ВЫСШЕЕ ПРОФЕССИОНАЛЬНОЕ ОБРАЗОВАНИЕ

П. Б. ХОРЕВ

ТЕХНОЛОГИИ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОГРАММИРОВАНИЯ

Рекомендовано
Учебно-методическим объединением вузов по университетскому политехническому образованию в качестве учебного пособия для студентов, обучающихся по направлению 654600 «Информатика и вычислительная техника»

Москва
ACADEMIA
2004

УДК 681.3.06(075.8)
ББК 32.973-018я73
Х-792

Рецензенты:
доцент кафедры Информационных систем МГТУ «Станкин»,
канд. техн. наук *М. М. Маран*;
доцент кафедры Прикладной математики МЭИ (ТУ),
канд. техн. наук *К. Г. Меньшикова*

Хорев П. Б.

Х-792 Технологии объектно-ориентированного программирования: Учеб. пособие для студ. высш. учеб. заведений / Павел Борисович Хорев. — М.: Издательский центр «Академия», 2004. — 448 с.

ISBN 5-7695-1522-8

Излагаются основные понятия технологии программирования. Большое внимание уделяется программированию для операционной системы Windows. Рассматриваются наиболее часто используемые в учебном процессе и разработке программного обеспечения системы программирования: Microsoft Visual C++, Borland C++ Builder и Borland Delphi.

Для студентов технических специальностей высших учебных заведений.

УДК 681.3.06(075.8)
ББК 32.973-018я73

ISBN 5-7695-1522-8

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Научиться программировать непросто. Научиться программировать профессионально, т.е. эффективно, быстро и правильно, еще сложнее. С этим вряд ли кто будет спорить. Но можно ли научиться и, самое главное, научить так программировать? Данное учебное пособие и пытается дать ответ на этот вопрос.

Успешная профессиональная деятельность в любой сфере (и программирование здесь не исключение) возможна при условии освоения эффективной технологии такой деятельности. Слово «технология» можно перевести как «наука о мастерстве», т.е. совокупность систематизированных знаний о способах выполнения той или иной работы.

В основе современных технологий программирования лежит объектно-ориентированный подход, позволяющий, в частности, повысить надежность разрабатываемых программных средств и уменьшить объем нового программирования за счет использования классов, созданных ранее. Широко используемые в настоящее время универсальные языки программирования Object Pascal и C++ включают в себя развитые средства объектно-ориентированного подхода, изучение которых является обязательной частью второго этапа подготовки специалистов в области программирования.

Разработка приложений для операционной системы Windows основывается на объектно-ориентированном подходе. Кроме того, приложения Windows относятся к классу программ, управляемых событиями. Наибольшее распространение получили универсальные системы программирования для Windows Borland Delphi, Borland C++ Builder и Microsoft Visual C++. В системах программирования фирмы Borland используется библиотека классов Visual Component Library (VCL), а в системе Visual C++ используется библиотека Microsoft Foundation Classes (MFC). В этих системах программирования используются также совершенно различные технологии так называемого визуального проектирования программ. Поэтому при изучении современных технологий разработки программных средств целесообразно освоение как системы программирования Delphi (или C++ Builder), так и Visual C++.

Профессиональный разработчик программных средств должен владеть несколькими языками программирования и навыками создания программного обеспечения с применением различных систем программирования.

Данное учебное пособие состоит из шести глав. В главе 1 излагаются основные понятия технологий программирования — жизненный цикл программного обеспечения и способы проведения его основных этапов (составления технического задания, разработки внешних спецификаций, проектирования структур данных и алгоритмов, проверки правильности

и документирования программ). Рассмотрены также основы автоматизации процессов разработки программ.

Хотя вопросы объектно-ориентированного проектирования программных комплексов включены в учебное пособие, основное внимание уделяется технологиям объектно-ориентированного программирования, принципы которого изложены в главе 2. В частности, подробно рассматриваются классы для организации ввода-вывода, создания динамических структур данных и обработки исключений. Здесь же приводятся базовые сведения о разработке приложений Windows с использованием различных систем программирования. Излагаются принципы разработки программ, управляемых событиями (сообщениями), обработки стандартных событий и событий, созданных автором программы.

В этой и последующих главах вначале излагаются общие принципы решения той или иной задачи на базе объектно-ориентированного подхода, а затем приводятся сведения об использовании этих принципов в конкретных системах программирования и примеры программ.

Глава 3 посвящена созданию пользовательского интерфейса в приложениях Windows — меню, диалоговых окон, панелей управления и строк состояния. Рассматриваются и те вопросы, которые обычно опускаются в руководствах по программированию (в частности, динамическое изменение меню и создание перемещаемых панелей управления, содержащих кроме кнопок произвольные элементы). Особое внимание уделяется вопросам проверки правильности введенных пользователем данных.

В главе 4 рассмотрены методы организации работы с документами в приложениях Windows — их просмотра и редактирования, сохранения и восстановления, печати и регистрации в системном реестре. Изложены редко рассматриваемые способы модификации стандартных диалогов Windows и организации окон предварительного просмотра документов перед печатью в различных системах программирования.

Глава 5 посвящена некоторым дополнительным, но важным вопросам программирования для Windows — созданию и использованию справочных подсистем, разработке приложений с многодокументным интерфейсом, предотвращению повторного запуска приложений.

В главе 6 рассмотрены заключительные этапы разработки программного обеспечения — организация тестирования многомодульных программ, применение средств автоматизации отладки и подготовка окончательного варианта разработанного программного средства.

В необходимых случаях материал учебного пособия, касающийся конкретных систем программирования, сопровождается указаниями на особенности различных версий этих систем.

Автор выражает искреннюю признательность студентам Московского энергетического института (технического университета), Московского государственного горного университета, Московского государственного технического университета гражданской авиации и Московского государственного социального университета, которым он преподавал курс технологии программирования, за стимулирование поиска новых методических приемов и помощь в создании этого учебного пособия. Автор также глубоко признателен своей жене и дочери за неоценимую моральную поддержку во время работы над учебным пособием.

Глава 1
ОСНОВЫ ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ

1.1. Жизненный цикл и критерии качества программы

На заре развития программирования (в историческом смысле это было совсем недавно) компьютерная программа рассматривалась как результат научного творчества и искусства. Программистов было совсем немного, а их труд считался сродни волшебству (см. повесть Аркадия и Бориса Стругацких «Понедельник начинается в субботу»*). В настоящее время ситуация изменилась: к программе подходят как к результату сложного технологического процесса, а программисты — это прежде всего настоящие профессионалы своего дела, искусные, т. е. квалифицированные, мастера.

Качественные изменения произошли также в объемах и сложности разрабатываемых программ. Автор данного пособия хорошо помнит время, когда компилятор с языка программирования мог быть написан одним человеком. Сейчас над проектами программ работают коллективы разработчиков, каждый из которых занимается своим делом: первые определяют архитектуру программного комплекса, вторые занимаются реализацией его отдельных компонентов, третьи выполняют тестирование программных модулей, четвертые пишут документацию. И это еще не все категории участников, так как необходимы также менеджеры проекта, секретари, «архивариусы» и т. п.

Подобно другим промышленным изделиям или продуктам программы должны оцениваться по своим качеству и сложности. Качество программного продукта может характеризоваться различными показателями, каждый из которых должен по возможности подлежать измерению и количественной оценке путем ввода соответствующих формализованных метрик. Это позволит упорядочить разработку, эксплуатацию и сопровождение программных средств, придать более четкий характер взаимодействию заказчика, разработчика и пользователя программного обеспечения.

Под сложностью программного обеспечения обычно понимают характеристику программы, отражающую возможности человека по ее разработке и (или) восприятию. Сложность программ также может быть количественно измерена на основе различных метрик. Из теории программирования известны метрики Холстеда, Мак-Кейба и др.

Изменение содержания программирования привело к возрастанию необходимости в совершенных методах и инструментальных средствах

---

* Стругацкий А., Стругацкий Б. Полн. собр. соч.: В 11 т. — Т. 3. — Донецк: Сталкер, 2002.

разработки программ, которые обеспечивали бы сочетание высокого качества программного продукта с короткими сроками его разработки. Исследованием и созданием таких методов и средств и занимается *технология программирования* — наука (совокупность обобщенных и систематизированных знаний) об оптимальных способах организации процесса разработки программных средств, обеспечивающих в заданных условиях получение программного продукта требуемого качества.

Технология программирования должна охватывать содержание всего жизненного цикла программного обеспечения, основными этапами которого являются следующие.

1. Постановка задачи и составление технического задания на разработку программного средства.
2. Уточнение постановки задачи и разработка внешних спецификаций программного средства.
3. Проектирование архитектуры программного комплекса, структур данных и алгоритмов работы его компонентов.
4. Кодирование текстов программных модулей на выбранном языке программирования.
5. Тестирование (проверка правильности) и отладка (поиск и исправление ошибок) программного комплекса.
6. Документирование программного продукта (подготовка описания программы и руководства пользователя).
7. Эксплуатация, сопровождение и модификация (обновление) программного средства.

К основным показателям качества программного продукта можно отнести:

- функциональную пригодность;
- надежность;
- эффективность;
- практичность;
- мобильность.

Под *функциональной пригодностью* понимают характеристику программного средства с точки зрения удовлетворения исходных требований пользователя. Этот показатель качества программ можно представить совокупностью следующих характеристик:

- полнота и точность реализованных программным средством функций;
- корректность (степень соответствия требованиям к программному средству, полученная применением формальных методов доказательства правильности);
- защищенность от несанкционированного доступа и потерь информации;
- совместимость (способность программного средства выполнять свои функции при изменении условий функционирования);
- способность к взаимодействию (простота сопряжения) с другими программными средствами, достигаемая обеспечением доступа к программе по различным каналам — через оперативную память, через файл, по сети и т. п.

Примерами метрик, позволяющих количественно оценить различные характеристики функциональной пригодности, являются:

• отношение числа претензий пользователя к сроку эксплуатации программного средства;

• отношение суммарного числа исключенных, модифицированных и добавленных компонентов программного средства после начала его эксплуатации к общему числу его первоначальных компонентов:

• отношение числа функций программного средства, определенных в техническом задании, к числу функций во внешних спецификациях;

• отношение числа шифруемых данных к общему числу данных, которые должны быть защищены;

• отношение числа функций, способных к взаимодействию, к общему числу функций программного средства.

По аналогии с другими техническими устройствами *надежность* программного средства можно определить как вероятность его работы в течение определенного периода времени, рассчитанную с учетом стоимости для пользователя каждого отказа. Поскольку отказ программного средства — это проявление содержащейся в нем ошибки, следует уточнить понятие «ошибка в программном обеспечении».

Если попытаться определить ошибку в программе как несоответствие ее поведения эталонным характеристикам, которые содержатся в различных документах (техническом задании, внешних спецификациях, руководстве пользователя), то это не приведет к успеху, так как сами документы могут содержать ошибки. Поэтому наиболее точным будет определение ошибки в программе как несоответствие ее поведения разумным ожиданиям пользователя. Слово «разумным» в данном определении подчеркивает тот факт, что проявлением ошибки в программе не будет считаться ситуация, когда пользователь нарушает установленный порядок взаимодействия с программой и вводит данные, допускающие неоднозначную интерпретацию (например, программа запрашивает дату в формате «день. месяц. год», а пользователь вводит «10.02.03», подразумевая не 10 февраля, а 2 октября 2003 г.).

Из данного ранее определения ошибки в программе следует, что надежность программного обеспечения не является его внутренним свойством. Этот показатель связан с тем, как используется программа, и характеризует также уровень квалификации пользователя, качество и условия (температуру, влажность, вибрацию) работы аппаратного обеспечения. Еще одним следствием приведенного определения является то, что никогда нельзя обнаружить все ошибки в программе, так как их проявление зависит как от самой программы, так и от действий ее пользователя. Слово «вероятность» в определении надежности программного средства по существу означает вероятность того, что пользователь программы не введет некоторый конкретный набор данных, выводящий ее из строя.

Надежность программного средства можно представить совокупностью следующих характеристик:

• целостность программного средства (его способность к защите от отказов операционной системы и аппаратных средств);

• живучесть (способность к входному контролю данных и их проверке в ходе работы программы);

• завершенность (бездефектность) программного средства, характеризующая качество проведенного тестирования;

• работоспособность (способность программы к восстановлению своих возможностей после сбоя в работе операционной системы, технического обеспечения или аппаратных средств).

Приведем примеры метрик надежности, позволяющих количественно оценить различные характеристики этого показателя:

• отношение числа исправленных при тестировании программного средства ошибок к прогнозируемому общему числу ошибок в программе;

• отношение общего времени функционирования программного средства к числу наблюдаемых отказов в нем;

• отношение числа обнаруженных ошибок к объему программного средства;

• отношение числа использованных при проверке программного средства тестов к его объему;

• отношение суммарного времени простоя программного средства к числу наблюдаемых отказов в нем.

Остановимся на разнице в надежности аппаратного и программного обеспечения компьютера. Отказы в аппаратуре объясняются тремя факторами: ошибками проектирования, дефектами производства и сбоями. Поскольку дефектные образцы аппаратуры обычно выбраковываются на начальном этапе ее эксплуатации, начиная с некоторого момента времени вероятность отказа аппаратуры определяется сбоями и практически не меняется. По мере «старения» аппаратуры число сбоев начинает расти и вероятность ее отказа вновь повышается.

Отказы в программном обеспечении определяются только ошибками в его проектировании, и по мере их исправления в ходе эксплуатации программы вероятность отказов снижается (в предположении, что при исправлении ошибок не вносятся новые).

Под *эффективностью* программного средства понимают оперативное выполнение ими своих функций (временная эффективность) и рациональное использование ресурсов компьютера (ресурсная экономичность). Поскольку одновременное достижение временной эффективности и ресурсной экономичности невозможно, разработчику программного обеспечения необходимо сделать выбор, который обычно делается в пользу временной эффективности.

Эффективность программы можно представить системой следующих характеристик:

• скорость обработки данных (интервал времени, требуемый на выполнение определенного процесса);

• пропускная способность (число выполненных процессов за интервал времени);

• занятость центрального процессора, оперативной и внешней памяти, коммуникаций и других ресурсов.

Метриками, используемыми для количественной оценки характеристик эффективности, могут быть:

• разность времени получения результата работы программы и времени ее инициализации;

• разность времени начала вывода результата работы программы и времени конца ввода исходных данных;

• число выполненных транзакций (обработанных запросов) за единицу времени;

• суммарный объем оперативной памяти, занятой ее кодом и данными;

• число файлов, используемых при выполнении программы.

С развитием вычислительной техники вопрос приоритетности показателей надежности и эффективности программ решался по-разному.

В начальный период из-за недостатка компьютерной техники и ее низкой надежности на первый план выходили показатели эффективности программного обеспечения.

В настоящее время более важным показателем является надежность программы, так как для большинства пользователей невысокая эффективность еще может быть терпима, но низкая надежность — нет (например, в системах электронной торговли и предоставления услуг).

После того как высокая надежность программного обеспечения станет нормой, на первый план вновь может выйти показатель эффективности (например, в системах искусственного интеллекта).

Под *практичностью* программного средства понимают легкость его использования.

К основным характеристикам практичности относятся:

понятность (способность к пониманию пользователем концепции построения программного средства);

обучаемость (способность к обучению пользователя работе с программным средством);

простота использования программного средства.

Количественная оценка характеристик практичности программы может быть получена с помощью следующих метрик:

• отношение числа четко поясняемых функций программы к общему числу ее функций;

• отношение числа функций программы, поясняемых примерами, к общему числу функций;

• отношение числа опубликованных руководств пользователя к общему числу необходимых для освоения программы руководств;

• отношение числа функций с контекстной помощью к общему числу функций программы;

• отношение числа четких сообщений к общему числу сообщений программы.

Конечно, приведенные выше метрики предполагают экспертную оценку используемых в них величин, например числа четких сообщений. Эта оценка должна быть получена в результате опроса пользователей программного средства.

Важнейшим фактором, влияющим на практичность программного средства, является наличие дружественного интерфейса пользователя, что предполагает:

• использование динамических (изменяющихся в зависимости от состояния программы) меню, содержащих команды для выполнения пользователем функций программы;

• наличие панелей управления и панелей инструментов с графическими элементами, дублирующими основные команды меню;

• поддержку многооконного интерфейса, дающего пользователю возможность работать с несколькими документами или несколькими представлениями одного документа;

• применение для ввода исходных данных, необходимых для решения задачи, панелей диалога — специальных окон определенного формата, в которых используются стандартные элементы управления вводом данных (списки, редакторы, выключатели и т. п.);

• широкое использование компьютерной графики для представления результатов решения задачи, функциональной структуры программного комплекса, хода выполнения отдельных этапов работы и т. п.;

• наличие электронной справочной подсистемы, организованной в виде гипертекста с возможностью поиска нужного раздела по характеризующим его содержание ключевым словам, а также получения контекстной помощи (подсказки) по активным элементам интерфейса пользователя;

• использование для диалога с пользователем (названий команд меню, текстов всех выводимых сообщений и подсказок, справочной информации) только родного для него языка;

• предупреждение возможных ошибок пользователя с помощью блокировки (или исключения) тех команд меню и элементов управления, которые не могут быть выполнены в данный момент;

• максимально быструю сигнализацию о допущенной пользователем ошибке с предоставлением ему возможности исправления ситуации;

• поддержку работы пользователя с двумя стандартными устройствами ввода — клавиатурой и мышью.

Под *мобильностью* (переносимостью) программы понимают сложность изменений, которые нужно внести в само программное средство или его окружение для обеспечения функционирования программы в других операционных системах. Мобильность программного средства может быть на уровне:

• исполнимых кодов (не требуется внесение никаких изменений);

• объектных кодов (требуется перекомпоновка программы с использованием новых системных модулей);

• исходных кодов (требуются повторные компиляция и компоновка программного средства с использованием новой системы программирования);

• алгоритмов (требуются внесение изменений в исходные тексты модулей программы, ее повторная компиляция и перекомпоновка).

1.2. Постановка задачи и разработка внешних спецификаций

На этапе постановки задачи составляется, согласовывается и утверждается совместно заказчиком и исполнителем *техническое задание* на разработку программного средства. Обычно техническое задание включает в себя следующие пункты.

1. Основание для разработки программного обеспечения. Здесь указываются наименования заказчика и исполнителя, а также реквизиты документа (обычно договора), на основании которого ведется разработка.

2. Назначение разрабатываемого программного средства. Здесь определяются функциональное и эксплуатационное назначение программы, т.е. указывается, для решения каких задач и в каких условиях она предназначена.

3. Требования к разрабатываемому программному продукту. Здесь определяются виды требований:

- по составу выполняемых программой функций;
- организации (структуре) входных и выходных данных;
- временной эффективности;
- надежности;
- условиям эксплуатации программы (параметрам окружающей среды, числу и квалификации пользователей);
- составу и характеристикам поддерживаемых аппаратных средств;
- информационной и программной совместимости с другими программными средствами, включая методы решения (например, математические), языки и системы программирования, операционные системы.

4. Требования к программной документации. Здесь определяются состав разрабатываемых документов, специальные требования к их содержанию, способ представления (на бумажном и (или) электронном носителе).

5. Технико-экономическое обоснование. Здесь указывается ожидаемый (расчетный) экономический эффект от применения разрабатываемого программного средства и проводится сравнение с его имеющимися аналогами.

6. Стадии и этапы разработки программного средства (календарный план). Здесь определяются содержание основных этапов выполнения проекта, сроки их выполнения, исполнители работ и перечень документов, представляемых по завершении этапов.

7. Порядок контроля и приемки программного средства. Здесь указываются виды его испытаний и общие требования к организации приемки работы (состав приемочной комиссии, форма и содержание заключительного акта и т.п.).

Каким бы детальным не было техническое задание, оно в принципе не может включать в себя подробное описание поведения разрабатываемой программы. Такое описание не требуется заказчику, так как его интересует только решение поставленной задачи, и не ясно исполнителю, так как может быть получено только в ходе реального выполнения проекта. Получение описания требуемого поведения программы создается на следующем этапе ее жизненного цикла — этапе уточнения постановки задачи и *разработки внешних спецификаций*.

Выполнение этого этапа требует от разработчиков специальных знаний, высокой квалификации и определенного опыта. Поэтому разработкой внешних спецификаций в крупных проектах занимаются отдельные специалисты — спецификаторы, или системные аналитики.

Назовем основные положения, которыми следует руководствоваться при разработке внешних спецификаций, иначе называемых спецификациями требований, или спецификациями задачи:

- поведение программы определяется как для правильных (допустимых) исходных данных, так и для неправильных;

• обязательно исследуются и определяются аспекты, связанные с реакцией программы на возникающие при ее выполнении программные и аппаратные ошибки, с учетом определенных в техническом задании ограничений по надежности разрабатываемого программного средства;

• учитываются определенные в техническом задании ограничения по эффективности разрабатываемой программы, которые могут быть заданы в виде функций от объемов обрабатываемых данных или в абсолютных единицах измерения времени и объемов памяти;

• учитываются ограничения на сроки разработки программы, так как жесткие сроки могут обусловить выбор не самых эффективных, но хорошо известных и апробированных решений;

• отдельно исследуются и выделяются те разделы спецификаций, которые с высокой вероятностью могут быть в дальнейшем подвергнуты изменениям;

• исследуется степень новизны разрабатываемого программного средства и определяется возможность использования в проекте имеющихся модулей и компонентов для снижения трудоемкости разработки.

Рассмотрим применение сформулированных выше положений на конкретном примере уточнения постановки задачи. Пусть имеется файл, содержащий заголовки научных статей по различным отраслям знаний, а также список ключевых слов, характеризующих некоторую предметную область. Необходимо построить упорядоченный по заданным ключевым словам список нужных пользователю заголовков.

Вначале проанализируем аспекты, связанные с нормальной работой разрабатываемой программы. Для решения задачи пользователю необходимо ввести информацию об исходном файле (имени и месторасположении), списке ключевых слов по интересующей его предметной области и выходном файле (также имени и месторасположении). Необходимо ответить на следующие вопросы.

• Каков способ задания имени входного файла — только в результате диалога с пользователем или с дополнительной возможностью ввода в командной строке (с помощью двойного щелчка мышью на значке файла)?

• Какова реакция программы на ввод неправильного имени файла или пути к нему (несуществующего или содержащего другие данные, не относящиеся к решаемой задаче)?

• Каков способ задания ключевых слов (списком, выбором пользователя из всех слов заголовков, выбором из слов заголовков с возможным сохранением их в словаре для последующего использования)?

• Каков способ распознавания в тексте заголовка ключевых слов по их корням (например, компиляция, компилятор и компилируемый должны распознаваться как одно ключевое слово)? Конкретный алгоритм распознавания может быть достаточно простым и эффективным, но приближенным или весьма сложным и трудно реализуемым, но точным.

• Каковы форматы выходного файла, если он не определен в техническом задании (например, поддержка всех форматов, используемых текстовым процессором Microsoft Word), и выходного файла?

Далее следует отразить во внешней спецификации ответы на вопросы, связанные с реакцией программы на возможные ошибки:

• обработка заголовков, содержащих орфографические и (или) синтаксические ошибки (игнорировать возможность появления таких заголовков во входном файле, помещать их в отдельный файл для последующего просмотра самим пользователем, помещать их в конец выходного файла);

• обработка ошибок в словаре ключевых терминов, которые могут, например, возникнуть при его редактировании пользователем в какомлибо текстовом редакторе;

• попытка пользователя завершить работу без сохранения сделанных в ходе сеанса изменений в словаре ключевых терминов (необходимо предусмотреть в этом случае выдачу специального предупреждения и возможность выполнить сохранение словаря).

Следующий момент уточнения постановки задачи в нашем случае — определение реакции программы на аппаратные сбои. Наиболее важными здесь являются вопросы сохранения словаря ключевых терминов и выходного файла. Возможным решением может быть, например, периодическое автоматическое сохранения словаря и выходного файла на диске.

Остановимся на вопросе вероятных модификаций разрабатываемой программы. Поскольку вполне возможна последующая автоматизированная обработка выходного файла, необходимо предусмотреть в разрабатываемой внешней спецификации возможность простого переформатирования результирующего файла.

Внешние спецификации подобно техническому заданию могут разрабатываться на естественном языке. Однако присущие текстам на естественном языке недостатки (неоднозначность понятий, возможность составления внутренне противоречивых спецификаций, возможность сложно проверяемой неполноты спецификаций с точки зрения требований технического задания) приводят к недостаточной корректности разработанных спецификаций. Поэтому в реальных проектах на этапе уточнения постановки задачи часто используется специальный класс языков — *языки спецификации*, занимающие промежуточное место между естественными языками и языками программирования.

Выделяют две группы языков спецификации:

• языки спецификации (описания) требований к программам (ограниченные естественные языки);

• языки функциональной спецификации (формальные языки).

Внешняя спецификация на *языке описания требований* разбивается на секции (фиксированные участки), определяющие выполняемые программой функции, ограничения на входные и выходные данные, взаимосвязь между ними, экранные формы для организации диалога с пользователем, указание на используемый в программе метод решения задачи, реакцию программы на ошибочные ситуации (аномалии), используемые для проверки правильности программы тесты. Синтаксис языка спецификации требований может быть выбран, например, похожим на синтаксис инициализационных файлов операционной системы Windows (ini-файлов). Каждая секция в этом случае может начинаться с ключевого слова, заключенного в квадратные скобки, например:

[Аномалии]

Внутри секции каждая строка спецификации представляется в виде «имя = значение», например:

Причина = Ввод имени несуществующего файла

Реакция = Вывод сообщения об ошибке и предоставление возможности повторного ввода имени файла

Языки функциональной спецификации являются языками программирования сверхвысокого уровня, тексты на которых создаются на основе формальных правил синтаксиса, а смысл этих текстов определяется формальными правилами семантики. Основными свойствами текстов на языках функциональной спецификации (ЯФС) должны быть однозначность и понятность, полнота и непротиворечивость описания задачи. Спецификация на ЯФС определяет, что дано в задаче и что нужно получить, не уточняя способ решения.

Языки функциональной спецификации обычно базируются на каком-либо математическом методе описания семантики функций, а также поддерживают спецификацию функций разрабатываемой программы для какой-либо конкретной предметной области. Чаще всего для построения языков спецификации используются следующие подходы: табличный, алгебраический, графический и логический.

В *табличном подходе* используются таблицы решений. Верхняя часть такой таблицы определяет различные ситуации, в которых требуется выполнить некоторые действия (операции). Каждая строка этой части задает ряд значений некоторой переменной или некоторого условия, которые указаны в первом столбце этой строки. От значений этих переменных или условий зависит выбор определяемых ситуаций. В каждом следующем столбце таблицы решений указывается комбинация значений переменных (условий), определяющая конкретную ситуацию.

Нижняя часть таблицы решений определяет действия, которые требуется выполнить в той или иной ситуации, определяемой в верхней части таблицы решений. Она также состоит из нескольких строк, каждая из которых связана с каким-либо одним конкретным действием, указанным в первом столбце этой строки. В остальных столбцах этой строки указывается, следует ли выполнять это действие в данной ситуации или не следует. Таким образом, первый столбец нижней части этой таблицы представляет собой список обозначений действий, которые могут выполняться в той или иной ситуации, определяемой этой таблицей. В каждом следующем столбце этой части указывается комбинация действий, которые следует выполнить в ситуации, определяемой в том же столбце верхней части таблицы решений.

Ниже приведен фрагмент спецификации задачи с помощью таблицы решений (табл. 1.1).

Из языков функциональной спецификации, использующих *алгебраический подход*, назовем язык SDL (Specification and Description Language — язык спецификаций и определений). В основе SDL лежит теория конечных автоматов (автоматов Мили). В соответствие с этой теорией программа (в терминологии SDL — процесс) представляется множеством своих состояний, множествами входных и выходных сигналов, а также функциями переходов и выходов. Дополнительно предполагается следующее:

Таблица 1.1

Условия	Ситуации при выполнении программы			
...	...	...	...	...
Входной файл с заданным именем существует	...	Да	Нет	...
...	...	...	...	...
Вывод сообщения об ошибке в имени файла	...	Нет	Да	...
Закрытие панели диалога для ввода имени входного файла	...	Да	Нет	...
...	...	...	...	...
Действия	Комбинации выполняемых действий			

• выходной сигнал интерпретируется и как сигнал, посылаемый окружающей среде, и как имя выполняемого автоматом действия;

• входные сигналы могут поступать в произвольные моменты времени, а состояния ожидают поступления сигналов;

• для каждого состояния автомата может задаваться свое множество допустимых входных сигналов, а все остальные поступающие сигналы игнорируются;

• допускается отсутствие выходного сигнала при переходе автомата в другое состояние, а также выдача нескольких выходных сигналов;

• допускается наличие у автомата (помимо состояний) внутренних переменных, значения которых могут изменяться во время переходов.

В состав языка SDL входят типичные для языков сверхвысокого уровня средства — кванторы (существования, существования единственного объекта, существования более одного объекта, всеобщности), циклы по структурам данных (массивам, файлам и т.п.), ассоциативные ссылки на части составных объектов по их свойствам и т.п. Приведем пример записи ссылки на часть массива (возможно, единственный элемент) по заданному свойству:

```
<операция> ELEMENT имя переменной OF имя массива
   SUCHTHAT <условие>
<операция>::=FIRST | LAST | MIN | MAX | ANY | ALL
```

В качестве условия должно быть задано логическое выражение, истинность которого для очередного элемента обеспечивает его включение в выделяемую часть массива.

Язык SDL предназначен в первую очередь для спецификации сложных управляющих систем, представляющих собой совокупность процессов, взаимодействующих между собой в реальном масштабе времени (систем связи, операционных систем, распределенных баз данных и т.п.).

Наряду с линейным синтаксисом язык SDL имеет и графический, что позволяет его отнести и к языкам функциональной спецификации,

использующим *графический подход*. Другими примерами языков спецификации, основанных на графическом подходе, являются граф-схемы алгоритмов и сети Петри.

Сеть Петри представляет собой граф, который состоит из вершин двух типов — позиций и переходов, связанных между собой дугами, причем две вершины одного типа не могут быть соединены непосредственно. Для отражения динамики в сеть вводятся маркеры (метки), размещаемые в вершинах-позициях. Если все позиции, связанные входящими дугами с некоторым переходом, маркированы, то переход срабатывает и маркеры переходят в позиции, связанные исходящими дугами с рассматриваемым переходом. В безопасной сети Петри в позициях не может быть более одного маркера, а в живых сетях Петри имеется возможность срабатывания любого перехода.

Спецификации, созданные на рассмотренных выше языках, обладают свойством исполнимости, т.е. они могут быть преобразованы в машинный язык и выполнены на компьютере (например, с помощью интерпретатора). Полученные таким образом программы никак не могут быть эффективными, так как вопросы реализации в спецификациях не отражаются, поэтому такие программы обычно называют макетами или прототипами реальных программных систем.

Полученный на основе внешней спецификации программы ее прототип может использоваться:

- для проверки правильности понимания исполнителем проекта поставленной заказчиком задачи;
- в качестве эталона при последующем тестировании реальной программной системы;
- в качестве элемента программной документации.

Языки функциональной спецификации, использующие *логический подход*, обычно основываются на многосортном языке логики предикатов первого порядка.

К важнейшим свойствам логических языков спецификации программ следует отнести:

- выразительную способность (возможность формально выражать широкий класс утверждений о свойствах программ);
- дедуктивную способность (возможность формально доказывать истинность записанных на языке спецификации утверждений).

С практической точки зрения логический язык спецификации должен удовлетворять следующим требованиям:

- расширяемость (возможность включения спецификатором новых понятий, отражающих свойства специфицируемой программы);
- модульность (возможность разбиения спецификации на относительно независимые части, модули спецификации);
- возможность создания иерархической структуры сложных понятий, допускающих их вложенность;
- приближенность используемой в спецификации нотации к естественному языку и используемому при реализации проекта языку программирования.

Выполнение этих требований обеспечит ясность спецификаций, возможность накапливания и последующего применения стандартных мо-

дулей спецификации, удобство перехода от спецификации к разработке программных модулей.

Наряду с использованием языков логической спецификации при разработке внешних спецификаций программ они также могут применяться в качестве языков внутренней спецификации (языков определения состояний программ в отдельных точках их выполнения). Внутренние спецификации программы могут использоваться для обеспечения:

- однозначного понимания текста программы всеми участниками разработки программного комплекса;
- исполнимости утверждений о свойствах программы.

Свойство исполнимости позволяет при тестировании программы получить индикаторы ее корректности; при этом записанные на языке логической спецификации утверждения могут использоваться как постусловия для подтверждения истинности полученных результатов или как предусловия для проверки необходимых для правильного выполнения фрагмента программы условий. Исполнимость внутренних спецификаций может использоваться при отладке программ для локализации ошибок, а также для проверки соответствия программы ее спецификации в динамическом режиме (при тестировании) на конкретных входных данных.

В заключение отметим, что не все свойства разрабатываемой программы могут быть эффективно определены с помощью формальных языков функциональной спецификации. Например, они плохо подходят для спецификации пользовательского интерфейса, при котором важную роль играют неформальные соображения. В этом случае удобнее применять неформальные языки спецификации требований.

1.3. Структуры данных, используемые при проектировании программ

Проектирование программы целесообразно начинать с выбора необходимых для решения задачи структур данных, поскольку от правильности такого выбора во многом будут зависеть основные показатели качества разрабатываемой программы — ее надежность и эффективность. Выбор неоптимальной структуры данных приведет к усложнению алгоритма решения задачи и, следовательно, к усложнению понимания, тестирования и отладки программы. Неудачный выбор структур данных не позволит добиться и нужной эффективности разрабатываемого программного средства.

Современные языки программирования в большинстве своем базируются на концепции типов данных — стандартных (имеющих фиксированные имена и не требующих определения в программе) и определяемых разработчиком с учетом особенностей решаемой задачи. *Тип данных* понимается как совокупность множества значений, допустимых для объектов данного типа (констант и переменных), и набора операций, разрешенных для выполнения над элементами этого множества значений.

Типы данных могут быть простыми и составными (структурными). *Простые типы данных* в своем определении не используют ссылок на другие типы данных. К стандартным простым типам данных относятся:

- арифметические (целый, целый без знака и вещественный), которые дополнительно могут различаться размером памяти, выделяемой для хранения объектов данного типа;
- символьный;
- булевский или логический.

Разработчик может ввести собственный простой тип данных (так называемый перечислимый тип), определив для него в программе множество допустимых значений (констант нового типа данных), например:

```
// определение перечислимого типа данных "Дни недели"
// язык Pascal
type Days=(Monday, Tuesday, Wednesday, Thursday,
        Friday, Saturday, Sunday);
// язык C++
enum Days {Monday, Tuesday, Wednesday, Thursday,
          Friday, Saturday, Sunday};
```

К объектам перечислимого типа применимы операции:

- сравнения;
- перехода к следующему значению;
- перехода к предыдущему значению.

Примеры использования в программе перечислимого типа данных:

```
// язык Pascal
uses . . ., Windows; {для использования функции CharToOem из набора
Windows API }
. . .
var
d: Days; // день недели
Msg: array [0..MAX_PATH] of Char; {буфер для преобразованного
сообщения на русском языке в консольном приложении Windows }
. . .
for d:=Sunday downto Monday do
case d of
Monday: begin
   CharToOem('Понедельник',Msg); Writeln(Msg);
end;
. . .
Sunday: begin
CharToOem('Воскресенье',Msg); Writeln(Msg);
end;
end;
// язык C++
#include <stdio.h> // импорт функции printf
#include <windows.h> /* импорт функции CharToOem из набора Windows
API */
. . .
char Msg[_MAX_PATH]; /* буфер для преобразованного сообщения на
русском языке в консольном приложении Windows */
. . .
for(Days d=Sunday;d>=Monday;(int&)d=d-1)
```

```
   switch(d){
   case Monday:
      CharToOem("Понедельник\n",Msg);
      printf(Msg);
      break;
...
      case Sunday:
      CharToOem("Воскресенье\n",Msg); printf(Msg);
      break; }
```

Различают упорядоченные (перечислимые) и неупорядоченные простые типы данных. Все простые типы данных, за исключением вещественных, относятся к перечисляемым.

Составные типы данных определяются с использованием имен других типов данных (как простых, так, возможно, и составных). Существуют следующие стандартные способы создания в программе составного типа данных:

• *массивы* (последовательности объектов одного и того же типа, доступ к которым возможен с помощью операции индексации []);

• *записи*, или *структуры* (объединения объектов разных типов, называемых *полями*, доступ к которым возможен с помощью так называемого составного имени — *имя записи, имя поля*);

• *строки* (последовательности объектов символьного типа, доступ к которым возможен с помощью операции индексации);

• *файлы* (последовательности объектов, размещенных во внешней памяти компьютера, доступ к которым возможен с помощью перемещения указателя текущего объекта);

• *указатели* (объекты, значениями которых могут быть адреса областей оперативной памяти компьютера, выделенной для хранения объектов определенного типа).

В некоторых языках программирования (например, C++) строки создаются в программе как массивы символов. В других языках программирования (например, в языке Pascal) к перечисленным выше способам создания составных типов данных добавляются *множества* (неупорядоченные объединения объектов одного и того же типа), для которых определены операции добавления объекта во множество, исключения объекта из множества и проверки вхождения объекта во множество. Типом данных объекта множества может быть любой упорядоченный простой тип данных.

У структур (записей) существует разновидность — *союзы* (*объединения*), или *записи с вариантами*, поля которых выравниваются по одному начальному адресу памяти, что дает возможность обращаться к одному и тому же значению с помощью операций, разрешенных для разных типов данных, например:

```
// доступ к отдельным байтам представления целого числа
// язык Pascal (версия для системы Borland Delphi)
// определение типа – записи с вариантами
type Bytes=record
   case Boolean of
// поле для представления целого числа
   true:(Number: Integer;);
```

```
// поле для представления массива байтов, выделенных под целое
// число
   false:(Chars: array[1..SizeOf(integer)] of Byte;);
   end;
var x: Bytes; // переменная – запись с вариантами
i: Integer; // параметр цикла
. . .
// присвоение значения целому числу
   x.Number:=400;
   WriteLn('Представление в памяти целого числа',x.Number);
{вывод значений последовательности байтов, выделенных под целое
число }
   for i:=1 to SizeOf(Integer) do WriteLn(x.Chars[i]);
{в результате выполнения этого фрагмента программы на экран будет
выведено:
Представление в памяти целого числа 400
144
1
0
0
}
. . .
// язык C++
// определение типа – объединения
union Bytes {
// поле для представления целого числа
   int Number;
// поле для представления массива байтов, выделенных под целое
//число
   unsigned char Chars[sizeof(int)]; };
Bytes x; // переменная – объединение
. . .
// присвоение значения целому числу
x.Number=400;
printf("Представление числа % #x \n",x.Number);
for(int i=0;i<sizeof(int);i++)
/* вывод значений последовательности байтов, выделенных под целое
число, в шестнадцатеричной системе счисления */
printf("%#x\n",(unsigned)x.Chars[i]);
/* в результате выполнения этого фрагмента программы на экран будет
выведено:
Представление числа Ø x190
0x90
0x1
0
0
*/
. . .
```

Массивы и структуры относятся к *статическим* структурам данных, так как их размеры полностью определяются на этапе компиляции программы. Множества в языке Pascal и строки также являются статическими структурами, так как их размеры определяется на этапе компиляции с учетом максимально возможного в них числа элементов.

Представление в программе размещаемых в оперативной памяти динамических структур данных (таких, как списки или очереди) будет рассмотрено в подразд. 2.4.

Файлы представляют собой динамические структуры данных, поскольку их размер может изменяться при выполнении программы путем добавления или удаления объектов.

Для создания собственных составных типов данных в современных языках программирования используется механизм классов, который подробно будет рассмотрен в гл. 2.

При разработке прикладных программ в среде операционной системы Windows (для вызова функций из набора Windows API, обращении к полям системных структур и в других случаях) программисту наряду со стандартными типами данных, определенных в используемом языке программирования, приходится иногда использовать и стандартные типы данных Windows:

- `BOOL` (булевский тип данных, длина которого совпадает с длиной целого типа, а для представления констант — тип, использующий идентификаторы `TRUE` и `FALSE`);
- `BYTE`, `INT`, `WORD`, `DWORD` (разновидности целого типа без знака);
- `LPSTR`, `LPTSTR` (указатели на переменные — строки символов, оканчивающиеся нулем);
- `LPCSTR`, `LPCTSTR` (указатели на константы — строки символов, оканчивающиеся нулем);
- `WPARAM`, `LPARAM` (разновидности целого типа без знака, используемые для представления параметров сообщений Windows, см. подразд. 2.7);
- `POINT`, `SIZE`, `RECT` (структуры для представления координат точек, размеров и координат углов прямоугольных областей на экране);
- `HANDLE`, `HICON`, `HMENU`, `HDC` и т.д. (дескрипторы различных объектов Windows, см. подразд. 2.6).

1.4. Управляющие структуры, используемые при проектировании программ. Способы записи алгоритмов

Подобно структурам данных управляющие структуры, используемые при проектировании алгоритма решения задачи, также могут быть разделены на простые и составные (структурные).

К *простым управляющим структурам* (операциям, используемым для управления вычислениями внутри выражений, и операторам, определяющим порядок выполнения действий во всей программе) относятся те, которые не содержат внутри себя других операторов или ссылок на них. В современных языках программирования к простым относятся операторы присваивания и вызова подпрограммы. Они используются для изменения или отображения состояния оперативной памяти во время выполнения программы и служат составными частями для построения составных управляющих структур.

Составные управляющие структуры (*операторы*) представляют собой средства объединения простых и составных управляющих структур для получения новых операторов программы, которые, в свою очередь, могут включаться в состав более сложных операторов.

Современные языки программирования включают в себя следующие составные управляющие структуры:

• *последовательность* (иногда выступающая в виде *оператора блока* begin — end или {}), которая используется для определения последовательного выполнения операторов программы;

• *условный выбор* (if — then или if — then — else), который используется для организации разветвления в программе с возможностью двух альтернатив);

• *переключение* (case — of или switch), которое используется для организации разветвления в программе с возможностью нескольких альтернатив;

• *цикл с предусловием* (while — do), который используется для организации в программе повторяющихся вычислений, а проверка необходимости повторения вычислений выполняется до тела цикла);

• *цикл с постусловием* (repeat — until или do — while), который используется для организации в программе повторяющихся вычислений, а проверка необходимости повторения вычислений выполняется после тела цикла;

• *параметрический цикл* (for), который используется для организации в программе повторяющихся вычислений, зависящих от некоторого параметра, а число повторений, как правило, известно перед началом выполнения оператора цикла.

Хотя в современных языках программирования и определен *оператор безусловного перехода* в произвольную точку программы (go to), его использование настоятельно не рекомендуется, так как может (особенно для организации перехода вверх по тексту программы) существенно ухудшить ясность программы и негативно сказаться на ее надежности. При необходимости изменить в программе определенную в ней последовательность выполнения операторов следует использовать специальные формы оператора безусловного перехода:

• преждевременного завершения цикла или иного составного оператора при выполнении некоторого условия (break);

• преждевременного выхода из подпрограммы (exit или return);

• досрочного завершения тела цикла и перехода к следующей итерации (continue).

При проектировании алгоритма решения задачи могут применяться как графические, так и текстовые языки. Примером графического языка записи алгоритмов является *язык блок-схем*. Пример использования языка блок-схем для записи алгоритма Евклида для нахождения наибольшего общего делителя gcd положительных целых чисел a и b приведен на рис. 1.1.

Главным достоинством языка блок-схем алгоритмов, как и всех графических способов их записи, является наглядность. К недостаткам же можно отнести следующие особенности таких языков:

• невозможность (или неудобство) включения в блок-схему описаний объектов (типов данных, переменных), которые используются в алгоритме;

• потенциальная возможность создания сложных и запутанных блок-схем, не дающих ясного представления об алгоритме;

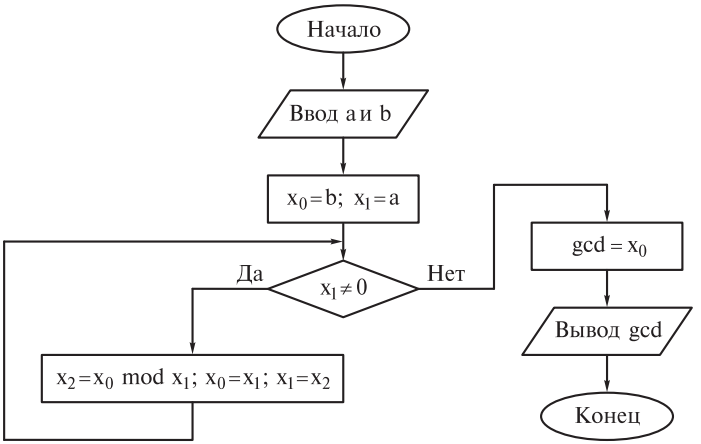

Рис. 1.1. Блок-схема алгоритма для нахождения наибольшего делителя

• сложность разбиения большой блок-схемы на отдельные фрагменты без потери наглядности;

• большая (по сравнению с другими способами записи алгоритмов) трудоемкость перехода от блок-схемы алгоритма к исходному коду программы.

Другим распространенным способом записи алгоритмов является так называемый *псевдокод*, объединяющий в себе понятность текстов на естественном языке и однозначность языков программирования. Далее приведен пример использования одного из вариантов псевдокода для описания алгоритма Евклида нахождения наибольшего общего делителя gcd двух целых положительных чисел a и b.

```
нач
   цел a, b, gcd;
   цел x0, x1, x2;
   ввод a, b;
   x0=b;
   x1=a;
   пока x1≠0 цикл
      x2=x0 mod x1;
      x0=x1;
      x1=x2;
   кцикл
   gcd=x0;
   вывод gcd;
кон
```

К достоинствам псевдокода следует отнести:

• возможность включения в запись алгоритма объявлений используемых в нем переменных;

• ясность записанного на нем алгоритма;

• простоту перехода к исходному коду программы;

• возможность разбиения записи сложного алгоритма на отдельные фрагменты без потери в понятности (соответствующий пример приведен в подразд. 1.5).

К недостаткам относится меньшая (по сравнению с графическими способами записи) наглядность записанного алгоритма.

1.5. Способы проектирования программ и их основные декомпозиционные структуры

В основе любого процесса проектирования сложного программного комплекса лежит метод декомпозиции (его разбиения на более простые составные части — компоненты, модули). К основным этапам этого процесса можно отнести:

1) проектирование архитектуры программного комплекса;

2) разработку внешних спецификаций для выделенных на первом этапе компонентов;

3) проектирование структур компонентов;

4) разработку спецификаций для структурных единиц (подпрограмм, классов, модулей), выделенных в каждом компоненте на третьем этапе проектирования;

5) проектирование структур данных и алгоритмов для выделенных в каждом компоненте структурных единиц.

Первые два этапа не являются обязательными и выполняются только при проектировании больших программных систем, которые могут быть разбиты на относительно независимые друг от друга компоненты (например, система программирования может быть разбита на текстовый редактор, подсистему управления файлами, компилятор, отладчик, справочную подсистему и т. п.).

На этапе проектирования архитектуры программного комплекса определяются:

1) функции, выполняемые каждым компонентом;

2) точные и однозначные сопряжения (интерфейсы) между компонентами;

3) структура передач управления между компонентами;

4) структура потоков данных;

5) иерархическая структура асинхронно (параллельно) выполняющихся процессов (если такое выполнение предусмотрено);

6) структура распределения оперативной памяти между компонентами;

7) структура использования компонентами разделяемых устройств (например, внешних коммуникаций).

Результаты проектирования архитектуры программного комплекса отражаются во внешних спецификациях его компонентов. Следующим этапом проектирования является проектирование структур компонентов программного комплекса. Цель этого этапа — определение всех составных частей компонента (будем называть их структурными единицами), их иерархии и интерфейсов между ними. Результаты выполнения этого этапа должны выражаться в виде спецификаций свойств структурных

единиц, на основании которых будет производиться проектирование их структур данных и алгоритма работы.

Как происходит проектирование структуры сложной программы (компонента большой программной системы)? Известны три основных подхода к решению этой задачи, три стратегии проектирования.

1. Стратегия проектирования «снизу вверх». После первоначального (во многом эвристического, основанного на личном опыте разработчиков) выделения структурных единиц начинаются их автономные и параллельные (с привлечением других участников проекта) проектирование, кодирование, тестирование и отладка. После этого выполняются комплексные тестирование и отладка всей программы. Основное преимущество этого подхода заключается в возможности ранней организации параллельной работы нескольких программистов. К главным недостаткам восходящего проектирования следует отнести:

- эвристичность (неформализуемость) начального этапа проектирования;
- принципиальную (неустранимую) сложность комплексной отладки программы, которая вызвана невозможностью исчерпывающе, полно определить интерфейсы между структурными единицами программы до начала их разработки;
- сложность внесения изменений в уже разработанную программу.

2. Стратегия проектирования «сверху вниз». В процессе проектирования программа разбивается на иерархически упорядоченные части (уровни), каждая из которых полностью определяет программу (точнее, ее промежуточную версию) и порождает (возможно) следующие иерархические уровни. На каждом уровне разработчик придерживается правил последовательной детализации сложного процесса и ограничения сложности каждого проектируемого сегмента. Декомпозиционный процесс продолжается до тех пор, пока сложность сегментов самого нижнего уровня проектирования позволит их непосредственно записать в нотации языка программирования.

К достоинствам стратегии проектирования «сверху вниз» следует отнести:

- отсутствие необходимости в комплексной отладке как таковой, так как спецификация сегмента определяется перед началом его проектирования;
- относительную простоту модификации программы, так как иерархическая организация позволяет точно определить, в какие сегменты следует внести требуемые изменения и как это скажется на функционировании других сегментов;
- потенциальную возможность осуществления тестирования и отладки параллельно с проектированием программы.

Недостатками нисходящей стратегии проектирования являются:

- более позднее распараллеливание работы по сравнению с восходящим проектированием;
- сложность и ответственность начального этапа проектирования, требующего от разработчика определенного опыта и высокой квалификации.

В качестве простого примера применения стратегии проектирования программы «сверху вниз» рассмотрим проектирование программы на-

хождения всех простых чисел в интервале от 2 до N с помощью «решета Эратосфена».

Первый шаг проектирования (B — массив простых чисел, p — проверяемое простое число и i — вспомогательная переменная):

```
нач
   цел B[N/2], p, i;
   B[0]=2;
   для i=3 до N шаг 2 цикл
      B[i/2]=i;
      кцикл
   p=3;
   пока p^2 ≤N цикл
      Удаление из B (замена 0) всех произведений p;
      p=следующее простое число после p;
      кцикл
   для i=0 до N/2-1 шаг 1 цикл
      если B[i]≠0 то вывод B[i];
      кесли
      кцикл
кон
```

Второй шаг проектирования — детализация сегмента «p= следующее число после p» (s — признак принадлежности p массиву B):

```
нач
лог s;
цикл
   p=p+1;
   s=false;
   для i=0 до N/2-1 шаг 1 цикл
      если B[i]=p то
         s=true;
            выход из цикла;
            кесли
         кцикл
   пока ¬s;
кон
```

Третий шаг проектирования — детализация сегмента «Удаление из B (замена 0) всех произведений p» (step и u — вспомогательные переменные):

```
нач
цел step, u;
step=p+p;
u=p*p;
пока u≤N цикл
   для i=0 до N/2-1 шаг 1 цикл
         если B[i]=u то
            B[i]=0;
               выход из цикла;
            кесли
         кцикл
      u=u+step;
   кцикл
кон
```

В реальных условиях нисходящая стратегия проектирования редко может быть применена, поскольку процесс проектирования программы, как правило, характеризуется регулярным возвращением на более ранние этапы (к более высоким уровням) для внесения необходимых изменений. Поэтому на практике чаще всего применяется сочетание стратегий «сверху вниз» и «снизу вверх» при ведущем положении нисходящего проектирования.

3. Стратегия «вертикального слоения». При использовании данной стратегии сначала создается каркас (скелет) программного комплекса с минимально необходимым набором функций, который в дальнейшем расширяется (как правило, с применением нисходящего проектирования, рис. 1.2).

На этапе проектирования программы могут использоваться специальные языки проектирования. В их качестве могут выступать языки программирования, в которых используются комментарии специального вида, определяющие свойства проектируемых структурных единиц программы. Примером языка проектирования такого рода может служить язык CLU (от cluster), в котором нисходящее проектирование программы поддерживается механизмами процедурной абстракции (абстрактной операции и абстрактного оператора) и абстракции данных (абстрактного типа данных).

Под *абстракцией* (или *абстрагированием*) обычно понимают метод проектирования, заключающийся во временном игнорировании «несущественных» подробностей реализации того или иного фрагмента программы.

В спецификации абстракции фиксируются только те аспекты решения, которые имеют непосредственное отношение к задаче. Последующие реализации абстракции (которых может быть несколько) должны быть согласованы именно по таким «существенным» аспектам, а в остальном могут отличаться. Иначе говоря, существенным является ответ на вопрос: что «делает» абстракция, а несущественным — как она это «делает».

Преимуществами поддерживаемой в языке CLU абстракции через спецификацию являются:

• локальность (каждая реализация может быть рассмотрена независимо от остальных, поэтому возможны использование абстракции без уяс-

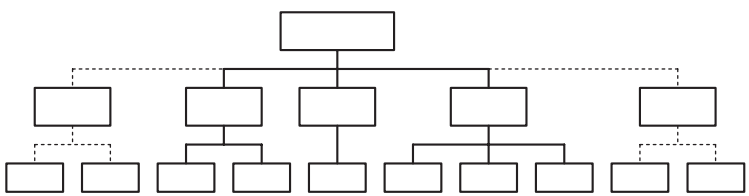

Рис. 1.2. Стратегия «вертикального слоения»:
штриховыми линиями показано добавление новых структурных единиц к каркасу программы

нения способа ее реализации и реализация без понимания деталей ее использования);

• модифицируемость (одна реализация может быть заменена другой без изменения других частей программы, что позволит повысить эффективность программы).

Другим примером языка проектирования является язык UML (Unified Modeling Language — унифицированный язык моделирования). Язык UML предназначен для спецификации, проектирования и документирования программных систем. Он упрощает сложный процесс проектирования программного обеспечения путем создания «чертежа» для построения системы. Поскольку язык UML опирается на методологию объектно-ориентированного анализа, более подробно он будет рассмотрен в подразд. 1.9.

После выделения в процессе проектирования программы (компонента программной системы) структурных единиц они должны быть реализованы в форме подпрограмм, сопрограмм, классов и модулей. Рассмотрим подробнее эти декомпозиционные структуры языков программирования.

Подпрограмма — это совокупность объявлений и операторов языка программирования, рассматриваемая как единое целое, имеющая имя и начинающая свое выполнение всегда с одной и той же точки (точки входа в подпрограмму). По месту их определения и использования подпрограммы подразделяются на внутренние (описанные внутри других подпрограмм) и внешние; по способу передачи результатов и способу вызова — на подпрограммы-операторы и подпрограммы-функции; по способу компоновки с вызывающей подпрограммой — на процедуры и макросы.

При вызове *процедуры* при выполнении программы управление передается в ее другую точку. После завершения выполнения операторами процедуры управление возвращается в точку программы, которая непосредственно следует за точкой вызова. При вызове *макроса* на этапе компиляции в код вызывающей подпрограммы в точке вызова включается макрорасширение (преобразованный путем подстановки параметров код макроса).

Использование макросов приводит к увеличению размера исполнимого кода программы, но уменьшает время ее выполнения за счет отсутствия накладных расходов, связанных с передачей управления и параметров в процедуру. Поскольку в макросах существуют ограничения на использование операторов выбора и цикла, эту форму наиболее целесообразно применять в тех случаях, когда подпрограмма состоит из нескольких последовательно выполняющихся операторов, а точка ее вызова расположена внутри цикла.

Среди языков программирования высокого уровня макросы наиболее часто используются в C и C++, например:

```
// Пример 1 — возведение в квадрат числа x
#define sqr(x) ((x)*(x))
. . .
int x,y;
. . .
```

```
y=sqr(x+1);
/* Пример 2 — преобразование в строку str и вывод сообщения mes на
русском языке в консольных приложениях Windows */
#define OutMsg(mes,str) CharToOem(mes,str);\
   printf(str);
. . .
char Msg[_MAX_PATH];
. . .
OutMsg("Воскресенье\n",Msg);
```

В отличие от процедур при вызове макросов происходит простая подстановка фактических параметров вместо указанных при определении макроса формальных параметров. Поэтому после вставки в текст программы макрорасширения возможно появление синтаксических ошибок в случае несоответствия типов данных в выражениях, а для соблюдения правильного порядка выполнения операций формальные параметры макросов целесообразно заключать в дополнительные круглые скобки (см. пример 1). Для перехода на следующую строку в теле макроса используется символ '\' (см. пример 2).

В языке C++ макросы могут быть созданы не только с помощью директивы препроцессора define, но и добавлением ключевого слова inline к прототипу любой функции, например:

```
inline char* cat_func(void);
```

По способу управления подпрограммы подразделяются на подпрограммы без параметров и подпрограммы с параметрами. Параметры в подпрограмму могут передаваться по значению и по ссылке.

При передаче параметра по *значению* при вызове процедуры значение фактического параметра копируется в специальную область памяти (стек) и адрес этой области передается в процедуру в качестве адреса соответствующего формального параметра. Это приводит к тому, что любые изменения параметров внутри тела процедуры не вызывают изменений значений фактических параметров в вызывающей программе. Фактическими параметрами, передаваемыми в процедуру по значению, могут быть любые выражения соответствующего формальному параметру типа.

Один или более последних элементов списка параметров могут быть объявлены как параметры со значением по умолчанию, например:

```
// язык Object Pascal
function P(x: Integer; y: Integer=0):Integer;
// язык C++
int f(int x, int *y=NULL);
```

В этом случае при вызове процедуры соответствующие фактические параметры могут быть опущены, а в теле процедуры им будут присвоены заданные в прототипе значения по умолчанию.

При передаче параметра *по ссылке* в процедуру передается адрес фактического параметра, что приводит к отражению любых изменений параметра внутри процедуры на значении фактического параметра в вызывающей программе. Фактические параметры, передаваемые в процедуру по ссылке, могут быть только именами переменных.

В языке Object Pascal для передачи параметра по ссылке перед его описанием в заголовке процедуры должно быть помещено ключевое слово var, например:

```
procedure swap(var x, y: Integer);
```

Если передача параметров по ссылке используется только из соображений экономии памяти (например, при передаче больших массивов или структур), то вместо ключевого слова var используется const. Если передача по ссылке используется для параметров, значения которых обязательно изменяются в теле процедуры, то вместо ключевого слова var может использоваться out (если перед началом выполнения процедуры значения таким фактическим параметрам не присвоены, то они всегда устанавливаются в 0).

В языке C++ для передачи параметров функций по ссылке используется ссылочный тип данных, например:

```
void swap (int& x, int& y);
```

Если изменение значения параметра в функции не предполагается, а передача по ссылке используется только для экономии памяти, то в прототипе функции перед описанием соответствующего параметра помещается ключевое слово const.

В некоторых случаях при проектировании программы необходимо иметь возможность передавать в процедуру параметры различных типов. Для реализации этой возможности используется передача «нетипизированных» параметров, например определяется и используется процедура сравнения двух областей памяти s и d произвольной длины size:

```
// язык Object Pascal
const N=10; // размер сравниваемых массивов
type
// тип данных, к которому преобразуются параметры
   Elements=array [1..MAXINT] of Byte;
// тип для определения массивов целых чисел длиной один байт
   Mas=array [1..N] of Byte;
function Equal(const s, d; size: Cardinal):Boolean;
var i: Cardinal; // параметр цикла
begin
i:=1;
while (Elements(s)[i]=Elements(d)[i]) and (i<=size) do Inc(i);
Result:=i>size;
end;
var
// сравниваемые массивы целых чисел
arr1, arr2:Mas;
// сравниваемые строки
str1, str2:String;
. . .
// вывод результата сравнения массивов
WriteLn(Equal(arr1,arr2,N));
// вывод результата сравнения строк
WriteLn(Equal(str1[1],str2[1],Length(str1)));
. . .
```

```
// язык C++
const N=10; // размер сравниваемых массивов
bool Equal(const void *s, const void *d, unsigned size)
{int i=0; // параметр цикла
   while(((unsigned char*)s)[i]==((unsigned
   char*)d)[i] && i<size) i++;
return i>=size; }
. . .
// сравниваемые массивы целых чисел длиной один байт
   unsigned char arr1[N], arr2[N];
// сравниваемые строки
   char str1[N], str2[N];
. . .
// вывод результата сравнения массивов
   printf("%d\n", Equal(arr1,arr2,N));
// вывод результата сравнения строк
   printf("%d\n", Equal(str1,str2,strlen(str1)));
. . .
```

При использовании в процедурах параметров без указания типа (в языке Object Pascal для этого применяется передача параметров по ссылке, а в языке C++ применяется передача указателя на произвольную область памяти) необходимо выполнять приведение этих параметров к нужному типу перед любой операцией над ними в теле процедуры.

Передача в процедуру параметров-массивов должна выполняться в форме передаваемых по ссылке так называемых открытых массивов (без указания размеров), например:

```
// язык Object Pascal
procedure f(var x: array of Integer);
// язык C++
void f(int x[], unsigned size);
```

Первый элемент такого массива всегда имеет индекс 0, а индекс последнего элемента вычисляется в теле процедуры с помощью встроенной функции High (язык Object Pascal) или с помощью обязательно передаваемого параметра, содержащего фактический размер массива (язык C++).

В языке Object Pascal фактическим параметром, передаваемым в процедуру вместо открытого массива, может быть выражение следующего вида:

```
[ список констант, разделенных запятыми ]
```

При передаче в качестве параметра процедуры многомерного массива «открытой» может быть только одна его размерность.

Для передачи в процедуру имени другой процедуры применяется *процедурный тип данных* (разновидность указателя), в определении которого указывается прототип соответствующих ему процедур, например:

```
// язык Object Pascal
const N=10; // размер массивов
// определение процедурного типа
type Func = function(x, y: Integer):Integer;
```

```
// определение подпрограмм
function Add(x, y: Integer): Integer;
begin
   Result:=sqr(x)+sqr(y);
end;
procedure Table(const x, y: array of Integer; out z: array of
Integer; f: Func);
var i: Integer; // параметр цикла
begin
   for i:=0 to High(x) do z[i]:=f(x[i], y[i]);
end;
// определение переменных
var Mas1, Mas2, Mas3: array [1..N] of Integer;
. . .
// вызов подпрограммы
   Table(Mas1, Mas2, Mas3, Add);
. . .
// язык C++
const N=10; // размер массивов
// определение процедурного типа
typedef int(*FUNC) (int, int);
// определение подпрограмм
int Add(int x, int y)
{
   return x*x + y*y;
}
void Table(int x[], int y[], int z[], unsigned size, FUNC f)
{
   for(int i=0; i<size; i++) z[i]=f(x[i], y[i]);
}
// определение переменных
int Mas1[N], Mas2[N], Mas3[N];
. . .
// вызов подпрограммы
   Table(Mas1, Mas2, Mas3, N, Add);
. . .
```

В языках C и C++ (в отличие от языка Pascal) фактические параметры при вызове процедуры помещаются в стек справа налево, т.е. на вершине стека всегда оказывается адрес первого параметра процедуры. Это позволяет создавать функции с переменным количеством параметров целого типа, например:

```
// вычисление суммы произвольного количества аргументов
#include <stdarg.h>
// size – количество суммируемых значений
int sum(unsigned size,…)
{int s=0; // сумма аргументов
va_list ap; // параметры переменной части списка
// "настройка" стека на начало переменной части списка параметров
   va_start(ap, size);
   for(int i=0; i<size; i++)
   {
// получение очередного параметра переменной части списка
```