Министерство образования и науки РФ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
АМУРСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
(ФГБОУ ВО «АмГУ»)

ТЕХНОЛОГИЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

сборник учебно-методических материалов
для направления подготовки 09.04.04 Программная инженерия

Благовещенск

2017

Печатается по решению
редакционно-издательского совета
факультета математики и информатики
Амурского государственного
университета

Составитель: Галаган Т.А.

Технология разработки программного обеспечения: сборник учебно-методических материалов для направления подготовки 09.04.04 – Благовещенск: Амурский гос. ун-т, 2017. – 51 с.

Модели жизненного цикла программного обеспечения

Жизненный цикл программного обеспечения (ПО) – период времени, который начинается с момента принятия решения о необходимости создания программного продукта и заканчивается в момент его полного изъятия из эксплуатации.

Жизненный цикл проекта включает все фазы от момента инициации до момента завершения. Переходы от одного этапа к другому редко четко определены, за исключением тех случаев, когда они формально разделяются принятием предложения или получением разрешения на продолжение работы..

Модель жизненного цикла ПО – структура, определяющая последовательность выполнения и взаимосвязи процессов, действий и задач на протяжении жизненного цикла. Модель жизненного цикла зависит от специфики, масштаба и сложности проекта и специфики условий, в которых система создается и функционирует.

Стандарт ГОСТ Р ИСО/МЭК 12207-2010 не предлагает конкретную модель жизненного цикла. Его положения являются общими для любых моделей жизненного цикла, методов и технологий создания ИС. Он описывает структуру процессов жизненного цикла, не конкретизируя, как реализовать или выполнить действия и задачи, включенные в эти процессы.

Модель ЖЦ ПО включает в себя:

1. Стадии;
2. Результаты выполнения работ на каждой стадии;
3. Ключевые события — точки завершения работ и принятия решений.

Стадия – часть процесса создания ПО, ограниченная определенными временными рамками и заканчивающаяся выпуском конкретного продукта (моделей, программных компонентов, документации), определяемого заданными для данной стадии требованиями.

На каждой стадии могут выполняться несколько процессов, определенных в стандарте ГОСТ Р ИСО/МЭК 12207-2010, и наоборот, один и тот же процесс может выполняться на различных стадиях. Соотношение между процессами и стадиями также определяется используемой моделью жизненного цикла ПО.

Водопадная (каскадная, последовательная) модель

Водопадная модель жизненного цикла (*waterfall model*) была предложена в 1970 г. У.Ройсом. Она предусматривает последовательное выполнение всех этапов проекта в строго фиксированном порядке. Переход на следующий этап означает полное завершение работ на предыдущем этапе. Требования, определенные на стадии формирования требований, строго документируются в виде технического задания и фиксируются на все время разработки проекта. Каждая стадия завершается выпуском полного комплекта документации, достаточной для того, чтобы разработка могла быть продолжена другой командой разработчиков.

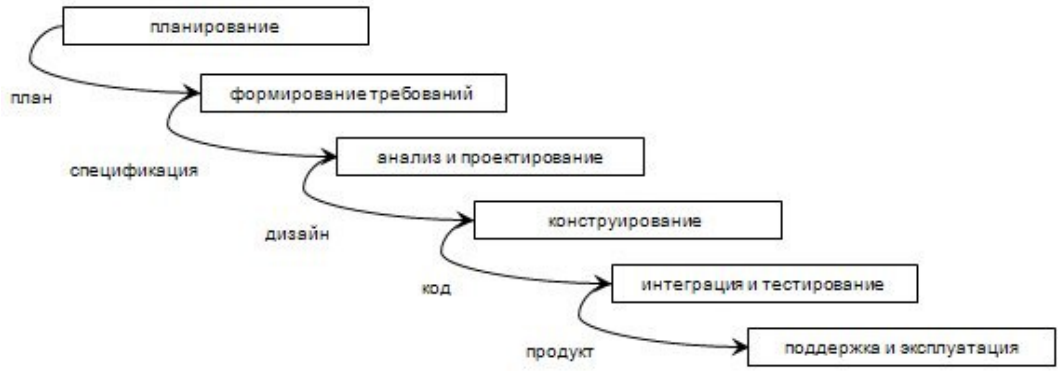


Рисунок 1. Каскадная модель жизненного цикла.

Этапы проекта в соответствии с каскадной моделью:

1. Формирование требований;
2. Проектирование;
3. Реализация;
4. Тестирование;
5. Внедрение;
6. Эксплуатация и сопровождение.

На рисунке изображены типичные фазы каскадной модели жизненного цикла и соответствующие активы проекта, являющиеся для одних фаз выходами, а для других - входами. Марри Кантор отмечает ряд важных аспектов, характерных для водопадной модели: “Водопадная схема включает несколько важных операций, применимых ко всем проектам:

- составление плана действий по разработке системы;
- планирование работ, связанных с каждым действием;
- применение операции отслеживания хода выполнения действий с контрольными этапами.

Преимущества:

- Полная и согласованная документация на каждом этапе;
- Легко определить сроки и затраты на проект.

Недостатки:

В водопадной модели переход от одной фазы проекта к другой предполагает полную корректность результата (выхода) предыдущей фазы. Однако неточность какого-либо требования или некорректная его интерпретация в результате приводит к тому, что приходится «откатываться» к ранней фазе проекта и требуемая переработка не просто выбивает проектную команду из графика, но приводит часто к качественному росту затрат и, не исключено, к прекращению проекта в той форме, в которой он изначально задумывался. По мнению современных специалистов, основное заблуждение авторов водопадной модели состоит в предположениях, что проект проходит через весь процесс один раз, спроектированная архитектура хороша и проста в использовании, проект осуществления разумен, а ошибки в реализации легко устраняются по мере тестирования. Эта модель исходит из того, что все ошибки будут сосредоточены в реализации, а потому их устранение происходит равномерно во время тестирования компонентов и системы. Таким образом, водопадная модель для крупных проектов мало реалистична и может быть эффективно использована только для создания небольших систем.

Итеративная и инкрементальная модель – эволюционный подход

Альтернативой последовательной модели является так называемая модель итеративной и инкрементальной разработки (англ. *iterative and incremental development, IID*), получившей также от Т. Гилба в 70-е гг. название *эволюционной модели*. Также эту модель называют *итеративной моделью* и *инкрементальной моделью*.

Модель IID предполагает разбиение жизненного цикла проекта на последовательность итераций, каждая из которых напоминает «мини-проект», включая все процессы разработки в применении к созданию меньших фрагментов функциональности, по сравнению с проектом в целом. Цель каждой *итерации* – получение работающей версии программной системы, включающей функциональность, определённую интегрированным содержанием всех предыдущих и текущей итерации. Результат финальной итерации содержит всю требуемую функциональность продукта. Таким образом, с завершением каждой итерации продукт получает приращение – *инкремент* – к его возможностям, которые, следовательно, развиваются *эволюционно*. Итеративность, инкрементальность и эволюционность в данном случае есть выражение одного и того же смысла разными словами со слегка разных точек зрения.

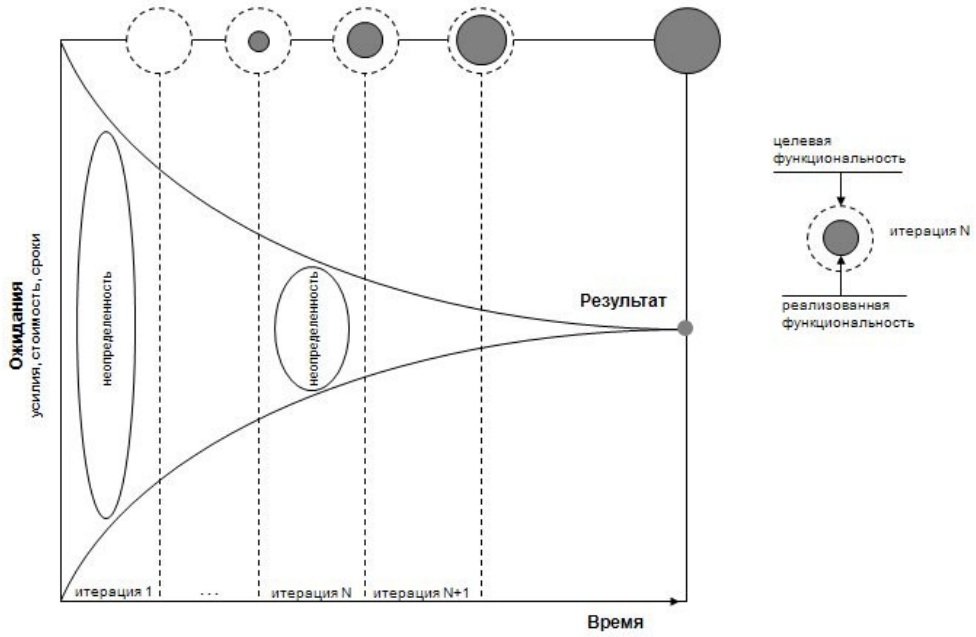


Рисунок 2. Снижение неопределенности и инкрементальное расширение функциональности при итеративной организация жизненного цикла.

По выражению Т. Гилба, «эволюция – прием, предназначенный для создания видимости стабильности. Шансы успешного создания сложной системы будут максимальными, если она реализуется в серии небольших шагов и если каждый шаг заключает в себе четко определённый успех, а также возможность «отката» к предыдущему успешному этапу в случае неудачи. Перед тем, как пустить в дело все ресурсы, предназначенные для создания системы, разработчик имеет возможность получать из реального мира сигналы обратной связи и исправлять возможные ошибки в проекте».

Подход IID имеет и свои отрицательные стороны, которые, по сути, - обратная сторона достоинств. Во-первых, целостное понимание возможностей и ограничений проекта очень долгое время отсутствует. Во-вторых, при итерациях приходится отбрасывать часть сделанной ранее работы. В-третьих, добросовестность специалистов при выполнении работ всё же снижается, что психологически объяснимо, ведь над ними постоянно довлеет ощущение, что «всё равно всё можно будет переделать и улучшить позже».

Различные варианты итерационного подхода реализованы в большинстве современных методологий разработки (RUP, MSF, XP).

Наиболее известным и распространенным вариантом эволюционной модели является *спиральная модель*, ставшая уже по-сути самостоятельной моделью, имеющей различные сценарии развития и детализации.

Спиральная модель

Спиральная модель (англ. *spiral model*) была разработана в середине 1980-х годов Барри Боэмом. Она основана на классическом цикле Деминга PDCA (plan-do-check-act). При использовании этой модели ПО создается в несколько итераций (витков спирали) методом прототипирования.

Каждая итерация соответствует созданию фрагмента или версии ПО, на ней уточняются цели и характеристики проекта, оценивается качество полученных результатов и планируются работы следующей итерации.

На каждой итерации оцениваются:

- риск превышения сроков и стоимости проекта;
- необходимость выполнения ещё одной итерации;
- степень полноты и точности понимания требований к системе;
- целесообразность прекращения проекта.

Важно понимать, что спиральная модель является не альтернативой эволюционной модели (модели IID), а специально проработанным вариантом. К сожалению, нередко спиральную модель либо ошибочно используют как синоним эволюционной модели вообще, либо (не менее ошибочно) упоминают как совершенно самостоятельную модель наряду с IID.

Отличительной особенностью спиральной модели является специальное внимание, уделяемое рискам, влияющим на организацию жизненного цикла, и контрольным точкам. Боэм формулирует 10 наиболее распространённых (по приоритетам) рисков:

1. Дефицит специалистов.
2. Нереалистичные сроки и бюджет.
3. Реализация несоответствующей функциональности.
4. Разработка неправильного пользовательского интерфейса.
5. Перфекционизм, ненужная оптимизация и оттачивание деталей.
6. Непрекращающийся поток изменений.
7. Нехватка информации о внешних компонентах, определяющих окружение системы или вовлеченных в интеграцию.
8. Недостатки в работах, выполняемых внешними (по отношению к проекту) ресурсами.
9. Недостаточная производительность получаемой системы.
10. Разрыв в квалификации специалистов разных областей.

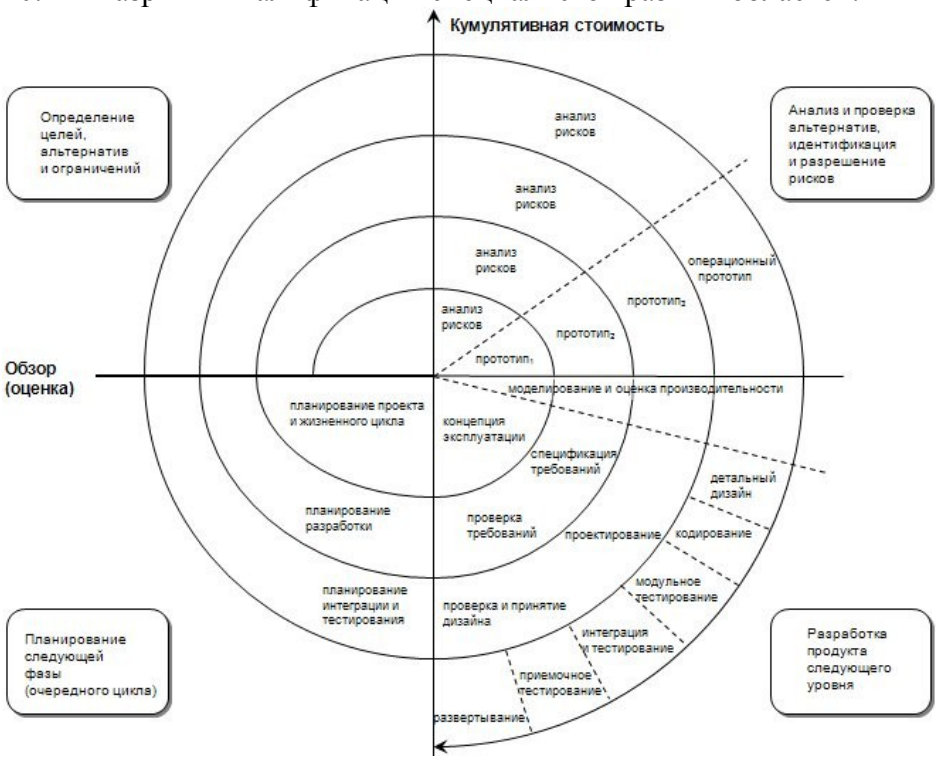


Рисунок 3. Оригинальная спиральная модель жизненного цикла разработки по Боэму

Модель уделяет специальное внимание раннему анализу возможностей повторного использования. Это обеспечивается, в первую очередь, в процессе идентификации и оценки альтернатив.

Модель предполагает возможность эволюции жизненного цикла, развитие и изменение программного продукта. Главные источники изменений заключены в целях, для достижения которых создается продукт. Подход, предусматривающий скрытие информации о деталях на определенном уровне дизайна, позволяет рассматривать различные архитектурные альтернативы так, как если бы мы говорили о единственном проектном решении,

что уменьшает риск невозможности согласования функционала продукта и изменяющихся целей (требований).

Модель предоставляет механизмы достижения необходимых параметров качества как составную часть процесса разработки программного продукта. Эти механизмы строятся на основе идентификации всех типов целей (требований) и ограничений на всех "циклах" спирали разработки. Например, ограничения по безопасности могут рассматриваться как риски на этапе специфицирования требований

Модель уделяет специальное внимание предотвращению ошибок и отбрасыванию ненужных, необоснованных или неудовлетворительных альтернатив на ранних этапах проекта. Это достигается явно определенными работами по анализу рисков, проверке различных характеристик создаваемого продукта (включая архитектуру, соответствие требованиям и т.п.) и подтверждение возможности двигаться дальше на каждом "цикле" процесса разработки.

Модель позволяет контролировать источники проектных работ и соответствующих затрат. По-сути речь идет об ответе на вопрос – как много усилий необходимо затратить на анализ требований, планирование, конфигурационное управление, обеспечение качества, тестирование, формальную верификацию и т.д. Модель, ориентированная на риски, позволяет в контексте конкретного проекта решить задачу приложения адекватного уровня усилий, определяемого уровнем рисков, связанных с недостаточным выполнением тех или иных работ.

Модель не проводит различий между разработкой нового продукта и расширением (или сопровождением) существующего. Этот аспект позволяет избежать часто встречающегося отношения к поддержке и сопровождению как ко "второсортной" деятельности. Такой подход предупреждает большого количество проблем, возникающих в результате одинакового уделения внимания как обычному сопровождению, так и критичным вопросам, связанным с расширением функциональности продукта, всегда ассоциированным с повышенными рисками.

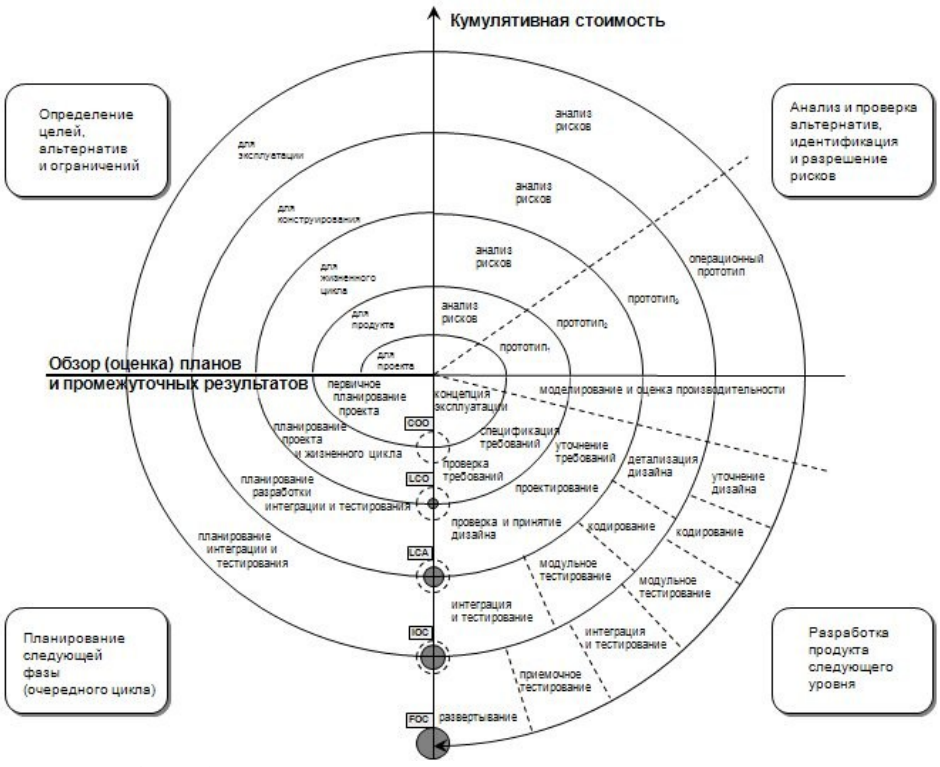


Рисунок 4. Обновленная спиральная модель с контрольными точками проекта.

Модель позволяет решать интегрированный задачи системной разработки, охватывающей и программную и аппаратную составляющие создаваемого продукта. Подход,

основанный на управлении рисками и возможности своевременного отбрасывания непривлекательных альтернатив (на ранних стадиях проекта) сокращает расходы и одинаково применим и к аппаратной части, и к программному обеспечению."

Общий набор контрольных точек в сегодняшней спиральной модели:

- *Concept of Operations (COO)* – концепция <использования> системы;
- *Life Cycle Objectives (LCO)* – цели и содержание жизненного цикла;
- *Life Cycle Architecture (LCA)* – архитектура жизненного цикла; здесь же возможно говорить о готовности концептуальной архитектуры целевой программной системы;
- *Initial Operational Capability (IOC)* – первая версия создаваемого продукта, пригодная для опытной эксплуатации;
- *FinalOperationalCapability (FOC)* – готовый продукт, развернутый (установленный и настроенный) для реальной эксплуатации.

Спецификация и планирование и разработка программного обеспечения

Процесс разработки ПО начинается с создания концептуального описания будущего продукта - образа «по-крупному» ("vision statement") в контексте требований рынка. На этом этапе «менеджер по продукту» - специалист-маркетолог доносит до менеджеров по разработке программного обеспечения поставленные цели и требования для пользователей, выдвигаемые функциональные возможности и их приоритеты.

Менеджеры по разработке ПО составляют функциональную спецификацию, в которой особенности ПО прописываются также с точки зрения пользователя, но с большей подробностью, глубиной и формализованностью; затрагиваются основные компоненты архитектуры – основные компоненты и их взаимосвязи.

В дальнейшем на изменения функциональности будут влиять как внутренние, так и внешние факторы – недостаток ресурсов, отставание от графика, изъяны в реализации, потенциальные рыночные продукты, конкурирующие с данным ПО. Исследования Microsoft покато в среднем показывают, что 25% функциональных особенностей исчезают ко времени выпуска.

На основе функциональной спецификации менеджеры по разработке, консультируясь с проектировщиками, начинают создавать горизонтальную архитектуру продукта, разбивая его на 3-4 группы. По количеству групп формируются подпроекты, работа над которыми будет вестись последовательно – по мере значимости их функций. Каждый подпроект завершается выпуском промежуточной «контрольной» версии продукта.

Архитектура проекта отображается на организационную структуру: для реализации отдельных функций в рамках одного подпроекта назначаются небольшие команды, работающие параллельно и максимально автономно.

С точки зрения современных представлений хорошо структурированная архитектура продукта – необходимое условие его успешной разработки.

Каждая из команд подпроекта состоит из менеджера по разработке (program manager), 3-8 разработчиков и стольких же тестировщиков. Каждая команда выполняет полный цикл разработки своей функции ПО – проектирование, кодирование, прототипирование, тестирование. Проектной документации не ведется. Функции документации выполняет сам код.

Для синхронизации работы отдельных разработчиков:

разрабатываемый продукт всегда существует в виде доступной всем командам централизованной базы данных, содержащей эталонную версию файлов с исходным кодом, которая периодически проходит процедуру генерации новой версии, что позволяет, не дожидаясь конца разработки, увидеть как отдельные функции работают в контексте всего продукта. (в компании Microsoft сборка выполняется ежедневно. При этом каждый может скачать необходимые данные, выполнит изменения и провести персональную

сборку, используются общие языки программирования (в основном Си и С++), общий стиль кодирования и стандартное средство разработки);

каждый разработчик работает в паре со «своим» тестировщиком, который, как правило, использует более современные средства, включая автоматически генерируемые и запускаемые тесты; (не каждая фирма может позволит тестировщика для каждого разработчика)

по крайней мере, два раза в неделю разработчик должен встраивать «персональный код» в общую базу (можно и чаще). При проявлении при этом какого-либо дефекта, тут же его зафиксировать;

специально назначенный разработчик (project build master) ежедневно выполняет в назначенное время полную сборку всего продукта на основе текущей версии;

менеджер отслеживает прогресс ежедневных сборок на основе метрик, показывающих, сколько новых «багов» выявлено и сколько активных исправлено.

В конце каждого подпроекта предусмотрен специальный период «буферное время», в течение которого разрешаются не запланированные проблемы. Для проектов из разряда приложений буферное время составляет 20-30% от всей продолжительности предпроекта, а для системных – 50%. Затем выпускается очередная «контрольная» версия, с которой активно работают пользователи.

Важнейшим механизмом, обеспечивающим обратную связь, является институт лабораторий пользователя. Здесь в домашней обстановке (кухняя, столовая, детская) работают сотрудники, имеющие не только компьютерное образование, но и знание психологии и эргономики. Их стремятся подобрать так, чтобы они представляли все категории потенциальных пользователей.

Разработчикам, особенно отвечающим за интерфейс, вменено присутствовать на тестовых экспериментах.

Две основные метрики, используемые на этом этапе: 1)процент пользователей, которым удалось выполнить некоторые осмысленные действия без руководства пользователя и 2)процент корректных шагов в первой попытке для решения поставленной задачи.

Эксперименты проводятся и на выезде: в школах, университетах, по месту жительства пользователей. В последней стадии «бэта» версии отправляются для опытной эксплуатации к партнерам корпорации. Затем выпускается «финальная» версия с проработанной документацией.

Традиционно через год выпускается исправленная и дополненная версия, а через два года – радикально переработанная.

Анализ требований и определение спецификаций программного обеспечения

Определение требований к программным продуктам

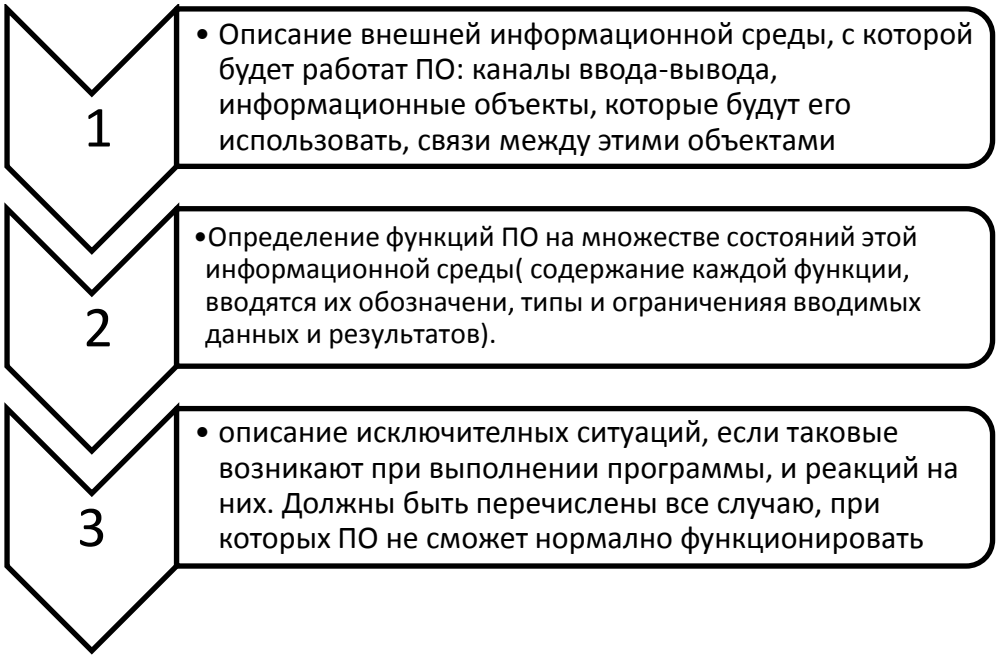
На этапе постановки задачи принимаются решения о его функциях, эксплуатационных ограничениях, выбора архитектуры, среды разработки и пр.

Функциональные требования описывают сервисы системы, ее поведение в определенных ситуациях, реакцию на входные действия и данные.

При написании функциональных требований формулируются функции программы, а не на какую кнопочку следует нажать, чтобы получит результат.

Функциональные требования документируются в спецификации требований к программному обеспечению. Они должны быть математически точны. Желательно, чтобы при их разработке применялись математические методы и формализованные языки.

Состав функциональной спецификации представлен на рисунке.



Эксплуатационные требования

Правильность – функционирование в соответствии с техническим заданием с вероятностью наличия ошибок.

Универсальность – обеспечение правильной работы при любых допустимых данных и защита от неправильных данных (речь идет о степени универсальности)

Надежность (помехозащищенность) – обеспечение полной повторяемости результатов, т.е. обеспечение их правильности при наличии различных сбоев (создание контрольных точек, ввод избыточной информации, дублирование)

Проверяемость – возможность проверки результата (необходимо документально фиксировать исходные данные, установленные режимы и др., особенно если сигналы поступают от датчиков)

Точность результата – обеспечение погрешности результатов не выше заданной. Жесткие требования предъявляются в системах навигации или управления технологическими процессами.

Защищенность – обеспечение конфиденциальности информации.

Программная совместимость – возможность функционирования с другим программным обеспечением, прежде всего с операционной системой.

Аппаратная совместимость – минимальная конфигурация оборудования.

Эффективность – использование минимально возможного количества ресурсов технических средств.

Адаптируемость – возможность быстрой модификации с целью приспособления к изменениям условий функционирования.

Повторная видимость – возможность повторного выполнения без перезагрузки. Это требование предъявляется к резидентному ПО, например к драйверам.

Реентерабельность – возможность параллельного использования несколькими процессами. Для выполнения необходимо создавать копию изменяемых данных для каждого процесса.

Выбор архитектуры

Архитектурой программного обеспечения называют совокупность базовых принципов его построения. Это структура, которая включает элементы программы, видимые

извне свойства этих элементов и связи между ними.

С точки зрения пользователей различают однопользовательские и многопользовательские.

В рамках однопользовательских выделяют:

Программа – упорядоченная последовательность формализованных инструкций для решения задачи с помощью компьютера.

Пакет программ – несколько отдельных программ, каждая из которых вводит необходимые данные и выводит результаты.

Программный комплекс – совокупность программ, совместно обеспечивающая решение небольшого класса сложных задач. Для выполнения задачи программой-диспетчером вызывается последовательно нескольких программ комплекса. Данные и результаты хранятся в пределах одного пользовательского проекта. Программа-диспетчер должна иметь интерфейс и простую справочную систему.

Программная система – организованная совокупность программ, позволяющая решить широкий класс задач из некоторой прикладной области. Программы взаимодействуют через общие данные. Интерфейс взаимодействия достаточно развит.

Многопользовательскую архитектуру реализуют системы, построенные по принципу «клиент – сервер».

Этапы разработки программного обеспечения

Процесс разработки (development process) в соответствии со стандартом предусматривает действия и задачи, выполняемые разработчиком. Он охватывает работы по созданию программного обеспечения и его компонентов в соответствии с заданными требованиями, включая оформление проектной и эксплуатационной документации, подготовку материалов, необходимых для проверки работоспособности и соответствия качества программных продуктов, материалов, необходимых для обучения персонала и т. д.

По стандарту процесс разработки включает следующие действия:

- подготовительную работу – выбор модели жизненного цикла, стандартов, методов и средств разработки, а также составление плана работ;
- анализ требований к системе – определение ее функциональных возможностей, пользовательских требований, требований к надежности и безопасности, требований к внешним интерфейсам и т. д.;
- проектирование архитектуры системы – определение состава необходимого оборудования, программного обеспечения и операций, выполняемых обслуживающим персоналом;
- анализ требований к программному обеспечению – определение функциональных возможностей, включая характеристики производительности, среды функционирования компонентов, внешних интерфейсов, спецификаций надежности и безопасности, эргономических требований, требований к используемым данным, установке, приемке, пользовательской документации, эксплуатации и сопровождению;
- проектирование архитектуры программного обеспечения – определение структуры программного обеспечения, документирование интерфейсов его компонентов, разработку предварительной версии пользовательской документации, а также требований к тестам и плана интеграции;
- детальное проектирование программного обеспечения – подробное описание компонентов программного обеспечения и интерфейсов между ними, обновление пользовательской документации, разработка и документирование требований к тестам и плана тестирования компонентов программного обеспечения, обновление плана интеграции компонентов;
- кодирование и тестирование программного обеспечения – разработку и документирование каждого компонента, а также совокупности тестовых процедур и данных для их тестирования, тестирование компонентов, обновление пользовательской докумен-

тации, обновление плана интеграции программного обеспечения;

- интеграцию программного обеспечения – сборку программных компонентов в соответствии с планом интеграции и тестирование программного обеспечения на соответствие квалификационным требованиям, представляющих собой набор критериев или условий, которые необходимо выполнить, чтобы квалифицировать программный продукт, как соответствующий своим спецификациям и готовый к использованию в заданных условиях эксплуатации;
- квалификационное тестирование программного обеспечения – тестирование программного обеспечения в присутствии заказчика для демонстрации его соответствия u1090 требованиям и готовности к эксплуатации; при этом проверяется также готовность и полнота технической и пользовательской документации;
- интеграцию системы - сборку всех компонентов системы, включая программное обеспечение и оборудование;
- квалификационное тестирование системы – тестирование системы на соответствие требованиям к ней и проверка оформления и полноты документации;
- установку программного обеспечения – установку программного обеспечения на оборудовании заказчика и проверку его работоспособности;
- приемку программного обеспечения – оценку результатов квалификационного тестирования программного обеспечения и системы в целом и документирование результатов оценки совместно с заказчиком, окончательную передачу программного обеспечения заказчику.

Указанные действия можно сгруппировать, условно выделив следующие основные этапы разработки программного обеспечения:

- постановка задачи (стадия «Техническое задание»);
- анализ требований и разработка спецификаций (стадия «Эскизный проект»);
- проектирование (стадия «Технический проект»);
- реализация (стадия «Рабочий проект»).

Традиционно разработка также включала этап сопровождения. Однако по международному стандарту в соответствии с изменениями, произошедшими в индустрии разработки программного обеспечения, этот процесс теперь рассматривается отдельно. Условность выделения этапов связана с тем, что на любом этапе возможно принятие решений, которые потребуют пересмотра решений, принятых ранее.

Постановка задачи

В процессе постановки задачи четко формулируют назначение программного обеспечения и определяют основные требования к нему. Каждое требование представляет собой описание необходимого или желаемого свойства программного обеспечения. Различают функциональные требования, определяющие функции, которые должно выполнять разрабатываемое программное обеспечение, и эксплуатационные требования, определяющие особенности его функционирования. Требования к программному обеспечению, имеющему прототипы, обычно определяют по аналогии, учитывая структуру и характеристики уже существующего программного обеспечения. Для формулирования требований к программному обеспечению, не имеющему аналогов, иногда необходимо провести специальные исследования, называемые предпроектными. В процессе таких исследований определяют разрешимость задачи, возможно, разрабатывают методы ее решения (если они новые) и устанавливают наиболее существенные характеристики разрабатываемого программного обеспечения. Для выполнения предпроектных исследований, как правило, заключают договор на выполнение научно-исследовательских работ. В любом случае этап постановки задачи заканчивается разработкой технического задания, фиксирующего принципиальные требования, и принятием основных проектных решений.

Анализ требований и определение спецификаций

Спецификациями называют точное формализованное описание функций и ограничений разрабатываемого программного обеспечения. Соответственно различают функци-

ональные и эксплуатационные спецификации. Совокупность спецификаций представляет собой общую логическую модель проектируемого программного обеспечения. Для получения спецификаций выполняют анализ требований технического задания, формулируют содержательную постановку задачи, выбирают математический аппарат формализации, строят модель предметной области, определяют подзадачи и выбирают или разрабатывают методы их решения. Часть спецификаций может быть определена в процессе предпроектных исследований и, соответственно, зафиксирована в техническом задании. На этом этапе также целесообразно сформировать тесты для поиска ошибок в проектируемом программном обеспечении, обязательно указав ожидаемые результаты.

Проектирование

Основной задачей этого этапа является определение подробных спецификаций разрабатываемого программного обеспечения. Процесс проектирования сложного программного обеспечения обычно включает:

- проектирование общей структуры – определение основных компонентов и их взаимосвязей;
- декомпозицию компонентов и построение структурных иерархий в соответствии с рекомендациями блочно-иерархического подхода;
- проектирование компонентов.

Результатом проектирования является детальная модель разрабатываемого программного обеспечения вместе со спецификациями его компонентов всех уровней. Тип модели зависит от выбранного подхода (структурный, объектный или компонентный) и конкретной технологии проектирования. Однако в любом случае процесс проектирования охватывает как проектирование программ (подпрограмм) и определение взаимосвязей между ними, так и проектирование данных, с которыми взаимодействуют эти программы или подпрограммы.

Принято различать также два аспекта проектирования:

- логическое проектирование, которое включает те проектные операции, которые непосредственно не зависят от имеющихся технических и программных средств, составляющих среду функционирования будущего программного продукта;
- физическое проектирование - привязка к конкретным техническим и программным средствам среды функционирования, т. е. учет ограничений, определенных в спецификациях.

Реализация

Реализация представляет собой процесс поэтапного написания кодов программы на выбранном языке программирования (кодирование), их тестирование и отладку.

Сопровождение

Сопровождение – это процесс создания и внедрения новых версий программного продукта. Причинами выпуска новых версий могут служить:

- необходимость исправления ошибок, выявленных в процессе эксплуатации предыдущих версий;
- необходимость совершенствования предыдущих версий, например, улучшения интерфейса, расширения состава выполняемых функций или повышения его производительности;
- изменение среды функционирования, например, появление новых технических средств и/или программных продуктов, с которыми взаимодействует сопровождаемое программное обеспечение.

На этом этапе в программный продукт вносят необходимые изменения, которые так же, как в остальных случаях, могут потребовать пересмотра проектных решений, принятых на любом предыдущем этапе. С изменением модели жизненного цикла программного обеспечения роль этого этапа существенно возросла, так как продукты теперь создаются итерационно: сначала выпускается сравнительно простая версия, затем следующая с большими возможностями, затем следующая и т. д. Именно это и послужило причиной

выделения этапа сопровождения в отдельный процесс жизненного цикла в соответствии с стандартом 1SO/IEC 12207.

Рассматриваемый стандарт только называет и определяет процессы жизненного цикла программного обеспечения, не конкретизируя в деталях, как реализовывать или выполнять действия и задачи, включенные в эти процессы. Эти вопросы регламентируются соответствующими методами, методиками и т. п.

Документация по сопровождению программных средств

Программные средства являются одним из наиболее гибких видов промышленных изделий и эпизодически подвергаются изменениям в течение всего времени их использования.

Для сохранения и повышения качества программного обеспечения необходимо регламентировать процесс модификации и поддерживать его соответствующим тестированием и контролем качества. В результате программное изделие со временем обычно улучшается как по функциональным возможностям, так и по качеству решения отдельных задач.

Документирование программных изделий:

При разработке программного обеспечения (ПО) создается и используется большой объем разнообразной документации.

Она необходима:

1) как средство передачи информации между разработчиками ПО,
2) как средство управления разработкой ПО,
3) как средство передачи пользователям информации, необходимой для применения и сопровождения ПО.

На создание этой документации приходится большая доля стоимости ПО.

Эту документацию можно разбить на две группы:

1. Документы управления разработкой ПО (software process documentation) управляют и протоколируют процессы разработки и сопровождения ПО, обеспечивая связи внутри коллектива разработчиков ПО и между коллективом разработчиков и менеджерами ПО (software managers) – лицами, управляющими разработкой ПО. Эти документы могут быть следующих типов:

• планы, оценки, расписания. Эти документы создаются менеджерами для прогнозирования и управления процессами разработки и сопровождения ПО.

• отчеты об использовании ресурсов в процессе разработки. Создаются менеджерами.

• стандарты. Эти документы предписывают разработчикам, каким принципам, правилам, соглашениям они должны следовать в процессе разработки ПО. Эти стандарты могут быть как международными или национальными, так и специально созданными для организации, в которой ведется разработка ПО.

• рабочие документы. Это основные технические документы, обеспечивающие связь между разработчиками. Они содержат фиксацию идей и проблем, возникающих в процессе разработки, описание используемых стратегий и подходов, а также рабочие (временные) версии документов, которые должны войти в ПО.

• заметки и переписка. Эти документы фиксируют различные детали взаимодействия между менеджерами и разработчиками.

- документы, входящие в состав ПО.

2. *Документы, входящие в состав* ПО *(*software product documentation*),* описывают программы ПО как с точки зрения их применения пользователями, так и с точки зрения их разработчиков и сопроводителей (в соответствии с назначением ПО). Следует отметить, что эти документы будут использоваться не только на стадии эксплуатации ПО (в ее фазах применения и сопровождения), но и на стадии разработки для управления процессом разработки (вместе с рабочими документами). Во всяком случае, они должны быть проверены (протестированы) на соответствие программам ПО.

Существует четыре основных типа документации на ПО:

1. архитектурная/проектная – обзор программного обеспечения, включающий описание рабочей среды и принципов, которые должны быть использованы при создании ПО;

2. техническая – документация на код, алгоритмы, интерфейсы, API;

3. пользовательская – руководства для конечных пользователей, администраторов системы и другого персонала;

4. маркетинговая.

Архитектурная/проектная документация*:*

Проектная документация обычно описывает продукт в общих чертах. Не описывая того, как что-либо будет использоваться, она скорее отвечает на вопрос «почему именно так?».

Техническая документация*:*

При создании программы, одного лишь кода, как правило, недостаточно. Должен быть предоставлен некоторый текст, описывающий различные аспекты того, что именно делает код. Такая документация часто включается непосредственно в исходный код или предоставляется вместе с ним.

Подобная документация имеет сильно выраженный технический характер и в основном используется для определения и описания API, структур данных и алгоритмов.

Пользовательская документация*:*

В отличие от технической документации, сфокусированной на коде и том, как он работает, пользовательская документация описывает лишь то, как использовать программу.

Обычно, пользовательская документация представляет собой руководство пользователя, которое описывает каждую функцию программы, а также шаги, которые нужно выполнить для использования этой функции.

Пользовательская документация ПО (user documentation) объясняет пользователям, как они должны действовать, чтобы применить данное ПО.

Различают две категории пользователей ПО: ординарных пользователей ПО и администраторов ПО.

Типичным можно считать следующий состав пользовательской документации для достаточно больших ПО:

- общее функциональное описание ПО. Дает краткую характеристику функциональных возможностей ПО. Предназначено для пользователей, которые должны решить, насколько необходимо им данное ПО.

- руководство по инсталляции ПО. Предназначено для системных администраторов. Должно детально предписывать, как устанавливать системы в конкретной среде, содержать описание машинно-считываемого носителя, на котором поставляется ПО, файлы, представляющие ПО, и требования к минимальной конфигурации аппаратуры.

- инструкция по применению ПО. Предназначена для ординарных пользователей. Содержит необходимую информацию по применению ПО, организованную в форме удобной для ее изучения.

- справочник по применению ПО. Предназначен для ординарных пользователей. Содержит необходимую информацию по применению ПО, организованную в форме удобной для избирательного поиска отдельных деталей.

- руководство по управлению ПО. Предназначено для системных администраторов. Оно должно описывать сообщения, генерируемые, когда ПО взаимодействует с другими системами, и как реагировать на эти сообщения. Кроме того, если ПО использует системную аппаратуру, этот документ может объяснять, как сопровождать эту аппаратуру.

Разработка пользовательской документации начинается сразу после создания внешнего описания. Качество этой документации может существенно определять успех ПО. Она должна быть достаточно проста и удобна для пользователя (в противном случае это ПС, вообще, не стоило создавать). Поэтому, хотя черновые варианты (наброски) пользовательских документов создаются основными разработчиками ПО, к созданию их окончательных вариантов часто привлекаются профессиональные технические писатели. Кроме того, для обеспечения качества пользовательской документации разработан ряд стандартов, в которых предписывается порядок разработки этой документации, формулируются требования к каждому виду пользовательских документов и определяются их структура и содержание.

Маркетинговая документация*:*

Для многих приложений необходимо располагать рядом с ними рекламные материалы, с тем, чтобы заинтересовать людей, обратив их внимание на продукт. Такая форма документации имеет целью:

- подогреть интерес к продукту у потенциальных пользователей
- информировать их о том, что именно делает продукт, с тем, чтобы их ожидания совпадали с тем, что они получат
- объяснить положение продукта по сравнению с конкурирующими решениями

Формальные требования к документации программного обеспечения описаны в ЕСПД (Единая система программной документации), неформально: состав документации к программному обеспечению состоит из описания внешнего эффекта ПО и описания его внутреннего устройства.

Первая часть документации (пользовательская), так называемая «Инструкция пользователю» (или «Руководство пользователю») предназначена для того, кто собирается использовать программное обеспечение, не вникая в подробности его внутреннего устройства; вторая («Руководство программисту») необходима при модификации ПО или при необходимости исправить в нем ошибку.

Модульное программирование

При разработке больших программ необходимо их упрощение, которое достигается модульным программированием, т.е. разработкой программы по частям, которые называются программными модулями. Другой его задачей является многократное использование программистских знаний.

Модуль характеризуют

- один вход и один выход,

- функциональная завершенность,

- логическая независимость (результат работы модуля не зависит от работы других модулей),

- слабые информационные связи с другими программными модулями (обмен информацией с другими модулями минимизирован),

- размер и сложность программного элемента ограничивается разумными пределами.

Программный модуль является самостоятельным программным продуктом.

Характеристиками программного модулями для оценки его приемлемости являются:

1. Размер модуля. Он должен составлять от нескольких десятков до нескольких сотен операторов.

2. Прочность модуля – мера его внутренних связей. Функционально прочный модуль реализует какую-либо одну определенную функцию и может использовать другие модули. Такой вид прочности рекомендуется для использования. Высшей степенью прочности обладает информационно прочный программный модуль – модуль, выполняющий несколько над одной структурой данных, которая неизвестна вне этого модуля.

3. Сцепление модуля – мера его зависимости по способу передачи данных от других модулей.

Для оценки степени сцепления разделяют шесть видов по:

содержимому – худший вид сцепления, например прямые ссылки на содержимое, не рекомендуется для использования;

общей областью данных – также плохой вид сцепления, нерекомендуемый для использования;

образцу – обмен данными, объединенными в структуры, обеспечивает неплохие характеристики по сравнению с предыдущими,

управлению – один модуль посылает другому некоторый информационный объект (флаг), предназначенный для управления внутренней логикой модуля. Таким способом часто выполняют настройку режимов работы ПО. Подобные настройки снижают наглядность взаимодействия и обеспечивают не лучшие характеристики технологичности ПО.

внешним ссылкам – модули ссылаются на один и тот же глобальный объект,

данным – данные передаются как значения параметров, это единственный вид сцепления, который рекомендуется для использования современной технологией программирования.

4. Рутинность модуля – независимость от предыстории обращений к нему. Модуль называют зависимым от предыстории, если результат обращения к нему зависит от внутреннего состояния этого модуля, хранящего следы обращений к нему. Хотя зависящие от предыстории модули провоцируют появление в программах неуловимых ошибок, во многих случаях они являются наиболее информационно прочными. Наиболее приемлемая здесь рекомендация: следует всегда использовать рутинный модуль, если это не приводит к плохим сцеплениям модулей; зависящие от предыстории модули следует использовать только в случаях, когда это необходимо для сцепления; в спецификации зависящего от предыстории модуля должна быть четко сформулирована эта зависимость, чтобы было возможно прогнозировать поведение данного модуля при разных последующих обращениях к нему.

5. Связность модулей – мера прочности соединения функциональных и информационных объектов внутри одного модуля. Размещение сильно связанных элементов в одном модуле уменьшает межмодульные связи, а размещение сильно связанных элементов в разные модули усложняет их взаимодействие. Различают виды связности (в порядке убывания уровня):

- функциональная – модуль предназначен для выполнения одной функции, имеет максимальную связность и, следовательно, простоту в компиляции, тестировании, сопровождении.

- последовательная – результат обработки одной функцией служит исходными данными для другой функции. Такой модуль можно разбить на два модуля и более,

- информационная – связь между функциями, обрабатывающими одни и те же данные,

- процедурная – связь между функциями или данными, которые являются частями одного процесса,

- временная – функции выполняются параллельно, а данные используются в некотором интервале времени,

- логическая – строится на основе объединения данных или функций в одну логическую группу,

- случайная – как правило, не связанные между собой элементы.

В таблице представлены характеристики различных видов связности по экспертным оценкам.

Вид связности	Сцепление в баллах	Наглядность (понятность)	Возможность изменения	Сопровождаемость
функциональная	10	хорошая	хорошая	хорошая
последовательная	9	хорошая	хорошая	хорошая
информационная	8	средняя	средняя	средняя
процедурная	5	средняя	средняя	средняя
временная	3	средняя	средняя	средняя
логическая	1	плохая	плохая	плохая
случайная	0	плохая	плохая	плохая

Из таблицы видно, что лучше использовать функциональную, последовательную и информационную связность.

Модульная структура программных продуктов

Модульная структура программы представляет собой древовидную структуру, в узлах которой размещаются программные модули, а направленные дуги показывают статическую подчиненность модулей.

Функция верхнего уровня обеспечивается главным модулем, он управляет выполнением нижестоящих модулей.

При определении набора модулей необходимо учитывать следующее:

1) Модуль вызывается на выполнение вышестоящим по иерархии модулем и, закончив работу, возвращает ему управление;

2) Принятие основных решений в алгоритме выносится на максимально высокий уровень по иерархии;

3) Если в разных местах алгоритма используется одна и та же функция, то она оформляется в отдельный модуль, который будет вызываться по мере необходимости.

Существуют разные методы разработки модульной структуры программы:

1. Метод восходящей разработки
2. Метод нисходящей разработки
3. Конструктивный метод
4. Архитектурный подход

Метод восходящей разработки

Сначала строится древовидная модульная структура программы. Затем поочередно проектируются и разрабатываются модули программы, начиная с модулей самого нижнего уровня, затем предыдущего уровня и т. д. То есть модули реализуются в таком порядке,

чтобы для каждого программируемого модуля были уже запрограммированы все модули, к которым он может обращаться. После того как все модули программы запрограммированы, производится их поочередное тестирование и отладка в таком же восходящем порядке. Достоинство метода заключается в том, что каждый модуль при программировании выражается через уже запрограммированные непосредственно подчиненные модули, а при тестировании использует уже отлаженные модули.

Недостатки метода восходящей разработки:

на нижних уровнях модульной структуры спецификации могут быть еще определены не полностью, что может привести к полной переработке этих модулей после уточнения спецификаций на верхнем уровне;

для восходящего тестирования всех модулей, кроме головного, который является модулем самого верхнего уровня, приходится создавать вызывающие программы, что приводит к созданию большого количества отладочного материала, но не гарантирует, что результаты тестирования верны;

головной модуль проектируется и реализуется в последнюю очередь, что не дает продемонстрировать его заказчику для уточнения спецификаций.

Метод нисходящей разработки

Как и в предыдущем методе, сначала строится модульная структура программы в виде дерева. Затем проектируются и реализуются модули программы, начиная с модуля самого верхнего уровня – головного, далее разрабатываются модули уровнем ниже и т. д. При этом переход к программированию какого-либо модуля осуществляется только в том случае, если уже запрограммирован модуль, который к нему обращается. Затем производится их поочередное тестирование и отладка в таком же нисходящем порядке. При таком порядке разработки программы вся необходимая глобальная информация формируется своевременно, т. е. ликвидируется весьма неприятный источник просчетов при программировании модулей. Существенно облегчается и тестирование модулей, производимое при нисходящем тестировании программы. Первым тестируется головной модуль программы, который представляет всю тестируемую программу, при этом все модули, к которым может обращаться головной, заменяются их имитаторами (так называемыми «заглушками»). Каждый имитатор модуля является простым программным фрагментом, реализующим сам факт обращения к данному модулю с необходимой для правильной работы программы обработкой значений его входных параметров и с выдачей, если это необходимо, подходящего результата. Далее производится тестирование следующих по уровню модулей. Для этого имитатор выбранного для тестирования модуля заменяется самим модулем, и добавляются имитаторы модулей, к которым может обращаться тестируемый модуль. При таком подходе каждый модуль будет тестироваться в «естественных» состояниях информационной среды, возникающих к моменту обращения к этому модулю при выполнении тестируемой программы. Таким образом, большой объем «отладочного» программирования заменяется программированием достаточно простых имитаторов используемых в программе модулей.

Недостатком нисходящего подхода к программированию является необходимость абстрагироваться от реальных возможностей выбранного языка программирования и придумывать абстрактные операции, которые позже будут реализованы с помощью модулей. Однако способность к таким абстракциям является необходимым условием разработки больших программных средств.

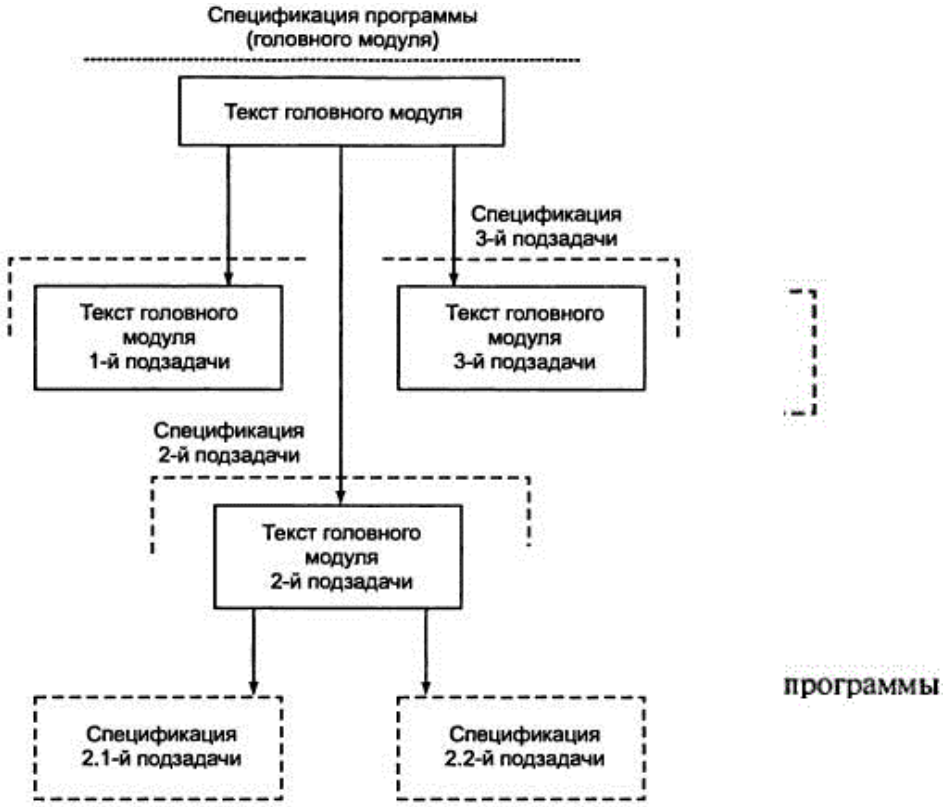
Рассмотренные выше методы (нисходящей и восходящей разработок), являющиеся классическими, требуют, чтобы модульная древовидная структура была готова до начала программирования модулей. Как правило, точно и содержательно разработать структуру программы до начала программирования невозможно. При конструктивном и архитектурном подходах к разработке модульная структура формируется в процессе реализации модулей.

Конструктивный подход

Конструктивный подход к разработке программы представляет собой модификацию нисходящей разработки, при которой модульная древовидная структура программы формируется в процессе программирования модуля. Сначала программируется головной модуль, исходя из спецификации программы в целом (спецификация программы является одновременно спецификацией головного модуля). В процессе программирования головного модуля в случае, если эта программа достаточно большая, выделяются подзадачи (некоторые функции) и для них создаются спецификации реализующих эти подзадачи фрагментов программы. В дальнейшем каждый из этих фрагментов будет представлен поддеревом модулей (спецификация выделенной функции является одновременно спецификацией головного модуля этого поддерева).

Таким образом, на первом шаге разработки программы (при программировании ее головного модуля) формируется верхняя часть дерева.

По тому же принципу производится программирование следующих по уровню специфицированных, но еще не запрограммированных модулей в соответствии со сформированным деревом. В результате к дереву добавляются очередные уровни, как показано на рис. 5



Второй шаг формирования модульной структуры программы при конструктивном подходе

Рисунок 5. Формирование уровней программируемых модулей в конструктивном подходе

Архитектурный подход

Архитектурный подход к разработке программы представляет собой модификацию восходящей разработки, при которой модульная структура программы формируется в процессе программирования модуля. Целью разработки в данном методе является повышение уровня языка программирования, а не разработка конкретной программы. Это означает, что для заданной предметной области выделяются типичные функции, специфицируются, а затем и программируются отдельные программные модули, выполняющие

эти функции. Сначала в виде модулей реализуются более простые функции, а затем создаются модули, использующие уже имеющиеся функции, и т. д. Это позволяет существенно сократить трудозатраты на разработку конкретной программы путем подключения к ней уже имеющихся и проверенных на практике модульных структур нижнего уровня, что также позволяет бороться с дублированием в программировании.

В связи с этим программные модули, создаваемые в рамках архитектурного подхода, обычно параметризуются, чтобы облегчить их применение настройкой параметров.

Нисходящая реализация

В классическом методе нисходящей разработки сначала все модули разрабатываемой программы программируются, а затем тестируются в нисходящем порядке. При этом тестирование и отладка модулей могут привести к изменению спецификации подчиненных модулей и даже к изменению самой модульной структуры программы. В результате может оказаться, что часть модулей вообще не нужна в данной структуре, а часть модулей придется переписывать. Более рационально каждый запрограммированный модуль тестировать сразу же до перехода к программированию другого модуля. Такой метод в литературе получил название метода нисходящей реализации.

Целенаправленная конструктивная реализация

В зависимости от того, в какой последовательности в процессе разработки программы обходятся узлы дерева, существуют разновидности описанных выше методов.

На рис. представлена общая схема классификации методов разработки структуры программы.

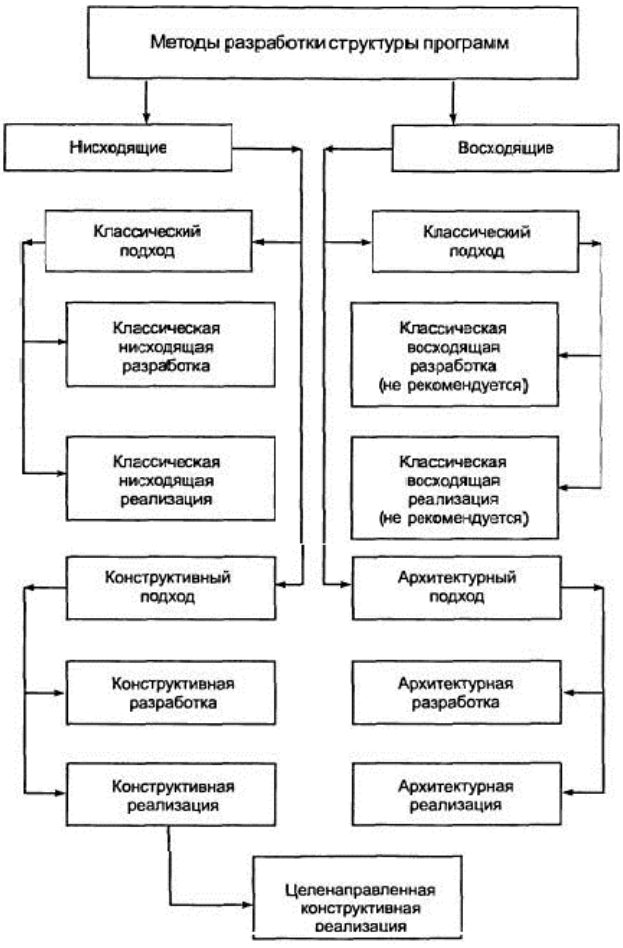


Рисунок 6. Классификация методов разработки программного обеспечения

Например, метод целенаправленной конструктивной реализации, при котором в рамках конструктивного подхода реализуются сначала модули, необходимые для самого простого рабочего варианта программы, остальные модули заменяются их имитаторами. Затем к имеющемуся варианту добавляются другие модули, обеспечивающие работу программы для других наборов данных, и так далее до полной реализации программы. Достоинством этого метода является то, что уже на достаточно ранней стадии создается работающий вариант разрабатываемой программы. Психологически это играет роль допинга, резко повышающего эффективность разработчика.

Особенности проектирования программного обеспечения на основе объектно-ориентированного подхода

Объектный подход к разработке сложных программных систем предполагает необходимость этапов анализа и моделирования предметной области до непосредственного кодирования программы на каком-либо языке программирования.

ООП привносит нам два ключевых понятия: Класс и Объект.

Класс – это абстрактный тип данных. С помощью класса описывается некоторая сущность (ее характеристики и возможные действия). Например, класс может описывать студента, автомобиль и т.д.

Описав класс, можно создать его экземпляр – объект. Объект – это уже конкретный представитель класса (переменная типа класс).

Объектный подход применяется на всех основных стадиях жизненного цикла программного обеспечения и включает в себя несколько этапов:

1 Объектно-ориентированный анализ – методология анализа предметной области, который выполняется с целью выделения объектов и классов в качестве требования к проектируемой системе.

2 Объектно-ориентированное проектирование – методология проектирования, объединяющая в себе процесс декомпозиции объектов и приемы их представления логической и физической моделью проектируемой системы.

3 Объектно-ориентированное программирование – методология (парадигма программирования), в основе которой лежит построение программы в форме взаимодействия объектов, каждый из которых является экземпляром определенного класса.

ООП основывается на нескольких базовых принципах

Инкапсуляция позволяет скрывать внутреннюю реализацию класса. Внутри него могут быть реализованы элементы - методы, поля, свойства, индексаторы, к которым доступ другим элементам программы (другим классам, объектам, пользователям) должен быть ограничен или даже полностью запрещен.

Наследование – механизм, позволяющий строить иерархию классов. Это предполагает определение базового класса (или прародителя), а затем его использование для построения производных классов (потомков). Причем каждый производный класс на¬следует все свойства своего прародителя, включая как данные, так и функции. И при этом они могут также иметь свои дополнительные свойства. Таким образом, они не дублируют свойства базовый класс, а только используют его возможности.

Производный класс, в свою очередь может стать базовым классом для нового класса, и таким образом создается иерархия классов.

Полиморфизм – способность объектов одной иерархии (с одним интерфейсом) иметь различную реализацию, т.е. реализовать одноименные методы и свойства собственным способом. Полиморфизм реализуется виртуальными и абстрактными методами и перегрузкой операций.

Принцип полиморфизма позволяет родственным объектам могут вести себя по-

разному. Для этого виртуальный метод базового класса переопределяют в каждом классе потомке различными способами.

Отличие переопределенных функций от виртуальных заключается в стратегии динамического связывания. Она применяется компилятором ко всем виртуальным методам и означает, что решение, какой из экземпляров виртуального метода должен быть вызван, принимается не на этапе компиляции, а на этапе выполнения, когда уже известен тип объекта.

Абстрактные методы и содержащие их абстрактные классы не имеют собственной реализации и, по своей сути, являются некоторыми шаблонами для производных классов.

Перегрузка операций позволяет применять обычные математические операции и операции сравнения не к полям объектами, а непосредственно к самим объектам.

Абстракция позволяет выделять из некоторой сущности только необходимые характеристики и методы, которые в полной мере (для поставленной задачи) описывают объект. Например, создавая класс для описания студента, мы выделяем только необходимые его характеристики, такие как ФИО, номер зачетной книжки, группа. Здесь нет смысла добавлять поле вес или имя его кота/собаки и т.д.

Класс является типом данных, определяемым пользователем. Он представляет собой модель реального объекта в виде данных и функций для работы с ними.

Данные, содержащиеся в описании класса, называются его полями, а функции – методами. Поля и методы являются элементами класса.

Элементы класса объявляются с различными спецификаторами доступа, которые позволяют реализовывать принцип инкапсуляции:

Закрытые элементы предназначены только для внутреннего ис-пользования в классе и обеспечивают принцип инкапсуляции. Как правило, они являются полями. Они доступны только методам класса, в состав которого они входят.

Открытые элементы доступны снаружи класса. Как правило, они отве¬чают за внешний интерфейс класса и являются методами.

Защищенные элементы доступны только для методов данного класса и классов, производных от него.

При создании класса программист самостоятельно решает, какие из элементов класса будут общедоступными, а какие закрытыми. Однако описание большинства классов имеет типовую схему: закрытая часть содержит данные (поля), а открытая – функции для работы с этими данными (методы). Секция защищенных элементов используется в случаях предположения, что данный класса будет в дальнейшем использоваться в качестве базового класса в наследовании.

Существуют следующие отношения между классами: ассоциация, наследование, агрегация, зависимость.

Если два класса взаимодействуют друг с другом концептуально, то их взаимодействие называется ассоциацией.

Ассоциация выявляется на ранней стадии проектирования, и в дальнейшем, как правило, конкретизируется.

Наследование предполагает, что производный класс, обладая всеми свойствами базового (родительского класса) имеет собственные свойства, характерные только для него.

Отношение «агрегация» подразумевает включение в качестве составной части объектов другого класса (отношение целое/часть).

Строгая агрегация имеет название – композиция. Она означает, что компонент не может исчезнуть, пока объект «целое» существует.

Отношение зависимости (использования) между классами показывает, что один класс пользуется некоторыми услугами другого класса.

Современное объектно-ориентированное программирование базируется не только на самих идеях объектно-ориентированного подхода, но и на применении шаблонов проектирования (паттернов), включающих в себя наиболее удачные решения типичных про-

блем, возникающих при разработке программ.

Достоинства объектно-ориентированного подхода в программировании заключаются в следующем:

- хорошая структурированность программ, что облегчает понимание их алгоритма;
- возможность разбиения программного продукта на небольшие компоненты и тестирование их по отдельности;
- легкость дальнейшего расширения.

Тестирование и отладка программного обеспечения

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

Общепринятая практика состоит в том, что после завершения продукта и до передачи его заказчику независимой группой тестировщиков проводится тестирование ПО. Эта практика часто выражается в виде отдельной фазы тестирования (в общем цикле разработки ПО), которая часто используется для компенсирования задержек, возникающих на предыдущих стадиях разработки.

Другая практика тестирования заключается в его проведении на протяжении всего жизненного цикла разработки и сопровождения программного обеспечения. Уровень тестирования определяет то, над чем производятся тесты: над отдельным модулем, группой модулей или системой, в целом. Проведение тестирования на всех уровнях системы - это залог успешной реализации и сдачи проекта. Второй путь трудозатратнее, но качество тестирование в нем выше.

Уровни тестирования:

1. Компонентное или Модульное тестирование
2. Интеграционное тестирование
3. Системное тестирование
4. Приемочное тестирование

Компонентное или Модульное тестирование (Component Testing or Unit Testing) – тестируется минимально возможный для тестирования компонент, например, класс или функция. Обычно компонентное (модульное) тестирование проводится вызывая код, который необходимо проверить и при поддержке сред разработки, таких как фреймворки (frameworks - каркасы) для модульного тестирования или инструменты для отладки. Все найденные дефекты, как правило, исправляются в коде без формального их описания.

Один из наиболее эффективных подходов к компонентному (модульному) тестированию – это подготовка автоматизированных тестов до начала основного кодирования (разработки) программного обеспечения. Это называется разработка от тестирования (test-driven development) или подход тестирования вначале (test first approach). При этом подходе создаются и интегрируются небольшие куски кода, напротив которых запускаются тесты, написанные до начала кодирования. Разработка ведется до тех пор пока все тесты не будут успешно пройдены.

Компонентное и модульное тестированием по-существу представляют одно и то же, разница лишь в том, что в компонентном тестировании в качестве параметров функций используют реальные объекты и драйверы, а в модульном тестировании – конкретные значения.

Интеграционное тестирование (Integration Testing) Интеграционное тестирование предназначено для проверки связи между компонентами, а также взаимодействия с различными частями системы (операционной системой, оборудованием либо связи между различными системами).

Уровни интеграционного тестирования:

1 Компонентный интеграционный уровень (*Component Integration testing*) Проверяется взаимодействие между компонентами системы после проведения компонентного тестирования.

2 Системный интеграционный уровень (*System Integration Testing*) Проверяется взаимодействие между разными системами после проведения системного тестирования.

Подходы к интеграционному тестированию:

Снизу вверх (*Bottom Up Integration*)

Все низкоуровневые модули, процедуры или функции собираются воедино и затем тестируются. После чего собирается следующий уровень модулей для проведения интеграционного тестирования. Данный подход считается полезным, если все или практически все модули, разрабатываемого уровня, готовы. Также данный подход помогает определить по результатам тестирования уровень готовности приложения

Сверху вниз (*Top Down Integration*)

Вначале тестируются все высокоуровневые модули, и постепенно один за другим добавляются низкоуровневые. Все модули более низкого уровня симулируются заглушками с аналогичной функциональностью, затем по мере готовности они заменяются реальными активными компонентами.

Большой взрыв (*"Big Bang" Integration*)

Все или практически все разработанные модули собираются вместе в виде законченной системы или ее основной части, и затем проводится интеграционное тестирование. Такой подход очень хорош для сохранения времени. Однако если тест кейсы и их результаты записаны не верно, то сам процесс интеграции сильно осложнится, что станет преградой для команды тестирования при достижении основной цели интеграционного тестирования

Системное тестирование (System Testing)

Основной задачей системного тестирования является проверка как функциональных, так и не функциональных требований в системе в целом. При этом выявляются дефекты, такие как неверное использование ресурсов системы, непредусмотренные комбинации данных пользовательского уровня, несовместимость с окружением, непредусмотренные сценарии использования, отсутствующая или неверная функциональность, неудобство использования и т.д. Для минимизации рисков, связанных с особенностями поведения в системы в той или иной среде, во время тестирования рекомендуется использовать окружение максимально приближенное к тому, на которое будет установлен продукт после выдачи.

Можно выделить два подхода к системному тестированию:

на базе требований (*requirements based*)

Для каждого требования пишутся тестовые случаи (test cases), проверяющие выполнение данного требования.

на базе случаев использования (*use case based*)

На основе представления о способах использования продукта создаются случаи использования системы (Use Cases). По конкретному случаю использования можно определить один или более сценариев. На проверку каждого сценария пишутся тест кейсы (test cases), которые должны быть протестированы.

Выделяют также внутреннее (алфа-тестирование) и публичное (бета-тестирование):

альфа-тестирование – имитация реальной работы с системой штатными разработчиками либо реальная работы с системой потенциальными пользователями/заказчиком на стороне разработчика. Часто альфа-тестирование применяется для законченного продукта в качестве внутреннего приемочного тестирования.

бета-тестирование – в некоторых случаях выполняется распространение версии с ограничениями (по функциональности или времени работы) для некоторой группы лиц, чтобы убедится, что продукт содержит достаточно мало ошибок. Иногда бета-тестирование выполняется для получения обратной связи о продукте от его будущих пользователей.

Приемочное тестирование (Acceptance Testing)

Формальный процесс тестирования, который проверяет соответствие системы требованиям и проводится с целью:

определения удовлетворяет ли система приемочным критериям;

вынесения решения заказчиком или другим уполномоченным лицом принимается приложение или нет.

Приемочное тестирование выполняется на основании набора типичных тестовых случаев и сценариев, разработанных на основании требований к данному приложению. Решение о проведении приемочного тестирования принимается, когда:

продукт достиг необходимого уровня качества;

заказчик ознакомлен с Планом Приемочных Работ(Product Acceptance Plan) или иным документом, где описан набор действий, связанных с проведением приемочного тестирования, дата проведения, ответственные и т.д. Фаза приемочного тестирования длится до тех пор, пока заказчик не выносит решение об отправлении приложения на доработку или выдаче приложения.

Тестовые артефакты

В соответствие с процессами или методологиями разработки ПО, во время проведения тестирования создается и используется определенное количество тестовых артефактов (документы, модели и т.д.). Наиболее распространенными тестовыми артефактами являются:

План тестирования (Test Plan) – это документ описывающий весь объем работ по тестированию, начиная с описания объекта, стратегии, расписания, критериев начала и окончания тестирования, до необходимого в процессе работы оборудования, специальных знаний, а также оценки рисков с вариантами их разрешения.

Набор тест кейсов и тестов (Test Case & Test suite) – это последовательность действий, по которой можно проверить соответствует ли тестируемая функция установленным требованиям.

Дефекты / Баг Репорты (Bug Reports / Defects) – это документы, описывающие ситуацию или последовательность действий, приведшую к некорректной работе объекта тестирования, с указанием причин и ожидаемого результата.

Подробнее о указанных артефактах можно узнать, пройдя по ссылке http://www.protesting.ru

Тестирование «белого ящика» и «черного ящика»

В терминологии профессионалов тестирования, фразы «тестирование белого ящика» и «тестирование чёрного ящика» относятся к тому, имеет ли разработчик тестов доступ к исходному коду тестируемого ПО, или же тестирование выполняется через пользовательский интерфейс либо прикладной программный интерфейс, предоставленный тестируемым модулем.

При тестировании белого ящика (англ. white-box testing, также говорят – прозрачного ящика), разработчик теста имеет доступ к исходному коду программ и может писать код, который связан с библиотеками тестируемого ПО. Это типично для юнит-тестирования (англ. unit testing), при котором тестируются только отдельные части системы. Оно обеспечивает то, что компоненты конструкции – работоспособны и устойчивы, до определённой степени. При тестировании белого ящика используются метрики покрытия кода или мутационное тестирование.

При тестировании чёрного ящика, тестировщик имеет доступ к ПО только через те же интерфейсы, что и заказчик или пользователь, либо через внешние интерфейсы, позволяющие другому компьютеру либо другому процессу подключиться к системе для тести-

рования. Например, тестирующий модуль может виртуально нажимать клавиши или кнопки мыши в тестируемой программе с помощью механизма взаимодействия процессов, с уверенностью в том, все ли идёт правильно, что эти события вызывают тот же отклик, что и реальные нажатия клавиш и кнопок мыши. Как правило, тестирование чёрного ящика ведётся с использованием спецификаций или иных документов, описывающих требования к системе. Как правило, в данном виде тестирования критерий покрытия складывается из покрытия структуры входных данных, покрытия требований и покрытия модели (в тестировании на основе моделей).

При тестировании серого ящика разработчик теста имеет доступ к исходному коду, но при непосредственном выполнении тестов доступ к коду, как правило, не требуется.

Если «альфа-» и «бета-тестирование» относятся к стадиям до выпуска продукта (а также, неявно, к объёму тестирующего сообщества и ограничениям на методы тестирования), тестирование «белого ящика» и «чёрного ящика» имеет отношение к способам, которыми тестировщик достигает цели.

Бета-тестирование в целом ограничено техникой чёрного ящика (хотя постоянная часть тестировщиков обычно продолжает тестирование белого ящика параллельно бета-тестированию). Таким образом, термин «бета-тестирование» может указывать на состояние программы (ближе к выпуску чем «альфа»), или может указывать на некоторую группу тестировщиков и процесс, выполняемый этой группой. Итак, тестировщик может продолжать работу по тестированию белого ящика, хотя ПО уже «в бете» (стадия), но в этом случае он не является частью «бета-тестирования» (группы/процесса).

Основные концепции проектирования операционных систем

Год за годом происходит эволюция структуры и возможностей операционных систем. В последнее время в состав новых операционных систем и новых версий уже существующих операционных систем вошли некоторые структурные элементы, которые внесли большие изменения в природу этих систем. Современные операционные системы отвечают требованиям постоянно развивающегося аппаратного и программного обеспечения. Они способны управлять работой многопроцессорных систем, работающих быстрее обычных машин, высокоскоростных сетевых приспособлений и разнообразных запоминающих устройств, число которых постоянно увеличивается. Несмотря на это есть ряд базовых концепций, которые реализованы во всех существующих операционных системах.

Модели операционных систем

Существует множество способов структурирования операционных систем. В небольших операционных системах, например, в MS-DOS, используют принцип организации системы, как набора процедур, каждую из которых может вызывать любая пользовательская процедура. Такая модель называется монолитной (Рис. 2.1).

Такая структура не обеспечивает изоляции данных, поскольку в разных участках кода используется информация об устройстве всей системы. Расширение операционных систем такого типа затруднительно, так как изменение некоторой процедуры может вызвать ошибки в других частях системы, на первый взгляд не имеющих к ней отношения.

Во всех монолитных операционных системах, кроме самых простых, приложения отделены от собственно кода операционной системы. Иными словами, код исполняется в привилегированном режиме процессора, который в литературе часто называется ***режимом ядра*** (kernel mode) и имеет доступ к данным системы и аппаратуре. Программы пользователя исполняются в непривилегированном, так называем ***режиме пользователя*** (user mode), в котором им предоставлен ограниченный набор интерфейсов и ограниченный доступ к системным данным. Когда программа пользовательского режима вызывает системный сервис, процессор перехватывает вызов и переключает вызывающий поток в режим ядра. Когда выполнение системного вызова завершается, операционная система переключает поток обратно в пользовательский режим и дает возможность вызывающей программе продолжить выполнение.

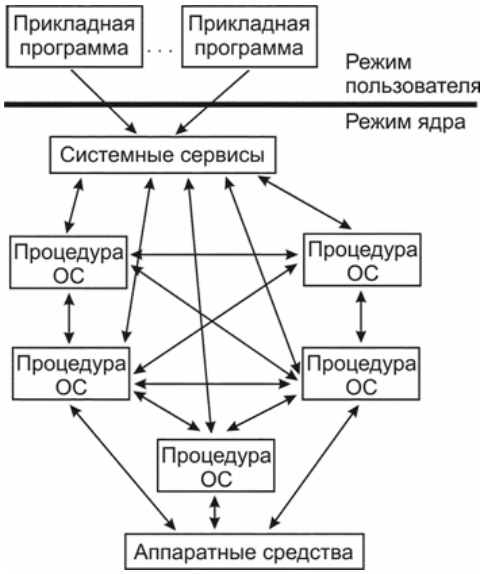


Рисунок 1. Монолитная модель операционной системы

Другой подход к структурированию системы предполагает разделение ее на модули, наслоенные один поверх другого. Каждый модуль предоставляет набор функций, которые могут вызываться другими модулями. Код, расположенный в некотором слое, вызывает код только из нижележащих слоев (Рис. 2). В некоторых операционных системах, строящихся по данной модели, например, в VAX/VMS или в системе Multics, многослойность даже принудительно обуславливается аппаратными средствами.

Одним из преимуществ послойной организации операционной системы является то, что код каждого слоя получает доступ только к необходимым ему интерфейсам и структурам данных нижележащих слоев. Таким образом, уменьшается объем кода, обладающего неограниченной властью. Кроме того, такая структура позволяет при отладке операционной системы начинать с самого нижнего слоя и добавлять по одному уровню до тех пор, пока вся система не стает работать правильно. Послойная структура облегчает и расширение систем, можно целиком заменить любой уровень, не затрагивая остальных частей.

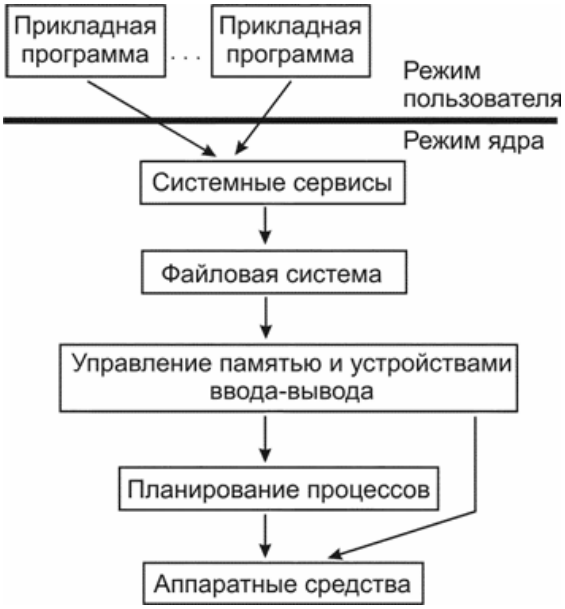


Рисунок 2. Послойная модель операционной системы

Третий подход к проектированию операционных систем – это модель клиент-сервер. Идея его состоит в разделении системы на несколько процессов, каждый из которых реализует один набор сервисов: например, распределение памяти, создание процессов или планирование процессов. Каждый сервер выполняется в режиме пользователя, все время проверяя, не обратился ли к нему за обслуживанием какой-либо клиент. Клиент, которым может быть либо другой компонент операционной системы, либо прикладная программа, запрашивает выполнение сервиса, посылая серверу сообщение. Ядро операционной системы, выполняющееся в режиме ядра, доставляет сообщение серверу. Тот выполняет запрашиваемые действия, после чего ядро возвращает клиенту результаты в виде другого сообщения (Рис. 3).

При использовании клиент-серверного подхода получается операционная система, состоящая из автономных компонентов небольшого размера. Поскольку все серверы выполняются как отдельные процессы в режиме пользователя, авария и, возможно, перезапуск одного из серверов не нарушает работы остальных частей системы. Более того, разные серверы могут выполняться на разных процессорах многопроцессорного компьютера или даже на разных компьютерах. Это делает операционную систему, построенную по такой модели пригодной для распределенных вычислительных сред.

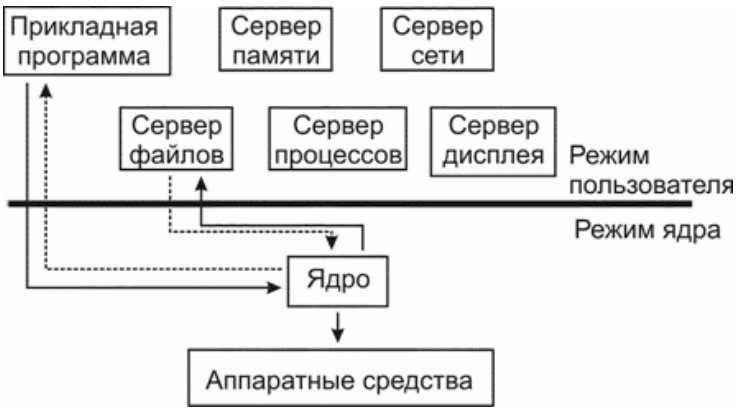


Рисунок 3. Модель клиент-сервер

Существует множество разновидностей клиент-серверных систем, часть которых выполняют в режиме ядра малый объем работы, в то время, как другие –больший. Например, операционная система Mach, один из современных примеров клиент-серверной архитектуры, имеет ядро минимального размера, в функции которого входит планирование потоков, передача сообщений, управление виртуальной памятью и драйверы устройств. Все остальное, включая различные интерфейсы прикладных программ, файловые системы, сетевую поддержку, работает в пользовательском режиме.

В последнее время все более популярной становится объектная модель операционной системы. Как и в случае других больших программных систем, трудно найти одну, главную программу, которая управляет всей операционной системой. Таким образом, вместо того, чтобы разрабатывать систему сверху вниз, по объектно-ориентированной методологии сначала рассматривают данные, с которыми должна работать программа для выполнения своей задачи. Для операционных систем такими данными являются системные ресурсы – файлы, процессы, память и так далее.

Основная цель разработки системы с ориентацией на данные – создание программного обеспечения, которое можно было бы легко, а главное дешево, изменять. Один из способов минимизации необходимых изменений в объектно-ориентированных программах – это сокрытие физического представления данных внутри объектов. Объект представляет собой структуру данных, физический формат которой скрыт в определении типа. Он имеет набор свойств, называемых атрибутами, и с ним работает группа сервисов.

Многие операционные системы используют объекты для представления системных ресурсов. Каждый системный ресурс, который могут совместно использовать несколько процессов, реализован как объект и обрабатывается объектными сервисами. Такой подход уменьшает влияние изменений, которые могут происходить в системе с течением времени. Например, если изменение в операционной системе вызвано изменением в аппаратуре, необходимо изменить только объект, представляющий данный аппаратный ресурс, и его сервисы. При этом код, который использует объект, не нуждается в модификации. Аналогично, если нужно ввести в систему поддержку нового устройства, создается новый объект, и добавление его к системе не нарушает существующего кода.

Помимо того, что уменьшается влияние изменений, построение операционной системы на основе объектов имеет еще ряд несомненных преимуществ:

- Доступ системы к ресурсам и работа с ними унифицированы. Создание, удаление и ссылка на объект осуществляется совершенно аналогично. И поскольку каждый ресурс – это объект, контроль использования ресурсов сводится к отслеживанию создания и использования объектов.

- Упрощается защита, так как для всех объектов она осуществляется одинаково. При попытке доступа к объекту подсистема защиты вмешивается и проверяет допустимость операции, независимо от того, какой ресурс представляет объект.

- Объекты предоставляют удобную и унифицированную базу для совместного использования ресурсов двумя или несколькими процессами. Для работы с объектами любого типа используются описатели объектов. Два процесса совместно используют объект тогда, когда каждый из них открыл его описатель. Система может отслеживать количество описателей, открытых для данного объекта, чтобы определить, действительно ли он все еще используется. После этого система может удалить объекты, которые более не используются.

Эволюция операционных систем

Первые вычислительные машины были очень дорогими, поэтому было важно использовать их как можно эффективнее. Простои, происходившие из-за несогласованности расписания, а также время, затраченное на подготовку задачи, – все это обходилось слишком дорого, и чтобы повысить эффективность работы, была предложена концепция пакетной операционной системы.

Первое упоминание об операционных системах данного вида относятся к 50-м годам 20-го столетия, и относится к компьютерам IBM 701 и IBM 704. Впоследствии эта концепция была усовершенствована, и в начале 60-х годов были разработаны пакетные операционные системы для компьютеров различных фирм.

Главная идея, лежащая в основе пакетных операционных систем, состоит в использовании особой программы, известной под названием ***монитор***. Используя операционную систему такого типа, пользователь не имел непосредственного доступа к вычислительной машине. Вместо этого он передавал свое задание на перфокартах или магнитной ленте оператору компьютера, который собирает разные задания в пакеты и помещает их в устройство ввода данных. Каждая программа составлена таким образом, что при завершении ее работы управление переходит к монитору, который автоматически загружает следующую программу. Тем самым уменьшается время простоя компьютера.

Монитор управляет последовательностью событий. Чтобы это было возможно, большая его часть должна всегда находиться в оперативной памяти и быть готовой к работе (Рис. 4). Эту часть монитора называют резидентным монитором. Оставшуюся часть составляют утилиты и общие функции, которые загружаются в начале выполнения каждого задания в виде подпрограмм, вызываемых программой пользователя, если они требуются.

Рисунок 4. Распределение памяти в пакетных системах

Монитор считывает с устройства ввода данных, в качестве которого обычно выступали устройства ввода с перфокарт или магнитной ленты, по одному заданию. При этом текущее задание размещается в области памяти, предназначенной для программ пользователя, и ему передается управление. По завершении задания оно возвращает управление монитору, который сразу же начинает считывать следующее задание. Результат исполнения каждого задания направляется на устройство вывода, например принтер.

После того как задание полностью считано в память, монитор отдает процессору команду перехода, по которой он должен начать исполнение программы пользователя. Процессор переходит к обработке программы пользователя и выполняет ее команды до

тех пор, пока не дойдет до конца или пока не возникнет сбойная ситуация. В любом из этих двух случаев следующей командой, которую процессор выполнит, будет команда монитора.

Таким образом, наличие монитора решает проблему очередности выполнения заданий и повышает загрузку вычислительной машины. Задания в пакетах выстраиваются в очередь и выполняются без простоев настолько быстро, насколько это возможно. Кроме того, монитор помогает в подготовке программы к исполнению. В пакетных операционных системах в каждое задание включаются простые команды языка управления заданиями (JCL – Job ControlLanguage). Это специальный тип языка программирования, используемый для того, чтобы отдавать команды монитору. Примерами таких команд могут служить команды ***$JOB***, ***$LOAD***, ***$RUN*** и ***$END***, соответственно обозначающие начало задания, загрузку программы, ее запуск и конец задания.

Таким образом, монитор, или пакетная операционная система представляет собой обычную компьютерную программу. Ее работа основана на способности процессора выбирать команды из различных областей основной памяти. При этом происходит передача и возврат управления.

Во время работы программы пользователя она не должна вносить изменения в область памяти, в которой находится монитор. Если же такая попытка предпринята, аппаратное обеспечение процессора должно обнаружить ошибку и передать управление монитору. Затем монитор снимет задачу с выполнения, распечатает сообщение об ошибке и загрузит следующее задание.

Чтобы предотвратить ситуацию, когда одна задача захватит безраздельный контроль над системой, используется системный таймер, который устанавливается в начале каждого задания. По истечении определенного промежутка времени программа пользователя останавливается и управление передается монитору.

Некоторые машинные команды имеют повышенные привилегии и могут исполняться только монитором. Если процессор натолкнется на такую команду во время исполнения программы пользователя, возникнет ошибка, при которой управление будет передано монитору. В число привилегированных команд входят команды ввода-вывода, и это значит, что все устройства ввода-вывода контролируются монитором. Это предотвращает случайное чтение программой пользователя команд управления, относящихся к следующему заданию. Если программе пользователя нужно произвести ввод-вывод, она должна запросить для выполнения этих операций монитор.

При работе пакетных операционных систем машинное время распределялось между исполнением программы пользователя и монитора. При этом в жертву приносились два вида ресурсов: монитор занимал некоторую часть оперативной памяти, им же потреблялось некоторое машинное время. И то и другое приводило к непроизводительным издержкам. Несмотря на это, простые пакетные системы существенно повышали эффективность использования компьютера.

Несмотря на повышение производительности, процессору часто приходилось простаивать даже при автоматическом выполнении заданий под управлением простой пакетной операционной системы. Проблема заключается в том, что устройства ввода-вывода работают намного медленнее, чем процессор. В среднем, 96% всего времени процессор ждет, пока устройства ввода-вывода закончат передачу данных в файл и из него, а лишь 4% тратятся на вычисления. Некоторое время процессор исполняет команды; затем, дойдя до команды ввода-вывода, он должен подождать, пока она не закончится. Только после этого процессор сможет продолжить работу (Рис. 5).

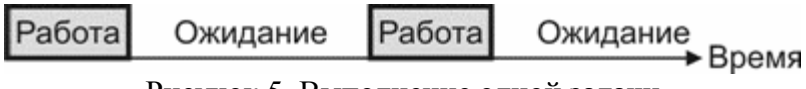


Рисунок 5. Выполнение одной задачи

Эффективность использования процессора можно повысить. Обычно у компьютера достаточно памяти, чтобы разместить в ней операционную систему, в нашем случае резидентный монитор, и программу пользователя. Предположим, что в памяти достаточно места для операционной системы и двух программ пользователя. Теперь, когда одно из заданий ждет завершения операций ввода-вывода, процессор может переключиться на другое задание, для которого в данный момент ввод-вывод, скорее всего, не требуется (Рис. 6).

Рисунок 6.Выполнение двух задач

Более того, если памяти достаточно для размещения большего количества программ, то процессор может выполнять их параллельно, переключаясь с одной на другую. Такой режим известен как ***многозадачность***, и является основной чертой современных операционных систем.

Такая организация работы многозадачной пакетной системы, как и работа простой пакетной системы, основана на некоторых аппаратных особенностях современных компьютеров. Наиболее важными из этих особенностей являются прерывания и процедура прямого доступа к памяти. Процесс прерывания программы в общих чертах описан в предыдущей главе. Прямой доступ к памяти обозначает механизм, когда информация из одной области памяти переносится в другую область памяти без участия основного процессора компьютера. Эти занимается специальное устройство, называемое контроллером прямого доступа к памяти. Процессору нужно лишь запрограммировать этот контроллер, указав адреса откуда и куда переписывать информацию.

Используя эти возможности, процессор генерирует команду ввода-вывода для одного задания и переходит к другому на то время, пока контроллером устройства выполняется ввод-вывод. После завершения операции ввода-вывода процессор получает прерывание, и управление передается программе обработки прерываний из состава операционной системы. Затем операционная система передает управление другому заданию.

Многозадачные операционные системы сложнее систем последовательной обработки заданий. Для того чтобы можно было обрабатывать несколько заданий одновременно, они должны находиться в основной памяти, а для этого требуется система управления памятью. Кроме того, если к работе готовы несколько заданий, процессор должен решить, какое из них следует запустить, для чего необходим некоторый алгоритм планирования. Эти концепции подробно обсуждаются в следующих главах данной книги.

Использование многозадачности в пакетной обработке может привести к существенному повышению эффективности работы. Однако для многих заданий желательно обеспечить такой режим, в котором пользователь мог бы непосредственно взаимодействовать с компьютером, то есть работать в интерактивном режиме.

Многозадачность не только позволяет процессору одновременно обрабатывать несколько заданий в пакетном режиме, но может быть использована и для обработки нескольких интерактивных заданий. Такую организацию называют ***разделением времени***, потому что процессорное время распределяется между различными пользователями. В системе разделения времени несколько пользователей одновременно получают доступ к

системе с помощью терминалов, а операционная система чередует исполнение программ каждого пользователя через малые промежутки времени. Таким образом, если нужно одновременно обслужить ***n*** пользователей, каждому из них предоставляется лишь ***1/n*** часть машинного времени, не считая затрат на работу операционной системы. Однако, принимая во внимание относительно медленную реакцию человека, время отклика на компьютере с хорошо настроенной системой будет сравнимо со временем реакции пользователя.

Как пакетная обработка, так и разделение времени используют многозадачность. Основное отличие этих подходов состоит в том, что целью пакетной многозадачности является повышение эффективности использования процессора, а целью систем разделения времени – обеспечение интерактивного режима работы пользователей. Кроме того, эти системы управляются по-разному. В пакетных многозадачных системах управление осуществляется с помощью языка управления заданиями, а в системах разделения времени – с помощью команд, вводимых вручную с терминала.

В первых системах разделения времени работа велась следующим образом. Программа всегда загружалась так, что ее начало находилось в ячейке с определенным номером, что упрощало управление, как монитором, так и памятью. Программа при этом была скомпилирована в абсолютных адресах. Приблизительно через каждые 0.5 секунды системные часы генерировали прерывание. При каждом прерывании таймера управление передавалось операционной системе, и процессор мог перейти в распоряжение другого пользователя. Таким образом, данные текущего пользователя через регулярные интервалы времени выгружались, а вместо них загружались другие. Перед считыванием программы и данных нового пользователя программа и данные предыдущего пользователя записывались на диск для сохранения до дальнейшего выполнения. Впоследствии, когда очередь этого пользователя наступит снова, код и данные его программы будут восстановлены в основной памяти. Чтобы уменьшить обмен с диском, содержимое памяти, занимаемое данными пользователя, записывается на него лишь в том случае, если для загрузки новой программы не хватает места.

С появлением разделения времени и многозадачности перед создателями операционных систем появилось несколько проблем. Если в памяти находится несколько заданий, их нужно защищать друг от друга, иначе одно задание может, например, изменить данные другого задания. Если в интерактивном режиме работают несколько пользователей, то файловая система должна быть защищена, чтобы к каждому конкретному файлу доступ был только у определенных пользователей. Нужно разрешать конфликтные ситуации, возникающие при работе с различными ресурсами, например с принтером и устройствами хранения данных.

Одним из основополагающих понятий, помогающих понять структуру операционных систем, является понятие процесса. Этот термин впервые был применен в 60-х годах 20-го века разработчиками операционной системы Multics – предшественницы системы UNIX, и с тех пор широко используется. Есть много определений термина процесс, в том числе:

- выполняющаяся программа;
- экземпляр программы, выполняющейся на компьютере;
- объект, который можно идентифицировать и выполнять на процессоре;
- единица активности, которую можно охарактеризовать единой цепочкой последовательных действий, текущим состоянием и связанным с ней набором системных ресурсов.

В процессе развития компьютерных систем при решении проблем, связанных с распределением времени и синхронизацией, был сделан определенный вклад в развитие концепции процесса. Были разработаны системы групповой обработки нескольких программ, разделения времени и транзакций в реальном времени. Как вы уже знаете, многозадачный режим дает возможность процессору и устройствам ввода-вывода работать одновременно, повышая тем самым эффективность использования компьютерной системы.

При одновременной обработке многих заданий, каждое из которых включает в себя длинную последовательность действий, нельзя проанализировать все возможные комбинации последовательностей событий. Ввиду отсутствия систематических средств обеспечения координации и взаимодействия разных видов деятельности систем, используются специальные методы, основанные на представлении о той среде, работу которой должна контролировать операционная система. При этом возникают следующие проблемы.

- Часто случается так, что программа должна приостановить свою работу и ожидать наступления какого-то события в системе. Например, программа, которая начала операцию ввода-вывода, не сможет продолжать работу, пока в буфере не будут доступны необходимые ей данные. В этом случае требуется передача сигнала от какой-то другой программы. Недостаточная надежность сигнального механизма может привести к тому, что сигнал будет потерян или что будет получено два таких сигнала.

- Часто один и тот же совместно используемый ресурс одновременно пытаются использовать несколько пользователей или несколько программ. Если этот доступ не контролируется должным образом, возможно возникновение ошибок. Для корректной работы требуется некоторый механизм взаимного исключения, позволяющий в каждый момент времени выполнять операцию только одной программе. Правильность реализации такого взаимного исключения при всех возможных последовательностях событий крайне трудно проверить.

- Результат работы каждой программы обычно должен зависеть только от ее ввода и не должен зависеть от работы других программ, выполняющихся в этой же системе. Однако в условиях совместного использования памяти и процессора программы могут влиять на работу друг друга, переписывая общие области памяти непредсказуемым образом. При этом результат работы программ может зависеть от порядка, в котором они выполняются.

- Возможны ситуации, в которых две или большее число программ зависают, ожидая действий друг друга. Например, двум программам может понадобиться, чтобы устройства ввода-вывода выполнили некоторую операцию, например, копирование с диска на магнитную ленту. Одна из этих программ осуществляет управление одним из устройств, а другая – другим устройством. Каждая из них ждет, пока другая программа освободит нужный ресурс. Выйти из такой тупиковой ситуации может помочь система распределения ресурсов.

Для решения перечисленных проблем нужен метод, основанный на слежении за различными выполняющимися процессором программами и управлении ими. В основе такого метода лежит концепция процесса. Процесс условно можно разделить на три компонента.

- Выполняющаяся программа.
- Данные, нужные для ее работы.
- Контекст или состояние программы.

Последний элемент является очень важным. Состояние процесса включает в себя всю информацию, нужную операционной системе для управления процессом, и процессору – для его выполнения. Данные, характеризующие это состояние, включают в себя содержимое различных регистров процессора, таких, как программный счетчик и регистры данных. Сюда же входит информация, использующаяся операционной системой, такая, как приоритет процесса и сведения о том, находится ли данный процесс в состоянии ожидания какого-то события, связанного с вводом-выводом. Подробно о контексте процесса будет рассказано в третьей главе данной книги.

В современных операционных системах процесс реализуется в виде структуры данных. Он может выполняться или находиться в состоянии ожидания. Состояние процесса в каждый момент времени заносится в специально отведенную область данных. Использование структуры позволяет развивать мощные методы координации и взаимодействия процессов. В рамках операционной системы на основе данных о состоянии процесса

путем их расширения и добавления в них дополнительной информации о процессе можно разрабатывать новые возможности операционных систем.

Операционная система должна выполнять изоляцию процессов, то есть следить за тем, чтобы ни один из независимых процессов не смог изменить содержимое памяти, отведенное другому процессу, и наоборот. Программы должны динамически размещаться в памяти в соответствии с определенными требованиями. Распределение памяти должно быть прозрачным для программиста. Таким образом, программист будет избавлен от необходимости следить за ограничениями, связанными с конечностью памяти, а операционная система повышает эффективность работы вычислительной системы, выделяя заданиям только тот объем памяти, который им необходим. При совместном использовании памяти на каждом ее иерархическом уровне есть вероятность, что одна программа обратится к пространству памяти другой программы. Такая возможность может понадобиться, если она заложена в принцип работы данного приложения. С другой стороны, это угроза целостности программ и самой операционной системы. Операционная система должна следить за тем, каким образом различные пользователи могут осуществлять доступ к различным областям памяти. Обычно операционные системы выполняют эти требования с помощью средств виртуальной памяти.

С ростом популярности систем разделения времени, а в последствие, с возникновением компьютерных сетей, возникла проблема защиты информации. В зависимости от обстоятельств, природа угрозы, нависшей над операционной системой, может быть самой разнообразной. Однако в компьютеры и операционные системы могут быть встроены некоторые инструменты общего назначения, поддерживающие различные механизмы защиты и обеспечивающие безопасность. Большую часть задач по обеспечению безопасности и защиты информации можно условно разбить на три категории.

- Контроль над доступом, который связан с регулированием доступа пользователя к системе в целом, к ее подсистемам и данным, а также к различным ресурсам и объектам системы.

- Контроль над перемещением информации с помощью регулирования потока данных внутри системы и при их доставке пользователю.

- Сертификация, которая должна гарантировать, что механизмы доступа и перемещения данных работают в соответствии со своими спецификациями и обеспечивают проводимую политику защиты и безопасности.

Одной из важных задач операционной системы является управление имеющимися в ее распоряжении ресурсами, а также их распределение между разными активными процессами. При разработке стратегии распределения ресурсов необходимо принимать во внимание следующие факторы.

- Обычно нужно, чтобы всем процессам, претендующим на какой-то определенный ресурс, предоставлялся к нему одинаковый доступ. В особенности это касается заданий, принадлежащих к одному и тому же классу, т.е. заданий с аналогичными требованиями к ресурсам.

- С другой стороны, может понадобиться, чтобы операционная система по-разному относилась к заданиям различного класса, имеющим различные запросы. Нужно попытаться сделать так, чтобы операционная система выполняла распределение ресурсов в соответствии с целым набором требований. Операционная система должна действовать в зависимости от обстоятельств. Например, если какой-то процесс ожидает доступа к устройству ввода-вывода, операционная система может спланировать выполнение этого процесса так, чтобы как можно скорее освободить устройство для дальнейшего использования другими процессами.

- Операционная система должна повышать пропускную способность системы, сводить к минимуму время ее отклика и, если она работает в системе разделения времени, обслуживать максимально возможное количество пользователей.

Легко заметить, что в общем случае, перечисленные требования противоречат друг другу. Проблемой в разработке операционных систем является поиск нужного соотношения в каждой конкретной ситуации.

Задача управления ресурсами и их распределения является типичной задачей системы массового обслуживания, поэтому здесь широко применяется соответствующий аппарат и методы проектирования. Это позволяет моделировать активность системы, что позволяет следить за ее производительностью и вносить коррективы в ее работу.

С добавлением в операционные системы все новых функций, а также с ростом возможностей управляемого операционными системами аппаратного обеспечения и его разнообразия возрастает степень их сложности. Так, современная система UNIX по своей сложности намного превосходит свой почти игрушечный первоначальный вариант, разработанный несколькими талантливыми программистами в начале 70-х годов 20-го века. То же самое произошло с простой системой MS-DOS, со временем переросшей в сложные и мощные операционные системы OS/2 и Windows XP.

Увеличение размера полнофункциональных операционных систем и сложности выполняемых ими задач стало причиной возникновения трех широко распространенных проблем. Во-первых, операционные системы доходят до пользователей с хроническим опозданием. Это касается как выпуска новых операционных систем, так и обновления уже существующих. Во-вторых, в системах появляются скрытые ошибки, которые начинают проявлять себя в рабочих условиях и требуют исправления и доработки системы. В-третьих, рост производительности зачастую происходит не так быстро, как планируется.

Программное обеспечение должно состоять из модулей, что упростит организацию процесса его разработки и облегчит выявление и устранение ошибок. Модули по отношению друг к другу должны иметь тщательно разработанные и максимально простые интерфейсы, что также облегчит задачи программиста. Кроме того, меньше усилий потребует эволюция такой системы. Если взаимодействие модулей друг с другом происходит по простым и четким правилам, изменение любого модуля окажет минимальное влияние на остальные.

Однако оказывается, что для больших операционных систем, код которых состоит из десятков миллионов строк, принцип модульного программирования сам по себе не избавляет от всех проблем. По этой причине возросла популярность концепции уровней иерархии, а также информационной абстракции.

В иерархической структуре современной операционной системы различные функции находятся на разных уровнях в зависимости от их сложности, временных характеристик и степени абстракции. Систему можно рассматривать как набор уровней, каждый из которых выполняет свой ограниченный круг заданий, входящий в комплекс задач операционной системы. Работа компонентов определенного уровня основывается на работе компонентов, находящихся на более низком уровне. Функции более высокого уровня используют примитивы нижнего по отношению к нему уровня. В идеале уровни должны быть определены так, чтобы при изменении одного из них не изменялись остальные.

В каждой отдельно взятой операционной системе перечисленные принципы применяются по-разному, но для учебных целей можно представить обобщенную модель иерархической операционной системы.

- В первый уровень абстракции входят электронные схемы. Объектами данного уровня являются регистры, ячейки памяти и логические элементы, составляющие схему компьютера. Над этими объектами выполняются различные действия, такие, как очистка содержимого регистра или считывание ячейки памяти.

- Второй уровень содержит набор команд процессора. В число операций, выполняемых на этом уровне, входят операции, которые допускаются набором команд машинного языка, например сложение, вычитание, загрузка значения из регистра или сохранение в нем.

- Третий уровень абстракции вводит понятие программы, а также операции вызова и возврата.

- Четвертый уровень определяет прерывания, которые заставляют процессор сохранить текущий контекст и выполнить подпрограмму обработки прерывания.

Следует заметить, что на самом деле первые четыре уровня не являются частями операционной системы, они составляют аппаратное обеспечение процессора. Однако на этих уровнях уже появляются некоторые элементы операционной системы, такие, как программы обработки прерываний.

- На пятом уровне абстракции вводится понятие процесса, под которым подразумевается работающая программа. В число фундаментальных требований к операционной системе, способной поддерживать одновременную работу нескольких процессов, входят способность приостанавливать процессы и возобновлять их выполнение. Для этого необходимо сохранять содержимое регистров аппаратного обеспечения, чтобы можно было переключаться с одного процесса на другой. Кроме того, если процессы должны взаимодействовать между собой, необходим механизм их синхронизации. Одной из важнейших концепций устройства операционных систем является семафор — простейший способ передачи сигналов, который рассмотрен в следующих главах данной книги.

- Шестой уровень определяет компоненты операционной системы, которые взаимодействуют со вспомогательными запоминающими устройствами компьютера. На этом уровне происходит периферийными устройствами и собственно передача данных. Отметим, что для планирования работы и уведомления процесса о завершении запрошенной операции шестой уровень обычно использует компоненты пятого уровня абстракции.

- Седьмой уровень определяет логическое адресное пространство процессов. Организуется виртуальное адресное пространство в виде блоков, которые могут перемещаться между основной памятью и вспомогательным запоминающим устройством.

- Восьмой уровень иерархии отвечает за обмен информацией и сообщениями между процессами. На этом уровне происходит более богатый обмен информацией, чем на пятом уровне, обеспечивающем первичный сигнальный механизм для синхронизации процессов. Одним из наиболее мощных инструментов подобного типа является канал передачи данных между процессами.

На девятом уровне обеспечивается долгосрочное хранение файлов. Данные, хранящиеся на вспомогательном запоминающем устройстве, рассматриваются как абстрактные объекты переменной длины, в противоположность интерпретации внешней памяти на шестом уровне, как набору дорожек, секторов и блоков данных.

- На десятом уровне абстракции предоставляется доступ к внешним устройствам с помощью стандартных интерфейсов.

- Одиннадцатый уровень поддерживает связь между внешними и внутренними идентификаторами системных ресурсов и объектов. Внешний идентификатор представляет собой имя, а внутренний идентификатор – это адрес, используемый системой для определения объекта и управления им. Эта связь поддерживается с помощью системных таблиц, которые включают в себя не только взаимное отображение внешних и внутренних идентификаторов, но и такие характеристики, как, например, права доступа.

На двенадцатом уровне предоставляется полнофункциональные средства поддержки процессов. Возможности этого уровня намного превосходят возможности пятого уровня, на котором поддерживается только содержимое регистров процессора, имеющее отношение к процессу, и логика диспетчеризации процессов. На двенадцатом уровне эта информация используется для упорядоченного управления процессами. Сюда же относится и виртуальное адресное пространство процессов, список объектов и процессов, с которыми оно может взаимодействовать, и правила, ограничивающие это взаимодействие.

Тринадцатый уровень иерархии является самым верхним и обеспечивает взаимодействие операционной системы с пользователем. Этот уровень часто называется ***оболочкой*** (shell), так как он отделяет пользователя от деталей внутреннего устройства

операционной системы и представляет ее пользователю как набор сервисов. Оболочка принимает команды пользователя, интерпретирует их, создает необходимые процессы и управляет ими. На этом уровне, например, может быть реализован графический интерфейс, предоставляющий пользователю возможность выбора команды с помощью меню и отображающий результаты работы на экране.

Описанная гипотетическая модель операционной системы дает представление об ее структуре и может, служить прототипом при реализации конкретной операционной системы.

Разработка технического задания на создание программ

Стандарт ГОСТ 19.201-78. устанавливает порядок построения и оформления технического задания на разработку программы или программного изделия для вычислительных машин, комплексов и систем независимо от их назначения и области применения.

Общие положения

1 Техническое задание оформляют в соответствии с ГОСТ 19.106-78 на листах формата А4 и А3 по ГОСТ 2.301-68, как правило, без заполнения полей листа. Номера листов (страниц) проставляют в верхней части листа над текстом.

2 Лист утверждения и титульный лист оформляют в соответствии с ГОСТ 19.104-78. Информационную часть (аннотацию и содержание), лист регистрации изменений допускается в документ не включать.

3 Для внесения изменений и дополнений в техническое задание на последующих стадиях разработки программы или программного изделия выпускают дополнение к нему. Согласование и утверждение дополнения к техническому заданию проводят в том же порядке, который установлен для технического задания.

4 Техническое задание должно содержать следующие разделы:

•название программы и область применения;

•основание для разработки;

•назначение разработки;

•технические требования к программе или программному изделию;

•технико-экономические показатели;

•стадии и этапы разработки:

•порядок контроля и приемки;

•приложения.

В зависимости от особенностей программы или программного изделия допускается уточнять содержание разделов, вводить новые разделы или объединять отдельные из них.

5 Содержание разделов

5.1. В разделе «Наименование и область применения» указывают наименование, краткую характеристику области применения программы или программного изделия и объекта, в котором используют программу или программное изделие.

5.2. В разделе «Основание для разработки» должны быть указаны:

•документ (документы), на основании которых ведется разработка,

•организация, утвердившая этот документ, и дата его утверждения;

•наименование и (или) условное обозначение темы разработки.

5.3. В разделе «Назначение разработки» должно быть указано функциональное и эксплуатационное назначение программы или программною изделия.

5.4. Раздел «Технические требования к программе или программному изделию» должен содержать следующие подразделы:

•требования к функциональным характеристикам;

•требования к надежности;

•условия эксплуатации;

•требования к составу и параметрам технических средств;

•требования к информационной и программной совместимости;

•требования к маркировке и упаковке;

•требования к транспортированию и хранению;

•специальные требования.

5.5. В подразделе «Требования к функциональным характеристикам» должны быть указаны требования к составу выполняемых функций, организации входных и выходных данных, временным характеристикам и т.п.

5.6. В подразделе «Требования к надежности» должны быть указаны требования к обеспечению надежного функционирования (обеспечение устойчивого функционирования, контроль входной и выходной информации, время восстановления после отказа и т. п.).

5.7. В подразделе «Условия эксплуатации» должны быть указаны условия эксплуатации (температура окружающего воздуха, относительная влажность и т. п. для выбранных типов носителей данных), при которых должны обеспечиваться заданные характеристики, а также вид обслуживания, необходимое количество и квалификация персонала.

5.8. В подразделе «Требования к составу и параметрам технических средств» указывают необходимый состав технических средств с указанием их технических характеристик.

5.9. В подразделе «Требования к информационной и программной совместимости» должны быть указаны требования к информационным структурам на входе и выходе и методам решения, исходным кодам, языкам программирования. При необходимости должна обеспечиваться защита информации и программ.

5.10. В подразделе «Требования к маркировке и упаковке» в общем случае указывают требования к маркировке программного изделия, варианты и способы упаковки.

5.11. В подразделе «Требования к транспортированию и хранению» должны быть указаны для программного изделия условия транспортирования, места хранения, условия хранения, условия складирования, сроки хранения в различных условиях.

5.12. В разделе «Технико-экономические показатели» должны быть указаны: ориентировочная экономическая эффективность предполагаемая годовая потребность, экономические преимущества разработки по сравнению с лучшими отечественными и зарубежными образцами или аналогами.

5.13. В разделе «Стадии и этапы разработки» устанавливают необходимые стадии разработки, этапы и содержание работ (перечень программных документов, которые должны быть разработаны, согласованы и утверждены), а также, как правило, сроки разработки и определяют исполнителей.

5.14. В разделе «Порядок контроля и приемки» должны быть указаны виды испытаний и общие требования к приемке работы.

5.15. В приложениях к техническому заданию при необходимости приводят:

• перечень научно- исследовательских и других работ, обосновывающих разработку;

• схемы алгоритмов, таблицы, описания, обоснования, расчеты и другие документы, которые могут быть использованы при разработке;

• другие источники разработки.

Пример разработки технического задания.

1. Введение.

Работа выполняется в рамках проекта «Автоматизированная система оперативно-диспетчерского управления электро-, теплоснабжением корпусов предприятия».

2 Основание для разработки

1. Основанием для данной работы служит договор № 12-к от 10 января 2017 г.

2. Наименование работы:

«Модуль автоматизированной системы оперативно-диспетчерското управления теплоснабжением корпусов предприятия».
3. Исполнители: ОАО « ProgramLabs».
4. Соисполнители: нет.

3 Назначение разработки

Создание модуля для контроля и оперативной корректировки состояния основных параметров теплообеспечения корпусов предприятия «Темп».

4 Технические требования

4.1. Требования к функциональным характеристикам.

4.1.1. Состав выполняемых функций. Разрабатываемое программное обеспечение должно обеспечивать:

• сбор и анализ информации о расходовании тепла, горячей и холодной воды по данным теплосчетчиков SA-94 на всех тепловых выходах;

• сбор и анализ информации с устройств управления системами воздушного отопления и кондиционирования типа РТ1;

• предварительный анализ информации на предмет нахождения параметров в допустимых пределах и сигнализирование при выходе параметров за пределы допуска;

• выдачу рекомендаций по дальнейшей работе;

• отображение текущего состояния набора параметров – циклически постоянно (режим

работы круглосуточный), при сохранении периодичности контроля прочих параметров;

• визуализацию информации по расходу теплоносителя:

- текущей (аналогично показаниям счетчиков);

- с накоплением за прошедшие сутки, неделю, месяц - в виде почасового графика для информации за сутки и неделю;

- суточный расход - для информации за месяц.

Для устройств управления приточной вентиляцией текущая информация должна содержать номер приточной системы и все параметры, выдаваемые на собственный индикатор.

По отдельному запросу осуществляются внутренние настройки. В конце отчетного периода система должна архивировать данные.

4.1.2. Организация входных и выходных данных.

Исходные данные в систему поступают в виде значений с датчиков, установленных в помещениях предприятия. Эти значения отображаются на компьютере диспетчера. После анализа поступившей информации оператор диспетчерского пункта устанавливает необходимые параметры для устройств, регулирующих отопление и вентиляцию в помещениях.

Возможна также автоматическая установка некоторых параметров для устройств регулирования.

Основной режим использования системы – ежедневная работа.

4.2. Требования к надежности.

Для обеспечения надежности необходимо проверять корректность получаемых данных с датчиков.

4.3. Условия эксплуатации и требования к составу и параметрам технических средств.

Для работы системы должен быть выделен ответственный оператор. Требования к составу и параметрам технических средств уточняются на этапе эскизного проектирования системы.

4.4. Требования к информационной и программной совместимости.

Программа должна работать на платформах Windows 8.

4.5. Требования к транспортировке и хранению. Программа поставляется на лазерном носителе информации.

Программная документация поставляется в электронном и печатном виде.

4.6. Специальные требования.

Программное обеспечение должно иметь дружественный интерфейс, рассчитанный на пользователя (в плане компьютерной грамотности) средней квалификации. Ввиду объемности проекта задачи предполагается решать поэтапно. При этом модули ПО, созданные в разнос время, должны предполагать возможность наращивания системы и быть совместимы друг с другом. Поэтому документация на принятое эксплуатационное ПО должна содержать полную информацию, необходимую для работы программистов с ним.

Язык программирования – по выбору исполнителя, должен обеспечивать возможность интеграции программного обеспечения с некоторыми видами периферийного оборудования.

5 Требования к программной документации

Основными документами, регламентирующими разработку будущих программ, должны быть документы Единой Системы Программной Документации (ЕСПД);

руководство пользователя,

руководство администратора, описание применения.

6 Технико-экономические показатели

Эффективность системы определяется удобством использования системы для контроля и управления основными параметрами теплообеспечения помещений предприятия, а также экономической выгодой, полученной от внедрения аппаратно-программного комплекса.

7 Порядок контроля и приемки

После передачи Исполнителем отдельного функционального модуля программы Заказчику, последний имеет право тестировать модуль в течение 7 дней. После тестирования Заказчик должен принять работу по данному этапу или в письменном виде изложить причину отказа от принятия. В случае обоснованного отказа Исполнитель обязуется доработать модуль.

8 Календарный план работ.

Наименование этапа	Сроки этапа	Результат выполнения этапа
1 Изучение предметной области	01.02.2017 – 28.02.2017	Предложения по разработке программного обеспечения Проектирование системы. Выбор средства реализации. Разработка системы. Акт сдачи-приемки предложений по реализации системы.
2 Разработка программного модуля по сбору и анализу информации со счётчиков и устройств управления.	01.03.2017 – 31.08.2017	Завершенный программный комплекс. Внедрение системы на одном из корпусов предприятия.
3 Тестирование и отладка модуля.	01.09.2017 – 30.11.2017	Готовая система контроля теплообеспечения.
4. Внедрение автоматизированной системы.	01.12.2017 – 30.12.2017	Готовая система контроля системы во всех корпусах, установленная в диспетчерском пункте. Программная документация. Акт сдачи- приёма работ.

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ К ЛАБОРАТОРНЫМ ЗАНЯТИЯМ

Лабораторная работа 1. Назначение и содержание соглашения о требованиях

Цель работы: научиться подготавливать программную документацию по организации коллективной разработки ПО.

В соглашении о требованиях должно содержаться письменное изложение того, что будет сделано и что не будет выполняться при выпуске программного обеспечения.

Документ «Соглашение о требованиях» является основным средством управления разработкой программного обеспечения или генеральным планом его разработки.

Все участники разработки программного обеспечения должны выполнять то, что установлено в документе «Соглашение о требованиях» или запрашивать и получать разрешение на его изменение.

Предполагается, что все утверждения, включенные в соглашение о требованиях, являются требованиями, если они не определены как цели.

Каждый документ «Соглашение о требованиях» должен точно соответствовать установленной форме. Тогда каждый раздел можно будет найти в одном и том же месте аналогичного документа любой разработки программного обеспечения. В документ целесообразно включить заголовки всех предусмотренных разделов, если только специально не оговариваются условия, при которых какой-либо раздел может быть опущен. Тогда при рассмотрении документа будет решаться вопрос, действительно ли такие разделы нужны.

Соглашения о требованиях пишутся на естественном языке в терминах понятных и пользователю и разработчику программного обеспечения. Стороны должны четко представлять каждое требование.

Следует напомнить, что пользователь несет ответственность за проверку требований на полноту и точность, а разработчик – за проверку их на осуществимость и понятность.

Задание. Разработать соглашение о требованиях на разрабатываемый программный продукт.

Лабораторная работа 2. Спецификация программного обеспечения

Цель работы: получение практических навыков в анализе предметной области, выявлении требований к программному обеспечению, представление программного обеспечения в виде модулей.

На этапе определения спецификаций осуществляется точное описание функций, реализуемых ЭВМ, а также задаются структуры входных и выходных данных, методы и средства их размещения. Определяются алгоритмы обработки данных.

Центральным вопросом определения спецификаций является проблема организации базы данных. При этом решается комплекс вопросов, имеющих отношение к структуре файлов, организации доступа к ним, модификации и удаления.

В случае, когда новая система создается на основе существующих, составной частью спецификаций является схема (алгоритм) приведения существующей базы данных к новому формату. Такое преобразование может потребовать разработку специальной программы, которая становится ненужной после ее первого и единственного использования.

Все эти вопросы должны быть отражены в функциональных спецификациях, которые представляют собой документ, являющийся основополагающим в процессе разработки системы, так как содержит конкретное описание последней.

Чем подробней составлены спецификации, тем меньше вероятность возникновения ошибок.

В спецификациях должны быть представлены данные для тестирования элементов системы и системы в целом. Это требование является объективным и обязательным, так как на данном этапе на параметры тестирования не будет оказывать влияние конкретная реализация системы.

Так как функциональные спецификации описывают принятые решения в целом, данный документ можно использовать для начальных оценок временных затрат, числа специалистов и других ресурсов, необходимых для проведения работ.

В общем случае спецификации определяют те функции, которые должна выполнять система, не указывая, каким образом это достигается. Составление подробных алгоритмов на этом этапе преждевременно и может вызвать нежелательные осложнения.

Задание. Результатом выполнения лабораторной работы должен являться оформленный по международному стандарту IEEE 830 документ «Спецификация программного обеспечения».

Лабораторная работа 3. Архитектурный проект

Цель работы: получение практических навыков проектирования программного обеспечения с использованием унифицированного языка моделирования.

Задание. Результатом выполнения лабораторной работы должен являться документ «Архитектурный проект», включающий в себя следующие UML-диаграммы с подробным описанием:

- вариантов использования;
- классов;
- последовательности;
- деятельности;
- состояний;
- компонентов.

Лабораторная работа 4. Реализация программного обеспечения

Цель работы: реализация программного обеспечения согласно спецификации и архитектурному проекту на программное обеспечение. Получение навыков разработки программного обеспечения в команде с использованием системы управления версиями.

Задание. Результатом выполнения лабораторной работы является реализованное программное обеспечение, а также технический отчет, описывающий процесс разработки.

Лабораторная работа 5. Технология написания тестов

Цель работы: научиться определять классы эквивалентности в спецификациях и подготавливать необходимый набор тестов по ним.

Этап тестирования обычно в финансовых затратах составляет половину расходов на создание системы. Плохо спланированное тестирование приводит к существенному увеличению сроков разработки системы и является основной причиной срывов графиков разработки.

В процессе тестирования используются данные, характерные для системы в рабочем состоянии, т. е. данные для тестирования выбираются случайным образом. План проведения испытаний должен быть составлен заранее, обычно на этапе проектирования.

Тестирование подразумевает три стадии:

1. автономное;
2. комплексное;

3. системное.

При автономном тестировании модуль проверяется с помощью данных, подготовленных программистом. При этом программная среда модуля имитируется с помощью программ управления тестированием, содержащих фиктивные программы вместо реальных подпрограмм, к которым имеется обращение из данного модуля (заглушки). Подобную процедуру называют программным тестированием, а программу тестирования – UUT (тестирующей программой). Модуль, прошедший автономное тестирование, подвергается комплексному тестированию.

В процессе комплексного тестирования проводится совместная проверка групп программных компонентов. В результате имеем полностью проверенную систему. На данном этапе тестирование обнаруживает ошибки, пропущенные на стадии автономного тестирования. Исправление этих ошибок может составлять до четверти от общих затрат.
Системное (или оценочное) тестирование – это завершающая стадия проверки системы, т. е. проверка системы в целом с помощью независимых тестов. Независимость тестов является главным требованием. Обычно Заказчик на стадии приемки работ настаивает на проведении собственного системного тестирования. Для случая, когда сравниваются характеристики нескольких систем (имеется альтернативная разработка), такая процедура известна как сравнительное тестирование.

Технология тестирования, классы эквивалентности.

Одним из способов изучения данного вопроса является исследование стратегии тестирования, называемой стратегией черного ящика, тестированием с управлением по данным или тестированием с управлением по входу-выходу. При использовании этой стратегии программа рассматривается как черный ящик.

Иными словами, такое тестирование имеет целью выяснение обстоятельств, в которых поведение программы не соответствует ее спецификации. При таком подходе обнаружение всех ошибок в программе является критерием исчерпывающего входного тестирования. Последнее может быть достигнуто, если в качестве тестовых наборов использовать все возможные наборы входных данных.

Разработка тестов методом эквивалентного разбиения осуществляется в два этапа:
1) выделение классов эквивалентности;
2) построение тестов.

Классы эквивалентности выделяются путем выбора каждого входного условия (обычно это предложение или фраза в спецификации) и разбиением его на две или более группы. Для проведения этой операции используют таблицу, подобную следующей:

Входные условия	Классы эквивалентности
Правильные	Неправильные

Различают два типа классов эквивалентности: правильные классы эквивалентности, представляющие правильные входные данные программы, и неправильные классы эквивалентности, представляющие все другие возможные состояния условий (т. е. ошибочные входные значения). Таким образом, придерживаются одного из принципов необходимости сосредоточивать внимание на неправильных или неожиданных условиях.

Задание. Идентифицировать входные условия и по ним определить классы эквивалентности, построить соответствующие тесты. В заключении, дать рекомендации по устранению ошибок, выявленных в результате тестирования.

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ К КУРСОВОМУ ПРОЕКТИРОВАНИЮ

Курсовое проектирование включает в себя следующие этапы:

1) анализ предметной области;

2) выбор модели жизненного цикла, используемого при реализации программного обеспечения (ГОСТ Р ИСО/МЭК 12207-2010 Процессы жизненного цикла программных средств);

3) определение требований к программному обеспечению;

4) разработка технического задания на разработку программного обеспечения согласно (ГОСТ 19.201-78 Техническое задание, требования к содержанию и оформлению)

5) проектирование программного обеспечения;

6) реализация проекта согласно технической документации;

7) внесение изменений в техническую документацию, если это необходимо.

8) выполнение тестирования и отладки, описание этих процессов;

9) разработка документации к ПО (руководство оператора и системного администратора);

10) оформление пояснительной записки.

Пояснительная записка должна включать следующие разделы:

- титульный лист;
- содержание;
- введение;
- анализ предметной области;
- обоснование выбора модели жизненного цикла;
- описание реализации (руководство программиста);
- заключение;
- используемые источники;
- приложение 1. Документ «Спецификация требований ПО»;
- приложение 2. Документ «Архитектурный проект».
- приложение 3. Руководство оператора;
- приложение 4. Руководство системного администратора.

11) оформление презентации и защита программного обеспечения перед комиссией.

Презентация должна отражать основные этапы разработки и результат работы программы.

Задание на курсовой проект

1. Составить общее описание программного обеспечения. Сформулировать основные функции ПО (детально функциональные требования будут сформированы в следующих пунктах). Так же необходимо определить взаимодействия ПО с другими системами (если это необходимо). Выявить классы и характеристики пользователей, которые будут использовать ПО.

2. Выбрать и обосновать выбор методологии (модели) жизненного цикла ПО, которую следует использовать для разработки ПО согласно вашему варианту.

3. Разработать документ «Спецификация требований ПО», разбив ПО на функциональные блоки (модули) и определив функциональные и не функциональные требования на каждый модуль. Также сформулировать требования к внешнему программному и аппаратному обеспечению. Структура документа:

1Введение

1.1. Цели

1.2. Соглашения о терминах
1.3. Предполагаемая аудитория и последовательность восприятия
1.4. Масштаб проекта
2. Общее описание
2.1. Видение продукта
2.2. Функциональность продукта
2.3. Классы и характеристики пользователей
2.4. Среда функционирования продукта (операционная среда)
2.5. Рамки, ограничения, правила и стандарты
2.6. Документация для пользователей
2.7. Допущения и зависимости
3. Функциональность системы
3.1. Функциональный блок X (таких блоков может быть несколько)
3.1.1. Описание и приоритет
3.1.2. Причинно-следственные связи, алгоритмы (движение процессов)
3.1.3. Функциональные требования
4. Нефункциональные требования
4.1. Требования к производительности
4.2. Требования к сохранности (данных)
4.3. Критерии качества программного обеспечения
4.4. Требования к безопасности системы

4. Разработать документ «Архитектурный проект», включающий в себя необходимый набор из следующих UML-диаграмм с подробным описанием:

- вариантов использования;
- классов;
- последовательности;
- деятельности;
- состояний;
- компонентов.

5. В документе привести описание и прототип пользовательского интерфейса будущего программного обеспечения. Прототип пользовательского интерфейса должен включать в себя шаблон основного окна ПО и 3-ех второстепенных (например, окно настроек, окно управления пользователями и другие).

6. Осуществить проектирование и реализацию программного обеспечения модели, согласно вашему варианту.

Варианты задания на курсовое проектирование

Темы курсовых проектов ежегодно меняются в зависимости от предметной области, рассматриваемой студентом в рамках выполнения диссертационного исследования. Примерами тем курсовых проектов являются:

1. Разработка автоматизированной справочной геоинформационной системы
2. Разработка Windows-приложения для подбора параметров робастной системы управления существенно нестационарным объектом с использованием генетического алгоритма
3. Разработка web-приложения оценки степени угроз информационной безопасности сайта компании
4. Разработка клиент-серверного Linux-приложения для сбора информации о характеристиках сетевых устройств
5. Разработка программного модуля прогнозирования курса в краткосрочном периоде

6. Разработка программно-аппаратного комплекса для управления моделью движущейся платформы посредством сети Wi-Fi

7. Разработка Android-приложения для контроля выполнения заказов клиентов компании

8. Разработка программного модуля 3D-моделей залов краеведческого музея

9. Разработка программного модуля системы поддержки принятия решений

10. Разработка документно-ориентированной системы управления базой данных распределенной архитектуры

11. Разработка информационной системы моделирования городских транспортных потоков

12. Разработка Windows-приложения расчета коэффициентов автоматизированной систему управления нестационарным объектом с использованием много поточных вычислений

МЕТОДИЧЕСКИЕ РЕКОМЕНДАЦИИ ДЛЯ САМОСТОЯТЕЛЬНОЙ РАБОТЫ СТУДЕНТОВ

Самостоятельная работа по дисциплине включает изучение учебной литературы, разбор конспектов лекций и материалов практических занятий, подготовку к выполнению индивидуальных заданий к лабораторным занятиям, отчетов по их выполнению, выполнение курсового проектирования, а также подготовку к тестированию и непосредственно к экзамену.

Задания к лабораторным занятиям выдаются заранее, как правило, на первом занятие текущего семестра. Для их успешного выполнения необходимо предварительное освоение теоретического материала и разбор, приведенных примеров программ, проработка алгоритма решения собственных заданий. Для этого можно воспользоваться учебно-методическим обеспечением для самостоятельной работы, указанным в рабочей программе.

Задания к курсовому проектированию также выдается в начале семестра. Выполнение курсового проекта происходит постепенно по указанным этапам в ходе изучения необходимого материала. Во время выполнения курсового проекта необходимо посещать индивидуальные консультации с руководителем для промежуточного контроля и уточнения вопросов, возникающих в ходе его выполнения.

Для подготовки к выполнению лабораторных работ и усвоения теоретического материала студентам рекомендуется самостоятельно организовать по месту проживания дополнительное рабочее место, оборудованное персональным компьютером, подключённым к сети Интернет, и установленным программным обеспечением, необходимым для разработки программ и указанном в рабочей программе. Общие методические рекомендации по составлению программ на языке программирования дополнительные рекомендации по каждой теме представлены в предыдущих разделах.

В отчете по выполнению индивидуального варианта заданий должны содержатся следующие сведения: формулировка задания, входные и выходные данные, текст программы, тестовые (контрольные) значения входных данных и рассчитанные выходные данные.

Итоговый контроль – экзамен проводится на основании перечня вопросов, представленного в рабочей программе. Подготовка к экзамену заключается в изучении и тщательной проработке студентом конспектов по всем видов занятий. При подготовке к экзамену рекомендуется использовать литературу, рекомендованную в рабочей программе, ЭВМ и все теоретические знания, и практические навыки, полученные во время проведения и выполнения всех видов занятий.

Экзамен проводится в виде устного ответа по билету. Экзаменационный билет включает два теоретических вопроса. На экзамен студент предоставляет отчеты по выполнению всех лабораторных заданий. Ответы на поставленные вопросы студент дает после предварительной подготовки. Преподаватель имеет право задать дополнительные вопросы, если ответ дан неполный или затруднительно однозначно оценить ответ.

ЛИТЕРАТУРА

Аллен, Э. Типичные ошибки проектирования [Текст] / Э. Аллен. - СПб. : Питер, 2003 – 224 с.

Брауде, Э. Д. Технология разработки программного обеспечения [Текст] / Э. Д. Брауде. – СПб. : Питер, 2004 – 655 с.

Орлов, С. А. Технологии разработки программного обеспечения [Текст]: Разработка сложных программных систем: Учеб. пособие: Рек. Мин. обр. РФ / С. А. Орлов. – 2-е изд. – СПб. : Питер, 2003 – 474 с.

Пайлок, Д. Управление разработкой программного обеспечения. / Д. Пайлок. – СПб: Питер, 2011

Пилон, Д. Управление разработкой программного обеспечения [Текст] / Д.Пилон, Р. Майлз ; пер. с англ. В. Шрага. - СПб. : Питер, 2011 – 458 с.

Синицын С.В. Основы разработки программного обеспечения на примере языка C [Электронный ресурс] / С.В. Синицын, О.И. Хлытчиев. — 2-е изд. — Электрон. Текстовые данные. — М. : Интернет-Университет Информационных Технологий (ИНТУИТ), 2016 — 211 c. — 2227-8397. — Режим доступа: http://www.iprbookshop.ru/73700.html

Синицын С.В. Верификация программного обеспечения [Электронный ресурс] : учебное пособие / С.В. Синицын, Н.Ю. Налютин. — Электрон. текстовые данные. — Москва, Саратов: Интернет-Университет Информационных Технологий (ИНТУИТ), Вузовское образование, 2017 — 368 c. — 978-5-4487-0074-3. — Режим доступа: http://www.iprbookshop.ru/67396.html

Технология разработки программного обеспечения [Электронный ресурс] : учеб. – метод. пособие / АмГУ, ФМиИ ; сост. Т. А. Галаган. – Благовещенск : Изд-во Амур. гос. ун-та, 2015 – 49 с. http://irbis.amursu.ru/DigitalLibrary/AmurU_Ediiion/6799.pif

Павловская, Т. А. C#. Программирование на языке высокого уровня [Текст] : учеб. : рек. Мин. обр. РФ / Т. А. Павловская. - СПб. : Питер, 2007. - 432 с. : рис. - (Учебник для вузов). - Библиогр.: с. 425 . - Алф. указ.: с. 427 . - ISBN 978-5-91180-174-8 (в пер.) :

СОДЕРЖАНИЕ