С. В. СИНИЦЫН, А. С. МИХАЙЛОВ, О. И. ХЛЫТЧИЕВ

# ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ ВЫСОКОГО УРОВНЯ

## Учебник

*Рекомендовано*
*Учебно-методическим объединением по образованию*
*в области прикладной информатики в качестве учебника*
*для студентов высших учебных заведений, обучающихся*
*по специальности «Прикладная информатика (по областям)»*
*и другим экономическим специальностям*

ACADEMIA

Москва
Издательский центр «Академия»
2010

УДК 681.3.06(075.8)
ББК 32.973-018я73
С384

Рецензенты:
зам. заведующего кафедрой «Проблемы управления» МИРЭА, проф. *В. М. Лохин*;
проф. кафедры «Математическое обеспечение и администрирование информационных систем» Московского государственного университета экономики, статистики и информатики, канд. экон. наук, доцент *В. П. Грибанов*

**Синицын С. В.**

С384 Программирование на языке высокого уровня : учебник для студ. высш. учеб. заведений / С. В. Синицын, А. С. Михайлов, О. И. Хлытчиев. — М. : Издательский центр «Академия», 2010. — 400 с.

ISBN 978-5-7695-6673-8

Рассмотрены общие принципы высокоуровневого программирования, непосредственным образом связанного с изучением алгоритмов, в рамках технологического процесса. Излагаемый материал можно рассматривать как синтез трех аспектов: технологического, вычислительного и языкового. Технология определяет этапы решения задач, отвечает на вопрос, что программировать, позволяет оценить результаты. Алгоритмизация дает ответ на вопрос, как возможно решить поставленные задачи. Высокоуровневый язык обеспечивает программистов средствами реализации решения поставленных задач. При подготовке учебника использован опыт передовых предприятий, занимающихся промышленной разработкой программного обеспечения и специализирующихся на высокоточных программных системах.

Для студентов высших ученых заведений.

УДК 681.3.06(075.8)
ББК 32.973-018я73





**ISBN 978-5-7695-6673-8**

# ПРИНЯТЫЕ СОКРАЩЕНИЯ

АЛУ — арифметико-логическое устройство
БИС — большая интегральная схема
ЖЦ — жизненный цикл
ИС — интегральная схема
КЛП — корень — левое — правое
МП — микропроцессор
ОЗУ — оперативно-запоминающее устройство
ООП — объектно-ориентированное программирование
ОС — операционная система
ПЗУ — постоянно-запоминающее устройство
ПО — программное обеспечение
СБИС — сверхбольшая интегральная схема
СС — система сопряжения
УУ — устройство управления
ША — шина адреса
ШД — шина данных
ШУ — шина управления
ЭВМ — электронная вычислительная машина
ЯВУ — язык высокого уровня
API — Application Programming Interface
ASCII — American Standard Code for Information Interchange
COTS — Commercial Of-The-Shelf
ER — Entity-Relationship
IDEF0 — Integrated DEFinition method 0
SADT — Structure Analysis and Design Technique
SQM — Software Quality Management
SUT — Software Under Test
UML — Unified Modeling Language
V&V — Validation and Verification
WDT — Watch Dog Timer

## Что такое программирование и какова роль языка программирования в этом процессе

Давайте под программированием понимать процесс составления инструкций для их последующего выполнения формальным механизмом, именуемым в быту *компьютером*. В данном случае это электронное устройство, которое может хранить в своей памяти инструкцию (программу), выполнять в определенной последовательности отдельные части (команды) этой инструкции и изменять в соответствии с этими командами состояние своей памяти и, возможно, подключенных к этому компьютеру внешних устройств.

Как правило, конечной целью является не сама программа (инструкция), а тот эффект, который вызывает ее исполнение. Это может быть перевод самого компьютера в некоторое новое состояние (изменение состояния памяти), внешние эффекты, производимые внешними устройствами (звуки, краски на экране, печать символов на бумаге) и т. д. Часто это называют решением некоторой задачи: показ видеоролика, обмен сообщениями, начисление заработной платы, расчет орбиты космического аппарата и пр.

В этой трактовке процесс программирования и решение задачи можно сравнить с процессом приготовления пиццы. Представьте себе, что вам потребовалось накормить пиццей своих друзей. Исключим из рассмотрения собственно процесс кормления и поедания, а посмотрим, что ему предшествует, т. е. наша задача — приготовить пиццу.

Самый простой вариант — запросить доставку пиццы по телефону. Это предполагает, что в вашем городе имеется служба, которая обладает всеми необходимыми ресурсами и инструкциями по приготовлению и доставке готовой продукции. От вас требуется только определить параметры необходимого продукта (большая или маленькая, с грибами или с мясом, тип соуса и т. д.). Это можно рассматривать как программу (инструкцию), предполагается, что все остальное (сколько и чего класть, сколько запекать и прочее) ваш компьютер знает сам.

Другими словами, вам потребовался язык программирования с очень мощными командами (изготовить такую-то пиццу, привезти ее по такому-то адресу), параметры которых и определили

процесс решения задачи. А если у вас не столь продвинутый компьютер (в городе нет службы доставки пиццы, и вам придется готовить ее самому)? Что тогда?

Тогда ваша программа (инструкция) решения задачи будет намного длиннее. В свою очередь, команды ее будут проще. Они будут описывать, сколько взять муки, как приготовить начинку, в каком режиме запекать и т. п.

Таким образом, существенную роль в понимании программирования играет понятие *уровня языка*, на котором пишутся программы. Упрощенно можно сказать, что уровень языка тем выше, чем меньшим числом его команд можно выразить решение задачи. На нижнем уровне языков находятся *языки кодов* вычислительной машины. В конечном итоге именно она выполняет составленную программистом программу.

Следует иметь в виду, что и эти, так называемые *машинные языки* тоже отличаются по своему уровню. Так, в одной машине операция по вычислению суммы двух чисел может выглядеть следующим образом: взять первое число, прибавить к нему второе, записать результат, т. е. машина последовательно выполнит три команды своего машинного языка (кода). В другой машине для этого действия потребуется всего одна команда (сложить операнды с записью результата).

## Зарождение языков программирования

Для лучшего понимания текущего момента в области языков программирования давайте проследим пути их развития.

Первые программы составлялись программистами непосредственно на машинных языках (в кодах). Отсутствие единообразия в парке вычислительных машин не позволяло переносить программы, составленные для одной машины на другую, из-за различия в системе команд (отсутствие мобильности). Кроме того, процесс кодирования программы требовал от программиста детального знания нескольких десятков (100 — 200) машинных кодов и правил их применения, что требовало его длительного специального обучения. В результате процесс составления программы был чрезвычайно трудоемок, а ее понимание другим программистом — весьма затруднительным.

Первый шаг к автоматизации программирования был сделан по пути создания *машинно-ориентированного языка*, в котором коды команд были заменены их мнемоническими обозначениями. Например, не 17, а сл (сложить), не 37, а выч (вычесть) и т. д. Кроме того, вместо явного указания адресов операндов стало возможным использовать *мнемоничные названия* (идентификаторы).

Вычисление площади прямоугольника могло быть теперь записано в форме:

```
УМ ВЫСОТА,ШИРИНА,ПЛОЩАДЬ
```

Языки подобного рода стали называть *мнемокодами*, *автокодами*, *ассемблерами*. Последнее название чаще всего используется и сейчас. В ряде случаев создавались специализированные автокоды, инструкциям которых соответствовали десятки машинных команд. Программирование на подобных языках существенно сократило сроки подготовки программ и упростило обучение программистов.

Перевод с языка мнемонических команд в машинные коды стал выполняться специальными программами — *транслятора-ми*. Программа-транслятор отображает символические имена операндов в адреса памяти вычислительной машины и заменяет мнемонические названия операций на соответствующие машинные коды.

## Языки программирования высокого уровня

Следующим шагом к автоматизации программирования стало создание *машинно-независимого языка*. В его основу были положены формульные виды записи вычислений, напоминающие алгебраические выражения. Тем самым решались сразу три проблемы: переносимость (мобильность) программы с одной машины на другую, возможность взаимного понимания (общение) программистов и простота подготовки программистов (обучение).

К 1960 г. было изобретено несколько десятков языков подобного рода. Из всех стоит выделить три: Фортран (FORTRAN), Алгол (ALGOL) и Кобол (COBOL). Первый был характерен высокой популярностью и широкой поддержкой со стороны фирмы IBM. Его формульная запись программы и высокая эффективность трансляции Фортран-программы в машинный код снискала уважение у большинства инженеров-программистов.

Алгол был создан специальным комитетом с целью формирования компактного и понятного всем программистам языка. Его синтаксис послужил основой для большинства последующих языков программирования. Он (синтаксис) был настолько прост и точен, что быстро выдвинул Алгол на первое место в качестве языка публикации. Другими словами, с появлением Алгола большинство описаний алгоритмов решения задач стало публиковаться на нем.

Кобол тоже создавался специальным комитетом, но с другими целями. Задачей Кобола было установить возможность общения не только между программистами. Программы на Коболе должны были быть понятны для пользователей вычислительной техники, которые сами программы не пишут, но решают с их помощью свои прикладные задачи: начисляют зарплату, рассчитывают налоги, обмениваются информацией, распределяют ресурсы.

Любопытным в Коболе было и наличие средств описания свойств той гипотетической вычислительной машины, на которой должна была выполняться Кобол-программа. Это достигалось как наличием специального раздела программы — раздела оборудования, предназначенного для спецификации внешних устройств, так и специальными шаблонами описания данных. Последние не только позволяли определять разрядную сетку каждого данного, но и задавать необходимую точность вычислений (положение десятичной точки в мантиссе числа).

Таким образом, уже к началу 1960-х гг. в программистском сообществе сложилось довольно ясное представление о тех задачах, которые должен решать язык программирования высокого уровня, понимая под этим термином машинно-независимый язык:

1) автоматизировать процесс подготовки (кодирование) программ для вычислительной машины;

2) являться средством мышления (конструирования алгоритмов) программиста;

3) выступать средством общения программистов между собой, а не только между программистом и машиной;

4) быть максимально простым для изучения и способствовать процессу подготовки новых программистов;

5) позволять публиковать идеи решения задач на вычислительной машине и желательно в форме, пригодной для понимания широким кругом лиц, не имеющих специальной подготовки.

Третий и пятый пункты следует прокомментировать дополнительно.

## Общение и сопровождение

Некоторые исследователи утверждают, что программисты до 70 % своего времени тратят на общение друг с другом, а не с машиной. Если это действительно так, то язык программирования — это та среда, в которой один программист пытается передать свои мысли другому. Публикация своих размышлений в виде статьи или книги — тот же процесс общения. Специфика этой формы в том, что автор не в силах комментировать написанное в ответ

на вопросы читателя. Следовательно, опубликованный текст должен быть самодостаточен для понимания.

Приведем пример, упоминавшийся ранее, с расчетом площади прямоугольника. Сравним две формы записи:

```
A = B * C и ПЛОЩАДЬ = ВЫСОТА * ШИРИНА
```

Первая запись безусловно требует комментариев. Что такое **A**? Что такое **B**? Что такое **C**? Может, речь идет о числе банок, упакованных по **B** штук в **C** ящиков? Вторая практически понятна, если речь идет о прямоугольнике. Все будет еще понятнее, если язык допускает записи следующего вида:

```
ПЛОЩАДЬ ПРЯМОУГОЛЬНИКА = ВЫСОТА * ШИРИНА
ПЛОЩАДЬ ТРЕУГОЛЬНИКА = ВЫСОТА * ШИРИНА / 2
```

или

```
ПЛОЩАДЬ ТРЕУГОЛЬНИКА = ВЫСОТА * ОСНОВАНИЕ / 2
```

Таким образом, наш язык допускает использование имен объектов (параметров) программы произвольной длины. В ряде случаев и этого оказывается недостаточно. Требуются дополнительные комментарии к тем действиям, которые делаются в программе. Поэтому все современные языки программирования допускают включение в программу дополнительных строк — комментариев, не влияющих на выполнение самой программы, но являющихся ее частью. Более того, одним из критериев «хорошей» программы в настоящее время считается примерное равенство объема текста комментариев и самих вычислений. Еще лучше, если комментарии пишутся с соблюдением принятых правил их оформления, позволяющих впоследствии автоматизировать создание и сопровождение технической документации путем извлечения текста комментариев из исходного кода программы.

Еще одна проблема, которая решается языком программирования как средством общения (публикации), — сопровождение программ. Действительно, большинство программ эксплуатируются годами. За время их эксплуатации происходят изменения внешней среды, которые должны отразиться на их работе. Например, выходит новый закон, и способ вычисления налогов должен быть изменен. Вполне возможно, что автор программы давно покинул место своей работы, и новому программисту надо внести соответствующие изменения.

Новому человеку без помощи автора надо локализовать область, которая должна быть модифицирована, разработать способ изменения, соответствующий новым потребностям, произвести изменения и убедиться в их корректности. И во всем этом про-

цессе (процессе сопровождения) основным рабочим материалом является текст программы на языке программирования. Сложности, которые подстерегают программиста на этом пути, породили расхожее мнение, что легче написать свою новую программу, чем понять чужую.

## Эволюция языков программирования

Попытки создания языков, одинаково успешно решающих все пять задач, оказались неудачными. Отметим, однако, три из них, которые относятся к периоду 1960 — 1980 гг.

Первым упомянем язык Паскаль (Pascal). Автором его был Н. Вирт. Целью создания языка было решение четвертой задачи — обучения. Язык унаследовал от Алгола строгое описание синтаксиса, компактность. Ориентируясь исключительно на простые учебные задачи, Паскаль рассматривал структуру программы как нечто неделимое, целое. При этом он приобрел достаточно строгую семантику своих операторов, что сделало Паскаль не только широко распространенным языком обучения, но и популярным языком публикаций (средством решения пятой задачи).

Конечно, подобный успех языка привел к попытке его использования и для решения первой задачи — кодирования. Появилось множество реализаций «паскалеподобного» языка, обладающего модульной структурой программы. Эти реализации допускали эффективное использование архитектурных свойств вычислительной машины, но трудно назвать их удачными. Единственный язык, базирующийся на Паскале и широко используемый на сегодняшний день, — это Делфи (Delphi).

Еще одна интересная попытка создания универсального средства решения всех задач связана с языком Ада (Ada). Язык разрабатывался специальным международным комитетом по инициативе министерства обороны США. Ситуация, подтолкнувшая к попытке создания подобного языка, была обусловлена огромным объемом накопившихся программ, которые были написаны на сотнях языков (диалектах Алгола, Фортрана, Кобола, Паскаля и т. д.). Сопровождать все эти программы было чрезвычайно трудоемко.

Существенный вклад в понимание проблемы был сделан в подготовительный период работы комитета. Результатом его работы явилось формулирование и публикация требований к «*универсальному*» *языку* программирования. Далее в процессе конкурсного отбора выявился язык-победитель. Именно его доработка и привела к созданию языка, получившего название Ада.

По итогам работы комитета был создан язык со строгой типизацией данных, строго описанным, но, увы, далеко не компакт-

ным синтаксисом. Заказчиками языка предполагалось, что помимо задачи кодирования Ада должен взять на себя функции языка публикаций. Даже если эффективная реализация программы требовала использования другого языка кодирования (например, Фортрана), решение задачи должно было сопровождаться ее Ада-описанием.

Однако сложность восприятия текста на Аде (даже с учетом его строгости и точности) настолько затруднило процесс сопровождения программ, что попытку сделать Аду универсальным языком программирования следует считать неуспешной. Добавим к этому сложность обучения и низкую эффективность реализации некоторых механизмов Ады. В итоге поиски «хорошего» продолжались в области простых и мобильных языков программирования.

Одним из наиболее удачных продвижений в направлении разработки эффективного средства кодирования явилось создание языка Си (C). В большой степени его популярности способствовало успешное распространение переносимой операционной системы UNIX и ее аналогов, написанных практически полностью на языке Си.

Переносимость (мобильность) Си-программы — ее основное достоинство. Средства языка позволяют написать программу, ко-

Таблица В.1.1

**Выполнение фрагмента с шестью плюсами**

| Выполняемый фрагмент (выделен подчеркиванием) | Содержимое переменной **i** | Комментарий к выполнению фрагмента |
|---|---|---|
| `int i = 1` | 1 | Выделяется память для переменной и устанавливается заданное значение |
| `i += i++ + ++i` | 2 | Извлекается текущее значение переменной, после чего она увеличивается на 1 |
| `i += 1 + ++i` | 3 | Увеличивается значение переменной на 1 и извлекается новое значение |
| `i += 1 + 3` | 3 | Производится сложение значений операндов |
| `i += 4` | 7 | К текущему содержимому переменной прибавляется вычисленное значение |

торая будет одинаково выполняться на любой вычислительной машине, имеющий транслятор с языка Си.

К недостаткам языка можно отнести не всегда понятную семантику программы, обусловленную в том числе тем, что один и тот же символ может означать различные операции в зависимости от его места в тексте. Например, любопытно начинающему программисту предложить указать значение переменной `i` в следующем фрагменте программы:

Си:
```
int i = 1;
i += i++ + ++i;
```

Как показала практика, даже не все трансляторы правильно переводят подобную программу в машинный код. Проиллюстрировать выполнение данного фрагмента можно при помощи табл. В.1.1.

Конечно, мало кто из программистов напишет подобную программу, однако фрагменты текстов вида:

Си:
```
**p++ += a * b;
```

встречаются в реальных программах достаточно часто. И все-таки компактность языка и возможность его эффективной трансляции сделали Си очень популярным.

Часть I

# ТЕХНОЛОГИЯ ПРОГРАММИРОВАНИЯ

Глава 1

# ПРОЦЕСС СОЗДАНИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

## 1.1. Появление технологии программирования

В начале развития компьютерной индустрии проблема создания программ состояла в том, чтобы в принципе получить работающую программу, и лишь затем программисты добивались от нее необходимых результатов путем длительной отладки. В то время считалось невероятно сложным сразу написать правильно работающую большую программу, и очень много программ содержало ошибки. Однако в скором времени появилось множество больших программ, которые работали годами без каких-либо ошибок. Такой качественный скачок был связан со структурным программированием и правильно поставленным процессом разработки. В 1969 г. Э. Дейкстра опубликовал статью «Структурное программирование», положившую начало работам над методами программирования, которые в корне изменили представления о возможностях в области разработки программного обеспечения (ПО). Огромный вклад в развитие отечественной школы программирования, автоматизацию программирования внес академик А. П. Ершов [16]. Фактически процесс создания программ стал целой технологией, включающей в себя огромное число методов, процедур, процессов, правил и инструментов.

В ходе развития компьютерной техники сложность ПО росла, усложнялись требования к ПО, и соответственно росло время построения программных систем и численность команд разработчиков. Сформировались отдельные этапы создания ПО, такие как разработка требований, моделирование, проектирование, программная реализация, тестирование и верификация. Возникла сложная задача управления, которая могла быть разрешена путем построения методологии разработки, включающей в себя описание процесса, организации работ, методов и средств.

## 1.2. Жизненный цикл разработки ПО

Проектирование любой программной системы включает в себя несколько этапов или процессов, и чем лучше управление проектом, тем сильнее они выражены. Методология проектирования программных систем описывает процесс создания и сопровождения систем в виде жизненного цикла (ЖЦ) разработки, представляя его как некоторую последовательность этапов и выполняемых на них процессов. Для каждого этапа определяются состав и последовательность выполняемых работ, получаемые результаты, методы и средства, необходимые для выполнения работ, роли и ответственность участников и т.д. Такое формальное описание ЖЦ разработки позволяет спланировать и организовать процесс коллективной разработки и обеспечить управление этим процессом. Обобщенное представление о стадиях и процессах отражается в основных моделях процесса создания ПО.

Понятия жизненного цикла и модели жизненного цикла несколько отличаются, однако термин «жизненный цикл» часто используется в смысле «модели жизненного цикла». Модель жизненного цикла разработки ПО — структура, содержащая процессы, действия и задачи, которые осуществляются в ходе разработки, функционирования и сопровождения программного продукта в течение всей жизни системы, от определения требований до завершения ее использования.

На данный момент можно выделить следующие, часто упоминаемые и используемые модели жизненного цикла:

- каскадная модель;
- процессная или поэтапная модель с промежуточным контролем;
- итерационная модель;
- формальные преобразования;
- интеграция ранее созданных компонент.

### 1.2.1. Каскадная модель

Каскадная модель (или, как ее еще называют, водопадная) (рис. 1.1) рассматривает последовательное выполнение всех этапов проекта в строго фиксированном порядке. Эта модель происходит от структуры диаграммы Ганта для поэтапного процесса. Переход на следующий этап означает полное завершение работ на предыдущем этапе.

Результаты, полученные в ходе выполнения одного этапа, используются для выполнения следующего этапа.

Марри Кантор [19] отмечает ряд важных аспектов, характерных для каскадной модели.

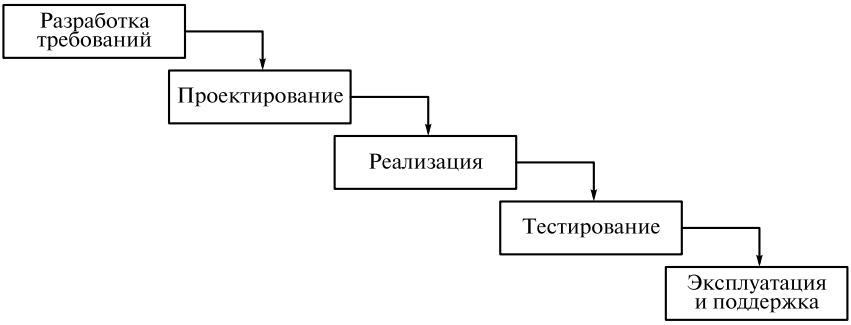


Рис. 1.1. Каскадная модель ЖЦ разработки

Каскадная схема включает несколько важных операций, применимых ко всем проектам:

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

При разработке относительно простых программных систем каждое приложение представляло собой единый, функционально и информационно независимый блок. Для разработки такого типа приложений эффективным оказался каскадный способ. Каждый этап завершался после полного выполнения и документального оформления всех предусмотренных работ.

В результате можно выделить следующие положительные стороны применения каскадного подхода:

- результат каждого этапа — законченный документ, отвечающий критериям полноты и непротиворечивости;
- заранее заданная последовательность этапов упрощает задачу планирования и позволяет вести контроль сроков завершения каждого этапа.

Каскадный подход хорошо зарекомендовал себя при построении простых систем, когда в самом начале разработки можно достаточно точно и полно сформулировать все требования к системе. Основным недостатком этого подхода является то, что реальный процесс создания сложной системы, как показывает практика, никогда полностью не укладывается в такую жесткую схему из-за большой динамики корректировок и уточнений, которые могут поступать как в результате дальнейшего развития проекта, так и извне, в качестве новых требований и уточнений [6]. Порой система настолько сложна, что невозможно четко и однозначно определить все требования на начальном этапе, поэтому постоянно возникает потребность в возврате к предыдущим этапам

и уточнении или пересмотре ранее принятых решений, которые в каскадной модели должны согласовываться и утверждаться в конце каждого этапа и оставаться неизменными на все время разработки. Это в конечном итоге часто приводит к получению не той системы, которая реально необходима.

В результате реальный процесс создания программной системы оказывается соответствующим поэтапной модели с промежуточным контролем.

Как утверждает Ф. Брукс в своей книге [6]: «Основное заблуждение каскадной модели состоит в предположениях, что проект проходит через весь процесс один раз, архитектура хороша и проста в использовании, проект осуществления разумен, а ошибки в реализации устраняются по мере тестирования. Иными словами, каскадная модель исходит из того, что все ошибки будут сосредоточены в реализации, а потому их устранение происходит равномерно во время тестирования компонентов и системы».

### 1.2.2. Процессная или поэтапная модель с промежуточным контролем

Чтобы получить возможность повторного выполнения любого этапа для внесения необходимых уточнений и исправлений, рассматриваются обратные связи между всеми этапами (рис. 1.2). Основной принцип контроля между этапами сохраняется.

Разработка ведется циклическими итерациями и прослеживается обратная связь между этапами, т.е. на любом этапе можно вернуться к любому предыдущему этапу и повлиять на результаты. Межэтапные корректировки позволяют учитывать существующие зависимости и влияния результатов различных этапов друг

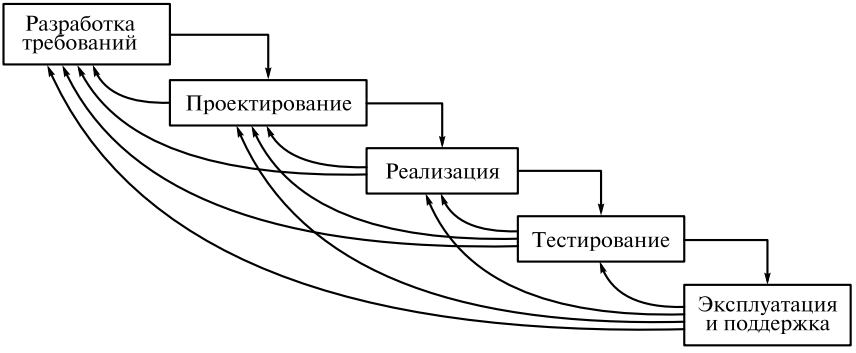


Рис. 1.2. Поэтапная или процессная модель ЖЦ разработки

на друга. Каждый этап длится весь период разработки и может рассматриваться как непрерывный процесс.

Одна из проблем процессной модели ЖЦ разработки заключается в невозможности оценить конечный результат до окончания всего процесса разработки ПО наряду с необходимостью внесения постоянных уточнений. Не всегда возможно внести очередные изменения, не видя того, что получается, а будущие пользователи охотнее вносят изменения и уточнения в ходе оценки какого-либо прототипа системы (который не строится в данной модели).

Специальным этапом проектирования может являться математическое моделирование будущей программной системы. Разработка математической модели заключается в выделении наиболее существенных аспектов поведения и свойств будущей программной системы с последующей их формализацией с применением выбранного математического аппарата. Основное назначение математической модели заключается в возможности исследования на ней поведения будущей системы до начала программной реализации. Анализ моделей может быть полезен для выявления возможных нетривиальных ошибок и дефектов будущей программной системы, которые весьма затруднительно обнаружить в ходе обычно тестирования уже разработанного программного обеспечения. Часто на моделях можно доказывать выполнимость (или невыполнимость) тех или иных свойств будущей системы. Безусловно, разработка и анализ математических моделей требуют времени и высокой квалификации программистов. При разработке простых программ и (или) выполнении небольших проектов риск отсутствия исследования моделей будущей программной системы может быть оправдан. Однако попытки сэкономить на моделировании при разработке действительно сложных распределенных программных систем практически всегда оборачиваются тем, что после разработки и начала опытной эксплуатации в ключевых аспектах поведения систем обнаруживаются существенные недостатки, исправление которых приводит как минимум к неоправданному затягиванию сроков окончания разработки. Мы более подробно рассмотрим некоторые методы математического моделирования и анализа в соответствующей главе данного учебника.

### 1.2.3. Спиральная модель

Основная идея спиральной модели заключается в том, что на этапах анализа и проектирования реализуемость технических решений и степень удовлетворения потребностей заказчика проверяется путем создания прототипов (рис. 1.3). Каждый виток спирали соответствует созданию работоспособного фрагмента или версии системы. Это позволяет уточнить требования, цели и ха-

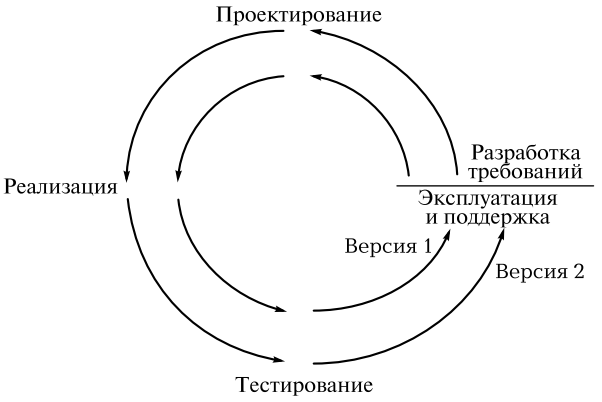


Рис. 1.3. Спиральная модель ЖЦ разработки

рактеристики проекта, определить качество разработки, спланировать работы следующего витка спирали. Таким образом, происходит последовательная конкретизация деталей проекта, и в результате выбирается обоснованный вариант, который удовлетворяет действительным требованиям заказчика и доводится до реализации.

Каждый виток спирали — это новая версия системы, созданная по уточненным требованиям. Очередная версия помогает понять реализуемость тех или иных решений, оценить качество получаемого продукта, сроки реализации различных решений. Процессу проектирования уделяется особое внимание, в то время как первые версии реализации могут быть лишь прототипами, обосновывающими выбранные архитектурные и технические решения.

Спиральная модель разработки ПО отражает итеративность разработки сложных систем. Каждый виток отражает появление очередной новой версии системы или нового прототипа, который появляется раньше окончания всех работ по проекту, что позволяет более тесно взаимодействовать с конечными пользователями, уточнять и дополнять требования, основываясь на работоспособном прототипе системы. Этот подход приближает конечную систему к реальным потребностям пользователей, которые не всегда с первого раза могут быть четко отражены в требованиях, но проявляются в ходе использования прототипа. Более того, использование прототипа может показать неэффективность или неудобство предлагаемых в требованиях схем работы до появления основной системы.

Спиральная модель не определяет четко фиксированных этапов, поэтому на каждом витке может быть использована любая