ФЕДЕРАЛЬНОЕ АГЕНТСТВО ПО ОБРАЗОВАНИЮ

ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«САМАРСКИЙ ГОСУДАРСТВЕННЫЙ АЭРОКОСМИЧЕСКИЙ
УНИВЕРСИТЕТ имени академика С.П. КОРОЛЕВА»

Л.С. ЗЕЛЕНКО

ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ И ПРОГРАММНАЯ ИНЖЕНЕРИЯ

Часть 1

Утверждено Редакционно-издательским советом университета в качестве учебного пособия

САМАРА
Издательство СГАУ
2006

УДК 004.4 (075)
ББК 32.97
З 484

Инновационная образовательная программа «Развитие центра компетенции и подготовка специалистов мирового уровня в области аэрокосмических и геоинформационных технологий»

Рецензенты: д-р техн. наук, проф. М. А. К о р а б л и н
д-р техн. наук, проф. С. А. П р о х о р о в

Л. С. Зеленко

З 484 **Технологии программирования и программная инженерия. Ч. 1:** учеб. пособие / *Л. С. Зеленко.* – Самара: Изд-во Самар. гос. аэрокосм. ун-та, 2006. - 96 с.: ил.

ISBN 5-7883-0480-6

В пособии рассматриваются основные понятия двух взаимосвязанных дисциплин «Технологии программирования» и «Программная инженерия», описываются принципы разработки сложных программных систем. Особое внимание уделено основным методологиям, на которых базируются технологии программирования, и моделям жизненного цикла программных систем. Также рассматриваются вопросы качества программных систем и показатели для оценки их работоспособности. Содержание учебного пособия является расширенным вариантом курса лекций, который читает автор на факультете информатики СГАУ.

Рекомендуется для использования в учебном процессе специальностей 010501 – «Прикладная математика и информатика», 010600 – «Прикладная математика и физика», а также студентам, обучающимся по специальностям, связанным с информатикой, информационными технологиями и разработкой прикладных программных систем.

УДК 004.4 (075)
ББК 32.97

ISBN 5-7883-0480-6

Оглавление

ПРЕДИСЛОВИЕ

Настоящее учебное пособие является расширенным вариантом курса лекций «Технологии программирования», который читается автором в течение последних лет на кафедрах информационных систем и технологий и программных систем СГАУ. Содержание курса сложилось в результате анализа многих научных источников, обобщения и методического осмысления опыта, который был накоплен специалистами по разработке программного обеспечения, а также с учетом опыта работы автора данного курса.

Появление в новом государственном образовательном стандарте (направления 510200 «Прикладная информатика и математика», 511900 «Информационные технологии», 552800 «Информатика и вычислительная техника») дисциплины «Технологии программирования» позволило в полном объеме рассмотреть все вопросы, связанные с разработкой сложных программных систем, до этого элементы технологий рассматривались лишь в курсе «Алгоритмические языки и программирование».

В настоящее время очень близко к технологиям программирования примыкает программная инженерия, поэтому в рамках данного учебного пособия параллельно рассматриваются и вопросы, связанные с этой дисциплиной.

Учебное пособие рассчитано на читателя, который имеет опыт программирования на языках высокого уровня (в том числе объектно-ориентированных) и разработки программ средней сложности. Его цель - помочь лицам, приступающим к разработке более сложных программных продуктов, рационально организовать свой программистский труд.

ВВЕДЕНИЕ

В современных условиях очень динамично развивается рынок интегрированных программных систем, которые позволяют автоматизировать деятельность предприятий и учреждений самого различного профиля (финансовых, промышленных, офисных) и самых различных размеров с разнообразными схемами иерархии, начиная от малых предприятий численностью в несколько десятков человек и завершая крупными корпорациями численностью в десятки тысяч сотрудников.

Быстрое увеличение сложности и размеров современных комплексов программ при одновременном повышении ответственности выполняемых функций резко повысило требования со стороны пользователей к их качеству, надежности функционирования и безопасности применения. Разработку сложных программных систем уже невозможно было вести «по старинке», без использования эффективных инструментальных средств и средств автоматизации всех этапов разработки. Время талантливых одиночек прошло, в разработке стали принимать участие большие коллективы программистов, работу которых необходимо было рационально организовывать.

Сегодня можно говорить о становлении новой отрасли, обеспечивающей **индустриальную реализацию процессов** разработки, сопровождения и изъятия из эксплуатации программного обеспечения (ПО). Исследования процессов разработки проектов программных систем показали, что во многих случаях стоимость и длительность их реализации значительно превышали предполагаемые, а характеристики качества не соответствовали требуемым, что наносило ущерб заказчикам, пользователям и разработчикам. Этот ущерб можно было значительно уменьшить, проведя своевременный анализ, прогнозирование и сократив риски возможного нарушения требований технических заданий и спецификаций на характеристики, выделяемые ресурсы и технологию создания конкретных комплексов программ.

При индустриальном подходе к разработке и сопровождению ПО особый вес приобретают **технологические характеристики** разрабатываемых программ, для получения качественных программных продуктов необходимо руководствоваться следующими принципами [1]:

- ***эффективностью*** – результаты должны отвечать заданным требованиям и стандартам в условиях ограниченных ресурсов;
- ***практичностью*** – результаты должны иметь конкретных заказчиков;
- ***фундаментальностью*** – результаты должны базироваться на знаниях фундаментальных наук;
- ***наследуемостью*** – результаты должны обобщать накопленный опыт, исключая деятельность «с нуля»;
- ***сопровождаемостью*** – результаты, находясь в эксплуатации, обязательно должны обслуживаться.

Начало работ в данной области относится к концу 60-х – началу 70-х годов, когда **рост сложности** программных систем (ПС) стал приводить к снижению качества их функционирования и появлению большого количества ошибок. Сложность ПС постоянно увеличивалась из-за:

- увеличения объемов кода (миллионы строк);
- увеличения количества связей между элементами систем;
- увеличения количества разработчиков (сотни человек);
- увеличения количества пользователей (сотни и тысячи).

Первые существенные результаты при решении данной проблемы были получены А. П. Ершовым, В. М. Глушковым, Е. А. Жоголевым, В. В. Липаевым, в работах которых описаны базовые методологии и технологии программирования того времени: структурное программирование и методология процедурной (алгоритмической) декомпозиции. К 90-м годам появилась новая парадигма в программировании, построенная на объектной декомпозиции предметной области, которая привела к методологии объектно-ориентированного анализа и проектирования. Появляется понятие **программной инженерии** как практического приложения научных знаний в проектировании и конструировании ПС, а также для создания документации, необходимой для их разработки, эксплуатации и сопровождения.

Основной концепцией программной инженерии стало понятие **жизненного цикла ПО.** Жизненный цикл рассматривается как совокупность всех действий, которые надо выполнить на протяжении всей «жизни изделия» (ПО). В настоящее время применяется несколько **моделей жизненного цикла**, которые отличаются набором фаз (этапов, стадий) проек-

та по созданию ПО, отдельных процессов, операций и задач. В настоящее время используются как классические модели жизненного цикла: каскадная (водопадная) и спиральная, так и их модификации, которые охватывают все этапы жизненного цикла ПО и успешно применяются для решения практических задач.

Только **скоординированное, комплексное применение в проектах** (с начала проектирования до внедрения программных систем) современных методов и промышленных технологий позволит достичь высокого качества, необходимого для использования их в сложных системах обработки информации.

1 ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ. ОСНОВНЫЕ ПОНЯТИЯ И ОПРЕДЕЛЕНИЯ

Новым видом современных изделий является ***программный продукт*** (ПП) – комплексы программ управления и обработки информации. Для эффективного производства программного продукта необходимо создание и внедрение современных технологий, позволяющих создавать сложные комплексы программ в сжатые сроки и при минимальных затратах труда [21].

Программирование – сравнительно молодая и быстро развивающаяся область науки и техники, технологии программирования начали развиваться сравнительно недавно (с середины 60-х годов), но к настоящему времени их можно считать самостоятельной областью прикладной науки. Хотя понятие «технология» в русском языке имеет ясное определение, понятие «технологии программирования» требует некоторого уточнения из-за необходимости определения, что следует считать продуктом этой технологии, а также из-за не всегда адекватного перевода данного термина на русский язык в литературе по программированию. Спектр существующих в настоящее время технологий достаточно широк, они постоянно совершенствуются и модифицируются на основе опыта реальных разработок, некоторые их них уходят в прошлое, некоторые «сращиваются», соединяя в себе наиболее важные достижения друг друга.

Процесс создания новой технологии может занимать как многие годы, так и несколько месяцев, все зависит от того, какие аппаратные и программные средства имеются в распоряжении программистов, какими методами и методологиями они владеют, какие инструментальные средства используют при разработке ПП.

1.1 Основные понятия и определения

Как уже говорилось, создание программ – исключительно трудоемкая работа, сопряженная с высокой квалификацией разработчиков и значительными затратами времени. Программирование можно считать новым типом интеллектуального *сплава науки и искусства*.

Примечание: Действительно, мастерство отдельных программистов восхищает, но даже самая совершенная сложная про-

грамма, по известному программистами афоризму, содержит хотя бы одну ошибку. Конечно, ошибок можно было бы избежать, если программирование опиралось бы только на науку, однако и в науке необходимо искусство, т.е. мастерство, умение, талант. Поэтому роль искусства в программировании весьма значительна.

Создание сложных программных систем должно быть основано на подходящих и весьма эффективных технологиях программирования, поддерживающих процесс программирования на всех этапах конструирования программ.

Прежде чем говорить о технологии программирования, необходимо хорошо разобраться в глубинной сущности понятий ***информации***, ***данных***, ***алгоритма, программы, программирования***, ибо сама технология не может быть оторвана от этих сущностей.

Понятия алгоритма и программы – одни их фундаментальных понятий современной информатики и кибернетики, однако точных и строгих математических определений для них не существует. Они рассматриваются как основополагающие первичные понятия (такие как точка, прямая, формула и т.д.), которые не определяются, а лишь интерпретируются, уточняются, разъясняются.

Алгоритм (algorithm) – заранее заданная последовательность четко определенных правил или команд для получения решения задачи за конечное число шагов (это набор предписаний, однозначно определяющий содержание и последовательность выполнения операций для достижения заданной цели или решения поставленной задачи).

Примечание: Принято считать, что сам термин ***алгоритм*** происходит от имени персидского математика Абу-Ждафара Мохаммеда Ибн-Мусы Аль-Хорезми, который в 825 г. описал правила выполнения арифметических действий в десятичной системе счисления. Понятие алгоритма стало предметом соответствующей теории – теории алгоритмов, которая занимается изучением общих свойств алгоритмов, какой-то период времени языки программирования назывались алгоритмическими.

Разработав алгоритм, можно перейти к программированию на выбранном языке. Программирование принято считать первичным, а программу – вторичной, так как программа появляется только после заверше-

ния процесса программирования. Но и здесь не все так просто, как кажется, поскольку легко усмотреть относительность и диалектичность связи *процесса* программирования с его результатами – программами.

Программа (program) – это набор операторов, который может быть представлен как единое целое в некоторой вычислительной системе и который используется для управления поведением этой системы.

Программирование (programming) *(в широком смысле)* – все технические операции, необходимые для создания программы, включая анализ требований и все стадии разработки и реализации.

Программирование *(в узком смысле)* – процесс кодирования и отладки программы в рамках реального проекта.

Целью программирования является описание процессов обработки данных (в дальнейшем – просто ***процессов***). Согласно IFIP (Международная федерация по обработке информации) и ICC (Международный вычислительный центр) ***данные*** (***data***) – это представление фактов и идей в формализованном виде, пригодном для передачи и переработки в некоем процессе, а ***информация*** (***information***) – это смысл, который придается данным при их представлении.

На термине «информация» стоит остановиться более подробно, т. к. он является основополагающим понятием информатики и имеет большое значение для наук, с ней связанных. Существует несколько определений данного понятия.

Основоположник кибернетики Н. Винер считал, что «Информация – это информация, а не энергия и не материя», и поставил информацию в один ряд с другими наиболее общими философскими категориями.

В словаре русского языка С.И. Ожегов [20] определил информацию как: 1) сведения об окружающем мире и протекающих в нем процессах; 2) сообщения, осведомляющие о положении дел, о состоянии чего-либо.

В законе «Об информации, информационных технологиях и защите информации» [11] дается следующее определение информации: «Информация – сведения о лицах, предметах, фактах, событиях, явлениях и процессах независимо от формы их представления».

Итак, данные участвуют в некотором ***информационном процессе (information process)*** или в обработке данных. ***Обработка данных (data processing)*** – это выполнение систематической последовательности дейст-

вий с данными. Данные представляются и хранятся на так называемых *носителях данных*. Совокупность носителей данных, используемых при какой-либо обработке данных, называется ***информационной средой (data medium)***. Набор данных, содержащихся в какой-либо момент в информационной среде, называется *состоянием* этой информационной среды. Поэтому *процесс* можно определить как последовательность сменяющих друг друга состояний некоторой информационной среды [10].

Описать процесс – это значит определить последовательность состояний заданной информационной среды. Формализованное описание процесса называется *программой,* с ее помощью информационный процесс на компьютере порождается автоматически. С другой стороны, программа должна быть понятной и человеку, так как и при разработке программ, и при их использовании часто приходится выяснять, какой именно процесс она порождает. Поэтому программа составляется на удобном для человека формализованном *языке программирования*, с которого она автоматически переводится на язык соответствующего компьютера с помощью *транслятора*.

Человеку (*программисту*), прежде чем составить программу на удобном для него языке программирования, приходится проделывать большую подготовительную работу по уточнению постановки задачи, выбору метода ее решения, выяснению специфики применения требуемой программы, прояснению общей организации разрабатываемой программы и многое другое. Использование этой информации может существенно упростить задачу понимания программы человеком, поэтому весьма полезно ее как-то фиксировать в виде отдельных документов (часто не формализованных, рассчитанных только для восприятия человеком). Таким образом, мы подходим к определению технологии программирования, которая позволяет овладеть процессами разработки программ, уточнить и упорядочить все действия, регламентировать и систематизировать их описание.

> ***Примечание:*** Хотя программирование – весьма специфический вид деятельности, вместе с тем технология программирования не может не иметь хотя бы несколько общих черт с технологиями другого рода, например, в машиностроении, легкой промышленности и т.д. Поэтому дадим сначала определение технологии вообще.

Технология (technology) – совокупность производственных процессов в определенной отрасли производства, а также научное описание способов производства, это совокупность технологических элементов (средств, устройств, методов, приемов, документов), используемых для обработки исходных материалов с целью получения конечной продукции.

Технология программирования (programming technology) – это совокупность методов и средств, используемых в процессе разработки программных продуктов, представляет собой набор технологических инструкций, включающих в себя:

- указание последовательности технологических операций;
- перечисление условий, при которых выполняется та или иная операция;
- описание самих операций, где для каждой операции выделены исходные данные, результаты, а также инструкции, нормативы, стандарты, критерии и методы оценки и т.п. (рисунок 1) [6].

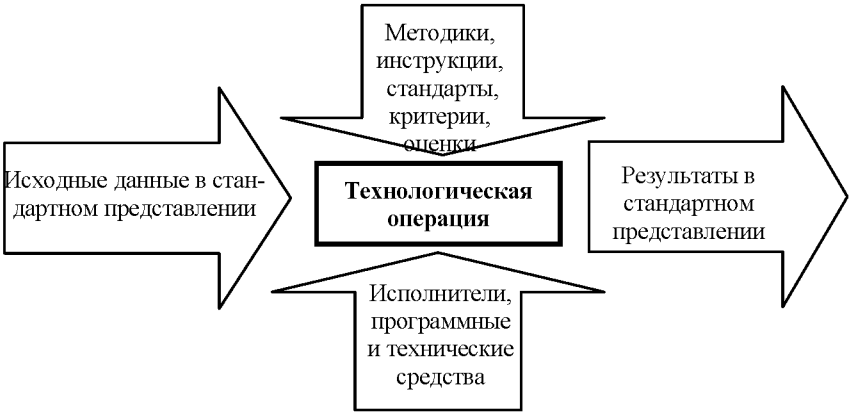


Рисунок 1 - Общая схема технологической операции

Примечание: Эффективность технологий отражается непосредственно на затратах совокупного общественного труда, разработка сложных ПП требует больших затрат и происходит в условиях ограниченных ресурсов. Поэтому необходимо осуществлять баланс между достигаемым качеством и ресурсами, которые требуются для реализации конкретного проекта. Кроме того, разработка ПП характеризуется высокой долей творческого труда, особенно на начальных и завершающих этапах.

Отсюда принципиальной особенностью современных технологий является активное участие руководителей проекта в создании концепций и планов на базе прототипов завершенных разработок.

Непрерывное увеличение объемов и сложности программных комплексов, а также рост требований к их качеству привели к тому, что простейшие технологии программирования небольших программ в настоящее время развились в сложные технологии ***проектирования***, ***разработки*** и ***сопровождения*** интегрированных комплексов программ (рисунок 2).

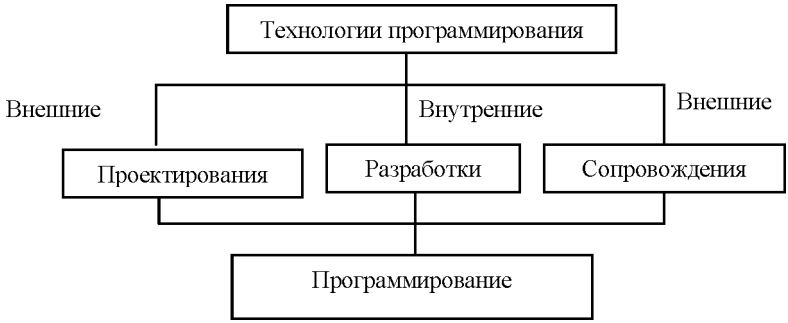


Рисунок 2 – Составные части технологии программирования

Любая технология имеет две стороны:

- принципиальную (внутреннюю);
- организационно-производственную (внешнюю).

Такое же положение и в области программирования: *внутренняя технология первична*, а *внешняя – вторична* (это в чем-то напоминает философскую связь между содержанием и формой). Без внутренних новинок, присущих только данной технологии, не может быть технологии вообще. Новая идея о том, как конструировать программы, образует тот генетический материал, который в дальнейшем при благоприятных условиях может привести к новой технологии. Такая идея не появляется из ничего, она, как правило, – сложный результат изучения действительного положения вещей, проникновения в такие сущности, которые раньше «не просматривались».

Примечание: Часть идей ведет к созданию действительно новых способов мышления и программирования, даже не связанных с конкретным языком. В этом случае говорят о ***моделях (model)*** или ***парадигмах (paradigm) программирования,*** которые могут послужить основой самостоятельных языков программирования.

Технология программирования определяет способ описания проектируемого ПП, точнее модели, используемой на конкретном этапе разработки. Поэтому различают *технологии*, используемые на конкретном этапе разработки или для решения отдельных задач этих этапов, и *технологии*, охватывающие несколько этапов или весь процесс разработки. В основе первых, как правило, лежит ограниченно применимый метод для решения конкретной задачи. В основе вторых – базовый подход, определяющий совокупность методов, или ***методологию***.

Не следует путать технологию программирования с методологией. В технологии программирования методы рассматриваются «сверху» – с точки зрения организации технологических процессов, а в методологии программирования методы рассматриваются «снизу» – с точки зрения основ их построения.

Методология программирования – совокупность механизмов, применяемых в процессе разработки программного обеспечения и объединенных одним общим философским подходом [4][1].

В литературе широко используется близкое к технологии программирования понятие ***программной инженерии (software engineering)***, определяемой как систематический подход к разработке, эксплуатации, сопровождению и изъятию из обращения программных средств [9][2]. Главное различие между технологией программирования и программной инженерией как дисциплинами для изучения заключается в способе рассмотрения и систематизации материала. В технологии программирования акцент делается на изучении процессов разработки ПП (*технологических процессов*) и порядке их прохождения – методы и инструментальные средства разработки ПП *используются* в этих процессах (их применение и образует

[1] В кн. Буч Г. Объектно-ориентированное проектирование с примерами применения. С. 25
[2] В кн. Дзержинский Ф.Я., Калиниченко И.М.. Дисциплина программирования: концепция и опыт реализации методических средств программной инженерии. С. 9-16

технологические процессы). Тогда как в программной инженерии изучаются различные методы и инструментальные средства разработки ПС с точки зрения достижения определенных целей – эти методы и средства могут использоваться в разных технологических процессах (и в разных технологиях программирования) [36].

Подводя итог всему вышесказанному, рассматривая ту или иную технологию программирования, мы будем:

- рассматривать все процессы разработки ПС, начиная с момента возникновения замысла ее создания до написания необходимой документации (программной, системной, для пользователя);
- рассматривать не только вопросы построения программных конструкций и выбора структур данных, но и вопросы описания функций и принимаемых решений с точки зрения их человеческого (неформального) восприятия;
- использовать ряд базовых принципов (о них будет идти речь позже) для достижения поставленной цели – выпуска надежной ПС.

Такой взгляд на технологию программирования будет существенно влиять на организацию технологических процессов, на выбор в них методов и инструментальных средств.

1.2 Технологии программирования: этапы развития и базовые методологии программирования

Если попытаться охарактеризовать современный уровень развития компьютерных и информационных технологий, то первое, на что следует обратить внимание, – это возрастающая сложность не только отдельных физических и программных компонентов, но и лежащих в основе этих технологий концепций и идей [15].

Чтобы разобраться в существующих технологиях программирования и определить основные тенденции их развития, целесообразно рассмотреть эти технологии в историческом контексте, выделяя основные этапы развития программирования как науки. Сделаем краткую характеристику развития программирования по десятилетиям.

В ***50-е годы*** мощность компьютеров (первого поколения) была невелика, а программирование для них велось в основном в машинном коде. Программирование было искусством: программисту необходимо было

отслеживать не только последовательность выполняемых операций, но и местоположение данных при программировании (рисунок 3).

Рисунок 3 – Структура первых программ

Примечание: В соответствии с принципами *фон Неймана* программы и данные хранились в одной и той же памяти (*концепция хранимой программы*) и выполнялись последовательно на одном процессоре.

В это время решались главным образом научно-технические задачи (расчет по формулам), задание на программирование содержало, как правило, достаточно точную постановку задачи. Появление ассемблеров, а затем и языков программирования высокого уровня (Fortran, Basic) упростило программирование вычислений, снизив уровень детализации операций, и позволило повысить уровень сложности программ. В этот период использовалась интуитивная технология программирования: почти сразу приступали к кодированию программы, при этом задание могло несколько раз изменяться (это сильно увеличивало время ее разработки), минимальная документация оформлялась уже после того, как программа начинала работать. Тем не менее, именно в этот период родилась фундаментальная для технологии программирования *концепция модульного программирования*, ориентированная на преодоления трудностей программирования в машинном коде.

В ***60-е годы*** можно было наблюдать бурное развитие и широкое использование языков программирования высокого уровня (Fortran, Algol-60, PL/1), значение которых в технологии программирования явно преувеличивалось. Появление в этих языках средств реализации механизма подпрограмм стало революционным. Это позволило создать большие библиотеки расчетных и служебных подпрограмм, которые можно было сохранять и затем использовать в разных программах.

Примечание: Языки программирования первоначально делали упор на ***синтаксические аспекты*** и проблемы записи про-

грамм, т.е. на *форму*. Это значительно увело в сторону от насущных проблем ***семантики***, т.е. *содержательного смысла* анализа и синтеза программ, что не прошло бесследно. Надежда на то, что языки программирования высокого уровня решат все проблемы, возникающие в процессе разработки больших программ, не оправдалась.

Первоначально программа состояла из основной программы, области глобальных данных и набора подпрограмм, выполняющих обработку либо всех данных, либо только ее части (рисунок 4). Такая архитектура повышала вероятность искажения части глобальных данных какой-либо подпрограммой, поэтому для повышения надежности программ была предложена новая идея: использовать часть данных как локальные внутри подпрограмм (рисунок 5).

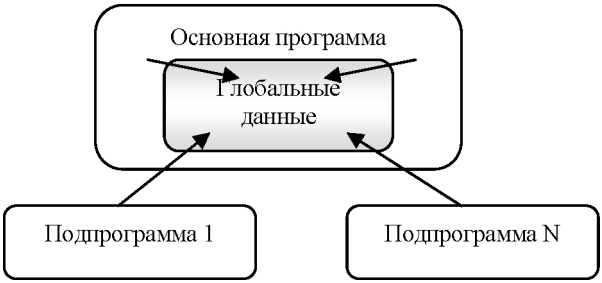
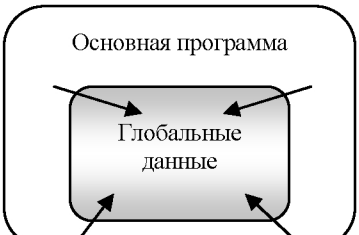


Рисунок 4 – Архитектура программы с глобальной областью данных

Основной методологией того времени стала *методология процедурно-ориентированного программирования*, в основе которой лежала процедурная или алгоритмическая организация структуры программного кода.

Примечание: Это было настолько естественно для решения вычислительных задач, что ни у кого не вызывала сомнений целесообразность такого подхода. Эта идея оказалась настолько жизнеспособной, что долгое время доминировала над всем процессом разработки программ.



Разработка больших программ превратилась в серьезную проблему, для снижения количества ошибок и повышения надежности программную систему стали разбивать на более мелкие фрагменты. Основой для такого разбиения стала *процедурная (алгоритмическая) декомпозиция*, при которой отдельные части программы или модули представляли собой совокупность процедур для решения некоторой совокупности задач.

В середине 60-х годов в языках программирования появилась специальная синтаксическая конструкция, и закрепилось новое понятие ***процедуры (procedure)***. Главная особенность *процедурного программирования* заключается в том, что программа всегда имеет начало во времени и окончание, для начала действий последующей процедуры необходимо завершение всех действий предшествующей процедуры.

С появлением транзисторов, а затем и интегральных схем, стоимость компьютеров резко снизилась, а их производительность росла почти экспоненциально. Появление в компьютерах 2-го поколения прерываний привело к развитию мультипрограммирования и возможности создания больших программных систем. Широко стала использоваться коллективная разработка, которая поставила ряд серьезных технологических проблем, разразился «кризис программирования»: фирмы, взявшиеся за разработку сложных ПС, срывали сроки их завершения (некоторые проекты так никогда и не были завершены), существенно увеличивалась стоимость таких проектов (стоимость программного обеспечения стала приближаться к стоимости аппаратуры).

Объективно это было вызвано несовершенством технологии программирования. Применяемый в это время подход разработки сложных ПС ***«снизу-вверх»***, при котором вначале разрабатывались самые простые подпрограммы, а затем из них конструировались более сложные, привел к тому, что при согласовании подпрограмм выявлялось большое количество ошибок, на устранение которых требовалось очень много времени. Процесс тестирования и отладки стал занимать больше времени, чем процесс кодирования (по оценкам некоторых специалистов, время «сборки» ПС стало занимать до 80% всего времени на разработку). Это стало началом серьезных размышлений над методологией и технологией программирования, стал развиваться новый подход к программированию, который был назван ***структурным (structured programming)***.

> ***Примечание:*** Появление и интенсивное использование условных операторов и оператора безусловного перехода goto стало предметом острых дискуссий среди специалистов по программированию. Дело в том, что бесконтрольное применение в программе оператора безусловного перехода goto способно серьезно осложнить понимание кода. Ситуация казалась настолько драматичной, что в литературе зазвучали призывы исключить оператор goto из языков программирования. Именно с этого времени принято считать хорошим стилем программирования — программирование без goto [15].

В ***70-е годы*** получили широкое распространение информационные системы (ИС) и базы данных (БД), так как к середине 70-х годов стоимость хранения одного бита информации на компьютерных носителях стала меньше, чем на традиционных носителях. Это резко повысило интерес к компьютерным системам хранения данных. Базовой методологией стала методология структурного программирования, в основе которой лежит *процедурная декомпозиция* (разбиение на части) ПС и организация отдельных модулей в виде совокупности выполняемых процедур.

> ***Примечание:*** В отличие от процедурного, структурный подход требовал представления задачи в виде некоторой иерархии подзадач, для достижения общей идеи проектирование системы должно было вестись по принципу ***«сверху-вниз»*** с применением метода *пошаговой детализации*.

Появились языки программирования 3-го поколения ***(Clu, Pascal, Modula-2)***, отличительной особенностью которых явилось наличие развитых средств *абстрагирования типов* для структурирования данных. Появление пользовательских типов позволило уменьшить количество ошибок при работе с глобальными данными. Стала развиваться технология ***модульного программирования (modular programming)***, которая предполагает объединение нескольких подпрограмм, использующих одни и те же данные, в отдельно компилируемые модули, связи между которыми устанавливались через специальный интерфейс, а доступ к реализации регулировался механизмами импорта-экспорта путем соответствующих деклараций (рисунок 6).

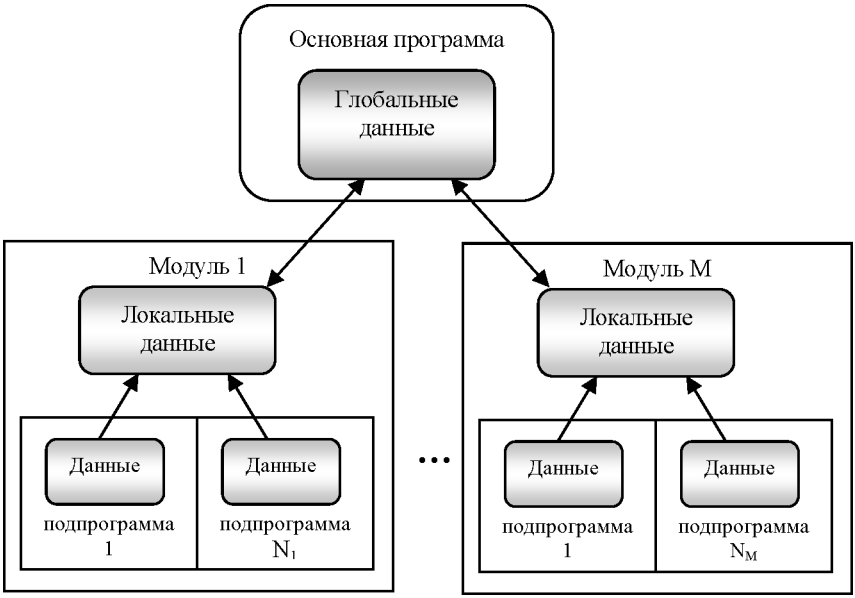


Рисунок 6 –Архитектура модульных программ

Примечание: Модульное программирование стало удачной основой для реализации ***инкапсуляции*** и механизма ***импорта-экспорта***. Инкапсуляция позволила рассматривать модуль как программный эквивалент определенного класса объектов, со-

держащий в себе всю информацию об объектах этого класса[14][3].

Таким образом, технология программирования в это время интенсивно развивается в следующих направлениях:

- широкое внедрение нисходящей *функциональной* разработки ПС и структурного программирования;
- развитие абстрактных типов данных и модульного программирования (в частности, возникновение идеи разделения спецификации и реализации);
- исследование проблем обеспечения *надежности* и *мобильности* ПС;
- создание методики управления коллективной разработкой ПС;
- появление инструментальных программных средств (программных инструментов) поддержки технологии программирования.

80-е годы характеризуются широким внедрением персональных компьютеров во все сферы человеческой деятельности и тем самым созданием обширного и разнообразного контингента пользователей ПС, что, в свою очередь, сместило акценты на решаемые задачи (на первый план стали выходить задачи обработки и манипулирования данными). Стало очевидным, что традиционные методы процедурного программирования не способны справиться ни с растущей сложностью программ и их разработки, ни с необходимостью повышения их надежности. Во второй половине 80-х годов возникла настоятельная потребность в новой методологии программирования, которая была бы способна решить весь этот комплекс проблем. Такой методологией стало ***объектно-ориентированное программирование*** (ООП) [4].

В объектно-ориентированном подходе к разработке программ центральным является понятие класса объектов. ***Класс (class)*** определяется как множество объектов, обладающих внутренними (имманентными) свойствами, которые играют роль *классообразующих признаков* и присущи любому объекту класса. Классы образуют иерархию с *наследованием свойств*, основанных на таксономических моделях обобщения [14].

[3] В кн. Кораблин М.А. Программирование, ориентированное на объекты. С.6.

Примечание: Механизм наследования свойств в ООП позволяет повысить лаконичность программ путем использования деклараций «класс-подкласс» и их надежность (любой подкласс может быть разработан на основе уже созданного и отлаженного надкласса). Использование данного механизма непосредственно связано с возможностью расслоения *свойств предметной области*, для которой разрабатывается ПС, и определения отношения «класс-подкласс» (для многих областей определение таких отношений проблематично).

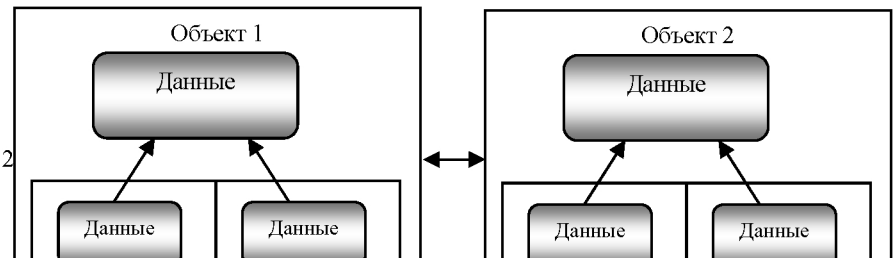
Отличительной особенностью ООП является организация взаимодействия объектов путем *посылки сообщений* (рисунок 7). Разработка новых архитектурных решений, адекватных этой концепции, связана с созданием многопроцессорных ЭВМ, хотя может быть решена и на однопроцессорных ЭВМ с помощью механизмов, обеспечивающих «логический» параллелизм: сопроцессов, событийных взаимодействий, метода дискретно-событийного управления и т.п.

Основными принципами (парадигмами) ООП являются:

- ***инкапсуляция (encapsulation)*** – объединение в классе данных (*свойств*) и *методов* (процедур обработки), сокрытие отдельных деталей внутреннего устройства классов от внешних по отношению к нему объектов или пользователей;
- ***наследование (inheritance)*** – возможность вывода нового класса из старого с частичным изменением свойств и методов;
- ***полиморфизм (polymorphism)*** – определение свойств и методов объекта по контексту (полиморфизм подразумевает отделение идеи «что делать» от ее воплощения внутри иерархии класса объектов «как делать»).

Широкое распространение методологии ООП оказало влияние на процесс разработки программ:

- появляются языки программирования (например, Ада, Object Pascal, C++), учитывающие требования технологии программирования;



- процедурно-ориентированная декомпозиция программ уступила место объектно-ориентированной декомпозиции, при которой отдельными структурными единицами программы стали являться не процедуры и функции, а классы и объекты с соответствующими свойствами и методами;
- программа перестала быть последовательностью предопределенных на этапе кодирования действий, а стала событийно-управляемой (это обстоятельство стало доминирующим при разработке широкого круга современных приложений);
- развиваются методы и языки спецификации ПС, начинается бурный процесс стандартизации технологических процессов и, прежде всего, документации, создаваемой в этих процессах [18];
- создаются различные инструментальные среды разработки и сопровождения ПС.

Середина 80-х годов считается «переломной точкой» в технологии программирования (проектирования) ПС – появилась концепция автоматизированной разработки программного обеспечения ***(Computer Aided Software/System Engineering***, **CASE**). Предпосылками появления и развития CASE-технологий явилось изменение масштабов создаваемых ПС: рост сложности их функционирования и разработки, а также изменение масштабов показателей качества ПС. Появление первых CASE-средств было встречено с определенной настороженностью. Ранние CASE-средства были простой надстройкой над некоторой системой управления базами данных (СУБД), графическая нотация, реализованная в том или ином CASE-средстве, не имела строгого синтаксиса (в отличие от языков программирования), и попытки предложить подходящий синтаксис для визуального представления концептуальных схем БД были восприняты далеко неоднозначно.

В ***90-х годах*** продолжается бурное развитие технологий программирования, основанных на объектном подходе. Были созданы *среды визуального программирования* (Borland Delphi, Visual C++, C++ Builder и т.д.), с помощью которых можно спроектировать интерфейс ПС. Но наиболее существенным обстоятельством в развитии методологии ООП явилось осознание того факта, что процесс написания программного кода может быть отделен от процесса проектирования структуры программы, связанного с общим анализом требований к будущей системе, а также с анализом конкретной предметной области, для которой она разрабатывается. Появилась специальная методология, получившая название методологии ***объектно-ориентированного анализа и проектирования (ООАП).***

> ***Примечание:*** Необходимость анализа *предметной области* до начала написания программы была осознана давно, при разработке масштабных проектов и баз данных (при проектировании БД возникает необходимость в предварительной разработке концептуальной схемы, которая отражала бы общие взаимосвязи предметной области и особенности организации соответствующей информации).

Под ***предметной областью (application domain)*** принято понимать ту часть реального мира, которая имеет существенное значение или непосредственное отношение к процессу функционирования программы. Дру-

гими словами, предметная область включает в себя только те объекты и взаимосвязи между ними, которые необходимы для описания требований и условий решения некоторой задачи [36]. Для выделения или идентификации компонентов предметной области было предложено несколько способов и правил, а сам процесс получил название *концептуализации предметной области*. При этом под ***компонентом*** понимают некоторую абстрактную единицу, которая обладает функциональностью, т. е. может выполнять определенные действия, связанные с решением поставленных задач.

> ***Примечание:*** Появление методологии ООАП потребовало, с одной стороны, разработки различных средств концептуализации предметной области, а с другой — соответствующих специалистов, которые владели бы этой методологией. На данном этапе появляется относительно новый тип специалиста, который получил название *аналитика* или *архитектора*. Наряду со специалистами по предметной области аналитик участвует в построении концептуальной схемы будущей ПС, которая затем преобразуется программистами в код. При этом отдельные компоненты выбираются таким образом, чтобы при последующей разработке их было удобно представить в форме классов и объектов. В этом случае немаловажное значение приобретает и сам язык представления информации о концептуальной схеме предметной области.

Разделение процесса разработки сложных программных приложений на отдельные *этапы* способствовало становлению концепции ***жизненного цикла (ЖЦ) программы (SLC – Software Lifetime Cycle),*** при этом каждый из них имеет свои методы и средства автоматизации [17][4]:

1. Системный анализ предметной области, планирование, организация и управление разработкой различных версий ПС;
2. Разработка и накопление программных и информационных компонент для их многократного применения в определенной проблемно-ориентированной области создания ПС;
3. Сборка, отладка и испытания базовых версий ПС из подготовленных программных и информационных компонент;

[4] В кн. Липаев В. В. Отладка сложных программ: Методы, средства, технология. С.21

4. Модификация и развитие версий ПС, а также состава их компонент для изменения и расширения характеристик базовых версий ПС.

С ***середины 90-х годов*** (и до сегодняшнего времени) CASE-средства становятся основным инструментом, поддерживающим автоматизацию на всех этапах: они обеспечивают унификацию процессов моделирования, автоматизированный анализ системных требований и выработку первичных требований к ПС, обеспечивают конфигурационное управление проектами, модификацией и развитием версий. Постепенно методология ООАП перерастает в ***методологию системного анализа и системного моделирования***.

Методология системного анализа служит концептуальной основой системно-ориентированной декомпозиции предметной области. В этом случае исходными компонентами концептуализации являются системы и взаимосвязи между ними. При этом понятие ***системы*** является более общим, чем понятия классов и объектов в ООАП [36]. Результатом системного анализа является построение некоторой ***модели (model)*** системы или предметной области.

> ***Примечание:*** Общим свойством всех моделей является их подобие оригинальной системе или системе-оригиналу. Важность построения моделей заключается в возможности их использования для получения информации о свойствах или поведении системы-оригинала. При этом процесс построения и последующего применения моделей для получения информации о системе-оригинале получил название моделирование.

Процесс разработки моделей и их последующего конструктивного применения требовал от разработчиков не только знания общей методологии системного анализа, но и наличия у них соответствующих изобразительных средств или языков для фиксации результатов моделирования и их документирования. Естественный язык не вполне подходил для этой цели, поскольку обладает неоднозначностью и неопределенностью. Для построения моделей были разработаны достаточно серьезные теоретические методы, основанные на развитии математических и логических средств моделирования, а также предложены различные формальные и графические нотации, отражающие специфику решаемых задач. Появление ***унифицированного языка моделирования (Unified Modeling Lan-***

guage, UML), который первоначально был ориентирован на решение задач первых двух этапов ЖЦ – программ, а затем стал использоваться на всех этапах разработки ПС, было воспринято с большим оптимизмом всеми программистами.

Появление ***сети Интернет*** дало возможность подключить к ней огромное количество пользователей, это поставило ряд проблем (как технологического, так и юридического и этического характера) регулирования доступа к информации. Остро встала проблема защиты компьютерной информации и передаваемых по сети сообщений. С появлением Интернет [30]:

- сильно возросла важность масштабируемых архитектур, поддерживающих неограниченное число пользователей;
- возникает необходимость в защите организации от попыток взлома серверов и проникновения через Интернет вирусов и небезопасных компонентов;
- возрастает асинхронное использование распределенных ресурсов;
- появилась необходимость в разработке пользовательских интерфейсов, не требующих специальной установки (обозреватели);
- появилась необходимость в отделении логики приложений от пользовательского интерфейса;
- появилась необходимость создания многоуровневых приложений и переноса их на мощные Интернет-серверы;
- появилась необходимость доступа к функциям приложения с различных клиентских платформ и возможность создания интерактивных приложений, использующих функции WEB-серверов;
- появилась необходимость и возможность создания ПС, работающих на разных платформах (с некоторыми ограничениями в совместимости и надежности);
- появилась возможность публиковать огромный объем информации, благодаря снижению стоимости сопровождения увеличивается доступность продукции и объем продаж;
- появилась возможность создания новых типов приложений как чисто информационных (простой HTML), так и активных (Active X, Java).

Решить эти проблемы во многом удалось за счет применения ***компонентного подхода*** к разработке ПС, который предполагает построение программного обеспечения (ПО) из отдельных компонентов – физически отдельно существующих частей ПО, взаимодействующих между собой через ***стандартизованные двоичные интерфейсы.*** Объекты-компоненты можно собирать в динамические библиотеки или исполняемые файлы, распространять в двоичном коде и использовать в любом языке программирования, поддерживающем соответствующую технологию (рисунок 8) [12].

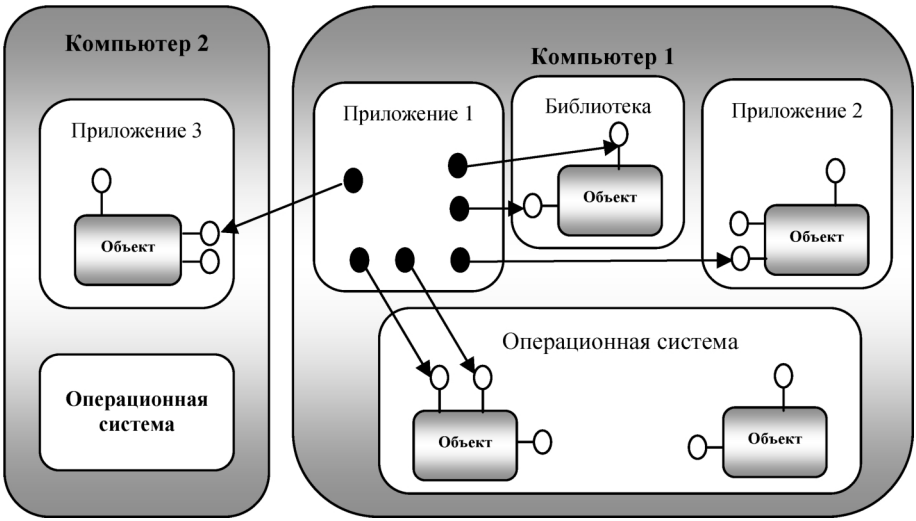


Рисунок 8 – Компонентная модель взаимодействия объектов

Компонентный подход лежит в основе технологий, разработанных на базе **COM** (**Component Object Model** – *компонентная модель объектов*) и технологии создания распределенных приложений **CORBA** (**Common Object Request Broker Architecture** – *общая архитектура с посредником обработки запросов объектов*). Эти технологии используют сходные принципы и различаются только особенностями реализации. О них более подробно расскажем в следующих главах.

1.3 Иерархия понятий в технологии программирования

Рассмотрим иерархическое построение понятий, связанных в единое целое в рамках единой технологии программирования (рисунок 9), и приведем различного рода классификации, которые позволят понять взаимосвязь этих понятий.

Рисунок 9 – Иерархия понятий в технологии программирования

Аппаратура – вот тот «кит», который определяет все новое в программировании. К конечном счете все новые идеи в программировании идут от новых технологий в разработке аппаратного обеспечения ЭВМ. Развитие вычислительной техники происходит скачкообразно, это приводит к пересмотру общих концепций и возникновению новых парадигм программирования. Так, появление прерываний в ЭВМ 2-го поколения привело к мультипрограммированию; создание микропроцессоров к появлению ПЭВМ и к развитию мощных графических средств, и как следствие, к появлению CASE-инструментов; создание многопроцессорных сис-

тем привело к появлению параллелизма вычислений; появление мощных ПЭВМ и развитие телекоммуникационных технологий – к появлению сети Интернет и т.д.

Каждое поколение ЭВМ программировалось по-своему. Первое – машинными командами, для второго были созданы первые языки программирования высокого уровня – Fortran, Algol, PL/1 и др., а затем и Pascal, которые оказались подходящими и для программирования ЭВМ третьего поколения. Однако языки программирования постоянно менялись и совершенствовались, чтобы служить адекватными моделями средств программирования. ЭВМ четвертого и пятого поколений были рассчитаны на «интеллектуальные пакеты программ», для которых традиционное программирование не подходит.

> ***Примечание:*** Синтаксический груз и сейчас отягощает понимание процесса программирования, т. к. при разработке языков программирования первоначально делали упор не на семантику, а на синтаксис (см. п. 1.2).

По одной из классификаций языки программирования делятся на императивные (операторные или процедурные) и декларативные (логические или функциональные) (рисунок 10) [22][5].

Императивная программа состоит из последовательности операторов и предложений, управляющих последовательностью их выполнения (операторы ввода/вывода, присваивания, передачи управления и т.п.). В основе такого программирования лежат взятие значения какой-либо переменной, совершение над ней действия и сохранение нового значения с помощью операторов присваивания, и так до тех пор, пока не будет получено желаемое окончательное значение.

[5] В кн. Мир ЛИСПа. Т.1 С.195.

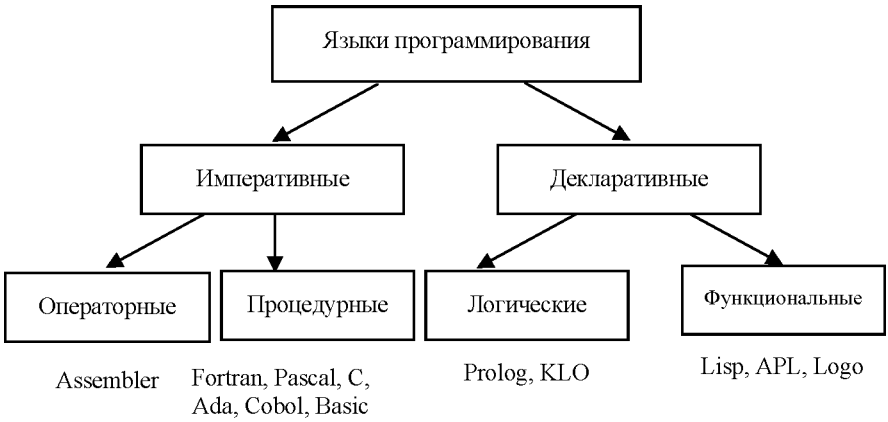


Рисунок 10 – Классификация языков программирования

В отличие от процедурного программирования (в котором решение сводится к разработке алгоритма, выполняющего действие) в декларативном программировании решение задачи получают из описания структуры и условий задачи. Декларативные формализмы отвечают на вопрос «Что?». Чтобы в декларативном языке можно было выполнять вычисления, в него вводится процедурная семантика. В декларативной программе последовательность и способ выполнения программы не фиксируются, и она (программа) может работать в обоих направлениях. Наиболее существенными классами декларативных языков являются функциональные (аппликативные) и логические.

> ***Примечание:*** На практике языки программирования не являются чисто процедурными, функциональными или логическими, содержат в себе черты языков различных типов. На процедурном языке часто можно написать функциональную программу или ее часть, и наоборот. Точнее было бы говорить вместо типа языка о стиле или методе программирования.

Часть идей в программировании ведет к созданию действительно новых способов мышления и программирования, даже не связанных с кон-

кретным языком. **Парадигма – это новая модель конструирования программы и взаимодействия ее с данными.** На рисунке 11 приведены основные парадигмы программирования, которые используются в настоящее время.

Было бы неправильным отрывать программы от обрабатываемых ими данных. Хорошо известна формула Н. Вирта «*Алгоритмы + структуры данных = программы*», отражающая точку зрения, что данные и программы диалектически переплетаются сложным образом [7].

> ***Примечание:*** Нельзя однозначно утверждать, что приоритетно: программа или данные, так как программы могут использовать в качестве данных другие программы (*синтаксические анализаторы, трансляторы, интерпретаторы*); данные могут использоваться как программы (в системах программирования, основанных на *базах знаний*).

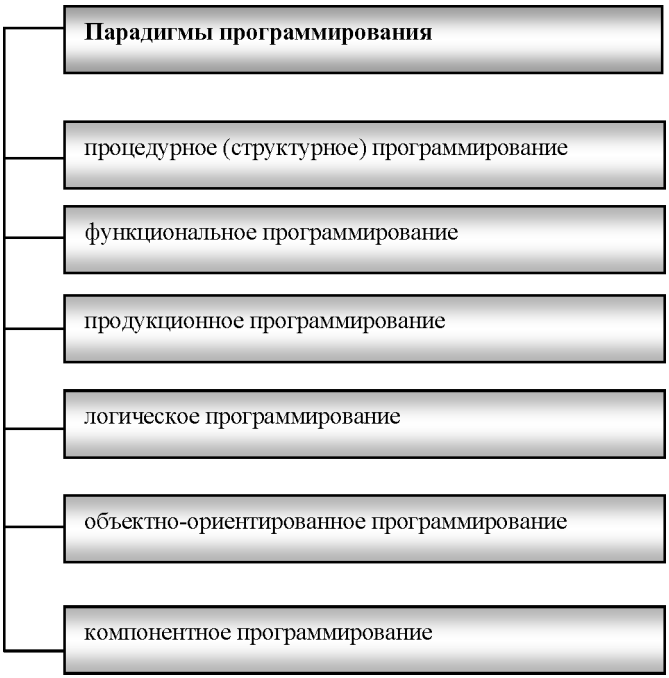


Рисунок 11– Основные парадигмы программирования

В программировании данные и знания могут выражаться как в виде пассивных декларативных структур данных, так и в виде процедурных знаний по их интерпретации и обработке [23][6]. В традиционных вычислительных машинах фон неймановского типа представление данных и команд на машинном уровне одинаково. В языках высокого уровня, напротив, данные и программа обычно четко отделены друг от друга, поэтому воздействие на программу возможно только с помощью параметров. В логическом программировании в представлении данных и программы наблюдается другая крайность: там нет необходимости отделять данные от программы, программу можно трактовать как совокупность логических правил. В функциональном программировании между программой и данными достигнут некоторый компромисс: форма представления программы и данных едина, и программу можно свободно трактовать как данные, также как данные можно интерпретировать и применять как программу (это решает программист). В объектной модели данные объединяются в единую структуру (класс) вместе с процедурами, их обрабатывающими (см. п.1.2).

Методы программирования включают все варианты структурирования данных. Программы представляют собой в конечном счете конкретные формулировки абстрактных алгоритмов, основанные на конкретных представлениях и структурах данных. Решение о структурировании данных программист обязан принимать с учетом алгоритмов, применяемых к этим данным, и наоборот, структура и выбор алгоритмов существенным образом зависят от структуры данных [8].

В научно-технических вычислениях в основном используются числовые данные и (или) структуры, на них базирующиеся, а в системах искусственного интеллекта для представления данных в основном используются символьные структуры, поэтому для задач из области искусственного интеллекта вместо традиционного алгоритмического подхода применяются различные эвристические методы поиска решения.

Традиционное программирование базируется главным образом на ***алгоритмическом способе (методе)*** решения задачи, при котором для принятия решения, как правило, достаточно информации, чтобы сделать вер-

[6] В кн. Мир ЛИСПа. Т.2. С.19

ный выбор. Решение будет найдено всегда, когда оно вообще возможно. Для традиционных численных вычислений характерно большое количество простых действий (вычисление, присваивание, вывод и т.д.) над большим количеством простых элементов (числа, строки, массивы, структуры данных и т.д.). В численных вычислениях большое внимание уделяется *количественной стороне* и численным значениям данных.

Когда имеющиеся в наличии данные недостаточны или когда другие способы не пригодны для принятия решения, приходится прибегать к ***эвристическим методам,*** которые позволяют получить решение задачи с использованием процедуры самообучения. Они обычно основываются на связанных с задачей специальных знаниях, простейших правилах, интуитивных критериях, базирующихся на предыдущем опыте. При обработке ***знаний*** важна *качественная сторона данных*, т.е. особенности их хранения и их функциональные особенности. Знания в большинстве случаев представлены в символьном виде или *в виде символьных структур*, при обработке важна не столько форма рассматриваемых знаний, сколько их содержание и значение.

> Эвристические системы могут быть ориентированы на сбор информации, необходимой для получения требуемого решения. Эвристические методы используются в системах искусственного интеллекта, в которых происходит совершенствование определенных процедур на основе опыта решения проблем в некоторой области.

Таким образом, методы программирования можно применять поразному для решения различных задач. Однако то или иное их применение в первую очередь определяется тем, удовлетворяет ли разрабатываемая система установленным для нее ***требованиям*** – спецификациям (***specification***). Увеличение размеров ПС и усложнение программирования предъявляют все большие требования к качеству программ. Автоматизация программирования и автоматический синтез программ также предъявляют высокие требования к используемому языку и методам программирования.

Для разработки надежных программ необходимо выбирать адекватные системы и ***среды программирования***, которые должны поддерживать

весь технологический процесс разработки: от проектирования до сопровождения.

Среда программирования образуется из набора различных вспомогательных средств программирования и управления данными, которые часто объединяются *интегрированную рабочую среду,* в которой различные вспомогательные средства можно вызывать друг из друга, не обращаясь к операционной системе. Основная цель создания среды программирования – поддержание процессов программирования (кодирования), отладки и тестирования ПС. Наиболее известными средами программирования являются среды визуального программирования Borland Delphi (язык программирования Object Pascal), MS Visual C++, Power Builder, MS Visual Basic, MS Visual Studio и т.д.

Широко используемые для проектирования и разработки ПС CASE-средства представляют собой *инструментальные системы технологии программирования,* предназначенные для поддержки всех процессов разработки и сопровождения в течение всего жизненного цикла ПС, и ориентированы на коллективную разработку больших программных систем с длительным жизненным циклом (более подробно рассмотрим эти вопросы в последующих главах).

КОНТРОЛЬНЫЕ ВОПРОСЫ К ГЛАВЕ 1:

1. Дайте определения понятий алгоритм, программирование, информация.
2. Что включает в себя понятие «технология программирования»?
3. Чем отличается технология программирования от методологии программирования и программной инженерии?
4. Перечислите основные этапы развития технологии программирования и соответствующие им методологии программирования.
5. Расскажите про эволюцию в архитектуре программ.
6. В чем отличие методологии структурного проектирования программных систем от методологии объектно-ориентированного проектирования?
7. В чем особенности методологии объектно-ориентированного анализа и проектирования?

8. Что такое парадигма программирования? Перечислите их основные особенности.
9. Какие парадигмы программирования лежат в основе объектно-ориентированного программирования?
10. Что такое предметная область? В чем заключается принцип концептуализации предметной области?
11. Приведите классификацию языков программирования. В чем их особенности?

2 ОБЩИЕ ПРИНЦИПЫ РАЗРАБОТКИ СЛОЖНЫХ ПРОГРАММНЫХ СИСТЕМ

Как уже говорилось ранее (см. п. 1.1), программирование – специфический вид интеллектуальной деятельности человека, разработка программного обеспечения имеет ряд специфических особенностей [10]:

- Прежде всего, следует отметить некоторое противостояние: *неформальный* характер требований к ПС (постановка задачи) и понятия ошибки в нем, но *формализованный* основной объект разработки – программы ПС. Тем самым разработка ПС содержит определенные этапы формализации, а переход от неформального к формальному существенно неформален.
- Разработка ПС носит *творческий характер* (на каждом шаге приходится делать какой-либо выбор, принимать какое-либо решение), а не сводится к выполнению какой-либо последовательности регламентированных действий (хотя регламент в работе больших коллективов разработчиков необходимо соблюдать). Несмотря на то что ПС относятся к промышленным программным продуктам, процесс разработки ПС ближе к процессу проектирования каких-либо сложных устройств, но никак не к их массовому производству. Этот творческий характер разработки ПС сохраняется до самого ее конца.
- Программная система представляет собой некоторую совокупность текстов (т.е. *статических* объектов), смысл же (семантика) этих текстов выражается процессами обработки данных и действиями пользователей, запускающих эти процессы (т.е. является *динамическим*). Это предопределяет выбор разработчиком ряда специфичных приемов, методов и средств.
- Программная система при своей эксплуатации не расходуется и не расходует используемых ресурсов.

2.1 Особенности современных программных систем как объектов разработки

Разработка программного обеспечения за последние 20 лет сильно изменилась (см. п. 1.2). Обновились традиционные методы управления,

которые теперь учитывают накопленный за долгое время опыт. Раньше считалось, что нужно разработать как можно больше функций за наименьшее время, и именно по этой причине современные программные продукты так сложны. Однако давление конкуренции, растущее влияние пользователей, постоянное изменение ситуации на технологическом рынке и непрестанно развивающиеся технологии привели к тому, что организации стремятся упростить процесс разработки. Главное же, что побудило их к этому – провал огромного количества проектов. Сегодня цель разработчиков – создать как можно лучшую программу при минимуме затрат, а их девиз – «выпустить нужный продукт в нужное время» [30].

В объектах программной инженерии (ПС) за последние годы произошли следующие коренные изменения:

- резко возросли *масштабы* и функциональная *размерность* программных компонентов (приложений, компонентов динамических библиотек и информационных массивов БД), готовых к использованию в различных приложениях и сочетаниях;
- увеличилась *трудоемкость* создания таких комплексов ПС и БД и наполнения их информацией (сотни человеко-лет), *длительность* жизненного цикла – несколько лет или десятилетий, ПС часто функционируют в нескольких версиях, существенно различающихся функциональными характеристиками и качеством;
- комплексы взаимодействующих ПС и информационных массивов БД могут размещаться на *различных* по архитектуре удаленных *аппаратных и операционных платформах* и переноситься между ними;
- для обеспечения мобильности начинает применяться *стандартизация структуры и интерфейсов* их компонентов с операционной и внешней средой;
- формализуется спектр *показателей качества*, резко возросли требования к надежности и безопасности функционирования ПС.

Перечисленные выше изменения объектов и проблемы обеспечения их ЖЦ вызвали ряд принципиальных *изменений в методологии* их создания и развития:

- на смену индивидуальному программированию приходит *методология коллективной, индустриальной разработки* с профессиональным разделением труда;
- возникла необходимость *применения автоматизированных методов* и средств управления *сложными проектами* и коллективами;
- высокая стоимость и большие ресурсы привели к необходимости детального технико-экономического *анализа и обоснования проектов* до начала их осуществления;
- создание ПС не завершается после сертификации первой версии, а, как правило, они длительное время развиваются и *модифицируются*, превращаясь в серию версий;
- для согласования взаимодействия многих программных компонентов и информационных массивов с операционной средой и обеспечения их мобильности, начала применяться идеология *открытых систем*;
- накопление высоко качественных ПС повысило актуальность развития и применения методов и средств автоматизации их переноса на различные платформы и *конфигурационного управления* ими в распределенных системах;
- повысились требования к *профессиональной квалификации* специалистов в области разработки ПС.

В результате внедрения современных методологий и технологий огромное значение приобретает обеспечение ***высокого качества*** ПС. Рост доверия к возможностям ПС приводит к расширению их применения, быстрому увеличению объемов разработок, а также к возрастанию важности выполняемых ими функций. Качество стало основой конкурентоспособности и возможности широкого применения ПС.

2.2 Показатели качества программных систем

Номенклатура и требуемые значения показателей качества определяются прежде всего *функциональным назначением* конкретных ПС. Прежде всего, хорошая программа должна делать то, что ожидает от нее заказчик – то есть удовлетворять требованиям заказчика. Это приводит к широкому спектру показателей качества в спецификации требований к программному продукту.

Про факторы качества и цели программирования много писалось, а также много делалось попыток их выделения и анализа. В общем случае под ***качеством (quality) программ*** понимается то, насколько они соответствуют установленным для них требованиям – спецификациям и сколько высоко установлена планка этих требований, это совокупность черт и характеристик ПС, которые влияют на ее способность удовлетворять заданные потребности пользователей [26, 16].

Перечислим наиболее важные ***показатели качества*** и связанные с ними подходы к программированию:

- ***Корректность (correctness).*** В первую очередь программа должна правильно работать, ошибочную программу невозможно использовать, даже если у нее будут другие полезные свойства. Доказательство общей правильности программы проводится не путем анализа ее прогона, а на основе анализа программы как статического математического объекта, на который распространяются аксиомы и правила логического вывода. Для этого требуется тщательный учет *структуры программы* и *семантики языка* программирования. Доказательство и обеспечение корректности программ обычно более сложно, чем больше и сложнее сами программы. Исследования, связанные с поиском способов доказательства правильности программ, внесли существенный вклад в разработку технологий программирования, благодаря этим исследованиям удалось достичь лучшего понимания структуры языков программирования, принципов и методов программирования.
- ***Надежность (reliability)*** – это способность ПС выполнять возложенные на нее функции при поступлении требований на их выполнение. Понятие надежности существенно отличается от понятия доказательства правильности программы, т.к. *правильность* – это некоторое *статическое свойство*, а надежность относится к *динамическим требованиям*, предъявляемым к системе, и способности системы удовлетворять этим требованиям [22]. Доказательство правильности программы не дает полного решения проблемы надежности ПО практически используемых систем из-за ограничений, накладываемых машинной арифметикой, неопределенностью результатов и параллелизмом операций. Очень трудным делом может оказаться и

разработка спецификации, согласно которой в дальнейшем будет производиться верификация программы, т.к. невозможно показать правильность самой спецификации в смысле правильного отображения замыслов разработчиков.

> ***Примечание:*** Программа, являющаяся «правильной», может считаться ненадежной, если спецификации, которым она удовлетворяет, не охватывают всех требований пользователя к этой программе. Программы искусственного интеллекта, например, не всегда дают верные решения из-за наших недостаточных или ошибочных знаний об области применения или из-за используемых в решении эвристик, хотя технически программы и не содержат ошибок. Стоимость ошибки, проявившейся в программе обычно тем больше, чем позднее обнаружена ошибка и чем обширнее используется программа.

Надежность ПО включает в себя такие составные свойства:

- как *отказоустойчивость* – возможность восстановления программы и данных в случае сбоев в работе;
- *безопасность* – сбои в работе программы не должны приводить к опасным последствиям (авариям);
- *защищенность* от случайных или преднамеренных внешних воздействий («защита от дурака», вирусов, спама т.п.).

- ***Эффективность (efficiency)*** – это отношение уровня услуг, предоставляемых ПС пользователю при заданных условиях, к объему используемых ресурсов. Эффективность ПО оценивается следующими показателями: время выполнения кода, загруженность процессора, объем требуемой памяти, время отклика и т.п. Вычислительная машина традиционно являлась критическим ресурсом, увеличение быстродействия должно достигаться не только за счет применения новой более производительной аппаратуры, но и за счет применения новых более эффективных алгоритмов, «распараллеливания» вычислений и т.д., то есть *эффективность программирования* становится все более важным фактором.
- ***Сопровождаемость (maintainability) и адаптируемость (adaptability)*** – это характеристики ПС, которые позволяют минимизировать

усилия по внесению изменений для устранения в нем ошибок и по его модификации в соответствии с изменяющимися потребностями пользователей. Сопровождаемость означает, что программа должна быть написана с расчетом на ее дальнейшее развитие. Это *критическое свойство* системы, т.к. изменения ПО неизбежны вследствие изменения бизнеса. Сопровождение программы выполняют, как правило, не те люди, которые ее разрабатывали. Сопровождаемость включает такие элементы как наличие и понятность проектной документации, соответствие проектной документации исходному коду, понятность исходного кода, простота изменений исходного кода, простота добавления новых функций.

- ***Тестируемость (testability) и корректируемость***. Тестирование – один из способов доказательства правильности или ошибочности программы. Тестирование разделяется на *автономное*, когда проверяется работа отдельных модулей (компонентов), и *комплексное*, когда проверяется правильность совместной работы составных частей системы, при этом особое внимание уделяется взаимодействию компонентов. Тестирование программ и проверка их корректности – часто наиболее объемная работа, чем их написание. Плохую программу нередко легче сделать заново, чем выполнить тестирование и коррекцию.
- ***Отлаживаемость (debuggability).*** Почти во всех уже оттестированных программах позже находятся ошибки или возникают потребности внести исправления. В отличие от тестирования, которое служит лишь для установления факта существования ошибок, отладка необходима их для локализации и устранения. Программы нужно создавать так, чтобы позднее было просто локализовать и исправить ошибки.
- ***Переносимость (portability).*** Машины и технические средства развиваются и дешевеют быстрее, чем программы. Поэтому ПО должно быть легко переносимым на новые и более дешевые машины и с одной платформы на другую.
- ***Удобочитаемость (readability) и понятность программ.*** Помимо разработчиков и другие должны при необходимости в состоянии понять смысл программы и ознакомиться с текстом работающих функ-

ций (и прочей документацией). С увеличением среднего размера программ все большее их количество приходится создавать коллективными усилиями, чем еще больше подчеркивается значимость удобочитаемости.

- ***Удобство использования (usability).*** ПС должна быть легкой в использовании, причем именно тем типом пользователей, на которых она рассчитана. Это включает в себя интерфейс пользователя и адекватную документацию. Причем, пользовательский интерфейс должен быть не интуитивно, а профессионально понятным пользователю.
- ***Полезность (validity).*** Верно работающая программа не найдет своего применения, если она не решает проблему пользователя и не соответствуют его ожиданиям. Разработка ПО и его последующее сопровождение могут оказаться убыточными.

Функциональность и надежность являются обязательными критериями качества ПС, причем обеспечение надежности будет красной нитью проходить по всем этапам и процессам разработки ПС. Остальные критерии используются в зависимости от потребностей пользователей в соответствии с требованиями к ПС.

При разработке ПС **необходимо осуществлять баланс между достигаемым качеством и ресурсами**, которые требуются для реализации проекта. Гарантии качества сложных ПС возможны только при условии непрерывного контроля качества выполняемых работ и отчетных документов, это требует постоянного и непрерывного взаимодействия разработчика с заказчиком или потенциальными пользователями программы, что отразилось на изменении технологии программной инженерии.

2.3 Проблемы разработки сложных программных систем

Большинство программных систем объективно очень сложны. «Сложность программного обеспечения – отнюдь не случайное его свойство», писал Брукс [3], она вызывается четырьмя основными причинами [5]:

- сложностью реальной предметной области, из которой исходит заказ на разработку;
- трудностью управления процессом разработки;
- необходимостью обеспечить достаточную гибкость программы;

- неудовлетворительными способами описания поведения больших дискретных систем.

Сложность реальной предметной области. Индустрия программного и информационного обеспечения непрерывно расширяет области применения ПС в отраслях народного хозяйства, в науке, в быту, функциональные возможности программ постоянно расширяются, расширяется и круг пользователей, растет их влияние на использование программных продуктов. К программам предъявляется множество различных, порой взаимоисключающих требований (часто не формулируемых явно), такие как удобство, производительность, стоимость, выживаемость и надежность. Сложность задачи и порождает ту сложность программного продукта, о которой пишет Брукс [3].

Эта *внешняя сложность* обычно возникает из-за «нестыковки» между пользователями системы и ее разработчиками: разработчики ПС не являются специалистами в автоматизируемых предметных областях, а специалист предметной области (пользователь, заказчик) не может, как правило, сформулировать проблему в нужном ракурсе. **У пользователей и разработчиков разные взгляды на сущность проблемы, и они делают различные выводы о возможных путях ее решения.**

> ***Примечание:*** Требования, которые выдвигает пользователь, не всегда можно взаимнооднозначно зафиксировать на естественном языке и языке схем, они открыты для различных интерпретаций и часто содержат элементы, относящиеся скорее к дизайну, чем к необходимым требованиям разработки.

Внутренние сложности возникают в результате изменений требований к ПС уже в процессе разработки, т. к. само осуществление программного проекта часто изменяет проблему, рассмотрение первых результатов работы системы заставляет пользователей лучше понять и отчетливей сформулировать то, что им действительно нужно, а разработчикам позволяет задавать более осмысленные вопросы пользователям.

Большая ПС – это крупное капиталовложение, и при изменении внешних требований в системе должны быть предусмотрены возможности для внесения необходимых изменений (см. п.2.2), опыт показывает, что существенный процент затрат на разработку программных систем тратится именно на сопровождение.

Трудности управления процессом разработки. Современные ПС слишком сложны, их размер исчисляется десятками тысяч или даже миллионами строк кода на языках высокого уровня. Ни один человек никогда не сможет полностью понять и создать такую систему. Чтобы решить все необходимые проблемы, требуется привлечение команды разработчиков, в идеале как можно меньшей по численности. Всегда будут возникать значительные трудности, связанные с организацией *коллективной разработки*. Чем больше разработчиков, тем сложнее связи между ними и тем сложнее координация, особенно если участники работ географически удалены друг от друга, что типично в случае очень больших проектов. Таким образом, при коллективном выполнении проекта главной задачей руководства является поддержание единства и целостности разработки.

Гибкость программного обеспечения. Программирование обладает предельной гибкостью, и разработчик может сам обеспечить себя всеми необходимыми элементами, относящимися к любому уровню абстракции. Компании-разработчики стремятся к созданию библиотек компонентов, которые можно было бы использовать в дальнейших разработках. Однако в этом случае такие компоненты приходится делать все более универсальными и унифицированными (стандартизованными), что в конечном итоге увеличивает трудоемкость и сложность разработки.

Проблема описания поведения больших дискретных систем. Так как исполнение ПС осуществляется на цифровом компьютере, то любая ПС – это система с дискретными состояниями, число возможных ее состояний конечно, в больших системах это число в соответствии с правилами комбинаторики очень велико, т. к. внутри большой прикладной программы могут существовать сотни и даже тысячи переменных и несколько потоков управления. Полный набор этих переменных, их текущих значений, текущего адреса и стека вызова для каждого процесса описывает *состояние информационной среды* в каждый момент времени (см. п. 1.1).

Основной способ проектирования системы – когда она разделяется на части так, чтобы одна часть минимально воздействовала на другую. Однако каждое событие, внешнее по отношению к ПС, может перевести систему в новое состояние, и, более того, переход из одного состояния в другое не всегда детерминирован. При неблагоприятных условиях внешнее событие может нарушить текущее состояние системы из-за того, что ее

разработчики не смогли предусмотреть все возможные варианты, в дискретных системах любое внешнее событие может повлиять на любую часть внутреннего состояния системы. Это является главной причиной обязательного тестирования наших систем; но дело в том, что всеобъемлющее тестирование таких программ провести невозможно. Пока в распоряжении разработчиков нет ни математических инструментов, ни интеллектуальных возможностей для ***полного*** *моделирования поведения* больших дискретных систем, они должны удовлетвориться разумным уровнем уверенности в их правильности. Универсальный язык моделирования ***UML (Unified Modeling Language)*** хотя и решает большую часть проблем и является хорошим средством коммуникации в рамках проекта, но не все.

2.4 Структура сложных систем

Исходя из анализа организации и поведения сложных биологических, социальных и технических систем, можно вывести пять общих признаков любой сложной системы. Основываясь на работе Саймона и Эндо, Куртуа предлагает следующее наблюдение [25]:

1. *«Сложные системы часто являются **иерархическими** и состоят из взаимозависимых подсистем, которые, в свою очередь, также могут быть разделены на подсистемы, и т.д., вплоть до самого низкого уровня».*

> ***Примечание 1:*** «Тот факт, что многие сложные системы имеют почти разложимую иерархическую структуру, является главным фактором, позволяющим нам понять, описать и даже «увидеть» такие системы и их части» [35]. В самом деле, мы можем понять лишь те системы, которые имеют иерархическую структуру.
>
> ***Примечание 2:*** Важно осознать, что архитектура сложных систем складывается и из компонентов, и из иерархических отношений этих компонентов: «Все системы имеют подсистемы, и все системы являются частями более крупных систем... Особенности системы обусловлены отношениями между ее частями, а не частями как таковыми» [32].

2. *«Выбор, какие **компоненты** в данной системе считаются элементарными, относительно произволен и в большой степени оставляется на усмотрение исследователя. Низший уровень для одного наблюдателя может оказаться достаточно высоким для другого».*

Примечание 3: Саймон называет иерархические системы *разложимыми,* если они могут быть разделены на четко идентифицируемые части, и *почти разложимыми,* если их составляющие не являются абсолютно независимыми.

3. *«**Внутрикомпонентная связь** обычно сильнее, чем связь между компонентами. Это обстоятельство позволяет отделять «высокочастотные» взаимодействия внутри компонентов от «низкочастотной» динамики взаимодействия между компонентами»* [32].

Примечание 4: Это различие внутрикомпонентных и межкомпонентных взаимодействий обусловливает разделение функций между частями системы и дает возможность относительно изолированно изучать каждую часть. Многие сложные системы организованы достаточно экономными средствами.

4. *«Иерархические системы обычно состоят из немногих типов подсистем, по-разному скомбинированных и организованных».*

Примечание 5: Разные сложные системы содержат одинаковые структурные части. Эти части могут использовать общие компоненты (мелкие или более крупные структуры). Сложные системы имеют тенденцию к развитию во времени, и Саймон утверждает, что сложные системы будут развиваться из простых гораздо быстрее, если для них существуют устойчивые промежуточные формы.

5. *«Любая работающая сложная система является результатом развития работавшей более простой системы... Сложная система, спроектированная «с нуля», никогда не заработает. Следует начинать с работающей простой системы».*

Примечание 6: В процессе развития системы объекты, первоначально рассматривавшиеся как сложные, становятся элементарными, и из них строятся более сложные системы. Более то-

го, невозможно сразу правильно создать элементарные объекты: с ними надо сначала повозиться, чтобы больше узнать о реальном поведении системы, и затем уже совершенствовать их.

2.5 Основные подходы к созданию сложных программных систем

Обнаружение общих абстракций и механизмов значительно облегчает понимание и разработку сложных систем, ***абстрагирование*** следует рассматривать как один из методов, используемый для решения сложных задач. Это понятие рассматривалось многими выдающимися программистами всех времен. «Следует рассматривать абстракцию как основной интеллектуальный прием, позволяющий ослабить количественные ограничения, которые накладываются на рассуждения с перечислением», – писал один из основоположников структурного программирования Э. Дейкстра[7], кроме того, он считал, что «абстракция проникает во все аспекты программирования». К. Хоор считает, что «абстрагирование проявляется в нахождении сходств между определенными объектами, ситуациями или процессами реального мира, и в принятии решений на основе этих сходств, отвлекаясь на время от имеющихся различий».[8] М. Шоу определила это понятие так: «Упрощенное описание или изложение системы, при котором одни свойства и детали выделяются, а другие опускаются. Хорошей является такая абстракция, которая подчеркивает детали, существенные для рассмотрения и использования, и опускает те, которые на данный момент несущественны»[9].

Суммируя эти разные точки зрения, можно дать следующее определение: ***абстракция*** выделяет существенные характеристики некоторого объекта, отличающие его от всех других видов объектов и, таким образом, четко определяет его концептуальные границы с точки зрения наблюдателя.

Значительное упрощение в понимании сложных задач достигается за счет образования из абстракций иерархической структуры. Определим иерархию следующим образом: ***Иерархия*** – это упорядочение абстракций, расположение их по уровням.

---

[7] В кн. Дал О., Дейкстра Э., Хоор К. Структурное программирование. С. 18.
[8] Там же. С. 98.
9 Shaw M. October 1984. Abstraction Techniques in Modern Programming Languages. P. 10.

Существует два вида иерархий:

1. «***часть-целое***», когда систему можно разделить на подсистемы, взаимосвязанные между собой, но при этом внутренние связи элементов внутри подсистем сильнее, чем связь между подсистемами;
2. «***простое-сложное***», которая показывает развитие систем в процессе их эволюции и построена на механизме наследования свойств.

Таким образом, программные системы, будучи в значительной степени отражением природных и технических систем, строятся путем разбиения сложного объекта на сравнительно независимые части путем ***декомпозиции***. При создании очень сложных объектов процесс декомпозиции повторяется многократно, данный метод разработки получил название ***пошаговой детализации***. На каждом уровне детализации используется свой ***уровень абстракции (Level of abstraction).***

В настоящее время рассматривают два вида декомпозиции систем:

- ***алгоритмическую***, когда она рассматривается как обычное разделение алгоритмов, где каждый модуль системы выполняет один из этапов общего процесса (используется при структурном проектировании систем, в качестве примера можно привести структурную схему системы);
- ***объектно-ориентированную,*** когда система представлена совокупностью автономных действующих лиц, которые взаимодействуют друг с другом, чтобы обеспечить поведение системы, соответствующее более высокому уровню. Каждый объект обладает своим собственным поведением, и каждый из них моделирует некоторый объект реального мира (используется при объектно-ориентированном проектировании).

Какая декомпозиция сложной системы правильнее – по алгоритмам или по объектам? В этом вопросе есть подвох, и правильный ответ на него: важны оба аспекта. Разделение по алгоритмам концентрирует внимание на *порядке происходящих событий*, а разделение по объектам придает особое значение агентам, которые являются либо объектами, либо субъектами действия.

Примечание: Нельзя сконструировать сложную систему одновременно двумя способами, тем более, что эти способы по сути

ортогональны. Сначала необходимо разделить систему либо по алгоритмам, либо по объектам, а затем, используя полученную структуру, попытаться рассмотреть проблему с другой точки зрения.

С чего лучше начать? Опыт многих программистов подсказывает, что полезнее начинать с объектной декомпозиции [34], которая имеет несколько чрезвычайно важных преимуществ перед алгоритмической:

- ✓ объектная декомпозиция уменьшает размер программных систем за счет повторного использования общих механизмов, что приводит к существенной экономии выразительных средств;
- ✓ объектно-ориентированные системы более гибки и проще эволюционируют со временем, потому что они развиваются из меньших систем, в которых мы уже уверены;
- ✓ объектная декомпозиция помогает разобраться в сложной программной системе, предлагая разумные решения относительно выбора подпространства большого пространства состояний.

2.5.1 Структурный подход к разработке программных систем

Немного истории. Структурный подход к разработке программ стал первой осознанно применяемой технологией в области программирования. В начале 70-х годов в литературе по программированию развернулась дискуссия об использовании оператора безусловного перехода Go To и о его возможно отрицательном влиянии на *качество программирования.* Инициатором дискуссии был Э. Дейкстра («Структурное программирование»), в ней принимали участие все наиболее известные программисты того времени: Д. Кнут («Искусство программирования»), Н. Вирт (автор языков программирования Pascal, Modula, Modula-2), Д. Баррон и т.д. Так как программы того времени стали достаточно объемными, то программы необходимо было разрабатывать так, чтобы их легко было читать, разбирать, изменять, использовать. На эти качества программ с особой остротой обратил внимание американский ученый в области программирования и информатики Р. Флойд в 1978 г. Требование *структурированности программ* было сформулировано им в качестве важнейшей ***парадигмы программирования***. При этом Р. Флойд отметил, что ни одна из сущест-

вующих технологий и языков программирования того времени не удовлетворяет в полной мере этой парадигме.

Итак, в результате дискуссии был выработан новый более строгий стиль программирования, который стали называть ***структурным программированием (Structures programming)***.

В основе структурного подхода к разработке ПС лежит *алгоритмическая декомпозиция*, когда система разбивается на функциональные подсистемы, которые, в свою очередь, делятся на подфункции, подразделяемые на задачи и так далее. Процесс разбиения продолжается вплоть до конкретных процедур (алгоритмов) (см. рисунок 12). При этом автоматизируемая система сохраняет целостное представление, в котором все составляющие компоненты взаимоувязаны. Разработка системы стала вестись по принципу «сверху-вниз», в отличие от применяемой ранее «снизу-вверх», когда при переходе от отдельных задач ко всей системе терялась целостность и возникали проблемы при информационной стыковке отдельных компонентов.

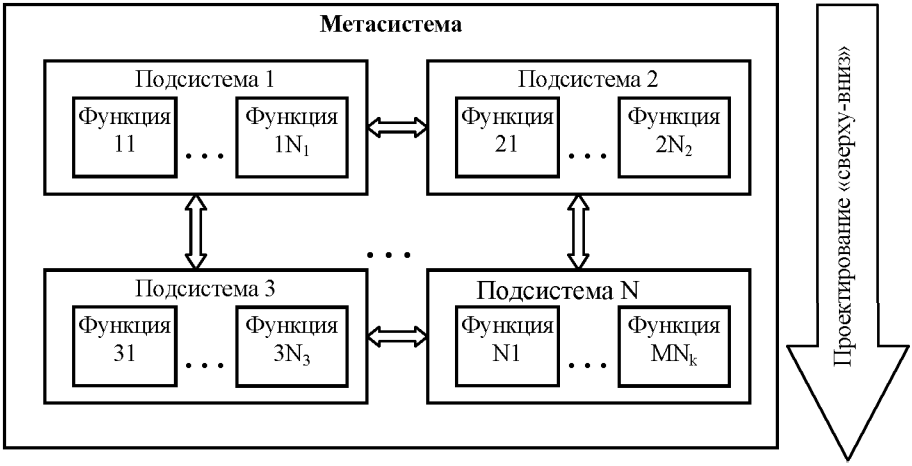


Рисунок 12 - Иерархия проектируемой системы

Все наиболее распространенные методологии структурного подхода базируются на ряде общих принципов [13]. В качестве двух базовых принципов используются следующие:

- ***принцип «разделяй и властвуй»*** – принцип решения сложных проблем путем их разбиения на множество меньших независимых задач, легких для понимания и решения [27][10];
- ***принцип иерархического упорядочивания*** – принцип организации составных частей проблемы в иерархические древовидные структуры с добавлением новых деталей на каждом уровне.

Выделение двух базовых принципов не означает, что остальные принципы являются второстепенными, поскольку игнорирование любого из них может привести к непредсказуемым последствиям (в том числе и к провалу всего проекта). Основными из этих принципов являются следующие:

- ***принцип абстрагирования*** – заключается в выделении существенных аспектов системы и отвлечения от несущественных;
- ***принцип формализации*** – заключается в необходимости строгого методического подхода к решению проблемы;
- ***принцип непротиворечивости*** – заключается в обоснованности и согласованности элементов;
- ***принцип структурирования данных*** – заключается в том, что данные должны быть структурированы и иерархически организованы.

При структурном проектировании первым шагом при получении спецификации программы является рассмотрение ее применительно к некоторой «идеальной» вычислительной машине, которая должна предусматривать как соответствующий набор структур данных, так и соответствующее множество операций над этими структурами. Структуры данных должны быть организованы таким образом, чтобы предусмотреть все многообразие состояний машины верхнего уровня, все подробности нужно было спрятать на разных уровнях реализации и внутри модулей так, чтобы они не влияли на использование модуля на более высоком уровне. Для этого между модулями устанавливались четко определенные интерфейсы.

[10] Dijktra E., p. 5.

Таким образом, окончательная программа основана на представлениях «по уровням абстрактных машин» (*levels of abstract machine*) (рисунок 13), где вычислительная машина верхнего уровня (логический уровень 1) идеально приспособлена к конкретной прикладной задаче, а машина на нижнем уровне (логический уровень N) непосредственно выполняет команды на выбранном языке программирования. Таким образом, процесс разработки программы не является просто процессом «разбивки на подпрограммы», поскольку на каждом уровне *уточняются* как *структуры данных,* так и *операции* над ними.

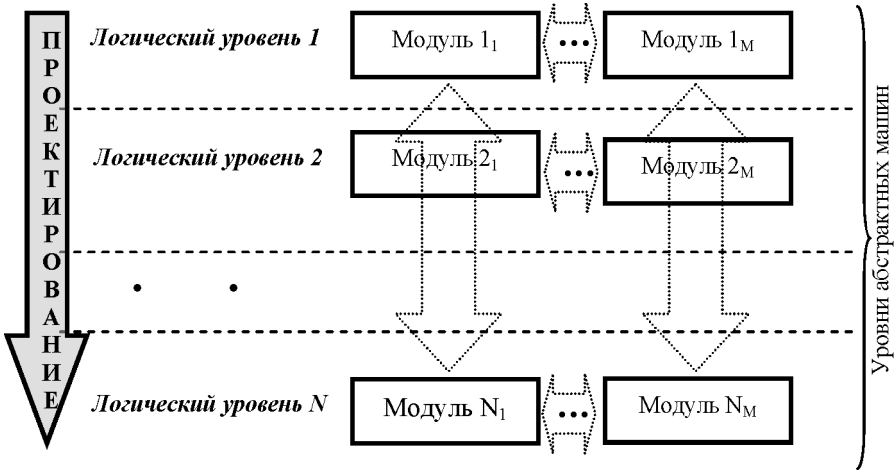


Рисунок 13 – Уровни абстрактных машин при структурном проектировании

В настоящее время большое количество существующих программ написано, используя перечисленные выше принципы. Тем не менее структурный подход не позволяет обеспечить ограничение доступа к данным; он также не предоставляет достаточных средств для организации параллелизма. Структурный метод не может обеспечить создание предельно сложных систем, и он, как правило, неэффективен в объектных и объектно-ориентированных языках программирования

2.5.2 Объектный подход к разработке программных систем

Немного истории. Термин «объект» появился практически независимо в различных областях, связанных с компьютерами, и почти одновременно в начале 70-х годов для обозначения того, что может иметь различные проявления, оставаясь целостным. Объектами назывались компоненты системы или фрагменты представляемых знаний [38].[11]

Объектно-ориентированный подход был связан со следующими событиями:

- ✓ прогресс в области архитектуры ЭВМ (создание компьютеров с descriptor-based и capability-based архитектурами);
- ✓ развитие объектно-ориентированных языков программирования, таких как Simula, Smalltalk, CLU, Ada;
- ✓ развитие методологии программирования, включая принципы модульности и скрытия данных.

Базовыми принципами объектно-ориентированной технология являются:

- ✓ ***абстрагирование;***
- ✓ ***инкапсуляция;***
- ✓ ***модульность;***
- ✓ ***иерархичность.***

Принципы являются *базовыми* в том смысле, что без любого из них модель не будет объектной. Кроме главных, имеются еще три дополнительных принципа (они полезны в объектной модели, но не обязательны):

- ✓ ***типизация;***
- ✓ ***параллелизм;***
- ✓ ***сохраняемость.***

Каждый из этих принципов сам по себе не нов, но в объектной модели они впервые применены в совокупности, таким образом, объектный подход к созданию ПС отражают эволюционное, а не революционное развитие проектирования; новая методология не порывает с прежними методами, а строится с учетом предшествующего опыта.

Примечание: Вместе с тем объектно-ориентированный анализ и проектирование принципиально отличаются от традицион-

[11] Yonezawa A., Tokoro M. , p.2.

ных подходов структурного проектирования: здесь нужно по-другому представлять себе процесс декомпозиции (объектный), а архитектура получающегося программного продукта в значительной степени выходит за рамки представлений, традиционных для структурного программирования.

Выбор правильного набора абстракций для заданной предметной области представляет собой главную (и самую сложную) задачу объектно-ориентированного подхода. Абстракция и инкапсуляция дополняют друг друга: абстрагирование направлено на наблюдаемое поведение объекта, а инкапсуляция занимается внутренним устройством объекта. Инкапсуляция выполняется посредством сокрытия информации (и внутренней структуры объекта и реализации его методов), таким образом, она определяет четкие границы между различными абстракциями.

В ООП выделяют несколько видов абстракций:

Абстракция сущности	Объект представляет собой полезную модель некой сущности в предметной области
Абстракция поведения	Объект состоит из обобщенного множества операций
Абстракция виртуальной машины	Объект группирует операции, которые либо вместе используются более высоким уровнем управления, либо сами используют некоторый набор операций более низкого уровня
Произвольная абстракция	Объект включает в себя набор операций, не имеющих друг с другом ничего общего

Модули образуют физическую структуру системы, это свойство становится особенно полезным, когда система состоит из многих сотен классов. В большинстве языков, поддерживающих принцип модульности как самостоятельную концепцию, интерфейс модуля отделен от его реализации (C++, Ada), что полностью согласуется с принципом инкапсуляции. Правильное разделение программы на модули является почти такой же сложной задачей, как выбор правильного набора абстракций.

Модули выполняют роль физических контейнеров, в которые помещаются определения классов и объектов при логическом проектировании системы.

Примечание: Для небольших задач допустимо описание всех классов и объектов в одном модуле. Однако для большинства программ лучшим решением будет сгруппировать в отдельный модуль логически связанные классы и объекты, оставив открытыми те элементы, которые совершенно необходимо видеть другим модулям.

Перечислим приемы и правила, которые позволяют составлять модули из классов и объектов наиболее эффективным образом:

1. Особенности системы, подверженные изменениям, следует скрывать в отдельных модулях; в качестве межмодульных можно использовать только те элементы, вероятность изменения которых мала.
2. Все структуры данных должны быть обособлены в модуле; доступ к ним будет возможен для всех процедур этого модуля и закрыт для всех других. Доступ к данным из модуля должен осуществляться только через процедуры данного модуля [31].

Примечание: Многие компиляторы создают отдельный сегмент кода для каждого модуля. Поэтому могут появиться ограничения на размер модуля. Динамика вызовов подпрограмм и расположение описаний внутри модулей могут сильно повлиять на локальность ссылок и на управление страницами виртуальной памяти. При плохом разбиении процедур по модулям учащаются взаимные вызовы между сегментами, что приводит к потере эффективности кэш-памяти и частой смене страниц.

Таким образом, принципы абстрагирования, инкапсуляции и модульности являются взаимодополняющими.

Понятие *типа* взято из теории абстрактных типов данных. Типизация позволяет защититься от использования объектов одного класса вместо другого. В разных языках программирования реализованы разные механизмы типизации, от слабой (C++) до сильной (Object Pascal), а в некоторых такой механизм отсутствует (Smalltalk).

Языки, в которых типизация отсутствует, обладают большей гибкостью. Но на практике, особенно при программировании «в большом», надежность языков со строгой типизацией с лихвой компенсирует некоторую потерю в гибкости.

Примечание: Теслер отметил следующие важные преимущества строго типизированных языков:

- «Отсутствие контроля типов может приводить к загадочным сбоям в программах во время их выполнения.
- В большинстве систем процесс редактирование-компиляция-отладка утомителен, и раннее обнаружение ошибок просто незаменимо.
- Объявление типов улучшает документирование программ.
- Многие компиляторы генерируют более эффективный объектный код, если им явно известны типы» [37].

В объектно-ориентированных языках параллелизм достигается за счет механизма многопоточности (процессов управления) и наличия прерываний, при этом главное внимание уделяется абстрагированию и синхронизации процессов. В настоящее время существует целый класс языков, ориентированных на параллельные вычисления (Ada, Modula-2, Occam).

И, наконец, несколько слов о сохраняемости. Она предполагает, что во время вычислений необходимо сохранять промежуточные данные и/или объекты во времени. Большинство языков программирования не имеет встроенных средств для этого (хорошим исключением является только язык Smalltalk), поэтому, как правило, сохраняемость достигается за счет применения СУБД, многие из которых встроены в оболочку интегрированных средств разработки ПС.

В заключение хотелось бы отметить, что объектная модель принципиально отличается от моделей, которые связаны с более традиционными методами структурного анализа, проектирования и программирования. Это не означает, что объектная модель требует отказа от всех ранее найденных и испытанных временем методов и приемов. Скорее, она вносит некоторые новые элементы, которые добавляются к предшествующему опыту. Наиболее важно, что объектный подход позволяет создавать системы, которые удовлетворяют пяти признакам хорошо структурированных сложных систем.

КОНТРОЛЬНЫЕ ВОПРОСЫ К ГЛАВЕ 2:

1. Перечислите основные особенности разработки программных систем.
2. Какие коренные изменения произошли в программной инженерии за последнее время?
3. В чем заключаются принципиальные изменения, произошедшие в методологии программирования?
4. Перечислите основные показатели качества программных систем.
5. Что такое надежность программного обеспечения? Какие составные части ее определяют?
6. В чем отличие отладки от тестирования?
7. В чем состоит сложность программных систем? Перечислите признаки сложных программных систем.
8. Что такое абстракция и абстрагирование? Приведите примеры абстракций в объектно-ориентированном программировании.
9. В чем состоит принцип пошаговой детализации?
10. Что такое декомпозиция? Где она применяется? Приведите примеры.
11. В чем состоит структурный подход к разработке программных систем? Перечислите основные принципы методологии структурного проектирования.
12. В чем состоит объектный подход к разработке программных систем? Перечислите базовые принципы методологии объектно-ориентированного проектирования.

3 ЖИЗНЕННЫЙ ЦИКЛ ПРОГРАММНЫХ СИСТЕМ

Немного истории. Появление понятия жизненного цикла ПО было связано с кризисом программирования, который наметился в конце 60-х – начале 70-х годов прошлого века (п. 1.2). Суть кризиса состояла в том, что программные проекты все чаще стали выходить из-под контроля: нарушались сроки, превышались запланированные объемы финансирования, результаты не соответствовали требуемым. Многие проекты вообще не доводились до завершения. Кроме того, оказалось, что недостаточно разработать программу, нужно ее еще сопровождать, а этап сопровождения часто требовал больше средств, чем разработка.

Впервые о жизненном цикле ПО заговорили в 1968 г. в Лондоне, где состоялась встреча 22 руководителей проектов по разработке ПО. На встрече анализировались проблемы и перспективы проектирования, разработки, распространения и поддержки программ. Было отмечено, что применяющиеся принципы и методы разработки ПО требовали постоянного усовершенствования. Именно на этой встрече была предложена концепция ***жизненного цикла программного обеспечения (SLC – Software Lifetime Cycle)*** как последовательности шагов-стадий, которые необходимо выполнить в процессе создания и эксплуатации ПО. В настоящее время она стала *методологической основой* промышленной инженерии.

Итак, жизненный цикл промышленного изделия – это последовательность этапов (фаз, стадий):

- проектирование;
- изготовление образца;
- организация производства;
- серийное производство;
- эксплуатация;
- ремонт;
- вывод из эксплуатации,

состоящих из технологических процессов, действий и операций. Организация промышленного производства с позиции жизненного цикла позволяет рассматривать все его этапы во взаимосвязи, что ведет к сокращению сроков, стоимости и трудозатрат.

3.1 Стандарты и проблемы жизненного цикла ПО

Процесс стандартизации и сертификации давно вошел и в программную инженерию, где он составляет *основу промышленного производства* программных продуктов, т.к. обеспечивает качество продуктов и продвижение их на рынок.

Рассмотрим понятия стандартизация и сертификация более подробно, т.к. многие начинающие программисты эти понятия смешивают.

Стандарт (standard) является нормативно-техническим документом, утверждаемым компетентным органом, устанавливающим комплекс норм, правил по отношению к предмету стандартизации, или типовым образцом, эталоном, моделью, которые принимаются как исходные для сопоставления с ними других предметов.

> ***Например:*** ГОСТ ЕСПД – единая система программной документации – документы, описывающие состав и структуру документации на разработку программ для ЭВМ (общее описание, техническое задание, эскизный проект, технический проект, описание применения). Типовые образцы – эталоны мер и весов (эталон метра, хранящийся в Париже в палате мер и весов).

Стандарт может быть разработан как на материально-технические предметы (продукцию, эталоны, образцы веществ), так и на нормы, правила, требования организационно-методического и общетехнического характера. Пример, вузы работают в соответствии с государственными образовательными стандартами, представленными в виде паспортов специальностей. Стандартизация распространяется на все сферы человеческой деятельности: науку, технику, промышленное и сельскохозяйственное производство, строительство, здравоохранение, транспорт и т.д., а следовательно, и на производство программных продуктов.

Сертификация в переводе с латыни означает «сделано верно». Для того чтобы убедиться в том, что продукт «сделан верно», надо знать, каким требованиям он должен соответствовать, и каким образом возможно получить достоверные доказательства этого соответствия. Общепризнанным способом такого доказательства служит сертификация соответствия, которая предполагает обязательное участие третьей стороны, осуществляется по правилам определенной процедуры, включающей обязательные испытания на соответствие стандарту.

В настоящее время принято выделять следующие основные типы стандартов:

- ***Корпоративные стандарты*** разрабатываются крупными фирмами (корпорациями) с целью повышения качества своей продукции. Такие стандарты разрабатываются на основе собственного опыта и с учетом требований мировых стандартов. Корпоративные стандарты не сертифицируются, но являются обязательными для применения внутри корпорации. В условиях рыночной конкуренции могут иметь закрытый характер. В сфере информационных технологий известны стандарты, разработанные Microsoft, Intel, IBM.
- ***Отраслевые стандарты*** действуют в пределах организаций некоторой отрасли (министерства). Например, СНИП – строительные нормы и правила. Разрабатываются с учетом требований мирового опыта и специфики отрасли. Являются, как правило, обязательными для отрасли. Подлежат сертификации.
- ***Государственные стандарты*** (ГОСТы) принимаются государственными органами, имеют силу закона. Разрабатываются с учетом мирового опыта или на основе отраслевых стандартов. Могут иметь как рекомендательный, так и обязательный характер (стандарты безопасности). Для сертификации создаются государственные или лицензированные органы сертификации.
- ***Международные стандарты.*** Разрабатываются, как правило, специальными международными организациями на основе мирового опыта и лучших корпоративных стандартов. Имеют сугубо рекомендательный характер. Право сертификации получают организации (государственные и частные), прошедшие лицензирование в международных организациях.

На всем протяжении развития технологий программирования и программной инженерии разрабатывались и стандарты ЖЦ ПО. Наиболее известными среди них являются:

- ✓ **1985 г.** (уточнен в 1988 г.) **DOD-STD-2167 A** – *Разработка программных средств для систем военного назначения*. Первый формализованный и утвержденный стандарт жизненного цикла для проектирования ПС систем военного назначения по заказам Министерства обороны США. Этим документом регламентированы восемь фаз (этапов) при создании

сложных критических ПС и около 250 типовых обязательных требований к процессам и объектам проектирования на этих этапах.

✓ **1994 г. MIL-STD-498**. *Разработка и документирование программного обеспечения.* Принят Министерством обороны США для замены DOD-STD-2167 A и ряда других стандартов. Он предназначен для применения всеми организациями и предприятиями, получающими заказы Министерства обороны США. Стандарт содержит рекомендации по обеспечению и реализации процессов ЖЦ сложных критических ПС высокого качества и надежности, функционирующих в реальном времени (описано около 75 процессов).

✓ **1995 г. IEEE 1074**. *Процессы ЖЦ для развития программного обеспечения*. Охватывает полный жизненный цикл ПС, в котором выделяются шесть крупных базовых процессов, которые детализируются 16 частными процессами (детализируются в совокупности 65 процессов-работ). Содержание каждого частного процесса начинается с описания его общих функций и задач и перечня действий — работ при последующей детализации. Для каждого процесса в стандарте представлена *входная и результирующая информация* о его выполнении и краткое *описание сущности* процесса. В стандарте внимание сосредоточено преимущественно на непосредственном создании ПС и на процессах предварительного проектирования. В приложении представлены варианты адаптации максимального состава компонентов ЖЦ ПС к конкретным особенностям типовых проектов.

Разработка стандартов ЖЦ и их практическое применение сталкивались с рядом ***проблем***:

- внедрение стандартов требовало вложения значительных средств, что не всегда окупалось;
- было неясно, все ли требуемые процессы надо выполнять и в какой мере;
- для различных типов ПО (информационные системы, системы реального времени, бизнес системы) приводились различные требования;
- высокая динамика отрасли приводила к быстрому устареванию стандартов;

• в различных корпоративных стандартах была терминологическая неоднозначность;

• во многих случаях применение стандартов было вызвано только требованиями заказчиков, хотя на практике превращалось в тормоз и приводило к невыполнению проектов.

Разрешением проблем стандартизации ЖЦ ПО явилась разработка и принятие в 1995 г. стандарта **ISO/IEC 12207:** «Information Technology – Software Life Cycle Processes»[12]. В 2000 г. он был принят в России как «ГОСТ 12207. Процессы жизненного цикла программных средств» [29].

Стандарт ISO 12207 разрабатывался с учетом лучшего мирового опыта на основе вышеперечисленных стандартов. Основными результатами данного стандарта являются:

- *введение единой терминологии* по разработке и применению ПО (стандарт предназначен не только для разработчиков, но и для заказчиков, пользователей, всех заинтересованных лиц);
- *разделение понятий* жизненного цикла ПО и модели ЖЦ ПО (сам жизненный цикл определяется как полная совокупность всех процессов и действий по созданию и применению ПО, а модель ЖЦ – как конкретный вариант организации ЖЦ, обоснованно выбранный для каждого конкретного случая);
- *описание организации* ЖЦ и его *структуры* (процессов);
- *выделение процесса адаптации стандарта* для построения конкретных моделей ЖЦ.

3.2 Жизненный цикл и этапы разработки программного обеспечения

Дадим точное определение понятия ЖЦ ПО. Под ***жизненным циклом программного обеспечения*** понимают непрерывный процесс, который начинается с момента принятия решения о необходимости создания программного обеспечения и заканчивается в момент его полного изъятия из эксплуатации.

[12] **ISO** - International Organisation of Standardisation - Международная организация по стандартизации
IEC - International Electromechanical Commission - Международная электротехническая комиссия

Процесс (process) – набор взаимосвязанных работ, которые преобразуют исходные данные в выходные результаты.

Стандарт ISO 12207 определяет организацию ЖЦ программного продукта как совокупность процессов, каждый из которых разбит на действия, состоящие из отдельных задач. В соответствии с ним все процессы ЖЦ делятся на три группы (рисунок 14):

- основные;
- вспомогательные;
- организационные.

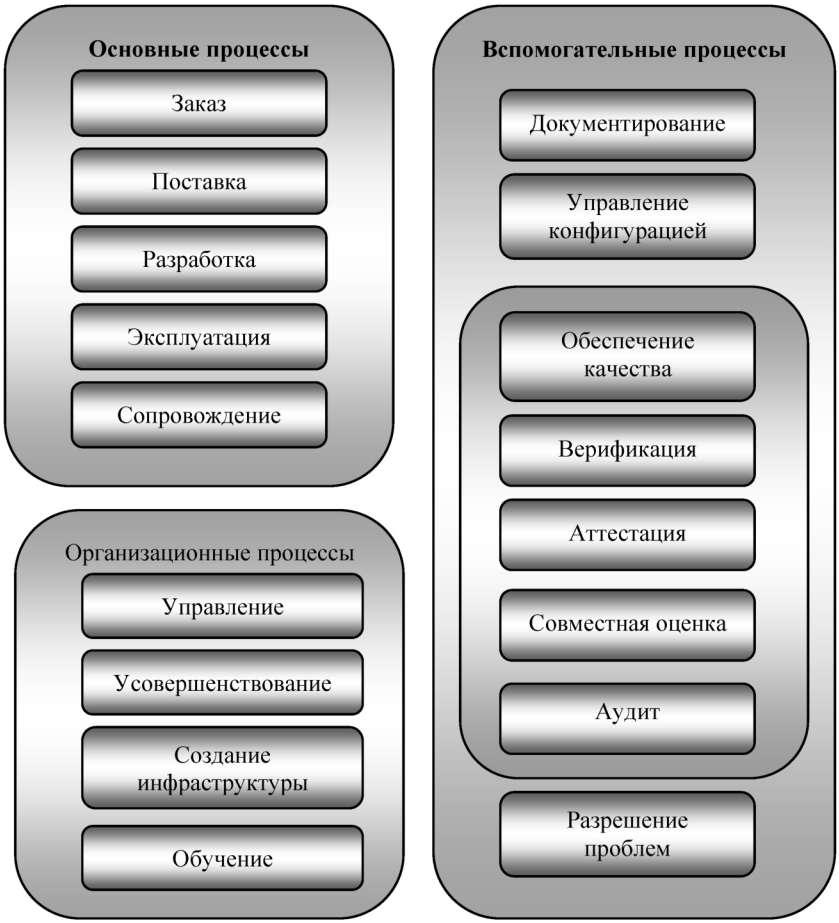


Рисунок 14 – Структура процессов ЖЦ ПО

Отдельно описан процесс адаптации стандарта, содержащий основные работы, которые должны быть выполнены при адаптации настоящего стандарта к условиям конкретного программного проекта.

К числу **основных процессов** относятся:

- **Заказ**. Определяет *работы заказчика*, то есть организации, которая приобретает систему, программный продукт или программную услугу.
- **Поставка**. Определяет *работы поставщика*, то есть организации, которая поставляет систему, программный продукт или программную услугу заказчику.
- **Разработка**. Определяет *работы разработчика*, то есть организации, которая проектирует и разрабатывает программный продукт.
- **Эксплуатация**. Определяет *работы оператора*, то есть организации, которая обеспечивает эксплуатационное обслуживание вычислительной системы в заданных условиях в интересах пользователей.
- **Сопровождение**. Определяет *работы персонала сопровождения*, то есть организации, которая предоставляет услуги по сопровождению программного продукта, состоящие в контролируемом изменении программного продукта с целью сохранения его исходного состояния и функциональных возможностей. Данный процесс охватывает перенос и снятие с эксплуатации программного продукта.

Вспомогательными процессами являются:

- **Документирование**. Определяет *работы по описанию информации*, выдаваемой в процессе жизненного цикла.
- **Управление конфигурацией**. Определяет *работы по управлению* конфигурацией.
- **Обеспечение качества**. Определяет *работы по объективному обеспечению* того, чтобы программные продукты и процессы соответствовали требованиям, установленным для них, и реализовывались в рамках утвержденных планов. Совместные анализы, аудиторские проверки, верификация и аттестация могут использоваться в качестве методов обеспечения качества.
- **Верификация**. Определяет *работы* (заказчика, поставщика или независимой стороны) *по верификации* ПП по мере реализации программного проекта.

Верификация (verification) – это процесс определения того, отвечает ли текущее состояние разработки, достигнутое на данном этапе, требованиям этого этапа.

- **Аттестация**. Определяет *работы* (заказчика, поставщика или независимой стороны) *по аттестации* программных продуктов программного проекта.
- **Совместный анализ**. Определяет *работы по оценке состояния* и результатов какой-либо работы. Данный процесс может использоваться двумя любыми сторонами, когда одна из сторон (проверяющая) проверяет другую сторону (проверяемую) на совместном совещании.
- **Аудит**. Определяет *работы по определению соответствия требованиям*, планам и договору. Данный процесс может использоваться двумя сторонами, когда одна из сторон (проверяющая) контролирует программные продукты или работы другой стороны (проверяемой).
- **Решение проблем**. Определяет *процесс анализа и устранения проблем* (включая несоответствия), независимо от их характера и источника, которые были обнаружены во время осуществления разработки, эксплуатации, сопровождения или других процессов.

Организационными процессами являются:

- **Управление**. Определяет *основные работы по управлению*, включая управление проектом, при реализации процессов жизненного цикла.
- **Создание инфраструктуры**. Определяет основные *работы по созданию* основной *структуры процесса* жизненного цикла.
- **Усовершенствование**. Определяет основные *работы,* которые организация (заказчика, поставщика, разработчика, оператора, персонала сопровождения или администратора другого процесса) выполняет *при создании, оценке, контроле и усовершенствовании* выбранных процессов жизненного цикла.
- **Обучение**. Определяет *работы по* соответствующему *обучению* персонала.

Примечание: Стандарт ISO 12207 разрабатывался 9 лет и достаточно быстро устарел. В 1998г. выходит новый стандарт ISO/IEC TR 15504: «Information Technology – Software Process Assessment (Оценка процессов разработки ПО)». В этом документе рассматриваются вопросы аттестации, определения зре-

лости и усовершенствования процессов жизненного цикла ПО. Один из разделов документа содержит *новую классификацию процессов* жизненного цикла: базовый; расширенный; новый; составляющий; расширенный составляющий, – являющуюся развитием стандарта ISO 12207.

В соответствии с новой классификацией в трех группах процессов вводятся пять категорий процессов:

- основные процессы:
  - **CUS**: потребитель-поставщик (приобретение, поставка, эксплуатация);
  - **ENG**: инженерная (разработка, сопровождение);
- вспомогательные процессы:
  - **SUP**: вспомогательная (аналогично ISO 12207);
- организационные процессы:
  - **MAN**: управленческая (административное управление, управление проектами, управление качеством, управление рисками);
  - **ORG**: организационная (организационные установки, усовершенствование (создание, аттестация, усовершенствование), административное управление кадрами, создание инфраструктуры, измерение, повторное использование).

Примечание: Наиболее важными в рамках учебного процесса являются инженерные процессы, которые непосредственно определяют, реализуют или поддерживают программный продукт, его взаимодействие с системой и документацию на него. В тех случаях, когда система целиком состоит из программных средств, инженерные процессы имеют отношение только к созданию и поддержанию этих программных средств.

3.2.1 Инженерные процессы

Процесс разработки (development process) в соответствии со стандартом предусматривает действия и задачи, выполняемые разработчиком, и охватывает работы по созданию ПО и его компонентов в соответствии с созданными требованиями, включая оформление проектной и эксплуатационной документации, а также подготовку материалов, необходимых для

проверки работоспособности и соответствия качества программных продуктов, материалов, необходимых для обучения персонала, и т.д.

По стандарту процесс разработки включает следующие действия:

- *подготовительную работу* – выбор модели ЖЦ (см. далее п. 3.3), стандартов, методов и средств разработки, а также составление плана работ;
- *анализ требований к системе* – определение ее функциональных возможностей, пользовательских требований, требований к надежности и безопасности, требований к внешним интерфейсам и т.п.;
- *проектирование архитектуры системы* – определение состава необходимого оборудования, программного обеспечения и операций, выполняемых обслуживающим персоналом;
- *анализ требований к ПО* – определение функциональных возможностей, включая характеристики производительности, среды функционирования компонентов, внешних интерфейсов, спецификаций надежности и безопасности, эргономических требований, требований к используемым данным, установке, приемке, пользовательской документации, эксплуатации и сопровождению;
- *проектирование архитектуры ПО* – определение структуры ПО, документирование интерфейсов его компонентов, разработка предварительной версии пользовательской документации, а также требований к тестам и плана интеграции;
- *детальное проектирование ПО* – подробное описание компонентов ПО и интерфейсов между ними, обновление пользовательской документации, разработка и документирование требований к тестам и плана тестирования, обновление плана интеграции ПО;
- *кодирование и тестирование ПО* – разработка и документирование каждого компонента, тестирование компонентов, обновление пользовательской документации, обновление плана интеграции ПО;
- *интеграцию ПО* – сборка программных компонентов в соответствии с планом интеграции и тестирование ПО на соответствие квалификационным требованиям, представляющим собой набор критериев и условий, которые необходимо выполнить, чтобы квалифицировать ПП как соответствующий своим спецификациям и готовый к использованию в заданных условиях эксплуатации;

- *квалификационное тестирование ПО* – тестирование ПО в присутствии заказчика для демонстрации его соответствия требованиям и готовности к эксплуатации, при этом проверяется также готовность и полнота технической и пользовательской документации;
- *интеграцию системы* – сборка всех компонентов системы, включая ПО и оборудование;
- *квалификационное тестирование системы* – тестирование системы на соответствие требованиям к ней и проверка полноты и оформления документации;
- *установку ПО* – установка ПО на оборудовании заказчика и проверка его работоспособности;
- *приемку ПО* – оценка результатов квалификационного тестирования ПО и системы в целом и документирование результатов оценки совместно с заказчиком, окончательная передача системы заказчику.

Рассматриваемый стандарт только называет и определяет процессы ЖЦ ПО, не конкретизируя в деталях, как реализовать или выполнить то или иное действие или задачу. Эти вопросы регламентируются соответствующими методологиями, построенными на модели ЖЦ ПО.

3.3 Модель жизненного цикла ПО

Модель жизненного цикла ПО (life cycle model) описывает набор фаз (этапов, стадий) проекта по созданию ПО, в которых выполняются отдельные процессы, разбитые на операции и задачи. В глоссарии PMI[13] [28] даются следующие определения этих понятий:

Жизненный цикл проекта – набор обычно последовательных фаз проекта, количество и состав которых определяется потребностями управления проектом организацией или организациями, участвующими в проекте.

Фаза проекта – объединение логически связанных операций проекта, обычно завершающихся достижением одного из основных результатов (состав, количество и порядок выполнения фаз определяются особенностью проекта).

[13] **PMI** - Project Management Institute - Международный Институт Проектного Менеджмента (Управления Проектами)

Процесс – набор взаимосвязанных ресурсов и работ, благодаря которым входные воздействия преобразуются в выходные результаты.

Операция, работа – элемент работ проекта. У операций обычно имеется ожидаемая длительность, потребность в ресурсах, стоимость. Операции могут далее подразделяться на задачи.

К настоящему времени сложилось несколько типовых моделей ЖЦ ПО, которые проявили себя в определенных условиях, имеют определенные преимущества, недостатки и условия применимости. Эти модели определяются особенностью задач, ограничениями на ресурсы, опытом разработчиков и т.д. и устанавливают некоторые принципы организации модели жизненного цикла ПО [2].

3.3.1 Каскадная модель разработки ПО

Принципы. В изначально существовавших однородных ПС каждое приложение представляло собой единое целое. Для разработки такого типа приложений применялся каскадный способ (1970–1985). Его основной характеристикой является разбиение всей разработки на этапы, причем переход с одного этапа на следующий происходит только после того, как будет полностью завершена работа на текущем. Каждый этап завершается выпуском полного комплекта документации, достаточной для того, чтобы разработка могла быть продолжена другой командой разработчиков [10]. *Основа модели* – сформулированные требования в техническом задании (ТЗ), которые меняться не должны. *Критерий качества результата* – соответствие продукта установленным требованиям.

Каскадная модель (водопад – waterfall) включает выполнение следующих фаз (рисунок 15):

- **исследование концепции** — происходит исследование требований, разрабатывается видение продукта и оценивается возможность его реализации;
- **определение требований** — определяются программные требования для предметной области системы, предназначение, линии поведения, производительность и интерфейсы;
- **разработка проекта** — разрабатывается и формулируется логически последовательная техническая характеристика программной системы, включая структуры данных, архитектуру ПО, интер-

фейсные представления и процессуальную (алгоритмическую) детализацию;

- **реализация** — эскизное описание ПО превращается в полноценный программный продукт. Результат: исходный код, база данных и документация. В реализации обычно выделяют два этапа: реализацию компонентов ПО и интеграцию компонентов в готовый продукт. На обоих этапах выполняется кодирование и тестирование, которые тоже иногда рассматривают как два подэтапа;
- **эксплуатация и поддержка** – подразумевает запуск и текущее обеспечение, включая предоставление технической помощи, обсуждение возникших вопросов с пользователем, регистрацию запросов пользователя на модернизацию и внесение изменений, а также корректирование или устранение ошибок;
- **сопровождение** — устранение программных ошибок, неисправностей, сбоев, модернизация и внесение изменений.

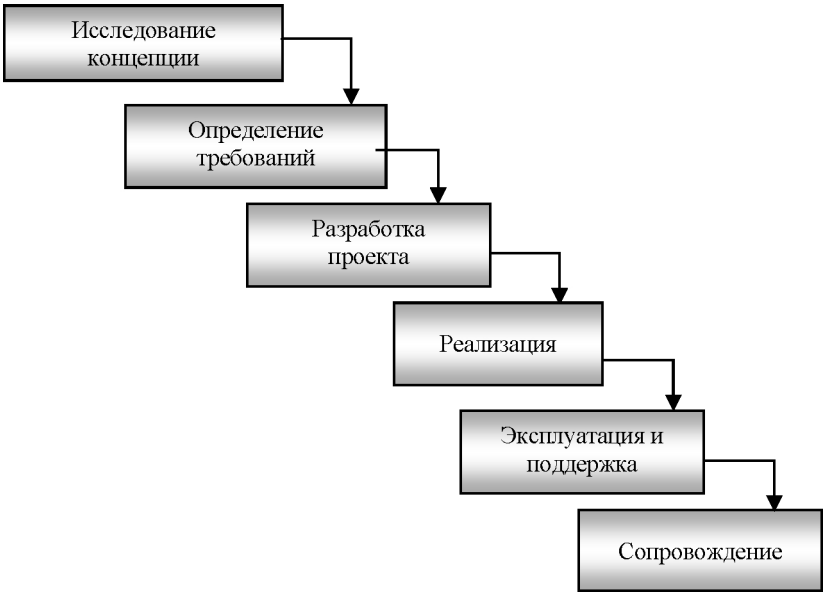


Рисунок 15 – Каскадная схема разработки программного обеспечения

Основными ***принципами*** каскадной модели являются:

- строго последовательное выполнение фаз;
- каждая последующая фаза начинается лишь тогда, когда полностью завершено выполнение предыдущей фазы;
- каждая фаза имеет определенные критерии входа и выхода: входные и выходные данные;
- каждая фаза полностью документируется;
- переход от одной фазы к другой осуществляется посредством формального обзора с участием заказчика.

Преимущества и недостатки. Каскадная модель имеет следующие ***преимущества***:

- *проста* и *понятна* заказчикам, так как часто используется другими организациями для отслеживания проектов, не связанных с разработкой ПО;
- проста и удобна в применении;
- процесс *разработки* выполняется *поэтапно*;
- ее структурой может руководствоваться даже слабо подготовленный в техническом плане или неопытный персонал;
- она способствует осуществлению строгого контроля менеджмента проекта;
- каждую стадию могут выполнять независимые команды (*все документировано*);
- позволяет достаточно точно планировать сроки и затраты.

При использовании каскадной модели для «неподходящего» проекта могут проявляться следующие ее ***недостатки***:

- *невозможно вернуться* на одну или две фазы назад, чтобы исправить какую-либо проблему или недостаток, это приведет к значительному увеличению затрат и сбою в графике;
- *интеграция* компонентов, на которой обычно выявляется большая часть ошибок, выполняется *в конце разработки*, что сильно увеличивает стоимость устранения ошибок;
- *запаздывание результатов*, если в процессе выполнения проекта требования изменились, то получится устаревший результат.

Примечание: Недостатки каскадной модели особо остро проявляются в случае, когда трудно (или невозможно) сформули-

ровать требования или требования могут меняться в процессе выполнения проекта. В этом случае разработка ПО имеет принципиально циклический характер.

Применимость. Каскадная модель впервые четко сформулирована в 1970 году Ройсом [33], на начальном периоде она сыграла ведущую роль как метод регулярной разработки сложного ПО. В семидесятых-восьмидесятых годах XX века модель была принята как стандарт министерства обороны США. Со временем недостатки каскадной модели стали проявляться все чаще, и возникло мнение, что она безнадежно устарела.

Между тем, каскадная модель не утратила своей *актуальности* при решении следующих типов задач:

- ✓ требования и их реализация максимально четко определены и понятны; используется неизменяемое определение продукта и вполне понятные технические методики (задачи научно-вычислительного характера, операционные системы и компиляторы, системы реального времени управления конкретными объектами);
- ✓ повторная разработка типового продукта (автоматизированного бухгалтерского учета, начисления зарплаты и т.п.);
- ✓ выпуск новой версии уже существующего продукта, если вносимые изменения вполне определены и управляемы (перенос уже существующего продукта на новую платформу).

В настоящее время используется усовершенствованная каскадная модель (**итерационная**), которая позволяет при необходимости вернуться на любой предыдущий этап и внести необходимые изменения (рисунок 16).

Примечание: Итерационная модель является более жизненной, чем классическая каскадная модель, т.к. создание ПО всегда связано с устранением ошибок. Следует отметить, что уже в первых работах, посвященных каскадной модели, отмечалось это обстоятельство и предлагался итерационный вариант каскадной модели. *Практически все применяемые модели жизненного цикла имеют итерационный характер, но цели итераций могут быть разными.* Основная опасность использования такой схемы связана с тем, что разработка никогда не будет завершена, постоянно находясь в состоянии уточнения и усовершенствования.

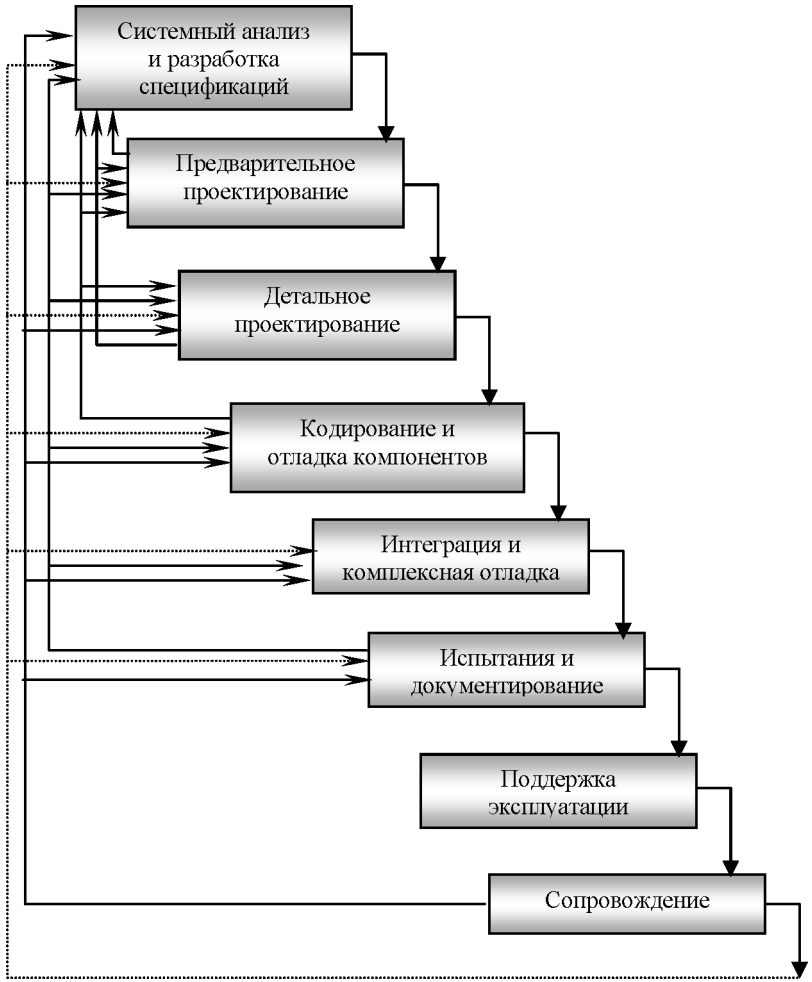


Рисунок 16 - Обобщенная каскадная модель ЖЦ ПО

3.3.2 Спиральная модель разработки ПО

Принципы. На практике, при решении достаточно большого количества задач, разработка ПО имеет циклический характер, когда после выполнения некоторых стадий приходится возвращаться на предыдущие. Это может происходить по следующим основным причинам:

- *из-за ошибок разработчиков*, допущенных на ранних стадиях и выявленных на поздних стадиях (это ошибки анализа, проектирования, кодирования, выявляемые, как правило, на стадии тестирования);
- *из-за изменения требований* в процессе разработки – «ошибки» заказчиков (это или неготовность заказчиков сформулировать требования, или изменения требований, вызванные изменениями ситуации в процессе разработки (изменения рынка, новые технологии и т.п.)).

Спиральная модель была предложена как альтернатива каскадной модели для преодоления перечисленных выше проблем и учитывает повторяющийся характер разработки программного обеспечения.

Основными ***принципами*** спиральной модели являются:

- разработка вариантов продукта, соответствующих различным вариантам требований с возможностью вернуться к более ранним вариантам;
- создание *прототипов* ПО как средства общения с заказчиком для уточнения и выявления требований;

> ***Прототипом*** называют действующий программный продукт, реализующий отдельные функции и внешние интерфейсы разрабатываемой ПС.

- планирование следующих вариантов с оценкой альтернатив и анализом рисков, связанных с переходом к следующему варианту;
- переход к разработке следующего варианта до завершения предыдущего в случае, когда риск завершения очередного варианта (прототипа) становится неоправданно высок;
- использование каскадной модели как схемы разработки очередного варианта;
- активное привлечение заказчика к работе над проектом (заказчик участвует в оценке очередного прототипа ПО, в уточнении требований при переходе к следующему, в оценке предложенных альтернатив очередного варианта и оценке рисков).

В спиральной модели разработка вариантов продукта представляется как набор циклов раскручивающейся спирали (рисунок 17). Каждому циклу спирали соответствует такое же количество стадий, как и в модели кас-

кадного процесса. При этом начальные стадии, связанные с анализом и планированием, представлены более подробно с добавлением новых элементов. В каждом цикле выделяются четыре базовые фазы:

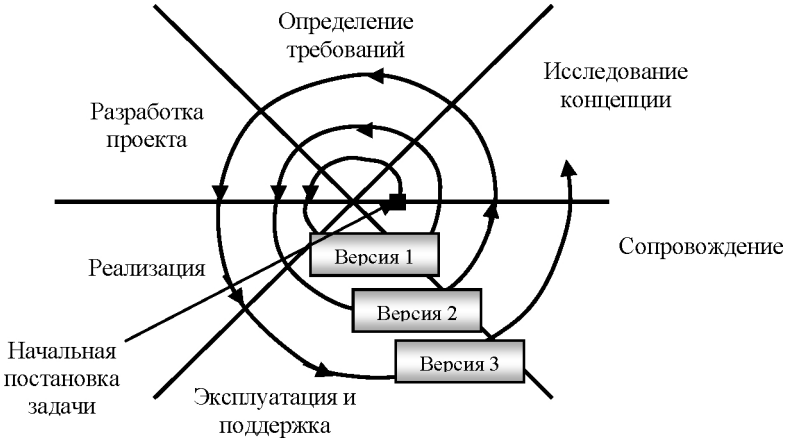


Рисунок 17 - Спиральная модель ЖЦ ПС

- определение целей, альтернативных вариантов и ограничений;
- оценка альтернативных вариантов, идентификация и разрешение рисков;
- разработка продукта следующего уровня;
- планирование следующей фазы.

Разработка проекта начинается с анализа *общей постановки задачи*. Здесь на первой фазе определяются общие цели, устанавливаются предварительные ограничения, определяются возможные альтернативы подходов к решению задачи. Далее проводится оценка подходов, устанавливаются их риски. На шаге разработки создается концепция (видение) продукта и путей его создания.

Следующий цикл начинается с *планирования требований* и деталей ЖЦ продукта для оценки затрат. На фазе определения целей устанавливаются альтернативные варианты требований, связанные с ранжировкой требований по важности и стоимости их выполнения. На фазе оценки устанавливаются риски вариантов требований. На фазе разработки – специ-

фикация требований (с указанием рисков и стоимости), готовится ***демо-версия*** ПО для анализа требований заказчиком.

Следующий цикл – *разработка проекта* – начинается с планирования разработки. На фазе определения целей устанавливаются ограничения проекта (по срокам, объему финансирования, ресурсам и т.п.), определяются альтернативы проектирования, связанные с альтернативами требований, применяемыми технологиями проектирования, привлечением субподрядчиков и т.п. На фазе оценки альтернатив устанавливаются риски вариантов и делается выбор варианта для дальнейшей реализации. На фазе разработки выполняется проектирование и создается демо-версия, отражающая основные проектные решения.

Следующий цикл – *реализация проекта* – также начинается с планирования. Варианты реализации могут отличаться из-за применяемых технологий реализации и привлекаемых ресурсов. Оценка альтернативных вариантов и связанных с ними рисков на этом цикле определяется степенью «проработанности» технологий и «качеством» имеющихся ресурсов. Фаза разработки выполняется по каскадной модели, результатом является действующий вариант (прототип) продукта.

Таким образом, *особенностями* спиральной модели являются:

- ✓ до начала разработки ПО проходит несколько полных циклов анализа требований и проектирования;
- ✓ количество циклов модели (как в части анализа и проектирования, так и в части реализации) не ограничено и определяется сложностью и объемом задачи;
- ✓ в модели предусматриваются возвраты на оставленные варианты при изменении стоимости рисков.

Преимущества и недостатки. Спиральная модель (по отношению к каскадной) имеет следующие очевидные ***преимущества***:

- более *тщательное проектирование* (несколько начальных итераций) с оценкой результатов проектирования, что позволяет выявить ошибки проектирования на более ранних стадиях и уменьшить как стоимость проекта, так и время его разработки;
- *поэтапное уточнение требований* в процессе выполнения итераций, что позволяет более точно удовлетворить требованиям заказчика;

- *участие заказчика в выполнении проекта* с использованием прототипов программы. Заказчик видит, что и как создается, не выдвигает необоснованных требований, оценивает реальные объемы финансирования;
- планирование и управление рисками при переходе на следующие итерации позволяет разумно планировать использование ресурсов и обосновывать финансирование работ;
- возможность разработки сложной программной системы «по частям», выделяя на первых этапах наиболее значимые требования.

Основные ***недостатки*** спиральной модели связаны с ее *сложностью*:

- сложность *анализа и оценки рисков* при выборе вариантов;
- сложность *поддержания версий* продукта (хранение версий, возврат к ранним версиям, комбинация версий);
- сложность *оценки точки перехода* на следующий цикл;
- *бесконечность модели* – на каждом витке заказчик может выдвигать новые требования, которые приводят к необходимости следующего цикла разработки.

Применимость. Циклический характер разработки ПО отражен в спиральной модели ЖЦ, предложенной Б. Боэмом в 1988 году [24].

Спиральную модель целесообразно применять при следующих условиях:

- ✓ когда пользователи не уверены в своих потребностях или когда требования слишком сложны и могут меняться в процессе выполнения проекта и необходимо прототипирование для анализа и оценки требований;
- ✓ когда достижение успеха не гарантировано и необходима оценка рисков продолжения проекта;
- ✓ когда проект является сложным, дорогостоящим и обоснование его финансирования возможно только в процессе его выполнения;
- ✓ когда речь идет о применении новых технологий, что связано с риском их освоения и достижения ожидаемого результата;
- ✓ при выполнении очень больших проектов, которые в силу ограниченности ресурсов можно делать только по частям.

3.3.3 Другие типы моделей жизненного типа

Существуют некоторые другие типы моделей, которые можно рассматривать как «промежуточные» между каскадной и спиральной моделями, они используют отдельные преимущества каждой из базовых моделей и достигают успеха для определенных типов задач.

V-образная модель была создана как разновидность обобщенной (итерационной) каскадной модели. *Целью итераций* в этой модели является обеспечение *процесса тестирования*: тестирование продукта обсуждается, проектируется и планируется на ранних этапах жизненного цикла разработки. План испытания приемки заказчиком разрабатывается на этапе планирования, а компоновочного испытания системы – на фазах анализа, разработки проекта и т.д. Этот процесс разработки планов испытания обозначен пунктирной линией между прямоугольниками V-образной модели (см. рисунок 18). Помимо планов, на ранних этапах разрабатываются также и тесты, которые будут выполняться при завершении параллельных этапов.

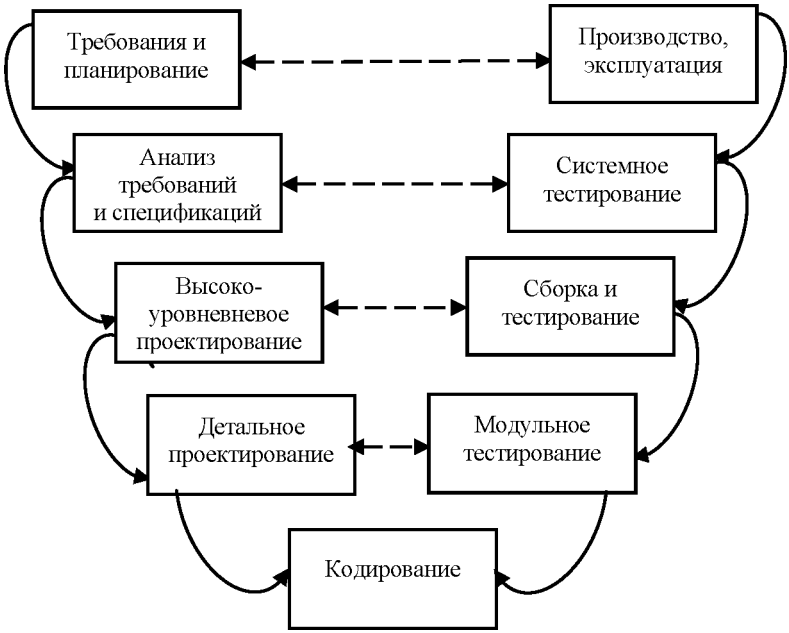


Рисунок 18 – V-образная модель жизненного цикла ПО

Инкрементная (пошаговая) модель представляет собой процесс поэтапной реализации всей системы и поэтапного наращивания (приращения) функциональных возможностей. На первом шаге необходим полный заранее сформулированный набор требований, которые делятся по некоторому признаку на части. Далее выбирается первая группа требований и выполняется полный проход по каскадной модели. После того как первый вариант системы, выполняющий первую группу требований, сдан заказчику (см. рисунок 19), разработчики переходят к следующему шагу (второму инкременту) по разработке варианта, выполняющего вторую группу требований т.д.

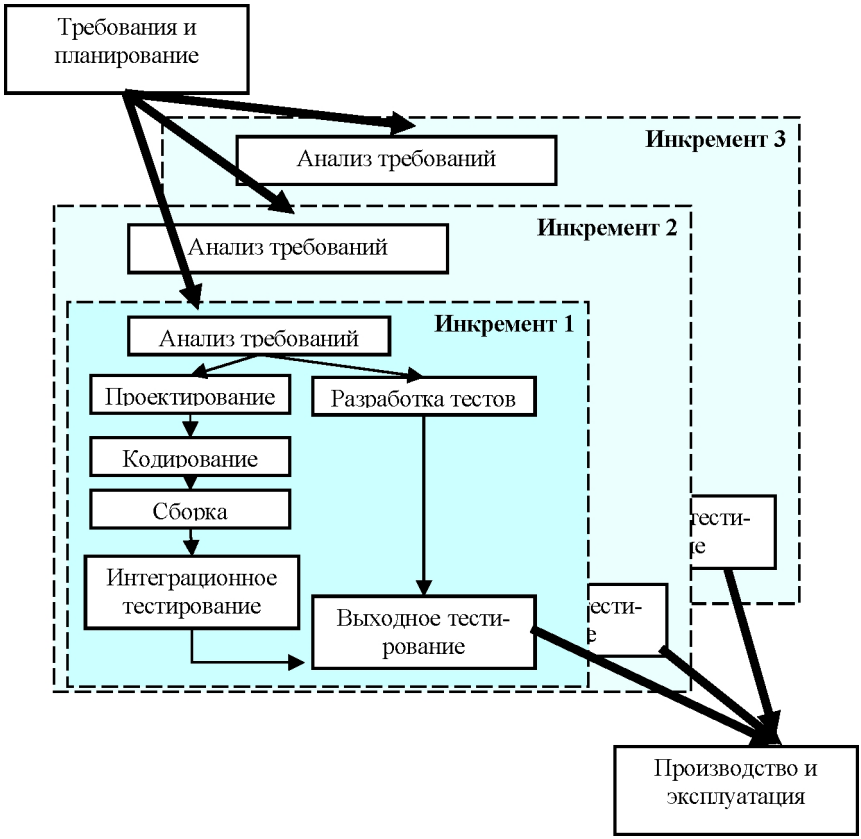


Рисунок 19 – Инкрементная модель ЖЦ ПС

Особенностью инкрементной модели является разработка приемочных тестов на этапе анализа требований, что упрощает приемку варианта заказчиком и устанавливает четкие цели разработки очередного варианта системы. Инкрементная модель особенно эффективна в случае, когда задача разбивается на несколько относительно независимых подзадач (разработка подсистем «Зарплата», «Бухгалтерия», «Склад», «Поставщики») в рамках единого проекта, для внутренней итерации можно использовать не только каскадную, но и другие типы моделей.

Модель быстрого прототипирования предназначена для быстрого создания прототипов ПС с целью уточнения требований заказчика и поэтапного развития системы в конечный продукт. Скорость (высокая производительность) выполнения проекта обеспечивается планированием разработки прототипов и участием заказчика в процессе разработки.

Начало жизненного цикла разработки помещено в центре эллипса (рисунок 20). Совместно с пользователем разрабатывается предварительный **план проекта** на основе его предварительных требований и формируется документ, описывающий в общих чертах примерные графики и результирующие данные.

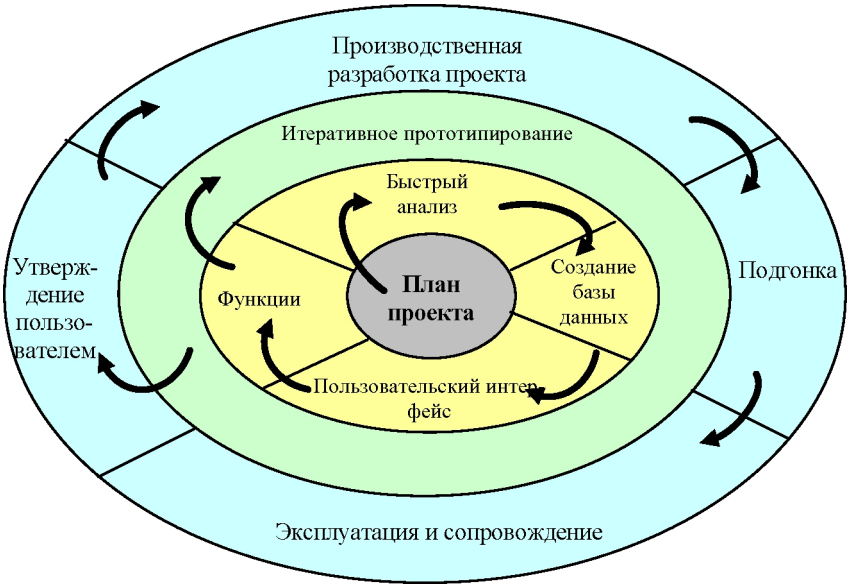


Рисунок 20 – Модель быстрого прототипирования

После этого переходят к созданию исходного прототипа на основе быстрого анализа, проекта базы данных, пользовательского интерфейса и некоторых функций. Затем начинается итерационный цикл быстрого прототипирования: разработчик проекта демонстрирует очередной прототип, пользователь (заказчик) оценивает его функционирование, совместно определяются проблемы и пути их преодоления для перехода к следующему прототипу. Этот процесс продолжается до тех пор, пока пользователь не согласится, что очередной прототип в точности отображает все его требования.

Получив одобрение пользователя, быстрый прототип преобразуют в детальный проект, настраивают на производственное использование, и он становится полностью действующей системой.

При разработке производственной версии программы, может понадобиться более высокий уровень функциональных возможностей, могут измениться системные ресурсы, необходимые для обеспечения полной рабочей нагрузки, или появятся ограничения во времени. После этого следуют тестирование в предельных режимах, определение измерительных критериев и настройка, а затем, как обычно, функциональное сопровождение.

3.3.4 Технология быстрой разработки приложений RAD

Один из подходов к разработке ПО в рамках спиральной модели ЖЦ – получившая широкое распространение методология (технология) быстрой разработки приложений RAD (***Rapid Application Development***) [6]. Данная модель очень хорошо подходит к разработке учебных программ, т.к. включает в себя три составляющие:

- небольшую команду программистов (от 2 до 10 человек);
- короткий, но тщательно проработанный производственный график (от 2 до 6 мес.);
- повторяющийся цикл, при котором разработчики по мере того, как приложение начинает обретать форму, запрашивают и реализуют в продукте требования, полученные через взаимодействие с заказчиком.

Рассмотрим данную модель более подробно. Команда разработчиков должна представлять собой группу профессионалов, имеющих опыт в

анализе, проектировании, генерации кода и тестировании ПО с использованием CASE-средств, способных хорошо взаимодействовать с конечными пользователями и трансформировать их предложения в рабочие прототипы. Жизненный цикл ПО по методологии RAD состоит из **четырёх фаз** (рисунок 21):

1. Анализа и планирования требований;
2. Проектирования;
3. Построения;
4. Внедрения.

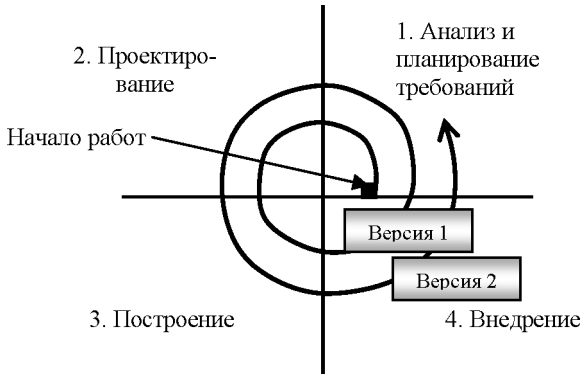


Рисунок 21 – Модель ЖЦ по технологии RAD

На **первой фазе** ***анализа и планирования требований*** пользователи системы определяют функции, которые она должна выполнять, выделяют наиболее приоритетные из них, требующие проработки в первую очередь, описывают информационные потребности (связи). Формулирование требований к системе осуществляется в основном силами пользователей под руководством специалистов-разработчиков. Ограничивается масштаб проекта, устанавливаются временные рамки для каждой из последующих фаз. Кроме того, определяется сама возможность реализации проекта в заданных размерах финансирования, на имеющихся аппаратных средствах и т.п.

Результатом фазы должны быть: список расставленных по приоритету функций будущей ПС; предварительная функциональная модель ПС; предварительная информационная модель ПС.

На **второй фазе *проектирования*** часть пользователей принимают участие в техническом проектировании системы под руководством специалистов-разработчиков и, взаимодействуя с ними, уточняют и дополняют требования к системе, которые не были выявлены на предыдущей фазе. Более подробно рассматриваются *процессы системы*. При необходимости корректируется функциональная модель, создаются частичные прототипы: экранов, отчетов, устраняющие неясности или неоднозначности. Устанавливаются требования *разграничения доступа к данным*. На этой же фазе происходит определение необходимой документации. После детального определения состава процессов оценивается количество функциональных элементов разрабатываемой системы и ***принимается решение о разделении системы на подсистемы.***

Результатом данной фазы должны быть: общая информационная модель системы; функциональные модели системы в целом и подсистем; точно определенные интерфейсы между автономно разрабатываемыми подсистемами; построенные прототипы экранов, отчетов, диалогов.

> В отличие от традиционного подхода, при котором использовались специфические средства прототипирования, не предназначенные для построения реальных приложений, а прототипы выбрасывались после того, как выполняли задачу устранения неясностей в проекте, в *подходе RAD каждый прототип развивается в часть будущей системы*. Таким образом, на следующую фазу передается более полная и полезная информация.

На **третьей фазе *построения*** выполняется непосредственно сама быстрая разработка приложения (реализация подсистем). На данной фазе разработчики производят итеративное построение реальной системы на основе полученных в предыдущей фазе моделей, а также требований нефункционального характера. Конечные пользователи на этой фазе оценивают получаемые результаты и вносят коррективы, если в процессе разработки система перестает удовлетворять определенным ранее требованиям. Тестирование системы осуществляется в процессе разработки.

После окончания разработки подсистем производится постепенная интеграция данной части системы с остальными, формируется полный программный код, выполняется тестирование системы в целом. Завершается физическое проектирование системы: определяется необходимость

распределения данных; осуществляется анализ использования данных; производится физическое проектирование базы данных; определяются требования к аппаратным ресурсам; определяются способы увеличения производительности; завершается разработка документации проекта.

Результатом фазы является готовая система, удовлетворяющая всем согласованным требованиям.

На **четвертой фазе *внедрения*** производятся обучение пользователей, организационные изменения и параллельно с внедрением новой системы осуществляется работа с существующей системой (до полного внедрения новой). Так как фаза построения достаточно непродолжительна, планирование и подготовка к внедрению должны начинаться заранее, как правило, на этапе проектирования системы.

> Технология RAD (как и любая другая) не может претендовать на универсальность, она хороша в первую очередь для относительно небольших проектов, разрабатываемых для конкретного заказчика. Она неприменима для разработки операционных систем; сложных расчетных программ с большим объемом программного кода и сложными уникальными алгоритмами управления; приложений, в которых отсутствует ярко выраженная интерфейсная часть, наглядно определяющая логику работы системы (приложения реального времени), так как итерационный подход предполагает, что несколько первых версий не будут полностью соответствовать требованиям.

В заключение перечислим **основные принципы технологии RAD:**

- разработка приложений итерациями;
- необязательность полного завершения работ на каждом этапе ЖЦ;
- обязательное вовлечение пользователей на этапе разработки;
- использование прототипирования, позволяющего выяснить и удовлетворить все требования конечного пользователя;
- тестирование и развитие проекта одновременно с разработкой;
- грамотное руководство разработкой, четкое планирование и контроль выполнения работ.

КОНТРОЛЬНЫЕ ВОПРОСЫ К ГЛАВЕ 3:

1. Что такое стандартизация и сертификация программного продукта?
2. Какие существуют типы стандартов?
3. Перечислите наиболее известные стандарты жизненного цикла, которые использовались для разработки программного обеспечения?
4. Что такое жизненный цикл ПО?
5. Перечислите основные этапы жизненного цикла ПО. Что такое процесс, действие, задача?
6. Какие типы процессов и конкретные процессы вы запомнили?
7. Расскажите об основных инженерных процессах жизненного цикла ПО.
8. Что такое модель жизненного цикла ПО? Дайте определения основных понятий, связанные с понятием «модель».
9. Какие типы моделей вы знаете? В чем их преимущества, недостатки, область применимости?
10. Что вы можете сказать об особенностях каскадной модели жизненного цикла?
11. В чем отличие обобщенной каскадной модели от базовой?
12. Что вы можете сказать об особенностях спиральной модели жизненного цикла?
13. Перечислите составляющие технологии RAD. Для разработки каких типов ПО можно применять технологию RAD?
14. Опишите основные фазы жизненного цикла по технологии RAD.
15. Перечислите основные принципы технологии RAD.

СПИСОК ЛИТЕРАТУРЫ

1. Аптекарь М. Д., Рамазанов С. К., Фрегер Г. Е. История инженерной деятельности. – Киев, 2003. – 204 с. : ил.
2. Арчибальд Р. Модели жизненного цикла высокотехнологичных проектов. http://www.pmprofy.ru/content/rus/107/1073-article.html
3. Брукс Ф. Мифический человеко-месяц или как создаются программные системы. – СПб. : Символ-плюс, 1999. – 321 с. : ил.
4. Буч Г. Объектно-ориентированное проектирование с примерами применения. – М.: Конкорд, 1992. – 586с. : ил.
5. Буч Г. Объектно-ориентированный анализ и объектно-ориентированное проектирование на С++. – М. : Бином, – 2001. – 558 с. : ил.
6. Вендров А. М. CASE-технологии. Современные методы и средств проектирования информационных систем. – М. : Финансы и статистика, – 1999. – 256 с. : ил.
7. Вирт Н. Алгоритмы + структуры данных = программы : Пер. с англ. – М. : Мир, 1985. – 406 с.: ил.
8. Дал О., Дейкстра Э., Хоор К. Структурное программирование: Пер. с англ. – М.: Мир, 1975. – 247 с. : ил.
9. Дзержинский Ф. Я., Калиниченко И.М. Дисциплина программирования : концепция и опыт реализации методических средств программной инженерии. – М.: ЦНИИ информации и технико-экономических исследований по атомной науке и технике, 1988. – 245 с. : ил.
10. Жоголев Е. А. Технологии программирования. – М. : Научный мир, 2004. – 216 с. : ил.
11. Закон РФ № 149-ФЗ от 29.07.2006. «Об информации, информационных технологиях и защите информации»// Российская газета, № 165 от 27.07.2006 г.
12. Иванова Г. С. Технология программирования: Учебник для вузов. – 2-е изд., стереотип. – М. : Изд-во МГТУ им. Н.Э.Баумана, 2003. – 320 с.: ил.
13. Калянов Г. Н. CASE: Структурный системный анализ (автоматизация и применение). – М. : «Лори», 1996. – 356 с. : ил.
14. Кораблин М. А. Программирование, ориентированное на объекты: Учебное пособие. – Самара: изд-во СГАУ, 1994. – 94 с.
15. Леоненков А. В. Самоучитель UML. – СПб : BXB Петербург, – 2001. – 304 с. : ил.
16. Липаев В. В. Качество программного обеспечения. – М.: Финансы и статистика, 1983. – 263 с. : ил.
17. Липаев В. В. Отладка сложных программ: Методы, средства, технология. –М. : Энергоатомиздат, 1993. – 384 с. : ил.
18. Липаев В. В., Филиппов Е. Н. Мобильность программ и данных в открытых информационных системах. – М. : Научная книга, 1997. – 297 с. : ил.

19. Майерс Г. Надежность программного обеспечения. – М. : Мир, 1980. – 375 с. : ил.
20. Ожегов С. И. Словарь русского языка. – М. : Мир и образование, 2006. – 1328 с.
21. Технология проектирования комплексов программ АСУ/ В. В. Липаев, Л. А. Серебровский, П. Г. Гаганов и др.; Под ред. Ю. В. Афанасьева, В. В. Липаева. – М. : Радио и связь, 1983. – 256 с. : ил.
22. Хювенен Э., Сеппянен Й. Мир ЛИСПа: Пер. с финск. В 2 т. Т.1 : Введение в язык Лисп и функциональное программирование.– М. : Мир, 1990. – 447 с. : ил.
23. Хювенен Э., Сеппянен Й. Мир ЛИСПа: Пер. с финск. В 2 т. Т.2 : Методы и системы программирования.– М. : Мир, 1990. – 319 с. : ил.
24. Boehm B.«A Spiral Model of Software Development and Enhancement», IEEE Computer, Vol. 21, No. 5, pp. 61–72, 1988.
25. Courtois P. June 1985. On Time and Space Decomposition of Complex Structures. Communications of the ACM vol.28(6), p.596.
26. Criteria for Evaluation of Software. ISO TC97/SC7 #383.
27. Dijktra E. 1979. Programming Considered as a Human Activity. Classics in Software Engineering. New York, NY: Yourdon Press.
28. http://www.pmi.ru/glossary/.
29. http://www.staratel.com/iso/InfTech/DesignPO/ISO12207/ISO12207-99/ISO12207.htm.
30. Microsoft Corporation. Принципы проектирования и разработки программного обеспечения. Учебный курс MCSD: Пер. с англ. – М.: Издательско-торговый дом «Русская редакция», 2000. –608 с. : ил.
31. Parnas D., Clements P., Weiss D. 1983. Enhancing Reusability with Information Hiding. Proceedings of the Workshop on Reusability in Programming. Stratford, CT: ITT Programming. p.241.
32. Rechtin E. October 1992. The Art of Systems Architecting. IEEE Spectrum, vol.29 (10), p.66.
33. Royce W.W. Managing the Development of Large Software Systems. http://facweb.cti.depaul.edu/jhuang/is553/Royce.pdf.
34. Shaw M. October 1984. Abstraction Techniques in Modern Programming Languages. IEEE Software vol.1 (4).
35. Simon H. 1982. The Sciences of the Artificial. Cambridge, MA: The MIT Press. – p.218.
36. Sommerville I. Software engineering. – Addison-Wesley Publishing Company, 1992. p.87.
37. Tesler L. August 1981. The Smalltalk Environment. Byte vol.6(8), p.142.
38. Yonezawa A., Tokoro M. 1987. Objectt-Oriented Concurrent Programming. Cambridge, MA: The MIT Press.

СПИСОК ТЕРМИНОВ

Название термина		Определение
Русское	Английское	
Абстракция	***Abstraction***	Существенные характеристики объекта, которые отличают его от всех других объектов и четко определяют его концептуальные границы для наблюдателя
Алгоритм	***Algorithm***	Заранее заданная последовательность четко определенных правил или команд для получения решения задачи за конечное число шагов
Алгоритмическая декомпозиция	***Algorithmic decomposition***	Процесс разделения системы на части, каждая из которых отражает этап общего процесса. Применение структурного подхода к проектированию приводит к алгоритмической декомпозиции, которая фокусируется на потоке управления в системе
Верификация	***Verification***	Процесс определения того, отвечает ли текущее состояние разработки, достигнутое на данном этапе, требованиям этого этапа
Данные	***Data***	Представление фактов и идей в формализованном виде, пригодном для передачи и переработки в некоем процессе
Жизненный цикл программы	***Software Lifetime Cycle***	Непрерывный процесс, который начинается с момента принятия решения о необходимости создания программы и заканчивается в момент ее полного изъятия из эксплуатации
Иерархия	***Hierarchy***	Упорядочение абстракций, расположение их по уровням. В терминах иерархии «часть/целое» объект находится на более высоком уровне абстракции, чем другие, если он строится на основе этих объектов; в терминах иерархии «общее/частное» высокоуровневые абстракции носят более обобщенный характер, чем низкоуровневые
Инкапсуляция	***Encapsulation***	Объединение в классе данных

		(*свойств*) и *методов* (процедур обработки), сокрытие отдельных деталей внутреннего устройства классов от внешних по отношению к нему объектов или пользователей
Информационная среда	***Data medium***	Совокупность носителей данных, используемых при какой-либо обработке данных
Носитель данных		Материал с определенными физическими свойствами, который может использоваться для хранения данных (магнитные лента или диск, оптический диск, бумага для распечатки)
Информация	***Information***	Смысл, который придается данным при их представлении
		Сведения о лицах, предметах, фактах, событиях, явлениях и процессах независимо от формы их представления
Качество программы	***Software quality***	Совокупность черт и характеристик программной системы, которые влияют на ее способность удовлетворять заданные потребности пользователей
Класс	***Class***	Множество объектов, обладающих внутренними (имманентными) свойствами, которые играют роль *классообразующих признаков* и присущи любому объекту класса
Методология программирования	***Methodology of programming***	Совокупность механизмов, применяемых в процессе разработки программного обеспечения и объединенных одним общим философским подходом
Модель	***Model***	Абстракция, которая служит для упрощения или представления какой-либо концепции или процесса
Модель жизненного цикла	***Life cycle model***	Описывается набор фаз (этапов, стадий) проекта по созданию продукта, в которых выполняются отдельные процессы, разбитые на операции и задачи
Модуль	***Module***	Единица кода, служащая строительным блоком физической структуры системы; программный блок, который содержит объявления, выраженные в

		соответствии с требованиями языка и образующие физическую реализацию части или всех классов и объектов логического проекта системы (состоит из интерфейсной части и реализации)
Модульное программирование	***Modular programming***	Организация программы в виде совокупности модулей со строгим соблюдением правил их взаимодействия
Надежность	***Reliability***	Способность системы программного обеспечения выполнять возложенные на нее функции при поступлении требований на их выполнение
Наследование	***Inheritance***	Возможность вывода нового класса из старого с частичным изменением свойств и методов
		Отношение между классами, при котором класс использует структуру или поведение другого (одиночное наследование) или других (множественное наследование) классов. Наследование вводит иерархию «общее/частное», в которой подкласс наследует от одного или нескольких более общих суперклассов. Подклассы обычно дополняют или переопределяют унаследованную структуру и поведение
Обработка данных	***Data processing***	Выполнение систематической последовательности действий с данными, которые представляются и хранятся на так называемых носителях данных
Объектная модель	***Object model***	Совокупность основополагающих принципов, лежащих в основе объектно-ориентированного проектирования; парадигма программирования, основанная на принципах абстрагирования, инкапсуляции, модульности, иерархичности, типизации, параллелизма и устойчивости
Объектное программирование	***Object-based programming***	Метод программирования, основанный на представлении программы как совокупности объектов, каждый из которых является экземпляром некоторого типа.

		Типы образуют иерархию, но не наследственную. В программах типы рассматриваются как статические, а объекты имеют более динамическую природу, которую ограничивают статическое связывание и мономорфизм
Объектно-ориентированная декомпозиция	***Object-oriented decomposition***	Процесс разбиения системы на части, соответствующие классам и объектам предметной области. Мир рассматривается как совокупность объектов, согласованно действующих для обеспечения требуемого поведения
Объектно-ориентированное программирование	***Object-oriented programming***	Методология реализации, при которой программа организуется, как совокупность объектов, каждый из которых является экземпляром какого-либо класса, а классы образуют иерархию наследования. При этом классы обычно статичны, а объекты очень динамичны, что «поощряется» динамическим связыванием и полиморфизмом
Объектно-ориентированное проектирование	***Object-oriented design***	Методология проектирования, соединяющая процесс объектно-ориентированной декомпозиции и систему обозначений для представления логической и физической, статической и динамической моделей проектируемой системы. Система обозначений состоит из диаграмм классов, объектов, модулей и процессов
Парадигма программирования	***Paradigm***	Новая модель конструирования программы и взаимодействия ее с данными
Полиморфизм	***Polymorphism***	Определение свойств и методов объекта по контексту (подразумевает отделение идеи «что делать» от ее воплощения внутри иерархии класса объектов «как делать»)

		Положение теории типов, согласно которому имена (например, переменных) могут обозначать объекты разных (но имеющих общего родителя) классов. Следовательно, любой объект, обозначаемый полиморфным именем, может по-своему реагировать на некий общий набор операций
Предметная область	***Application domain***	Часть реального мира, которая имеет существенное значение или непосредственное отношение к процессу функционирования программы, она включает в себя только те объекты и взаимосвязи между ними, которые необходимы для описания требований и условий решения некоторой задачи
Программа	***Program***	Набор операторов, который может быть представлен как единое целое в некоторой вычислительной системе и который используется для управления поведением этой системы.
Программирование (в широком смысле)	***Programming***	Все технические операции, необходимые для создания программы, включая анализ требований и все стадии разработки и реализации
Программирование (в узком смысле)		Процесс кодирования и отладки программы в рамках реального проекта
Программная инженерия	***Software engineering***	Систематический подход к разработке, эксплуатации, сопровождению и изъятию из обращения программных средств
Процесс	***Process***	Набор взаимосвязанных работ, которые преобразуют исходные данные в выходные результаты
Процесс разработки	***development process***	В соответствии со стандартом предусматривает действия и задачи, выполняемые разработчиком, и охватывает работы по созданию ПО и его компонентов в соответствии с созданными

		требованиями, включая оформление проектной и эксплуатационной документации, а также подготовку материалов, необходимых для проверки работоспособности и соответствия качества программных продуктов, материалов, необходимых для обучения персонала, и т.д.
Сопровождаемость	***Maintainability***	Характеристики ПС, которые позволяют минимизировать усилия по внесению изменений в систему при устранении в ней ошибок и/или при ее модификации из-за изменяющихся потребностей пользователей
Среда разработки	***Framework***	Набор классов, предоставляющих некоторые базовые услуги в определенной области. Таким образом, среда разработки экспортирует классы и механизмы, которые клиенты могут использовать или адаптировать в своих целях
Стандарт	***Standard***	Нормативно-технический документ, утверждаемый компетентным органом, устанавливающий комплекс норм, правил по отношению к предмету стандартизации; типовой образец, эталон, модель, которые принимаются как исходные для сопоставления с ними других предметов
Структурное программирование	***Structured programming***	Методология программирования, основанная на предположении, что логичность и понятность программ обеспечивает ее надежность, облегчает модификацию и ускоряет обработку
Структурное проектирование	***Structured design***	Метод проектирования, основанный на алгоритмической декомпозиции
Технология	***Technology***	Совокупность производственных процессов в определенной отрасли производства, а также научное описание способов производства, это совокуп-

		ность технологических элементов (средств, устройств, методов, приемов, документов), используемых для обработки исходных материалов с целью получения конечной продукции
Технология программирования	***Programming technology***	Совокупность методов и средств, используемых в процессе разработки программных продуктов, представляет собой набор технологических инструкций
Унифицированный язык моделирования	***Unified modeling language***	Графический язык для визуализации, специфицирования, конструирования и документирования программных систем
Уровень абстракции	***Level of abstraction***	Относительное упорядочение абстракций по структурам классов, объектов, модулей или процессов
Эффективность	***Efficiency***	Отношение уровня услуг, предоставляемых ПС пользователю при заданных условиях, к объему используемых ресурсов (различают эффективность по времени, по памяти, по оборудованию)

Учебное издание

Зеленко Лариса Сергеевна

**ТЕХНОЛОГИИ ПРОГРАММИРОВАНИЯ
И ПРОГРАММНАЯ ИНЖЕНЕРИЯ
Часть 1**

Учебное пособие

Технический редактор *Э.И. Коломиец*
Редакторская обработка *Н.С. Куприянова*
Корректорская обработка *Л.Я. Чегодаева*
Доверстка *А.А. Нечитайло*

Подписано в печать 12.12.06 . Формат 60x84 1/16.
Бумага офсетная. Печать офсетная.
Усл. печ. л. 5,6. Усл. кр.-отт. 5,7. Печ. л. 6,0.
Тираж 50 экз. Заказ . ИП-67/2006

Самарский государственный
аэрокосмический университет.
443086 Самара, Московское шоссе, 34.

Изд-во Самарского государственного
аэрокосмического университета.
443086 Самара, Московское шоссе, 34.