПО.

2. Модель *использования системы* - динамическая модель, которая показывает взаимодействие данной системы со своим окружением.

Модель окружения системы можно представить с помощью схемы связей, которая дает простую блок-схему общей архитектуры системы (см. рис 7.1)

С помощь пакетов языка UML, ее можно представить в развернутом виде как совокупность подсистем. *Пакет* – это способ организации элементов модели в более крупные блоки, которыми впоследствии позволяется манипулировать как единым целым.

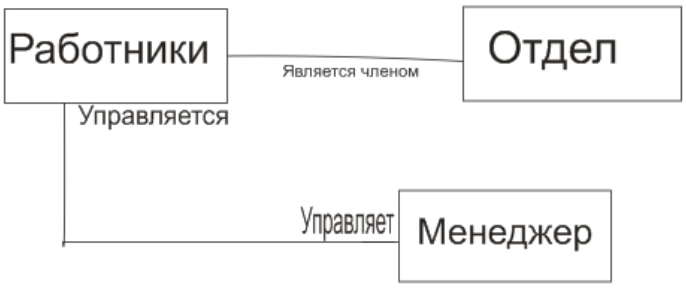


Рис. 9.1 Модель связей

Хорошо спроектированный пакет группирует семантически близкие элементы, которые имеют тенденцию изменяться совместно, причем доступ к содержимому пакета извне строго контролируется.

Самыми хорошими технологическими характеристиками является вариант проектирования системы, при котором каждый пакет включает интерфейс , содержащий описание всех ресурсов данного пакета, и взаимодействие пакетов осуществляется только через этот интерфейс. Изменения реализации ресурсов пакета в этом случае не затрагивает других пакетов. И только изменения в интерфейсе могут потребовать изменения пакетов, использующих ресурсы данного пакета.

Выделяются следующие группы классов или пакеты:

- пользовательский интерфейс – классы, реализующие объекты интерфейса с пользователем;
- библиотека интерфейсных компонентов – классы реализующие интерфейсные компоненты: окна, кнопки, метки и т.п.;
- объекты управления – классы, реализующие сценарии вариантов ипользования;

- объекты задачи - классы, реализующие объекты предметной области ;
- интерфейс базы данных – классы, реализующие интерфейс с базой данных ;
- база данных;
- базовые структуры данных - классы, реализующие внутренние структуры данных, такие, как деревья, списки и т.п.;
- обработка ошибок – классы исключений, реализующие обработку нештатных ситуаций.

Последние два пакета, как правило, объявляются глобальными, так как их элементы могут использовать классы всех пакетов.

В качестве примера на рис.7.2 показаны пакеты, которые организуют в классическую трехуровневую архитектуру классы, являющиеся частями представления информационной системы с точки зрения проектирования

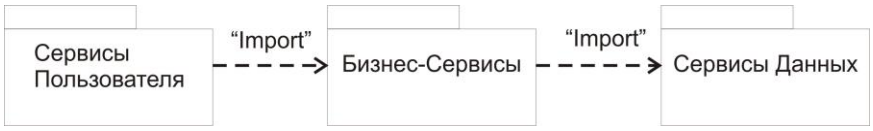


Рис.9.2 Моделирование групп элементов.

Элементы пакета *Сервисы пользователя* предоставляют визуальный интерфейс для ввода и вывода информации. Элементы пакета *Сервисы данных* обеспечивают доступ к данным и их обновление. Пакет *Бизнес-сервисы* является связующим звеном, он охватывает все классы и другие элементы, отвечающие за выполнение бизнес задачи, - в том числе бизнес правила, которые диктуют стратегию манипулирования данными в ответ на запросы пользователя.

При моделировании взаимодействия системы с ее окружением применяется абстрактный подход языка UML, который состоит в разработке модели вариантов использования, в которой каждый вариант использования представляет собой определенное взаимодействие с системой. Простейшая модель варианта использования в виде диаграммы вариантов использования для банка, показана на рис.7.3

Существуют различные форматы (шаблоны) для описания вариантов использования (Use Case): полный формат шаблона, свободный формат, формат в две колонки, шаблон Rational Unified Process (RUP), а также методика известного специалиста по Use Case Элистера Кокберна [6].

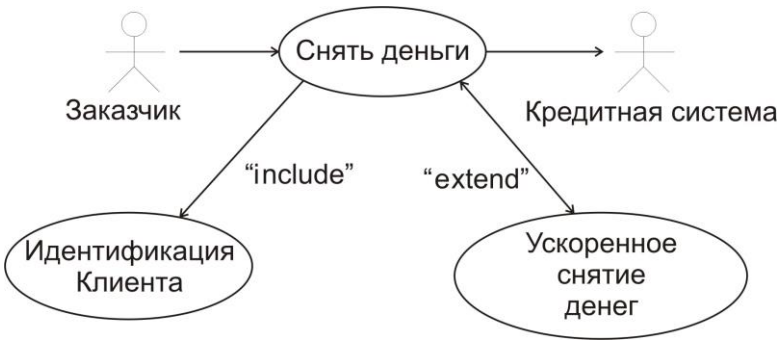


Рис 9.3 Диаграмма вариантов использования для банка.

Нельзя забывать, что варианты использования описывают поведение системы в соответствии с требованиями пользователя, при этом происходит идентификация классов объектов, операций и взаимодействия объектов. Для такого представления можно рекомендовать следующие таблицы [4]: таблица распределения требований по субъектам и вариантам использования; таблица спецификации вариантов использования; таблица установления действий в основном и альтернативных потоках; таблица соответствия функциональных требований и классов – сущностей.

Определение объектов

Одним из принципов объектно-ориентированного проетирования является определение кроме основных объектов – задачи, также и всех других объектов системы, точнее *классов объектов*. Классы объектов на этапе проектирования получают более детальное описание, с точки зрения свойств, операций и взаимодействий между ними.

Существует множество подходов к определению классов объектов.

1. Использование грамматического анализа естественного языкового описания системы.

2. Использование в качестве объектов ПО событий, объектов и ситуаций реального мира из области приложения.

3. Применение подхода, при котором разработчик с начала полностью определяет поведение системы, а затем определяются компоненты системы, отвечающие за различные режимы работы, при этом основное внимание уделяется тому, кто инициирует и кто осуществляет данные режимы. Компоненты системы, отвечающие за основные режимы работы и считаются объектами.

4. Применение подхода, основанного на сценариях использования системы. При этом для каждого сценария идентифицируются необходимые объекты, атрибуты и операции.

Модели архитектуры

Модели системной архитектуры показывают объекты и классы объектов, составляющих систему, и при необходимости типы взаимоотношений между этими объектами. Такие модели служат мостом

между требованиями к системе и её реализацией, то есть должны быть достаточно абстрактными, и в то же время содержать достаточное количество информации. Следовательно, в процессе проектирования важно решить, какие требуются модели и какой должна быть степень их детализации.

Существует два типа объектно-ориентированных моделей системной архитектуры.

1. *Статические модели*, описывающие статическую структуру системы в терминах классов объектов и взаимоотношений между ними: обобщения, зависимости и агрегирования.

2. *Динамические модели*, описывающие динамическую структуру системы и взаимодействия между объектами системы (но не классами объектов): в виде последовательности составленных объектами запросов к сервисам и описания реакции системы на взаимодействия между объектами.

Рассмотрим три модели, из множества, поддерживаемых языком моделирования UML.

1. *Модели подсистем*, которые показывают логически сгруппированные объекты. Они представлены с помощью диаграмм классов, в которых каждая подсистема обозначается как пакет. Модели подсистем являются статическими.

2.Модели последовательности, которые показывают последовательность взаимодействий между объектами. Они представляются в UML с помощью диаграмм последовательности или кооперативных диаграмм. Это динамические модели.

2. *Модели конечного автомата*, которые показывают изменение состояния отдельных объектов в ответ на определенные события. В UML они представлены в виде диаграмм состояний. Модели конечного автомата являются динамическими.

К другим моделям относятся: модели вариантов использования, модели объектов, модели обобщения и наследования, модель агрегирования.

Таким образом, главное преимущество объектно-ориентированного подхода к проектированию системы состоит в том, что он упрощает задачу внесения изменений в системную архитектуру, поскольку представление состояния объекта на оказывает на архитектуру никакого влияния.

Основная литература – 1,7,2,

Контрольные вопросы и упражнения.

1. Почему в проектировании систем применение подхода, который полагается на слабо связанные объекты, скрывающие информацию о своем представлении, приводит к созданию системной архитектуры, которую затем можно легко модифицировать?.

2 Покажите на примерах разницу между объектом и классом объектов?

3. При каких условиях можно разрабатывать систему, в которой объекты выполняются параллельно?

4. Определите возможные объекты в системах “Запись студентов на университетские курсы”, “Интернет - магазин ”.

5. Запишите точные определения интерфейсов на языке C++,C# для объектов, определенных в упражнении 4.

6. Нарисуйте диаграмму последовательности для системы “Запись студентов на университетские курсы”.

7. Нарисуйте диаграмму классов для системы “Интернет - магазин ”.

Лекция 9. Информационное обеспечение ИС. Анализ пригодности

В объектно-ориентированном проектировании большую роль играют выявление и систематизация всех структурных и поведенческих тонкостей предметной области. Одним из известных современных подходов для этих целей служит, разработанный Дугом Розенбергом и Кендаллом Скоттом процесс ICONIX [9]. Методология процесса заключается в *распределении поведения* между классами. Для этого используются четыре этапа проектирования: *моделирование предметной области, моделирование вариантов использования, анализ пригодности и построение диаграмм последовательности*.

Процесс ICONIX представляет собой нечто среднее между очень громоздким рациональным унифицированным процессом (Rational Unified Process - RUP) и весьма компактной методологией экстремального программирования. Процесс ICONIX , как и RUP , основан на вариантах использования, но не характеризуется множеством его недостатков. В этом процессе тоже применяется язык моделирования UML, однако основное внимание уделяется анализу требований. Процесс ICONIX , по представлению Айвара Джекобсона, основан на вариантах использования, вот почему с помощью этого процесса создаются вполне конкретные и простые для понимания варианты использования, которые легко использовать для разработки системы.

В объектно-ориентированных системах структура кода определяется классами. Поэтому, прежде чем приступать к кодированию, нужно выяснить, какие классы понадобятся. С этой целью следует нарисовать одну или несколько диаграмм классов. Для каждого класса необходимо описать все атрибуты, то есть данные-члены, и операции, которые представляют собой функции программы. Другими словами, мы должны идентифицировать все функции и удостовериться, что у нас есть данные, с которыми эти функции должны работать. Нужно будет показать, как функции и данные инкапсулируются в классах. В конечном счете мы намерены получить очень детальные диаграммы классов *уровня проектирования* – такого уровня детализации, при котором диаграмма классов может служить шаблоном для создания кода.

Вопрос таким образом, заключается в том, как перейти от вариантов использования к диаграммам классов уровня проектирования. В UML для этой цели лучше всего подходят *диаграммы последовательности* –

идеальное средство для принятия решений о распределении поведения. Решения о приписывании поведения классам принимаются по мере создания и анализа диаграмм последовательности.

Сложность проектирования заключается в том, как перейти от вариантов использования к диаграммам последовательности! В большинстве случаев это непростая задача, так как варианты использования описывают систему на уровне требований, а диаграммы последовательности – дают детальное представление с точки зрения проектирования. *Именно преодоление пропасти между «что» и «как» - центральная тема в процессе ICONIX*.

Первоначально диаграммы пригодности были включены в язык UML лишь частично. Они появились в работе Айвара Джекобсона и были утверждены в стандарте UML как дополнения. Важность этого вида диаграмм для моделирования никоим образом не оспаривается.

Диаграмма пригодности – диаграмма классов анализа, на которой вместо обычных прямоугольников – символов классов – изображаются пиктограммы трех видов: граничные объекты, управляющие объекты и сущностные объекты.

Граничные объекты – объекты системы, с которыми непосредственно взаимодействуют актеры. Это могут быть: экранные формы, диалоговые окна и меню, элементы GUI.

Сущностные объекты - таблицы БД, файлы, результаты поиска, а также все доменные объекты(понятия) предметной области.

Управляющие объекты (контроллеры) – объекты, заключающие в себе логику приложения. В этих объектах инкапсулируются бизнес-правила и стратегии, тем самым локализуются все их изменения и при этом не затрагиваются интерфейс пользователя и схемы базы данных. Эти объекты служат соединением между пользователями и хранимыми данными.

Анализ пригодности для варианта использования, выполняется путем исследования его текста с изображением *актеров, граничных, сущностных и управляющих объектов*, а также исследованием связей между различными элементами на диаграмме пригодности. Исследование должно проводиться как с основными, так и с альтернативными сценариями варианта использования, а вся нотация должна поместиться на одной диаграмме.

К основным правилам общения между элементами диаграммы пригодности следует отнести: актеры могут общаться только с граничными объектами; граничные объекты –только с управляющими объектами и с актерами; сущностные объекты – только с управляющими объектами; управляющие объекты – с граничными, сущностными и с другими управляющими объектами, но только не с актерами.

Порядок анализа варианта использования

1. Внимательно прочесть описание последовательности действий в тексте варианта использования.

2. Нарисовать диаграмму пригодности и проверить соответствие выявленной последовательности действий варианта использования с

последовательностью ассоциаций на диаграмме пригодности, и при необходимости, переписывать текст варианта использования для устранения неоднозначностей, а также для введения явных ссылок на граничные и сущностные объекты.

Результаты анализа пригодности используются не только для улучшения текста варианта использования, но и для постоянного уточнения статической модели системы (диаграммы классов).

Основная литература – 9.

Контрольные вопросы:

1. Что такое объекты «реального мира» (предметной области) и какие между ними ассоциации ?
2. Каковы базовые принципы процесса ICONIX ?
3. Каковы особенности процесса ICONIX ?
4. Приведите краткое описание основных этапов процесса ICONIX .
5. Какие вехи должны быть в объектно-ориентированном процессе ? и какие требования предъявляются к этому процессу ?

Лекция 10. Моделирование информационного обеспечения. Модели реализации объектно-ориентированных программных систем

Статические и динамические модели описывают логическую реализацию системы, отражают логический мир программного приложения. *Модели реализации* – обеспечивают представление системы в физическом мире, рассматривая вопросы упаковки логических элементов в компоненты и размещения компонентов в аппаратных узлах.

Основной задачей логического проектирования при объектном подходе является разработка классов для реализации объектов, полученных при объектной декомпозиции, что предполагает полное описание полей и методов каждого класса.

Физическое проектирование при объектном подходе включает объединение классов и других программных ресурсов в программные компоненты, а также размещение этих компонентов на конкретных вычислительных устройствах.

Компонентная диаграмма – первая из двух разновидностей диаграмм реализации, моделирующих физические аспекты объектно-ориентированных систем. Компонентная диаграмма показывает организацию набора компонентов и зависимости между ними. Элементами компонентных диаграмм являются компоненты и интерфейсы, примечания, ограничения, а также пакеты или подсистемы, используемые для группировки элементов модели в крупные фрагменты.

Компонент – это физический фрагмент реализации системы, который заключает в себе программный код (исходный, двоичный, исполняемый), сценарные описания или наборы команд операционной системы (командные файлы).

По UML *компонент* – это физическая и заменяемая часть системы, которая соответствует набору интерфейсов и обеспечивает реализацию этого набора интерфейсов. Компоненты имеют только операции, которые доступны через их интерфейсы.

Компонент является базисным строительным блоком *физического представления ПО*. Класс – базисный строительный блок *логического представления ПО*.

Интерфейс – список операций, которые определяют услуги класса или компонента. Очень важна взаимосвязь между компонентами и интерфейсом. Возможны два способа их отображения. В первом способе интерфейс изображается в форме пиктограммы (рис.8.1)

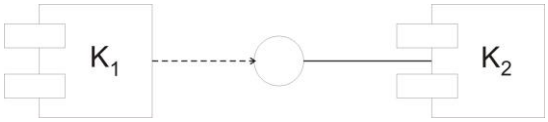


Рис.9.4 Представление интерфейса в форме пиктограммы.

Здесь компонент $К_2$, который реализует интерфейс, соединяется со значком интерфейса (кружком), простой линией. Компонент $К_1$, который использует интерфейс, связан с ним отношением зависимости. Второй способ (рис.8.2), использует развернутую форму изображения интерфейса, в которой могут показываться его операции.

Рис.9.5 Развернутая форма представления интерфейса.

Компонент $К_2$, который реализует интерфейс, подключается к нему отношением реализации. Компонент $К_1$, который получает доступ к услугам другого компонента через интерфейс, по – прежнему подключается к интерфейсу отношением зависимости.

Тот факт, что между двумя компонентами всегда находится интерфейс, устраняет их прямую зависимость. Компонент, использующий интерфейс, будет функционировать правильно вне зависимости от того, какой компонент реализует этот интерфейс. Это очень важно и обеспечивает гибкую замену компонентов в интересах развития системы.

Повторное использование компонентов – магистральный путь развития программного инструментария, так как приводит к более надежному и дешевому коду. Основная цель программных компонентов – допускать сборку системы из двоичных заменяемых частей. Они должны обеспечить начальное создание системы из компонентов, а затем ее развитие – добавление новых компонентов и замену старых без перестройки системы в

целом. Ключ к воплощению такой возможности - интерфейсы. После того, как интерфейс определен, к выполняемой системе можно подключить любой компонент, который удовлетворяет ему или обеспечивает этот интерфейс. Механизм замены компонента другим оговорен современными компонентными моделями (COM, COM+, CORBA, JavaBeans), требующими незначительных преобразований или предоставляющими утилиты, которые автоматизируют механизм.

Компонентные диаграммы в проектировании используют для моделирования статического представления реализации системы. Представление поддерживает управление конфигурацией системы, составляемой из компонентов.

- периода компиляции (среди текстовых компонентов);
- периода сборки, линковки (среди объектных двоичных компонентов);
- периода выполнения (среди машинных компонентов).

Реализация системы может включать большое количество разнообразных компонентов: исполняемых элементов, динамических библиотек, файлов данных, справочных документов, файлов инициализации, файлов регистрации, сценариев и файлов установки. Моделирование перечисленных компонентов, отношений между ними – важная часть управления конфигурацией системы.

Диаграммы размещения

Это вторая разновидность диаграмм реализации UML, моделирующих физические аспекты объектно-ориентированных систем. Диаграмма размещения показывает конфигурацию обрабатываемых узлов в период работы системы, а также компоненты, “живущие” в этих узлах. Диаграммы размещения могут включать компоненты, содержать пакеты или подсистемы, используемые для группировки элементов модели в крупные фрагменты, и даже содержать объекты.

Узел – это физический элемент, который существует в период работы системы и представляет собой компьютерный ресурс, имеющий память, а возможно, и способность обработки.

Диаграммы размещения используют для моделирования статического представления того, как размещается система. Это представление поддерживает распространение, поставку и инсталляцию частей, образующих физическую систему.

Графически диаграмма размещения – это граф из узлов, соединенных ассоциациями. Экземпляры узлов могут содержать экземпляры

компонентов, живущих или запускаемых в узлах, которые могут содержать объекты.

Изобразим типовую трехуровневую систему в виде диаграммы размещения (рис. 8.3)

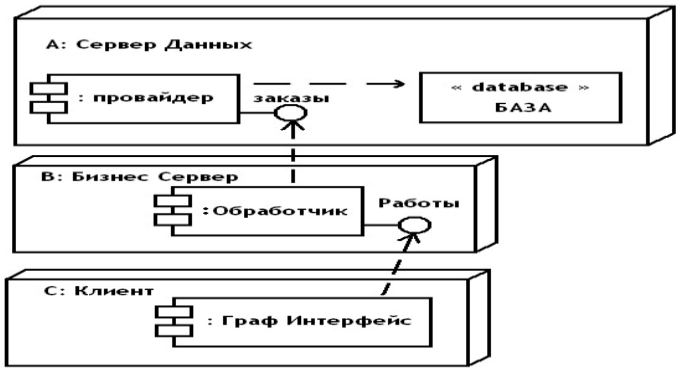


Рис. 9.6 Моделирование размещения компонентов

Уровень базы данных реализован экземпляром A, узла Сервер Данных.

Уровень бизнес – логики представлен экземпляром B, узла Бизнес Сервер.

Уровень графического интерфейса пользователя образован экземпляром C, узла Клиент.

Иногда полезно определить физическое распределение компонентов по процессорам и другим устройствам, например, в случае распределенных систем.

Основная литература – 1, 2, 12.

Контрольные вопросы

1. В чем основное назначение моделей реализации?
2. Где применяют диаграммы компонентов и размещения?
3. Чем отличается узел от компонента?
4. Чем полезен интерфейс?
5. Чем COM - объект отличается от обычного объекта?
6. Как описывается COM – интерфейс, и как он реализуется?
7. Что такое сервер COM – объекта, и какие типы серверов вы знаете?

Лекция 11. Унифицированный язык визуального моделирования Unified Modeling Language (UML). Проектирование пользовательского интерфейса.

Проектирование графического интерфейса пользователя (GUI-*Graphical User Interface)* представляет собой междисциплинарную деятельность. Оно требует усилий многофункциональной бригады - один человек, как правило, не обладает знаниями, необходимыми для реализации многоаспектного подхода к проектированию GUI-интерфейса.

Поэтому надлежащее проектирование GUI-интерфейса требует объединения навыков художника- графика, специалиста по анализу требований, системного проектировщика, программиста, эксперта по технологии, специалиста в области социальной психологии, а также, возможно, некоторых других специалистов, в зависимости от характера системы.

Разработчики должны:

- знать основные принципы проектирования интерфейса пользователя;
- освоить различные стили взаимодействия пользователя с программными системами;
- знать разные стили представления информации и то, в каких случаях целесообразно графическое представление данных;
- знать основные правила проектирования средств поддержки пользователя, встроенных в ПО;
- иметь представление об основных показателях удобства использования систем.

Грамотно спроектированный интерфейс пользователя крайне важен для успешной работы системы. Сложный в применении интерфейс, как минимум, приводит к ошибкам пользователя. Иногда они просто отказываются работать с программной системой, несмотря на ее функциональные возможности.

Итерационный процесс проектирования пользовательского интерфейса можно изобразить следующим образом (рис.9.1).

Наиболее эффективным подходом к проектированию интерфейса пользователя (UI) является разработка с применением моделирования пользовательских функций. В начале процесса прототипирования создаются бумажные эскизы (макеты) интерфейса, а затем разрабатываются экранные формы, моделирующие взаимодействие с пользователем. Важным этапом процесса проектирования UI является *анализ деятельности пользователей системы.* Для анализа, как правило одновременно , применяются различные методики, а именно: анализ задач, этнографический подход, опросы пользователей и наблюдения за их работой.

Рис 9.7 Процесс проектирования интерфейса пользователя.

Принципы проектирования интерфейса пользователя. Основой принципов проектирования UI являются человеческие возможности. К основным принципам, применимым при проектировании любых UI

относятся: контроль – на стороне пользователя; учет знаний пользователя ; согласованность; минимум неожиданностей; способность к восстановлению; руководство пользователя; учет разнородности пользователей.

Взаимодействие с пользователем. Заключается в решении вопросов о том, как пользователь будет вводить данные в систему и как данные будут представлены пользователю. Существует пять стилей взаимодействия: непосредственное манипулирование; выбор из меню; заполнение форм; командный язык и естественный язык. В одном приложении может использоваться одновременно несколько разных стилей. Например, в операционной системе MS Windows, поддерживается несколько стилей: прямое манипулирование пиктограммами, представляющими файлы и папки, выбор команд из меню, ручной ввод некоторых команд,таких как команды конфигурирования системы, использование форм (диалоговых окон)

Пользовательские интерфейсы приложении World Wide Web базируются на средствах, предоставляемых языками разметки Web- страниц вместе с другими языками, которые связывают программы с компонентами Web-страниц. В Web- приложениях можно создавать интерфейсы, в которых применялся бы стиль прямого манипулирования, что пока представляет достаточно сложную, в аспекте программирования, задачу.

Представление информации.

Хорошим тоном при проектировании систем считается отделение представления данных от самих данных. В этом случае изменения в представлении данных на экране пользователя происходят без изменения самой системы. Каждое представление имеет связанный с ним объект контроллера, который обрабатывает введенные пользователем данные и обеспечивает взаимодействие с устройствами. Такой подход реализован в паттерне Обозреватель (Observer), который используется тогда, когда необходимы разные представления состояния объекта. Он выделяет нужный объект и представляет его в разных формах.

Чтобы найти наилучшее представление информации, необходимо знать, с какими данными работают пользователи и каким образом они применяются в системе.

Принимая решение по представлению данных, разработчик должен учитывать ряд следующих факторов:

1.Что нужно пользователю - точные значения данных или соотношения между значениями?

2. Насколько быстро будут происходить изменения значений данных и нужно ли их немедленно показывать пользователю?

3. Должен ли пользователь предпринимать какие-либо действия в ответ на изменения данных?

4. Нужно ли пользователю взаимодействовать с отображаемой информацией посредством интерфейса с прямым манипулированием?

5. Информация должна отображаться в текстовом или числовом формате? Важны ли относительные значения элементов данных?

В представлении информации правильное использование цветов делает интерфейс пользователя более удобным для понимания и управления. Приведем некоторые правила эффективного использования цвета в UI:

- Используйте ограниченное количество цветов.
- Используйте разные цвета для показа изменений в состоянии системы.
- Для помощи пользователю используйте цветовое кодирование.
- Используйте цветовое кодирование продуманно и последовательно.
- Осторожно используйте дополняющие цвета.

Средства поддержки пользователя.

Интерфейс пользователя всегда должен обеспечивать некоторый тип оперативной справочной системы. Справочные системы - один из основных аспектов проектирования UI. Справочную систему приложения составляют: сообщения, генерируемые системой в ответ на действия пользователя, диалоговая справочная система и документация, поставляемая с системой.

Сообщения об ошибках.

Первое впечатление, которое пользователь получает при работе с программной системой, основывается на сообщениях об ошибках. Неопытные пользователи, совершив ошибку, должны понять, появившееся сообщение об ошибке. Сообщение об ошибке должно быть связано с контекстно - зависимой справкой .

Проектирование справочной системы. Справочная система должна предоставлять разные типы информации для помощи пользователю в затруднительных ситуациях. Для этого, справочная система должна иметь разные средства и разные структуры сообщений. Все справочные системы имеют сложную сетевую структуру, в которой каждый раздел справочной информации может ссылаться на несколько других информационных разделов. Структура такой сети, как правило, иерархическая, с перекрестными ссылками.

Справочную систему можно реализовать в виде группы связанных Web-страниц или с помощью обобщенной гипертекстовой системы, интегрированной с приложением. Иерархическая структура легко реализуется в виде гипертекстовых ссылок. Web- системы имеют преимущества: они просты в реализации и не требуют специального ПО.

Документация пользователя.

Для того, чтобы документация, поставляемая совместно с программной системой, была полезна всем системным пользователям, она должна содержать пять следующих документов: функциональное описание; документ по инсталляции системы; вводное руководство, описывающее ее «повседневное» использование; справочное руководство администратора. Для опытных пользователей системы удобны разного рода предметные указатели, которые помогают быстро просмотреть список возможностей системы и способ их использования.

Оценивание интерфейса.

Это процесс, в котором оценивается удобство использования интерфейса и степень его соответствия требованиям пользователя. Таким образом , оценивание интерфейса является частью общего процесса тестирования и аттестации систем ПО. Существуют простые методики оценивания интерфейсов пользователя:

- Анкеты, в которых пользователь дает оценку интерфейсу.
- Наблюдения за работой пользователей, с последующим обсуждением их способов использования системы при решении конкретных задач.
- Видеонаблюдения типичного использования системы.
- Добавление в систему программного кода, который собирал бы информацию о наиболее часто используемых системных сервисах и наиболее распространенных ошибках.

Основная литература -7, 4, 8.

Контрольные вопросы и упражнения

1. Перечислите и дайте краткое определение руководящих принципов разработки GUI- интерфейса.
2. В чем заключается сущность подхода «документ - представление» к построению графического интерфейса?
3. Каким образом окна и элементы управления окнами представляются на диаграммах навигации по окнам? Соответствует ли это представление назначению диаграмм видов деятельности в языке UML? Объясните свой ответ.
4. Опишите ситуации, в которых неразумно или невозможно поддерживать интерфейс пользователя.
5. Какие факторы следует учитывать при проектировании интерфейсов, использующих меню, для таких систем, как банкоматы? Опишите основные черты интерфейса банкомата, которым вы пользуетесь.
6. Обсудите преимущества графического способа отображения информации и приведите примеры приложений, в которых более уместно использовать графическое представление числовых данных, а не табличное.

Лекция 12. Этапы проектирования ИС с применением UML и Rational Rose. Тестирование объектно-ориентированных систем

Тестирование ПО – запуск исполняемого кода с тестовыми данными и исследование выходных данных и рабочих характеристик программного продукта для проверки правильности системы.

Тестирование – это динамический метод верификации и аттестации, так как применяется к исполняемой системе.

Верификацией и аттестацией - называют процессы проверки и анализа, в ходе которых проверяется соответствие ПО своей спецификации и требованиям заказчика. Верификация и аттестация охватывают полный жизненный цикл ПО - они начинаются на этапе анализа требований и

завершаются проверкой программного кода на этапе тестирования готовой программной системы.

Верификация отвечает на вопрос, *правильно ли создана* система.

Аттестация отвечает на вопрос, *правильно ли работает* система.

Согласно этим определениям, верификация проверят соответствие ПО системной спецификации, в частности функциональным и нефункциональным требованиям. Аттестация – более общий процесс, во время аттестации необходимо убедиться, что программный продукт соответствует ожиданиям заказчика. Аттестация проводится после верификации.

В верификации и аттестации используется две основные методики проверки и анализа систем.

Инспектирование ПО – анализ и проверка различных представлений (артефактов) системы на всех этапах процесса разработки. Инспектирование – это статический метод верификации и аттестации, поскольку им не требуется исполняемая система.

Тестирование ПО.

Во многих книгах, посвященных тестированию ПО, описывается процесс тестирования программных систем, реализующих функциональную модель ПО, но не рассматривается отдельно тестирование объектно-ориентированных систем. Несмотря на то, что большая часть методов тестирования подходит для любых видов, для тестирования объектно-ориентированных систем необходимы специальные методы, которые мы и рассмотрим далее.

Системы, разработанные по функциональной модели и объектно-ориентированные системы имеют существенные отличия:

-объекты представляют собой нечто большее, чем отдельные подпрограммы или функции;

-объекты, интегрированные в подсистемы, обычно слабо связаны между собой и поэтому сложно определить «самый верхний уровень» системы;

-при анализе повторно используемых объектов их исходный код может быть недоступным для тестировщиков.

Эти отличия означают, что при проверке объектов можно применять тестировании: методом белого ящика, основанное на анализе кода, а при тестировании сборки - следует использовать другие подходы.

Применительно к объектно-ориентированным системам можно определить следующие уровни тестирования.

- Тестирование отдельных методов(операций), ассоциированных с объектами. Обычно методы представляют собой функции или процедуры. Поэтому здесь можно использовать тестирование методами черного и белого ящиков.

Функциональное тестирование, или тестирование *методом черного ящика* базируется на том, что все тесты основываются на спецификации системы или ее компонентов. Поведение системы как черного ящика можно определить только посредством изучения ее входных и соответствующих

им выходных данных, то есть проверяется не реализация ПО, а только ее выполняемые функции. Метод структурного тестирования ,так называемый метод белого ящика, предполагает создание тестов на основе структуры системы и ее реализации. Такой метод, как правило, применяется к относительно небольшим программным элементам, например, к подпрограммам или методам, ассоциированным с объектами. При таком подходе тестировщик анализирует программный код и для получения тестовых данных использует знания о структуре компонента.

•Тестирование отдельных классов объектов.

Принцип тестирования методом черного ящика остается без изменений ,однако понятие «класса эквивалентности» необходимо расширить.

Подход к тестовому покрытию систем требует, чтобы все операторы в программе выполнялись хотя бы один раз, а также, чтобы выполнялись все ветви программы. При тестировании объектов полное тестовое покрытие включает:

- раздельное тестирование всех методов, ассоциированных с объектом;
- проверку всех атрибутов, ассоциированных с объектом;
- проверку всех возможных состояний объекта, для чего необходимо моделирование событий, приводящих к изменению состояния объекта.

Например, нужны тесты, которые проверяли бы задания атрибутов объекта после его инсталляции. Нужно также определить контрольные тесты для методов объекта и протестировать эти методы независимо. При тестировании состояний объекта, используется модель его состояний (диаграмма состояний UML), с помощью которой можно определить последовательность состояний, которые нужно протестировать.

Использование наледования усложняет разработку тестов для классов объектов. Если класс представляет методы, унаследованные от подклассов, то необходимо протестировать все подклассы со всеми унаследованными методами.

• Тестирование кластеров объектов.

Нисходящая и восходящая сборки оказываются не пригодными для создания групп связанных объектов. Поэтому здесь следует применять другие методы тестирования, например, основанные на сценариях. В объектно-ориентированных системах нет непосредственного эквивалента тестированию модулей. Однако считается, что группы классов, которые совместно предоставляют набор сервисов, следует тестировать вместе. Такой вид тестирования называется *тестирование кластеров*.

Создание кластеров основывается на выделении методов и сервисов, реализуемых посредством этих кластеров. При тестировании сборки объектно-ориентированных систем используется три подхода.

- *Тестирование сценариев и вариантов использования.*

Варианты использования (Use Case), или сценарии, описывают какой-либо один режим работы системы. Тестирование может базироваться на описании этих сценариев и кластеров объектов, реализующих данный Use Case.

- *Тестировании потоков*.

Этот подход основывается на проверке системных откликов на ввод данных или группу входных событий. Объектно-ориентированные системы, как правило, событийно-управляемые, поэтому для них особенно подходит данный вид тестирования. При использовании этого подхода необходимо знать, как в системе проходит обработка потоков основных и альтернативных событий.

- *Тестирование взаимодействий между объектами*.

Этот метод тестирования групп взаимодействующих объектов. Этот промежуточный уровень тестирования сборки системы основан на определении путей «метод-сообщение», отслеживающих последовательности взаимодействий между объектами. Диаграммы последовательности UML можно использовать для определения тестируемых операций и для разработки тестовых сценариев. Тестирование сценариев часто оказывается более эффективным, чем другие методы тестирования. Сам процесс тестирования можно спланировать так, чтобы в первую очередь проверялись наиболее вероятные сценарии и только затем исключительные сценарии. Сценарии определяются исходя из разработанных вариантов использования и, при необходимости, можно использовать диаграммы взаимодействия, которые отображают реализацию вариантов использования посредством системных объектов.

После выбора сценариев для тестирования системы важно убедиться, что все методы каждого класса будут выполняться хотя бы один раз. Конечно, все комбинации методов выполнить невозможно, но, по крайней мере, можно удостовериться, что все методы протестированы как часть какой-либо последовательности выполняемых методов.

8. Тестирование интерфейсов

Как правило, тестирование интерфейса выполняется в тех случаях, когда модули или подсистемы интегрируются в большие системы. Каждый модуль или подсистема имеет заданный интерфейс, который вызывается другими компонентами системы. Цель тестирования интерфейса – выявить ошибки, возникающие в системе вследствие ошибок в интерфейсах или неправильных предположений об интерфейсах.

Данный тип тестирования особенно важен в объектно-ориентированном проектировании, в частности, при повторном использовании объектов и классов объектов. Объекты в значительной степени определяются с помощью интерфейсов и могут повторно использоваться в различных комбинациях с разными объектами и в разных системах. Во время тестирования отдельных объектов невозможно выявить ошибки интерфейса, так как они являются скорее результатом взаимодействия между объектами, чем изолированного поведения одного объекта.

Между компонентами программы могут быть разные типы интерфейсов и соответственно разные типок ошибок интерфейсов. К таким

типам интерфейсов относятся: *параметрические интерфейсы, интерфейсы разделяемой памяти, процедурные интерфейсы и интерфейсы передачи сообщений.*

Ошибки в интерфейсах являются наиболее распространенными типами ошибок в сложных системах и делятся на три класса: неправильное использование интерфейсов, неправильное понимание интерфейсов и ошибки синхронизации.

Существует несколько общих правил тестирования интерфейсов:

- используйте экстремальные значения параметров, передаваемых внешним компонентам, что с высокой вероятностью обнаруживает несоответствия в интерфейсах;
- тестируйте интерфейс с нулевыми параметрами указателя;
- при вызове компонента через процедурный интерфейс, используйте тесты, вызывающие сбой в работе компонента;
- разрабатывайте тесты, генерирующие в несколько раз большее количество сообщений, чем будет в обычной работе в системах передачи сообщений;
- при взаимодействии нескольких компонентов через разделяемую память, разрабатывайте тесты, которые изменяют порядок активизации компонентов.

Унифицированный процесс разработки программных систем

В данной теме главное внимание сосредоточено на детальном обсуждении унифицированного процесса разработки ПО – RUP на базе которого возможно построение самых разнообразных схем конструирования программных приложений.

Rational Unified Process (RUP) - это основа технологии разработкиПО, разработанная и продаваемая компанией Rational Software. Он включает в себя передовой мировой опыт разработки ПО, и обеспечивает дисциплинарный подход к распределению и управлению задачами и областями ответственности в организации, занимающейся разработкой ПО. Применяя этот процесс, команды разработчиков программ смогут создавать высококачественное ПО, отвечающее потребностям своих конечных пользователей, и делать это в рамках предсказуемого графика и бюджета.

RUP направляет профессионалов в области программного обеспечения на эффективное применение лучших современных практических методов, таких как итеративная разработка, применение архитектурно – центрического подхода, вариантов использования, уменьшение риска на всех стадиях процесса и непрерывная проверка программы.

В основе Rational Unified Process лежат несколько фундаментальных принципов, собранные из множества успешных проектов:

- Начинайте вести наступление на главные риска раньше и ведите его непрерывно, или они сами пойдут в наступление на вас.
- Обеспечьте выполнение требований заказчиков.

- Сконцентрируйтесь на исполняемой программе.
- Приспосабливайтесь к изменениям с самого начала проекта.
- Рано закладывайте исполняемую архитектуру.
- Стройте систему из компонентов.
- Работайте вместе как одна команда.
- Сделайте качество образом жизни, а не запоздалой идеей.

RUP использует *итеративный подход* – при каждой итерации производится немного работы с требованиями, анализа, проектирования, реализации и тестирования. Каждая итерация основывается на результатах предыдущих итераций и производит исполняемую программу, которая на один шаг приближается к окончательному продукту.

Сам Rational Unified Process создавался с использованием методики, похожей на используемую в проектировании программ. В частности, моделирование производилось с помощью *Software Process Engineering Metamodel* (SPEM) – стандарта моделирования процессов, основанного на UML. У процесса есть две структуры: *динамическая структура*, в которой речь идет о жизненном цикле или временном измерении проекта, и *статическая структура,* которая имеет дело с тем, как элементы процесса – задачи, дисциплины, артефакты и роли логически группируются в основные дисциплины процесса.

RUP обеспечивает *структурированный подход к итеративной разработке*, разделяя проект на четыре фазы: Начало, Проектирование, Построение и Внедрение. Каждая фаза сопровождается так называемой вехой, - контрольной точкой процесса, в которой проверяется достижение целей очередной фазы, и принимается решение о переходе (или не переходе) в следующую фазу. Каждая из четырех фаз RUP, таким образом, имеет веху и четко определенный набор целей. Эти цели используются для определения того, какие задачи выполнять и какие артефакты создавать. Каждая фаза концентрируется строго на том, что единственно необходимо для достижения бизнес целей фазы.

Все элементы процесса – роли, задачи, артефакты и связанные с ними концепции, руководства и шаблоны сгруппированы в логические контейнеры, называемые *Дисциплинами* (Disciplines). В стандартном продукте RUP дисциплин всего девять. К ним относятся: бизнес – моделирование, управление требованиями, управление проектом, управление изменениями и среда.

Каждый рабочий поток процесса: сбор требований, анализ, проектирование, реализация и тестирование определяет набор связанных артефактов и действий. Напомним, что артефактом является документ, отчет, выполняемый элемент и т.п.. Артефакт может вырабатываться, обрабатываться или потребляться. Между артефактами потоков существуют зависимости. Например, модель Use Case, генерируемая в ходе сбора требований, уточняется моделью анализа из процесса проектирования,

реализуется моделью реализации из процесса реализации и проверяется тестовой моделью из процесса тестирования.

Модель - наиболее важная разновидность артефакта. Предусмотрены девять моделей, вместе они покрывают все решения по визуализации, спецификации, конструированию и документированию программных систем:

- бизнес – модель. Определяет абстракцию организации, для которой создается система;
- модель области определения. Фиксирует контекстное окружение системы;
- модель Use Case. Определяет функциональные требования к системе.
- модель анализа. Интерпретирует требования к системе в терминах проектной модели;
- проектная модель. Определяет словарь предметной области проблемы и ее решения;
- модель размещения. Определяет аппаратную топологию, в которой исполняется система;
- модель реализации. Определяет части, которые используются для сборки и реализации физической системы;
- тестовая модель. Определяет тестовые варианты для проверки системы;
- модель процессов. Определяет параллелизм в системе и механизмы синхронизации.

Технические артефакты подразделяются на четыре основных набора:

- набор требований. Описывает, *что* должна делать система;
- набор проектирования. Описывает, *как* должна быть сконструирована система;
- набор реализаций. Описывает сборку разработанных программных компонентов;
- набор размещения. Обеспечивает всю информацию о поставляемой конфигурации.

Набор требований может включать модель Use Case, модель нефункциональных требований, модель области определения, модель анализа, а также другие формы выражения нужд пользователя.

Набор проектирования может включать проектную модель, тестовую модель и другие формы выражения сущности системы.

Набор реализаций группирует все данные о программных элементах, образующих систему (программный код, файлы конфигурации, файлы данных, программные компоненты, информацию о сборке системы).

Набор размещения группирует всю информацию об упаковке, отправке, установки и запуске системы.

Каждый технологический процесс сопровождается *риском*. При разработке программного продукта неудовлетворительным результатом (НУ) может быть: превышение бюджета, низкая надежность, неправильное функционирование и т.д. Влияние риска вычисляют по выражению

Показатель Риска = Вероятность (НУ) *Потеря (НУ).

Управление риском включает шесть действий:

1. Идентификация риска – выявление элементов риска в проекте.
2. Анализ риска – оценка вероятности и величины потери по каждому элементу риска.
3. Ранжирование риска - упорядочение элементов риска по степени их влияния.
4. Планирование управления риском – подготовка к работе с каждым элементом риска.
5. Разрешение риска – устранение или разрешение элементов риска.
6. Наблюдение риска – отслеживание динамики элементов риска, выполнение корректирующих действий.

Первые три действия относятся к этапу оценивания риска, последние три действия – к этапу контроля риска.

Выделяют три категории источников риска: проектный риск, технический риск и коммерческий риск. После идентификации элементов риска следует количественно оценить их влияние на программный проект, решить вопросы о возможных потерях. Эти вопросы решаются на шаге анализа риска. И, наконец, в общий план программного проекта интегрируется план управления каждым элементом риска, то есть набор функций управления каждым элементом риска.

Основная литература –7, 12.

Контрольные вопросы.

1. Перечислите четыре составляющие производства ПО.
2. Назовите существующие модели процессов разработки ПО.
3. Что такое управление проектом? Составляющие управления проектом.
4. Какую структуру имеет унифицированный процесс разработки?
5. Какие рабочие потоки имеются в унифицированном процессе, и поясните назначение этих потоков?
6. Какие технологические артефакты определены в унифицированном процессе, поясните назначение этих артефактов?
7. В чем суть анализа риска и планирования управления риском?
8. Обсудите различия между тестированием методов черного и структурным тестированием. Подумайте, каким образом можно совместно использовать эти методы в процессе тестирования дефектов.
9. Разработайте набор тестовых данных для следующих компонентов:

- программа сортировки массивов целых чисел;
- программа, которая вычисляет количество символов (отличных от пробелов) в текстовых строках;

10. Создайте сценарии тестирования состояний банкомата.
11. В чем состоит суть методики тестирования интеграции объекто-ориентированных систем, основанной на использовании ?
12. Поясните содержание тестирования, основанного на сценариях.
13. Перечислите методы тестирования взаимодействия классов.

14. Напишите код для класса *Счет* с атрибутом – баланс, методами доступа и методом добавить(). Исходите из того, что класс *Счет* имеет состояния *Платежеспособный, Пустой* и *Задолженность*, и они реализованы с использованием образца проектирования State. Напишите полный набор модульных тестов для класса *Счет*, в том числе и тесты на основе состояний

Список литературы

Основная:

1 Орлов С.А., Технология разработки программного обеспечения. Учебник. - СПб: Питер, 2002,2012.

2 С.В. Маклаков BPWin, и ERWin. CASE-разработки информационных систем. - М.:ДИАЛОГ-МИФИ, 2000 - 256 с.

3 Грейди Буч, Джеймс Рамбо, Айвар Джекобсон, Язык UML. Руководство пользователя: Пер. с англ - М.: ДМК Пресс, 2001.

4 Марка Д.А., Мак Гоуэн К. Методология структурного анализа и проектирования. М., "МетаТехнология", 1993.

5 Калянов Г.Н. CASE. Структурный системный анализ (автоматизация и применение). М., "Лори", 1996.

6 Шлеер С., Меллор С. Объектно-ориентированный анализ: моделирование мира в состояниях. Киев, "Диалектика", 1993.

7 Гагарина, Л.Г. Технология разработки программного обеспечения.- М.: ФОРУМ-ИНФРА-М, 2009,2011,2012.

8 Панюкова Т.А. Проектирование программных средств.-М.: «ЛИБРОКОМ»,2012

9 Крылов Е.В. Техника разработки программ. Кн.2. Технология, надежность и качество программного обеспечения. - М.,2008

10 Скопин И.Н. Основы менеджмента программных проектов. -М.: «Бином», 2004,2009,2012

11 Кулямин В.В. Технологии программирования. Компонентный подход.-М.,2007,2014

Дополнительная:

12 Шилдт Г. JAVA Полное руководство. -М.: «Вильямс», 2012,2013

13 Лафоре Р. Структуры данных и алгоритмы JAVA. - СПб.: «Питер», 2011

14 Гергель В.П. Теория и практика параллельных вычислений. - М.: «Интернет-УИТ: Бином», 2007,2013

15 Черников Б.В. Управление качеством программного обеспечения. - М.: «Форум», «Инфра-М», 2012