1.1. Технология программирования и основные этапы ее развития

Технологией программирования называют совокупность методов и средств, используемых в процессе разработки программного обеспечения. Как любая другая технология, технология программирования представляет собой набор технологических инструкций, включающих:

- указание последовательности выполнения технологических операций;
- перечисление условий, при которых выполняется та или иная операция;
- описания самих операций, где для каждой операции определены исходные данные, результаты, а также инструкции, нормативы, стандарты, критерии и методы оценки и т. п.

Кроме набора операций и их последовательности, технология также определяет способ описания проектируемой системы, точнее модели, используемой на конкретном этапе разработки.

Различают технологии, используемые на конкретных этапах разработки или для решения отдельных задач этих этапов, и технологии, охватывающие несколько этапов или весь процесс разработки. В основе первых, как правило, лежит ограниченно применимый метод, позволяющий решить конкретную задачу. В основе вторых обычно лежит базовый метод или подход, определяющий совокупность методов, используемых на разных этапах разработки, или методологию.

Первый этап - «стихийное» программирование. (до середины 60-х годов XX в.) В этот период практически отсутствовали сформулированные технологии. Первые программы имели простейшую структуру. Они состояли из собственно программы на машинном языке и обрабатываемых ею данных. Появление ассемблеров позволило вместо двоичных или 16-ричных кодов использовать символические имена данных и мнемоники кодов операций. В результате программы стали более «читаемыми». Типичная программа того времени состояла из основной программы, области глобальных данных и набора подпрограмм (в основном библиотечных), выполняющих обработку всех данных или их части.

Второй этап - структурный подход к программированию (60-70-е годы XX в.). Структурный подход к программированию представляет собой совокупность рекомендуемых технологических приемов, охватывающих выполнение всех этапов разработки программного обеспечения. В основе структурного подхода лежит декомпозиция (разбиение на части) сложных систем с целью последующей реализации в виде отдельных небольших подпрограмм. С появлением других принципов декомпозиции (объектного, логического и т. д.) данный способ получил название процедурной декомпозиции.

Третий этап - объектный подход к программированию (с середины 80-х до конца 90-х годов XX в.). Объектно-ориентированное программирование определяется как технология создания сложного программного обеспечения, основанная на представлении программы в виде совокупности объектов, каждый из которых является экземпляром определенного типа {класса), а классы образуют иерархию с наследованием свойств. Взаимодействие программных объектов в такой системе осуществляется путем передачи сообщений. Основным достоинством объектно-ориентированного программирования по сравнению с модульным программированием

является «более естественная» декомпозиция программного обеспечения, которая существенно облегчает его разработку.

Четвертый этап - компонентный подход и CASE-технологии (с середины 90-х годов XX в. до нашего времени). Компонентный подход предполагает построение программного обеспечения из отдельных компонентов - физически отдельно существующих частей программного обеспечения, которые взаимодействуют между собой через стандартизованные двоичные интерфейсы.

Компонентный подход лежит в основе технологий, разработанных на базе COM (Component Object Model - компонентная модель объектов), и технологии создания распределенных приложений CORBA (Common Object Request Broker Architecture - общая архитектура с посредником обработки запросов объектов). Эти технологии используют сходные принципы и различаются лишь особенностями их реализации.

1.2. Проблемы разработки сложных программных систем

Дополнительными факторами, увеличивающими сложность разработки программных систем, являются :

- сложность формального определения требований к программным системам;
- отсутствие удовлетворительных средств описания поведения дискретных систем с большим числом состояний при недетерминированной последовательности входных воздействий;
- коллективная разработка;
- необходимость увеличения степени повторяемости кодов

Сложность определения требований к программным системам. Во-первых, при определении требований необходимо учесть большое количество различных факторов. Во-вторых, разработчики программных систем не являются специалистами в автоматизируемых предметных областях, а специалисты в предметной области, как правило, не могут сформулировать проблему в нужном ракурсе.

Отсутствие удовлетворительных средств формального описания поведения дискретных систем. В процессе создания программных систем используют языки сравнительно низкого уровня.

Коллективная разработка. Из-за больших объемов проектов разработка программного обеспечения ведется коллективом специалистов

Необходимость увеличения степени повторяемости кодов. На сложность разрабатываемого программного продукта влияет и то, что для увеличения производительности труда компании стремятся к созданию библиотек компонентов, которые можно было бы использовать в дальнейших разработках.

1.3. Блочно- иерархический подход к созданию сложных систем

В основе блочно-иерархического подхода лежат декомпозиция и иерархическое упорядочение. Важную роль играют также следующие принципы:

- непротиворечивость - контроль согласованности элементов между собой;
- полнота — контроль на присутствие лишних элементов;
- формализация - строгость методического подхода;
- повторяемость — необходимость выделения одинаковых блоков для удешевления и ускорения разработки

- локальная оптимизация - оптимизация в пределах уровня иерархии.

Совокупность языков моделей, постановок задач, методов описаний некоторого иерархического уровня принято называть уровнем проектирования.

Различные взгляды на объект проектирования принято называть аспектами проектирования.

Помимо того, что использование блочно-иерархического подхода делает возможным создание сложных систем, он также:

- упрощает проверку работоспособности, как системы в целом, так и отдельных блоков;
- обеспечивает возможность модернизации систем, например, замены ненадежных блоков с сохранением их интерфейсов.

Необходимо отметить, что использование блочно-иерархического подхода применительно к программным системам стало возможным только после конкретизации общих положений подхода и внесения некоторых изменений в процесс проектирования.

1.4. Жизненный цикл и этапы разработки программного обеспечения

Жизненным циклом программного обеспечения называют период от момента появления идеи создания некоторого программного обеспечения до момента завершения его поддержки фирмой-разработчиком или фирмой, выполнявшей сопровождение. Процесс жизненного цикла определяется как совокупность взаимосвязанных действий, преобразующих некоторые входные данные в выходные. Каждый процесс характеризуется определенными задачами и методами их решения, а также исходными данными и результатами. По стандарту процесс разработки включает следующие действия:

- подготовительную работу
- анализ требований к системе
- проектирование архитектуры
- анализ требований к ПО
- проектирование архитектуры ПО
- детальное проектирование ПО
- кодирование и тестирование ПО
- интеграцию ПО
- квалификационное тестирование ПО
- интеграцию системы
- квалификационное тестирование системы
- установку ПО
- приемку ПО

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

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

1.5. Эволюция моделей жизненного цикла программного обеспечения

Каскадная модель. Первоначально (1970-1985 годы) была предложена и использовалась каскадная схема разработки программного обеспечения, которая предполагала, что переход на следующую стадию осуществляется после того, как полностью будут завершены проектные операции предыдущей стадии и получены все исходные данные для следующей стадии. Достоинствами такой схемы являются:

- получение в конце каждой стадии законченного набора проектной документации, отвечающего требованиям полноты и согласованности;
- простота планирования процесса разработки.

Именно такую схему и используют обычно при блочно-иерархическом подходе к разработке сложных технических объектов, обеспечивая очень высокие параметры эффективности разработки. Однако данная схема оказалась применимой только к созданию систем, для которых в самом начале разработки удавалось точно и полно сформулировать все требования. Это уменьшало вероятность возникновения в процессе разработки проблем, связанных с принятием неудачного решения на предыдущих стадиях. На практике такие разработки встречается крайне редко.

Модель с промежуточным контролем. Схема, поддерживающая итерационный характер процесса разработки, была названа схемой с промежуточным контролем. Контроль, который выполняется по данной схеме после завершения каждого этапа, позволяет при необходимости вернуться на любой уровень и внести необходимые изменения.

Основная опасность использования такой схемы связана с тем, что разработка никогда не будет завершена, постоянно находясь в состоянии уточнения и усовершенствования.

Спиральная модель. Программное обеспечение создается не сразу, а итерационно с использованием метода прототипирования, базирующегося на создании прототипов. Именно появление прототипирования привело к тому, что процесс модификации программного обеспечения перестал восприниматься, как «необходимое зло», а стал восприниматься как отдельный важный процесс.

Прототипом называют действующий программный продукт, реализующий отдельные функции и внешние интерфейсы разрабатываемого программного обеспечения.

На первой итерации, как правило, специфицируют, проектируют, реализуют и тестируют интерфейс пользователя. На второй - добавляют некоторый ограниченный набор функций. На последующих этапах этот набор расширяют, наращивая возможности данного продукта.

Основным достоинством данной схемы является то, что, начиная с некоторой итерации, на которой обеспечена определенная функциональная полнота, продукт можно предоставлять пользователю, что позволяет:

- сократить время до появления первых версий программного продукта;
- заинтересовать большое количество пользователей, обеспечивая быстрое продвижение следующих версий продукта на рынке;
- ускорить формирование и уточнение спецификаций за счет появления практики использования продукта;
- уменьшить вероятность морального устаревания системы за время разработки.

Основной проблемой использования спиральной схемы является определение моментов перехода на следующие стадии. Для ее решения обычно ограничивают сроки прохождения каждой стадии, основываясь на экспертных оценках.

Изменение жизненного цикла программного обеспечения при использовании CASE-технологий. CASE-технологии представляют собой совокупность методологий анализа, проектирования, разработки и сопровождения сложных программных систем, основанных как на структурном, так и на объектном подходах, которые поддерживаются комплексом взаимосвязанных средств автоматизации.

Нотацией называют систему обозначений, используемых для описания некоторого класса моделей. Нотации бывают графические и текстовые. В CASE-технологиях нотации используют для описания структуры проектируемой системы, элементов данных, этапов обработки и т. п.

Средства - инструментарий для поддержки методов: средства создания и редактирования графического проекта, организации проекта в виде иерархии уровней абстракции, а также проверки соответствия компонентов разных уровней. Различают:

- CASE - средства анализа требований, проектирования спецификаций и структуры, редактирования интерфейсов (первое поколение CASE-I);
- CASE - средства генерации исходных текстов и реализации интегрированного окружения поддержки полного жизненного цикла разработки программного обеспечения (второе поколение CASE-II).
- CASE-I в основном включают средства для поддержки графических моделей, проектирования спецификаций, экранных редакторов и словарей данных. CASE-II отличается существенно большими возможностями, обеспечивая: контроль, анализ и связывание системной информации и информации по управлению процессом проектирования, построение прототипов и моделей системы, тестирование, верификацию и анализ сгенерированных программ.

Автоматизируя трудоемкие операции, современные CASE-средства существенно повышают производительность труда программистов и улучшают качество создаваемого программного обеспечения. Они:

- обеспечивают автоматизированный контроль совместимости спецификаций проекта;
- уменьшают время создания прототипа системы;
- ускоряют процесс проектирования и разработки;
- автоматизируют формирование проектной документации для всех этапов жизненного цикла;
- частично генерируют коды программ для различных платформ разработки;
- поддерживают технологии повторного использования компонентов системы;
- обеспечивают возможность восстановления проектной документации по имеющимся исходным кодам.

Появление CASE-технологий изменило все этапы жизненного цикла программного обеспечения, при этом наибольшие изменения касаются анализа и проектирования, которые предполагают строгое и наглядное описание разрабатываемого программного обеспечения

1.6. Ускорение разработки программного обеспечения. Технология RAD

Разработка спиральной модели жизненного цикла программного обеспечения и CASE-технологий позволили сформулировать условия, выполнение которых сокращает сроки создания программного обеспечения.

Современная технологии проектирования, разработки и сопровождения программного обеспечения, должна отвечать следующим требованиям:

- поддержка полного жизненного цикла программного обеспечения;
- гарантированное достижение целей разработки с заданным качеством и в установленное время;
- возможность выполнения крупных проектов в виде подсистем, разрабатываемых группами исполнителей ограниченной численности (3-7 человек) с последующей интеграцией составных частей, и координации ведения общего проекта;
- минимальное время получения работоспособной системы;
- возможность управления конфигурацией проекта, ведения версий проекта и автоматического выпуска проектной документации по каждой версии;
- независимость выполняемых проектных решений от средств реализации (СУБД, операционных систем, языков и систем программирования);
- поддержка комплексом согласованных CASE-средств, обеспечивающих автоматизацию процессов, выполняемых на всех стадиях жизненного цикла.

Этим требованиям отвечает технология RAD (Rapid Application Development - Быстрая разработка приложений). Эта технология ориентирована, как следует из названия, на максимально быстрое получение первых версий разрабатываемого программного обеспечения. Она предусматривает выполнение следующих условий:

- ведение разработки небольшими группами разработчиков каждая из которых проектирует и реализует отдельные подсистемы проекта - позволяет улучшить управляемость проекта;
- использование итерационного подхода способствует уменьшению времени получения работоспособного прототипа;
- наличие четко проработанного графика цикла, рассчитанного не более чем на три месяца, существенно увеличивает эффективность работы.

1.7. Оценка качества процессов создания программного обеспечения

Возросли требования к качеству разрабатываемого программного обеспечения, что требует совершенствования процессов их разработки. На настоящий момент существует несколько стандартов, связанных с оценкой качества этих процессов, которое обеспечивает организация-разработчик. К наиболее известным относят:

- международные стандарты серии ISO 9000 (ISO 9000 - ISO 9004);
- CMM - Capability Maturity Model - модель зрелости (совершенствования) процессов создания программного обеспечения;
- рабочая версия международного стандарта ISO/IEC 15504: Information Technology - Software Process Assessment; эта версия более известна подназванием SPICE - (Software Process Improvement and Capability dEtermi-nation — определение возможностей и улучшение процесса создания про граммного обеспечения).

2.1. Понятие технологичности программного обеспечения

Под технологичностью понимают качество проекта программного продукта, от которого зависят трудовые и материальные затраты на его реализацию и последующие модификации. Хороший проект сравнительно быстро и легко кодируется, тестируется, отлаживается и модифицируется.

Технологичность программного обеспечения определяется проработанностью его

моделей, уровнем независимости модулей, стилем программирования и степенью повторного использования кодов.

Чем лучше проработана модель разрабатываемого программного обеспечения, тем четче определены подзадачи и структуры данных, хранящие входную, промежуточную и выходную информацию, тем проще их проектирование и реализация и меньше вероятность ошибок, для исправления которых потребуется существенно изменять программу. Высокая технологичность проекта особенно важна, если разрабатывается программный продукт, рассчитанный на многолетнее интенсивное использование, или необходимо обеспечить повышенные требования к его качеству.

2.2. Модули и их свойства

При проектировании достаточно сложного программного обеспечения после определения его общей структуры выполняют декомпозицию компонентов в соответствии с выбранным подходом до получения элементов, которые, по мнению проектировщика, в дальнейшей декомпозиции не нуждаются.

Как уже упоминалось раньше, в настоящее время используют два способа декомпозиции разрабатываемого программного обеспечения, связанные с соответствующим подходом:

- процедурный (или структурный - по названию подхода);
- объектный.

Результатом процедурной декомпозиции является иерархия подпрограмм (процедур), в которой функции, связанные с принятием решения, реализуются подпрограммами верхних уровней, а непосредственно обработка - подпрограммами нижних уровней. Это согласуется с принципом вертикального управления, который был сформулирован вместе с другими рекомендациями структурного подхода к программированию. Он также ограничивает возможные варианты передачи управления, требуя, чтобы любая подпрограмма возвращала управление той подпрограмме, которая ее вызвала.

Модули. Модулем называют автономно компилируемую программную единицу. Термин «модуль» традиционно используется в двух смыслах. Первоначально, когда размер программ был сравнительно невелик, и все подпрограммы компилировались отдельно, под модулем понималась подпрограмма, т. е. последовательность связанных фрагментов программы, обращение к которой выполняется по имени. Со временем, когда размер программ значительно вырос, и появилась возможность создавать библиотеки ресурсов: констант, переменных, описаний типов, классов и подпрограмм, термин «модуль» стал использоваться и в смысле автономно компилируемый набор программных ресурсов.

Данные модуль может получать и/или возвращать через общие области памяти или параметры.

Первоначально к модулям (еще понимаемым как подпрограммы) предъявлялись следующие требования:

- отдельная компиляция;
- одна точка входа;
- одна точка выхода;
- соответствие принципу вертикального управления;
- возможность вызова других модулей;
- небольшой размер (до 50-60 операторов языка);

- независимость от истории вызовов;
- выполнение одной функции.

Сцепление модулей. Сцепление является мерой взаимозависимости модулей, которая определяет, насколько хорошо модули отделены друг от друга. Модули независимы, если каждый из них не содержит о другом никакой информации. Чем больше информации о других модулях хранит модуль, тем больше он с ними сцеплен.

Различают пять типов сцепления модулей:

- по данным;
- по образцу;
- по управлению;
- по общей области данных;
- по содержимому.

Сцепление по данным предполагает, что модули обмениваются данными, представленными скалярными значениями.

Сцепление по образцу предполагает, что модули обмениваются данными, объединенными в структуры.

При сцеплении по управлению один модуль посылает другому некоторый информационный объект (флаг), предназначенный для управления внутренней логикой модуля. Таким способом часто выполняют настройку режимов работы программного обеспечения.

Сцепление по общей области данных предполагает, что модули работают с общей областью данных.

В случае сцепления по содержимому один модуль содержит обращения к внутренним компонентам другого, что полностью противоречит блочно-иерархическому подходу.

Как правило, модули сцепляются между собой несколькими способами. Учитывая это, качество программного обеспечения принято определять по типу сцепления с худшими характеристиками.

Связность модулей. Связность - мера прочности соединения функциональных и информационных объектов внутри одного модуля. Если сцепление характеризует качество отделения модулей, то связность характеризует степень взаимосвязи элементов, реализуемых одним модулем.

Различают следующие виды связности (в порядке убывания уровня):

- функциональную;
- последовательную;
- информационную (коммуникативную);
- процедурную;
- временную;
- логическую;
- случайную.

При функциональной связности все объекты модуля предназначены для выполнения одной функции

При последовательной связности функций выход одной функции служит исходными данными для другой функции

Информационно связанными считают функции, обрабатывающие одни и те же данные

Процедурно связаны функции или данные, которые являются частями одного процесса

Временная связность функций подразумевает, что эти функции выполняются параллельно или в течение некоторого периода времени

Логическая связь базируется на объединении данных или функций в одну логическую группу

В том случае, если связь между элементами мала или отсутствует, считают, что они имеют случайную связность.

Библиотеки ресурсов. Различают библиотеки ресурсов двух типов:

библиотеки подпрограмм и библиотеки классов.

Библиотеки подпрограмм реализуют функции, близкие по назначению, например, библиотека графического вывода информации. Связность подпрограмм между собой в такой библиотеке - логическая, а связность самих подпрограмм - функциональная, так как каждая из них обычно реализует одну функцию.

Библиотеки классов реализуют близкие по назначению классы. Связность элементов класса - информационная, связность классов между собой может быть функциональной - для родственных или ассоциированных классов и логической - для остальных.

2.3. Нисходящая и восходящая разработка программного обеспечения

При проектировании, реализации и тестировании компонентов структурной иерархии, полученной при декомпозиции, применяют два подхода:

- восходящий;
- нисходящий.

Восходящий подход. При использовании восходящего подхода сначала проектируют и реализуют компоненты нижнего уровня, затем предыдущего и т. д. По мере завершения тестирования и отладки компонентов осуществляют их сборку, причем компоненты нижнего уровня при таком подходе часто помещают в библиотеки компонентов.

Для тестирования и отладки компонентов проектируют и реализуют специальные тестирующие программы.

Подход имеет следующие недостатки:

- увеличение вероятности несогласованности компонентов вследствие неполноты спецификаций;
- наличие издержек на проектирование и реализацию тестирующих программ, которые нельзя преобразовать в компоненты;
- позднее проектирование интерфейса, а соответственно невозможность продемонстрировать его заказчику для уточнения спецификаций и т. д.

Нисходящий подход. Нисходящий подход предполагает, что проектирование и последующая реализация компонентов выполняется «сверху-вниз», т. е. вначале проектируют компоненты верхних уровней иерархии, затем следующих и так далее до самых нижних уровней. В той же последовательности выполняют и реализацию компонентов. При этом в процессе программирования компоненты нижних, еще не реализованных уровней заменяют специально разработанными отладочными модулями - «заглушками», что позволяет тестировать и отлаживать уже реализованную часть.

При использовании нисходящего подхода применяют иерархический, операционный и комбинированный методы определения последовательности проектирования и реализации компонентов.

Иерархический метод предполагает выполнение разработки строго по уровням.

Исключения допускаются при наличии зависимости по данным, т. е. если обнаруживается, что некоторый модуль использует результаты другого, то его рекомендуется программировать после этого модуля. Основной проблемой данного метода является большое количество достаточно сложных заглушек. Кроме того, при использовании данного метода основная масса модулей разрабатывается и реализуется в конце работы над проектом, что затрудняет распределение человеческих ресурсов.

Комбинированный метод учитывает следующие факторы, влияющие на последовательность разработки:

- достижимость модуля - наличие всех модулей в цепочке вызова данного модуля;
- зависимость по данным - модули, формирующие некоторые данные, должны создаваться раньше обрабатывающих;
- обеспечение возможности выдачи результатов - модули вывода результатов должны создаваться раньше обрабатывающих;
- готовность вспомогательных модулей - вспомогательные модули, например, модули закрытия файлов, завершения программы, должны создаваться раньше обрабатывающих;
- наличие необходимых ресурсов.

2.4. Стиль оформления программы

С точки зрения технологичности хорошим считают стиль оформления программы, облегчающий ее восприятие как самим автором, так и другими программистами, которым, возможно, придется ее проверять или модифицировать. Именно исходя из того, что любую программу неоднократно придется просматривать, следует придерживаться хорошего стиля написания программ.

Стиль оформления программы включает:

- правила именования объектов программы (переменных, функций, типов, данных и т. п.);
- правила оформления модулей;
- стиль оформления текстов модулей.

2.5 Стиль оформления тестов модулей

Правила оформления модулей. Каждый модуль должен предваряться заголовком, который, как минимум, содержит:

- название модуля;
- краткое описание его назначения;
- краткое описание входных и выходных параметров с указанием единиц измерения;
- список используемых (вызываемых) модулей;
- краткое описание алгоритма (метода) и/или ограничений;
- ФИО автора программы;
- идентифицирующую информацию (номер версии и/или дату последней корректировки).Стиль оформления текстов модулей. Стиль оформления текстов модулей определяет использование отступов, пропусков строк и комментариев, облегчающих понимание программы. Как правило, пропуски строк и комментарии используют для визуального разделения частей модуля.

2.6. Эффективность и технологичность

Традиционно эффективными считают программы, требующие минимального времени выполнения и/или минимального объема оперативной памяти. Особые требования к эффективности программного обеспечения предъявляют при наличии ограничений (на время реакции системы, на объем оперативной памяти и т. п.). В случаях, когда обеспечение эффективности не требует серьезных временных и трудовых затрат, а также не приводит к существенному ухудшению технологических свойств, необходимо это требование иметь в виду.

Способы экономии памяти. Принятие мер по экономии памяти предполагает, что в каких-то случаях эта память неэкономно использовалась. Учитывая, что анализировать имеет смысл только операции размещения данных. существенно влияющие на характеристику эффективности, следует обращать особое внимание на выделение памяти под данные структурных типов.

Способы уменьшения времени выполнения. Как уже упоминалось выше, для уменьшения времени выполнения в первую очередь необходимо анализировать циклические участки программы с большим количеством повторений. При их написании необходимо по возможности:

- выносить вычисление константных, т. е. не зависящих от параметров цикла, выражений из циклов;
- избегать «длинных» операций умножения и деления, заменяя их сложением, вычитанием и сдвигами;
- минимизировать преобразования типов в выражениях;
- оптимизировать запись условных выражений - исключать лишние проверки;
- исключать многократные обращения к элементам массивов по индексам.

2.7. Программирование «с защитой от ошибок»

Любая из ошибок программирования, которая не обнаруживается на этапах компиляции и компоновки программы, в конечном счете может проявиться тремя способами: привести к выдаче системного сообщения об ошибке, «зависанию» компьютера и получению неверных результатов.

Проверки правильности выполнения операций ввода-вывода. Причинами неверного определения исходных данных могут являться, как внутренние ошибки - ошибки устройств ввода-вывода или программного обеспечения, так и внешние ошибки - ошибки пользователя. При этом принято различать:

- ошибки передачи - аппаратные средства, например, вследствие неисправности, искажают данные;
- ошибки преобразования - программа неверно преобразует исходные данные из входного формата во внутренний;
- ошибки перезаписи - пользователь ошибается при вводе данных, например, вводит лишний или другой символ;
- ошибки данных - пользователь вводит неверные данные.

Ошибки передачи обычно контролируются аппаратно

Проверка допустимости промежуточных результатов. Проверки промежуточных результатов позволяют снизить вероятность позднего проявления не только ошибок неверного определения данных, но и некоторых ошибок кодирования и проектирования.

Однако следует также иметь в виду, что любые дополнительные операции в программе требуют использования дополнительных ресурсов (времени, памяти и т. п.) и могут также содержать ошибки. Поэтому имеет смысл проверять не все промежуточные результаты, а только те, проверка которых целесообразна, т. е. возможно позволит обнаружить ошибку, и не сложна. Например:

- если каким-либо образом вычисляется индекс элемента массива, то следует проверить, что этот индекс является допустимым;
- если строится цикл, количество повторений которого определяется значением переменной, то целесообразно убедиться, что значение этой переменной не отрицательно;
- если определяется вероятность какого-либо события, то целесообразно проверить, что полученное значение не более 1, а сумма вероятностей всех возможных независимых событий равна 1 и т. д.

Предотвращение накопления погрешностей. Чтобы снизить погрешности результатов вычислений, необходимо соблюдать следующие рекомендации:

- избегать вычитания близких чисел (машинный ноль);
- избегать деления больших чисел на малые;
- сложение длинной последовательности чисел начинать с меньших по абсолютной величине;
- стремиться по возможности уменьшать количество операций;
- использовать методы с известными оценками погрешностей;
- не использовать условие равенства вещественных чисел;
- вычисления производить с двойной точностью, а результат выдавать - с одинарной.

Обработка исключений. Поскольку полный контроль данных на входе и в процессе вычислений, как правило, невозможен, следует предусматривать перехват обработки аварийных ситуаций.

2.8. Сквозной структурный контроль

Сквозной структурный контроль представляет собой совокупность технологических операций контроля, позволяющих обеспечить как можно более раннее обнаружение ошибок в процессе разработки.

Сквозной структурный контроль должен выполняться на специальных контрольных сессиях, в которых, помимо разработчиков, могут участвовать специально приглашенные эксперты. Время между сессиями определяет объем материала, который выносится на сессию: при частых сессиях материал рассматривают небольшими порциями, при редких - существенными фрагментами. Материалы для очередной сессии должны выдаваться участникам заранее, чтобы они могли их обдумать.

Одна из первых сессий должна быть организована на этапе определения спецификаций. На этой сессии проверяют полноту и точность спецификаций, при этом целесообразно присутствие заказчика или специалиста по предметной области, которые смогут определить, насколько правильно и полно определены спецификации программного обеспечения. На этапе проектирования вручную по частям проверяют алгоритмы разрабатываемого программного обеспечения на конкретных наборах данных и сверяют полученные результаты с соответствующими спецификациями. Основная задача - убедиться в правильности понимания спецификаций и проанализировать достоинства и недостатки концептуальных решений, закладываемых в

проект.

На этапе реализации проверяют план (последовательность) реализации модулей, набор тестов, а также тексты отдельных модулей.

Для всех этапов целесообразно иметь списки наиболее часто встречающихся ошибок, которые формируют по литературным источникам и исходя из опыта предыдущих разработок. Такие списки позволяют сконцентрировать усилия на конкретных моментах, а не проверять все подряд. При этом все найденные ошибки фиксируют в специальном документе, но не исправляют их.

Помимо раннего обнаружения ошибок, сквозной структурный контроль обеспечивает своевременную подготовку качественной документации по проекту.

3. Определение требований к программному обеспечению и исходных данных для его проектирования

Этап постановки задачи - один из наиболее ответственных этапов создания программного продукта. На этом этапе формулируют основные требования к разрабатываемому программному обеспечению.

3.1. Классификация программных продуктов по функциональному признаку

По назначению все программные продукты можно разделить на три группы: системные, прикладные и гибридные.

К *системным* обычно относят программные продукты, обеспечивающие функционирование вычислительных систем (как отдельных компьютеров, так и сетей). Это - операционные системы, оболочки и другие служебные программы (утилиты).

ОС, как правило, управляют ресурсами (процессором и памятью), процессами (задачами и потоками) и устройствами.

Оболочки служат для организации более удобного интерфейса пользователя с файловой системой.

К утилитам принято относить программы и системы, непосредственно не входящие в состав операционной системы, но обеспечивающие выполнение определенных функций.

Продукты общего назначения используют разные группы пользователей. К ним можно отнести текстовые редакторы, электронные таблицы, графические редакторы, информационные системы общего назначения, и тп.

Профессиональные продукты предназначены для специалистов в различных областях, например, к ним можно отнести:

- системы автоматизации проектирования, ориентированные на различные технические области;
- системы-тренажеры;
- бухгалтерские системы;
- издательские системы;
- профессиональные графические системы;
- экспертные системы и т. д.

Системы автоматизации производственных процессов отличаются от профессиональных тем, что они ориентированы на пользователей разных профессий, связанных единым производственным процессом.

Обучающие программы и системы в соответствии со своим названием предназначены

для обучения.

К развлекающим относят игровые программы, музыкальные программы, информационные системы, но с тестами развлекающего характера, например гороскопы и т. п.

Гибридные системы сочетают в себе признаки системного и прикладного программного обеспечения.

3.2 Основные эксплуатационные требования к программным продуктам

Эксплуатационные требования определяют некоторые характеристики разрабатываемого программного обеспечения, проявляемые в процессе его функционирования. К таким характеристикам относят:

- правильность - функционирование в соответствии с техническим заданием;
- универсальность - обеспечение правильной работы при любых допустимых данных и защиты от неправильных данных;
- надежность (помехозащищенность) - обеспечение полной повторяемости результатов, т. е. обеспечение их правильности при наличии различного рода сбоев;
- проверяемость - возможность проверки получаемых результатов;
- точность результатов - обеспечение погрешности результатов не выше заданной;
- защищенность - обеспечение конфиденциальности информации;
- программная совместимость - возможность совместного функционирования с другим программным обеспечением;
- аппаратная совместимость - возможность совместного функционирования с некоторым оборудованием;
- эффективность - использование минимально возможного количества ресурсов технических средств, например, времени микропроцессора или объема оперативной памяти;
- адаптируемость - возможность быстрой модификации с целью приспособления к изменяющимся условиям функционирования;
- повторная входимость - возможность повторного выполнения без перезагрузки с диска (Это требование обычно к ПО, резидентно загруженному в оперативную память, например драйверам. Необходимо так организовать программу, чтобы никакие ее исходные данные не затирались в процессе выполнения или восстанавливались в начале или при завершении каждого вызова.)
- реентерабельность - возможность «параллельного» использования не сколькими процессами. (В этом случае все данные, изменяемые программой в процессе выполнения, должны быть выделены в специальный блок, копия которого создается для каждого процесса при вызове программы).

Сложность многих программных систем не позволяет сразу сформулировать четкие требования к ним. Обычно для перехода от идеи создания некоторого программного обеспечения к четкой формулировке требований, которые могут быть занесены в техническое задание, необходимо выполнить предпроектные исследования в области разработки.

3.3 Предпроектные исследования предметной области

Целью предпроектных исследований является преобразование общих нечетких знаний о предназначении будущего программного обеспечения в сравнительно точные

требования к нему.

Существуют два варианта неопределенности:

- неизвестны методы решения формулируемой задачи - такого типа неопределенности обычно возникают при решении научно-технических задач;

- неизвестна структура автоматизируемых информационных процессов обычно встречается при построении автоматизированных систем управления предприятиями.

- В первом случае во время предпроектных исследований определяют возможность решения поставленной задачи и методы, позволяющие получить требуемый результат, что может потребовать соответствующих научных исследований как фундаментального, так и прикладного характера, разработки и исследования новых моделей объектов реального мира.

Во втором случае определяют:

- структуру и взаимосвязи автоматизируемых информационных процессов;

- распределение функций между человеком и системой, а также между аппаратурой и программным обеспечением;

- функции программного обеспечения; внешние условия его функционирования и особенности его интерфейсов, как с пользователями, так и при необходимости - с аппаратной частью;

- требования к программным и информационным компонентам, необходимые аппаратные ресурсы, требования к базам данных и физические характеристики программных компонент.

Результаты предпроектных исследований предметной области используют в процессе разработки технического задания.

3.4 Разработка технического задания

Техническое задание представляет собой документ, в котором сформулированы основные цели разработки, требования к программному продукту, определены сроки и этапы разработки и регламентирован процесс приемно-сдаточных испытаний.

Факторы, определяющие характеристики разрабатываемого программного обеспечения:

- исходные данные и требуемые результаты, которые определяют функции программы или системы;

- среда функционирования (программная и аппаратная) - может быть задана, а может выбираться для обеспечения параметров, указанных в техническом задании;

- возможное взаимодействие с другим программным обеспечением и/или специальными техническими средствами - также может быть определено, а может выбираться исходя из набора выполняемых функций.

Разработка технического задания выполняется в следующей последовательности. Устанавливают набор выполняемых функций, а также перечень и характеристики исходных данных. Затем определяют перечень результатов, их характеристики и способы представления. Далее уточняют среду функционирования программного обеспечения: конкретную комплектацию и параметры технических средств, версию используемой операционной системы и, возможно, версии и параметры другого установленного программного обеспечения, с которым предстоит взаимодействовать будущему программному продукту.

В случаях, когда разрабатываемое программное обеспечение собирает и хранит

некоторую информацию или включается в управление каким-либо техническим процессом, необходимо также четко регламентировать действия программы в случае сбоев оборудования и энергоснабжения.

На техническое задание существует стандарт ГОСТ 19.201-78 «Техническое задание. Требования к содержанию и оформлению». В соответствии с этим стандартом техническое задание должно содержать следующие разделы:

- введение;
- основания для разработки;
- назначение разработки;
- требования к программе или программному изделию;
- требования к программной документации;
- технико-экономические показатели;
- стадии и этапы разработки;
- порядок контроля и приемки.

Введение должно включать наименование и краткую характеристику области применения программы или программного продукта, а также объекта (например, системы) в котором предполагается их использовать.

Раздел Основания для разработки должен содержать наименование документа, на основании которого ведется разработка, организации, утвердившей данный документ, и наименование или условное обозначение темы разработки. Таким документом может служить план, приказ, договор и т. п.

Раздел *Назначение разработки* должен содержать описание функционального и эксплуатационного назначения программного продукта с указанием категорий пользователей.

Раздел Требования к программе или программному изделию должен включать 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.

В разделе *Требования* к программной документации указывают необходимость наличия руководства программиста, руководства пользователя, руководства системного программиста, пояснительной записки и т. п.

В разделе *Технико-экономические показатели* рекомендуется указывать ориентировочную экономическую эффективность, предполагаемую годовую потребность и экономические преимущества по сравнению с существующими аналогами.

В разделе *Стадии и этапы разработки* указывают стадии разработки, этапы и содержание работ с указанием сроков разработки и исполнителей.

В разделе *Порядок контроля* и приемки указывают виды испытаний и общие требования к приемке работы.

3.5 Принципиальные решения начальных этапов проектирования

На начальных этапах процесса проектирования должны быть приняты принципиальные решения, во многом определяющие этот процесс, а также качество и трудоемкость разработки:

3.5.2 Выбор архитектуры программного обеспечения

Архитектурой программного обеспечения называют совокупность базовых концепций (принципов) его построения. Архитектура программного обеспечения определяется сложностью решаемых задач, степенью универсальности разрабатываемого программного обеспечения и числом пользователей, одновременно работающих с одной его копией. Различают:

- однопользовательскую архитектуру, при которой программное обеспечение рассчитано на одного пользователя, работающего за персональным компьютером;
- многопользовательскую архитектуру, которая рассчитана на работу в локальной или глобальной сети.

Кроме того, в рамках однопользовательской архитектуры различают:

- программы;
- пакеты программ;
- программные комплексы;
- программные системы.

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

Программой называют адресованный компьютеру набор инструкций, точно описывающий последовательность действий, которые необходимо выполнить для решения конкретной задачи.

Пакеты программ представляют собой совокупность программ, решающих задачи некоторой прикладной области. Например, пакет графических программ, пакет математических программ. Программы такого пакета связаны между собой только принадлежностью к определенной прикладной области.

Программные комплексы представляют собой совокупность программ, совместно обеспечивающих решение небольшого класса сложных задач одной прикладной области.

Программные системы представляют собой организованную совокупность программ (подсистем), позволяющую решать широкий класс задач из некоторой прикладной области. В отличие от программных комплексов программы, входящие в программную систему, взаимодействуют через общие данные.

Многопользовательские программные системы в отличие от обычных программных систем должны организовывать сетевое взаимодействие отдельных компонентов программного обеспечения, что еще усложняет процесс его разработки.

3.5.3 Выбор типа пользовательского интерфейса

Различают четыре типа пользовательских интерфейсов:

- примитивные - реализуют единственный сценарии работы, например, ввод данных - обработка - вывод результатов;
- меню - реализуют множество сценариев работы, операции которых организованы в иерархические структуры, например, «вставка»: «вставка файла», «вставка символа» и т. д.;
- со свободной навигацией - реализуют множество сценариев, операции которых не

привязаны к уровням иерархии, и предполагают определение множества возможных операций на конкретном шаге работы; интерфейсы данной формы в основном используют Windows-приложения;

- прямого манипулирования - реализуют множество сценариев, представленных в операциях над объектами, основные операции инициируются перемещением пиктограмм объектов мышью, данная форма реализована в интерфейсе самой операционной системы Windows альтернативно интерфейсу со свободной навигацией.

Тип пользовательского интерфейса во многом определяет сложность и трудоемкость разработки, которые существенно возрастают в порядке перечисления типов.

Кроме того, выбор типа интерфейса включает выбор технологии работы с документами. Различают две технологии:

- однодокументная, которая предполагает однодокументный интерфейс (SDI - Single Document Interface);
- многодокументная, которая предполагает многодокументный интерфейс (MDI - Multiple Document Interface).

Многодокументную технологию используют, если программное обеспечение должно работать с несколькими документами одновременно, например, с несколькими текстами или несколькими изображениями. Однодокументную — если одновременная работа с несколькими документами не обязательна.

3.5.4 Выбор языка программирования

В большинстве случаев никакой проблемы выбора языка программирования реально не существует. Язык может быть определен:

- организацией, ведущей разработку;
- программистом, который по возможности всегда будет использовать хорошо знакомый язык;
- устоявшимся мнением и т. п.

Все же все существующие языки программирования можно разделить на следующие группы:

- универсальные языки высокого уровня;
- специализированные языки разработчика программного обеспечения;
- специализированные языки пользователя;
- языки низкого уровня.

В группе универсальных языков высокого уровня безусловным лидером на сегодня является язык C/C++. Действительно различные версии C и C++ имеют целый ряд очень существенных достоинств:

- многоплатформенность - для всех используемых в настоящее время платформ существуют компиляторы с языка C и C++;
- наличие операторов, реализующих основные структурные алгоритмические конструкции (условную обработку, все виды циклов);
- возможность программирования на низком (системном) уровне с использованием адресов оперативной памяти;
- огромные библиотеки подпрограмм и классов.

Однако C и C++ имеют и серьезные недостатки:

- отсутствие полноценных встроенных структурных типов данных;
- наличие синтаксических неоднозначностей, которые также не позволяют

компилятору контролировать правильность программы;

- ограниченный контроль параметров, передаваемых в подпрограмму, что также обнаруживается только в процессе отладки программы, и т. п.

Альтернативой С и С++ среди универсальных языков программирования, используемых для создания прикладного программного обеспечения, на сегодня является Pascal, компиляторы которого в силу четкого синтаксиса обнаруживают помимо синтаксических и большое количество семантических ошибок.

Специализированные языки пользователя обычно являются частью профессиональных сред пользователя, характеризуются узкой направленностью и разработчиками программного обеспечения не используются.

Языки низкого уровня позволяют осуществлять программирование практически на уровне машинных команд. При этом получают самые оптимальные, как с точки зрения времени выполнения, так и с точки зрения объема необходимой памяти программы.

3.5.5 Выбор среды программирования

Средой программирования называют программный комплекс, который включает специализированный текстовый редактор, встроенные компилятор, компоновщик, отладчик, справочную систему и другие программы, использование которых упрощает процесс написания и отладки программ.

Наиболее часто используемыми являются визуальные среды Delphi, С++ Builder фирмы Borland (Inprise Corporation), Visual C++, Visual Basic фирмы Microsoft, Visual Ada фирмы IBM и др.

Между основными визуальными средами этих фирм Delphi, С++ Builder и Visual C++ имеется существенное различие: визуальные среды фирмы Microsoft обеспечивают более низкий уровень программирования «под Windows». Это является их достоинством и недостатком. Достоинством -так как уменьшается вероятность возникновения «нестандартной» ситуации, т. е. ситуации, не предусмотренной разработчиками библиотеки компонентов, а недостатком - так как это существенно загружает программиста «рутинной» работой, от которой избавлен программист, работающий с Delphi или С++ Builder.

3.5.6 Выбор или формирование стандартов разработки:

Реальное применение любой технологии проектирования требует формирования или выбора ряда стандартов, которые должны соблюдаться всеми участниками проекта:

- стандарт проектирования;
- стандарт оформления проектной документации;
- стандарт интерфейса пользователя.

Стандарт проектирования должен определять:

- набор необходимых моделей (схем, диаграмм) на каждой стадии проектирования и степень их детализации;
- правила фиксации проектных решений на диаграммах, в том числе правила именования объектов и соглашения по терминологии, набор атрибутов для всех объектов и правила их заполнения на каждой стадии, правила оформления диаграмм, включая требования к форме и размерам объектов;
- требования к конфигурации рабочих мест разработчиков, включая настройки операционной системы и используемых CASE-средств;
- механизм обеспечения совместной работы над проектом, в том числе и правила интеграции подсистем проекта и анализа проектных решений на непротиворечивость.

Стандарт оформления проектной документации должен регламентировать:

- комплектность, состав и структуру документации на каждой стадии;
- требования к ее содержанию и оформлению;
- правила подготовки, рассмотрения, согласования и утверждения документов.

Стандарт интерфейса пользователя должен определять:

- правила оформления экранов (шрифты и цветовую палитру), состав и расположение окон и элементов управления;
- правила пользования клавиатурой и мышью;
- правила оформления текстов помощи;
- перечень стандартных сообщений;
- правила обработки реакции пользователя.

4. Анализ требований, определение спецификаций и проектирование программного обеспечения при структурном подходе

Разработка любого ПО начинается с анализа требований. В результате анализа получают спецификации разрабатываемого ПО: выполняют декомпозицию и содержательную постановку решаемых задач, уточняют их взаимодействие и эксплуатационные ограничения. В процессе определения спецификаций строят общую модель предметной области, как некоторой части реального мира, с которой будет тем или иным способом взаимодействовать разрабатываемое программное обеспечение, и конкретизируют его основные функции.

4.1 Спецификации программного обеспечения при структурном подходе

Спецификации представляют собой полное и точное описание функций и ограничений разрабатываемого программного обеспечения. При этом одна часть спецификаций (функциональные) описывает функции разрабатываемого программного обеспечения, а другая часть (эксплуатационные) определяет требования к техническим средствам, надежности, информационной безопасности и т. д.

Определение отражает главные требования к спецификациям. Применительно к функциональным спецификациям подразумевается, что:

- требование полноты означает, что спецификации должны содержать всю существенную информацию, где ничего важного не было бы упущено, и отсутствует несущественная информация, например детали реализации, что бы не препятствовать разработчику в выборе наиболее эффективных решений;
- требование точности означает, что спецификации должны однозначно восприниматься как заказчиком, так и разработчиком.

Формальные модели, используемые на этапе определения спецификаций можно разделить на две группы: модели, зависящие от подхода к разработке (структурного или объектно-ориентированного), и модели, не зависящие от него. Так диаграммы переходов состояний, которые демонстрируют особенности поведения разрабатываемого программного обеспечения при получении тех или иных сигналов извне, и математические модели предметной области используют при любом подходе к разработке.

В рамках структурного подхода на этапе анализа и определения спецификаций используют три типа моделей: ориентированные на функции, ориентированные на данные и ориентированные на потоки данных. Каждую модель целесообразно

использовать для своего специфического класса программных разработок.

4.2 Диаграммы переходов состояний

Диаграмма переходов состояний является графической формой предоставления конечного автомата — математической абстракции, используемой для моделирования детерминированного поведения технических объектов или объектов реального мира.

На этапе анализа требований и определения спецификаций диаграмма переходов состояний демонстрирует поведение разрабатываемой программной системы при получении управляющих воздействий. Под, управляющими воздействиями или сигналами в данном случае понимают управляющую информацию, получаемую системой извне.

Для построения диаграммы переходов состояний необходимо в соответствии с теорией конечных автоматов определить: основные состояния, управляющие воздействия (или условия перехода), выполняемые действия и возможные варианты переходов из одного состояния в другое.

Полученную диаграмму переходов состояний следует согласовать с заказчиком программного обеспечения.

4.3 Функциональные диаграммы

Функциональными называют диаграммы, в первую очередь отражающие взаимосвязи функций разрабатываемого программного обеспечения. Активностная модель, предложенная Д. Россом в составе методологии функционального моделирования SADT (Structured Analysis and Design Technique - технология структурного анализа и проектирования) в 1973 г:

Отображение взаимосвязи функций активностной модели осуществляется посредством построения иерархии функциональных диаграмм, схематически представляющих взаимосвязи нескольких функций. Каждый блок такой диаграммы соответствует некоторой функции, для которой должны быть определены: исходные данные, результаты, управляющая информация и механизмы ее осуществления - человек или технические средства.

Блоки на диаграмме размещают по «ступенчатой» схеме в соответствии с последовательностью их работы или доминированием, которое понимается как влияние, оказываемое одним блоком на другие. В функциональных диаграммах SADT различают пять типов влияний блоков друг на друга:

- вход - выход блока подается на вход блока с меньшим доминированием, т. е. следующего;

управление - выход блока используется как управление для блока с меньшим доминированием (следующего);

обратная связь по входу - выход блока подается на вход блока с большим доминированием (предыдущего);

обратная связь по управлению - выход блока используется как управляющая информация для блока с большим доминированием (предыдущего);

выход-исполнитель - выход блока используется как механизм для другого блока.

Дуги могут разветвляться и соединяться вместе различными способами. Разветвление означает, что часть или вся информация может использоваться в каждом ответвлении дуги.

В процессе построения иерархии диаграмм фиксируют всю уточняющую информацию и строят словарь данных, в котором определяют структуры и элементы данных, показанных на диаграммах.

Таким образом, в результате получают спецификацию, которая состоит из иерархии функциональных диаграмм, спецификаций функций нижнего уровня и словаря, имеющих ссылки друг на друга.

Функциональную модель целесообразно применять для определения спецификаций программного обеспечения, не предусматривающего работу со сложными структурами данных, так как она ориентирована на декомпозицию функций.

4.4 Диаграммы потоков данных

Диаграммы потоков данных позволяют специфицировать как функции разрабатываемого программного обеспечения, так и обрабатываемые им данные. При использовании этой модели систему представляют в виде иерархии диаграмм потоков данных, описывающих асинхронный процесс преобразования информации с момента ввода в систему до выдачи пользователю. На каждом следующем уровне иерархии происходит уточнение процессов, пока очередной процесс не будет признан элементарным.

В основе модели лежат понятия внешней сущности, процесса, хранилища (накопителя) данных и потока данных.

Внешняя сущность - материальный объект или физическое лицо, выступающие в качестве источников или приемников информации, например, заказчики, персонал, поставщики, клиенты, банк и т. п.

Процесс - преобразование входных потоков данных в выходные в соответствии с определенным алгоритмом.

Хранилище данных - абстрактное устройство для хранения информации.

Поток данных - процесс передачи некоторой информации от источника к приемнику.

Таким образом, диаграмма иллюстрирует как потоки данных, порожденные некоторыми внешними сущностями, трансформируются соответствующими процессами (или подсистемами), сохраняются накопителями данных и передаются другим внешним сущностям - приемникам информации.

Построение иерархии диаграмм потоков данных начинают с диаграммы особого вида — контекстной диаграммы, которая определяет наиболее общий вид системы.

На следующем этапе каждую подсистему контекстной диаграммы детализируют при помощи диаграмм потоков данных. В процессе детализации соблюдают правило балансировки - при детализации подсистемы можно использовать компоненты только тех подсистем, с которыми у разрабатываемой подсистемы существует информационная связь (т. е. с которыми она связана потоками данных).

Окончательно разработку модели выполняют в два этапа.

1 этап - построение контекстной диаграммы - включает выполнение следующих действий:

- классификацию множества требований и организацию их в основные функциональные группы - процессы;
- идентификацию внешних объектов - внешних сущностей, с которыми система должна быть связана;
- идентификацию основных видов информации - потоков данных, циркулирующей

между системой и внешними объектами;

- предварительную разработку контекстной диаграммы;
- изучение предварительной контекстной диаграммы и внесение в нее изменений по результатам ответов на возникающие при изучении вопросы по всем ее частям;
- построение контекстной диаграммы путем объединения всех процессов предварительной диаграммы в один процесс, а также группирования потоков.

2 этап - формирование иерархии диаграмм потоков данных – включает для каждого уровня:

- проверку и изучение основных требований по диаграмме соответствующего уровня (для первого уровня - по контекстной диаграмме);
- декомпозицию каждого процесса текущей диаграммы потоков данных с помощью детализирующей диаграммы или - если некоторую функцию сложно или невозможно выразить комбинацией процессов, построение спецификации процесса;
- добавление определений новых потоков в словарь данных при каждом появлении их на диаграмме;
- проведение ревизии с целью проверки корректности и улучшения наглядности модели после построения двух-трех уровней.

Полная спецификация процессов включает также описание структур данных, используемых как при передаче информации в потоке, так и при хранении в накопителе.

4.5 Структуры данных и диаграммы отношений компонентов данных

4.5.1 Структуры данных

Структурой данных называют совокупность правил и ограничений, которые отражают связи, существующие между отдельными частями (элементами) данных.

Различают абстрактные структуры данных, используемые для уточнения связей между элементами, и конкретные структуры, используемые для представления данных в программах.

Все абстрактные структуры данных можно разделить на три группы: структуры, элементы которых не связаны между собой, структуры с неявными связями элементов - таблицы и структуры, связь элементов которых указывается явно - графы.

В первую группу входят множества и кортежи. Наиболее существенная характеристика элемента данных в этих структурах - его принадлежность некоторому набору, т. е. отношение вхождения.

Ко второй группе относят векторы, матрицы, массивы (многомерные), записи, строки, а также таблицы, включающие перечисленные структуры в качестве частей.

4.5.2 Модели данных

В зависимости от описываемых типов отношений модели структур данных принято делить на иерархические и сетевые.

Иерархические модели позволяют описывать упорядоченные или неупорядоченные отношения вхождения элементов данных в компонент более высокого уровня, т. е. множества, таблицы и их комбинации. К иерархическим моделям относят модель Джексона-Орра, для графического представления которой можно использовать:

- диаграммы Джексона, предложенные в составе методики проектирования программного обеспечения того же автора в 1975 г.;
- скобочные диаграммы Орра, предложенные в составе методики проектирования

программного обеспечения Варнье-Орра (1974).

Сетевые модели основаны на графах, а потому позволяют описывать связность элементов данных независимо от вида отношения, в том числе комбинации множеств, таблиц и графов. К сетевым моделям, например, относят модель «сущность-связь» (ER - Entity-Relationship), обычно используемую при разработке баз данных.

4.5.3 Диаграммы Джексона

В основе диаграмм Джексона лежит предположение о том, что структуры данных, так же, как и программ, можно строить из элементов с использованием всего трех основных конструкций: последовательности, выбора и повторения.

Каждая конструкция представляется в виде двухуровневой иерархии, на верхнем уровне которой расположен блок конструкции, а на нижнем - блоки элементов. Нотации конструкций различаются специальными символами в правом верхнем углу блоков элементов. В изображении последовательности дополнительный символ отсутствует. В изображении выбора ставится символ «о» (латинское) - сокращение английского «или» (or). Конструкции последовательности и выбора должны содержать по два или более элементов второго уровня. В изображении повторения в блоке единственного (повторяющегося) элемента ставится символ «*».

4.5.4 Скобочные диаграммы Орра

Диаграмма Орра базируется на том же предположении о сходстве структур программ и данных, что и диаграмма Джексона. Отличие состоит лишь в нотации. Автор предлагает для представления конструкций данных использовать фигурные скобки

4.5.5 Сетевая модель данных

Сетевые модели данных используют в тех случаях, если отношение между компонентами данных не исчерпываются включением. Для графического представления разновидностей этой модели используют несколько нотаций. Наиболее известны из них следующие:

- нотация П. Чена;
- нотация Р. Баркера;
- нотация IDEF1 (более современный вариант этой нотации - IDEF1X используется в CASE-системах, например в системе ERWin).

Нотация Баркера является наиболее распространенной.

Базовыми понятиями сетевой модели данных являются: сущность, атрибут и связь.

Сущность - реальный или воображаемый объект, имеющий существенное значение для рассматриваемой предметной области.

Каждая сущность должна:

- иметь уникальное имя;
- обладать одним или несколькими атрибутами, которые либо принадлежат сущности, либо наследуются через связь;
- обладать одним или несколькими атрибутами, которые однозначно идентифицируют каждый экземпляр сущности.

Каждая сущность обладает одним или несколькими атрибутами. Атрибут - любая характеристика сущности, значимая для рассматриваемой предметной области и предназначенная для квалификации, идентификации, классификации, количественной характеристики или выражения состояния сущности.

Атрибуты делятся на ключевые, т. е. входящие в состав уникального идентификатора, который называют первичным ключом, и описательные - прочие.

Связь - поименованная ассоциация между двумя или более сущностями, значимая для рассматриваемой предметной области.

Каждая сущность может быть связана любым количеством связей с другими сущностями модели. Связь предполагает некоторое отношение сущностей, которое характеризуется количеством экземпляров сущности, участвующих в связи с каждой стороны.

4.6 Разработка структурной и функциональной схем

Процесс проектирования сложного программного обеспечения начинают с уточнения его структуры, т. е. определения структурных компонентов и связей между ними. Результат уточнения структуры может быть представлен в виде структурной и/или функциональной схем и описания (спецификаций) компонентов.

4.6.1 Структурная схема разрабатываемого программного обеспечения

Структурной называют схему, отражающую состав и взаимодействие по управлению частей разрабатываемого программного обеспечения.

Структурные схемы пакетов программ не информативны, поскольку организация программ в пакеты не предусматривает передачи управления между ними. Поэтому структурные схемы разрабатывают для каждой программы пакета, а список программ пакета определяют, анализируя функции, указанные в техническом задании.

Самый простой вид программного обеспечения - программа, которая в качестве структурных компонентов может включать только подпрограммы и библиотеки ресурсов. Разработку структурной схемы программы обычно выполняют методом пошаговой детализации.

Структурными компонентами программной системы или программного комплекса могут служить программы, подсистемы, базы данных, библиотеки ресурсов и т. п.

Структурная схема программного комплекса демонстрирует передачу управления от программы-диспетчера соответствующей программе.

Структурная схема программной системы, как правило, показывает наличие подсистем или других структурных компонентов. В отличие от программного комплекса отдельные части (подсистемы) программной системы интенсивно обмениваются данными между собой и, возможно, с основной программой. Структурная же схема программной системы этого обычно не показывает.

4.6.2 Функциональная схема

Функциональная схема или схема данных - схема взаимодействия компонентов программного обеспечения с описанием информационных потоков, состава данных в потоках и указанием используемых файлов и устройств. Для изображения функциональных схем используют специальные обозначения, установленные стандартом.

Функциональные схемы более информативны, чем структурные.

Все компоненты структурных и функциональных схем должны быть описаны. При структурном подходе особенно тщательно необходимо прорабатывать спецификации межпрограммных интерфейсов, так как от качества их описания зависит количество самых дорогостоящих ошибок.

4.7 Использование метода пошаговой детализации для проектирования структуры программного обеспечения

Структурный подход к программированию в том виде, в котором он был

сформулирован в 70-х годах XX в., предлагал осуществлять декомпозицию программ методом пошаговой детализации. Результатом декомпозиции является структурная схема программы, которая представляет собой многоуровневую иерархическую схему взаимодействия подпрограмм по управлению. Минимально такая схема отображает два уровня иерархии, т. е. показывает общую структуру программы. Однако тот же метод позволяет получить структурные схемы с большим количеством уровней.

Метод пошаговой детализации реализует нисходящий подход и базируется на основных конструкциях структурного программирования. Он предполагает пошаговую разработку алгоритма. Каждый шаг при этом включает разложение функции на подфункции. Так на первом этапе описывают решение поставленной задачи, выделяя общие подзадачи, на следующем аналогично описывают решение подзадач, формулируя при этом подзадачи следующего уровня. Таким образом, на каждом шаге происходит уточнение функций проектируемого программного обеспечения. Процесс продолжают, пока не доходят до подзадач, алгоритмы решения которых очевидны.

Декомпозируя программу методом пошаговой детализации, следует придерживаться основного правила структурной декомпозиции, следующего из принципа вертикального управления: в первую очередь детализировать управляющие процессы декомпозируемого компонента, оставляя уточнение операций с данными напоследок. Это связано с тем, что приоритетная детализация управляющих процессов существенно упрощает структуру компонентов всех уровней иерархии и позволяет не отделять процесс принятия решения от его выполнения: так, определив условие выбора некоторой альтернативы, сразу же вызывают модуль, ее реализующий.

Детализация операций со структурами в последнюю очередь позволит отложить уточнение их спецификаций и обеспечит возможность относительно безболезненной модификации этих структур за счет сокращения количества модулей, зависящих от этих данных.

Кроме этого, целесообразно придерживаться следующих рекомендаций:

- не отделять операции инициализации и завершения от соответствующей обработки, так как модули инициализации и завершения имеют плохую связность (временную) и сильное сцепление (по управлению);
- не проектировать слишком специализированных или слишком универсальных модулей, так как проектирование излишне специальных модулей увеличивает их количество, а проектирование излишне универсальных модулей повышает их сложность;
- избегать дублирования действий в различных модулях, так как при их изменении исправления придется вносить во все фрагменты программы, где они выполняются - в этом случае целесообразно просто реализовать эти действия в отдельном модуле;
- группировать сообщения об ошибках в один модуль по типу библиотеки ресурсов, тогда будет легче согласовать формулировки, избежать дублирования сообщений, а также перевести сообщения на другой язык.

При этом, описывая решение каждой задачи, желательно использовать не более 1-2-х структурных управляющих конструкций, таких, как ветвление, что позволяет четче представить себе структуру организуемого вычислительного процесса.

4.8 Структурные карты Константайна

На структурной карте отношения между модулями представляют в виде графа,

вершинам которого соответствуют модули и общие области данных, а дугам - межмодульные вызовы и обращения к общим областям данных.

Различают четыре типа вершин:

- модуль - подпрограмма,
- подсистема - программа,
- библиотека - совокупность подпрограмм, размещенных в отдельном модуле,
- область данных - специальным образом оформленная совокупность данных, к которой возможно обращение извне.

При этом отдельные части программной системы (программы, подпрограммы) могут вызываться последовательно, параллельно или как сопрограммы.

Чаще всего используют последовательный вызов, при котором модули, передав управление, ожидают завершения выполнения вызванной программы или подпрограммы, чтобы продолжить прерванную обработку.

Под параллельным вызовом понимают распараллеливание вычислений на нескольких вычислителях, когда при активизации другого процесса данный процесс продолжает работу. На однопроцессорных компьютерах в мультипрограммных средах в этом случае начинается попеременное выполнение соответствующих программ.

Если стрелка, изображающая вызов, касается блока, то обращение происходит к модулю целиком, а если входит в блок, то - к элементу внутри модуля.

При необходимости на структурной карте можно уточнить особые условия вызова: циклический вызов, условный вызов и однократный вызов - при повторном вызове основного модуля однократно вызываемый модуль не активизируется.

Связи по данным и управлению обозначают стрелками, параллельными дуге вызова, направление стрелки указывает направление связи.

Структурные карты Константайна позволяют наглядно представить результат декомпозиции программы на модули и оценить ее качество, т. е. соответствие рекомендациям структурного программирования (сцепление и связность).

4.9 Проектирование структур данных

4.9.1 Основные параметры при проектировании

Под проектированием структур данных понимают разработку их представлений в памяти. Основными параметрами, которые необходимо учитывать при проектировании структур данных, являются:

- вид хранимой информации каждого элемента данных;
- связи элементов данных и вложенных структур;
- время хранения данных структуры («время жизни»);
- совокупность операций над элементами данных, вложенными структурами и структурами в целом.

Вид хранимой информации определяет тип соответствующего поля памяти. В качестве элементов данных в зависимости от используемого языка программирования могут рассматриваться:

- целые и вещественные числа различных форматов;
- символы;
- булевские значения: true и false;

а также некоторые структурные типы данных, например:

- строки;

- записи;
- специально объявленные классы.

При этом для числовых полей очень важно правильно определить диапазон возможных значений, а для строковых данных - максимально возможную длину строки.

4.9.2 Представление данных в оперативной памяти

Различают две базовые структуры организации данных в оперативной памяти: векторную и списковую.

Векторная структура представляет собой последовательность байт памяти, которые используются для размещения полей данных. Последовательное размещение организованных структур данных позволяет осуществлять прямой доступ к элементам: по индексу (совокупности индексов) - в массивах или строках или по имени поля - в записях или объектах.

Однако выполнение операций добавления и удаления элементов при использовании векторных структур для размещения элементов массивов может потребовать осуществления многократных сдвигов элементов.

Структуры данных в векторном представлении можно размещать как в статической, так и в динамической памяти. Расположение векторных представлений в динамической памяти иногда позволяет существенно увеличить эффективность использования оперативной памяти. Желательно размещать в динамической памяти временные структуры, хранящие промежуточные результаты, и структуры, размер которых сильно зависит от вводимых исходных данных.

Списковые структуры строят из специальных элементов, включающих помимо информационной части еще и один или несколько указателей - адресов элементов или вложенных структур, связанных с данным элементом. Размещая подобные элементы в динамической памяти можно организовывать различные внутренние структуры. Однако при использовании списковых структур следует помнить, что:

- для хранения указателей необходима дополнительная память;
- поиск информации в линейных списках осуществляется последовательно, а потому требует больше времени;
- построение списков и выполнение операций над элементами данных, хранящимися в списках, требует более высокой квалификации программистов, более трудоемко, а соответствующие подпрограммы содержат больше ошибок и, следовательно, требуют более тщательного тестирования.

4.9.3 Представление данных во внешней памяти

Современные операционные системы поддерживают два способа организации данных во внешней памяти: последовательный и с прямым доступом.

При последовательном доступе к данным возможно выполнение только последовательного чтения элементов данных или последовательная их запись. Такой вариант предполагается при работе с логическими устройствами типа клавиатуры или дисплея, при обработке текстовых файлов или файлов, формат записей которых меняется в процессе работы.

Прямой доступ возможен только для дисковых файлов, обмен информацией с которыми осуществляется записями фиксированной длины. Адрес записи такого файла можно определить по ее номеру, что и позволяет напрямую обращаться к нужной записи.

При выборе типа памяти для размещения структур данных следует иметь в виду, что:

- в оперативной памяти размещают данные, к которым необходим быстрый доступ как для чтения, так и для их изменения;
- во внешней - данные, которые должны сохраняться после завершения программы.

Возможно, что во время работы данные целесообразно хранить в оперативной памяти для ускорения доступа к ним, а при ее завершении - переписывать во внешнюю память для длительного хранения.

4.10 Проектирование программного обеспечения, основанное на декомпозиции данных

Практически одновременно были предложены методики проектирования программного обеспечения Джексона и Вар-нье-Орра, основанные на декомпозиции данных. Обе методики предназначены для создания «простых» программ, работающих со сложными, но иерархически организованными структурами данных. При необходимости разработки программных систем в обоих случаях предлагается вначале разбить систему на отдельные программы, а затем использовать данные методики.

4.10.1 Методика Джексона

Методика Джексона. При создании своей методики М. Джексон исходил из того, что структуры исходных данных и результатов определяют структуру программы.

Методика основана на поиске соответствий структур исходных данных и результатов. Однако при ее применении возможны ситуации, когда на каких-то уровнях соответствия отсутствуют. Например, записи исходного файла сортированы не в том порядке, в котором соответствующие строки должны появляться в отчете. Такие ситуации были названы «столкновениями».

Разработка структуры программы в соответствии с методикой выполняется следующим образом:

- строят изображение структур входных и выходных данных;
- выполняют идентификацию связей обработки (соответствия) между этими данными;
- формируют структуру программы на основании структур данных и обнаруженных соответствий;
- добавляют блоки обработки элементов, для которых не обнаружены соответствия;
- анализируют и обрабатывают несоответствия, т.е. разрешают «столкновения»;
- добавляют необходимые операции (ввод, вывод, открытие/закрытие файлов и т. п.);
- записывают программу в структурной нотации (псевдокоде).

4.10.2 Методика Варнье-Орра

Методика Варнье-Орра базируется на том же положении, что и методика Джексона, но основными при построении программы считаются структуры выходных данных и, если структуры входных данных не соответствуют структурам выходных, то их допускается менять. Таким образом, ликвидируется основная причина столкновений.

Однако на практике не всегда существует возможность пересмотра структур входных данных: эти структуры уже могут быть строго заданы, например, если используются данные, полученные при выполнении других программ, поэтому данную методику применяют реже.

4.11 Ca$e-технологии, основанные на структурных методологиях анализа и проектирования

К нашему времени накоплен опыт успешного использования большинства известных

методологий структурного анализа и проектирования в соответствующих CASE-средствах. Наибольшее распространение получили методологии: SADT, структурного системного анализа Гейна-Сар-сона и др.

Наибольшее применение нашли структурные методологии, использующие диаграммы потоков данных. Это вызвано двумя причинами:

- диаграммы потоков данных более детально по сравнению с функциональными диаграммами отображают специфику многочисленных в настоящее время информационных систем: не требуют строгой типизации обрабатываемой информации, предусматривают возможность хранения данных, конкретизируют взаимодействие с внешним миром, предусматривают получение комплексной модели программного обеспечения и т. п.;
- разработан метод построения проектных спецификаций (структурных карт Джексона или Костантайна) по диаграммам потоков данных, что позволяет автоматически создавать такие спецификации.

Несмотря на то, что последнее время все большее распространение получают объектно-ориентированные средства разработки программного обеспечения, структурные методологии продолжают совершенствовать. Их успешно применяют при разработке многих программных продуктов, например, для уточнения требований к системам, основной частью которых являются базы данных, очень часто используют диаграммы потоков данных.

5 Анализ требований, определение спецификаций и проектирование программного обеспечения при объектном подходе

Модели разрабатываемого программного обеспечения при объектном подходе основаны на предметах и явлениях реального мира. В основе этих моделей также лежит описание требуемого поведения разрабатываемого программного обеспечения, т. е. его функциональности, но это поведение связывается с состояниями элементов (объектов) конкретной предметной области.

Таким образом, на этапе анализа ставятся две задачи:

- уточнить требуемое поведение разрабатываемого программного обеспечения;
- разработать концептуальную модель его предметной области с точки зрения поставленных задач.

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

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

5.1 UML - стандартный язык описания разработки прогр. продуктов с использованием объектн. подхода

1995 – создается UML (Unified Modeling Language - унифицированный язык моделирования).

Всего UML предлагает девять дополняющих друг друга диаграмм, входящих в различные модели:

- диаграммы вариантов использования

- диаграммы классов
- диаграммы пакетов
- диаграммы последовательностей действий
- диаграммы кооперации
- диаграммы деятельностей
- диаграммы состояний объектов
- диаграммы компонентов;
- диаграммы размещения.

Все указанные диаграммы по возможности используют единую графическую нотацию, что облегчает их понимание.

Помимо указанных диаграмм, как и при структурном подходе, спецификация обязательно включает словарь терминов, а также различного рода описания и текстовые спецификации. Конкретный набор документации определяется разработчиком. Спецификация разрабатываемого программного обеспечения при использовании UML объединяет несколько моделей: использования, логическую, реализации, процессов, развертывания.

Модель использования представляет собой описание функциональности программного обеспечения с точки зрения пользователя.

Логическая модель описывает ключевые абстракции программного обеспечения, т. е. средства, обеспечивающие требуемую функциональность.

Модель реализации определяет реальную организацию программных модулей в среде разработки.

Модель процессов отображает организацию вычислений и оперирует понятиями «процессы» и «нити». Она позволяет оценить производительность, масштабируемость и надежность программного обеспечения.

И, наконец, модель развертывания показывает особенности размещения программных компонентов на конкретном оборудовании.

Определение «вариантов использования»

В процессе анализа выявляют внешних пользователей разрабатываемого программного обеспечения и перечень отдельных аспектов его поведения в процессе взаимодействия с конкретными пользователями. Аспекты поведения программного обеспечения были названы «вариантами использования» или «прецедентами». Вариант использования представляет собой характерную процедуру применения разрабатываемой системы конкретным действующим лицом, в качестве которого могут выступать не только люди, но и другие системы или устройства.

5.2.1 Характеристика вариантов

Не следует путать вариант использования с конкретными операциями будущей системы. В зависимости от цели выполнения конкретной процедуры различают следующие варианты использования:

- основные - обеспечивают требуемую функциональность разрабатываемого программного обеспечения:
- вспомогательные - обеспечивают выполнение необходимых настроек системы и ее обслуживание
- дополнительные - обеспечивают дополнительные удобства для пользователя

Вариант использования можно описать кратко или подробно. Краткая форма описания содержит: название варианта использования, его цель, действующих лиц, тип варианта

использования и его краткое описание.

5.2.2 Диаграммы вариантов использования

Диаграммы вариантов использования позволяют наглядно представить ожидаемое поведение системы. Основными понятиями диаграмм вариантов использования являются: действующее лицо, вариант использования, связь.

Действующее лицо - сущность, которая взаимодействует с ним с целью получения или предоставления какой-либо информации

Вариант использования - очевидная для действующего лица процедура, решающая его конкретную задачу.

Связь - взаимодействие действующих лиц и соответствующих вариантов использования

Использование подразумевает, что существует некоторый фрагмент поведения разрабатываемого программного обеспечения, который повторяется в нескольких вариантах использования.

Расширение применяют, если имеется два подобных варианта использования, различающиеся наличием в одном из них некоторых дополнительных действий

5.3 Построение концептуальной модели предметной области

Диаграммы классов - центральное звено объектно-ориентированных методов разработки программного обеспечения, поэтому все существующие методы используют диаграммы классов. UML предлагает использовать три уровня диаграмм классов в зависимости от степени их детализации:

- концептуальный уровень, на котором диаграммы классов, называемые в этом случае контекстными, демонстрируют связи между основными понятиями предметной области;
- уровень спецификаций, на котором диаграммы классов отображают интерфейсы классов предметной области, т. е. связи объектов этих классов;
- уровень реализации, на котором диаграммы классов непосредственно показывают поля и операции конкретных классов.

Концептуальные модели оперируют понятиями предметной области, атрибутами этих понятий и отношениями между ними. Основным понятиям в модели ставятся в соответствие классы. Класс при этом традиционно понимают как совокупность общих признаков задан ной группы объектов предметной области. В качестве атрибутов представляют некоторые, существенные с точки зрения решаемой задачи характеристики объектов. Под отношением классов понимают статическую, т. е. не зависящую от времени, связь между классами. Различают два основных вида отношений: ассоциация и обобщение. Отношение ассоциации означает наличие связи между экземплярами классов или объектами, например, класс.

Связь между экземплярами классов подразумевает некоторые роли, которые соответствующие объекты играют по отношению друг к другу.

5.3.1 Диаграмма последовательностей системы

Диаграмма последовательностей системы — графическая модель, которая для определенного сценария варианта использования показывает генерируемые действующими лицами события и их порядок. Для построения диаграммы последовательностей системы необходимо:

- представить систему как «черный ящик» и изобразить для нее линию жизни -

вертикальную пунктирную линию, подходящую к блоку снизу;

- идентифицировать каждое действующее лицо и изобразить для него линию жизни (много действующих лиц бывает в вариантах совместного использования программного обеспечения);
- из описания варианта использования определить множество системных событий и их последовательность;
- изобразить системные события в виде линий со стрелкой на конце между линиями жизни действующих лиц и системы, а также указать имена событий и списки передаваемых значений.

5.3.2 Системные события и операции

В отличие от внутренних событий, события, которые генерируются для системы действующими лицами, называют системными. Системные события инициируют выполнение соответствующего множества операций, также называемых системными. Каждую системную операцию называют по имени соответствующего сообщения.

Множество всех системных операций определяют, идентифицируя системные события всех вариантов использования. Для наглядности системные операции изображают в виде операций абстрактного класса (типа) System. Каждую системную операцию необходимо описать. Обычно описание системной операции содержит:

- имя операции и ее параметры;
- описание обязанности;
- указание типа;
- названия вариантов использования, в которых она используется;
- примечания для разработчиков алгоритмов и т. д.;
- описание обработки возможных исключений;
- описание вывода неинтерфейсных сообщений;
- предположение о состоянии системы до выполнения операции (предусловие);
- описание изменения состояния системы после выполнения операции (постусловие).

5.3.3 Диаграммы деятельностей

В зависимости от степени детализации диаграммы деятельностей так же, как диаграммы классов, используют на разных этапах разработки. На этапе анализа требований и уточнения спецификаций диаграммы деятельностей позволяют конкретизировать основные функции разрабатываемого программного обеспечения.

Под деятельностью в данном случае понимают задачу (операцию), которую необходимо выполнить вручную или с помощью средств автоматизации. В теоретическом плане диаграммы деятельности являются обобщенным представлением алгоритма, реализующего анализируемый вариант использования. На диаграмме деятельность обозначается прямоугольником с закругленными углами. Диаграммы деятельностей позволяют описывать альтернативные и параллельные процессы.

5.4 Разработка структуры программного обеспечения при объектном подходе

Большинство классов можно отнести к определенному типу, который применительно к данному подходу называют стереотипом, например:

- классы-сущности (классы предметной области);
- граничные (интерфейсные) классы;
- управляющие классы;

- исключения и т. д.

Классы-сущности используют для представления сущностей реального мира или внутренних элементов системы.

Граничные классы обеспечивают взаимодействие между действующими лицами и внутренними элементами системы.

Управляющие классы служат для моделирования последовательного поведения, заложенного в один или несколько вариантов использования. Пакетом при объектном подходе называют совокупность описаний классов и других программных ресурсов. Диаграмма пакетов показывает, из каких частей состоит проектируемая программная система, и как эти части связаны друг с другом. Связь между пакетами фиксируют, если изменения в одном пакете могут повлечь за собой изменения в другом.