Оглавление

[**Лекция 6 Методологии разработки ПО и модели жизненного цикла (часть1)** 2](#_Toc464457963)

[**6.1 Методологии разработки ПО** 2](#_Toc464457964)

[***6.1.1 Новые методологии разработки ПО*** 2](#_Toc464457965)

[***6.1.2 Классические методологии разработки*** 5](#_Toc464457966)

[**6.2 Модели жизненного цикла программного обеспечения** 7](#_Toc464457967)

[***6.2.1 Введение*** 7](#_Toc464457968)

[***6.2.2 Модели традиционного представления о жизненном цикле*** 8](#_Toc464457969)

[***6.2.2.2 Классическая итерационная модель*** 12](#_Toc464457970)

[***6.2.2.3 Каскадная модель*** 14](#_Toc464457971)

[**6.3 Связь модели ЖЦ ПО с управлением проектом разработки ПО: модель фазы-функции** 18](#_Toc464457972)

## **Лекция 6 Методологии разработки ПО и модели жизненного цикла (часть1)**

### **6.1 Методологии разработки ПО**

**Методология** — это система принципов, а также совокупность идей, понятий, методов, способов и средств, определяющих стиль разработки программного обеспечения.

Методология — это реализация стандарта**. Сами стандарты лишь говорят о том, что должно быть, оставляя свободу выбора и адаптации**.  
Конкретные вещи реализуется через выбранную методологию. Именно она определяет, как будет выполняться разработка. Существует много успешных методологий создания программного обеспечения. Выбор конкретной методологии зависит от размера команды, от специфики и сложности проекта, от стабильности и зрелости процессов в компании и от личных качеств сотрудников.

Методологии представляют собой ядро теории управления разработкой программного обеспечения. К существующей классификации в зависимости от используемой в ней **модели жизненного цикла (водопадные и итерационные методологии)** добавилась более общая классификация на **прогнозируемы и адаптивные** методологии.

 Выбору методологии разработки программного обеспечения уделяется повышенное внимание: как показывает опыт, без правильной методологии даже небольшие проекты вряд ли могут быть успешными. В настоящее время все больше разработчиков, аналитиков и руководителей российских проектов начинают это осознавать.

### ***6.1.1 Новые методологии разработки ПО***

**Прогнозируемые методологии** фокусируются на детальном планировании будущего. Известны запланированные задачи и ресурсы на весь срок проекта. Команда с трудом реагирует на возможные изменения. План оптимизирован исходя из состава работ и существующих требований. Изменение требований может привести к существенному изменению плана, а также дизайна проекта. Часто создается специальный комитет по «управлению изменениями», чтобы в проекте учитывались только самые важные требования.  
  
 **Адаптивные методологии** нацелены на преодоление ожидаемой неполноты требований и их постоянного изменения. Когда меняются требования, команда разработчиков тоже меняется. Команда, участвующая в адаптивной разработке, с трудом может предсказать будущее проекта. Существует точный план лишь на ближайшее время. Более удаленные во времени планы существуют лишь как декларации о целях проекта, ожидаемых затратах и результатах.  
  
 **SCRUM** — методология, предназначенная для небольших команд (до 10 человек). Весь проект делится на итерации (спринты) продолжительностью 30 дней каждый. Выбирается список функций системы, которые планируется реализовать в течение следующего спринта. Самые важные условия — неизменность выбранных функций во время выполнения одной итерации и строгое соблюдение сроков выпуска очередного релиза, даже если к его выпуску не удастся реализовать весь запланированный функционал. Руководитель разработки проводит ежедневные 20 минутные совещания, которые так и называют — scrum, результатом которых является определение функции системы, реализованных за предыдущий день, возникшие сложности и план на следующий день. Такие совещания позволяют постоянно отслеживать ход проекта, быстро выявлять возникшие проблемы и оперативно на них реагировать.  
  
 **KANBAN** – гибкая методология разработки программного обеспечения, ориентированная на задачи.   
 Основные правила:

* + визуализация разработки:
    - разделение работы на задачи;
    - использование отметок о положение задачи в разработке;
* ограничение работ, выполняющихся одновременно, на каждом этапе разработки;
* измерение времени цикла (среднее время на выполнение одной задачи) и оптимизация процесса.

Преимущества KANBAN:

* уменьшение числа параллельно выполняемых задач значительно уменьшает время выполнения каждой отдельной задачи;
* быстрое выявление проблемных задач;
* вычисление времени на выполнение усредненной задачи.

**DYNAMIC SYSTEM DEVELOPMENT METHOD** появился в результате работы консорциум из 17 английских компаний. Целая организация занимается разработкой пособий по этой методологии, организацией учебных курсов, программ аккредитации и т.п. Кроме того, ценность DSDM имеет денежный эквивалент.   
 Все начинается с изучения осуществимости программы и области ее применения. В первом случае, вы пытаетесь понять, подходит ли DSDM для данного проекта. Изучать область применения программы предполагается на короткой серии семинаров, где программисты узнают о той сфере бизнеса, для которой им предстоит работать. Здесь же обсуждаются основные положения, касающиеся архитектуры будущей системы и план проекта.   
 **Далее процесс делится на три взаимосвязанных цикла**: цикл функциональной модели отвечает за создание аналитической документации и прототипов, цикл проектирования и конструирования — за приведение системы в рабочее состояние, и наконец, последний цикл — цикл реализации — обеспечивает развертывание программной системы.   
 Базовые принципы, на которых строится DSDM, это активное взаимодействие с пользователями, частые выпуски версий, самостоятельность разработчиков в принятии решений и тестирование в течение всего цикла работ. Как и большинство других гибких методологий, DSDM использует короткие итерации, продолжительностью от двух до шести недель каждая. **Особый упор делается на высоком качестве работы и адаптируемости к изменениям в требованиях.**  
  
 **MICROSOFT SOLUTIONS FRAMEWORK[[1]](#footnote-1)** — методология разработки программного обеспечения, предложенная корпорацией Microsoft. MSF опирается на практический опыт Microsoft и описывает управление людьми и рабочими процессами в процессе разработки решения.  
Базовые концепции и принципы модели процессов MSF:

* + единое видение проекта — все заинтересованные лица и просто участники проекта должны чётко представлять конечный результат, всем должна быть понятна цель проекта;
  + управление компромиссами — поиск компромиссов между ресурсами проекта, календарным графиком и реализуемыми возможностями;
  + гибкость – готовность к изменяющимся проектным условиям;
  + концентрация на бизнес-приоритетах — сосредоточенность на той отдаче и выгоде, которую ожидает получить потребитель решения;
  + поощрение свободного общения внутри проекта;
  + создание базовых версии — фиксация состояния любого проектного артефакта, в том числе программного кода, плана проекта, руководства пользователя, настройки серверов и последующее эффективное управление изменениями, аналитика проекта.

MSF предлагает проверенные методики для планирования, проектирования, разработки и внедрения успешных IT-решений. Благодаря своей гибкости, масштабируемости и отсутствию жестких инструкций MSF способен удовлетворить нужды организации или проектной группы любого размера. Методология MSF состоит из принципов, моделей и дисциплин по управлению персоналом, процессами, технологическими элементами и связанными со всеми этими факторами вопросами, характерными для большинства проектов.  
  
 **RATIONAL UNIFIED PROCESS** — методология разработки программного обеспечения, созданная компанией Rational Software.   
В основе методологии лежат 6 основных принципов:  
компонентная архитектура, реализуемая и тестируемая на ранних стадиях проекта;

* работа над проектом в сплочённой команде, ключевая роль в которой принадлежит архитекторам;
* ранняя идентификация и непрерывное устранение возможных рисков;
* концентрация на выполнении требований заказчиков к исполняемой программе;
* ожидание изменений в требованиях, проектных решениях и реализации в процессе разработки;
* постоянное обеспечение качества на всех этапах разработки проекта.

**Использование методологии RUP направлено на итеративную модель** разработки. Особенность методологии состоит в том, что степень формализации может меняться в зависимости от **потребностей** проекта. Можно по окончании каждого этапа и каждой итерации создавать все требуемые документы и достигнуть максимального уровня формализации, а можно создавать только необходимые для работы документы, вплоть до полного их отсутствия. За счет такого подхода к формализации процессов методология является достаточно гибкой и широко популярной. Данная методология применима как в небольших и быстрых проектах, где за счет отсутствия формализации требуется сократить время выполнения проекта и расходы, так и в больших и сложных проектах, где требуется высокий уровень формализма, например, с целью дальнейшей сертификации продукта. Это преимущество дает возможность использовать одну и ту же команду разработчиков для реализации различных по объему и требованиям.  
  
 Таким образом, существует множество различных методологий разработки программного обеспечения, они не универсальны и описываются различными принципами. Выбор методологии разработки для конкретного проекта зависит от предъявляемых требований.

### ***6.1.2 Классические методологии разработки***

1. Все **продукты** процессов программной инженерии представляют собой некоторые описания, а именно:
   1. Тексты требований к разработке;
   2. Согласования договорённостей с заказчиком;
   3. Описания архитектуры и структуры данных;
   4. Тексты программ;
   5. Документацию;
   6. Инструкции и т.п.
2. Главными ресурсами разработки ПС являются **сроки, время и стоимост**ь, которые необходимо правильно использовать на процессах ЖЦ;
3. ЖЦ ПО следует представлять в виде четырёх обобщённых фаз:
   1. Концепция (инициация, идентификация, отбор)[[2]](#footnote-2);
   2. Анализ(определение);
   3. Выполнение (практическая реализация или внедрение, производство и развёртывание, проектирование или конструирование, сдача в эксплуатацию);
   4. Закрытие (завершение, включая оценивание).
4. Так как эти фазы определены очень широко, как правило, для каждой категории и подкатегории проекта внутри каждой фазы выделяют несколько подфаз. **В общем случае фазы и подфазы не обязательно должны выполняться линейно и последовательно**;
5. Для жизненного цикла можно выделить (и применять) понятия модели ЖЦ и методологии (метода):
   1. **Модель ЖЦ** – **это концептуальный взгляд на его организацию, что подразумевает описание фаз и принципы перехода между ними;**
   2. **Методология ЖЦ задаёт (описывает)**:
      1. Комплекс работ по фазам,
      2. Детальное содержание этих работ,
      3. Ролевую ответственность специалистов на всех этапах ЖЦ,
      4. **Лучшие практики** в рамках модели и методологии, позволяющие максимально эффективно воспользоваться ими.

### **6.2 Модели жизненного цикла программного обеспечения**

### ***6.2.1 Введение***

Рассматривается моделирование жизненного цикла программного обеспечения как **основа технологичной разработки программ**. Представлены разные подходы к моделированию жизненного цикла, отражающие различные представления о назначении такого моделирования. Описываются особенности объектно-ориентированного моделирования жизненного цикла, в том числе и учет непрерывно поступающих требований к разрабатываемому проекту.

Понятие жизненного цикла программного обеспечения появилось, когда программистское сообщество осознало необходимость перехода от кустарных ремесленнических методов разработки программ к технологичному промышленному их производству. Как обычно происходит в подобных ситуациях, программисты попытались перенести опыт других индустриальных производств в свою сферу. В частности, было заимствовано понятие жизненного цикла.

Аналогия жизненного цикла программного обеспечения с техническими системами имеет более глубокие корни, чем это может показаться на первый взгляд. Программы не подвержены физическому износу, но в ходе их эксплуатации обнаруживаются ошибки (неисправности), требующие исправления. Ошибки возникают также от изменения условий использования программы. Последнее же является принципиальным свойством программного обеспечения, иначе оно теряет свой смысл. Поэтому правомерно говорить *о старении программ*, хотя не о физическом старении, а о моральном.

Необходимость внесения изменений в действующие программы как из-за обнаруживаемых ошибок, так и по причине развития требований приводит по сути дела к тому, что разработка программного обеспечения продолжается после передачи его пользователю и в течение всего времени жизни программ. Деятельность, связанная с решением довольно многочисленных задач такой продолжающейся разработки, получила название *сопровождения программного обеспечения.*

Исторически развитие концепций жизненного цикла связано с поиском для него **адекватных моделей**. Как и всякая другая, модель жизненного цикла является **абстракцией реального процесса**, в которой **опущены детали, несущественные** с точки зрения назначения модели.

**Различие назначений применения моделей определяет их разнообразие.**

Основные причины, из-за которых нужно изучать вопросы моделирования жизненного цикла программного обеспечения, можно сформулировать следующим образом.

**Во-первых**, это знание даже для непрофессионального программиста помогает понять, на что можно рассчитывать при заказе или приобретении программного обеспечения и что нереально требовать от него. В частности, неудобные моменты работы с программой, ее ошибки и недоработки обычно устраняются в ходе продолжающейся разработки, и есть основания ожидать, что последующие версии будут лучше. Однако кардинальные изменения концепций программы — задача другого проекта, который совсем необязательно будет во всех отношениях лучше данной системы.

**Во-вторых**, модели жизненного цикла — основа знания технологий программирования и инструментария, поддерживающего их. **Программист всегда применяет в своей работе инструменты**, но квалифицированный программист знает, где, когда и как их применять. Именно в этом помогают понятия моделирования жизненного цикла: любая технология базируется на определенных представлениях о жизненном цикле, выстраивает свои методы и инструменты вокруг фаз и этапов жизненного цикла.

**В-третьих**, общие знания того, как развивается программный проект, дают наиболее надежные ориентиры для его планирования, позволяют экономнее расходовать ресурсы, добиваться более высокого качества управления. **Все это относится к сфере профессиональных обязанностей руководителя программного проекта**.

Базовые особенности ЖЦ ПО следует (удобно) изучать, **абстрагируясь от специфики разработки конкретных программных систем**.

Интересно проследить, как развитие традиционных моделей адаптировалось к потребностям объектно-ориентированного проектирования.

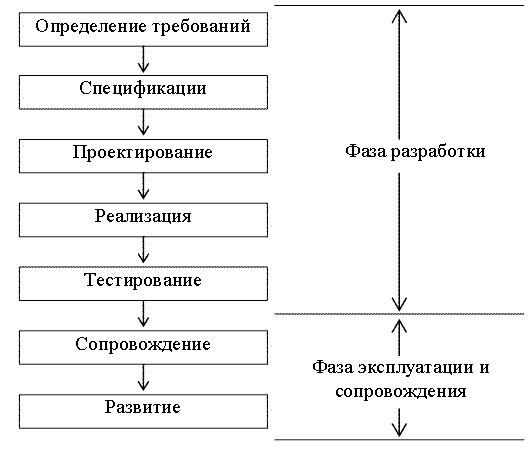
### ***6.2.2 Модели традиционного представления о жизненном цикле***

***6.2.2.1 Общепринятая модель***

Вероятно, самым распространенным мотивом обращения к понятию жизненного цикла является потребность в систематизации работ в соответствии с технологическим процессом. Этому назначению хорошо соответствует так называемая *общепринятая модель*жизненного цикла программного обеспечения, согласно которой программные системы проходят в своем развитии две *фазы:*

* разработка,
* сопровождение.

Фазы разбиваются на ряд *этапов*(рис. 6.1).



**Рис. 6.1. Общепринятая модель жизненного цикла программного обеспечения**

Разработка начинается с *идентификации потребности*в новом приложении, а заканчивается передачей продукта разработки в эксплуатацию.

Первым этапом фазы разработки является *постановка задачи и определение требований*. Определение требований включает:

* **описание общего контекста задачи**,
* ожидаемых функций системы и
* ее ограничений.

На этом этапе заказчик совместно с разработчиками принимают решение о создании системы**. Особенно существенен этот этап для нетрадиционных приложений.**

В случае положительного решения начинается этап ***спецификации системы в соответствии с требованиями****.* Разработчики программного обеспечения пытаются осмыслить выдвигаемые заказчиком требования и зафиксировать их в виде **спецификаций системы**.

Важно подчеркнуть, что назначение **этих спецификаций** — описывать внешнее поведение разрабатываемой системы, а не ее внутреннюю организацию, т.е. отвечать на вопрос**, *что***она должна делать, а не ***как***это будет реализовано.

Здесь говорится о назначении, а не о форме спецификаций, поскольку на практике **при отсутствии подходящего языка спецификаций**, к сожалению, нередко приходится прибегать к описанию «*что*» посредством «*как*».

**Проблемы языка спецификаций не в том, что нельзя (или трудно) строго и четко описать, что требуется в проекте. В большей степени они связаны с необходимостью добиваться и поддерживать соответствие описания «что» нечетким, неточным и часто противоречивым требованиям со стороны внешних по отношению к проекту людей. Нет оснований полагать, что эти люди будут знакомы с «самым хорошим языком спецификаций», что они будут заботиться о корректности своих требований. Задача этапа спецификаций в том и состоит, чтобы описание программы выстроить в виде логически выверенной системы, понятной как для заказчика данной разработки, будущих пользователей, так и для исполнителей проекта.**

Ссылки на языки спецификаций:

* <https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D1%81%D0%BF%D0%B5%D1%86%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D0%B9>
* <http://www.studfiles.ru/preview/5316946/page:20/>
* <http://allrefs.net/c20/45e6h/p26/>
* <http://www.intuit.ru/studies/courses/945/237/lecture/6128>
* <http://sp.cmc.msu.ru/courses/fmsp/msc&sdl.pdf>
* <http://pandia.ru/text/78/597/34816.php>
* <http://www.intuit.ru/studies/professional_retraining/945/courses/237/lecture/6116>
* <http://www.intuit.ru/studies/courses/1041/218/lecture/27257?page=2>
* <http://5fan.ru/wievjob.php?id=14723>

Прежде чем приступать к созданию проекта по спецификациям, они должны быть тщательно проверены на соответствие исходным целям, полноту, совместимость (непротиворечивость) и однозначность.

Проблемы языка спецификаций не в том, что нельзя (или трудно) строго и четко описать, что требуется в проекте. В большей степени они связаны с необходимостью добиваться и поддерживать соответствие описания «*что*» нечетким, неточным и часто противоречивым требованиям со стороны внешних по отношению к проекту людей. Нет оснований полагать, что эти люди будут знакомы с «самым хорошим языком спецификаций», что они будут заботиться о корректности своих требований. Задача этапа спецификаций в том и состоит, чтобы описание программы выстроить в виде логически выверенной системы, понятной как для заказчика данной разработки, будущих пользователей, так и для исполнителей проекта.

Разработка проектных решений, отвечающих на вопрос, *как*должна быть реализована система, чтобы она могла удовлетворять специфицированным требованиям, выполняется на этапе *проектирования.* **Поскольку сложность системы в целом может быть очень большой, главной задачей этого этапа является последовательная декомпозиция системы до уровня *очевидно реализуемых*модулей или процедур.**

На следующем этапе *реализации*, или *кодирования*каждый из этих модулей программируется на наиболее подходящем для данного приложения языке. С точки зрения автоматизации этот этап традиционно является наиболее развитым.

В рассматриваемой модели фаза разработки заканчивается этапом *тестирования*(автономного и комплексного) и *передачей*системы в *эксплуатацию.*

Фаза эксплуатации и сопровождения включает в себя всю деятельность по обеспечению нормального функционирования программных систем, в том числе фиксирование вскрытых во время исполнения программ ошибок, поиск их причин и исправление, повышение эксплуатационных характеристик системы, адаптацию системы к окружающей среде, а также, при необходимости, и более существенные работы по совершенствованию системы. Все это дает право говорить об *эволюции системы.* В связи с этим, фаза эксплуатации и сопровождения разбивается на два этапа: собственно, *сопровождение*и *развитие.* В ряде случаев на данную фазу приходится большая часть средств, расходуемых в процессе жизненного цикла программного обеспечения.

Понятно, что внимание программистов к тем или иным этапам разработки зависит от конкретного проекта. **Часто разработчику нет необходимости проходить через все этапы, например, если создается небольшая хорошо понятная программа с ясно поставленной целью**. *Проблемы сопровождения, плохо понимаемые разработчиками небольших программ для личного пользования, являются в то же время очень важными для больших систем.*

Такова краткая характеристика общепринятой модели. В литературе встречается много вариантов, развивающих ее в сторону детализации и добавления промежуточных фаз, этапов, стадий и отдельных работ (например, по документированию и технологической подготовке проектов) в зависимости от особенностей программных проектов или предпочтений разработчиков.

### ***6.2.2.2 Классическая итерационная модель***

Общепринятая модель жизненного цикла является идеальной, так как только очень простые задачи проходят все этапы без каких-либо ***итераций* — возвратов на предыдущие шаги технологического процесса**.

При программировании, например, может обнаружиться, что реализация некоторой функции очень громоздка, неэффективна и вступает в противоречие с требуемой от системы производительностью. В этом случае требуется перепроектирование, а может быть, и переделка спецификаций. При разработке больших нетрадиционных систем необходимость в итерациях возникает регулярно на любом этапе жизненного цикла как из-за допущенных на предыдущих шагах ошибок и неточностей, так и из-за изменений внешних требований к условиям эксплуатации системы.

Таковы мотивы *классической итерационной модели*жизненного цикла (рис. 6.2).

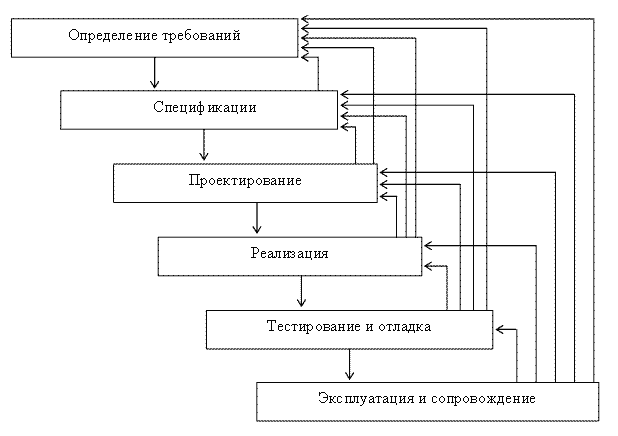


Рис. 6.2. Классическая итерационная модель

Стрелки, ведущие вверх, обозначают возвраты к предыдущим этапам, квалифицируемые как требование повторить этап для исправления обнаруженной ошибки. В этой связи может показаться странным переход от этапа «Эксплуатация и сопровождение» к этапу «Тестирование и отладка». Дело в том, что рекламации, предъявляемые в ходе эксплуатации системы, часто даются в такой форме, которая нуждается в их перепроверке. Чтобы понять, о каких ошибках идет речь в рекламации, разработчикам полезно предварительно воспроизвести пользовательскую ситуацию у себя, т.е. выполнить действия, которые обычно относят к тестированию.

Классическая итерационная модель **абсолютизирует** возможность возвратов на предыдущие этапы. **Однако это обстоятельство отражает существенный непреодолимый аспект программных разработок, не опирающихся на объектно-ориенти­ро­ван­ное проектирование: стремление заранее предвидеть все ситуации использования системы и невозможность в подавляющем большинстве случаев достичь этого.**

Все традиционные технологии программирования направлены лишь на то, чтобы минимизировать возвраты. Но суть от этого не меняется: **при возврате всегда приходится повторять построение того, что уже считалось готовым.**

Иное положение с объектно-ориентированными технологиями. Для них можно отказаться от **завершенности фаз и этапов**. Вместо этого предлагается распределять наращивание функциональности и интерфейсных возможностей **по итерациям**, **позволяет ослабить требование переделки старого при возвратах**.

По существу, классическая схема остается верной, но только в рамках одной итерации и с одной важной поправкой: **все полезное, что было сделано ранее, сохраняется**.

Понятно, **что для программной системы в целом новый подход требует и новых моделей жизненного цикла, отражающих его особенности, отмеченные ранее**. Об этом будет идти речь после изучения основных вариантов традиционных моделей жизненного цикла.

### ***6.2.2.3 Каскадная модель***

Некоторой более строгой разновидностью классической модели является так называемая *каскадная модель*, которую можно рассматривать **в качестве показательного примера того**, **какими методами можно минимизировать возвраты.**

Характерные черты каскадной модели:

* **завершение** каждого этапа (они почти те же, что и в классической модели) **проверкой полученных результатов** с целью устранить как можно большее число проблем, связанных с разработкой изделия;
* циклическое повторение пройденных этапов (как в классической модели).

Мотивация каскадной модели связана с так называемым ***управлением качеством***программного обеспечения. В связи с ней уточняются понятия этапов, некоторые из них структурируются (спецификация требований и реализация).

На рис. 6.3 приведена схема каскадной модели, построенная как модификация классической итерационной модели. В каждом блоке, обозначающем этап, указано действие, которым этап завершается (наименования этих действий отмечены серым фоном). Из рисунка видно, что в этой модели тестирование не выделяется в качестве отдельного этапа, а считается лишь порогом, через который нужно перейти, чтобы завершить этап, точно так же, как и другие подобные действия.

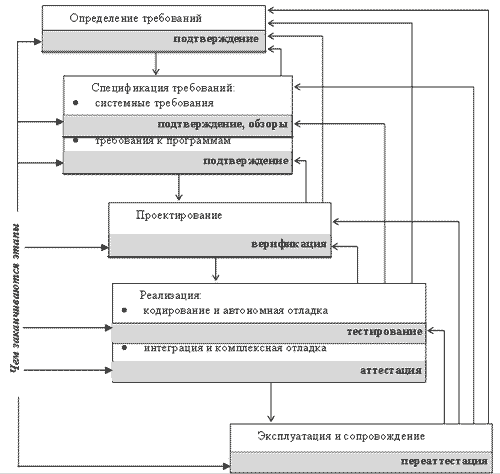


Рис.6.3 Каскадная модель ЖЦ ПО

В соответствии с каскадной моделью завершение этапа определения системных требований включает фиксацию их в виде специальных документов, называемых *обзорами*того, что от системы требуется (описание функций), а спецификация требований к программам —*подтверждением*выполнения зафиксированных в обзорах функций в планируемых к реализации программах. Кроме того, подтверждение предполагается и на первом этапе, т.е. после определения требований. Это отражает тот факт, что полученные требования необходимо согласовывать с заказчиком.

Результат проектирования *верифицируется,* т.е. проверяется, что принятая структура системы и реализационные механизмы обеспечивают выполнимость специфицированных функций.

Реализация контролируется путем *тестирования*компонент, а после интеграции компонент в систему и комплексной отладки проводится *аттестация,* т.е. проверка-фиксация фактически реализованных функций системы, описание ограничений реализации и т.п.

В ходе эксплуатации и сопровождения изделия устанавливается, насколько хорошо система соответствует пользовательским запросам, т.е. осуществляется *переаттестация.*

Каждая из указанных проверок может отослать разработчиков системы к повторению любого из ранее пройденных этапов, что иллюстрируется стрелками на рис. 6.3. В то же время, каскадная модель разработана в ответ на требование практики разработки программных проектов, в которых за счет преодоления проверочных барьеров достигается минимизация возвратов к пройденным этапам. Такая минимизация возможна не только в плане количества откатов по схеме: за счет ужесточения проверок разработчики пытаются ликвидировать прямые возвраты через несколько этапов. Соответствующая схема, называемая *строгой каскадной моделью,* представлена на рис. 6.4.

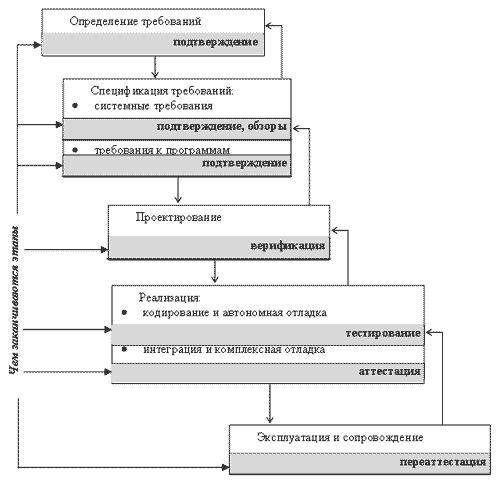


Рис. 6.4. Строгая каскадная модель

Поучительно проследить, как в строгой каскадной модели исправляются ошибки ранних этапов**. В соответствии с данной схемой разработчики любого этапа в качестве исходных материалов для своей деятельности, т.е. *задания на разработку,* получают результаты предыдущего этапа, прошедшие соответствующую проверку (в идеале исполнители этапа могут вовсе не знать о более ранних этапах**). При проведении работ этапа может быть выяснено, что задание невыполнимо по одной из следующих причин:

* оно противоречиво, т.е. содержит несовместные или невыполнимые требования;
* не выработаны критерии для выбора одного из возможных вариантов решения.

Обе ситуации квалифицируются как *ошибки задания,* т.е. как ошибки предыдущего этапа. Для исправления обнаруженных ошибок работы предыдущего этапа возобновляются. В результате ошибки либо ликвидируются, либо констатируется невозможность их непосредственного исправления. В первом случае работы этапа, вызвавшего возврат, возобновляются с откорректированным заданием. Второй случай квалифицируется как ошибка более раннего этапа.

Строгая каскадная модель фиксирует **два важных момента** жизненного цикла:

* точное разделение работ, заданий и ответственности разработчиков этапов и тех, кто, проверяя работы, инициирует переход к следующему этапу;
* малые циклы между соседними этапами, в результате которых до­стигается компромиссное задание.

**Первый момент** — это шаг к осознанию фактического разделения труда, из которого вполне осуществимо явное выделение технологических и организационных функций, выполняемых на каждом этапе.

В результате появляется возможность постановки **задачи автоматизированной поддержки этих функций**.

**Второй момент** можно трактовать как совместное выполнение работ соседних этапов, т.е. их перекрытие. Однако в рамках каскадной модели эти обстоятельства отражаются лишь косвенно. **Продуктивность явного включения их в качестве элементов модели жизненного цикла демонстрируется в следующем разделе.**

### **6.3 Связь модели ЖЦ ПО с управлением проектом разработки ПО: модель фазы-функции**

Чрезвычайно важным мотивом развития моделей жизненного цикла программного обеспечения является **потребность в подходящем средстве для комплексного управления проектом.**

По существу, это утверждение указывает на то, что **модель должна служить основой организации взаимоотношений между разработчиками, и, таким образом, одной из ее целей является поддержка функций менеджера.**

**Это приводит к необходимости наложения на модель контрольных точек и функций, задающих организационно-временные рамки проекта.**

Наиболее последовательно такое дополнение классической схемы реализовано в модели Гантера в виде матрицы «фазы—функции». Уже из упоминания о матрице следует, что модель Гантера имеет два измерения:

* ***фазовое,*** отражающее этапы выполнения проекта и сопутствующие им события;
* ***функциональное****,* показывающее, какие организационные функции выполняются в ходе развития проекта и **какова их интенсивность на каждом из этапов**.

В модели Гантера отражено то, что выполнение функции на одном этапе может продолжаться и на следующем. На рис. 6.5 представлено фазовое измерение модели. Жирной чертой (с разрывом и стрелкой, обозначающей временное направление) изображен процесс разработки. Контрольные точки и наименования событий указаны под этой чертой. Они пронумерованы. Все развитие проекта в модели привязывается к этим контрольным точкам и событиям.

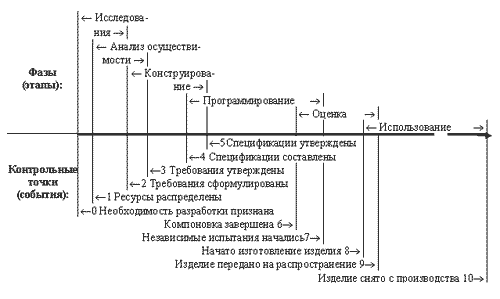
****

Рис. 6.5. Фазовое измерение модели фазы—функции

В данной модели жизненный цикл распадается на следующие перекрывающие друг друга фазы (этапы):

* *исследования*— этап начинается, когда необходимость разработки признана руководством проекта (контрольная точка 0), и заключается в том, что для проекта обосновываются требуемые ресурсы (контрольная точка 1) и формулируются требования к разрабатываемому изделию (контрольная точка 2);
* *анализ осуществимости*— начинается на фазе исследования, когда определены исполнители проекта (контрольная точка 1), и завершается утверждением требований (контрольная точка 3). Цель этапа — определить возможность конструирования изделия с технической точки зрения (достаточно ли ресурсов, квалификации и т.п.), будет ли изделие удобно для практического использования, ответить на вопросы экономической и коммерческой эффективности;
* *конструирование*— этап начинается обычно на фазе анализа осуществимости, как только документально зафиксированы предварительные цели проекта (контрольная точка 2), и заканчивается утверждением проектных решений в виде официальной спецификации на разработку (контрольная точка 5);
* *программирование*— начинается на фазе конструирования, когда становятся доступными основные спецификации на отдельные компоненты изделия (контрольная точка 4), но не ранее утверждения соглашения о требованиях (контрольная точка 3). Совмещение данной фазы с заключительным этапом конструирования обеспечивает оперативную проверку проектных решений и некоторых ключевых вопросов разработки. Цель этапа — реализация программ компонентов с последующей сборкой изделия. Он завершается, когда разработчики заканчивают документирование, отладку и компоновку и передают изделие службе, выполняющей независимую оценку результатов работы (независимые испытания начались — контрольная точка 7);
* *оценка*— фаза является буферной зоной между началом испытаний и практическим использованием изделия. Она начинается, как только проведены внутренние (силами разработчиков) испытания изделия (контрольная точка 6) и заканчивается, когда подтверждается готовность изделия к эксплуатации (контрольная точка 9);
* *использование*— начинается в ходе передачи изделия на распространение и продолжается, пока изделие находится в действии и интенсивно эксплуатируется. Этап связан с внедрением, обучением, настройкой и сопровождением, возможно, с модернизацией изделия. Он заканчивается, когда разработчики прекращают систематическую деятельность по сопровождению и поддержке данного программного изделия (контрольная точка 10).

На протяжении фаз жизненного цикла разработчики выполняют следующие технологические (организационные) функции (классы функций):

* планирование,
* разработка,
* обслуживание,
* выпуск документации,
* испытания,
* поддержка,
* сопровождение.

Перечисленные функции на разных этапах имеют различное содержание, требуют различной интенсивности, но, что особенно важно для модели, совмещаются при реализации проекта. Это функциональное измерение модели, наложение которого на фазовое измерение дает изображение матрицы фаз—функций в целом (см. рис. 6.6, на котором интенсивность выполняемых функций отражается густотой закраски клеток матрицы).

Состав организационных функций и их интенсивность могут меняться от проекта к проекту в зависимости от его особенностей, от того, что руководство проекта считает главным или второстепенным. К примеру, если исходная квалификация коллектива не очень высока, в список функций может быть добавлено обучение персонала. Иногда бывает важно разграничить планирование и контроль (по Гантеру контрольные функции явно не выделяются). При объектно-ориентированном проектировании роль моделирования возрастает настолько, что его целесообразно перевести из разряда методов проектирования в явно выделенную технологическую функцию, о чем речь впереди.

Модель учитывает соотношение технологических функций и фаз жизненного цикла, чем она выгодно отличается от простых (или ограниченных?) ранее рассмотренных «идеальных» моделей. По-видимому, простота-ограни­ченность «идеальных» моделей есть следствие отождествления выделяемых этапов с технологической операцией, преобладающей при их выполнении. В то же время, задача отражения итеративности в модели Гантера в явном виде не предусматривается. Хотя само по себе перекрытие смежных фаз проекта и выпуск соответствующей событиям документации — путь к минимизации возвратов к выполненным этапам, более содержательные средства описания итераций в модель не закладываются.

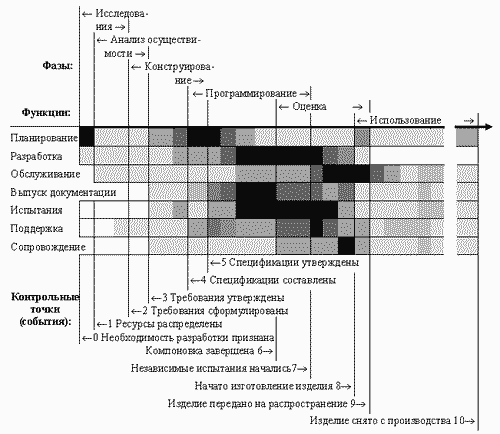


Рис. 6.6. Матрица фазы—функции модели Гантера

Если попытаться развить модель Гантера с целью учета итеративности, то, очевидно, придется предусмотреть *расщепление линии жизненного цикла,* как это представлено на рис. 6.7. Но это влечет и расщепление матрицы интенсивностей выполняемых функций: было бы необоснованно считать, что интенсивности при возвратах сохраняются. В целом, по мере продвижения разработки к своему завершению, они должны уменьшаться. Таким образом, матрица интенсивностей приобретает новое измерение, отражающее итеративный характер развития проекта.

Итеративность неизбежна при разработке сложных программных изделий, а потому ее планирование целесообразно. Однако рассматривая традиционные подходы к развитию проектов, можно заметить, что они не пытаются использовать итеративность в качестве метода проектирования и стремятся лишь к минимизации возвратов.

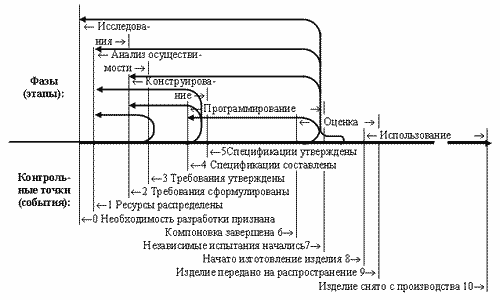


Рис. 6.7. Учет итеративности в модели фазы—функции (фазовое измерение, показаны лишь некоторые возвраты)

1. <http://www.intuit.ru/studies/courses/4806/1054/lecture/16121> [↑](#footnote-ref-1)
2. В скобках приведены используемые в различных источниках альтернативные термины [↑](#footnote-ref-2)