Министерство образования и науки Российской Федерации
КАЗАНСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
им. А.Н. ТУПОЛЕВА

Кафедра прикладной математики и информатики им. Ю.В. Кожевникова

А.И. РАХМАТУЛЛИН

ТЕХНОЛОГИИ РАЗРАБОТКИ ПРОГРАММНЫХ СИСТЕМ

Учебное пособие

Казань 2010

Содержание

Введение

Целью дисциплины является формирование фундаментальных знаний о принципах разработки программного обеспечения (ПО).

Задачами дисциплины является изучение основных понятий и положений методологии и технологии разработки ПО, общих принципов разработки программных систем, приобретение практических навыков использования инструментальных средств для разработки программных продуктов.

Изложение материала дисциплины ведётся аналогично изложению, принятому в пособии [1] с учётом особенностей дисциплины. Для получения дополнительной информации по данной дисциплине рекомендуются пособия [2, 3], а также словарь-справочник [4] и книга [5].

Следует отметить, что в иностранной литературе по данной тематике используется в основном английский язык и принята инженерно-практическая точка зрения рассмотрения возникающих проблем и предлагаемых решений. В связи с этим возможны различные интерпретации проблем и их решений на русском языке: от буквального перевода материала до его адаптации к давно установившейся русской терминологии. Как показало исследование определений основных понятий, ни одна из них не позволяет в полной мере адекватно передать смысл материала.

Поэтому в пособии используется системный подход к разработке ПО и систем и систематизированное изложение соответствующего материала, как это принято в пособии [1], с учётом современного состояния рассматриваемого вопроса, излагаемого в печатных и электронных публикациях на русском и английском языках.

Для большинства понятий приведён английский вариант, позволяющий исключить разночтения в терминологии при изучении другой литературы, в том числе на английском языке.

Раздел 1. Основы разработки ПО

В данном разделе рассматриваются основные понятия и определения, необходимые для ясного понимания всего материала дисциплины.

1.1. Основные понятия и определения

Изначально разработка компьютеров и программ для них была ориентирована на решение научных задач вычислительного характера. Это привело к представлению разработки ПО, приведённому на рис.1.1.

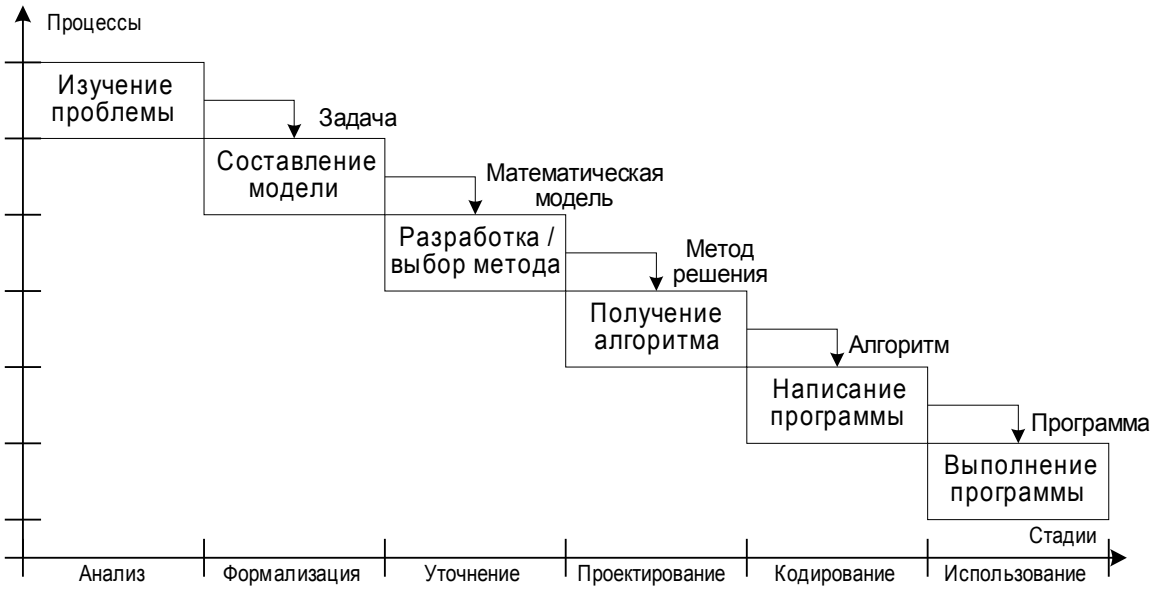


Рис.1.1. Начальное представление о разработке ПО

В настоящее время широко используются понятия системного (в том числе и процессного) подхода, адаптированные к разработке ПО. Приведём общие определения некоторых понятий.

Система и процесс

В системном подходе самым важным является понятие «система».

Система (system) — совокупность взаимодействующих единиц — элементов, функционирующих совместно для достижения определённых целей. *Элемент* (element) — относительно самостоятельная структурная «единица» системы.

Части системы (system parts) представляют собой определённые группы элементов этой системы. В системе можно выделить следующие части. *Компонент*

(component) — часть системы, выполняющая одну или несколько её функций. *Модуль* (module) — отдельный компонент или группа компонентов, обеспечивающая одну или ряд функций для работы других модулей и, в свою очередь, использующая функции, обеспечиваемые иными модулями.

Подсистема (subsystem) — часть системы, функции которой относительно независимы от функций остальных частей системы. С точки зрения структуры *подсистема* — это система, являющаяся частью рассматриваемой системы. Аналогично, *надсистема* (supersystem) — это система, частью которой является рассматриваемая система.

Процессный подход является составной частью системного подхода. В процессном подходе важным является понятие «процесс», который часто рассматривается как некоторая система.

Процесс (process) — совокупность взаимосвязанных единиц — операций, преобразующих некоторые входы в выходы для достижения определённых результатов. *Операция* (operation) — относительно самостоятельная структурная «единица» процесса.

Части процесса (process parts) представляют собой определённые группы операций этого процесса. В процессе также можно выделить следующие части. *Задача* (task) — часть процесса, выполняющая одну или несколько его функций. *Действие* (activity) — отдельная задача или группа задач, обеспечивающая одну или ряд функций для выполнения других действий и, в свою очередь, использующая функции, обеспечиваемые иными действиями.

Подпроцесс (subprocess) — часть процесса, функции которой относительно независимы от функций остальных частей процесса. С точки зрения структуры *подпроцесс* — это процесс, являющийся частью рассматриваемого процесса. Аналогично, *надпроцесс* (superprocess) — это процесс, частью которого является рассматриваемый процесс.

Понятия «система» и «процесс» тесно связаны: система выполняет некоторый процесс, процесс представляет собой функционирование некоторой системы.

Для рассматриваемого объекта система задаёт структурную точку зрения на этот объект, а процесс — функциональную точку зрения. Следует отметить, что приведённые определения системы и процесса в определённой степени соответствуют этим точкам зрения.

В разработке ПО оба понятия «система» и «процесс» играют существенную роль, позволяя рассматривать её с использованием системного подхода, а также грамотно осуществлять управление разработкой ПО.

Выделяют следующие понятия, связанные с ПО и его разработкой.

Программное обеспечение

Основными понятиями программирования являются алгоритм и программа.

Алгоритм (algorithm) — конечный упорядоченный набор чётко определённых правил для решения проблемы (ISO/IEC 2382-1:1993). Решаемая проблема возникает или рассматривается в рамках какой-либо предметной области. *Предметная область* (ПрО, application domain) — совокупность объектов, представляющих часть реального, гипотетического или абстрактного мира, и относящихся к ним понятий, а также связей между ними.

Программа (program) — формализованное описание алгоритма для его выполнения на компьютере. Из-за наличия других смыслов и значений понятия «программа» в настоящее время используют понятие *компьютерная программа* (computer program). Программа может состоять из определённых частей и включаться в состав различных групп программ.

Программный модуль (program module) — явным образом оформленная (разграниченная) часть программы, выполняющая ряд функций и применяемая только в составе какой-либо программы.

Программный компонент (program component) — программа, рассматриваемая как единое целое, выполняющая заданную функцию (или связный набор функций) и применяемая самостоятельно или в совокупности с другими програм-

мами. Фактически компонент — это модуль, который может использоваться самостоятельно или в совокупности с аналогичными компонентами.

Пакет программ (program package) — совокупность программ, решающих ряд задач некоторой ПрО. Пакет программ реализуется как *библиотека программ* (program library) — набор отдельных программ, связанных между собой лишь принадлежностью к определённой области.

Программный комплекс (software suite) — связанная совокупность программ, совместно обеспечивающих решение небольшого класса задач некоторой ПрО. Вызов программ осуществляется с помощью *диспетчера* (manager) — специальной программы, реализующей простой интерфейс с пользователем. Программы комплекса могут решать отдельные этапы конкретной задачи и потому совместно использовать данные и результаты решения разных этапов задачи.

Программная система (ПС, software system) — организованная совокупность программ (подсистем), позволяющая решать широкий класс задач некоторой ПрО. Программы осуществляют взаимодействие через общие данные.

В узком смысле *программное обеспечение* (ПО, software) — совокупность программ (обычно программный комплекс или программная система) на носителях данных. В широком смысле *ПО* — совокупность всех программ и данных (процедур, правил и документации на ПО), входящих в состав компьютера. Здесь *компьютер* (computer, букв. вычислитель) или *вычислительная машина* (computing machine) — это совокупность взаимосвязанного и взаимодействующего аппаратного (hardware), микропрограммного (firmware) и программного (software) обеспечения, используемая для решения поставленных задач. В этом смысле к компьютерам относят и *вычислительные системы* (computer system) — совокупности вычислительных машин.

Программное средство (software, software tool) — это ПО, снабжённое программной документацией. *Документация на ПО* (software documentation) — совокупность документов, содержащих сведения, необходимые для разработки и использования ПО. *Программная документация* (program documentation) — совокупность документов, содержащих сведения, необходимые для использования ПО.

Наиболее общим рассматриваемым понятием является система (system). С точки зрения разработки система включает в себя вычислительные системы, персонал и т.д. Таким образом, система является более ёмким понятием, чем ПО: она включает в себя ещё и окружение, в котором функционирует ПО, как таковое.

С инженерно-технической точки зрения наиболее часто используют понятие «система», реже «ПО», «ПС» и «программное средство». Поэтому в большинстве излагаемого материала указание на ПО обычно можно заменить на ПС, программное средство и (с некоторыми условиями) на систему в целом.

Программный продукт

С организационной и экономической точек зрения наиболее часто используют понятия «прототип» и «программный продукт».

Программный продукт (ПП, product, software product) — программное средство, являющееся продуктом промышленного производства, предназначенным для поставки, передачи, продажи пользователю. При рассмотрении его только как результата промышленного производства используют обычно понятие *программное изделие* (ПИ, software product).

Кроме продукта (результата человеческого труда) интерес представляет и услуга (участие по оказанию помощи в деятельности). *Услуга* (service, тж. сервис) — деятельность по оказанию помощи в эксплуатации продукта.

Для удобства продукт и услуга именуются кратко — решение. *Решение* (solution) — результат в виде предоставляемого набора продуктов и/или услуг, необходимый для удовлетворения определённой потребности. Данное понятие чаще всего используется в области информационных технологий.

Прототип (prototype, букв. прообраз, опытный образец, макет) — частичная, предварительная или возможная реализация решения. Прототип является пробным или промежуточным результатом разработки, по которому оценивается решение в целом.

С решением тесно связаны два важнейших понятия — «проект» и «команда».

Проект (project) — это комплекс действий временного характера, направленных на получение конкретного решения; суть решения — его содержание. *Команда* (team) — группа лиц, сформированная для выполнения проекта или его части. Проект реализуется командами, а команда работает над проектами.

В проекте могут принимать участие разные лица. Под лицом здесь понимается физическое или юридическое лицо, т.е. один человек, группа людей или некоторая организация в целом. *Заинтересованное лицо* (stakeholder) — лицо, чьи интересы могут быть затронуты процессами и результатами проекта. Среди заинтересованных лиц выделяют участников проекта и, в частности, исполнителей.

Участник проекта (project participant) или *член проекта* (project member) — лицо, принимающее непосредственное участие в проекте. *Исполнитель* (executive) — участник проекта, выполняющий поставленную перед ним работу в рамках этого проекта и несущий ответственность за её выполнение.

В проекте каждый участник играет некоторую роль (или набор ролей). *Роль* (role) — характер поведения и области ответственности (responsibility) участника.

Выделяют следующие основные роли:

— пользователь (user) — лицо, которое непосредственно участвует в эксплуатации продукта для получения результатов (user —обычный пользователь) и/или использует эти результаты (end user — конечный пользователь);
— оператор (operator) — лицо, которое занимается эксплуатацией продукта;
— заказчик (customer, более узко: acquirer — приобретатель, покупатель) — лицо, которое заказывает разработку продукта и приобретает его;
— поставщик (supplier) — лицо, которое предоставляет разработанный продукт;
— разработчик (developer) — лицо, которое выполняет разработку продукта, т.е. все действия по разработке в рамках жизненного цикла проекта.

Часто для различных разработчиков выделяют отдельные роли — (системный) аналитик, проектировщик, программист (кодер), тестировщик и т.п.

Особенностями проекта (в отличие от любой другой деятельности) являются получение конкретного результата и ограниченность проекта временными рамками, определяемые жизненным циклом проекта.

Если конкретный результат не указан, обычно говорят о процессе разработки.

Процесс разработки (development process) — совокупность взаимосвязанных действий по получению некоторого результата. Процесс разработки в этом смысле представляет абстракцию проекта и фактически оказывается синонимом подхода к разработке ПО или системы. Это проявляется и в названии некоторых технологических подходов.

Жизненный цикл проекта

Продукт можно рассматривать аналогично живому организму: он имеет *жизненный цикл* (ЖЦ), который начинается с «зарождения» (возможно, с зарождения замысла / идеи) и заканчивается его «смертью» (изъятием из употребления). Концепция ЖЦ оказывается чрезвычайно полезной при управлении проектом.

Целью программного проекта является решение — программный продукт или услуга. Для проекта и соответствующего ему решения также принято выделять *ЖЦ проекта* (PLC — project life cycle). В зависимости от того, что включается в решение (ПО, вычислительная система и т.п.) и что входит в проект (только разработка или ещё и эксплуатация), выделяют соответствующие ЖЦ.

Жизненный цикл ПО (ЖЦ ПО, SLC — software life cycle) — весь период разработки ПО и его эксплуатации, начиная с момента возникновения замысла (идеи) и заканчивая прекращением всех видов его использования.

Кроме ЖЦ ПО следует отметить следующие тесно связанные с ним жизненные циклы (рис.1.2):

— ЖЦ разработки ПО (ЖЦ РПО, SDLC — software development life cycle), который в отличие от ЖЦ ПО не включает эксплуатацию и сопровождение ПО;
— ЖЦ системы (ЖЦС, SLC — system life cycle), который относится ко всей системе в целом;
— ЖЦ разработки системы (ЖЦ РС, SDLC — system development life cycle), который в отличие от ЖЦС не включает эксплуатацию и сопровождение системы (аналогично ЖЦ РПО).

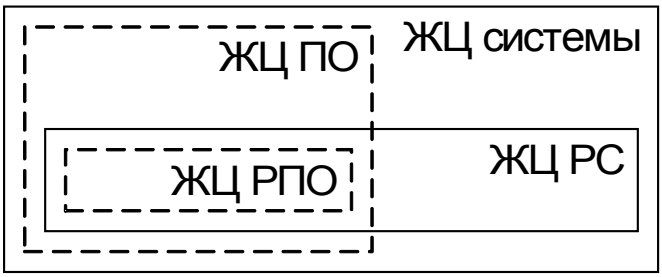


Рис.1.2. Взаимосвязь жизненных циклов

В настоящее время наибольший интерес представляет автоматизация поддержки системы и её разработки на всём протяжении ЖЦ, именно поэтому ведётся работа над общими стандартами на ЖЦ системы и ЖЦ её разработки и их согласование с имеющимися стандартами на ЖЦ ПО и ЖЦ его разработки.

1.2. Понятие «программирование»

Программирование может рассматриваться как научная дисциплина и как инженерная деятельность. В последнем случае под программированием можно понимать весь процесс создания программы или только непосредственно написание кода программы.

Программирование как дисциплина

Программирование рассматривают как научную дисциплину [1].

Информатика (Informatics) — наука, изучающая законы и методы накопления, обработки и передачи информации.

С теоретической точки зрения выделяют теоретическую информатику. *Теоретическая информатика* (букв. Theoretic Informatics) или *Информационная наука* (Information Science) — раздел информатики, изучающий информационные процессы и системы, в том числе структуру информации и её использование в различных областях человеческой деятельности.

С практической точки зрения выделяют прикладную информатику. *Прикладная информатика* (букв. Application Informatics) или *Вычислительная наука* (Computer Science) — совокупность разделов информатики и вычислительной техники, ориентированная на решение разнообразных вопросов автоматизации накопления, передачи и обработки информации.

Тогда программирование можно охарактеризовать следующим образом.

Программирование (Programming) — раздел информатики, изучающий описание процессов обработки данных. Следует отметить, что большинство разделов и направлений программирования обычно относят к прикладной информатике.

Разделы программирования

В программировании чётко выделяются разделы, перечисленные ниже.

1. *Теория программирования* (Programming Theory, тж. Programming Science — наука программирования): изучает математические абстракции программ, рассматриваемых как объекты, выраженные на формальном языке, обладающие определённой информационной и логической структурой и подлежащие автоматическому выполнению на компьютере. Это совокупность направлений, изучающих основные принципы программирования с помощью формальных математических методов.

Рис.1.3. Понятия и направления теории программирования

Она основывается на трёх понятиях: алгоритм, задача и вычислитель, и включает в себя следующие направления, связанные с ними (рис.1.3):

—структуры данных;

—информационный поиск и упорядочивание;

—формальные языки и грамматики, автоматы и другие абстрактные машины;

—синтаксический анализ программ;

—оценка трудоёмкости и теория сложности алгоритмов;

—эквивалентные преобразования алгоритмов;

—спецификация задач, доказательство свойств программ, автоматический синтез программ;

—семантика языков программирования (теория моделей программ).

2. *Методология программирования* (Programming Methodology): изучает методы с точки зрения основ их построения. Конкретная методология (методологический подход) — это объединённая основным принципом определённая совокупность методов, применяемых в процессе разработки ПО.
3. *Технология программирования* (Programming Technology): изучает процессы разработки ПО как технологические процессы, а также порядок их прохождения (с использованием знаний, методов и средств). Конкретная технология (технологический подход) содержит в себе определённый набор процессов, а также используемых в них знаний, методов и средств.
4. *Инженерия программирования* или *Программная инженерия* (SE — Software Engineering): изучает различные методы и инструментальные средства с точки зрения определённых целей, т.е. имеет очевидную практическую направленность. Инженерия понимается как инженерное дело, творческая техническая деятельность. Основная идея инженерии программирования состоит в том, что разработка ПО является формальным процессом, который можно изучать и совершенствовать.

Содержание инженерии весьма динамично и включает большое количество направлений, среди которых следует отметить следующие:

- процесс разработки в рамках проекта;
- моделирование ПрО;
- формирование требований к продукту;
- формальные спецификации;
- архитектура ПО;
- тестирование ПО;
- сопровождение и эволюция ПО;
- анализ ПО;

— инструментарий и окружение инженерии;
— математические основания инженерии;
— метрики ПО;
— экономика ПО;
— инженерия программирования специфичных систем (связующего обеспечения, систем реального времени, мобильных систем, распределённых систем, систем на основе Интернет и т.д.);
— инженерия программирования как учебная дисциплина.

Некоторые из этих направлений тесно связаны с методологией и технологией программирования, которые рассматривают их с соответствующих точек зрения.

5. *Инструментарий программирования* или *Программный инструментарий* (букв. Software Workbench): изучает системы программирования. Сюда входят все инструменты, поддерживающие процесс разработки ПО.

Замечания по терминологии

Понятие «технология». В настоящее время часто используются сочетания «CASE-технология», «Интернет-технология», «Java-технология» и т.п. В них слово «технология» применяется, как правило, в рекламных и маркетинговых акциях. В этом случае оно подчёркивает специфику определённого средства, поддерживающего ведение технологических процессов, например набора инструментов, совокупности стандартов или языка / среды программирования, и связанных с этим средством механизмов. Таким образом, в этих сочетаниях слово «технология» означает технологический инструментарий или (в общем случае) технологическую платформу.

Термин «метод». В каждом из таких разделов программирования, как методология, технология и инженерия, использовался термин «метод» (method). В общем случае *метод* представляет собой путь исследования или познания. Метод включает *средства* — с помощью чего осуществляется действие — и *способы* — каким образом осуществляется действие.

Рассмотрение метода. В методологии программирования методы рассматриваются с точки зрения *основ* их *построения*, в технологии программирования —

с точки зрения их *использования* при организации процессов, а в инженерии программирования — с точки зрения *достижения* с их помощью *определённых целей*.

В иностранной литературе по разработке ПО используется прежде всего последняя точка зрения. Поэтому большинство иностранных публикаций посвящено программной инженерии, а технологии часто именуются методологиями или инженериями, понимая под технологией только платформу, в частности, инструментарий (см. выше).

Направления программирования

Основные направления в программировании представлены тремя взаимосвязанными группами (рис.1.4).

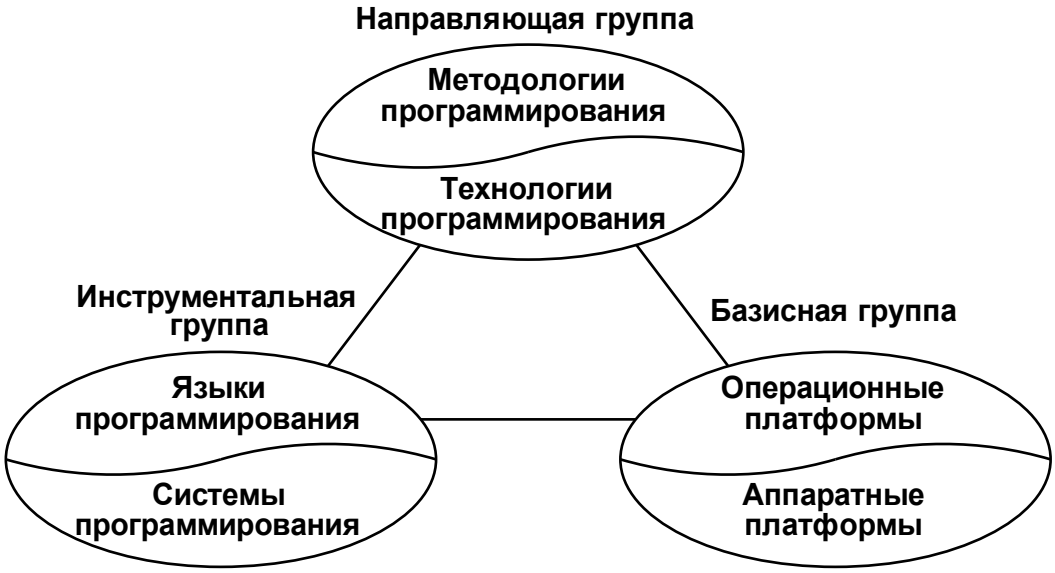


Рис.1.4. Группы направлений программирования

Первая — *направляющая* — группа содержит два направления, с которыми надо определиться перед началом работы с программным проектом. Выбранные методы и подходы определят основную идеологию и принципы проекта.

Вторая — *инструментальная* — группа содержит два направления — языковую и системную поддержку проекта.

Третья — *базисная* — группа содержит два направления, представляющие платформы — основу, на которой базируется программный проект.

Основная задача программирования на профессиональном уровне решается с помощью приведённых основных направлений.

Краткие рекомендации по их применению выглядят следующим образом:

1. Сначала следует определить методологию, которая будет включать совокупность методов и концепций, объединенных основным принципом.
2. Далее следует выбрать технологию, который будет определять совокупность процессов, применяемых при разработке программного продукта. Определенная ранее методология включает совокупность методов, которые будут применены в технологическом подходе.
3. Методология и технология определяют языки и системы программирования, необходимые для каждого процесса избранного технологического подхода.
4. Технологические процессы будут исполняться на некоторых аппаратной и операционной платформах. Заметим, что аппаратная и операционная платформы могут существенно определять наличие и специфику инструментов (систем программирования). В большинстве разработок следует избегать зависимости от платформ, однако ряд проектов (как правило, системных) в большой степени опирается на их хорошее знание.

Программирование как деятельность

Программирование также рассматривают как инженерную деятельность [4].

Программирование (Programming) — это совокупность процессов, связанных с созданием ПО и его реализацией. В связи с усложнением ПО и его создания вместо понятия «программирование» в настоящее время используется более общее понятие — *разработка программного обеспечения* (РПО, SD — software development). Исходное понятие сохранило своё значение только в узком смысле — как написание программы (program writing), синонимами которого являются кодирование (coding) и реализация (implementation).

В литературе, посвящённой разработке ПО, это изменение проявляется в использовании других названий для разделов программирования: теория программирования обычно называется Software Science (теория ПО, букв. наука ПО); методология программирования — Software Development Methodology (SDM, методология разработки ПО); технология программирования — Software Development

Technology (SDT, технология разработки ПО). С этой точки зрения программную инженерию называют также инженерией ПО, а программный инструментарий — инструментарием ПО.

В иностранной литературе большинство вопросов методологии и технологии разработки ПО принято рассматривать с инженерной точки зрения и относить соответствующие направления разработки к инженерии ПО.

В настоящее время активно развивается область, называемая *Системная инженерия* (SE — System Engineering, тж. Инженерия систем), в рамках которой речь идёт уже о разработке систем. Смена терминологии является результатом влияния системного подхода. В данном случае это означает понимание того, что необходимо учитывать не только само ПО, но его окружение, т.е. систему в целом, в котором ПО — всего лишь определённая (пусть и существенная) её часть.

1.3. Области разработки ПО

В рамках дисциплины «Технологии разработки программных систем» рассматриваются следующие области разработки ПО:

1. Методология разработки ПО (виды методологий и т.п. — см. раздел 2).
2. Технология разработки ПО (жизненный цикл ПО и т.п. — см. раздел 3, технологические подходы разработки ПО — см. раздел 4).
3. Ряд направлений инженерии ПО (практические вопросы проектирования и программирования и т.п., см. §5.1).
4. Ряд направлений инструментария ПО (среды, системы и средства разработки ПО и т.п., см. §5.2).

В рамках смежных дисциплин подробно рассматриваются следующие области разработки ПО:

1. Все направления (математические, алгоритмические и формально-языковые и другие аспекты) теории программирования.
2. Ряд направлений инженерии ПО (качество ПО и т.п.).
3. Ряд направлений инструментария ПО (конкретные языки и среды программирования и т.п.).

4. Управление разработкой ПО (управление проектами и т.п.).
5. Ряд специфических областей разработки ПО (бизнес-моделирование, анализ требований, тестирование и отладка и т.д.).

Необходимые для изучения представления из этих областей рассматриваются в рамках данного пособия для обеспечения систематизированного изложения курса и не претендуют на полноту.

Контрольные вопросы

Вопросы к §1.1

1. Дайте определение понятиям «система» и «процесс».
2. Дайте определение понятиям, связанным с системой.
3. Дайте определение понятиям, связанным с процессом.
4. Как связаны между собой понятия «система» и «процесс»?
5. Дайте определение понятиям «алгоритм» и «программа».
6. Дайте определение понятиям «программный модуль» и «программный компонент» как частям программы.
7. Дайте определение понятиям «пакет программ», «программный комплекс» и «программная система» как совокупностям программ.
8. Дайте определение понятиям «программное обеспечение» (ПО) и «программное средство».
9. Что понимается под компьютером и вычислительной системой?
10. Дайте определение понятиям «документация на ПО» и «программная документация».
11. Дайте определение понятиям «программный продукт» и «услуга».
12. Дайте определение понятиям «решение» и «прототип».
13. Дайте определение понятиям «проект» и «команда».
14. Дайте определение понятию «заинтересованное лицо».
15. Дайте определение понятиям «участник проекта» и «исполнитель».
16. Дайте определение понятию «роль». Перечислите основные роли участников.
17. В чём заключается концепция жизненного цикла?
18. Дайте определение понятиям, связанным с жизненным циклом проекта.
19. Как связаны между собой различные жизненные циклы? Приведите графическое представление этой взаимосвязи.

Вопросы к §1.2

20. Дайте определение понятиям, связанным с информатикой.
21. Дайте определение понятию «программирование» как научной дисциплине.
22. Перечислите и поясните разделы программирования.
23. Что представляет собой теория программирования?
24. Что представляет собой методология программирования?
25. Что представляет собой технология программирования?
26. Что представляет собой инженерия программирования?
27. Что представляет собой инструментарий программирования?
28. В чём заключается особенность использования понятия «технология».
29. Что представляет собой метод? Что включает в себя метод?
30. Как рассматривается метод в различных разделах программирования?
31. Перечислите основные направления программирования и их группы.
32. Сформулируйте рекомендации по применению основных направлений программирования при разработке ПО.
33. Дайте определение понятию «программирование» как инженерной деятельности.
34. В чём заключается разница между разработкой ПО и современным пониманием понятия «программирование»?
35. Что подразумевает разработка в рамках системной инженерии?

Вопросы к §1.3

36. Какие области разработки ПО изучаются в рамках данной дисциплины?
37. Какие области разработки ПО изучаются в рамках смежных дисциплин?

Раздел 2. Методология разработки ПО

В данном разделе рассматриваются основные методологии разработки ПО. Конкретная методология (методологический подход) представляет собой набор методов, объединённых некоторым общим принципом и поддерживаемых соответствующим набором концепций.

2.1. Основные понятия и определения

В узком смысле методология представляет собой определённый методологический подход. На сегодняшний день существует не так много методологий, особенно полных, т.е. учитывающих весь ЖЦ. Именно методология определяет, какие языки и системы программирования будут применяться для разработки ПО и какой технологический подход будет при этом использован.

Атрибуты методологий

Методология разработки ПО (методологический подход — methodology approach) — это объединённая основным принципом определённая совокупность методов, применяемых в процессе разработки ПО (по Г. Бучу).

С каждой методологией можно связать характерные для неё атрибуты:

1. Основной принцип, являющийся простым для формулирования и определяющий источник эффективности методологии.
2. Согласованное, связное множество методов, через которые реализуется данная методология.
3. Концепции (понятия, идеи), поддерживающие методы и позволяющие более точно их определить.

В рамках методологии разработки разрабатываются методики, которые применяются в подходах разработки (см. §3.1, «Технологический подход»).

Любая методология создаётся на основе уже накопленных в ПрО эмпирических фактов и практических результатов. Для методологий разработки ПО такими фактами и результатами являются уже существующие языки программирования.

Парадигма программирования

Когда методология применяется во время кодирования (программирования), очень часто её называют *парадигмой программирования* (programming paradigm) — способом мышления и программирования, не связанным с конкретным языком программирования.

Термин «парадигма» был впервые предложен Томасом Куном; он определял парадигму как свод норм научного мышления. *Парадигма* — это правило развития научного знания. Она в течение определённого времени даёт научному сообществу модель постановки проблем и их решений.

2.2. Классификация методологий

Один из подходов к классификации методологий, изложенный в [1], заключается в том, что существует некоторое ядро методологии со своими методами, которое уточняется некоторыми дополнительными особенностями — спецификами.

Этот подход напоминает принцип словообразования в русском языке — есть корень, к которому добавляются аффиксы (приставки, суффиксы и окончания), уточняющие смысл слова.

Ядра методологий

Ядро методологий (methodology core) определяется способом описания алгоритмов. Выделяют следующие основные ядра методологий:

- методология императивного программирования;
- методология объектно-ориентированного программирования;
- методология функционального программирования;
- методология логического программирования;
- методология ограничительного программирования.

Все перечисленные методологии находятся в диапазоне между двумя фундаментальными понятиями информатики — алгоритма и модели. В данном случае они перечислены в порядке уменьшения связи методологии с понятием алгоритм и увеличением связи с понятием модель.

Топологическая специфика

Каждое из «корней»-ядер может получить «приставку», определяемую некоторой *топологией* (topology) — структуру (информационных, управляющих или логических) узлов и связей программ, которая может быть хорошей или плохой. Топологии определяются совокупностью многочисленных факторов, связанных с абстракциями данных, управления и модульности. Например, к хорошей топологии приводит отказ от использования глобальных данных и оператора безусловного перехода, правильная модульная организация.

Пример. Если в императивной методологии придерживаться методов структурного представления (дающих хорошую топологию с позиций всех упомянутых абстракций), то мы получим хорошо известную методологию: структурного императивного программирования, которая более известна под её кратким именем — методология структурного программирования.

Следует отметить, что успех методологии объектно-ориентированного программирования изначально определила её хорошая топология, базирующаяся на абстрактных типах данных.

Реализационная специфика

Каждое из «корней»-ядер может получить «суффикс», определяющий некоторую *реализацию* (implementation) — организацию аппаратной поддержки — данной методологии. На данный момент наиболее известными являются две реализации — централизованная и распределённая, на основе которых строятся соответственно последовательные и параллельная методологии. К последовательным методологиям относят обычно фактически все ядра методологий, а к параллельным — соответствующие им модификации.

Примеры параллельных методологий:

— методология императивного параллельного программирования, её краткое название — методология параллельного программирования;
— методология логического параллельного программирования.

Для методологии объектно-ориентированного программирования параллельность неявно используется уже на уровне методов и концепций.

Смешанные методологии

Смешанные методологии основываются на объединении ряда методов нескольких (обычно родственных) методологий.

Наиболее часто объединяются методологии функционального и логического программирования, рассматриваемые как *методология декларативного программирования* (declarative programming methodology) в противоположность *методологии директивного программирования* (directive programming methodology), под которым понимают методологию императивного программирования и его модификации, а также методологию объектно-ориентированного программирования.

Кроме того, в рамках декларативного программирования часто выделяют методологию сентенциального программирования как самостоятельный подход.

Проводятся исследования в области объединения других методологий, в частности объектно-ориентированного и логического программирования. Ряд исследований посвящён вопросам унификации методологий программирования.

Другие методологии разработки ПО

В эту классификацию не вошли многие существующие методологии.

К известным, но редко выделяемым явно, относят следующие методологии:

— методология событийного программирования — подход, использующий взаимодействия через события при функционировании системы. Подход имеет два связанных между собой варианта: программирование на основе событий и программирование на основе приоритетов.

— методология автоматного программирования — подход, представляющий функционирование системы в виде конечного автомата. Подход имеет также другое название — программирование на основе состояний.

К мало известным (в настоящее время) относят следующие методологии:

— методология программирования, управляемого потоком данных, — подход, заключающийся в том, что операции срабатывают не последовательно, а в зависимости от готовности данных;

— методология доступ-ориентированного программирования — подход, в котором функции связываются с переменными таким образом, что при доступе к переменной автоматически будет вызываться соответствующая функция;
— методология нейросетевого программирования — подход, заключающийся в том, что на основе знаний от экспертов создаётся программа на нейронном языке программирования, которая затем компилируется в эквивалентную нейронную сеть из аналоговых нейронов.

Из приведённого далеко неполного списка методологий видно, что новые методологии основываются на применении к разработке ПО идей из самых разных областей научно-технической деятельности: операционные системы (прерывания), автоматическое управление (теория конечных автоматов), вычислительные системы (системы, управляемые потоком данных), программирование (доступ-ориентированное программирование), оптимизация (нейронные сети) и т.д.

Поддержка классов решаемых задач

Различные методологии программирования дают разный выигрыш для решения задач различных классов. Этот выигрыш можно оценить по двум параметрам:

1. Эффективность ПО на современных компьютерах.
2. Общие затраты на разработку ПО.

В частности в зависимости от вида эффективности ПО выделяют две ветви ориентации в развитии языков, поддерживающих методологии:

1. Ориентация на скорость исполнения кода программы.
2. Ориентация на высокий уровень и удобство программирования.

К первому классу относятся, как правило, только компилируемые языки, а ко второму — некоторые компилируемые и все интерпретируемые языки. В настоящее время такое разделение становится уже условным.

2.3. Происхождение методологий

Достоверная информация о происхождении методологий может быть полезной для их более точной классификации и предсказания появления и развития новых методологий.

В настоящее время выделяют три точки зрения на происхождение методологий: практическая, алгоритмическая и структурно-языковая.

Практическое происхождение

Первый и самый очевидный ответ на вопрос происхождения методологий заключается в их независимом происхождении на основе практического опыта программистов. Следует отметить, что очень многие методологии имеют автора-создателя или основателя научной школы, исследующей данную методологию.

В этом случае методологию можно рассматривать как концентрацию практического опыта программирования. Эта точка зрения позволяет оценить сложность создания достаточно подробной классификации методологий, которая охватывала бы большую часть существующих и развивающихся методологий, чрезвычайно разнородных по используемым в них методам и принципам.

Алгоритмическое происхождение

Это объяснение происходит из следующего утверждения:

Теория алгоритмов и логика — родители программирования.

Выделяют следующие 4 главные модели алгоритма:

1. Абстрактные вычислительные машины Тьюринга (Alan M. Turing — Алан М. Тьюринг) и Поста (Emily L. Post — Эмиль Л. Пост). Они определяют методологии императивного, автоматного и событийного программирования.
2. Рекурсивные функции Гильберта (David Hilbert — Давид Гильберт) и Аккермана (Wilhelm Friedrich Ackermann — Вильгельм Фридрих Аккерман). От них унаследовала свои идеи и конструкции методология структурного программирования.
3. Комбинаторная логика Шейнфинкеля (Moses Ilyich Schönfinkel — Моисей Исаевич Шейнфинкель) и Карри (Haskell Brooks Curry — Хаскел Брукс

Карри) и её современное представление — лямбда-исчисление Чёрча (Alonzo Church — Алонзо Чёрч). Эти идеи активно развиваются в методологии функционального программирования.

4. Нормальные алгорифмы Маркова (A.A. Markov — Андрей Андреевич Марков). Модель послужила основой методологий логического программирования и сентенциального программирования.

Рассмотренные главные модели алгоритма, математически эквивалентны, но на практике они породили разные направления в разработке ПО, в том числе некоторые методологии.

Структурно-языковое происхождение

Ещё одно объяснение берёт за основу понятие отображения структур языка. Согласно этой точке зрения сущность языка определяют три его составные части:

1. Структура данных (Д) — представление данных (и результатов).
2. Структура управления (У) — преобразование исходных данных в результат.
3. Логическая структура (Л) — определение преобразования задачи в алгоритм.

Каждая из трёх структур языка моделирования (ПрО) может быть отображена на любую из структур языка программирования. При этом каждое такое отображение определяет либо некоторую методологию, либо, по крайней мере, достаточно серьёзный метод.

В итоге получаются следующие 9 отображений:

1. Д → Д: представляет процесс укрупнения данных и операций над ними и приводит к методам модульности и абстрактных типов данных.
2. У → У: связано с понижением уровня структуры управления языка моделирования, ведёт к идее методологии структурного программирования.
3. Л → Л: лежит в основе методологии логического программирования.
4. Д → У: активизирует пассивные данные, преобразуя их в активные процессы; лежит в основе методологий функционального и сентенциального программирования; в значительной степени определяет методологию объектно-ориентированного программирования.

5. Д → Л: даёт возможность по совокупности операций построить логическую структуру и определяет методологию ограничительного программирования.
6. У → Д: лежит в основе методов интерпретации; определяет методологию доступ-ориентированного программирования.
7. У → Л: лежит в основе методов расшифровки смысла задачи.
8. Л → Д: может быть связано с типизацией данных и определяет метод развитой системы типов и приведений; также может быть связано с интерпретаторами, реализующими языки с развитой логической структурой.
9. Л → У: может быть использовано в системах структурного синтеза.

В этом случае методологию можно рассматривать как результат отображений структур языка.

Каждая из трёх структур языка состоит из разнородных подструктур. Это позволяет построить более точную классификацию, взяв за основу уже отображения этих подструктур.

2.4. Методологии программирования

Рассмотрим кратко перечисленные выше ядра методологий и их производные с учётом специфик методологий. Подробное изложение этого материала можно найти в [1].

Для большинства из этих методологий рассмотрение выполняется по следующей схеме: краткое пояснение, происхождение, методы и поддерживающие их концепции, вычислительная модель, класс задач, для которых предназначена методология.

Методология императивного программирования

Методология императивного программирования (imperative programming methodology) — подход, характеризующийся принципом последовательного изменения состояния вычислителя пошаговым образом. При этом управление изменениями полностью определено и полностью контролируемо.

Происхождение. Это исторически первая поддерживаемая аппаратно методология. Она ориентирована на классическую фон-неймановскую модель, остававшуюся долгое время единственной аппаратной архитектурой, получившей широкое практическое применение.

Методы и концепции. Метод изменения состояний — заключается в последовательном изменении состояний. Метод поддерживается концепцией алгоритма. Метод управления потоком исполнения — заключается в пошаговом контроле управления. Метод поддерживается концепцией потока исполнения.

Вычислительная модель. Императивное программирование основано на описании последовательного изменения состояний вычислителя. В качестве математической модели императивное программирование использует машину Тьюринга — Поста — абстрактное вычислительное устройство, предложенное на заре компьютерной эры для описания алгоритмов.

Класс задач. Методология наиболее пригодна для решения задач, в которых последовательное исполнение каких-либо команд является естественным. Пример — управление современными аппаратными средствами. Поскольку практиче-

ски все современные компьютеры императивны, эта методология позволяет порождать достаточно эффективный исполняемый код. С ростом сложности задачи императивные программы становятся всё менее и менее читаемыми. Программирование и отладка больших программ (например, компиляторов), написанных исключительно на основе данной методологии, может затянуться на долгие годы.

Методология объектно-ориентированного программирования

Методология объектно-ориентированного программирования (object-oriented programming methodology) — подход, использующий объектную декомпозицию, при которой статическая структура системы описывается в терминах объектов и связей между ними, а поведение системы — в терминах обмена сообщениями между объектами.

Этот подход можно рассматривать как дальнейшее развитие методологии структурного императивного программирования на основе введения абстрактных структур данных и непосредственного представления объектов ПрО, удобного для их использования при программировании.

Происхождение. На возникновение объектного мышления оказали влияние моделирование и представление данных, графические пользовательские интерфейсы и системное программирование (с понятием «процесс»). Моделирование реальных систем потребовало естественного описания сущностей — объектов и событий. Позже оказалось, что концепции объектно-ориентированного программирования (ООП) являются достаточно полезным дополнением к традиционному структурному программированию.

Методы и концепции. Метод объектной декомпозиции — заключается в выделении объектов и связей между ними. Метод поддерживается концепциями инкапсуляции, наследования и полиморфизма. Метод абстрактных типов данных — метод, лежащий в основе инкапсуляции. Метод поддерживается концепцией абстрагирования. Метод пересылки сообщений — заключается в описании поведения системы в терминах обмена сообщениями между объектами. Метод поддерживается концепцией сообщения.

Вычислительная модель. Вычислительная модель чистого ООП поддерживает явно только одну операцию, которой является посылка сообщения объекту. Сообщения могут иметь параметры, являющиеся объектами. Само сообщение также является объектом.

Класс задач. Методология является мощным средством для моделирования отношений между объектами практически в любой ПрО. Особенно удобно и легко в объектах выразить взаимодействие между различными элементами графического интерфейса пользователя.

Методология функционального программирования

Методология функционального программирования (functional programming methodology) — подход, согласно которому в программах единственным действием является вызов функции, единственным способом разделения программы на части — введение имени для функции и задание для этого имени выражения, вычисляющего значение функции, а единственным правилом композиции — оператор суперпозиции функции.

Происхождение. Методология является одной из старейших. По происхождению она тесно связана с лямбда-исчислением, изобретенным ещё в начале 30-х гг. XX в. логиком Алонзо Чёрчем (Alonzo Church). Для многих эта методология стала ассоциироваться с языком Lisp, созданным Джоном Маккарти (John McCarthy) в конце 50-х гг. XX в.

Методы и концепции. Метод аппликативности — заключается в том, что программа есть выражение, составленное из применения функций к аргументам. Программа состоит из совокупности определений функций, представляющих собой вызовы других функций и предложений, управляющих последовательностью вызовов. Метод поддерживается концепцией функции. Метод рекурсивного поведения — заключается в самоповторяющемся поведении, возвращающемся к самому себе. Метод поддерживается концепцией рекурсии. Метод настраиваемости — заключается в том, что можно легко порождать новые программные объекты по образцу, как значения соответствующих выражений (применение порождающей

функции к параметрам образца). Этому способствует то, что не только программа, но и любой программный объект (в идеале) является выражением. Метод поддерживается концепцией образца.

Вычислительная модель. Функциональное программирование не содержит понятия времени: программы являются выражениями, а исполнение программ заключается в вычислении этих выражений. В качестве математической модели функциональное программирование использует лямбда-исчисление Чёрча.

Класс задач. Методология обычно применяется для решения тех задач, которых трудно сформулировать в терминах последовательных операций. В эту категорию попадают практически все задачи, связанные с искусственным интеллектом. Это такие задачи, как обработка естественного языка, экспертные консультирующие системы, проблемы зрительного восприятия, и многие другие.

Методология логического программирования

Методология логического программирования (logical programming methodology) — подход, согласно которому программа содержит описание проблемы в терминах фактов и логических формул, а решение проблемы система выполняет с помощью механизмов логического вывода.

Происхождение. Методология начинает свой отсчёт времени с конца 60-х гг. XX в., когда Корделл Грин (Cordell Green) предложил использовать резолюцию как основу логического программирования. Алан Колмероэ (Alain Colmerauer) создал язык логического программирования Prolog в 1971 г. В основе логических языков обычно лежит какое-либо логическое исчисление с крупноблочными правилами вывода.

Методы и концепции. Метод единообразного применения механизма логического доказательства к программе. Метод поддерживается концепцией доказательства. Метод унификации — механизм сопоставления с образцом для перестройки структур данных. Метод поддерживается концепцией подстановки.

Вычислительная модель. Логическое программирование — это программирование в терминах фактов и правил вывода.

Класс задач. Класс задач логического программирования практически совпадает с классом задач методологии функционального программирования.

Методология сентенциального программирования

Методология сентенциального программирования (sentential programming methodology) — подход, согласно которому суть программы состоит в перестройке чёткой и достаточно сложной структуры данных в рамках заданных условий.

Происхождение. Идея и основа подхода — нормальные алгорифмы Маркова. На основе подхода построены две модели, основанные на разных вариантах отождествления — конкретизации и унификации. Метод конкретизации считается основой методологии сентенциального программирования и языка Рефал, созданного В.Ф. Турчиным (Валентин Фёдорович Турчин) в 1986 г.

Метод унификации совместно с формализмом математической логики стал основой методологии логического программирования и языка Prolog. Содержательно этот язык следует относить к сентенциальному программированию, так как целью его разработки послужили задачи математической лингвистики. Использование рекурсии вместо отождествления приводит уже к методологии функционального программирования.

Методы и концепции. Основной метод — метод отождествления. Метод поддерживается концепцией подстановки.

Вычислительная модель. Сентенциальное программирование — это программирование в форме обработки метавыражений.

Класс задач. Класс задач сентенциального программирования практически совпадает с классом задач методологии функционального и логического программирования, хотя в большей степени ориентирован на символьные преобразования.

Методология ограничительного программирования

Методология ограничительного программирования (constraint programming methodology) — подход, согласно которому в программе определяется тип данных решения, ПрО решения и ограничения на значение искомого решения; решение находится системой.

В ряде работ встречаются следующие названия этой методологии: программирование в ограничениях, программирование в постановках [задач] и постановочное программирование.

Происхождение. Методология возникла в начале 80-х гг. XX в. как перспективная область исследований на пересечении символьных вычислений, искусственного интеллекта, исследования операций и интервальной арифметики.

Методы и концепции. Метод описательной модели вычислений заключается в том, что программа на языке программирования содержит описание понятий и задач. Метод поддерживается концепцией модели.

Вычислительная модель. Ограничительное программирование — это программирование в терминах постановок задач.

Класс задач. Класс задач ограничительного программирования — задачи исследования операций и искусственного интеллекта. В таких задачах часто используется некоторое пространство решений, сужением которого достигается необходимый результат. Такие сужения можно естественным образом представить как ограничения.

Методология структурного императивного программирования

Методология структурного [императивного] программирования (structural [imperative] programming methodology) — подход, заключающийся в задании хорошей топологии императивных программ, в том числе отказе от использования глобальных данных и оператора безусловного перехода, разработке модулей с сильной связностью и слабым сцеплением.

Подход базируется на двух основных принципах построения:

1. Последовательная декомпозиция алгоритма решения задачи сверху вниз.
2. Использование структурного кодирования.

В противоположность методологии ООП данная методология известна под названием *методология процедурно-ориентированного программирования* (procedure-oriented programming methodology).

Происхождение. Методология является важнейшим развитием императивной методологии. Создателем структурного подхода считается Э.В. Дейкстра (Edsger W. Dijkstra — Эдсгер Вайб Дейкстра). Ему также принадлежит попытка соединить структурное программирование с методами доказательства программ.

Методы и концепции. Метод алгоритмической декомпозиции сверху вниз — заключается в пошаговой детализации постановки задачи, начиная с наиболее общей задачи. Данный метод обеспечивает хорошую структурированность. Метод поддерживается концепцией алгоритма. Метод модульной организации частей программы — заключается в разбиении программы на специальные компоненты, называемые модулями. Метод поддерживается концепцией модуля.

Метод структурного кодирования — заключается в использовании при кодировании трёх основных управляющих конструкций (следование, разветвление, повторение). Метки и оператор безусловного перехода являются трудно отслеживаемыми связями, без которых нужно обойтись. Метод поддерживается концепцией управления.

Класс задач. Класс задач для данной методологии соответствует классу задач для императивной методологии. При этом удаётся разрабатывать более сложные программы, поскольку их легко воспринимать и анализировать.

Методология императивного параллельного программирования

Методология [императивного] параллельного программирования ([imperative] parallel programming methodology) — подход, в котором предлагается использование явных конструкций для параллельного исполнения выбранных фрагментов программ.

Происхождение. Анализ эффективности решений сложных вычислительных задач показал, что скорость вычислений можно значительно улучшить, если использовать не одно, а несколько устройств одновременно. Создание многопроцессорных систем привело к крупным исследованиям в этой области. Ещё одна причина возникновения этой методологии связана с появлением сложных программ, требующих поддержки явного параллелизма (например, операционных систем).

Методы и концепции. Метод синхронизации исполняемого кода — заключается в использовании специальных атомических операций для осуществления взаимодействия между одновременно исполняемыми фрагментами кода. Метод поддерживается концепцией примитивов синхронизации.

Класс задач. Методология может очень эффективно применяться для обработки больших однородных массивов данных. Такие массивы часто встречаются в реализации вычислительных и статистических методов. Кроме этого, данная методология успешно применяется при моделировании, в операционных системах и системах реального времени.

Методология логического параллельного программирования

Методология логического параллельного программирования (logical parallel programming methodology) — подход, обобщающий логическую методологию на параллельное выполнение доказательств.

Происхождение. Язык Concurrent Prolog, предложенный Эхудом Шапиро (Ehud Shapiro) развивает логический подход к абстрактным спецификациям и включает несколько идей Дейкстры (в том числе, идею охраняемого предложения). При этом подходе ПрО описывается как формальная теория в некотором логико-математическом языке.

Контрольные вопросы

Вопросы к §2.1

1. Дайте определение понятию «методология» («методологический подход»).
2. Какие атрибуты связаны с каждой методологией?
3. Дайте определение понятию «парадигма программирования».

Вопросы к §2.2

4. Поясните рассмотренный в разделе принцип классификации методологий.
5. Перечислите основные ядра методологий.
6. Перечислите специфики методологий и методологии с этими спецификами.
7. Приведите примеры смешанных методологий.
8. Приведите примеры известных методологий, не вошедших в классификацию.
9. Приведите примеры мало известных методологий, не вошедших в классификацию.
10. Как определяется поддержка методологиями классов решаемых задач?

Вопросы к §2.3

11. Охарактеризуйте практическую точку зрения на происхождение методологий?
12. Охарактеризуйте алгоритмическую точку зрения на происхождение методологий? Приведите модели алгоритмов и соответствующие методологии.
13. Охарактеризуйте структурно-языковую точку зрения на происхождение методологий? Приведите отображения структур и соответствующие методологии.

Вопросы к §2.4

14. В чём суть методологии императивного программирования?
15. В чём суть методологии объектно-ориентированного программирования?
16. В чём суть методологии функционального программирования?
17. В чём суть методологии логического программирования?
18. В чём суть методологии сентенциального программирования?

19. В чём суть методологии ограничительного программирования?
20. В чём суть методологии структурного императивного программирования?
21. В чём суть методологии императивного параллельного программирования?
22. В чём суть методологии логического параллельного программирования?

Раздел 3. Технология разработки ПО

В данном разделе рассматриваются основные понятия технологии разработки ПО, модели ЖЦ ПО и классификации процессов, стадий и подходов. Конкретная технология (технологический подход) содержит в себе определённый набор процессов, а также используемых в них знаний, методов и средств.

3.1. Основные понятия и определения

В узком смысле технология представляет собой определённый технологический подход. Для технологии определяющими являются понятия «жизненный цикл» и «модель жизненного цикла», а также понятия «процесс» и «стадия».

Жизненный цикл ПО

Современные технологии разработки опираются на понятие *жизненного цикла* (LC — life cycle).

Жизненный цикл программного обеспечения (ЖЦ ПО, SLC — software life cycle) — весь период его разработки и эксплуатации, начиная с момента возникновения замысла (идеи) и заканчивая прекращением всех видов его использования.

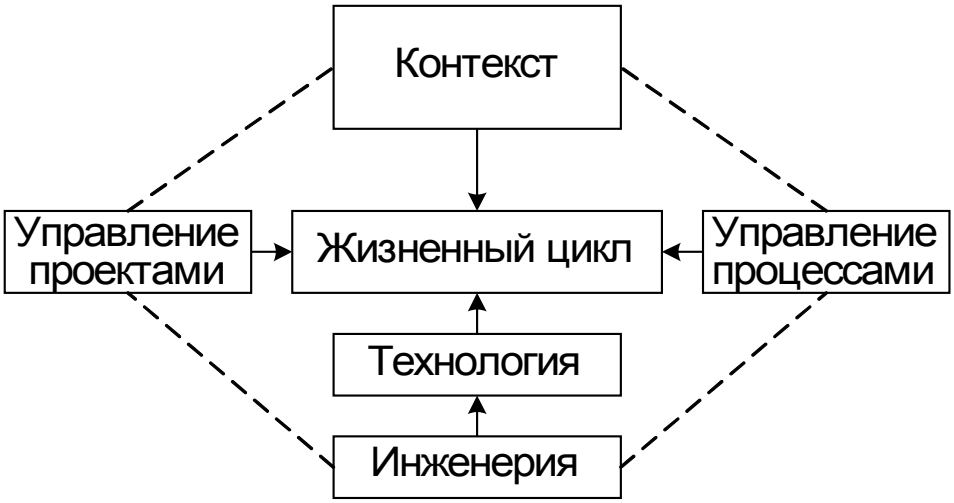


Рис.3.1. Представление ЖЦ в контексте деятельности

ЖЦ принято рассматривать в определённом контексте выполняемой деятельности и технологии, а также во взаимосвязи с управлением разработкой и инженерией [5] (рис.3.1). Контекст может быть культурный, организационный, процессный, технологический или иной. Составляющие технологии разработки пере-

числены ниже. Управление разработкой подразумевает здесь управление проектами и управление процессами. Инженерия в зависимости от ЖЦ может быть программной или системной.

В общем случае ЖЦ определяется *моделью* и описывается в форме технологии разработки — *технологического подхода*.

Модель ЖЦ (life cycle model) — структура, определяющая последовательность выполнения процессов и их взаимосвязь на протяжении ЖЦ. Упоминание ЖЦ обычно подразумевает указание какой-либо конкретной модели ЖЦ.

Технологический подход

Технология разработки ПО (технологический подход — technology approach) — это определённая совокупность процессов, включающих их детальное содержание и распределение по стадиям, а также ролевую ответственность участников проекта на всех стадиях выбранной модели ЖЦ ПО.

Технология часто определяет и саму модель. Обычно она основывается на методиках выбранной методологии разработки, а также рекомендует практики, что позволяет максимально эффективно воспользоваться соответствующей технологией и её моделью ЖЦ.

Аналогичным образом можно определить технологию разработки системы.

Процессы и стадии

Технологию удобно характеризовать в двух измерениях — вертикальном (представляющем процессы) и горизонтальном (представляющем стадии).

Связующим понятием между процессами и стадиями является «действие».

Действие (activity, тж. работа, вид деятельности) — часть деятельности по проекту, выполняемая отдельным исполнителем или группой исполнителей.

Фактически процессы и стадии представляют собой определённые наборы действий: по признаку преобразования данных действия объединяются в процессы, а повременному признаку и/или получаемому результату — в стадии.

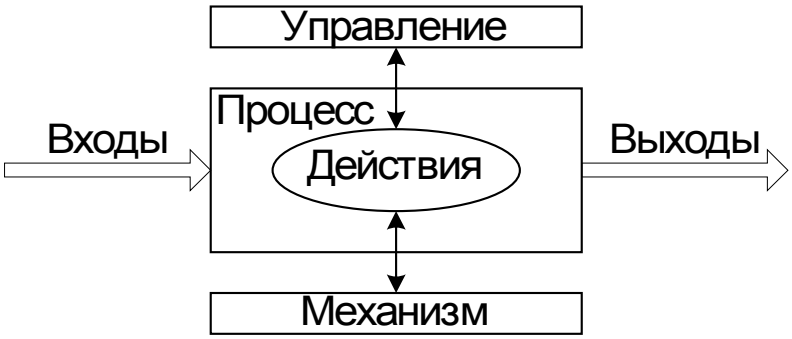


Рис.3.2. Представление процесса на диаграмме ICOM

Процесс (process) — совокупность взаимосвязанных действий проекта, преобразующих некоторые входные данные в выходные. Взаимосвязь действий заключается в их последовательности, завершённой с точки зрения содержания, временной и логической очерёдности.

Графически процесс представляется следующим образом (рис.3.2): на основе входов (**I** — Input, тж. ввод) при используемом управлении (**C** — Control, тж. регулирование) и механизме (**M** — Mechanism, тж. обеспечение) в результате выполнения действий процесса получаются выходы (**O** — Output, тж. вывод). Выход одного процесса может являться входом другого.

Управление заключается в организации процесса с учётом заданных ограничений (constraints). К ограничениям относятся время, стоимость, качество, характеристики процесса и т.д. Механизм включает в себя ресурсы (resources) и роли (roles) заинтересованных лиц. К ресурсам относятся вычислительные системы, персонал, финансы, материалы, применяемые методология и технология и т.п.

Процессы состоят из набора *действий*, а каждое действие — из набора *задач*. Дальнейшая детализация приводит к рассмотрению отдельных *операций* (operations). *Задача* (task, тж. задание) — планируемый элемент действия: задача определяется в плане проекта и её могут быть назначены ресурсы для выполнения.

Таким образом, иерархия понятий, связанных с процессом, выглядит следующим образом: Процессы → Действия → Задачи → Операции.

В некоторых подходах вместо понятия «процесс» используют понятие «поток работ». *Поток работ* (workflow — поток работ и документооборота, букв. рабочий поток, в некоторых переводах — дисциплина) — процесс, рассматриваемый

вместе с соответствующими ему артефактами и ролями. С точки зрения управления поток работ связан с *рабочим продуктом* (work product) — артефактом, производимым участником в некоторой заданной роли.

В ряде работ вводится также понятие «процедура». *Процедура* (procedure) — пошаговое описание направления задач для выполнения и завершения конкретного действия. В этом случае описание процесса представляет собой документированное определение действий, формализованных в виде процедур.

Стадия (stage) — группа действий проекта, ограниченная некоторыми временными рамками и часто заканчивающаяся выпуском произведённого результата, определяемого заданными требованиями. Стадии выделяются исходя из соображений разумного и рационального управления проектом.

Стадии часто состоят из *этапов* (step, тж. шаг), которые обычно имеют итерационный характер и поэтому представляются в виде итераций. В ряде подходов стадии объединяют в более крупные временные рамки — *фазы* (phases), в этом случае сами стадии имеют итерационный характер.

Таким образом, иерархия понятий, связанных со стадией, выглядит следующим образом: Фазы → Стадии → Этапы.

Тогда получаем следующее описание измерений технологии:

1. Вертикальное измерение отражает статические аспекты процессов и оперирует такими понятиями, как процессы, действия, задачи и операции.
2. Горизонтальное измерение представляет время, отражает динамические аспекты процессов и оперирует такими понятиями, как фазы, стадии и этапы.

Методики и практики

Как было отмечено выше, технология разработки основывается на методиках и рекомендует практики.

Методика (technique, букв. техника) — совокупность конкретизированных методов разработки в рамках заданной методологии, применяемая в одном или нескольких соответствующих этой совокупности процессах ЖЦ.

Практика (practice) — это определённая рекомендация по выполнению действий, для которых результаты проверяемы, но не передаваемы как материал для работы других процессов. Последовательность практик и последовательность действий внутри практики не задана. Практики не привязаны к проекту (при этом говорят, что у них нет «экземпляра»).

В отличие от практики процесс — совокупность действий заданной последовательности выполнения. Результаты одних действий идут как входы других действий. Процессы тесно связаны с проектом, в рамках которого они выполняются (при этом говорят, что они имеют «экземпляр» — сам проект).

Таким образом, технология определяется спецификой комбинации процессов и стадий, ориентированной на разные классы ПО и особенности участников проекта и дополненной методиками и практиками.

Управление разработкой

Как и для процесса, управление проектом заключается в организации проекта с учётом заданных ограничений. При этом эффективность управления проектом зависит от правильной оценки состояния этого проекта в любой момент времени. Эта оценка, в свою очередь, определяется обзором произведённых результатов.

Представление ограничений

Система ограничений определяется совокупностью приоритетов, установленных для проекта, и должна учитывать требования заинтересованных лиц.

К основным ограничениям относят:

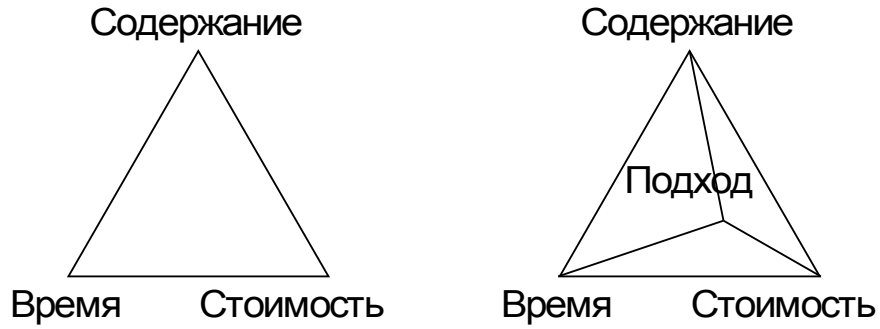
1. Содержание (проекта) — Scope.
2. Время (выполнения) — Time.
3. Стоимость (проекта) — Cost.

Кроме этого следует отметить и следующие ограничения, тесно связанные с основными ограничениями:

4. Ресурсы (людские и финансовые) — Resources.
5. Качество (приемлемое для проекта) — Quality.
6. Эффективность (результата проекта) — Efficiency.

В результате при учёте каждого имеющегося ограничения (constraint) получается некоторый многоугольник или многогранник ограничений. Но на практике оказывается, что по крайней мере три ограничения оказываются фиксированными. В этом случае говорят о треугольнике ограничений.

Часто встречающимся представлением системы ограничений является железный треугольник (рис.3.3, а). *Железный треугольник* (iron triangle) — это треугольник, каждая вершина которого представляет собой ограничение. Все вершины этого треугольника не должны фиксироваться одновременно, иначе получается перегрузка проекта ограничениями. Однако существует ещё одна вершина, которая всегда присутствует неявно, формируя четырёхгранник — пирамиду ограничений (рис.3.3, б). Это выбираемый подход разработки.

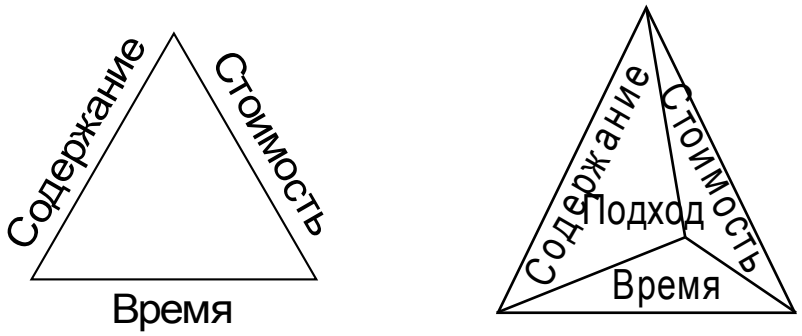


а) Железный треугольник б) Пирамида ограничений

Рис.3.3. Железный треугольник и пирамида ограничений

Другим представлением системы ограничений является треугольник компромиссов (рис.3.4, а). *Треугольник компромиссов* (tradeoff triangle) — это треугольник, каждая сторона которого представляет собой ограничение. Никакая из сторон этого треугольника не может быть изменена без воздействия на другие стороны. Поэтому изменение одного ограничения требует пересмотра других ограничений для постоянного поддержания баланса в этом треугольнике. Здесь также можно рассмотреть пирамиду компромиссов (рис.3.4, б). В этом четырёхугольнике ограничения представлены её гранями, а дополнительной гранью является подход разработки.

Следовательно, именно выбор технологического подхода, связанного с совокупностью процессов и стадий, сочетанием методик и практик, а также с уровнем формализации разработки, позволяет смягчить железный треугольник и сбалансировать треугольник компромиссов.



а) Треугольник компромиссов б) Пирамида компромиссов

Рис.3.4. Треугольник и пирамида компромиссов

Таким образом, классическая проблема управления заключается в поиске баланса в рамках системы ограничений, связанных с проектом.

Формализация разработки

Эффективное управление разработкой ПО требует определённых формализаций выполняемых работ и их результатов.

Выходами действий процесса могут быть собственно выходы (output — вывод, краткосрочный результат), исходы (outcome — последствие, среднесрочный результат) или произведённые результаты (долгосрочные результаты).

Произведённый результат (deliverable — выдаваемый результат) — определённый реальный результат, произведённый проектом. Различают следующие произведённые результаты: *внутренний* — для использования проектной командой, и *внешний* — для использования другими заинтересованными лицами.

Формализацией произведённого результата обычно является некоторый артефакт. Примерами артефактов являются модели и документы, такие как планы, спецификации, схемы, диаграммы, программный код и многое другое.

Артефакт (artifact) — формальный произведённый результат, создаваемый, изменяемый или используемый в нём при выполнении проекта. Артефакт определяет область ответственности: за его создание отвечают определённые исполните-

ли или их группы. В рамках отдельной задачи (и таймбокса) формальным произведённым результатом является *рабочий продукт* (work product), но в ряде подходов это понятие выступает синонимом артефакта.

Для формализации внесения изменений в произведённый результат используется понятие «базовая линия».

Базовая линия (baseline) — официально принятый вариант произведённого результата, обозначенный и зафиксированный в конкретный момент времени ЖЦ. Изменения, вносимые в базовую линию, должны быть предварительно утверждены, т.е. должны пройти через специальное формализованное действие. Для проекта в целом базовая линия обычно переводится как *базовый план* (baseline) — исходный план проекта с утверждёнными изменениями.

Для анализа проекта используются контрольные точки и вехи.

Контрольная точка (checkpoint) — событие в ЖЦ для проверки выполненной и оценки оставшейся работы по проекту. Моментами времени для контрольных точек часто выступают границы стадий.

Веха (milestone) — событие ЖЦ для обозначения завершения произведённого результата или их набора. Вехи часто используются в качестве контрольных точек.

Артефакты и вехи являются необходимыми элементами формализации для управления выполняемым проектом.

Вследствие итерационного выполнения работ (например, некоторого процесса и соответствующей ему стадии) получаемые результаты постепенно улучшаются до целевых результатов, диктуемых заданными требованиями, которые также могут изменяться.

Итерация (iteration) — ограниченная во времени повторяемая часть проекта. Итерацией может выступать весь цикл разработки или его часть (стадии или этапы), что определяется их длительностью и используемой моделью ЖЦ. В этом случае итерация цикла разработки называется макро-итерацией или просто *циклом*, итерация стадии — просто *итерацией*, а итерация этапа — микро-итерацией.

Для управляемого выполнения отдельных задач используются таймбоксы.

Таймбокс (timebox — букв. временной ящик, время-блок, time frame — временная рамка) — небольшой промежуток времени для выполнения конкретной задачи и произведения её результатов. Таймбокс характеризуется жёстко заданными временными рамками задачи. Эти рамки необходимо соблюсти, даже если придётся выдать не вполне завершённые результаты. В противном случае задача считается проваленной и приходится или отказаться от неё, или запланировать её повторно.

В зависимости от числа исполнителей таймбокс может занимать разное время: *персональный* — от нескольких часов до дня, *командный* — от нескольких дней до месяца. В зависимости от изменяемости требований выделяют: *открытый*, учитывающий изменяемые требования, и *замкнутый*, считающий требования фиксированными до своего конца.

Таймбоксы и итерации используются для управления работами проекта.

Таким образом, описание измерений технологии с учётом изложенного корректируется следующим образом:

1. Вертикальное измерение оперирует также такими понятиями, как произведённые результаты (в том числе артефакты, рабочие продукты) и исполнители (в том числе роли и ответственности).
2. Горизонтальное измерение оперирует также такими понятиями, как контрольные точки и вехи, итерации и таймбоксы.

Однако введённые понятия можно рассматривать и как составляющие технологии, характеризующие её в новом, третьем измерении. Это измерение оказывается связанным с формализацией, необходимой для управления проектом.

3.2. Основные классификации

Рассмотрим наиболее часто используемые классификации технологических процессов, стадий и подходов. Процессы и стадии классифицируются исходя из соображений разумного и рационального управления проектом.

Классификация процессов

Существует два основных набора технологических процессов:

1. *Классический набор* — совокупность основных процессов, сложившихся исторически в результате практического опыта разработки ПО.
2. *Стандартный набор* — совокупность процессов, определённых в стандарте ISO/IEC 12207:1995 «Information Technology — Software Life Cycle Processes» («Информационная технология — Процессы жизненного цикла ПО»).

Процессы классического набора фактически являются подмножеством стандартного, выступая там как процессы или действия процессов.

Классический набор включает 9 технологических процессов (в скобках после названий процессов приведены часто используемые сокращения этих названий):

1. Исследование [идеи] (И) — [Idea] Discovery (ID).
2. Управление [проектом] (У) — [Project] Management (PM).
3. Анализ [требований] (А) — [Requirements] Analysis (A).
4. Проектирование (Д — дизайн) — Design (D).
5. Кодирование, Конструирование (К) — Coding, Construction (C).
6. Тестирование (Т) — Testing (T).
7. Ввод в действие (В) — Installation (I).
8. Сопровождение (С) — Maintenance (M).
9. Снятие с эксплуатации (СЭ) — Retirement (R).

Перечисленные процессы подробно рассматриваются в §3.4.

Следует только сделать следующие замечания по перечисленным процессам:

1. Процесс 1 выделяется не во всех работах по технологии разработки ПО. В других работах он соответствует системному анализу или анализу ПрО.
2. Процесс 2 выполняется во время всего ЖЦ ПО, поэтому в большинстве моделей он явно не отображается. В то же время одним из важных действий управления является планирование, после которого собственно и начинается процесс 3. Процесс 2 подробно изучается такой дисциплиной, как Управление проектами (Project Management).
3. Процессы 3 и 4 в большинстве проектов оказываются тесно связанными и потому часто рассматриваются вместе.
4. Процесс 4 обычно разделяется на два взаимосвязанных подпроцесса: проектирование архитектуры (проектирование «в большом») и проектирование компонентов (проектирование «в малом»).
5. Процесс 5 также может встречаться под названиями Реализация (Implementation) и Программирование (Programming). В русском языке чаще используют названия Кодирование или Программирование, так как Конструирование обычно относят только к разработке трансляторов. Ряд работ по технологии разработки ПО выделяют из этого процесса ещё один процесс, обычно именуемый Сборка или Интеграция (Integration).
6. Процесс 6 может выполняться как для отдельных частей разрабатываемого ПО, так и для всего продукта в целом. Для формальных разработок он имеет другое название — Инспектирование (Inspection).
7. Процесс 7 в большинстве моделей явно не отображается.
8. Процесс 8 выполняется во время эксплуатации продукта.
9. Процесс 9 начинает выполняться обычно после завершения процесса 8, поэтому в большинстве моделей он также явно не отображается.

   Процессы 7 — 9 не относятся к ЖЦ разработки (ПО или системы).

Стандартный набор включает существенно большее число процессов, объединённых в три тематические группы:

1. Основные процессы (приобретение, поставка, разработка, эксплуатация, сопровождение).
2. Вспомогательные процессы (документирование, управление конфигурацией, обеспечение качества, верификация, аттестация, совместная оценка, аудит, разрешение проблем).
3. Организационные процессы (управление, создание инфраструктуры, усовершенствование, обучение).

Перечисленные процессы подробно рассматриваются в §3.6.

Для поддержки практического применения этого стандарта разработан ряд технологических документов:

— ISO/IEC TR 15271:1998 «Information Technology — Guide for ISO/IEC 12207 (Software Life Cycle Processes)» («Информационная технология — Руководство для ISO/IEC 12207 (Процессы жизненного цикла ПО)») и
— ISO/IEC TR 16326:1999 «Software Engineering — Guide for the Application of ISO/IEC 12207 to Project Management» («Программная инженерия — Руководство по применению ISO/IEC 12207 при управлении проектом»).

В настоящее время продолжается разработка нового стандарта ISO/IEC 15288:2002 «Systems Engineering — System Life Cycle Processes» («Системная инженерия — Процессы жизненного цикла систем»), в частности ведётся согласование этого стандарта с предыдущим стандартом ISO/IEC 12207:1995. Для поддержки практического применения этого стандарта разработан технологический документ ISO/IEC TR 19760:2003 «Systems Engineering — A Guide for the Application of ISO/IEC 15288 (System Life Cycle Processes)» («Системная инженерия — Руководство по применению ISO/IEC 15288 (Процессы жизненного цикла систем)»).

Существуют или разрабатываются российские аналоги указанных выше стандартов и технологических документов ISO/IEC.

Существуют также другие наборы процессов, основанные на стандартах различных видов (международные, организационные, фирменные).

Классификация стадий

Существует два основных вида формирования технологических стадий:

1. *Попроцессное формирование стадий* — выделение стадий, отражающих названия процессов (или отдельных действий процессов), большая часть времени которых проходит в данной стадии.
2. *Пофазное формирование стадий* — объединение стадий в фазы, отражающие крупные временные промежутки.

Попроцессное формирование стадий обычно используют для классических процессов (либо их под- или надмножества).

В подходах с этой классификацией обычно выделяют 9 классических стадий:

1. Исследование идеи (ИИ) — Idea Discovery (ID).
2. Планирование (П) — Planning (P).
3. Анализ требований (АТ) — Requirements Analysis (RA).
4. Проектирование (Д — дизайн) — Design (D).
5. Кодирование, Конструирование (К) — Coding, Construction (C).
6. Тестирование и отладка (ТиО) — Testing and Debugging (T&D).
7. Ввод в действие (В, ВвД) — Installation (I).
8. Эксплуатация и сопровождение (ЭиС) — Operation and Maintenance (O&M).
9. Снятие с эксплуатации (СЭ) — Retirement (R).

Стадии 1 и 3 — 9 определяются соответствующими классическими процессами 1 и 3 — 9, а стадия 2 — действием классического процесса 2, так как процесс 2 выполняется на протяжении всего проекта. Стадия 6 для формальных разработок имеет другое название — Инспектирование (Inspection).

Пофазное формирование стадий обычно используют для стандартных процессов (либо их под- или надмножества).

В большинстве подходов с этой классификацией выделяют 4 основные фазы (приведены условные названия):

1. Начало — Изучение ПрО и получение требований.
2. Середина — Анализ требований и проектирование.
3. Кульминация — Конструирование (кодирование и тестирование).
4. Переход — Внедрение (ввод в действие и опытная эксплуатация).

В ряде подходов выделяют 2 дополнительные фазы:

5. Работа — Эксплуатация и сопровождение.
6. Окончание — Снятие с эксплуатации.

Моменты завершения фаз являются вехами. Фазы обычно разбиваются на несколько итерации. Кроме того, вводится такое понятие как *цикл* (cycle) — период времени, включающий все фазы. Оно оказывается необходимым при разработке сложных систем.

Классификация проектов

Классификация подходов тесно связана с характеристиками выполняемых проектов. По каждому признаку классификации проектов можно выделить множество проектов, для которых будут указаны только граничные значения.

По виду заказчика выделяют:

— Проект для конкретного заказчика.

— Проект для широкого круга пользователей.

Разница между этими проектами наиболее существенно проявляется в отличии процессов управления и исследования идеи.

По степени коммерциализации выделяют:

— Коммерческий проект: исполнители ориентируются на получение прибыли от продажи продукта.

— Некоммерческий проект: исполнители ориентируются на разработку продукта и его передачу пользователю без получения прибыли.

Участие в таком проекте мотивируется в основном следующим: приобретение опыта в конкретной области программирования, приобретение статуса

и уважения, обучение работе в коллективе. Практически всегда программный продукт поставляется с открытым исходным текстом и лицензией на запрет его использования в коммерческих целях.

По масштабу, определяющему количество исполнителей и протяжённость (время выполнения) проекта, выделяют 5 категории проектов (табл.3.1).

Табл. 3.1. Категории проектов

Категория	Количество исполнителей	Протяжённость проекта
мелкий	от 1 до 3	от 1 часа до 2 месяцев
малый	от 3 до 10	от 2 до 6 месяцев
средний	от 10 до 30	от 6 месяцев до 1 года
крупный	от 30 до 100	от 1 года до 2 лет
гигантский	от 100 до 300 и более	от 2 до 6 лет и более

В первую очередь именно масштаб проекта определяет основные классы технологических подходов.

Классификация подходов

В настоящее время выделяют два класса подходов: строгие и гибкие.

Строгие (strict, тж. heavy — тяжёлые, hard — жёсткие) *подходы* ориентированы в основном на применение в средних, крупных и гигантских проектах с фиксированным объёмом работ. Поэтому одно из основных требований к таким проектам — предсказуемость (predictability).

Гибкие (flexile, тж. light — лёгкие, agile — живые) *подходы* ориентированы в основном на применение в мелких, малых или средних проектах в случае неясных или изменяющихся требований к системе. Поэтому одно из основных требований к таким проектам — непосредственное участие заказчика в проекте (наличие квалифицированного представителя заказчика в команде). Кроме того для большинства гибких подходов важным является ещё одно из основных требований — адаптируемость (adaptability).

Внутри классов подходов принято выделять группы подходов с одной и той же моделью ЖЦ и рядом принципов, общих для этих подходов.

К классу строгих подходов относят следующие группы:

1. Каскадные технологические подходы (см. §4.1).
2. Каркасные технологические подходы (см. §4.2).
3. Генетические технологические подходы (см. §4.5).
4. Формальные технологические подходы (см. §4.6).

К классу гибких подходов относят следующие группы:

1. Эволюционные технологические подходы (см. §4.3).
2. Адаптивные технологические подходы (см. §4.4).

Это разделение на группы носит во многом условный характер в связи с обилием существующих и возникающих подходов, которые сочетают возможности (модели и методы, методики и практики) самых разнообразных подходов.

Направления развития подходов

В настоящее время совершенствование технологических подходов ведётся по двум направлениям, которые достаточно сильно отличаются друг от друга своими задачами:

1. Максимизация качества разработки.
2. Максимизация скорости разработки.

Задачи, решаемые на первом направлении — надёжность, чёткость и формализация — ориентированы на военные разработки и системы реального времени. Это направление поддерживается строгими технологическими подходами.

Второе направление в большей степени обращено к искусству, импровизации и поиску, поэтому оно поддерживается гибкими технологическими подходами. Особенность разработки определяется появлением новых классов систем. За последние несколько десятков лет такими классами были, например, системы «клиент — сервер», распределённые системы, интерактивные системы, системы Интернет-приложений.

Ещё одно важное перспективное направление, связанное с технологией разработки ПО — исследование человеческих и социальных факторов в информатике и в частности, в программировании.

Предполагается, что в ближайшее время основные усилия в исследовании разработки ПО будут направлены на процесс проектирования. Это объясняется тем, что в большинстве категорий проектов основной объём работ приходится именно на проектирование, которое должны выполнять талантливые творческие специалисты.

3.3. Модели жизненного цикла ПО

Как было уже отмечено, технологический подход основывается на какой-либо модели ЖЦ ПО и часто сам определяет подходящую модель ЖЦ ПО.

Основными моделями ЖЦ ПО считаются следующие:

— Каскадная модель,

— Итеративная инкрементная модель,

— Эволюционная модель,

— Спиральная модель.

Рассмотрим часто встречающиеся модели ЖЦ ПО. Остальные модели кратко рассмотрены в разделах, посвящённых соответствующим им подходам.

Непланируемая модель

Непланируемая модель (ad-hoc model) или *модель «кодирование — исправление»* (code-and-fix model) является самой простой моделью ЖЦ ПО.

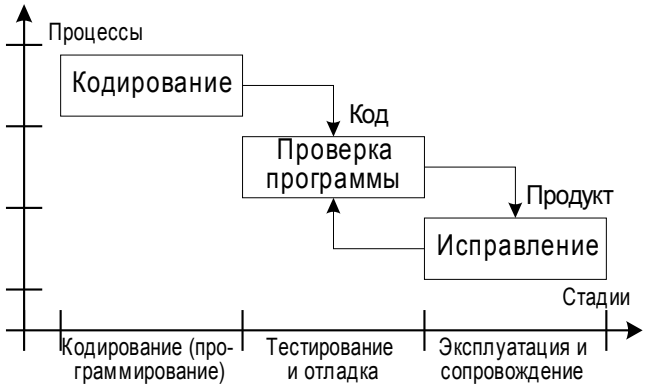


Рис.3.5. Непланируемая модель

Принцип модели (рис.3.5) заключается в написании кода программы без какого-либо серьёзного предварительного анализа требований и проектирования,

запусках программы для проверки его работоспособности и последующем исправлении ошибок и/или добавлении функциональности до получения варианта программы, удовлетворяющего пользователя.

Наиболее часто модель используется в мелких проектах. Она лежит в основе одноимённого подхода, условно относимого к группе гибких подходов (см. §4.3).

Каскадная модель

Классическая каскадная модель (pure stage-wise model) или *Модель водопада* (waterfall model) создана по аналогии с методиками из других инженерных областей, где существует стандартная практика поэтапного создания требуемого продукта от составления спецификаций до поставки заказчику.

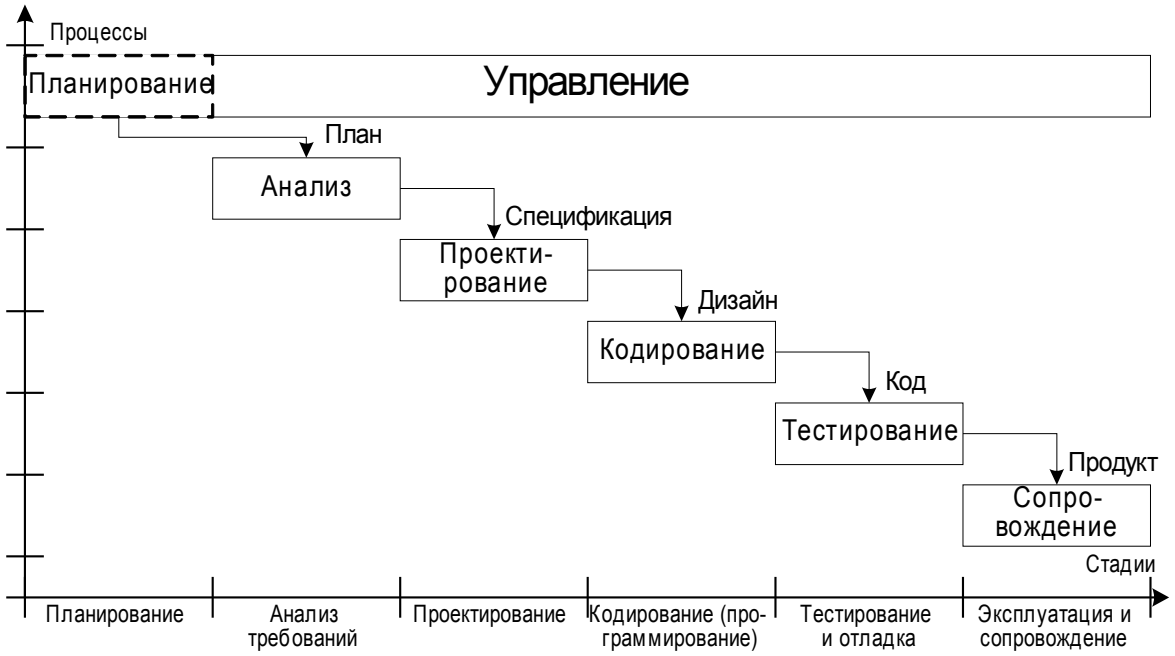


Рис.3.6. Классическая каскадная модель

Принцип модели (рис.3.6) заключается в однократном выполнении процессов в виде заранее ограниченных и однозначно упорядоченных во времени стадий, осуществляемых как бы в их естественных границах.

Все процессы, действия, задачи выполняются чётко последовательно, не допускается никаких перекрытий, параллелизма и возвратов назад. Это жёсткое ограничение делает такую модель практически нереализуемой, так как на уже законченных стадиях не допустимы никакие ошибки, что требует получения артефактов идеального качества.

На практике применяются каскадные модели без учёта этих ограничений.

Модифицированная каскадная модель (modified stage-wise model) или *Модель водоворота* (whirlpool model) является простейшим случаем таких моделей.

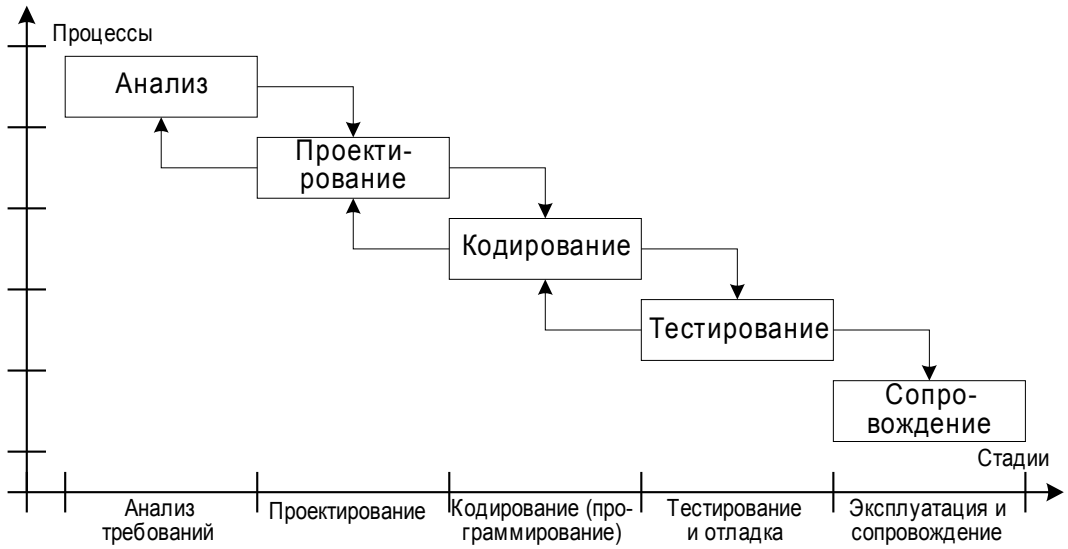


Рис.3.7. Модифицированная каскадная модель

Принцип модели (рис.3.7) заключается в возможности возвращения на предыдущую стадию в случае нахождения ошибки на текущей стадии и пересмотре или уточнении ранее принятых решений.

Классическая каскадная модель сформировалась в период с 1970 по 1980 годы и считается исходной для множества других моделей. Уинстон У. Ройс (Winston W. Royce) в своей статье 1970 г. привёл эту модель как пример нереалистичной нерабочей модели ЖЦ ПО.

Прототипируемая модель

Прототипируемая модель или *Модель прототипирования* (prototyping model, тж. макетная модель или модель макетирования) создана для решения проблем при разработке в условиях неопределённости исходных требований путём разработки прототипов требуемого продукта. Модель требует быстрого построения множества прототипов, поэтому реализация этой модели возможна только при использовании соответствующего инструментария автоматизации.

Классическая модель прототипирования (pure prototyping model) использует разработку прототипов для постепенного выявления всех требований.

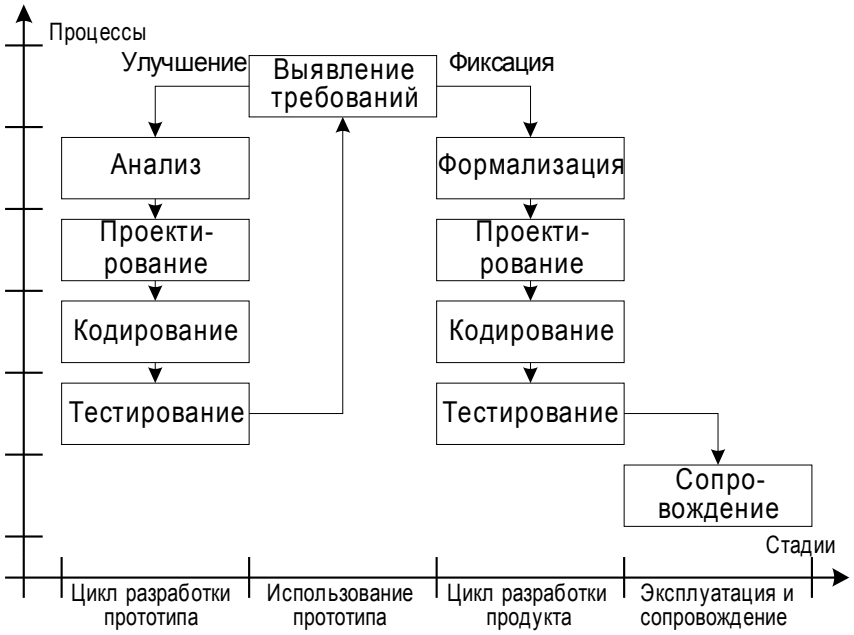


Рис.3.8. Классическая модель прототипирования

Принцип модели (рис.3.8) заключается в первоначальной разработке сильно упрощённого прототипа, который в соответствии с пожеланиями пользователя циклически усовершенствуется и усложняется до тех пор, пока требования к ПО не становятся очевидными. После этого выполняется формализация выявленных требований (составляется спецификация на продукт) и ведётся собственно разработка продукта по какой-либо (например, каскадной) модели. Разработка прототипа и продукта рассматривается как отдельная итерация ЖЦ ПО.

Модификацией классической модели прототипирования является *модель одноразового прототипирования* (throwaway prototyping model). Принцип этой модели заключается в том, что начальный прототип является одноразовым.

Одноразовый прототип (throwaway prototype) — прототип, который создаётся для уточнения и утверждения требований и вариантов дизайна, но при создании продукта не используется.

Вариантами классической модели прототипирования являются итеративная инкрементная модель и эволюционная модель, подробно рассмотренные ниже.

В итеративной инкрементной модели все требования считаются уже известными, а в эволюционной они формулируются постепенно. Это позволяет в первой модели начинать проект с реализации наиболее понятных требований, а во второй — с наиболее сложных.

Принцип разработки прототипированием

Благодаря прототипированию (рис.3.9) реализованные требования к ПО приближаются к целевым требованиям и снижаются неопределённости разработки (усилия, стоимости, сроки) вплоть до получения конечного продукта.

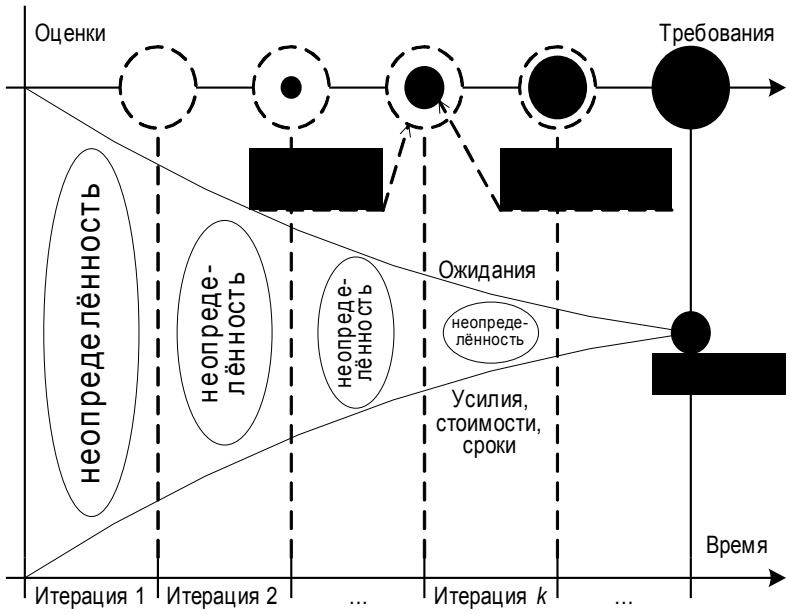


Рис.3.9. Принцип разработки прототипированием

В целом проекты с моделями прототипирования обычно завершаются на 40% раньше аналогичных проектов с каскадными моделями, а получающийся при этом код получается на 45% более коротким. Однако такой код обычно оказывается более запутанным и сложным для сопровождения, а документация на ПО является не столь полной и качественной.

Итеративная инкрементная модель

Итеративная инкрементная модель (iterative and increment model) или Модель запланированного усовершенствования продукта — использует разработку прототипов (выпусков) для последовательной реализации групп требований.

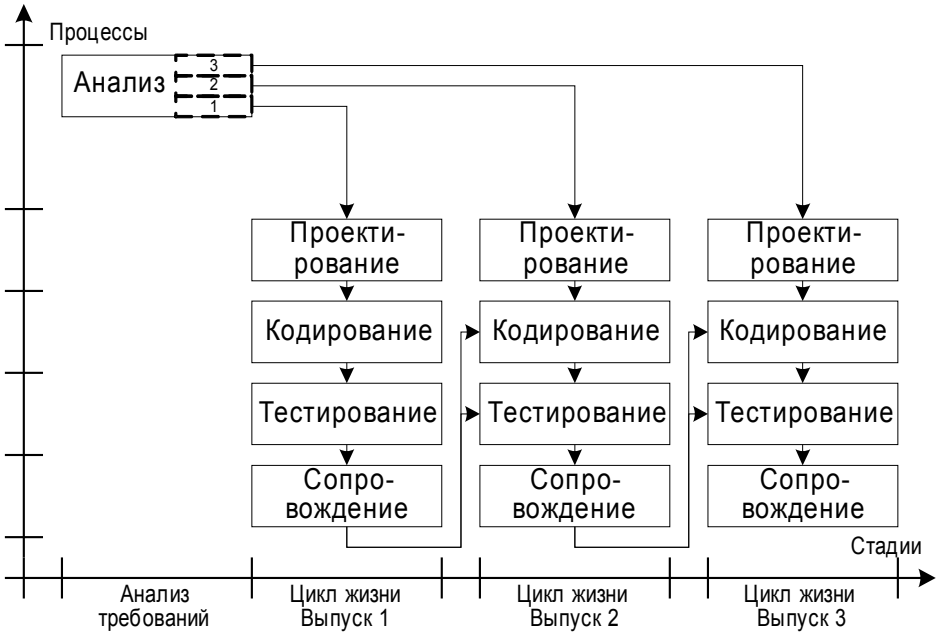


Рис.3.10. Итеративная инкрементная модель

Принцип модели (рис.3.10) заключается в предварительном выделении требований и разработке прототипов, по функциональности всё более приближающихся к продукту. Первый прототип-выпуск (release prototype) основывается на наиболее понятной группе требований, в последующие реализации добавляются всё новые группы требований, пока не будет закончено создание продукта. Для каждого прототипа выполняются необходимые процессы, причём анализ требований и проектирование архитектуры выполняются одновременно, а остальные процессы — индивидуально для каждого прототипа.

Принципы итеративности и инкрементности

Рассматриваемая модель в явном виде включает в своё название два принципа, характерные в том или ином виде для многих моделей прототипирования: итеративность и инкрементность разработки.

Итеративность означает разбиение ЖЦ на последовательность итераций, каждая из которых напоминает мини-проект. Цель каждой итерации — разработка прототипа, результатом последней итерации является продукт.

Инкрементность означает разработку продукта путём постепенного учёта требований к системе. Фактически это также приводит к разработке прототипов, причём последний (часто лишь по срокам) прототип считается продуктом.

Использование этих особенностей в разработке приводит к соответствующим стратегиям планирования (scheduling strategy).

Итеративная разработка (iterative development) — это дорабатывающая стратегия планирования (rework scheduling strategy), при которой отводится время для пересмотра и улучшения частей системы. Альтернатива этой стратегии — каскадная разработка системы на основе «классического водопада» («pure waterfall»).

Инкрементная разработка (increment development) — это конвейерная стратегия планирования (scheduling and staging strategy), при которой различные части системы разрабатываются в разное время или с разной скоростью и после завершения собираются в систему. Альтернатива этой стратегии — интегрированная разработка системы на основе метода «большого взрыва» («big bang»).

Между этими стратегиями есть одно существенное отличие. Результат инкремента не обязательно подлежит дальнейшему уточнению, а его тестирование и отзывы пользователей не используются для пересмотра планов или спецификаций успешных инкрементов. В отличие от этого результат итерации исследуется на предмет модификации и особенно для пересмотра целей успешных итераций.

Тем не менее, опыт разработки ПО показывает, что обе стратегии, хотя и независимы, тесно связаны друг с другом: модель с точки зрения структуры ЖЦ является итеративной, а с точки зрения развития продукта — инкрементной. Поэтому в большинстве работ эту модель именуют просто итеративной или инкрементной, выделяя тем самым только одну точку зрения разработки ПО.

Эволюционная модель

Эволюционная модель (evolution model) использует разработку прототипов (версий) для реализации частично установленных требований при последовательном уточнении и расширении этих требований.

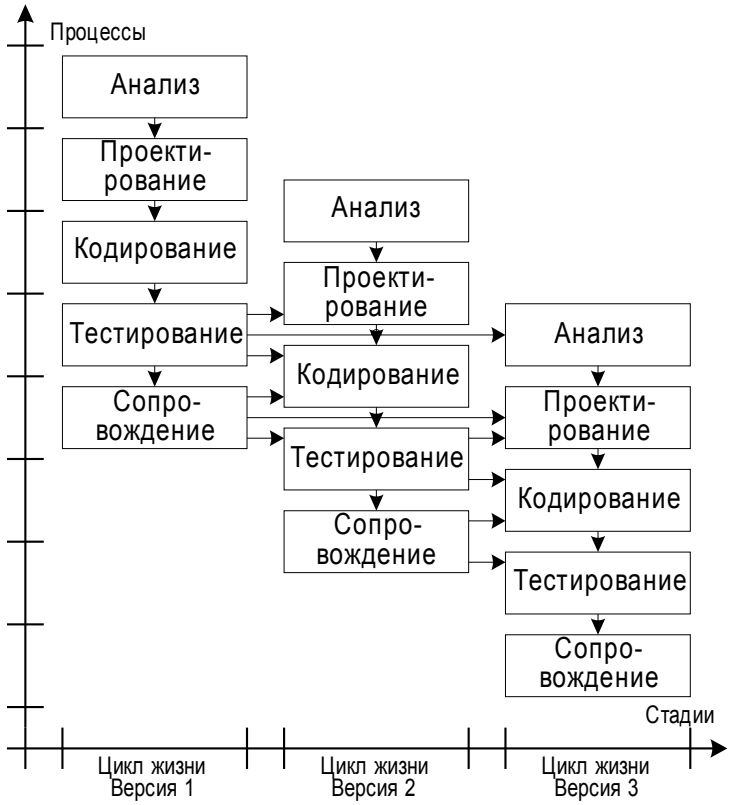


Рис.3.11. Эволюционная модель

Принцип модели (рис.3.11) заключается в постепенной формулировке требований и разработке прототипов, по требованиям всё более приближающихся к продукту. Первый прототип-версия (version prototype) основывается на наиболее сложной и непонятной группе требований, в последующих версиях эти требования уточняются и расширяются с учётом разработки ранних версий.

Принцип эволюционности

Рассматриваемая модель в явном виде включает в своё название принцип, характерный для ряда моделей прототипирования: эволюционность разработки.

Эволюционность (evolutionary) означает разработку продукта путём включения и доработки реализации требований по мере их прояснения.

Спиральная модель

Спиральная модель (spiral model) является результатом анализа и адаптации известных моделей: непланируемой, каскадной и прототипируемой.

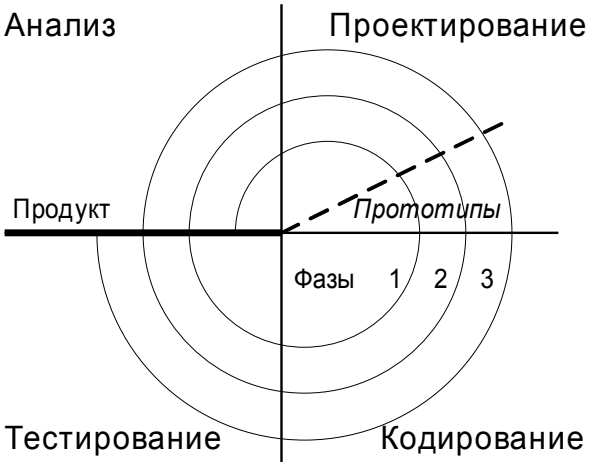


Рис.3.12. Спиральная модель в упрощённом виде

Модель получила своё название из-за графического представления в виде спирали, проходящей через 4 стадии разработки (рис.3.12): анализ, проектирование, кодирование и тестирование. Каждое прохождение этих стадий представляет собой фазу разработки.

Классическая спиральная модель

Классическая спиральная модель (pure spiral model) впервые сформулирована Барри У. Боэмом (Barry W. Boehm) в 1988 г. Поэтому её также называют *модель Боэма* (Boehm's model).

В графическом представлении модели используются полярные координаты (рис.3.13). При этом в заданный момент времени полярный угол соответствует успешности выполняемого проекта (progress through steps — прогрессу по этапам), а полярный радиус, точнее удаление по нему от полюса,— совокупной стоимости разработки (cumulative cost — букв. кумулятивная стоимость).

Отличительной особенностью этой модели является специальное внимание рискам, влияющим на организацию ЖЦ. *Риск* (risk) — это некоторое событие или обстоятельство, препятствующее нормальному достижению цели проекта.

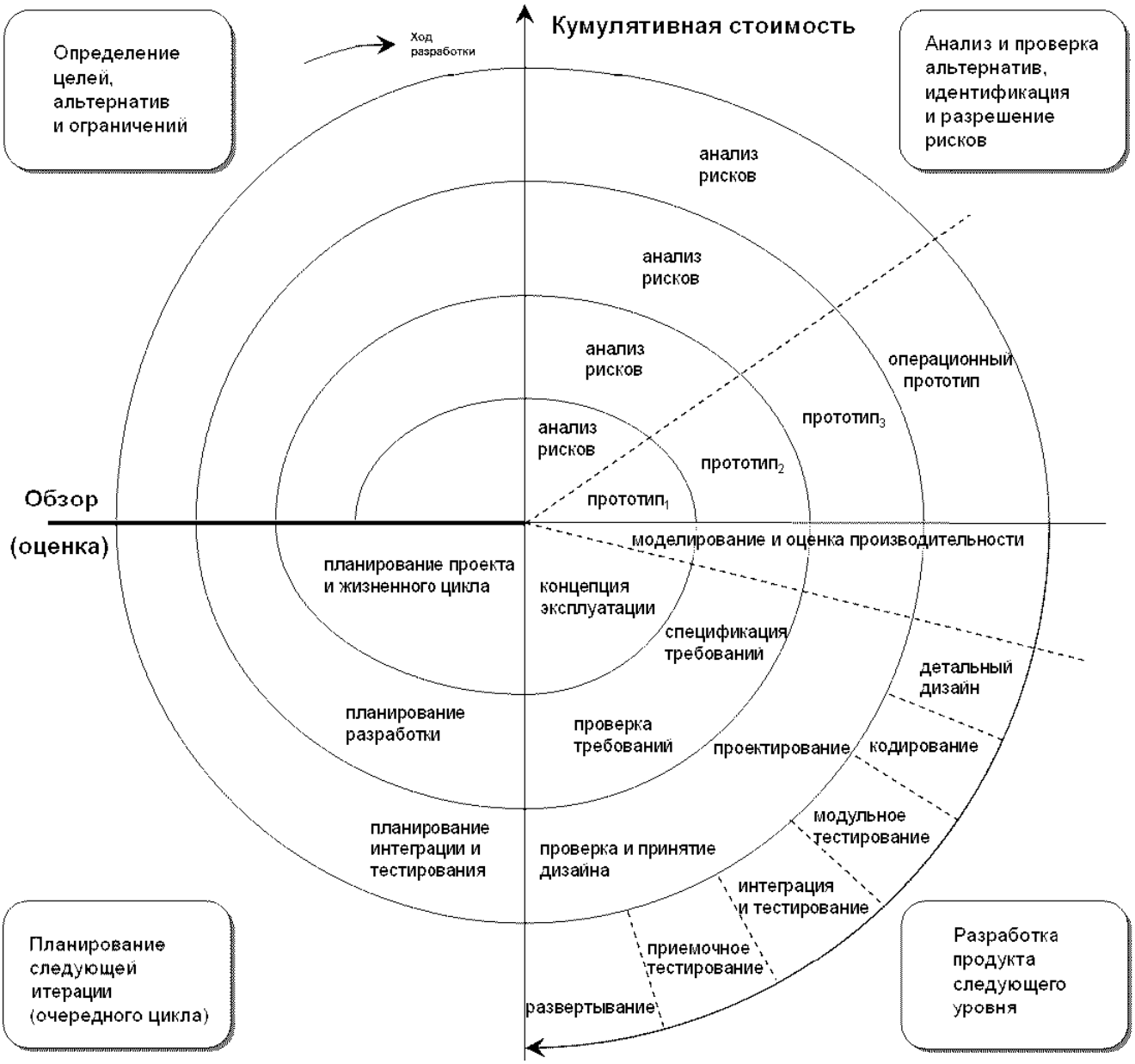


Рис.3.13. Классическая спиральная модель

Боэм выделяет 10 наиболее распространённых (по приоритетам) рисков:

1. Дефицит специалистов.
2. Нереалистичные сроки и бюджет.
3. Реализация несоответствующей функциональности.
4. Разработка неправильного пользовательского интерфейса.
5. «Золотая сервировка» (излишние стремления к эффективности).
6. Непрекращающийся поток изменений.
7. Нехватка информации о внешних компонентах.
8. Недостатки в работах, выполняемых вне проекта.

9. Недостаточная производительность получаемой системы.
10. «Разрыв» в квалификации специалистов разных областей знаний.

Большая часть рисков связана с организационными и процессными аспектами взаимодействия специалистов в команде. Для их идентификации, анализа и разрешения используется План управления рисками (Risk Management Plan).

Принцип модели (рис.3.13) заключается в разработке ПО путём прототипирования за несколько витков спирали, именуемых итерациями, циклами, фазами. Каждая итерация состоит из четырёх стадий:

1. Определение целей, альтернатив и ограничений.
2. Анализ и проверка альтернатив, идентификация и разрешение рисков.
3. Разработка продукта следующего уровня.
4. Планирование следующей итерации (очередного цикла).

Каждый процесс или группа процессов разработки в рамках итерации предваряются анализом рисков (risk analysis) и завершаются проверкой (verification).

В 2000 г. Боэм на основе опыта использования спиральной модели сформулировал 6 ключевых практик, обеспечивающих успешное её применение:

1. Параллельное, а не последовательное определение артефактов проекта.
2. Концентрация внимания в каждом цикле на следующих аспектах:
   —поставленным целям и ограничениям,
   —альтернативам организации процесса и технологических решений, закладываемых в продукт,
   —идентификации и разрешению рисков,
   —оценки со стороны заинтересованных лиц,
   —достижению согласия в том, что можно и необходимо двигаться дальше.
3. Использование соображений, связанных с рисками, для определения уровня усилий, необходимого для каждой работы на всех циклах спирали.
4. Использование соображений, связанных с рисками, для определения уровня детализации каждого артефакта, создаваемого на всех циклах спирали.

5. Управление жизненным циклом в контексте обязательств всех заинтересованных лиц на основе трёх контрольных точек:
   1. Цели ЖЦ — Life Cycle Objectives (LCO);
   2. Архитектура ЖЦ — Life Cycle Architecture (LCA);
   3. Начальный операционный вариант — Initial Operational Capability (IOC).
6. Уделение специального внимания проектным работам и артефактам создаваемого ПО (включая само разрабатываемое ПО, её окружение, а также эксплуатационные характеристики) и ЖЦ (разработки и использования).

Классическая спиральная модель носит обобщённый характер и требует конкретизации для получения пригодной на практике модели. Особенно это касается детализации её компонентов (итераций, стадий, процессов, практик и т.п.).

Модифицированная спиральная модель

Модифицированная спиральная модель (modified spiral model) представляет собой один из промежуточных вариантов по уровню детализации.

Детализация в этой модели (рис.3.14) связана с уточнением некоторых процессов, увеличением количества итераций при сокращении их длительности и соответствующим определением контрольных точек.

Данная модель содержит следующий общий набор контрольных точек:

1. Концепция эксплуатации — Concept of Operations (COO);
2. Цели ЖЦ — Life Cycle Objectives (LCO), включая содержание ЖЦ;
3. Архитектура ЖЦ — Life Cycle Architecture (LCA), здесь же можно говорить о готовности концептуальной архитектуры целевого ПО;
4. Начальный операционный вариант — Initial Operational Capability (IOC) — вариант ПС, готовый для опытной эксплуатации;
5. Конечный операционный вариант — Final Operational Capability (FOC) — вариант ПО в виде продукта, готового для реальной эксплуатации.

Последняя контрольная точка в ряде подходов на основе этой модели называется по-другому: Выпускаемый продукт — Product release (PR).

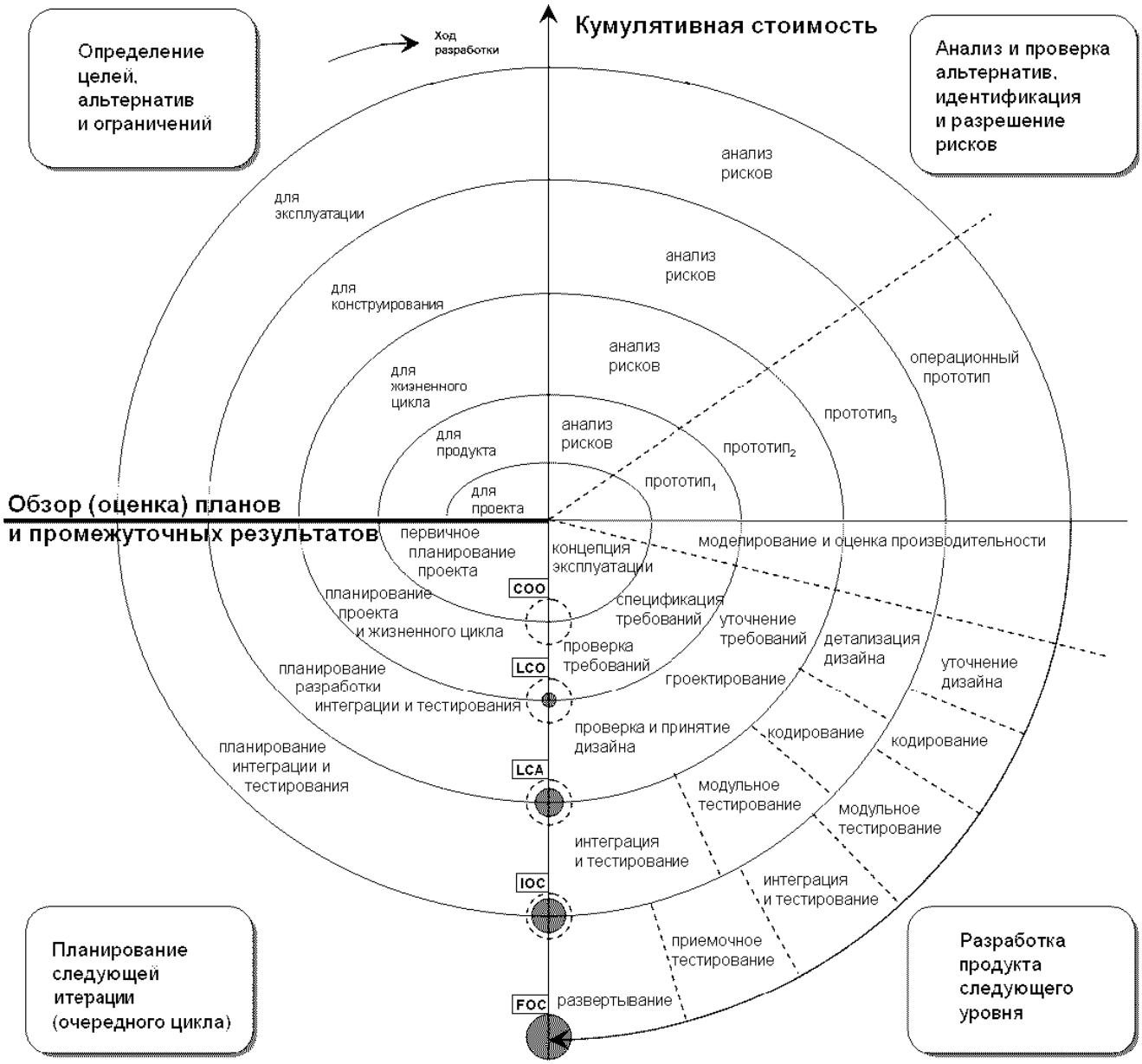


Рис.3.14. Модифицированная спиральная модель

Фактически получается эволюционный ЖЦ в форме спиральной модели.

Другие модели ЖЦ

Кроме приведённых моделей существуют и создаются другие модели. Наиболее известными в настоящее время являются модель V (V-model) и модель W (W-model), представляющие собой разновидности каскадной и прототипируемой моделей с учётом новых процессов и дополнительных связей.

Ряд специфических моделей рассматривается в разделе 4 при описании соответствующих технологических подходов.

3.4. Классические технологические процессы

Рассмотрим кратко 9 классических процессов.

Процесс 1. Исследование идеи

Исследование идеи — процесс ЖЦ ПО, который заключается в превращении возникшей идеи в определённую концепцию и в формировании проекта.

Этот классический процесс имеет следующие действия:

1. Возникновение (genesis) и первичное исследование идеи, которое носит максимально творческий и неформальный характер.
2. Детальное исследование идеи. Выработка концепции (concept). Постановка задачи. Создание «одностраничного описания проекта» и разработка его расширенной версии.
3. Экспертиза идеи специалистами. Принятие решения о начале планирования.

Следует отметить, что возникшая идея может привести либо к развитию (enhancement — расширение) уже существующей системы, либо к созданию (development — разработка) новой системы.

Аналогичным образом выглядит этот процесс и в случае получения заказа на разработку некоторого продукта.

Формальным результатом исследования идеи является одностраничное описание проекта.

Процесс 2. Управление

Управление проектом — процесс ЖЦ ПО, который заключается в принятии решений по правильной организации имеющихся ресурсов проекта в рамках поставленных ограничений для получения продукта, удовлетворяющего потребности пользователя и требования заказчика.

Этот классический процесс выполняется почти во время всего ЖЦ ПО, но он указывается как процесс 2 потому, что одним из важнейших действий управления является планирование, начинающее собственно разработку после процесса 1.

Данный процесс изучается специальной дисциплиной, называемой *управление проектами* (PM, Project Management — букв. проектный менеджмент). Поэтому здесь рассмотрим только краткие положения по управлению проектом.

Для проекта управление определяет, как, с помощью каких действий, будет достигнута цель проекта и создан необходимый результат. *Цель проекта* (project goal) описывает, какие задачи должны быть решены в результате проекта, а *содержание проекта* (project scope) — что именно является результатом проекта. Цель проекта разделяется на *целевые установки* или [*отдельные*] *цели* (objective, тж. задача) для распределения работ по времени и участникам разработки.

Получение результата проекта и достижение цели проекта — не одно и то же. Проект можно считать успешным при условии, что результат проекта соответствует заданному содержанию проекта и его цели. Степень достижимости цели проекта зависит от адекватности заданного содержания проекта его цели. Для обеспечения адекватности проекта необходимо *единое видение* (shared vision) проекта — ясное единообразное представление цели, установок и содержания проекта всеми заинтересованными лицами.

Формальным результатом планирования является план (plan) проекта, в том числе календарный план (schedule) проекта.

Процесс 3. Анализ

Анализ требований — процесс ЖЦ ПО, который заключается в уточнении, формализации и документировании требований заказчика. Основной вопрос, который решается здесь — «ЧТО должен делать будущий продукт?»

В этом классическом процессе наиболее важным является понимание понятия «требование». Существует несколько точек зрения на понятие «требование».

Требование (requirement) — условие или возможность, необходимая для решения проблемы или достижения определённых целей с помощью разрабатываемого продукта (с точки зрения бизнеса, пользователя и т.п.).

Требование к продукту (product requirement) — условие или возможность, которую должен удовлетворять или которой должен обладать продукт или его компонент для обеспечения условий разработки, связанных с контрактом, стандартами, спецификациями. Аналогично формулируется *требование к процессу* (process requirement) ЖЦ.

Спецификация требований является результатом формализации требований. В общем случае *спецификация* (specification) — достаточно полное и точное формальное описание работы, которую необходимо выполнить.

Спецификация требований (requirements specification) — это спецификация, включающая однозначно интерпретируемые требования, реализация которых проверяема, а стоимость и ресурсы — предсказуемы.

Существуют две существенно отличающиеся части спецификаций, соответствующие предъявляемым к ПО требованиям:

1. *Функциональные спецификации* (functional specifications) задают содержание функционирования системы. Они описывают функции ПО на основе предъявляемых к системе требований заинтересованных лиц.
2. *Эксплуатационные (нефункциональные) спецификации* (nonfunctional specifications) задают характеристики системы (надёжность, производительность, масштабируемость и др.) и ограничения её функционирования. Они описывают особенности ПО на основе правил и стандартов.

Анализ требований также включает *концептуальное моделирование* (conceptual modeling). Разработка модели ПрО — ключевой элемент этого процесса. Цель моделирования — понимание проблемы, задачи и методов их решения до того, как начнётся собственно решение.

Формальным результатом анализа является спецификация требований и концептуальная модель ПрО.

Процесс 4. Проектирование

Проектирование — процесс ЖЦ ПО, который заключается в исследовании структуры ПО и взаимосвязи его компонентов. Основной вопрос, который решается здесь — «КАК продукт будет удовлетворять полученным требованиям?».

В этом классическом процессе наиболее важным является представление разрабатываемого ПО (как единого целого) в виде системы, рассмотренное в §1.1.

Проектирование обычно разделяется на два взаимосвязанных подпроцесса:

1. *Проектирование архитектуры* (проектирование структуры, проектирование «в большом», architectural design — архитектурное проектирование, top-level design — высокоуровневое проектирование).
2. *Проектирование компонентов* (проектирование модулей, проектирование «в малом», detailed design — детализированное проектирование, bottom-level design — низкоуровневое проектирование).

Проектирование архитектуры заключается в декомпозиции структуры системы и организации её компонентов. Проектирование компонентов описывает специфическое поведение и характеристики отдельных компонентов системы для их кодирования.

В некоторых случаях выделяют связующий промежуточный подпроцесс:

3. *Проектирование взаимодействия* (проектирование управления, проектирование «в среднем», проектирование связи, mechanistic design — механистическое проектирование, middle-level design — среднеуровневое проектирование).

Проектирование взаимодействия определяет организацию взаимодействия компонентов системы, выделенных при проектировании архитектуры и детализируемых при проектировании компонентов.

Архитектура ПО (software architecture) — представление ПО как системы, состоящей из совокупности взаимодействующих частей. Компонент является относительно самостоятельной частью системы, в которой можно выделять только взаимосвязанные элементы.

Дизайн представляет собой целостный взгляд на архитектуру ПО и состоит из различных точек зрения (viewpoint). *Архитектурное представление* (architectural view) — это архитектура ПО с определённой точки зрения. *Архитектурная структура* (architectural structure) — дальнейшая детализация ПО, необходимая для реализации ПО и удовлетворения требований, предъявляемых к ПО. *Дизайн ПО* (design) — комплекс архитектурных представлений и архитектурных структур.

Формальным результатом проектирования являются дизайн продукта.

Процесс 5. Кодирование

Кодирование — процесс ЖЦ ПО, который заключается в написании программного кода разрабатываемого ПО.

Однако на практике такое определение оказывается слишком узким для этого классического процесса. Поэтому в настоящее время используется понятие «конструирование», определяемое следующим образом:

Конструирование — процесс ЖЦ ПО, который заключается в создании программного кода разрабатываемого ПО посредством комбинации дальнейшей детализации дизайна, кодирования и необходимого тестирования. В результате этот процесс оказывается наиболее связанным со смежными процессами — проектированием и тестированием.

К основным концепциям конструирования относят:

— Минимизация сложности: создание простого и легко читаемого кода.
— Ожидание изменений: создание легко адаптируемого кода.
— Конструирование с проверкой: создание легко тестируемого кода.
— Стандарты в конструировании: следование стандартам при создании кода.

Первые три концепции применяются и к проектированию, фактически они лежат в основе современных технологических подходов разработки ПО.

Поддержка этих концепций осуществляется заданием стиля программирования, единого для всего создаваемого программного кода проекта, и использованием методов защитного программирования (подробнее см. §5.1).

Конструирование можно рассматривать как привязку дизайна продукта к конкретному языку конструирования. *Язык конструирования* (ЯК, construction language) — это форма представления решения проблемы для её выполнения на компьютере.

Выделяют следующие типы ЯК:

1. Конфигурационный язык (configuration language) — простейший тип ЯК: позволяет задавать параметры выполнения ПО.
2. Инструментальный язык (toolkit language) — тип ЯК из повторно используемых элементов: обычно строится как сценарный язык (script), выполняемый в соответствующей среде.
3. Язык программирования (ЯП, PL — programming language) — наиболее гибкий тип ЯК: содержит минимальный объём информации о конкретных областях приложения и процессе разработки, требуя больше всего (по сравнению с другими типами ЯК) усилий на изучение и наработку опыта для эффективного применения при решении конкретных задач.

На выбор ЯК влияют 4 основных фактора.

1. Сравнительная пригодность ЯК для данной задачи.
2. Выбранная методология разработки ПО. Часто говорят, что некоторый язык поддерживает ту или иную методологию. Обычно это означает, что применение этого языка совместно с указанной методологией в совокупности дадут значительно больший эффект.
3. Степень важности для разработчика многочисленных характеристик и свойств, которые могут быть присущи или не присущи избираемому ЯК.
4. Степень знакомства программистов с ЯК. Результаты исследований говорят о том, что производительность программиста, работавшего на некотором языке более трёх лет, возрастает на треть по сравнению с программистом такого же уровня, но без опыта работы на данном языке.

Большинство вопросов, связанных с выполнением процесса кодирования (конструирования), относится к инженерии ПО.

Формальным результатом конструирования являются программный код.

Процесс 6. Тестирование

Тестирование — процесс ЖЦ ПО, который заключается в оценке и улучшении качества ПО. В общем случае тестирование состоит в обнаружении проблем нарушения работы ПО.

Этот классический процесс представляет собой динамическую верификацию поведения программы на ограниченном наборе тестов, выбранных соответствующим образом из обычно выполняемых действий ПрО и обеспечивающих проверку соответствия ожидаемому поведению системы.

Для описания нарушений работы ПО используется множество терминов.

Дефект (defect) — самое общее нарушение каких-либо требований или ожиданий, связанных с ПО. *Недостаток* (fault) — проявление дефекта в ПО: некорректный шаг, процесс или определение данных в программе.

Примером дефекта является сбой. *Сбой* (failure, тж. отказ) — наблюдаемое нарушение: некорректный результат, полученный из-за наличия недостатка.

Ошибка (error) — в зависимости от контекста — сбой или недостаток: отличие между корректным результатом и вычисленным с помощью ПО результатом или местоположение дефекта в ПО, приведшее к сбою. *Человеческая ошибка* (mistake) — действие человека, приведшее к некорректному результату.

При разработке с использованием формальных подходов вместо тестирования выполняют инспектирование — статическую верификацию поведения программы, которое не исключает применения статистического тестирования (подробнее см. §4.6, «Подходы формальной разработки»).

В настоящее время считается, что верификацию любого вида (тестирование / инспектирование) необходимо выполнять во время всего ЖЦ ПО.

Большинство вопросов, связанных с выполнением процесса верификации, относится к инженерии ПО.

Формальным результатом тестирования являются «оттестированный» программный код, т.е. код с исключением недостатков, выявленных по сбоям.

Процесс 7. Ввод в действие

Ввод в действие — процесс ЖЦ ПО, который заключается в передаче разработанного ПО в эксплуатацию.

Этот классический процесс существенно зависит от вида заказчика ПО и включает обычно следующие действия:

1. Распространение (доставка заказчику) продукта.
2. Инсталляция (установка на конкретные системы) продукта.

Процесс 8. Сопровождение

Сопровождение — процесс ЖЦ ПО, который заключается модификации ПО после ввода его в действие для улучшения качества и дальнейшего совершенствования. В общем случае сопровождение состоит в обеспечении эффективной поддержки эксплуатируемого ПО при сохранении его целостности.

Основным принципом этого классического процесса является *закон Биледи* (Laszlo Belady): используемый программный продукт подвергается непрерывным изменениям для поддержания его экономической выгоды. На практике именно недооценка сопровождения приводит к снижению эффективности ПО и отдачи от проекта. В настоящее время считается, что этот процесс по сути является *продолжающейся разработкой* (continued development).

Выделяют 4 категории сопровождения, рассматриваемых с точки зрения двух подходов и разбиваемых на две группы (табл.3.2).

Табл. 3.2. Категории сопровождения

Подход	Коррекция	Расширение
Проактивный	Профилактика	Совершенствование
Реактивный	Корректировка	Адаптация

Таким образом, получаем следующие категории сопровождения:

1. Корректирующее сопровождение (corrective) — модификация для исправления обнаруженных дефектов: устранение сбоев и т.д.
2. Адаптивное сопровождение (adaptive) — модификация для учёта требуемых изменений: учёт новых требований, изменение окружения ПО и т.д.

3. Совершенствующее сопровождение (perfective) — модификация для улучшения возможностей ПО: повышение характеристик ПО и т.д.
4. Профилактическое сопровождение (preventive) — модификация для предупреждения возможных проблем: определение и исправление скрытых дефектов до их реального появления в виде сбоев и т.д.

Подходы к сопровождению ПО на практике бывают двух видов:

1. Проактивный подход (proactive) заключается в «упреждающих» модификациях ПО, соответствующих профилактике и совершенствованию.
2. Реактивный подход (reactive) заключается в «реагирующих» модификациях ПО, соответствующих корректировке и адаптации.

Категории сопровождения можно разбить на две группы:

1. Коррекция» (correction) — модификация, соответствующая профилактике или корректировке.
2. Расширение» (enhancement) — модификация, соответствующая совершенствованию или адаптации.

Эти группы соответствуют разным уровням модификации ПО:

1. Ревизия (revision — исправление, пересмотр) — незначительные, часто локальные, изменения кода.
2. Реструктурирование (restructuring — реструктуризация) — повторная разработка небольшой части кода обычно без изменения интерфейса.
3. Реорганизация (reengineering — реинжиниринг) — перестройка существующего кода обычно в соответствии с новой моделью или методологией.
4. Реконструирование (reconstruction — реконструкция, повторное конструирование) — перекодирование существенной части кода.

Коррекция предполагает ревизию и реструктурирование, а расширение — реорганизацию и реконструирование. Кроме того, необходимо учитывать класс решаемой задачи и стоимость сопровождения.

Процесс 9. Снятие с эксплуатации

Снятие с эксплуатации — процесс ЖЦ ПО, который заключается в прекращении сопровождения эксплуатируемого ПО.

Этот классический процесс выполняется, когда невозможно выполнить ни одну из категорий сопровождения. Он осуществляется предварительным оповещением пользователей о прекращении сопровождения. Однако дальнейшее использование ПО возможно вплоть до его морального устаревания.

3.5. Методики анализа и проектирования

В настоящее время наиболее употребительными при разработке ПО являются две методологии — структурная и объектно-ориентированная. Принципиальное различие между ними заключается в разных способах декомпозиции систем:

1. *Структурная (процедурная, функциональная) декомпозиция* рассматривает структуру и поведение системы в терминах иерархии подпрограмм (процедур, функций) и передачи информации.
2. *Объектная декомпозиция* рассматривает структуру системы в виде объектов и связей между ними, а поведение системы — в терминах обмена сообщениями между объектами.

Следует отметить, что в основе многих объектно-ориентированных методов лежит структурный метод, которому придана объектная окраска.

Анализ и проектирование — два достаточно близких и тесно связанных процесса. Они выполняют общую задачу, результатом которой должно стать чёткое представление о системе, на основе которого будет создан программный код.

В основе анализа и проектирования лежат модели и методы, предназначенные для формализации требований. Модели включают в себя нотацию и словарь.

Нотация (notation) представляет собой соглашение о представлении: текстовом (словесном, формальном: описание) и/или графическом (модельном: схемы, графики, диаграммы).

Словарь (dictionary) содержит перечень используемых терминов с указанием их значений или толкований.

Модели и методы, объединённые в некоторую комбинацию, образуют *методики* или *методические подходы* к анализу и проектированию. Подходы имеют названия, причём большинство из них названы по именам своих авторов. Большинство методов и подходов применимы как при анализе, так и при проектировании.

Модели и методы анализа требований

Структурная методология. Основные модели и методы:

— Диаграммы потоков данных (ДПД, DFD — Data Flow Diagram).
— Диаграммы потоков управления (ДПУ, CFD — Control Flow Diagram).
— Таблицы / деревья решений (Decision Table / Tree).
— Сети Петри (Petri Net).
— Диаграммы зависимостей (Dependency Diagram).
— Диаграммы декомпозиции (Decomposition Diagram).
— Диаграммы функционального моделирования (SADT Diagram).

Объектно-ориентированная методология. Основные модели и методы:

— КОК-карты (Класс — Ответственность — Кооперация, CRC — Class — Responsibility — Collaboration).
— Диаграммы прецедентов (Use Case Diagram).
— Диаграммы классов и объектов (Class Diagram, Object Diagram).
— Диаграммы состояний (State Diagram).
— Диаграммы деятельности (Activity Diagram).
— Диаграммы последовательности (Sequence Diagram).

Модели и методы проектирования архитектуры

Структурная методология. Основные модели и методы:

— Нисходящее проектирование (Top-Down Design).
— Восходящее проектирование (Bottom-Up Design).

Объектно-ориентированная методология. Основные модели и методы:

— Проектирование предметных областей (Application Domain Design).
— Проектирование наведением мостов (Bridge Design).

Модели и методы проектирования компонентов

Структурная методология. Основные модели и методы:

— Диаграммы «сущность — связь» (ERD — Entity — Relationship Diagram).
— Структурные карты (Structure Card).
— Скобочные диаграммы (Assembly Line Diagram — Сборочно-конвейерная диаграмма) Варнье — Орра.
— Диаграммы деятельности (Activity Diagram).
— Диаграммы переходов состояний (STD — State Transition Diagram).
— Блок-схемы, структурные схемы (Flow Diagram).
— Схемы экранов (Screen chart).
— Псевдокод (Pseudocode).
— Блок-схемы, потоковые схемы (Flowchart).
— Диаграммы Несси — Шнейдермана (Nassi — Shneiderman Diagram).

Объектно-ориентированная методология. Основные модели и методы:

— Диаграммы кооперации (Collaboration Diagram).
— Диаграммы компонентов (Component Diagram).
— Диаграммы развёртывания (Deployment Diagram).

Подходы (методики) к анализу и проектированию

Структурная методология. Основные подходы:

— Подход Йордона / ДеМарко (Edward Yourdon and Tom DeMarco, SAD — Structured Analysis and Design — Структурированный анализ и проектирование).
— Подход Гейна — Сарсона (Chris Gane and Tom Sarson, SSA — Structured System Analysis — Структурный системный анализ).
— Подход Константайна (L.L. Constantine, SSD — Structure System Design — Структурное системное проектирование).
— Подход Джексона (Michael Jackson, 1975 г., JSD — Jackson Structured Development — Структурная разработка Джексона).

— Подход Варнье — Орра (Warnier and Ken Orr, 1974 г., DSSD — Data-Structured System Development — Разработка систем, структурированных данными).
— Подход Мартина (James Martin, 1970—1980 гг., IE — Information Engineering — Информационная инженерия).
— Подход структурированного анализа и проектирования (Douglas T. Ross, 1973 г., SADT — Structured Analysis and Design Technique — Методика структурированного анализа и проектирования).
— Подход промышленной технологии DATARUN.
— Подход промышленного метода Oracle (в рамках собственной технологии).

Объектно-ориентированная методология. Основные подходы:

— Подход на основе языка UML (Unified Modeling Language — Универсальный язык моделирования, УЯМ) — подход, основанный на первых трёх из перечисленных ниже подходов с использованием ряда методик из других подходов.
— Подход Гради Буча (Grady Booch, Booch).
— Подход Джеймса Рамбо (James Rumbaugh, OMT — Object Modeling Technique — Методика объектного моделирования).
— Подход Айвара Якобсона (Ivar Jacobson, OOSE — Object-Oriented Software Engineering — Объектно-ориентированная инженерия ПО).
— Подход Шлеер — Меллора (Sally Shlaer and Stephen Mellor, Recursive Design — Рекурсивное проектирование).

3.6. Стандартные технологические процессы

Обязательная реализация в ходе проекта типовых процессов ЖЦ даёт возможность использовать принципы и методы стандартизации, основанные на применении базовых стандартов и разработанных на их основе профилей стандартов для конкретного типа объекта (проекта, ПО или системы).

Базовый стандарт — принятый нормативный документ, регламентирующий типовые (возможно многовариантные) требования, нормы и правила применительно к данному объекту стандартизации.

Профиль стандарта — принятый нормативный документ, регламентирующий требования, нормы и правила, выбранные из базовых стандартов и при необходимости дополненные и/или уточнённые (ограниченные) применительно к конкретной классификационной группе данного объекта стандартизации.

Стандарт ISO/IEC 12207

Одним из фундаментальных взглядов на ЖЦ является стандарт процессов ЖЦ ISO/IEC 12207:1995 «Information Technology — Software Life Cycle Processes» («Информационная технология — Процессы жизненного цикла ПО»).

Цель разработки стандарта определена как создание общего каркаса (framework — фреймворк) по организации ЖЦ ПО для формирования общего понимания ЖЦ ПО всеми заинтересованными лицами, а также для возможности управления, контроля и совершенствования процессов ЖЦ. Стандарт ориентирован для применения всеми участниками проекта в рамках соответствующего ЖЦ ПО.

Стандарт определяет ЖЦ как *структуру дробления работ* (СДР, WBS — work breakdown structure). Детализация, техники и метрики проведения работ — вопрос инженерии ПО. Организация последовательности работ определяется моделью ЖЦ. Совокупность моделей, процессов, техник и организации проектной команды задаются технологией на основе методологии.

Таким образом, стандарт определяет высокоуровневую архитектуру ЖЦ ПО.

Архитектура ЖЦ ПО

ЖЦ ПО в рамках стандарта представляет собой структуру дробления работ на следующие элементы: процессы, действия и задачи. Она реализована на основе регламентации требований (в виде целей и результатов) к этим элементам, входящим в полную типовую структуру ЖЦ ПО (рис.3.15).

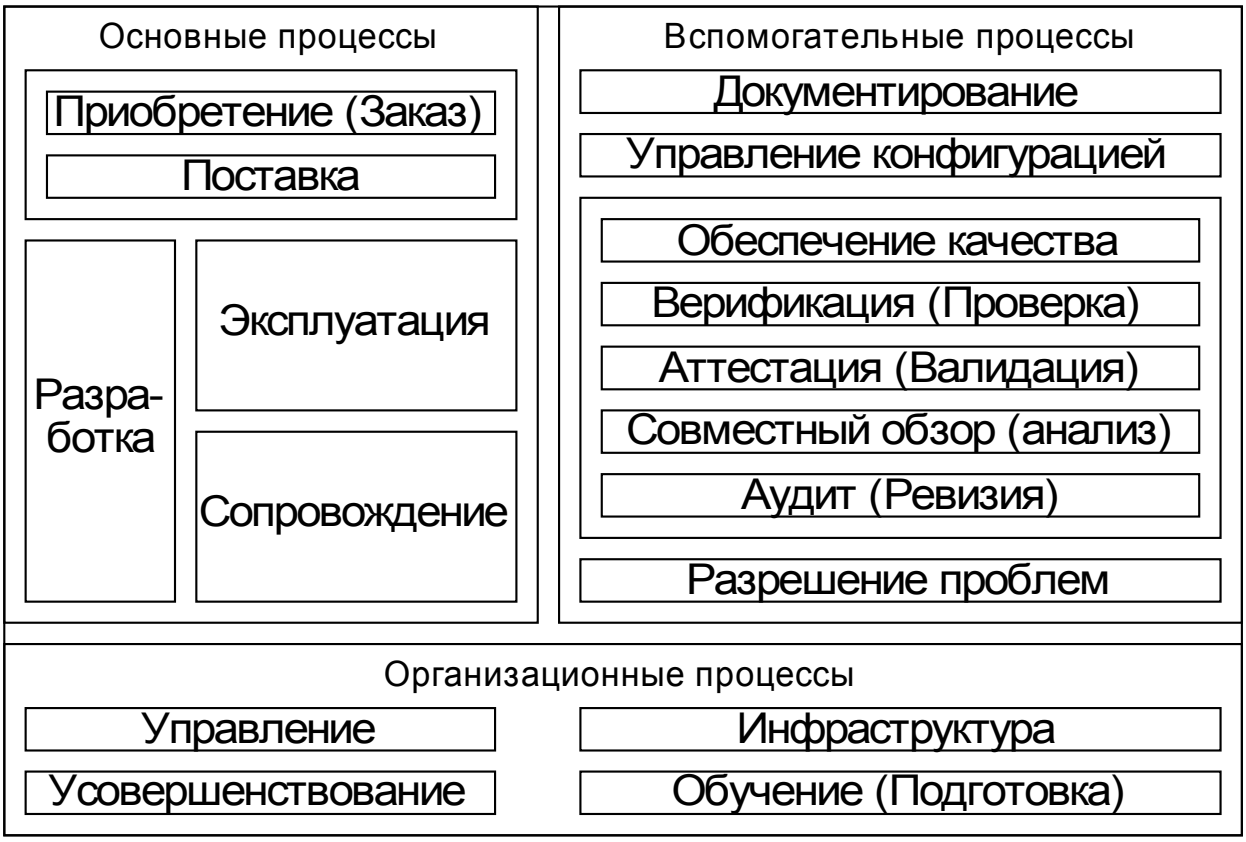


Рис.3.15. Общая структура стандартных процессов

Выделение процессов строится на основе двух важнейших принципов:

1. *Модульность* (Modularity):

—Задачи в процессе являются функционально связанными.

—Связь между процессами минимальна.

—Если функция используется более чем одним процессом, она сама является процессом.

—Если процесс Y используется процессом X и только им, значит, процесс Y принадлежит (является его частью или его задачей) процессу X, за исключением случаев потенциального использования процесса Y в других процессах в будущем.

2. *Ответственность* (Responsibility):

—Каждый процесс находится под ответственностью конкретного лица (управляется и/или контролируется им), определённого для заданного ЖЦ, например, в виде роли в проектной команде.

—Функция, чьи части находятся в компетенции различных лиц, не может рассматриваться как самостоятельный процесс.

Каждый процесс состоит из набора действий и решаемых при выполнении соответствующего действия задач. С точки зрения соподчинённости и важности все процессы разбиты на 3 группы: основные, вспомогательные и организационные (рис.3.15). Таким образом, общая иерархия элементов ЖЦ выглядит следующим образом: Группа процессов → Процессы → Действия → Задачи.

В общем случае разбиение процесса базируется на широко распространённом PDCA-цикле:

— P — Plan — Планирование.

— D — Do — Выполнение.

— C — Check — Проверка.

— A — Act — Действие (исправление планов).

Стандарт предлагает 5 точек зрения на процессы, соответствующих основным заинтересованным лицам (рис.3.16):

1. Заказчики и поставщики имеют договорную (контрактную) точку зрения.
2. Операторы службы поддержки и пользователи имеют эксплуатационную точку зрения.
3. Разработчики системы и специалисты по сопровождению имеют инженерную точку зрения.
4. Исполнители вспомогательных процессов имеют точку зрения поддержки.
5. Менеджеры имеют управленческую точку зрения.

Таким образом, архитектура строится как набор процессов и взаимных связей между ними. Так, основные процессы обращаются к вспомогательным процессам, а организационные процессы действуют на всём протяжении ЖЦ и связаны с основными процессами (рис.3.16).

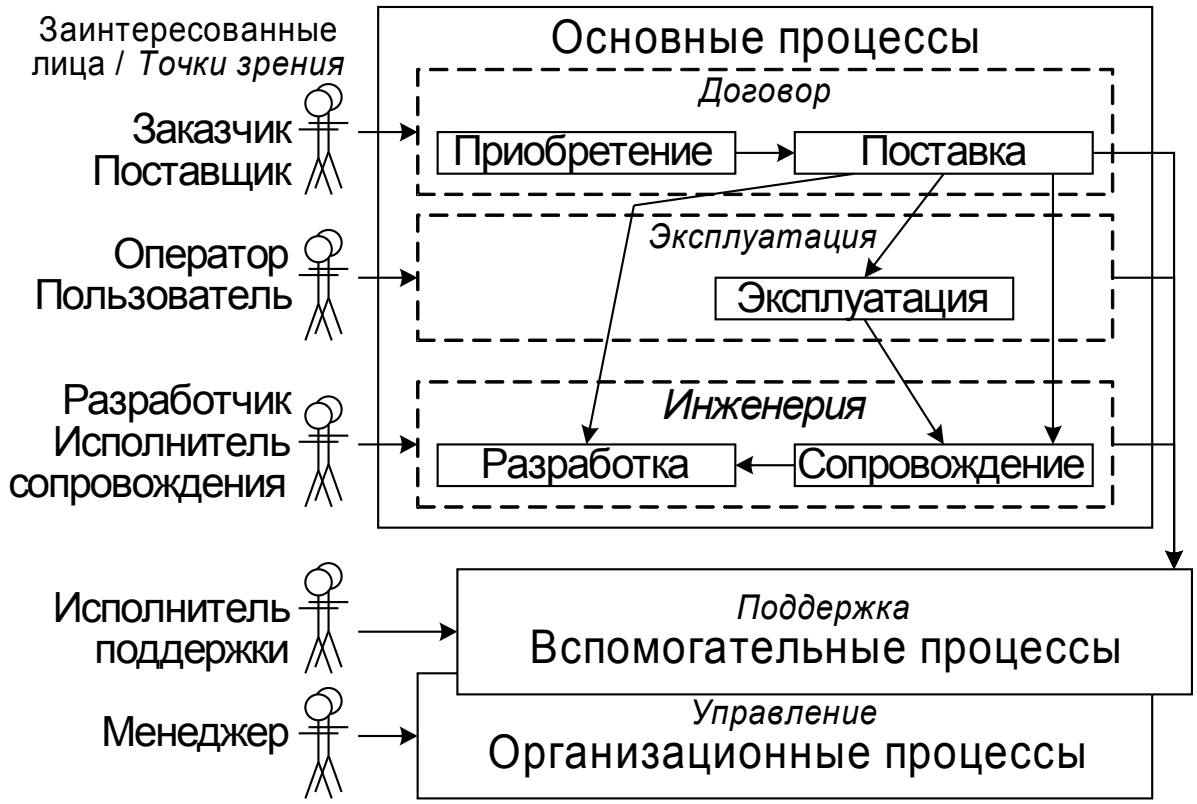


Рис.3.16. Взаимосвязь между стандартными процессами

Стандарт описывает следующие 17 процессов ЖЦ ПО в 3 группах (рис.3.15):

1. *Основные процессы* (Primary Processes — первичные процессы):
   1. Приобретение / Заказ (Acquisition).
   2. Поставка (Supply).
   3. Разработка (Development).
   4. Эксплуатация (Operation).
   5. Сопровождение (Maintenance).
2. *Вспомогательные процессы* (Supporting Processes — процессы поддержки):
   1. Документирование (Documentation).
   2. Управление конфигурацией (Configuration Management).
   3. Обеспечение качества (Quality Assurance).
   4. Верификация / Проверка (Verification).
   5. Аттестация / Валидация (Validation).
   6. Совместный обзор (Joint Review).
   7. Аудит / Ревизия (Audit).
   8. Разрешение проблем (Problem Resolution).

3. *Организационные процессы* (Organizational Processes):
   1. Управление (Management).
   2. Инфраструктура (Infrastructure).
   3. Усовершенствование (Improvement).
   4. Обучение (Training).

Стандарт также описывает следующие 4 стадии ЖЦ ПО:

1. Формирование концепции.
2. Разработка.
3. Сопровождение.
4. Снятие с эксплуатации.

Стандартные процессы

Рассмотрим кратко содержание процессов ЖЦ.

Основные процессы

Первые два процесса связаны с договорной точкой зрения (рис.3.15 и 3.16).

1. *Приобретение* состоит из действий заказчика, приобретающего продукт или услугу, связанную с продуктом, на основе договорных отношений. Процесс включает следующие действия:

— Инициирование (Initiation).
— Подготовка запроса на предложение (Request-for-proposal preparation).
— Подготовка и корректировка договора (Contract preparation and update).
— Мониторинг поставщика (Supplier monitoring).
— Приёмка и завершение (Acceptance and completion).

2. *Поставк*а состоит из действий поставщика, снабжающего заказчика продуктом или услугой, связанной с продуктом, на основе договорных отношений. Процесс включает следующие действия:

— Инициирование (Initiation).
— Подготовка предложения (Preparation of response).
— Разработка договора (Contract).
— Планирование (Planning).

— Выполнение и контроль (Execution and control).

— Обзор и оценка (Review and evaluation).

— Приёмка и завершение (Acceptance and completion).

Третий (как и пятый) процесс связан с инженерной точкой зрения (рис.3.16).

3. *Разработка* состоит из действий разработчика, создающего продукт или услугу, связанную с продуктом. Процесс включает следующие действия:

— Подготовка процесса (Process implementation).

— Анализ требований к системе (System requirements analysis).

— Проектирование [архитектуры] системы (System design).

— Анализ требований к ПО (Software requirements analysis).

— Архитектурное проектирование ПО (Software architectural design).

— Детализированное проектирование ПО (Software detailed design).

— Кодирование и тестирование ПО (Software coding and testing).

— Интеграция ПО (Software integration).

— Квалификационное тестирование ПО (Software qualification testing).

— Интеграция системы [в целом] (System integration).

— Квалификационные тестирование системы (System qualification testing).

— Установка ПО (Software installation).

— Поддержка приёмки ПО (Software acceptance support).

Стандарт отмечает, что действия могут пересекаться по времени, т.е. проводиться одновременно или с наложением, а также могут предполагать рекурсию и разбиение на итерации.

Четвёртый процесс связан с эксплуатационной точкой зрения (рис.3.16).

4. *Эксплуатация* состоит из действий оператора службы поддержки. Процесс включает следующие действия:

— Подготовка процесса (Process implementation).

— Эксплуатационное тестирование (Operational testing).

— Эксплуатация системы (System operation).

— Поддержка пользователя (User support).

Пятый (как и третий) процесс связан с инженерной точкой зрения (рис.3.16).

5. *Сопровождение* состоит из действий специалистов сопровождения. Процесс включает следующие действия:

— Подготовка процесса (Process implementation).
— Анализ проблем и модификаций (Problem and modification analysis).
— Реализация модификаций (Modification implementation).
— Обзор и приёмка при сопровождении (Maintenance review/acceptance).
— Миграция [в другую среду] (Migration).
— Снятие ПО с эксплуатации (Software retirement).

Вспомогательные процессы

Все процессы связаны с точкой зрения поддержки (рис.3.16). Кроме того, все процессы, кроме первого, второго и последнего (восьмого), определяют управление качеством (рис.3.15).

1. *Документирование*: действия по формализованному описанию информации, созданной на протяжении ЖЦ.

2. *Управление конфигурацией*: действия по управлению конфигурацией процесса и ПО.

3. *Обеспечение качества*: действия по объективному обеспечению соответствия процесса и ПО установленным требованиям и утверждённым планам.

4. *Верификация*: действия соответствующего субъекта (заказчика, поставщика или независимой стороны) по проверке соответствия (верификации) создаваемых артефактов (как промежуточных продуктов) установленным требованиям в виде спецификаций по мере реализации проекта.

5. *Аттестация*: действия соответствующего субъекта (заказчика, поставщика или независимой стороны) по проверке соответствия (аттестации, сертификации) создаваемого конечного продукта функциональному назначению. Он является более общим процессом, чем верификация, так как учитывает возможное несоответствие ожиданий заказчика и сформулированных требований к системе.

6. *Совместный обзор*: действия по оценке состояния и результатов какого-либо действия; Процесс может использоваться двумя любыми субъектами, когда один из субъектов проверяет другого субъекта при совместном рассмотрении результатов или хода выполнения соответствующих действий.

7. *Аудит*: действия независимых (по отношению к проекту) экспертов по определению соответствия деятельности субъекта принятым требованиям, планам и договору.

8. *Разрешение проблем*: действия по анализу и устранению (решению) проблем (включая обнаруженные несоответствия), независимо от их характера и источника, обнаруженных при реализации проекта.

Организационные процессы

Все процессы связаны с управленческой точкой зрения (рис.3.16). Кроме того, первый процесс определяют непосредственно управление (рис.3.15).

1. *Управление*: основные действия по управлению, включая управление проектом, при реализации процессов ЖЦ.

2. *Инфраструктура*: основные действия по выбору и поддержке базовой структуры какого-либо процесса ЖЦ, в том числе подходов, стандартов и инструментальных средств.

3. *Усовершенствование*: основные действия, выполняемые субъектом при создании, оценке, контроле и усовершенствовании выбранных процессов ЖЦ.

4. *Обучение*: действия по соответствующему обучению и последующему постоянному повышению квалификации персонала.

Адаптация стандарта

Адаптация стандарта подразумевает применение требований стандарта к конкретному проекту, например, в рамках создания внутриорганизационных регламентов ведения проектов ПО.

Адаптация включает следующие виды действий:

— Определение исходной информации для адаптации стандарта.

— Определение условий выполнения проекта.

— Отбор элементов ЖЦ, используемых в проекте или регламентах.
— Документирование и обоснование требований, решений и процессов, связанных с адаптацией и полученных в её результате.

Адаптация также подразумевает определение основных характеристик проекта, в частности выбор методологии и технологии разработки ПО, а также соответствующей модели ЖЦ ПО.

Соответствие проекта стандарту определяется как реализация в рамках конкретного проекта такой модели ЖЦ ПО, которая построена на основе выбора из этого стандарта соответствующих элементов. Выполнение процесса или действия считается завершённым, если решены все требуемые в них задачи в соответствии с предварительно установленными в договорной документации проекта критериями и требованиями.

Любая организация может применять стандарт в качестве условия обеспечения своих договоров. При этом она обязана определить и оговорить минимальный набор требуемых элементов, который обеспечивает проверку её соответствия этому стандарту. Дополнительные нестандартные элементы, необходимые для реализации проекта устанавливаются в договоре.

В следующем стандарте ISO/IEC 15288:2002 адаптация рассматривается уже как отдельный процесс ЖЦ в рамках отдельной группы процессов.

Стандарт ISO/IEC 15288

В настоящее время продолжается разработка нового стандарта ISO/IEC 15288:2002 «Systems Engineering — System Life Cycle Processes» («Системная инженерия — Процессы жизненного цикла систем»), в частности ведётся согласование этого стандарта с предыдущим стандартом ISO/IEC 12207:1995.

В отличие от предыдущего стандарта этот стандарт ориентирован на системы в целом (а не только на ПО в общем и программные системы в частности). Он касается систем, созданных человеком и включающих один или несколько из следующих элементов: аппаратное обеспечение, программное обеспечение, люди, процессы, процедуры, основные средства и природные ресурсы.

Стандарт предназначен для использования:

— заказчиком и поставщиком — для разработки взаимных соглашений (договоров, контрактов и т.п.), касающихся процессов и действий, которые отбираются, согласовываются и выполняются в контексте данного стандарта;
— организацией — для формирования среды необходимых процессов и оценки соответствия между заявленной и утверждённой моделью ЖЦ и её конкретной реализацией;
— проектной командой — для выбора, систематизации и применения элементов среды, сформированной для производства продукции или услуги, и оценки проекта на соответствие заявленной и сформированной среде.

Стандарт обеспечивает основы для моделирования и реализации общих процессов, составляющих ЖЦ систем, предоставляя возможность для их оценки и совершенствования, и, охватывая все концепции и идеи, имеющие отношение к этим системам, начиная от замысла и вплоть до момента снятия с эксплуатации. Процессы ЖЦ, задаваемые стандартом, могут использоваться однократно, многократно или рекурсивно, как по отношению к системе в целом, так и к любым её элементам, применяться для систем единичного и массового производства, а также адаптируемых к требованиям организации и/или проекта.

Процессы в стандарте образуют полное множество, из которого организация может конструировать модели ЖЦ систем, соответствующие их продуктам и услугам. Процессы могут поддерживаться инфраструктурой, включающей методы, процедуры, технологии, инструменты и обученный персонал. Организация может применять среду процессов для выполнения и управления проектами и развития систем на протяжении их ЖЦ.

Архитектура ЖЦ системы

Стандарт представляет ЖЦ системы (аналогично ЖЦ ПО в предыдущем стандарте) как структуру дробления работ.

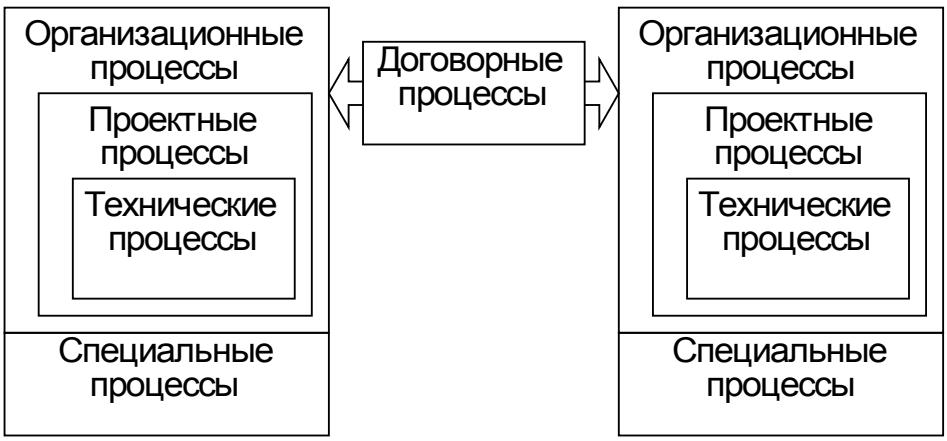


Рис.3.17. Взаимосвязь между группами процессов

Стандарт описывает следующие 26 процессов ЖЦ системы, объединённые в 5 групп (рис.3.17):

1. *Договорные процессы* (контрактные процессы, процессы контрактации, процессы [выработки] соглашений):
   1. Приобретение [системы].
   2. Поставка [системы].
2. *Организационные процессы* (обеспечивающие процессы, процессы предприятия, процессы [уровня] организаций):
   1. Управление инфраструктурой / Управление окружением.
   2. Управление инвестициями / Управление портфелем проектов.
   3. Управление процессами ЖЦ / Управление моделью ЖЦ.
   4. Управление ресурсами / Управление персоналом.
   5. Управление качеством.
3. *Проектные процессы* (процессы предприятия, процессы [уровня] проекта):

— Процессы управления проектами:

   1. Планирование [проекта].
   2. Мониторинг / Управление выполнением и контроль [проекта].

— Процессы поддержки проектов:

   3. Оценивание / Измерение.

4. Управление решениями / Выработка решений.
5. Управление рисками.
6. Управление конфигурацией.
7. Управление информацией.

4. *Технические процессы*:
   1. Определение требований [заинтересованных лиц].
   2. Анализ требований.
   3. Проектирование архитектуры.
   4. Реализация / Изготовление.
   5. Интеграция / Комплексирование.
   6. Верификация / Проверка.
   7. Переход / Ввод в действие.
   8. Аттестация / Валидация.
   9. Эксплуатация / Функционирование.
   10. Сопровождение / Обслуживание.
   11. Снятие с эксплуатации / Вывод из действия.
5. *Специальные процессы*:
   1. Настройка / Адаптация.

Стандарт также описывает следующие 6 стадий ЖЦ системы:

1. Формирование концепции: анализ потребностей, выбор концепции и проектных решений.
2. Разработка: проектирование системы.
3. Реализация: изготовление системы.
4. Эксплуатация: ввод в действие и использование системы.
5. Поддержка: обеспечение функционирования системы.
6. Снятие с эксплуатации: прекращение использования, демонтаж, архивирование системы.

Крайне важной для практики является детализация стандарта до уровня целей, результатов и конкретных действий. Помимо процессов стандарт определяет 208 действий и 123 различных результата этих действий.

Контрольные вопросы

Вопросы к §3.1

1. Дайте определение понятию «жизненный цикл ПО». Каким образом представляется ЖЦ ПО в контексте деятельности?
2. Дайте определение понятию «модель ЖЦ».
3. Дайте определение понятию «технология» («технологический подход»).
4. Каким образом характеризуется технология?
5. Дайте определение понятию «действие». Какие наборы действий выделяют?
6. Дайте определение понятию «процесс».
7. Приведите графическое представление процесса.
8. Приведите и поясните иерархию понятий, связанных с процессом.
9. Дайте определение понятиям «поток работ» и «рабочий продукт».
10. Дайте определение понятию «процедура» и поясните его.
11. Дайте определение понятию «стадия».
12. Приведите и поясните иерархию понятий, связанных со стадией.
13. Сформулируйте описание измерений технологии.
14. Дайте определение понятиям «методика» и «практика».
15. Перечислите ограничения проекта. Что такое треугольник ограничений?
16. Дайте определение понятиям, связанным с произведённым результатом.
17. Дайте определение понятиям «артефакт» и «рабочий продукт».
18. Дайте определение понятиям «базовая линия» и «базовый план».
19. Дайте определение понятиям «контрольная точка» и «веха».
20. Дайте определение понятиям «итерация» и «таймбокс».
21. Как понятия, связанные с формализацией, характеризуют технологию?

Вопросы к §3.2

22. Перечислите и поясните основные наборы технологических процессов.

23. Приведите классические технологические процессы.
24. Поясните особенности классических процессов.
25. Приведите группы стандартных технологических процессов.
26. Перечислите документы для поддержки стандарта по процессам ЖЦ ПО.
27. Как называется новый стандарт? Как осуществляется его поддержка?
28. Перечислите и поясните виды формирования технологических стадий.
29. Приведите классические стадии. Поясните их особенности.
30. Приведите и поясните основные фазы.
31. Приведите и поясните дополнительные фазы.
32. Дайте определение понятию «цикл».
33. Перечислите признаки классификации проектов. Приведите граничные значения или категории проектов по каждому признаку классификации.
34. Перечислите классы технологических подходов. В чём заключаются особенности их применения и предъявляемые основные требования?
35. Приведите группы технологических подходов в рамках каждого класса.
36. Какие существуют направления развития технологических подходов?

Вопросы к §3.3

37. Перечислите основные модели ЖЦ ПО.
38. В чём суть непланируемой модели? Приведите графическое представление этой модели.
39. В чём суть классической каскадной модели? Приведите графическое представление этой модели.
40. В чём суть модифицированной каскадной модели? Приведите графическое представление этой модели.
41. В чём суть прототипируемой модели? Объясните название этой модели.
42. В чём суть классической модели прототипирования? Приведите графическое представление этой модели.

43. Охарактеризуйте принцип разработки прототипированием. Приведите графическое представление.
44. В чём суть итеративной инкрементной модели? Приведите графическое представление этой модели.
45. Охарактеризуйте принципы итеративности и инкрементности.
46. Дайте определение стратегиям итеративной и инкрементной разработки.
47. В чём существенное отличие между итеративной и инкрементной разработки?
48. В чём суть эволюционной модели? Приведите графическое представление этой модели.
49. Охарактеризуйте принцип эволюционности.
50. В чём суть спиральной модели? Объясните название этой модели.
51. В чём суть классической спиральной модели? Приведите графическое представление этой модели.
52. Охарактеризуйте особенность классической спиральной модели. Что такое риск? Перечислите наиболее распространённые риски.
53. Перечислите ключевые практики для классической спиральной модели.
54. Перечислите контрольные точки для классической спиральной модели.
55. В чём суть модифицированной спиральной модели? Приведите графическое представление этой модели.
56. Перечислите контрольные точки для модифицированной спиральной модели.
57. Приведите примеры новых моделей ЖЦ ПО.

Вопросы к §3.4

58. Дайте определение классическому процессу «Исследование идеи».
59. Дайте определение классическому процессу «Управление».
60. Поясните понятия, связанные с целью проекта.
61. Дайте определение классическому процессу «Анализ».
62. Дайте определения понятиям, связанным с требованиями.

63. Дайте определения понятиям, связанным со спецификациями.
64. Дайте определение классическому процессу «Проектирование».
65. Дайте определения понятиям, связанным с системой.
66. На какие подпроцессы разделяется проектирование? В чём они заключаются?
67. Дайте определения понятиям, связанным с архитектурой ПО.
68. Дайте определение классическому процессу «Кодирование».
69. Дайте определение понятию «конструирование».
70. Перечислите фундаментальные основы конструирования.
71. Что такое язык конструирования? Перечислите языки конструирования.
72. Перечислите факторы, влияющие на выбор языка конструирования.
73. Дайте определение классическому процессу «Тестирование».
74. Что включает в себя тестирование.
75. Что представляет собой инспектирование?
76. Дайте определения понятиям, связанным с нарушением работы ПО.
77. Дайте определение классическому процессу «Ввод в действие».
78. Дайте определение классическому процессу «Сопровождение».
79. Сформулируйте основной принцип сопровождения.
80. Перечислите категории сопровождения с учётом подходов и групп.
81. Перечислите уровни модификации ПО. Как они связаны с сопровождением?
82. Дайте определение классическому процессу «Снятие с эксплуатации».

Вопросы к §3.5

83. Перечислите и поясните способы декомпозиции систем.
84. Дайте определения понятиям, связанным с моделями и методами анализа и проектирования.
85. Перечислите основные модели и методы анализа требований для структурной методологии.

86. Перечислите основные модели и методы анализа требований для объектно-ориентированной методологии.
87. Перечислите основные модели и методы проектирования архитектуры для структурной методологии.
88. Перечислите основные модели и методы проектирования архитектуры для объектно-ориентированной методологии.
89. Перечислите основные модели и методы проектирования компонентов для структурной методологии.
90. Перечислите основные модели и методы проектирования компонентов для объектно-ориентированной методологии.
91. Перечислите основные подходы (методики) к анализу и проектированию для структурной методологии.
92. Перечислите основные подходы (методики) к анализу и проектированию для объектно-ориентированной методологии.

Вопросы к §3.6

93. Дайте определения понятиям «базовый стандарт» и «профиль стандарта».

Стандарт ISO/IEC 12207

94. В чём заключается цель стандарта ISO/IEC 12207:1995?
95. Как определяется ЖЦ в стандарте ISO/IEC 12207:1995?
96. Какие элементы ЖЦ выделены в стандарте ISO/IEC 12207:1995? Приведите и поясните иерархию элементов ЖЦ. На чём базируется разбиение процесса.
97. Перечислите и поясните принципы выделения стандартных процессов.
98. Перечислите группы стандартных процессов. Приведите графическое представление групп процессов.
99. Поясните взаимосвязь между стандартными процессами. Приведите графическое представление этой взаимосвязи.
100. Перечислите точки зрения на процессы основных заинтересованных лиц.
101. Перечислите основные процессы стандарта ISO/IEC 12207:1995.

102. Перечислите вспомогательные процессы стандарта ISO/IEC 12207:1995.
103. Перечислите организационные процессы стандарта ISO/IEC 12207:1995.
104. Перечислите стадии по стандарту ISO/IEC 12207:1995.
105. Дайте краткое описание стандартного процесса «Приобретение». Перечислите включаемые в него действия.
106. Дайте краткое описание стандартного процесса «Поставка». Перечислите включаемые в него действия.
107. Дайте краткое описание стандартного процесса «Разработка». Перечислите включаемые в него действия.
108. Дайте краткое описание стандартного процесса «Эксплуатация». Перечислите включаемые в него действия.
109. Дайте краткое описание стандартного процесса «Сопровождение». Перечислите включаемые в него действия.
110. Дайте краткое описание вспомогательных стандартных процессов.
111. Дайте краткое описание организационных стандартных процессов.
112. Дайте краткое описание процесса адаптации стандарта. Перечислите включаемые в него действия.
113. Как определяется соответствие проекта стандарту ISO/IEC 12207:1995?
114. Как обеспечивается возможность использования организацией стандарта ISO/IEC 12207:1995 в своих договорах?

Стандарт ISO/IEC 15288

115. В чём заключается назначение стандарта ISO/IEC 15288:2002?
116. Приведите группы процессов по стандарту ISO/IEC 15288:2002. Приведите графическое представление взаимосвязи групп процессов.
117. Перечислите договорные процессы стандарта ISO/IEC 15288:2002.
118. Перечислите организационные процессы стандарта ISO/IEC 15288:2002.
119. Перечислите проектные процессы стандарта ISO/IEC 15288:2002.
120. Перечислите технические процессы стандарта ISO/IEC 15288:2002.

121. Перечислите специальные процессы стандарта ISO/IEC 15288:2002.

122. Перечислите стадии по стандарту ISO/IEC 15288:2002.

Раздел 4. Подходы разработки ПО

В данном разделе рассматриваются основные часто используемые и/или наиболее известные технологические подходы разработки ПО.

Многие подходы используют для своего описания понятия «процесс» (process), «процесс разработки ПО» (SDP — software development process) или «процесс разработки системы» (SDP — system development process), под которым понимается собственно подход к разработке, а не какой-либо конкретный процесс ЖЦ, описываемый этим подходом.

Кроме того, в названиях подходов можно встретить и такие понятия, как «каркас», «метод», «методология», «доставка», «разработка» и «инженерия» и т.п. Это связано прежде всего со следующими факторами:

— ориентация подхода на конкретные проекты и предлагаемые решения;
— использование в подхода определённых процессов и стадий ЖЦ;
— рекламная политика соответствующей фирмы, продвигающей подход.

Поэтому название подхода требует приемлемого перевода на русский язык.

4.1. Каскадные технологические подходы

Каскадные подходы (stage-wise approaches) являются строгими подходами, получившими также название *жёстких подходов* (hard approaches). Эти подходы основаны на одноимённых моделях ЖЦ. Они требуют определённости, полноты и точности требований, предъявляемых к ПО, и чёткости оценки полученного в результате разработки продукта.

Данные подходы рекомендуются для промышленной разработки систем военного назначения, аэрокосмических систем или систем управления технологическими процессами в реальном времени, а также других критических систем.

Выделяют каскадные подходы следующих двух видов:

1. Простые каскадные подходы: классический и модифицированный.
2. Развитые каскадные подходы: каскадно-возвратный, каскадно-итерационный, каскадно-перекрывающийся и каскадно-декомпозиционный.

Развитые каскадные подходы являются, как следует из названия, развитием простых каскадных подходов. Их появление связано с необходимостью устранения недостатков, присущих простым каскадным подходам.

Самым эффективным оказывается комбинирование нескольких развитых каскадных подходов для получения такого каскадного подхода, который сочетает достоинства используемых им подходов.

Наиболее известным является классический каскадный подход, на основе которого разработано множество строгих подходов и в противоположность ему создано множество гибких подходов.

В моделях ЖЦ для каскадных подходов используются классический набор процессов и попроцессное формирование стадий.

Простые каскадные подходы

Модели ЖЦ для простых каскадных подходов приведены в §3.3.

Классический каскадный подход

Классический каскадный подход (pure stage-wise, или waterfall — водопад) является теоретически наиболее привлекательным, но не рекомендуемым к практическому применению подходом из-за жёстких ограничений, наложенных на выполнение процессов и отдельных действий.

Снятие ограничений или учёт других особенностей разработки ПО приводит к более реалистичным каскадным подходам на основе соответствующих моделей.

Модифицированный каскадный подход

Модифицированный каскадный подход (modified stage-wise, или whirlpool — водоворот) заменяет одно из ограничений на более слабое.

В этом подходе допускается возврат только на непосредственно предыдущую стадию. Однако на практике ошибки одной стадий не всегда проявляются на стадии, непосредственно следующей за стадией, вызвавшей ошибку.

Развитые каскадные подходы

Модели ЖЦ для развитых каскадных подходов приведены при изложении соответствующих подходов.

Каскадно-возвратный подход

Каскадно-возвратный подход (feedback stage-wise, или intermediate check — промежуточный контроль) позволяет возвратиться на любую из предыдущих стадий ЖЦ.

Это принцип отражает итерационный характер разработки ПО. Кроме этого подход учитывает существенное запаздывание с достижением результата проекта из-за влияния корректировки при возвратах.

Каскадно-возвратная модель (reverted stage-wise model) или *модель с промежуточным контролем* (intermediate-check model) является усовершенствованным вариантом модифицированной каскадной модели и представляет собой модель с обратными связями между стадиями.

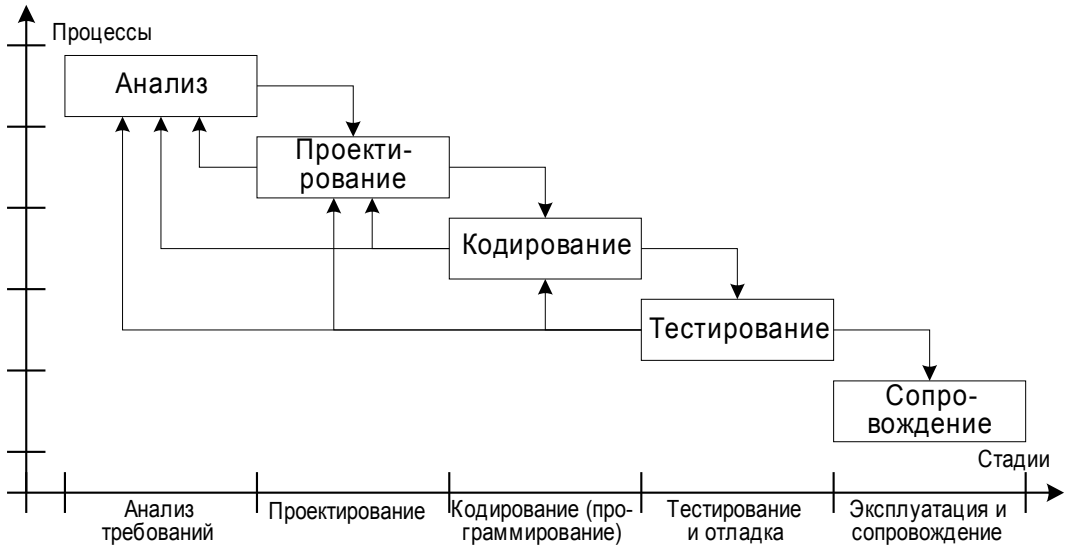


Рис.4.1. Каскадно-возвратная модель

Принцип модели (рис.4.1) проявляется в обработке ошибок, выявленных промежуточным контролем. Промежуточный контроль заключается в проведении проверок и корректировок проекта на каждой из стадий.

Если на какой-либо стадии в ходе такого контроля обнаружилась ошибка, допущенная на любой более ранней стадии, работы стадии, вызвавшей ошибку,

необходимо провести повторно. При этом анализируются причины ошибки и корректируются, по необходимости, исходные данные стадии или перечень проводимых работ.

Каскадно-итерационный подход

Каскадно-итерационный подход (iterated stage-wise, или waterfall with iterations — водопад с итерациями) предусматривает последовательные итерации каждого процесса.

Каждая итерация является завершённым этапом, и её итогом будет некоторый конкретный результат. Обычно данный результат будет промежуточным, не реализующим всю ожидаемую функциональность.

Каскадно-итерационная модель (iterated stage-wise model) является другим вариантом модифицированной каскадной модели и представляет собой модель с обратными связями внутри стадий.

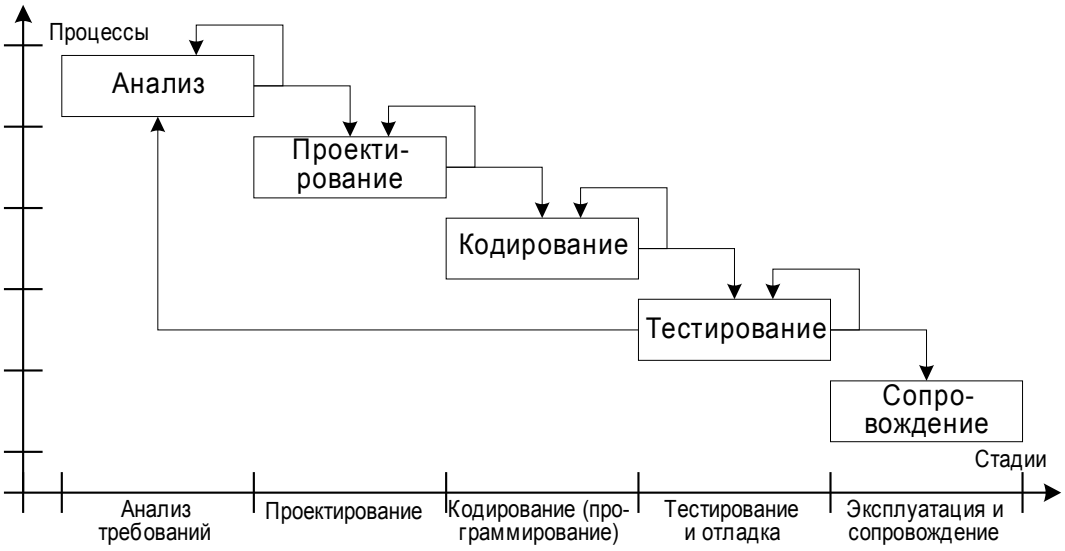


Рис.4.2. Каскадно-итерационная модель

Принцип модели (рис.4.2) заключается в повторении (итерировании) каждого процесса ЖЦ до тех пор, пока не будет достигнут требуемый результат процесса в целом.

Каскадно-перекрывающийся подход

Каскадно-перекрывающийся подход (overlapping stage-wise, или waterfall with overlapping — водопад с перекрытием) предполагает закрепление каждого процесса за отдельной командой, которая будет выполнять этот процесс во всех проектах.

Наличие специализированных команд и их раннее включение в работу над проектом позволяет до определённой степени сократить передаваемую документацию. Следующий процесс начинается до завершения текущего. Более того, несколько процессов могут выполняться параллельно.

Каскадно-перекрывающаяся модель (overlapping stage-wise model) является вариантом модифицированной каскадной модели без учёта ограничения на перекрытие процессов и представляет собой модель с параллелизмом процессов.

Создателем данной модели считается Питер ДеГрейс (Peter DeGrace). В литературе можно встретить и другое название этой модели — *модель сашими* (sashimi — японское название слоёного блюда из сырой рыбы).

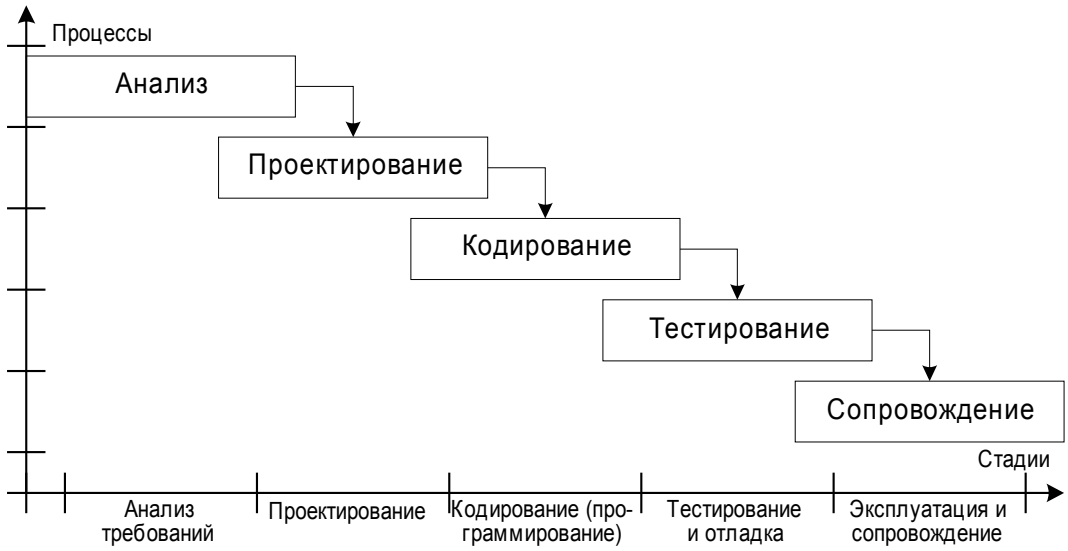


Рис.4.3. Каскадно-перекрывающаяся модель

Принцип модели (рис.4.3) заключается в выполнении процесса не точно в заданных временных рамках, а с началом в предыдущей стадии и завершением в последующей стадии, перекрывая соседние процессы и тем самым сокращая время на разработку ПО. Кроме этого, возникающие в текущем процессе проблемы могут быть разрешены в предыдущем ещё незаконченном процессе, как это происходит и в модифицированной каскадной модели.

Каскадно-декомпозиционный подход

Каскадно-декомпозиционный подход (partitioning stage-wise, или waterfall with subprocesses — водопад с подпроцессами) предполагает возможность представления проекта в виде композиции множества подпроектов.

Наличие необходимого числа команд для их выполнения позволяет значительно сократить время разработки и эффективным образом использовать имеющиеся ресурсы.

Каскадно-декомпозиционная модель (partitioning stage-wise model) является каскадной моделью с учётом декомпозиции системы на модули и представляет собой модель с параллелизмом подпроцессов.

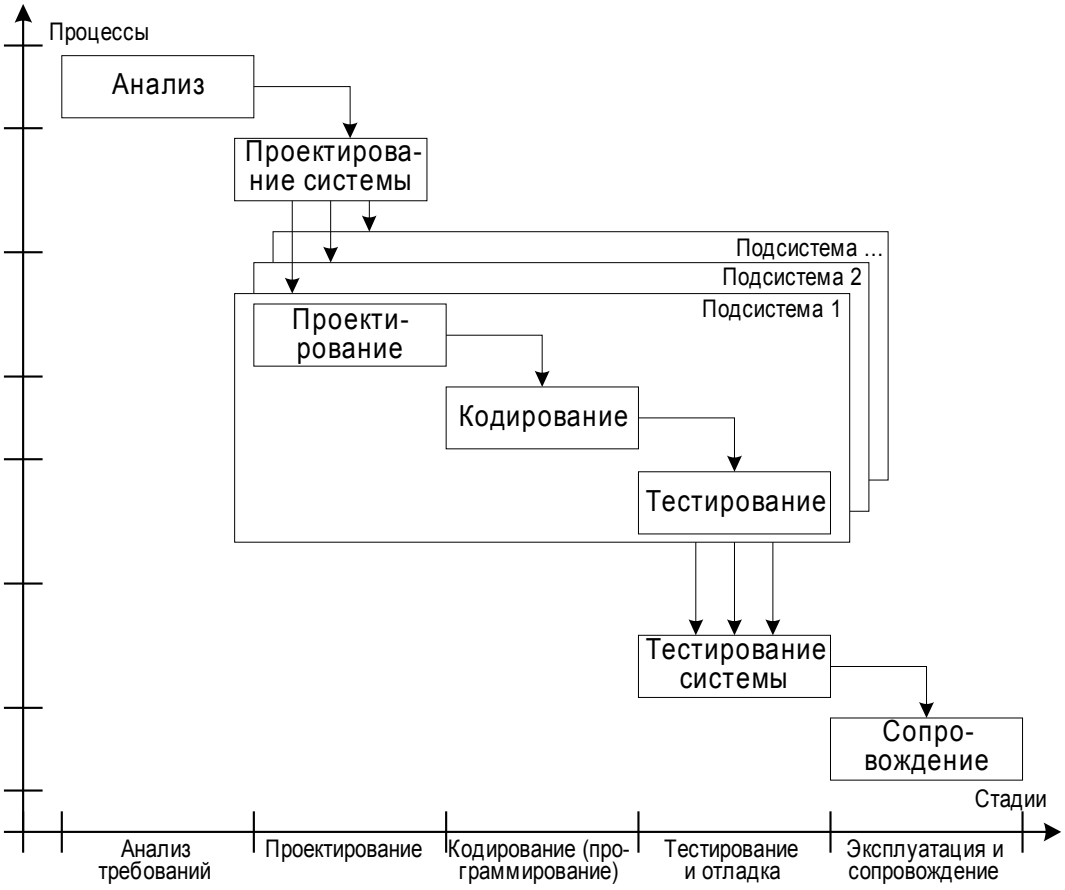


Рис.4.4. Каскадно-декомпозиционная модель

Принцип модели (рис.4.4) заключается в разделении проекта на подпроекты по числу выделенных компонентов и/или имеющихся команд.

4.2. Каркасные технологические подходы

Каркасные подходы (framework approaches) являются развитием каскадных подходов на основе применения развитых моделей ЖЦ и адаптации к современным организационным и экономическим условиям. Поэтому в настоящее время среди строгих подходов именно они чаще применяются на практике.

Выделяют каркасные подходы следующих двух видов:

1. Унифицированные каркасные подходы: Унифицированный процесс (UP) и его модификации, Рациональный унифицированный процесс (RUP).
2. Специализированные каркасные подходы: Каркас решений Майкрософт (MSF), Процесс ICONIX (ICONIX Process), Модельно-основанная (системная) архитектура и программная инженерия (MBASE).

Каркасные подходы предоставляют развитый и адаптируемый под конкретные условия и потребности технологический каркас (framework) для применения в реальных проектах. Он включает в себя набор потоков работ (workflow) и моделей (model) для процессов (или их групп) и представлений (форм) соответственно.

Модели ЖЦ для каркасных подходов приведены при изложении соответствующих подходов. Они являются развитием спиральной модели с учётом других моделей и практических особенностей разработки ПО. В этих моделях используются стандартный набор процессов и пофазное формирование стадий.

Унифицированный процесс (UP)

Унифицированный процесс (УП, UP — Unified Process) — классический унифицированный каркасный подход, развиваемый самостоятельно, отдельно от унифицированного каркасного подхода, предлагаемого фирмой IBM Rational.

Обзор подхода

Первая книга, в которой был описан этот подход,— «Унифицированный процесс разработки ПО» («The Unified Software Development Process») сотрудников Rational Software А. Якобсона, Г. Буча и Дж. Рамбо. Она была опубликована в 1999 г. В дальнейшем авторы книги предпочли для этого подхода название Rational Unified Process. Авторы последующих публикаций по этой тематике, не яв-

ляющиеся сотрудниками этой фирмы, используют название Unified Process. Это позволяет также исключить проблемы с нарушением авторских прав на RUP и Rational Unified Process — торговые марки IBM.

Данный подход представляет собой расширяемый каркас процессов, который может быть настроен для применения конкретными организациями или при выполнении определённых проектов. Он предназначен для описания обобщённого процесса разработки, включая те его элементы, которые являются общими по отношению к различным уточнениям и другим проектным особенностям.

УП обладает следующими особенностями:

1. Итеративность и инкрементность (Iterative and Incremental).
2. Управляемость прецедентами (Use Case Driven).
3. Ориентированность на архитектуру (Architecture Centric).
4. Сосредоточенность на рисках (Risk Focused).

Итеративность и инкрементность означает использование указанных принципов на основе спиральной модели. В ЖЦ выделяются фазы, состоящих из последовательности итераций, которые ограничены временными рамками. Результатом итерации является *инкремент* — очередной прототип.

Управляемость прецедентами предполагает формализацию функциональных требований в виде сценариев, группируемых в прецеденты, которые определяют также содержание итераций. Каждая итерация доводит выбранный набор прецедентов или отдельных сценариев до реализации инкремента.

Ориентированность на архитектуру связана с представлением об архитектуре как о центре, вокруг которого ведётся разработка. Для учёта всех аспектов ПО, осуществляется поддержка множества моделей и представлений архитектуры.

Сосредоточенность на рисках проявляется в обращении внимания на раннее выявление рисков и постоянном обеспечении организации работы с учётом наиболее существенных рисков.

Модель ЖЦ для УП отражает объём работ для каждого потока работ во всех фазах ЖЦ (рис.4.5).

В УП выделены 4 фазы:

1. Начало (Inception — тж. начинание, зачин).
2. Уточнение (Elaboration — тж. развитие, обработка).
3. Построение (Construction — тж. конструирование, реализация).
4. Внедрение (Transition — тж. переход, передача).

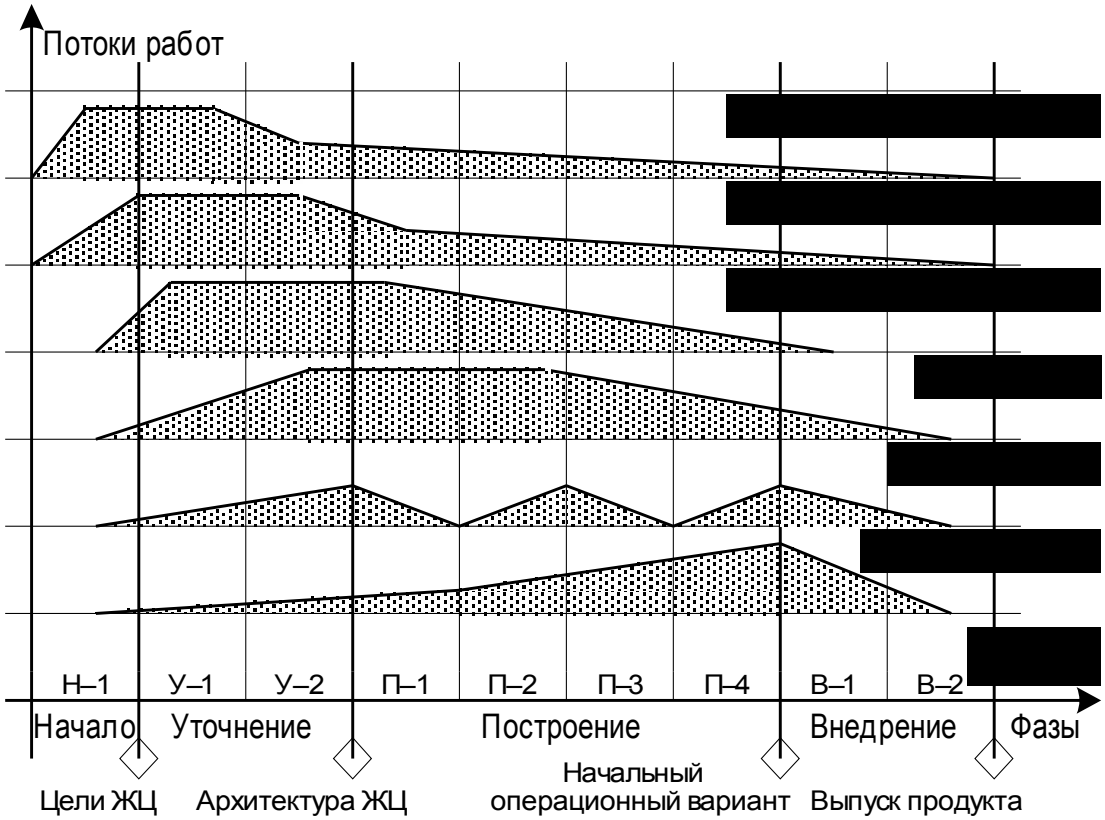


Рис.4.5. Модель ЖЦ для подхода UP

Каждая из фаз включает несколько итераций (в зависимости от конкретной фазы и проекта в целом). На протяжении этих фаз по проекту выполняются работы, сгруппированные в следующие потоки работ (рис.4.5):

1. Бизнес-моделирование (Business Modeling, тж. моделирование бизнес-процессов, в общем случае — моделирование ПрО).
2. Определение требований (Requirements [Capture], тж. сбор требований).
3. Анализ и проектирование (Analysis and Design, тж. моделирование).

4. Реализация (Implementation, тж. выполнение, кодирование).
5. Тестирование (Test).
6. Развёртывание (Deployment, тж. внедрение).

Конец каждой фазы является некоторой вехой. Всего выделено 4 вехи:

1. Цели ЖЦ — Life Cycle Objectives (LCO). Веха определяется достижением договорённости о жизнеспособности проекта и формированием базового плана.
2. Архитектура ЖЦ — Life Cycle Architecture (LCA). Веха определяется подтверждением выбора подхода на основе архитектуры в простейшей форме.
3. Начальный операционный вариант — Initial Operational Capability (IOC). Веха определяется доступностью решения, годного к употреблению.
4. Выпуск продукта — Product Release (PR). Веха определяется завершением разработки и передачей продукта в эксплуатацию.

Перечисленные вехи соответствуют последним четырём вехам модифицированной спиральной модели. Однако 4-я веха в этой модели называется по-другому: Конечный операционный вариант — Final Operational Capability (FOC).

Как следует из названий, потоки работ УП являются подмножеством действий стандартного процесса «Разработка», кроме того используется пофазное формирование стадий.

В отличие от процессов каскадного подхода все потоки работ УП выполняются практически во всех фазах ЖЦ. Для потоков работ в зависимости от фазы применяется соответствующая целевая установка проекта и определяется соответствующий объём работ (рис.4.5).

Модификации подхода

УП имеет множество модификаций. Они различаются числом и видом потоков работ, фаз и артефактов, а также другими особенностями.

Основными модификациями УП являются:

— Рациональный УП (РУП) — Rational UP (RUP) — детализация УП от IBM / Rational Software.
— Живой УП, или Активный УП — Agile UP (AUP) — гибкий вариант РУП от С.У. Амблера.

— Базисный УП — Basic UP (BUP) — гибкий вариант РУП от IBM.
— Корпоративный УП, или УП предприятия — Enterprise UP (EUP) — расширение РУП от С.У. Амблера.
— Сущностный УП — Essential UP (EssUP) — гибкий усовершенствованный вариант РУП от А. Якобсона.
— Открытый УП — Open UP (OpenUP) — часть EPF от Eclipse Foundation.
— Открытый/базисный УП — Open UP/Basic (OpenUP/Basic) — гибкая вариация OpenUP от IBM.
— Рациональный УП — Системная инженерия — Rational UP — System Engineering (RUP-SE) — версия РУП от Rational Software для Системной инженерии.
— Унифицированный метод Оракл — Oracle Unified Method (OUM) — собственный вариант УП от Oracle с использованием собственных продуктов Oracle.
— УП для образования — Unified Process for Education (UPEDU) — вариант РУП, приспособленный для обучения.

Как видно из этого списка, основой для большинства модификаций является не сам УП, а РУП, который подробно рассматривается ниже.

Рациональный унифицированный процесс (RUP)

Рациональный унифицированный процесс (РУП, RUP — Rational Unified Process) — унифицированный каркасный подход, предлагаемый фирмой Rational Software Corporation (с 2003 г. — подразделение IBM Corporation); поэтому возможен и другой перевод названия: *Унифицированный процесс [от] Rational Software*.

Обзор подхода

Исторически РУП является развитием подхода Объекторный Процесс (Objectory Process), принятого в компании Ericsson в 70-х — 80-х гг. XX в. Объекторный Процесс был создан Айваром Якобсоном (Ivar Jacobson) как дальнейшее развитие его методики анализа и проектирования, известной под названием Объектно-ориентированная инженерия ПО. Впоследствии, в 1987 г., автор основал соб-

ственную компанию Objectory AB именно для развития своего технологического подхода разработки ПО как отдельного продукта, который можно было бы переносить в другие организации.

После вливания Objectory AB в Rational в 1995 г. подход Якобсона был объединён с Рациональным Подходом (Rational Approach). Рациональный подход основан на работах Уолкера Ройса (Walker Royce), Филиппа Крухтена (Philippe Kruchten) и Гради Буча (Grady Booch).

Объединённый подход получил название Рациональный Объектный Процесс (РОП, ROP — Rational Objectory Process), затем, после включения поддержки бизнес-моделирования, был переименован в РУП. Кроме этого в подход была включена развивавшаяся в это время сотрудниками Rational объектно-ориентированная методика анализа и проектирования, получившая название языка UML.

РУП является довольно сложным, детально проработанным итеративным подходом разработки. Он представляет собой адаптируемый каркас процессов, который может быть специализирован для конкретных организаций или определённых проектов путём выбора наиболее подходящих элементов из предлагаемого каркаса. Существует множество вариантов РУП для различных областей (см. §4.2, «Унифицированный процесс»).

Rational Unified Process является также продуктом, предоставляемым IBM Rational. В этом качестве он представляет собой структурированную интерактивную базу знаний с шаблонами артефактов и подробным описанием различных типов действий, которое включает в себя общие принципы, конкретные рекомендации и примеры по эффективной разработке систем. База знаний регулярно обновляется и совершенствуется с учётом передового опыта.

Этот продукт входит как составная часть в набор инструментальных средств IBM Rational для поддержки РУП. Некоторые из этих средств благодаря возможности их расширения оказываются применимыми в качестве инструментария для других подходов. В частности, подход MSF долгое время основывался на этом наборе до разработки собственного набора средств.

Изучение опыта

Исследование различных неудачных проектов привело авторов РУП к выявлению признаков (symptom — симптом) и первопричин (root cause — исходная, основная причина) их провала.

Основными считаются следующие первопричины:

1. Непланируемое управление требованиями.
2. Неоднозначное и неопределённое взаимодействие.
3. Хрупкая архитектура (архитектура, не работающая в стрессовых условиях).
4. Непреодолимая сложность.
5. Необнаруженные несогласованности между требованиями, дизайном и реализацией.
6. Недостаточное тестирование.
7. Субъективная оценка состояния проекта.
8. Провал из-за накопившихся рисков.
9. Неконтролируемое распространение изменений.
10. Недостаточная автоматизация.

Проявлениями первопричин считаются следующие признаки:

1. Неточное понимание потребностей конечных пользователей.
2. Неспособность иметь дело с изменяющимися требованиями.
3. Не соответствующие друг другу модули.
4. Тяжёлое для сопровождения и расширения ПО.
5. Позднее обнаружение серьёзных недостатков в проекте.
6. Плохое качество ПО.
7. Неприемлемая производительность ПО.
8. Невозможность участников проекта даже совместно определить (или вспомнить), кто что изменил, когда, где и почему.
9. Ненадёжный процесс построения и выпуска.

Провал проекта обычно связан с некоторой комбинацией указанных признаков, вызванных рядом первопричин, хотя такие проекты заканчиваются неудачей каждый по-своему.

Лучшие практики

Результатом исследований стало выделение и обобщение лучших практик, позволяющих устранить первопричины провала проектов.

Лучшая практика (best practice) — практика разработки ПО, проверенная на многих реальных проектах и позволяющая в комплексе с другими лучшими практиками повысить эффективность проекта и обеспечить успех организации.

Лучшими считаются следующие практики, используемые в РУП:

1. Итеративная разработка (Develop iteratively).
2. Управление требованиями (Manage requirements).
3. Использование компонентной архитектуры (Use component architectures).
4. Визуальное моделирование (Model visually).
5. Проверка качества (Verify quality).
6. Контроль изменений (Control changes).

Смысл перечисленных лучших практик заключается в следующем.

Итеративная разработка заключается в создании требуемой системы итерациями (см. §3.3, Итеративная инкрементная модель, и рис.4.8), что обеспечивает:

— снижение воздействия серьёзных рисков на ранних этапах проекта, пока это ещё можно сделать с минимальными затратами;
— возможность организовать устойчивую обратную связь с будущими конечными пользователями с целью создания системы, реально отвечающей их потребностям;
— концентрация усилий на наиболее важных направлениях проекта;
— непрерывное итеративное тестирование продукта, позволяющее оценить успешность всего проекта в целом;
— раннее обнаружение несогласованностей между требованиями, дизайном и реализацией;
— равномерное распределение ресурсов проекта;
— реальная оценка текущего состояния проекта.

Управление требованиями включает в себя выявление, организацию и документирование требований к системе и подразумевает:

— организованный подход к управлению требованиями;
— взаимодействие участников проекта на базе выявленных и утверждённых требований;
— ранжирование требований по приоритету, фильтрация их по необходимым параметрам и выявление зависимости между ними;
— объективная оценка состояния проекта на основе реализованной функциональности и текущей производительности системы;
— раннее обнаружение различных несоответствий и расхождений;
— использование инструментальных средств для организации более эффективного процесса управления требованиями.

Использование компонентной архитектуры даёт возможность повысить эффективность процесса разработки за счёт:

— повышения гибкости архитектуры создаваемой системы;
— чёткого определения изменений, требующихся при доработке системы;
— наличия множества готовых коммерческих компонентов, которые построены на основе промышленных спецификаций, что облегчает реализацию и позволяет экономить проектные ресурсы;
— упрощения задач по управлению конфигурацией продукта;
— использования средств визуального моделирования, опирающихся на компонентный подход.

Визуальное моделирование — это способ представления проблемы с помощью моделей, построенных вокруг идей из исследуемого мира. Оно позволяет.

— однозначно описать функциональное поведение разрабатываемой системы;
— определить архитектурно значимые компоненты системы и сосредоточится на их реализации;
— обеспечить построение гибкой и надёжной архитектуры и системы в целом;

— исключить из рассмотрения второстепенные детали, не влияющие на решение задачи;
— выявить на ранних стадиях проекта ошибки проектирования и несогласованность в реализации отдельных компонент системы.

Проверка качества реализуется с помощью тестирования сценариев. Непрерывная проверка качества обеспечивает следующие выгоды:

— производит оценку состояния проекта по объективным показателям;
— позволяет на ранних стадиях проекта обнаружить несоответствия в требованиях, дизайне и реализации;
— акцентирует внимание на тех сторонах работы системы, которые имеют наибольшую важность и повышенный риск;
— дефекты выявляются на ранних этапах, снижая затраты на их устранение;
— автоматизированное тестирование обеспечивает снижение влияния «человеческого фактора» и повторяемость результатов.

Контроль изменений включает в себя управление запросами на изменение, управление конфигурацией и управление измерением и обеспечивает:

— контроль состояния проекта в целом и отдельных задач на основании статусов запросов на изменение;
— хранение историй изменений по каждому запросу на изменение;
— актуальную информацию по загрузке участников проекта;
— возможность оценки текущего состояния на основании тенденций по сокращению / увеличению новых запросов на изменение, вновь обнаруженным ошибкам, средним срокам выполнения запросов и т.п.;
— учёт трудозатрат участников проекта по выполняемым изменениям;
— упрощение коммуникаций между участниками проекта: необходимые данные об изменениях всегда доступны и актуальны.

Лучшие практики послужили основой для создания подхода РУП.

Ключевые принципы

РУП основан на наборе из 6 ключевых принципов бизнес-управляемой разработки (business-driven development), сокращённо называемый ABCDEF:

1. Адаптация процесса (**A** — Adapt the process).
2. Баланс приоритетов заинтересованных лиц (**B** — Balance stakeholder priorities).
3. Сотрудничество между командами (**C** — Collaborate across teams).
4. Демонстрация результата итерационно (**D** — Demonstrate value iteratively).
5. Эволюция (рост) уровня абстракции (**E** — Elevate the level of abstraction).
6. Фокусировка непрерывно на качестве (**F** — Focus continuously on quality).

Эти принципы были определены Пером Кроллом (Per Kroll) и Уолкером Ройсом (Walker Royce). Рассмотрим кратко суть этих принципов.

Адаптация процесса реализована в подходе поддержкой предварительно настроенных шаблонов РУП для проектов различного масштаба и возможностью непрерывного совершенствования РУП в организации.

Баланс приоритетов заинтересованных лиц ориентирует организацию с выяснения программных требований на выявление целей бизнеса и требований заинтересованных лиц, противоречия между которыми необходимо обязательно обсудить, разрешить и согласовать.

Сотрудничество между командами требуется для создания системы с учётом мнений всех заинтересованных лиц. Для этого необходимо использовать современные средства коммуникации для общения и обсуждения артефактов.

Демонстрация результата итерационно позволяет показывать инкремент итераций для привлечения заинтересованных лиц к корректировке проекта путём обратной связи. Это позволяет также выполнять итерационную оценку рисков.

Эволюция (рост) уровня абстракции связана с необходимостью организации обсуждения на различных архитектурных уровнях и предотвращения написания программного кода непосредственно по требованиям.

Фокусировка непрерывно на качестве обеспечивается выполнением проверок качества не только в конце каждой итерации, а непрерывной поддержкой этого действия в течение всего проекта, часто ежедневно и с поддержкой всей проектной команды. Важную роль здесь играет автоматизация выполнения тестов.

Жизненный цикл проекта

Модель ЖЦ для РУП отражает объём работ для каждого потока работ во всех фазах ЖЦ (рис.4.6).

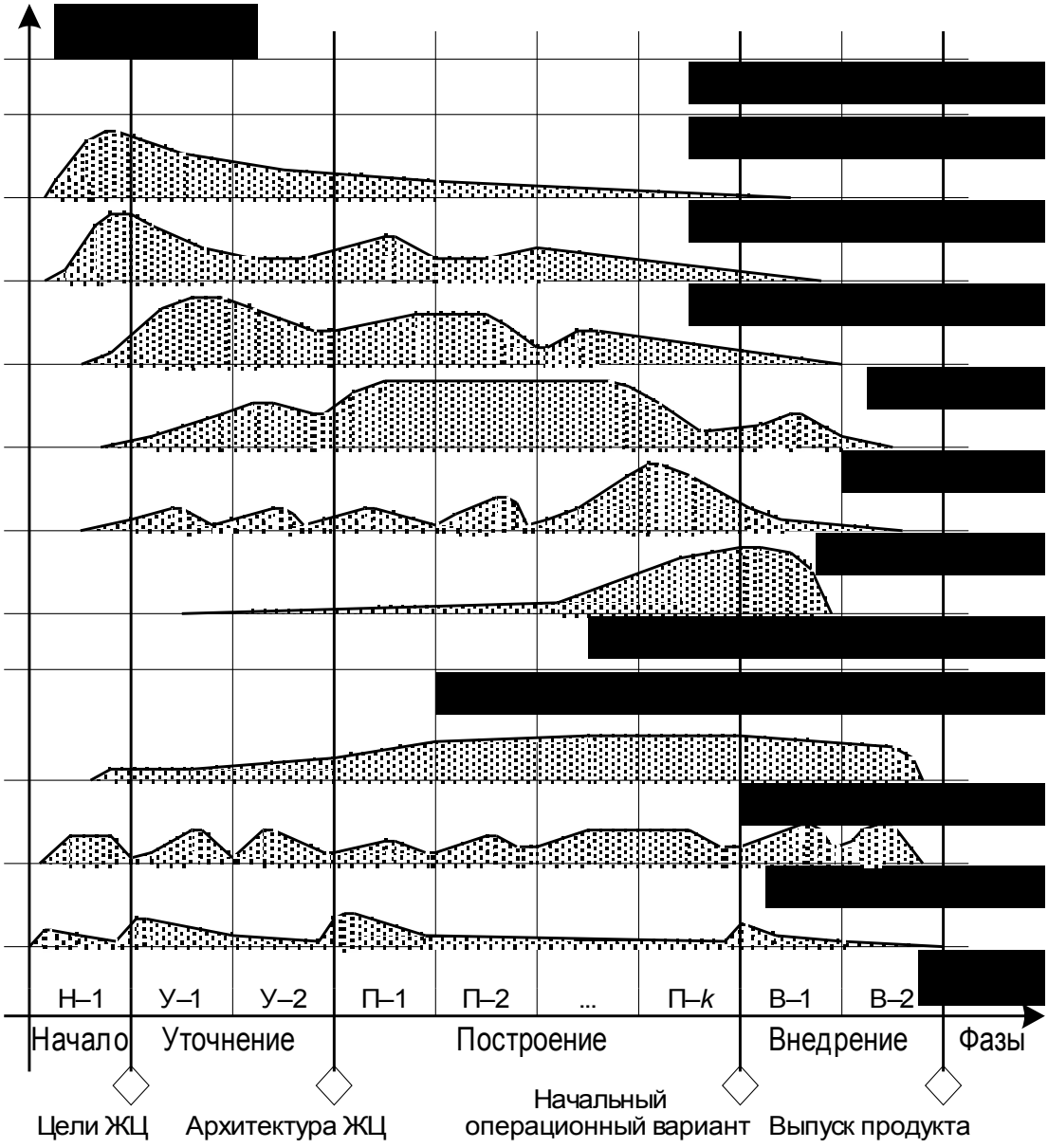


Рис.4.6. Модель ЖЦ для подхода RUP

В РУП, как и в УП (см. §4.2, «Унифицированный процесс»), также выделены 4 фазы, состоящие из ряда итераций.

Основная цель фазы 1 «Начало» (Inception) — достичь компромисса между всеми заинтересованными лицами относительно цели и установок (задач) проекта и выделяемых на него ресурсов. Произведённый результат — базовый план.

Основная цель фазы 2 «Уточнение» (Elaboration) — выполнить анализ ПрО и на базе основных, наиболее существенных требований разработать стабильную базовую архитектуру продукта, которая позволяет решать поставленные перед системой задачи и в дальнейшем используется как основа разработки системы. Произведённый результат — архитектура системы.

Основная цель фазы 3 «Построение» (Construction) — детальное прояснение требований и разработка системы, удовлетворяющей им, на основе спроектированной ранее архитектуры. Произведённый результат — вариант системы, реализующей все выделенные прецеденты.

Основная цель фазы 4 «Внедрение» (Transition) — сделать систему полностью доступной конечным пользователям. Произведённый результат — система, развёрнутая в её рабочей среде, адаптированная под нужды пользователей.

Конец каждой фазы является некоторой вехой. Всего выделено 4 вехи, совпадающие с вехами УП (см. §4.2, «Унифицированный процесс»), кроме того указаны критерии прохождения этих вех.

На протяжении этих фаз по проекту выполняются потоки работ. РУП выделяет 6 основных и 3 вспомогательных потока работ (рис.4.6).

Основные потоки работ, совпадающие с потоками работ УП:

1. Бизнес-моделирование (Business Modeling).
2. Определение требований (Requirements [Capture]).
3. Анализ и проектирование (Analysis and Design).
4. Реализация (Implementation).
5. Тестирование (Test).
6. Развёртывание (Deployment).

Вспомогательные потоки работ, связанные с управлением разработкой:

1. Управление конфигурацией и изменениями (Configuration and Change Management).
2. Управление проектом (Project Management).
3. Управление средой (Environment).

Приведём краткую характеристику всех потоков работ.

Бизнес-моделирование (в общем случае — моделирование ПрО) применяется, чтобы изучить ПрО, обеспечить единство понимания среди всех участников проекта и определить высокоуровневые требования, которые должны быть реализованы в ходе проекта при создании системы.

Определение требований позволяет прийти к соглашению с заинтересованными лицами, определить характеристики системы, предоставить чёткие инструкции участникам проекта о возможностях системы, создать базу для успешного планирования работ в проекте и оценки его статуса в любой момент ЖЦ.

Анализ и проектирование служат для последовательного преобразования выявленных требований к системе в спецификации особого вида, которые описывают, как следует конкретно реализовать конечный продукт. При этом нужно различать анализ и проектирование. Основное различие состоит в следующем. Спецификации анализа не зависят от конкретной платформы и технологии, для которой осуществляется создание ПО. Спецификации проектирования являются точным представлением проектируемой системы, часто позволяя автоматизировать процесс генерации на их основе программного кода.

Реализация необходима для выявления порядка организации программного кода в терминах отдельных подсистем, преобразования исходного кода в выполняемые компоненты, тестирования созданных компонентов и интеграции отдельных компонентов в подсистемы и систему.

Тестирование применяется, чтобы определять и контролировать качество создаваемых продуктов, следить за тем, насколько качественно осуществлена интеграция компонентов и подсистем, все ли требования к системе реализованы

и все ли выявленные ошибки устранены до того, как система будет развёрнута на оборудовании конечного пользователя.

Развёртывание предназначено для доставки разрабатываемого продукта к конечному пользователю. В ходе данного процесса производится новый выпуск / исправление системы, распространение выпуска / исправления, его установка на стороне конечного пользователя, обучение последнего навыкам эффективной работы с поставленным ПО, предоставление услуг по технической поддержке и т.п.

Управление конфигурацией и изменениями позволяет организовать эффективную командную работу с артефактами проекта, контролировать и управлять доступом к ним, вести историю изменений, обеспечить эффективное взаимодействие участников проекта, как в простых командах, так и в распределённых.

Управление проектом включает в себя непосредственное формирование условий для эффективного хода всего проекта, определение руководств и руководящих принципов для планирования, формирования команды и мониторинга проекта, выявление и управление рисками, организацию работы участников проекта, формирование бюджета, планирование фаз и итераций.

Управление средой позволяет осуществить поддержку всех участников проекта. В эту поддержку входят выбор инструментария и его приобретение, настройка и установка, конфигурирование процесса, доработка и адаптация методики, используемой для ведения проекта, обучение.

По трудоёмкости и затратам времени (на один цикл) фазы ЖЦ распределяются следующим образом (рис.4.7): Построение, Уточнение, Внедрение и Начало.

Нагрузка основных потоков работ РУП распределяется по фазам следующим образом: в фазе Начала — Бизнес-моделирование и Определение требований, в фазе Уточнения — Анализ и проектирование и Реализация, в фазе Построения — Реализация и Тестирование, в фазе Внедрения — Развёртывание.

В каждой итерации выполняются все потоки работ РУП, но с разной интенсивностью, зависящей от фазы, и в определённой взаимосвязи (рис.4.8).

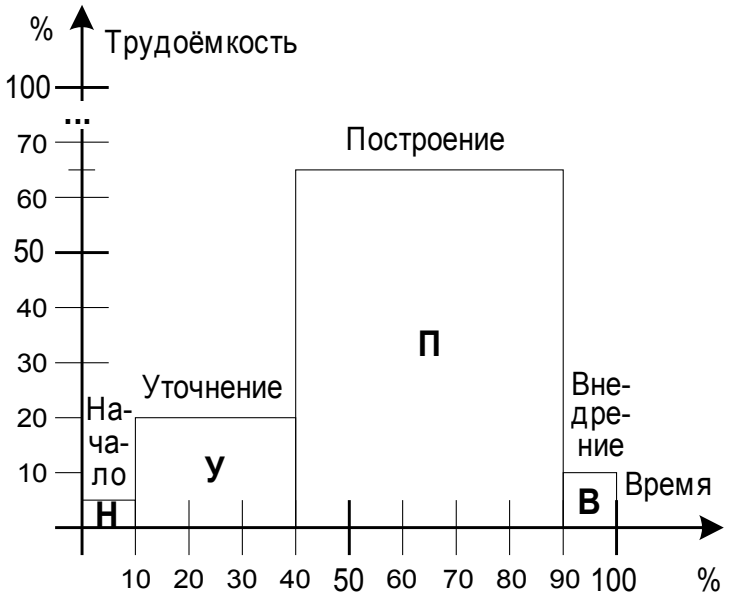


Рис.4.7. Трудоёмкость и затраты времени на фазы

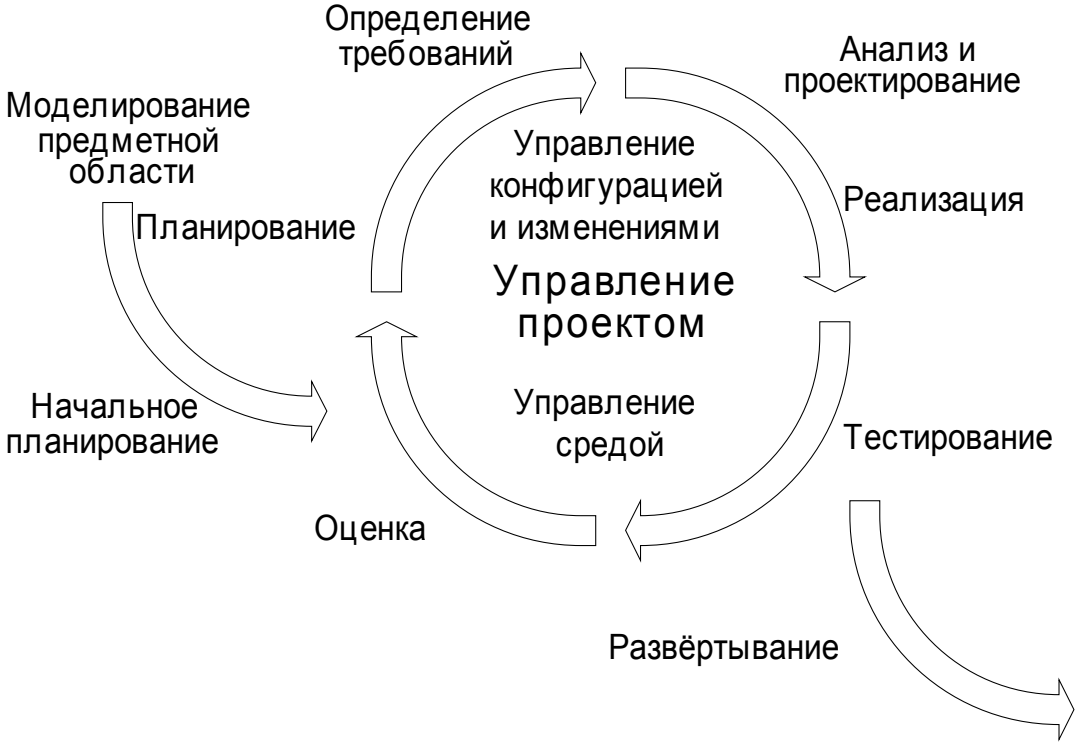


Рис.4.8. Итеративность разработки

Подход РУП включает в себя следующие элементы: 4 фазы, как минимум 8—9 итераций, 9 потоков работ, 57 действий, примерно 270 задач (шагов действий), 114 артефактов (рабочих продуктов) и 38 ролей (работников).

Каркас решений Microsoft (MSF)

Каркас решений Microsoft или *Фреймворк для создания решений от Microsoft* (МСФ, MSF — Microsoft Solutions Framework) — каркасный подход, предлагаемый фирмой Microsoft Corporation.

Обзор подхода

Первый вариант МСФ был представлен как MSF 1.0 в 1993 г. В 1997 г. публикуется MSF 2.0 и Microsoft начинает поставлять концепции МСФ в виде двух курсов, проводимых инструкторами. В 1999 г. выпускается версия 2.5, увеличивается количество курсов, которые представляют уже принципы МСФ. Версия 3.0, опубликованная в 2002 г., содержит существенные изменения, курсы организуются по моделям и дисциплинам МСФ. MSF 4.0, выпущенная в 2005 г., характеризуется обновлением ряда моделей.

МСФ представляет собой каркас процессов, основанных на принципах и использующих опробованные практики. Он обеспечивает адаптируемое наставление, основанное на опыте и лучших практиках, которые применяются в самой Microsoft и других организациях. Это позволяет повысить успешность поставки решения заказчику, уменьшить размер проектной команды, предотвратить риски и получить тем самым высококачественные результаты.

Microsoft Solutions Framework является также продуктом, предоставляемым Microsoft. В этом качестве он представляет собой базу знаний в виде пакета руководств, разделённого на несколько *белых книг* (white paper) — документов, каждый из которых охватывает определённую модель или дисциплину.

Этот продукт входит как составная часть в набор инструментальных средств Microsoft Visual Studio Team System для поддержки МСФ. До разработки этого набора фирма Microsoft основывалась на наборе IBM Rational.

Рассмотрим подробнее содержание продукта Microsoft Solutions Framework версии 4.0. Пакет руководств МСФ 4.0 состоит из 5 белых книг, описывающих две модели и три дисциплины MSF:

— Модель руководства МСФ (MSF Governance Model, ранее в версии 3.0 называлась Модель процессов МСФ — MSF Process Model),
— Модель проектной группы МСФ (MSF Team Model, букв. Модель команды МСФ),
— Дисциплина управления проектами МСФ (MSF Project Management Discipline),
— Дисциплина управления рисками МСФ (MSF Risk Management Discipline),
— Дисциплина управления подготовкой МСФ (MSF Readiness Management Discipline, букв. Дисциплина управления готовностью МСФ).

В этом разделе будет рассмотрена Модель руководства МСФ 4.0, которая собственно и относится к технологии разработки, так как содержит подробное описание ЖЦ для подхода. Модель проектной группы МСФ содержит оригинальную методику построения команд для проектов различного масштаба. Дисциплины управления МСФ содержит изложение соответствующей области управления.

МСФ 4.0 является комбинацией двух компонентов:

1. Описывающий компонент: Метамодель МСФ (MSF metamodel) — фундамент для создания конкретных технологий, включающий основные положения (принципы и концепции), модель проектной группы МСФ и общую часть модели руководства МСФ.
2. Предписывающий компонент: Конкретные технологии, предлагаемые Microsoft в качестве шаблонов для разработки организациями своих подходов.

В рамках МСФ 4.0 выделены два таких шаблона:

1. Строгий шаблон: МСФ для усовершенствования процессов в рамках Интеграции моделей зрелости возможностей (MSF for Capability Maturity Model Integration Process Improvement, MSF4CMMI).
2. Гибкий шаблон: МСФ для гибкой разработки ПО (MSF for Agile Software Development, MSF4ASD).

Как следует из названий этих шаблонов, к строгому каркасному подходу относится первый шаблон, а к гибкому эволюционному — второй шаблон. Поэтому

шаблон MSF4CMMI по сравнению с шаблоном MSF4ASD является более формализованным: он включает больше процессов и артефактов, в частности более высокую формализацию управления разработкой.

В целом МСФ не является таким тяжеловесным подходом как РУП и во многом опирается на конкретные практики.

Принципы и концепции

МСФ основан на наборе из 9 основополагающих принципов (foundational principles):

1. Работа в рамках единого видения (Work toward a shared vision).
2. Проявление живости, ожидание изменений (Stay agile, expect change).
3. Сотрудничество с заказчиками (Partner with customers).
4. Поощрение свободного общения (Foster open communication).
5. Обучение на любом опыте (Learn from all experiences).
6. Вкладывание [денег] в качество (Invest in quality).
7. Поставка инкрементного результата (Deliver incremental value).
8. Установление ясной подотчётности (Establish clear accountability).
9. Наделение полномочиями членов команды (Empower team members).

Эти принципы формируют общую суть моделей и дисциплин МСФ.

МСФ также предлагает набор из 9 ключевых концепций (key concepts):

1. Фокусировка на конечном результате (Focus on business value).
2. Поддержка своей клиентуры (Advocate for your constituency).
3. Чувство гордости за мастерство (Take pride in workmanship).
4. Просмотр всей картины (Look at the big picture).
5. Поставка на своих обязательствах (Deliver on your commitments).
6. Практика хорошего гражданства (Practice good citizenship).
7. Поощрение команды равных (Foster a team of peers).
8. Непрерывное обучение (Learn continuously).
9. Усвоение качеств обслуживания (Internalize qualities of service).

В МСФ 4.0 эти концепции названы мыслеукладами (mindsets, букв. «умственные взоры», тж. умонастроения) в связи со стремлением Microsoft к созданию и внедрению своей культуры разработки решений.

Модель руководства MSF

Модель руководства МСФ обладает следующими тремя особенностями:

1. Итеративный подход.
2. Подход, основанный на фазах и вехах.
3. Целостный подход к созданию и внедрению решений.

Итеративный подход предполагает разработку продукта в виде выпуска работающих прототипов-версий, первый из которых включает базовую функциональность, а остальные создаются с добавлением новых возможностей.

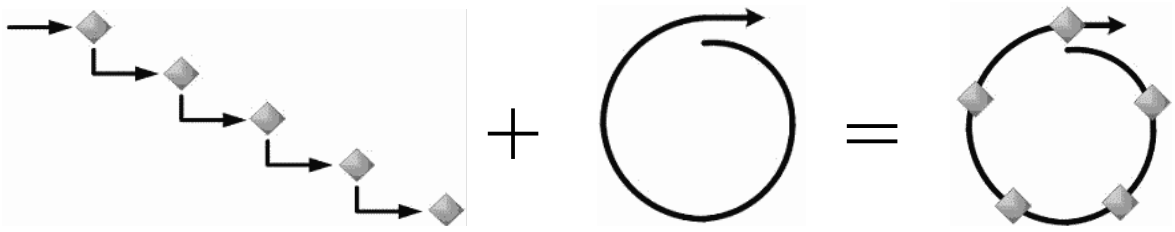
Подход, основанный на фазах и вехах, позволяет управлять проектом. Длительность фаз и их цикличность (итеративность) определяется использованием строгого или гибкого шаблона и самим проектом. Вехи разделяются на два типа:

— главные (major) — межфазные обобщённые вехи,

— промежуточные (interim) — внутрифазные проектно-зависимые вехи.

Целостный подход к созданию и внедрению решений акцентирует внимание на внедрении и дальнейшей поддержке созданного решения.

Модель ЖЦ (рис.4.9), представленная в Модели руководства МСФ, учитывает перечисленные особенности.



а) Каскадная схема б) Спиральная схема в) Схема ЖЦ для подхода

Рис.4.9. Модель ЖЦ как сочетание известных моделей

Как отмечается в самом продукте, модель ЖЦ проекта для подхода является сочетанием каскадной (рис.3.6) и спиральной (рис.3.13) моделей.

Получаемая таким образом модель (рис.4.9) фактически оказывается модификацией прототипируемой модели (см. §3.3), в котором все прототипы являются работающими (готовыми к эксплуатации). Модель ЖЦ для МСФ отражает один цикл разработки (рис.4.10).

Рис.4.10. Модель ЖЦ для подхода MSF

В МСФ выделено всего 5 фаз:

1. Представление (Envisioning).
2. Планирование (Planning).
3. Разработка (Developing).
4. Стабилизация (Stabilizing).
5. Развёртывание (Deploying).

Все фазы разграничены следующими главными вехами:

1. Концепция утверждена (Vision/scope approved).
2. Планы проекта утверждены (Project plans approved).
3. Содержание завершено (Scope complete).
4. Готовность выпуска утверждена (Release Readiness approved).
5. Развёртывание завершено (Deployment complete).

Для повышенного управления проектом внутри фаз выделяют ряд промежуточных вех, показывающих достижение результата в некоторой деятельности.

На фазе 1 «Представление» (тж. Выработка концепции, букв. Формирование видения) выполняется создание и сплочение команды на основе выработки единого видения. Основными задачами являются создание ядра команды (т.е. назначение ключевых членов) и подготовка документа с описанием концепции проекта, включающего видение и содержание проекта (vision/scope document).

Главная веха 1 «Концепция утверждена» считается достигнутой, если команда и заказчик пришли к соглашению об общих задачах и сроках проекта, включаемой и не включаемой в решение функциональности.

Рекомендуемые промежуточные вехи фазы: «Ядро команды сформировано» и «Черновой вариант концепции проекта составлен».

Результатами этой фазы являются:

— Описание видения и содержания.
— Документ оценки рисков.
— Описание структуры проекта.

На фазе 2 «Планирование» производится основная работа по составлению планов проекта. Она включает в себя подготовку командой функциональной спецификации, разработку дизайнов, подготовку рабочих планов, оценку проектных затрат и сроков разработки различных составляющих проекта.

Главная веха 2 «Планы проекта утверждены» считается достигнутой, если заказчик и команда пришли к соглашению о составе поставляемого решения и сроках поставок. Утверждённые спецификации, планы и календарные графики образуют *базовый план проекта* (project baseline).

Рекомендуемые промежуточные вехи фазы: «Аттестация технологий» (validation), «Базовая линия функциональной спецификации создана», «Базовая линия сводного плана проекта создана», «Базовая линия сводного календарного графика проекта создана» и «Среды разработки и тестирования развёрнуты».

Результатами этой фазы являются:

— Функциональная спецификация.
— План управления рисками.
— Сводный план и сводный календарный график проекта.

На фазе 3 «Разработка» команда фокусируется на создании компонентов решения (включая документацию и программный код). Некоторая часть этой работы может продолжаться на следующей фазе, если такая необходимость выявлена при тестировании. Эта фаза также включает в себя разработку инфраструктуры.

Главная веха 3 «Содержание завершено» считается достигнутой, если создание всех компонентов решения завершено и решение готово к тестированию и стабилизации.

Рекомендуемые промежуточные вехи фазы: «Концепция подтверждена» и «Сборка *n* завершена» (build).

Результатами этой фазы являются:

— Исходный и исполнимый код приложений.
— Скрипты установки и конфигурирования.
— Окончательная функциональная спецификация.
— Материалы поддержки решения.
— Спецификации и сценарии тестов.

На фазе 4 «Стабилизация» производится тестирование разработанного решения. При этом внимание фокусируется на его эксплуатации в реалистичной модели производственной среды (пилотное внедрение).

Главная веха 4 «Готовность выпуска утверждена» считается достигнутой, если команда завершила разрешение всех существенных проблем и производится выпуск или внедрение решения.

Рекомендуемые промежуточные вехи фазы: «Сходимость ошибок» (bug convergence), «Достижение нуля ошибок» (zero-bug bounce), «Выпуск-кандидат *k*» (release candidate), «Опытное тестирование завершено», «Тестирование приемлемости для пользователей завершено», «Пилотное внедрение завершено» (pilot release — пилотный выпуск).

Результатами этой фазы являются:

— «Золотой» выпуск (golden release).
— Документация выпуска.
— Материалы поддержки решения.

— Результаты и инструментарий тестирования.
— Исходный и исполнимый код приложений.
— Проектная документация.
— Обзор вехи.

На фазе 5 «Развёртывание» команда внедряет технологии и компоненты решения, стабилизирует внедрённое решение, передаёт работу персоналу поддержки и сопровождения и получает со стороны заказчика окончательное одобрение результатов проекта. По завершении внедрения команда производит анализ выполненной работы и удовлетворённости заказчика.

Главная веха 5 «Развёртывание завершено» считается достигнутой, если решение начало давать заказчику ожидаемый результат, а команда может свернуть свою деятельность.

Рекомендуемые промежуточные вехи фазы: «Ключевые компоненты развёрнуты», «Развёртывание на местах завершено», «Внедрённое решение стабилизировано».

Результатами этой фазы являются:
— Информационные системы эксплуатации и поддержки.
— Процедуры и процессы.
— Базы знаний, отчёты, журналы протоколов (logbooks).
— Версии проектных документов, массивы нагрузки (load sets) и программный код, разработанные во время проекта.
— Отчёт о завершении проекта (project close-out report).
— Окончательные версии всех проектных документов.
— Показатели удовлетворённости заказчика и пользователей.
— Описание последующих шагов.

Следует сделать следующие замечания по этой модели ЖЦ:
— Длительность фаз не одинакова.
— Деятельность может выходить за рамки одной фазы.
— Наличие / отсутствие некоторых фаз определяется выполняемым проектом.

Таким образом, МСФ предлагает модель ЖЦ, основанную на распределении работ в команде проекта по фазам, а не на выделении процессов, как это делается в большинстве других подходов.

Процесс ICONIX (ICONIX Process)

Процесс ICONIX (ICONIX Process) — каркасный подход, предлагаемый фирмой ICONIX Software Engineering, Inc. Название этого подхода официально не является аббревиатурой, хотя и пишется прописными буквами.

Обзор подхода

В то время как в Rational создавался UML, Дуг Розенберг (Doug Rosenberg) разрабатывал средство объектного моделирования — ObjectModeler. Для успешного обучения этому продукту потребовалась организация курсов по объектно-ориентированному анализу и проектированию. В 1992 г. Д. Розенберг разработал собственный подход, названный им Процесс ICONIX. В него были включены отобранные автором приемлемые методы из объектно-ориентированных методик Г. Буча, Дж. Рамбо и А. Якобсона. Следует отметить, что эти три методики составили основу языка UML.

Подход сочетает в себе идеи унифицированных (особенно RUP) и адаптивных (в частности XP) подходов. Критичное отношение к UML, RUP и тем более к XP позволило автору создать свой оригинальный взгляд на процесс разработки.

Подход по своему изложению находится явно ближе к каркасным подходам, чем к адаптивным, что и позволяет рассматривать его в этом параграфе. Он, как и RUP, использует прецеденты, но не требует излишней формализации разработки. В то же время он является небольшим компактным, как и XP, но при этом использует традиционный анализ и проектирование. Поэтому Процесс ICONIX может быть применён на практике и как строгий, и как гибкий подход.

Общая идея подхода состоит в минимизации времени, требуемого для преобразования сформулированных требований к системе в работающий код этой системы. Это достигается специальным отбором только основных моделей UML

(и диаграмм в частности), с помощью которых за 4 этапа выполняется необходимое преобразование.

Таким образом, Процесс ICONIX является упрощённым подходом, ориентированным в первую очередь на моделирование при анализе и проектировании. При этом упрощённость не приводит к снижению строгости разработки, а связана с облегчением разработки при применении этого подхода.

В качестве средств поддержки подхода может быть использовано как инструментальное средство Enterprise Architect самой фирмы, так и набор средств от IBM Rational с расширением (plug-in) для подключения необходимых моделей.

Основными особенностями подхода являются:

1. Упрощённое использование UML (Streamlined usage of the UML).
2. Высокая степень отслеживаемости (High degree of traceability).
3. Итеративность и инкрементность моделей (Iterative and incremental models).

Упрощённое использование UML означает использование минимального подмножества диаграмм UML для объектно-ориентированной разработки. Основные диаграммы позволяют в полной мере охватить анализ и проектирование. Остальные диаграммы могут использоваться по необходимости, что обеспечивает возможность настройки подхода для конкретного проекта.

Высокая степень отслеживаемости позволяет на всём протяжении ЖЦ обеспечить обратную связь с требованиями. Это гарантирует сохранение направления разработки в рамках предъявляемых требований и правильность перехода от анализа к проектированию.

Итеративность и инкрементность моделей связана с итеративностью и инкрементностью построения необходимых моделей во время разработки системы:

1. Итерационное построение моделей ПрО и прецедентов.
2. Итерационное повторение ЖЦ для инкрементной разработки системы.
3. Инкрементное построение статической модели системы (диаграммы классов различного уровня) при итеративном построении динамической модели системы (диаграммы прецедентов, робастности и последовательности).

Ключевые принципы

Сутью Процесса ICONIX является понимание того, что построение хороших моделей объектов является простым, если:

— сосредоточиться непосредственно на нахождении ответа на фундаментально важные вопросы о разрабатываемой системе,

— отказаться от рассмотрения излишних, ненужных проблем моделирования.

Этого можно достигнуть, если придерживаться направления разработки от требований пользователя и модели ПрО к работающему коду.

Разработка в рамках подхода выражается в виде трёх ключевых принципов:

1. Снаружи внутрь (outside-in).
2. Изнутри наружу (inside-out).
3. Сверху вниз (top-down).

Принцип «снаружи внутрь» определяет движение внутрь, исходя из требований пользователя, формализуемых в виде сценариев и прецедентов.

Принцип «изнутри наружу» задаёт движение вовне, исходя из основных абстракций ПрО, образующих соответствующую модель.

Принцип «сверху вниз» обозначает движение вниз — от высокоуровневых моделей к детализированному дизайну.

Модель ЖЦ иллюстрирует эти принципы (рис.4.11).

Жизненный цикл проекта

Модель ЖЦ для Процесса ICONIX отражает построение моделей во всех этапах ЖЦ, связанных с анализом и проектированием (рис.4.11).

В подходе выделено 4 этапа:

1. Анализ требований (Requirements Analysis).
2. Предварительное проектирование (Preliminary Design).
3. Детализированное проектирование (Detailed Design).
4. Реализация (Implementation).

Все этапы разграничены вехами, служащими для обзора работы, выполненной командой на соответствующих этапах:

1. Обзор требований (Requirements Review).

2. Обзор предварительного дизайна (PDR — Preliminary Design Review).
3. Обзор критического дизайна (CDR — Critical Design Review).
4. Поставка (Delivery).

Рис.4.11. Модель ЖЦ для Процесса ICONIX

На этапе 1 «Анализ требований» выполняются следующие действия:

— Определяются объекты ПрО и выясняются связи обобщения и агрегации между ними. На основе этого начинается создание высокоуровневых диаграмм классов — модели (сущностей) ПрО.
— Если возможно, строится прототип предполагаемой системы (модель пользовательского интерфейса). Или собирается вся необходимая информация об унаследованной системе, которую нужно реорганизовать.
— Определяются прецеденты в виде диаграмм прецедентов.
— Организуется группировка прецедентов в виде диаграммы пакетов.
— Размещаются функциональные требования к системе — в прецеденты и объекты ПрО.

Веха 6 «Обзор требований» позволяет установить, что:

— диаграммы прецедентов и модель ПрО совместно отражают все функциональные требования;
— заказчик понимает идею системы и понятно отражает её в требованиях;
— команда способна создать дизайн системы по выделенным требованиям.

Таким образом, осуществляется проверка того, что созданы все диаграммы прецедентов и модель ПрО, необходимые для создания системы.

На этапе 2 «Предварительное проектирование» выполняются следующие действия:

— Формируются описания прецедентов: основной ход событий прецедента («главный поток») и альтернативные ходы событий (редко используемые варианты и ошибочные ситуации).
— Выполняется анализ робастности. Для каждого прецедента:
  - определяется набор объектов, которые участвуют в выбранном сценарии,
  - строится диаграмма робастности с использованием стереотипов из UML Objectory (применяемых в подходе А. Якобсона),
  - обновляются диаграммы классов (добавляются новые объекты, а также новые атрибуты и ассоциации).
— Завершается обновление диаграмм классов. Это действие свидетельствует о завершении стадии анализа для проекта.
— Выбирается *техническая архитектура* (technical architecture) — технологические инструментарий и платформа (в частности, язык программирования и средство построения распределённых программных компонентов).

Веха 7 «Обзор предварительного дизайна» позволяет установить, что:

— описание прецедентов и диаграммы робастности соответствуют друг другу и оба представляют правильно и полностью поведение создаваемой системы;
— модель ПрО соответствует диаграммам робастности (в частности, все объекты-сущности, показанные на диаграммах робастности, представлены и в модели ПрО);
— классы-сущности включают в себя необходимые атрибуты;

— можно проследить потоки данных между именованными экранными формами системы через классы-сущности и, возможно, к лежащему в основе системы хранилищу данных.
— дизайн, разработка которого начинается, выглядит правдоподобным в контексте выбранной технической архитектуры системы.

Таким образом, осуществляется проверка того, что заданы все ключевые абстракции ПрО, необходимые для реализации поведения системы.

На этапе 3 «Детализированное проектирование» выполняются следующие действия:

— «Размещается» поведение системы. Для каждого прецедента:
  - определяются сообщения, передаваемые объектами, сами объекты и соответствующие вызываемые методы,
  - строится диаграмма последовательности: слева указывается текст выполняемого сценария, справа — соответствующий дизайн (сама диаграмма),
  - обновляется диаграмма классов (добавляются новые атрибуты и операции).
— Если необходимо, строятся дополнительные диаграммы:
  - диаграмма кооперации для основных взаимодействий между объектами,
  - диаграмма состояний для поведения системы реального времени.
— Завершается построение статической модели добавлением информации о детальном дизайне (например, области видимости значений и паттерны).
— Командой осуществляется проверка дизайна на соответствие все предъявляемым к системе требованиям. Это действие свидетельствует о завершении стадии проектирования для проекта.

Веха 8 «Обзор критического дизайна» позволяет установить, что:

— детальный дизайн в виде диаграмм последовательности и связанных с ними диаграмм классов соответствует прецедентам;
— детальный дизайн обладает достаточной глубиной (т.е. достаточно подробен), чтобы облегчить относительно небольшой и плавный переход к коду;
— дизайн удовлетворяет заданным критериям качества, позволяющим сделать оценку с различных точек зрения;

— дизайн должен удовлетворять внутренним стандартам, определённым в организации (например, использование паттернов, механизмов доступа к базам данных), что позволяет диаграммам последовательности и детальным диаграммам классов (детальной модели) отразить реальный дизайн системы;
— стабилизированы и утверждены все требования и техническая архитектура;
— решены все оставшиеся проблемы дизайна.

Таким образом, осуществляется проверка того, что определено соответствие детального дизайна требованиям, представленным в виде прецедентов.

На этапе 4 «Реализация» выполняются следующие действия:

— Если необходимо, строятся диаграммы развёртывания и диаграммы компонентов, которые могут оказаться полезными в стадии реализации.
— Пишется или генерируется программный код.
— Осуществляется модульное и интеграционное тестирование.
— Совершается системное тестирование и тестирование приемлемости для пользователей. Это действие свидетельствует о завершении стадий реализации и тестирования для проекта.

Веха 9 «Поставка» позволяет установить, что:

— модульные тесты соответствуют описанию прецедентов и диаграммам последовательности;
— созданный программный код соответствует диаграммам классов и последовательности, рассматриваемых как руководство к написанию кода.

Таким образом, осуществляется проверка того, что определено соответствие программного кода и тестов разработанным статической и динамической моделям — структуре и поведению системы исходя из предъявленных требований.

Одним из преимуществ Процесса ICONIX является то, что для каждого этапа указано 10 наиболее распространённых ошибок (Top 10 Errors) разработки и их разбор с целью исправления.

4.3. Эволюционные технологические подходы

Эволюционные подходы (evolutionary approaches) являются гибкими подходами, на основе которых в дальнейшем были созданы адаптивные подходы. Они основаны на различных моделях прототипирования и связаны, как следует из названия, с эволюционным представлением разработки продукта.

В эволюционных подходах явно видна та же суть, что и у непланируемого подхода, однако необходимость повышения различных характеристик создаваемого ПО привела в них к использованию ряда специальных методик и практик.

Особенностями эволюционных подходов являются:

1. Использование прототипирования.
2. Тесное взаимодействие с заказчиком.

Выделяют эволюционные подходы следующих двух видов:

1. Подходы прототипирования: Эволюционная доставка, Итеративная доставка, Постадийная доставка.
2. Подходы быстрой разработки: Итеративная инкрементная разработка (IID), Быстрая разработка приложений (RAD), Эволюционная быстрая разработка (ERD), Метод разработки динамичных систем (DSDM).

Фактически подходы прототипирования являются вариациями Итеративной инкрементной разработки, на котором основываются и другие подходы быстрой разработки.

Вначале рассмотрим кратко непланируемый подход.

Непланируемый подход

Непланируемый подход (ad-hoc, или code-and-fix — «кодирование — исправление») основан на одноимённой модели.

Фактически это всего лишь способ написания кода программы, простой проверки полученной программы и его модификации, так как он не предполагает серьёзного проектирования. Поэтому многие разработчики не считают его подходом, называя этот способ *кустарной разработкой* (do-it-yourself development — букв. разработка «сделай это для себя»).

Этот подход часто применяется студентами при выполнении ими лабораторных, курсовых и даже дипломных работ и проектов. Если в первых двух случаях он обычно срабатывает, то в последних двух случаях, требующих более высокой формализации процесса разработки ПО, явно проявляются недостатки кустарного программирования.

Непланируемый подход используется также при разработке небольших свободно распространяемых программ. Он рекомендуется в случае необходимости разработки простого демонстрационного прототипа или проверки некоторой программной концепции.

Идея непланируемой модели и подхода оказалась полезной с точки зрения ускорения разработки и получила своё развитие в прототипировании.

Подходы прототипирования

Прототипируемые подходы, или *подходы прототипирования* (prototyping approaches), являются одновременно развитием и альтернативой каскадных подходов и основаны, как следует из названия, на прототипировании.

Выделяют следующие основные подходы прототипирования:

1. Эволюционная доставка.
2. Итеративная доставка.
3. Постадийная доставка.

Эти подходы связаны с выделением в системе пользовательского интерфейса и ядра функциональности. Соответственно этому разрабатываются прототипы определённого вида: горизонтальный и вертикальный.

Прототип, включающий частичную или возможную реализацию пользовательского интерфейса, называется *горизонтальным прототипом* (horizontal prototype). Таким является первый прототип в подходе эволюционной доставки.

Прототип, включающий частичную или возможную реализацию функциональности, называется *вертикальным прототипом* (vertical prototype). Таким является первый прототип в подходе итеративной доставки.

Совмещённый прототип используется в подходе постадийной доставки.

Модели ЖЦ для прототипируемых подходов приведены при изложении соответствующих подходов. Они являются вариантами прототипируемой модели с учётом каскадной и других моделей.

Эволюционная доставка

Эволюционная доставка (evolutionary delivery) — эволюционный подход, ориентированный в первую очередь на создание пользовательского интерфейса.

Основой модели ЖЦ для подхода служит эволюционная модель, так как в начале разработки нет чётко сформулированных требований.

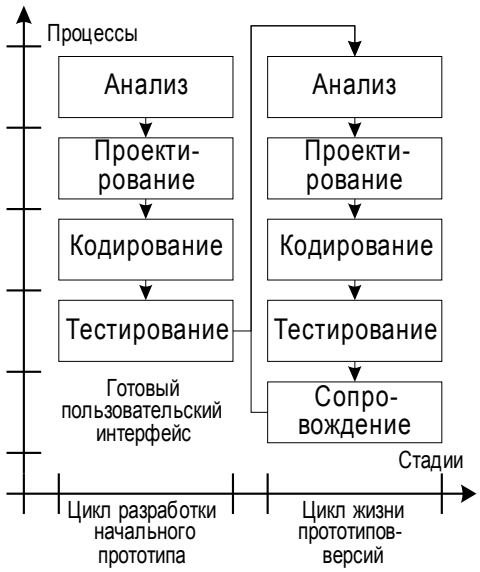


Рис.4.12. Модель ЖЦ для подхода Эволюционная доставка

Принцип модели (рис.4.12) заключается в том, что первый прототип-версия обычно уже включает развитый пользовательский интерфейс (это так называемый горизонтальный прототип). Далее, до тех пор, пока заказчик не сочтёт продукт законченным, в него вносится необходимая функциональность; при этом возможно небольшое изменения интерфейса.

Подход применяется в проектах с ярко выраженным преобладанием разработки пользовательского интерфейса. Существенным недостатком подхода является невозможность определить стоимость и продолжительность проекта.

Итеративная доставка

Итеративная доставка (iterative delivery) — эволюционный подход, ориентированный в первую очередь на создание необходимого ядра функциональности.

Основой модели ЖЦ для подхода служит итеративная инкрементная модель, так как в начале разработки известны чётко сформулированные требования.

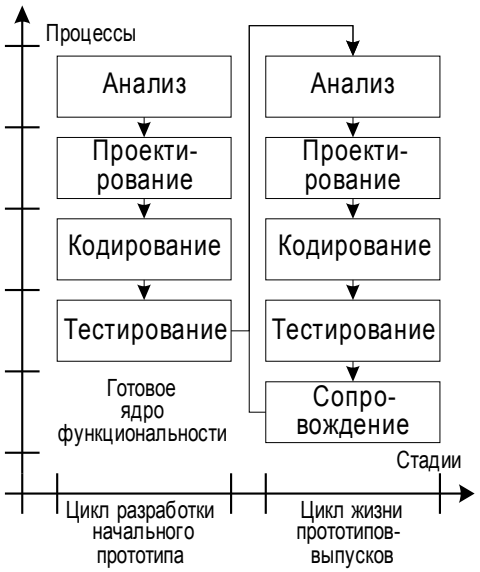


Рис.4.13. Модель ЖЦ для подхода Итеративная доставка

Принцип модели (рис.4.13) заключается в том, что первый прототип-выпуск обычно уже включает большую часть необходимой функциональности (это так называемый вертикальный прототип). Далее, до тех пор, пока заказчик не сочтёт продукт законченным, для него разрабатывается необходимый пользовательский интерфейс (экранные формы, отчёты и другие выходные данные); при этом возможно небольшое изменение функциональности.

Подход применяется в проектах с ярко выраженным преобладанием разработки функциональности. Существенным недостатком подхода также является невозможность определить стоимость и продолжительность проекта.

Постадийная доставка

Постадийная доставка (staged delivery) — эволюционный подход, ориентированный в первую очередь на создание работающих прототипов.

Подход предназначен решить недостаток двух предыдущих подходов прототипирования — невозможность определения сроков завершения проекта. Это достигается обеспечением работоспособности всех создаваемых прототипов.

Основой модели ЖЦ для подхода служит прототипируемая модель, совмещающая итеративную инкрементную и эволюционную модели. Это связано с тем, что в начале ЖЦ известны только основные сформулированные требования.

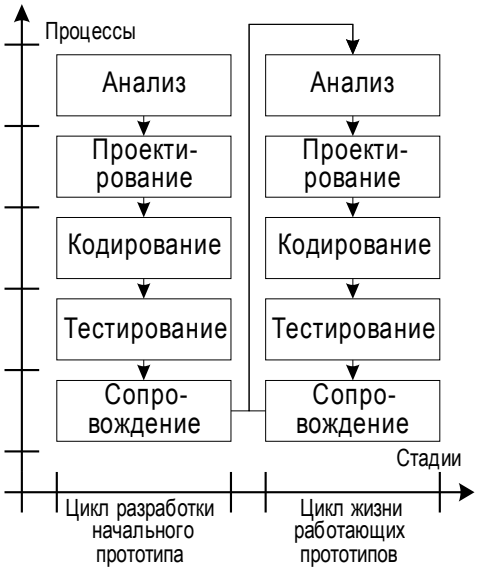


Рис.4.14. Модель ЖЦ для подхода Постадийная доставка

Принцип модели (рис.4.14) заключается в том, что первый прототип обычно включает основную часть необходимой функциональности и при этом является работающим (готовым к эксплуатации). Далее, до тех пор, пока заказчик не сочтёт продукт приемлемым, в рамках отдельных проектов на основе имеющегося прототипа создаётся очередной работающий прототип, включающие в себя реализации новых требований заказчика.

Итеративная инкрементная разработка (IID)

Итеративная инкрементная разработка (ИИР, IID — Iterative and Incremental Development) — подход разработки, являющийся альтернативой (классическому) каскадному подходу и использующий прототипы.

Идея повышения качества путём организации производства в виде коротких циклов «Планирование — Выполнение — Проверка — Действие» (PDCA — «Plan — Do — Check — Act») была предложена в 1939 г. в работе Уолтера Шуарта (Walter Shewhart), эксперта по качеству Bell Labs. Эта идея получила развитие в области разработки ПО и привела к созданию в середине 1950-х гг. подхода ИИР.

ИИР использовался в ряде исследовательских проектов, выполненных подразделением FSD (Federal Systems Division — Отделение федеральных систем) фирмы IBM по заказу Министерства обороны США.

ИИР стал одним из основных компонентов ряда современных строгих и гибких подходов, в том числе RAD, DSDM, RUP и многих живых подходов.

Подход основан на одноимённой модели, приведённой в §3.3.

Быстрая разработка приложений (RAD)

Быстрая разработка приложений (БРП, RAD — Rapid Application Development) — эволюционный подход, сформулированный Дж. Мартином в 1991 г.

Обзор подхода

БРП является развитием подхода ИИР и подходов прототипирования.

В 1980-х гг., основываясь на идеях Брайана Галлахера (Brian Gallagher), Барри Боэма (Barry Boehm) и Скотта Шульца (Scott Shultz), Джеймс Мартин (James Martin), сотрудник фирмы IBM, разработал подход БРП. В 1991 г. он опубликовал книгу под названием «Быстрая разработка приложений» («Rapid Application Development»), в которой он сформулировал основные положения. Название книги и послужило названием подхода.

В дальнейшем подход БРП стал пониматься более широко и включил в себя разнообразные методики, нацеленные на ускорение разработки приложений (например, каркасы приложений).

Разработка с использованием БРП часто связана с достижением компромисса между функциональностью и производительностью системы и ускоренной разработкой и обеспечением сопровождения.

БРП обладает следующими особенностями:

1. Организация команды: Малочисленная проектная команда разработчиков (от 2 до 10 человек) из опытных, разносторонне подготовленных специалистов, способных заменять друг друга.
2. Организация работы: Короткий, но тщательно проработанный график проекта (от 2 до 6 месяцев).
3. Управление командой: Максимальная интеграция управленческого аппарата с командой разработчиков.
4. Использование прототипирования: Повторяющийся цикл разработки и быстрая разработка прототипа перед каждой итерацией для демонстрирования пользователям и уточнения требований для этой итерации.

БРП оказывается эффективным в средних проектах для конкретного заказчика, ориентированных на создание системы с ярко выраженным интерфейсом пользователя, наглядно демонстрирующим логику работы этой системы.

На основе БРП созданы и другие подходы: Адаптивная разработка ПО (ASD), Метод разработки динамичных систем (DSDM) и т.д.

Основные принципы

К основным принципам БРП следует отнести следующие:

1. Разработка системы итерациями, способствующая параллельному созданию подсистем отдельной группой разработчиков.
2. Использование прототипирования, позволяющее полнее выяснить и удовлетворить потребности конечного пользователя.
3. Обязательное вовлечение пользователей в процесс разработки системы.
4. Ведение разработки немногочисленной хорошо управляемой командой профессионалов.

5. Грамотное руководство разработкой системы, чёткое планирование и контроль выполнения работ.
6. Необязательность полного завершения работ на каждой из фаз ЖЦ.
7. Необходимое применение CASE-средств, обеспечивающих целостность проекта на всём протяжении ЖЦ.
8. Применение средств управления конфигурацией, облегчающих внесение изменений в проект и сопровождение готовой системы.
9. Непременное использование генераторов кода.
10. Тестирование и развитие проекта одновременно с разработкой.

Для выполнения разработки согласно перечисленным принципам используется метрика — оценка размера приложения. Эта метрика вычисляется на основе так называемых функциональных элементов, к которым относят следующие элементы приложения:

— входной элемент (документ, экранная форма),
— выходной элемент (отчёт, документ, экранная форма),
— запрос (пара «вопрос — ответ», сообщение),
— файл приложения (совокупность записей данных, используемых внутри приложения),
— интерфейс приложения (совокупность записей данных, передаваемых другому приложению или получаемых от него).

Подобная метрика не зависит от используемого языка программирования. Размер приложения, которое может быть выполнено с помощью БРП, для хорошо отлаженной среды разработки с максимальным повторным использованием программных компонентов определяется следующим образом:

— один человек при числе функциональных элементов менее 1000,
— одна команда при числе функциональных элементов 1000 — 4000,
— 4000 функциональных элементов на одну команду разработчиков при числе функциональных элементов более 4000.

Жизненный цикл проекта

Основой модели ЖЦ для подхода служит итеративная инкрементная модель (см. §3.3), ориентированная на разработку работающих прототипов (рис.4.15).

В БРП выделены следующие 4 фазы:

1. Планирование и Анализ требований.
2. Проектирование.
3. Построение.
4. Внедрение.

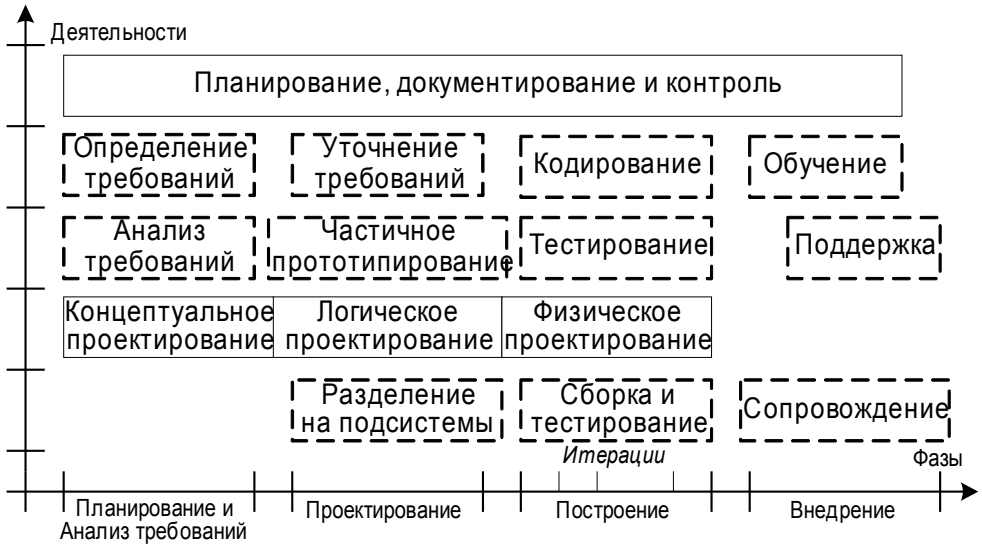


Рис.4.15. Схема модели ЖЦ для подхода RAD

На фазе 1 «Планирование и Анализ требований» пользователи системы определяют функции, которые она должна выполнять, выделяют наиболее приоритетные из них, описывают информационные потребности. Определение требований выполняется в основном силами пользователей под руководством специалистов-разработчиков.

Ограничивается масштаб проекта, определяются временные рамки для каждой из последующих фаз. Кроме того, определяется сама возможность реализации данного проекта в установленных рамках финансирования, на данном аппаратном обеспечении и т.п.

Результатами данной фазы должны быть список и приоритетность функций разрабатываемой системы, предварительные информационные и функциональные модели этой системы.

На фазе 2 «Проектирование» часть пользователей принимает участие в логическом проектировании системы под руководством разработчиков. Для быстрого получения работающих прототипов приложений используется CASE-средство. Пользователи, непосредственно взаимодействуя с ними, уточняют и дополняют требования к системе, которые не были выявлены на предыдущей фазе.

Более подробно рассматриваются процессы системы. Анализируется и корректируется функциональная модель. Каждый процесс рассматривается детально. При необходимости для каждого элементарного процесса создаётся частичный прототип (экран, диалог, отчёт), устраняющий неясности или неоднозначности. Определяются требования разграничения доступа к данным. На этой же фазе происходит определение набора необходимой документации.

После детального определения состава процессов оценивается количество функциональных элементов разрабатываемой системы и принимается решение о разделении её на подсистемы, поддающиеся реализации одной командой разработчиков за приемлемое для RAD-проектов время — порядка 60 — 90 дней. С использованием CASE-средства проект распределяется между различными командами (делится функциональная модель).

Результатами данной фазы должны быть: общая информационная модель системы, функциональные модели системы в целом и подсистем, точно определённые с помощью CASE-средства интерфейсы между автономно разрабатываемыми подсистемами, построенные прототипы экранов, отчётов, диалогов.

Все модели и прототипы должны быть получены с применением того CASE-средства, которое будут использоваться в дальнейшем при построении системы. Применение единой среды хранения информации о проекте позволяет избежать опасности искажения данных.

На фазе 3 «Построение» выполняется собственно быстрая разработка приложения. На этой фазе разработчики производят итеративное построение реальной системы на основе полученных в предыдущей фазе моделей, а также требований нефункционального характера. Программный код частично формируется при помощи автоматических генераторов, получающих информацию непосредственно

из репозитория CASE-средства. Конечные пользователи на этой фазе оценивают получаемые результаты и вносят коррективы, если в процессе разработки система перестаёт удовлетворять определённым ранее требованиям. Тестирование системы осуществляется непосредственно в процессе разработки.

После окончания работ каждой отдельной команды разработчиков производится постепенная интеграция соответствующей части системы с остальными, формируется полный программный код, выполняется тестирование совместной работы этой части с остальными, а затем тестирование системы в целом.

Завершается физическое проектирование системы: определяется необходимость распределения данных, производится 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 фазе 4 «Внедрение» производится обучение пользователей, выполняются организационные изменения. Параллельно с внедрением новой системы осуществляется работа с существующей системой (до полного внедрения новой).

Так как фаза построения достаточно непродолжительна, подготовка к внедрению должна начинаться заранее, как правило, в фазе проектирования.

Приведённая схема разработки не является абсолютной. Возможны различные её варианты в зависимости от целей проекта.

4.4. Адаптивные технологические подходы

Адаптивные подходы (adaptive approaches) являются гибкими подходами, получившими также название *живых подходов* (agile approaches). Они имеют много общего с эволюционными подходами, особенно с подходом RAD.

Особенностями адаптивных подходов являются:

1. Открытое взаимодействие.
2. Разработка короткими итерациями.
3. Адаптируемость процесса разработки.

Выделяют адаптивные подходы следующих видов:

1. Игровые (gaming) адаптивные подходы: Адаптивная разработка ПО (ASD), Экстремальное программирование (XP), Скрам (Scrum).
2. Управляемые (driven) адаптивные подходы: Управляемая тестами разработка (TDD), Управляемая возможностями разработка (FDD), Управляемая поведением разработка (BDD), Управляемая дизайном разработка (D3).
3. Унифицированные (unified) адаптивные подходы: Гибкие варианты UP (список приведён в §4.2, Унифицированный процесс (UP), Модификации UP).
4. Экономные (saving) адаптивные подходы: Бережливая разработка ПО (LSD).

В общем случае каждый адаптивный подход представляет собой определённый набор принципов и основанных на них практик, ориентированных на исполнение особенностей, как общих для всех адаптивных подходов, так и специфических для конкретных подходов. Это позволяет использовать при реализации реальных проектов сочетания различных адаптивных подходов, адаптируя таким образом процесс разработки к конкретным проектам.

Особенности живых подходов

Ряд адаптивных подходов также известен под общим названием *Живая разработка ПО* (Agile Software Development).

Живая разработка ПО

Адаптивные подходы изначально были задуманы как подходы, поддерживающие изменения (в противовес строгим каскадным подходам). Они только выигрывают от изменений, даже когда изменения происходят в них самих.

К началу XXI века было создано множество различных гибких подходов. Эволюционные подходы были одними из первых подходов, ориентированных на увеличение скорости разработки одновременно с уменьшением формализма. На их основе возникли адаптивные подходы, ориентированные не на сами процессы, а на людей, выполняющих эти процессы. Поэтому живыми подходами считается большинство адаптивных подходов и ряд гибких смешанных подходов.

В феврале 2001 г. 17 известных сторонников гибких подходов встретились в местечке Сноубёрд (Snowbird, штат Юта, США), для обсуждения вопросов создания ПО более лёгким, быстрым и «человеко-центрированным» способом. Кроме того они предложили общее название для подходов с указанными выше способами разработки: Живая разработка ПО (Agile Software Development).

Результатом этого обсуждения стал «Манифест Живой разработки ПО» («Manifesto for Agile Software Development»), известный под сокращённым названием «Живой манифест» («Agile Manifesto»).

Живой манифест

Живой манифест включает в себя уведомление (notice) с основными положениями и сам документ с принципами живой разработки ПО.

Основные положения при разработке ПО связаны с правильной оценкой:

1. Люди и их взаимодействие важнее процессов и средств.
2. Работающее ПО важнее исчерпывающей документации.
3. Сотрудничество с заказчиком важнее обсуждения контракта.
4. Реагирование на изменения важнее следования плану.

В манифесте приводятся следующие принципы:

1. Наивысшим приоритетом является удовлетворение заказчика путём своевременной и непрерывной поставки результативного ПО.
2. Приветствуются изменения требований, даже произошедшие во время разработки. Живые процессы используют изменение для конкурентоспособного преимущества заказчика.
3. Чаще доставляйте работающий продукт — от нескольких недель до нескольких месяцев — с предпочтением кратчайших отрезков времени.
4. Представители бизнеса и разработки должны работать совместно каждый день на всём протяжении проекта.
5. Стройте проекты вокруг мотивированных людей. Дайте им среду и поддержку, в которой они нуждаются, и доверяйте им выполнять работу.
6. Самый действенный и эффективный метод передачи информации команде разработки и внутри неё — беседа лицом к лицу.
7. Работающее ПО является первичным показателем прогресса.
8. Живые процессы способствуют жизнеспособной разработке. Спонсоры, разработчики и пользователи должны быть в силах неограниченно долго поддержать постоянный темп работы.
9. Непрерывное внимание на техническом совершенстве и хорошем дизайне улучшает живость.
10. Простота — искусство максимизации объёма несделанной работы — является основной.
11. Лучшие архитектуры, требования и дизайны исходят от самоорганизующихся команд.
12. Регулярно команда размышляет о том, как стать более эффективной, затем соответствующим образом настраивает и корректирует своё поведение.

Этим положениям и принципам должны удовлетворять гибкие подходы, которые относятся к живой разработке ПО (т.е. живым подходам).

Адаптивная разработка ПО (ASD)

Адаптивная разработка ПО (АРП, ASD — Adaptive Software Development) — живой подход, предложенный Джимом Хайсмитом (Jim Highsmith).

Обзор подхода

Идея представления процесса разработки как адаптивной системы была высказана Э.А. Эдмондсом (E.A. Edmonds) в его статье ещё в 1974 г. Продолжительная работа со строгими подходами разработки привела Дж. Хайсмита к выводу об ошибочности их применения в условиях постоянно меняющегося окружения и созданию своего подхода.

Изложение этого подхода приведено в его книге «Адаптивная разработка ПО: Подход сотрудничества при управлении сложными системами» («Adaptive Software Development: A Collaborative Approach to Managing Complex Systems»), изданной в 2000 г. В этой книге нет подробного описания практик, но она закладывает теоретическую основу адаптивных разработок. Это позволяет использовать АРП совместно с другими гибкими подходами (Crystal, FDD, XP). В настоящее время подходы АРП и Crystal Family объединены их авторами в единый подход.

Высокая и частая изменчивость окружения приводит к необходимости изменений и в процессе разработки. Поэтому ключевым положением АРП является естественность постоянной адаптации процесса для выполнения текущей работы. Теоретической основой подхода служат модели сложных адаптивных систем.

Одна из этих моделей основана на трёх ключевых понятиях: агент (agent), среда (environment) и проявление (emergence). Сложная адаптивная система (CAS — complex adaptive system) представляет собой среду, в которой агенты конкурируют и кооперируют друг с другом за выполнение работы. Проявление является ключевым свойством системы: результат работы есть итог сотрудничества агентов, а не действий отдельных агентов. Поэтому функционирование системы не может быть предсказано по поведению агентов.

Хайсмит рассматривает процесс разработки как сложную адаптивную систему: организация-разработчик — это среда, участники проекта — агенты, а про-

дукт — проявляемый результат сотрудничества участников. Такое рассмотрение приводит к иному пониманию разработки и формированию нового подхода.

Свойства подхода

Вместо статического ЖЦ «Планирование — Проектирование — Построение» («Plan — Design — Build») в АРП предлагается динамический ЖЦ «Обдумывание — Сотрудничество — Обучение» («Speculate — Collaborate — Learn»). Этот цикл ставит своей целью непрерывное обучение (рис.4.16).

Рис.4.16. Схема модели ЖЦ для подхода ASD

Цикл связан с постоянными изменениями, повторными оценками, попытками предугадать неизвестное на текущий момент будущее проекта и требует тесного взаимодействия между разработчиками, тестировщиками и заказчиками. При этом весь цикл не всегда представляет собой правильный круг, так как можно иногда отклоняться в сторону, чтобы изучить неисследованные до сих пор области.

Цикл обладает следующими свойствами (property):

1. Целенаправленность (Mission focused).
2. Компонентный подход (Component based).
3. Итеративность (Iterative).
4. Ограниченность по времени (Timeboxed).
5. Управляемость рисками (Risk driven).
6. Допущение изменений (Change tolerant).

Целенаправленность связана с ясной формулировкой задания (mission, тж. миссия), на основе которой определяются цель и содержание проекта. Артефакты, в которых зафиксирована цель проекта, не только помогают указать нужное направление работ, но и используются для того, чтобы в случае острой необходимости найти компромиссное решение.

Компонентный подход определяет построение ЖЦ исходя не из самих задач, а из результатов — компонентов системы. Здесь под компонентом понимается некоторый набор возможностей программы (или некоторых элементов, входящих в поставку системы), который должен быть разработан в течение итерации. Документация по отношению к возможности считается второстепенной, так как только работающий код позволяет заказчику ощутить реальные результаты работы.

Итеративность необходима при создании и переделке компонентов, чтобы адекватно учитывать изменения требований заказчика по мере развития продукта.

Ограниченность по времени требует установки фиксированных сроков поставки для каждого итеративного цикла. Наличие таймбоксов заставляет команду доводить работу, заданную на цикл, до конца и вместе с заказчиками постоянно пересматривать основные показатели проекта.

Управляемость рисками позволяет осознать и проанализировать риски.

Допущение изменений означает приемлемость и приветствование всех возникающих изменений, так как они составляют конкурентное преимущество, а не являются очередной проблемой.

Жизненный цикл проекта

Модель ЖЦ для АРП (рис.4.17) основана на приведённой выше схеме цикла.

В модели эта схема конкретизируется в виде трёх фаз:

1. Обдумывание.
2. Сотрудничество.
3. Обучение.

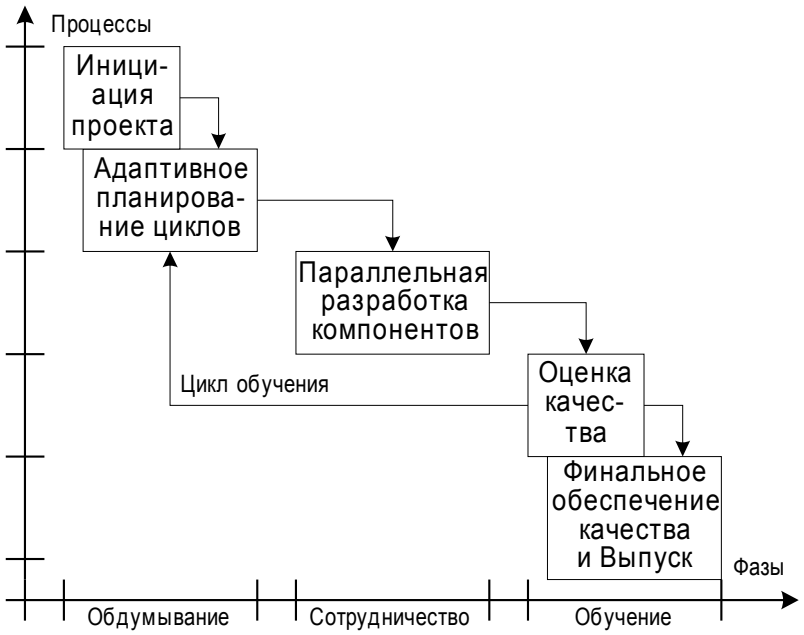


Рис.4.17. Модель ЖЦ для подхода ASD

На фазе 1 «Обдумывание» выполняются два процесса:

1. Инициация проекта.
2. Адаптивное планирование циклов.

Инициация проекта лишь немного отличается от начальной фазы других подходов. Она включает в себя следующие действия:

1. Определение цели и задач проекта.
2. Выявление и краткое описание ограничений и требований к системе.
3. Изучение расстановки сил в проекте (организация проекта и команды).
4. Первоначальная оценка размера и масштабности проекта.
5. Определение ключевых рисков.
6. Установление временных рамок для всего проекта.

Все расчёты являются предварительными и в дальнейшем могут измениться.

Адаптивное планирование циклов состоит из следующих действий:

1. Определение оптимального числа циклов и временных рамок каждого из них.
2. Определение цели и задач для каждого цикла разработки.
3. Соотнесение компонентов системы с циклами разработки.
4. Планирование циклов с учётом разрабатываемых компонентов.

Распределение компонентов по циклам — непростая задача. Главным критерием здесь является поставка заказчику в конце каждого цикла некоторой видимой, осязаемой, работающей части системы. К другим критериям относят следующие:

1. Первоочередная разработка компонентов с высокой степенью риска.
2. Учёт естественных зависимостей между компонентами.
3. Балансирование расходов различных используемых ресурсов.

На фазе 2 «Сотрудничество» выполняется один процесс — Параллельная разработка компонентов,— включающий отдельные параллельно реализуемые действия по разработке каждого запланированного на текущий цикл компонента.

При этом руководителей больше должно беспокоить то, как добиться наиболее эффективного взаимодействия и обеспечить согласованность выполняемых работ, нежели вопросы проектирования, тестирования и кодирования.

На фазе 3 «Обучение» выполняются два процесса:

1. Оценка качества.
2. Финальное обеспечение качества и Выпуск.

Адаптивность в АРП заключается в следующем: нужно определить цель и масштаб проекта, показать команде, какие компоненты ей необходимо разработать, а затем отойти в сторону и дать разработчикам самим решать, как они будут это делать. Чувство ответственности в команде поддерживается при помощи периодической оценки качества. В таком случае качество будет расти не благодаря дотошному контролю, а благодаря формированию соответствующих критериев для выпускаемого продукта и критическому его анализу. Постоянный анализ и оценка проделанной работы — ключ к обучению.

В конце каждого цикла разработки нужно знать:

1. Качество продукта с точки зрения заказчика.
2. Качество продукта с технической точки зрения.
3. Работоспособность команды и используемость практик.
4. Текущее положение дел в проекте (статус проекта).

Промежуточные контрольные точки в конце каждого цикла призваны обеспечить доступность и обозримость создаваемого продукта для участников проекта. Без этого невозможно увидеть и исправить различные дефекты, которые присутствуют в любом проекте.

В конце цикла заказчик получает определённый набор компонентов системы, которые он должен просмотреть и оценить. Это позволяет ему реально увидеть и опробовать разрабатываемую систему.

Для интеграции отдельных компонентов системы в течение каждого из циклов служат промежуточные сборки. Они позволяют увидеть и опробовать систему самой команде.

Экстремальное программирование (XP)

Экстремальное программирование (ЭП, XP — eXtreme Programming) — живой подход, предложенный Кентом Беком (Kent Beck).

Обзор подхода

Возникновение ЭП тесно связано с выполнением проекта C3 (Chrysler Comprehensive Compensation — букв. Всеобъемлющее компенсирование для Chrysler) — разработка системы учёта выплат работникам фирмы Daimler Chrysler. Суть проекта заключалась в разработке единой системы вместо множества разрозненных приложений. Именно на этом проекте К. Беком отрабатывались практики, лёгшие в основу подхода. В создании ЭП и работе над проектом приняли также участие Уорд Каннингем (Ward Cunningham) и Рон Джефрис (Ron Jeffries). В этот же период 5 из 20 участников проекта (в том числе и указанные выше автор и два соавтора) опубликовали несколько книг и множество статей, посвящённых ЭП.

В 1996 г. К. Бек стал руководить проектом, но в 2000 г. фирма прервала проект. Полученная к этому моменту система использовалась только для 10 тысяч человек из 87 тысяч работников. Поэтому успешный на словах проект на самом деле оказался провальным. Главной причиной этого является не сам подход, а его применение к неподходящему проекту. Для реализации проекта C3 (разра-

ботки формализуемой системы) необходимо было использовать один из строгих подходов.

В ЭП гораздо больше спорных моментов, чем в каком-либо другом подходе.

Ключевой и основополагающей деятельностью в ЭП считается кодирование, поэтому подход во многом использует идею непланируемой модели — «кодирование — исправление»,— расширяя её принципами эффективной организации работ. Эти принципы доводятся до крайности их реализации в используемых практиках.

При этом практики ЭП оказываются слишком тесно взаимосвязанными, что приводит к излишней жёсткости подхода. Практики считаются фиксированной частью, в то время как ЖЦ системы — адаптивной частью ЭП, что противоречит логике разработки и приводит к проблемам в проектах, использующих этот подход.

В 1999 г. вышло первое издание К. Бека «Экстремальное программирование в объяснении: Избирание изменения» («Extreme Programming Explained: Embrace Change»), а в 2004 г. — второе издание этой книги в соавторстве с Цинтией Андрес (Cynthia Andres), переработанное с учётом пятилетнего опыта применения ЭП. Для различения особенностей ЭП из первого и второго изданий обозначим ЭП этих изданий соответственно ЭП(1) и ЭП(2).

Категория: Ценности

ЭП представляется в виде тройки категорий, с помощью которых описываются стратегии, используемые в этом подходе: ценности (values), принципы (principles) и практики (practices). Ценности выражают общую направленность ЭП и конкретизируются в принципах, соответствие которым служит мерой приемлемости и отбора практик для этого подхода.

В ЭП(1) выделено 4 ценности:

1. Общение (Communication) для своевременного отражения изменений.
2. Простота (Simplicity) для реализации только текущих изменений.
3. Обратная связь (Feedback) для получения текущего состояния системы.
4. Мужество (Courage) для внесения существенных изменений.

В ЭП(2) добавлена пятая ценность, неявно присутствовавшая в ЭП(1):

5. Уважение (Respect) для эффективного взаимодействия членов команды.

Несмотря на общность ценностей, ЭП(1) и ЭП(2) во многом различаются в принципах и практиках.

Категория: Принципы

В ЭП(1) выделено 5 основных (basic) принципов:

1. Быстрая обратная связь (Rapid feedback) — эффективное получение реакции системы и заказчика на внесённые изменения.
2. Предполагаемая простота (Assume simplicity) — выбор наиболее простого решения, соответствующего текущей ситуации.
3. Постепенное изменение (Incremental change) — пошаговое внесение незначительных изменений в систему.
4. Избираемое изменение (Embracing change) — выбор решения наиболее актуальной проблемы, сохраняющего свободу действий.
5. Качественная работа (Quality work) — успешность выполнения работы с учётом заинтересованности разработчика.

Кроме этого, там же описаны 10 вспомогательных (specific) принципов:

1. Учение обучению (Teach learning) — приоритет обучения действию перед директивным управлением.
2. Небольшое начальное инвестирование (Small initial investment) — использование наименьшего приемлемого объёма ресурсов в начале проекта.
3. Игра на выигрыш (Play to win) — выполнение проекта с настроем на успешность его завершения.
4. Конкретные эксперименты (Concrete experiments) — проверка правильности решения путём проведения экспериментов.
5. Открытое честное общение (Open, honest communication) — готовность к обсуждению проблем и причин выбора решений при отсутствии наказания.
6. Работа в соответствии с человеческими инстинктами, а не против них (Work with people's instincts, not against them) — учёт краткосрочных интересов людей при решении долгосрочных проблем проекта.

7. Принимаемая ответственность (Accepted responsibility) — осознанное принятие ответственности участником команды.
8. Локальная адаптация (Local adaptation) — приспособление процесса разработки к организации и проекту.
9. Движение налегке (Travel light) — использование минимального набора простых и ценных средств в процессе разработки (т.е. кода и тестов).
10. Честные измерения (Honest measurement) — использование метрик, соответствующих уровню детализации.

В ЭП(2) выделено 14 принципов:

1. Человечность (Humanity) — человеческий фактор (потребности команды: базовая безопасность, благоустройство, принадлежность и близость).
2. Экономика (Economics) — экономическая ценность продукта, проявляемая в двух аспектах: текущая денежная ценность (time value of money) и вариантная ценность системы (option value of system).
3. Обоюдная выгода (Mutual Benefit) — сотрудничество, выгодное для всех заинтересованных лиц.
4. Самоподобие (Self-Similarity) — использование одних и тех же правил в разных контекстах разработки системы.
5. Улучшение (Improvement) — постоянное развитие системы и продвижение её разработки.
6. Разнообразие (Diversity) — неоднородность участников разработки со способностью разрешать конфликтные ситуации.
7. Размышление (Reflection) — обсуждение процесса разработки и состояния проекта в целом.
8. Поток (Flow) — постоянная размеренная разработка качественного ПО.
9. Возможность (Opportunity) — изучение проблем с целью получения знаний поиска новых решений.
10. Избыточность (Redundancy) — решение проблем несколькими способами одновременно для выбора приемлемого способа.

11. Неудача (Failure) — использование безуспешных решений как ценных уроков.
12. Качество (Quality) — работа над качеством системы, способствующая улучшению её разработки.
13. Детские шажки (Baby Steps) — быстрое внесение малых изменений в систему, позволяющее сохранить направление разработки и уменьшить проблемы.
14. Принимаемая ответственность (Accepted Responsibility) — самостоятельное принятие решений участником разработки о несении ответственности.

Следует отметить, что принципы ЭП(2) в большей степени соответствуют живой разработке ПО, чем принципы ЭП(1).

Категория: Практики

Различие принципов приводит к различию состава и содержания используемых практик в ЭП(1) и ЭП(2), хотя между ними присутствует взаимосвязь.

Для удобства рассмотрения практики объединены в следующие группы:

1. Исследование и Планирование.
2. Организация команды.
3. Реализация.
4. Продуцирование.

Названия групп связаны с командой и фазами разработки (см. ниже).

В ЭП(1) включено 12 практик:

— Исследование и Планирование:

1. Игра в планирование (The planning game) — быстрое определение задач для реализации в очередном выпуске.
2. Метафора [системы] (Metaphor) — простая и понятная иллюстрация функционирования системы (упрощённый аналог архитектуры системы).
3. Заказчик на месте [разработки] (On-site customer) — наличие представителя заказчика в команде.

— Организация команды:

4. Выдержанный темп [разработки] (Sustainable pace), др. назв. — 40-часовая [рабочая] неделя (40-hour week) — жёсткое ограничение сверхурочных работ (не более двух недель подряд).
5. Парное программирование (Pair programming) — написание кода двумя программистами, сидящими за одним компьютером.

— Реализация:

6. Простой дизайн (Simple design) — поддержка дизайна системы в максимально упрощённом виде.
7. Рефакторинг (Refactoring) — изменение структуры (переработка) кода без изменения функциональности системы.
8. Стандарты кодирования (Coding standards) — следование общим правилам и рекомендациям относительно написания кода.
9. Тестирование (Testing) — постоянное написание тестов для работоспособности вносимых изменений и системы в целом.
10. Коллективное владение [кодом] (Collective ownership) — возможность изменения любым членом команды произвольной части кода системы.

— Продуцирование:

11. Непрерывная интеграция (Continuous integration) — собирание системы множество раз в день после решения каждой задачи.
12. Малые выпуски [продукта] (Small releases) — быстрое внедрение выпусков, включающих небольшие изменения.

ЭП(2) включает в себя 13 основных (primary — первичные) практик, предназначенных для улучшения процесса разработки:

— Исследование и Планирование

1. Истории (Stories) — изложение функциональности системы с точки зрения заказчика в виде небольших рассказов.
2. Слабина (Slack) — включение в план задач, от выполнения которых при появлении проблем можно отказаться.

3. Недельный цикл (Weekly cycle) — малый цикл планирования с выборкой заказчиком реализуемых историй в начале недели.
4. Квартальный цикл (Quarterly cycle) — большой цикл планирования с обсуждением работы команды и темпов разработки.

— Организация команды:

5. Сидение вместе (Sit together) — работа команды в одном большом рабочем помещении для облегчения общения.
6. Цельная команда (Whole team) — команда разработчиков с необходимыми для проекта знаниями и навыками, сообща выполняющая общее дело.
7. Информативное рабочее пространство (Informative workspace) — заполнение рабочего помещения информацией о состоянии проекта и проделанной командой работе.
8. Энергичная работа (Energized work) — сохранение разработчиками свежести и энергичности для фокусировки на задачах и эффективном их решении путём ограничения сверхурочных работ.
9. Парное программирование (Pair programming) — написание кода двумя программистами, сидящими за одним компьютером, для повышения качества получаемого кода.

— Реализация:

10. Инкрементное проектирование (Incremental design) — выполнение проектирования постепенно во время кодирования.
11. Первично-тестовое программирование (Test-first programming) — написание тестов до начала кодирования для повышения качества программирования.

— Продуцирование:

12. Непрерывная интеграция (Continuous integration) — включение каждые 2 часа в репозиторий результатов работы каждого разработчика для правильной интеграции системы с учётом его изменений.

13. Десятиминутная сборка (Ten-minute build) — сборка системы с учётом выполнения всех тестов за 10 минут для частых сборок и быстрого получения отзывов от заказчика.

Кроме этого выделено 11 дополнительных (corollary — выводимые) практик, предназначенных к применению только после освоения основных практик:

— Исследование и Планирование

1. Контракт с обсуждаемым содержанием [работ] (Negotiated scope contract) — определение точного содержания работ путём заключения серии отдельных небольших контрактов.
2. Плата за использование (Pay-per-use) — оплата заказчиком каждого выпуска продукта для получения требуемой функциональности.
3. Реальное вовлечение заказчика (Real customer involvement) — участие представителей заказчика в работе команды при планировании.

— Организация команды:

4. Неразрывность команды (Team continuity) — постоянство состава команды на протяжении нескольких проектов.
5. Сжимающиеся команды (Shrinking teams) — выделение свободных членов эффективной команды для создания ими собственных команд.

— Реализация:

6. Анализ первопричин (Root-cause analysis) — исправление не только ошибок, но и их причин, чтобы исключить повторение аналогичных ситуаций.
7. Код и тесты (Code and tests) — сохранение программного кода и тестов — постоянных артефактов системы (остальное получается из кода и тестов).
8. Единая база кода (Single code base) — существование только одной официальной версии разрабатываемой системы (остальные варианты должны использоваться лишь в течение нескольких часов).
9. Разделяемый код (Shared code) — возможность изменения любым членом команды произвольной части кода системы.

— Продуцирование:

10. Ежедневное развёртывание (Daily deployment) — сборка каждую ночь нового выпуска системы и ввод её в действие для исключения проблем с совместимостью с используемым предыдущим выпуском.
11. Инкрементное развёртывание (Incremental deployment) — постепенная замена функций системы для сохранения её работоспособности.

Изменения в практиках связаны с добавлением ряда новых практик, разделением ряда практик на несколько новых, объединением двух практик в одну, переименованием трёх практик и исключением двух практик (рис.4.18, сплошные и пунктирные стрелки и перечёркивания).

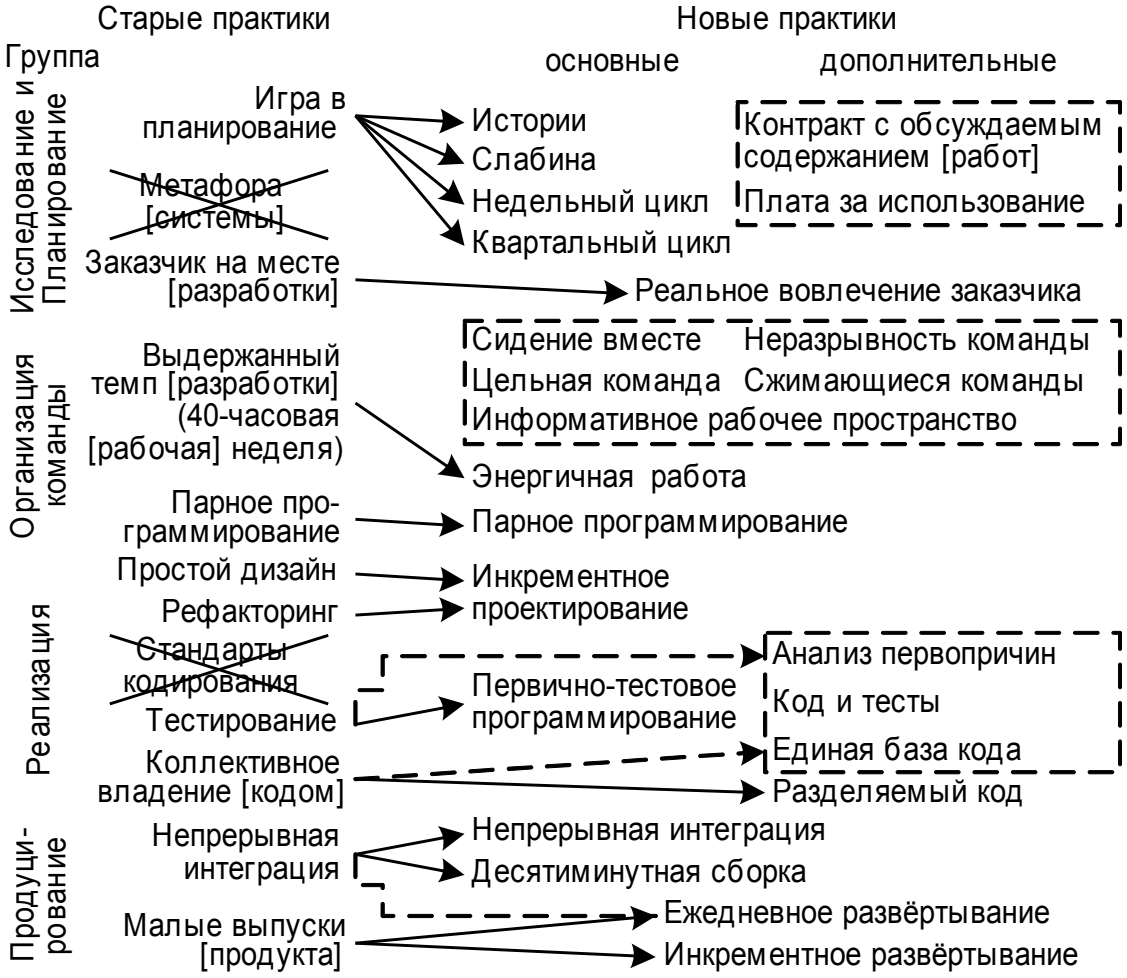


Рис.4.18. Сопоставление старых и новых практик ЭП

Сопоставление старых и новых практик позволяет сделать вывод о развитии подхода в направлении соглашения с заказчиком, организации команды и выполнения кодирования (рис.4.18, пунктирные прямоугольники).

Жизненный цикл проекта

В ЭП(1) рассматривается модель ЖЦ идеального проекта для ЭП (рис.4.19).

Согласно этой модели выделено 5 фаз ЖЦ:

1. Исследование (Exploration).
2. Планирование (Planning).
3. Реализация / Итерации (Implementation / Iterations).
4. Продуцирование / Обслуживание (Productionizing / Maintenance).
5. Смерть (Death).

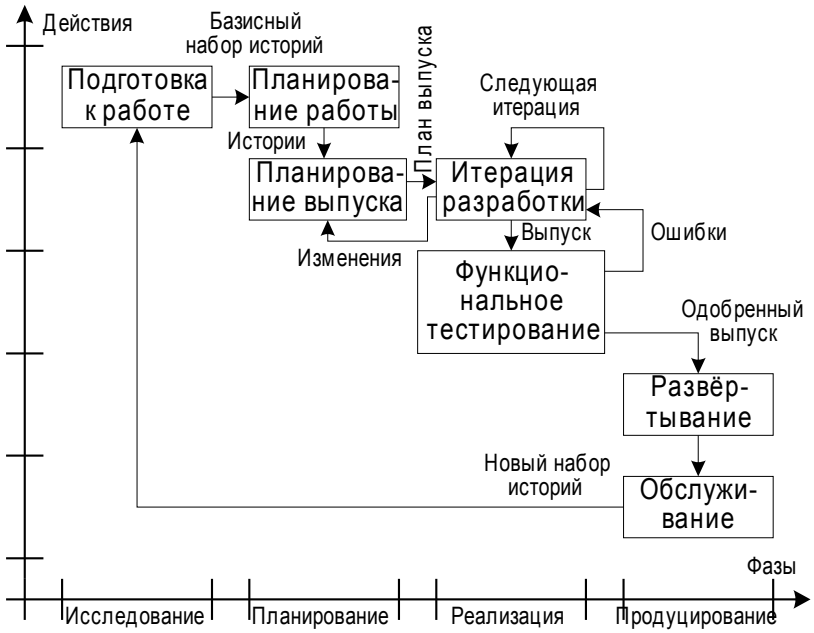


Рис.4.19. Схема модели ЖЦ для подхода XP

На фазе 1 «Исследование» выполняется предварительная подготовка к работе, в рамках которой можно выделить следующие операции:

1. Команда выбирает инструменты, осваивает необходимые знания и навыки, анализирует варианты архитектур системы, оценивает будущие задачи.
2. Заказчик готовит истории, сразу же обсуждаемые командой, формируя из них базисный набор историй, который будет реализован в выпуске продукта.

На фазе 2 «Планирование» выполняются планирование работы по созданию системы и планирование текущего выпуска продукта. Планирование работы предполагает в частности соглашение о сроках поставки первого выпуска. Планирова-

ние выпуска основано на отборе приоритетных историй из базисного набора, которые будут реализованы в очередном выпуске.

Планирование работы и выпуска основано на наблюдении за четырьмя ограничениями, названными в ЭП переменными (variables): стоимость, время, качество и содержание. Заказчиком должно фиксироваться не более трёх переменных, а команда выбирает результирующие значения остальных переменных.

На фазе 3 «Реализация» выполняются итерации разработки и функциональное (приёмочное) тестирование системы. Итерации разработки предназначены для реализации историй в рамках плана выпуска и формирования функциональных тестов. В ходе первой итерации формируется общий дизайн, для которого отбираются подходящие истории, для остальных итерации отбор историй выполняется заказчиком. Функциональное тестирование позволяет получить одобренный заказчиком выпуск, включающий истории с учётом плана выпуска.

На фазе 4 «Продуцирование» выполняются развёртывание системы в реальной среде эксплуатации и её обслуживание. Во время развёртывания и обслуживания системы заказчик может сформировать новый набор историй для их включения в следующий выпуск продукта.

На фазе 5 «Смерть» завершается эксплуатация системы. Это может произойти по двум причинам:

1. У заказчика нет историй для новых выпусков продукта.
2. Система находиться в плохом состоянии из-за большого числа дефектов.

На протяжении этих фаз ЖЦ выполняются 4 деятельности (activity), связанные с программированием (написанием кода):

1. Кодирование (Coding) — программирование в текстовом или графическом виде, для которого возможна трансляция во внутреннее представление. Исходный код — это артефакт и средство выражения идеи и обмена информацией.
2. Тестирование (Testing) — создание тестов для увеличения ЖЦ системы, получения уверенности в её работоспособности, проверки корректности кода и проверки историй заказчика. Тесты — это ресурс и ответственность.

3. Слушание (Listening) — организация правильного общения между командой и заказчиком для получения правильной и полной информации о системе.
4. Проектирование (Designing) — создание и развитие структуры, которая организует логику в системе, т.е. формирование дизайна системы. В отличие от других подходов выполняется непосредственно при кодировании.

   Основой для всех деятельностей является программный код системы.

4.5. Генетические технологические подходы

Генетические подходы (genetic approaches) являются строгими подходами. Они основаны на аналогии разработки ПО и различных (в частности, биологических) процессов происхождения.

Выделяют генетические подходы следующих трёх видов:

1. Синтезирующее программирование.
2. Конкретизирующее программирование.
3. Сборочное программирование.

Перечисленные виды подходов оказываются тесно связанными с методологиями разработки ПО:

1. Подход 1 ориентирован на методологии императивного (в том числе структурного и параллельного) программирования и ООП.
2. Подходы 2 и 3 ориентированы на специальные методологии, которые рассмотрены при описании указанных видов генетических подходов.
3. Методологии логического и функционального программирования могут лежать в основе подходов 1 и 2.
4. Методология сентенциального программирования лежит в основе подхода 2.
5. Методологии ограничительного, событийного и автоматного программирования могут лежать в основе подходов любых видов.

Эти виды подходов предназначены для облегчения разработки путём использования уже имеющихся частично формализованных или готовых элементов представления программного кода.

Фактически генетические подходы являются облегчённой версией соответствующих формальных генетических подходов (см. §4.6). Поэтому ряд положений, относящихся к формальным генетическим подходам, оказывается справедливым и для (обычных) генетических подходов. В частности на практике генетические подходы также используются не по отдельности, а в комбинации. Кроме того, возможно сочетание обычных и формальных генетических подходов.

Синтезирующее программирование

Синтезирующее программирование (synthesizing programming) — вид подходов, предполагающих синтез программы по её спецификации (в общем понимании — по постановке проблемы и методу её решения).

В отличие от программы на алгоритмическом языке (реализации) документ на языке спецификаций является лишь базисом для последующей реализации.

Для получения программы необходимо решить следующие основные задачи:

1. Доопределить детали, которые нельзя выразить при помощи языка спецификации, но необходимые для получения кода программы.
2. Выбрать алгоритмический язык реализации, а также программно-аппаратную платформу для реализации.
3. Зафиксировать отображение понятий языка спецификаций на язык реализации, а также программно-аппаратную платформу.
4. Осуществить трансформацию представления — преобразование из спецификации в код программы на языке реализации.
5. Протестировать полученный код программы.

Существует ряд подходов различного уровня применения в процессах ЖЦ, которые относятся к синтезирующему программированию и связаны с соответствующим языком спецификации.

К таким языкам спецификации относятся следующие:

— UML (Unified Modeling Language — Универсальный язык моделирования, УЯМ) — язык спецификации для объектно-ориентированной разработки ПО.

— SDL (Specification and Description Language — Язык спецификации и определения, ЯСО) — язык для однозначного специфицирования и описания поведения реактивных (реагирующих) и распределённых систем.
— ASN.1 (Abstract Syntax Notation One — Абстрактная синтаксическая нотация Один, АСН.1) — стандартный гибкий язык описания структур данных для представления, кодирования, передачи и декодирования данных.

Автоматическая генерация программ возможна для многих языков спецификаций, в частности для перечисленных выше языков.

Конкретизирующее программирование

Конкретизирующее программирование (concretizing programming) — вид подходов, предполагающих извлечение программы из существующей универсальной программы путём конкретизации общих элементов.

Существует много подходов различного уровня применения в процессах ЖЦ, которые относятся к конкретизирующему программированию.

Среди них следует отметить следующие подходы:

1. Обобщённое программирование.
2. Подход на основе паттернов и анти-паттернов.
3. Подход на основе архитектурных стилей.

Рассмотрим кратко суть этих подходов.

Обобщённое программирование (generic programming) — подход, ориентированный на написание алгоритмов на уровне языка программирования, применимых к различным типам данных. Этот подход основан на использовании шаблонов. *Шаблон* (template) — фрагмент кода, настраиваемый на конкретное требуемое представление при трансляции или выполнении программы.

Подход на основе паттернов или анти-паттернов ориентирован на соответственно применение или избегание некоторых решений, уже опробованных в различных проектах.

Паттерн (pattern — образец, шаблон) — повторно используемое решение для часто встречающейся проблемы в определённом контексте. В настоящее

время существуют паттерны для многих областей деятельности, но в области разработки они были классифицированы для облегчения проектирования систем.

Анти-паттерн (anti-pattern — антиобразец, антишаблон), или *ловушка* (pitfall, тж. западня) — часто встречающееся неправильно используемое решение для некоторой проблемы в определённом контексте. Как паттерн является примером хорошего процесса разрешения проблемы, так и анти-паттерн является примером плохого процесса, откуда и произошло исходное название этого решения.

Подход на основе архитектурных стилей ориентирован на первоначальное определение архитектуры разрабатываемой системы.

Архитектурный стиль (architecture style) — это набор архитектурных структур, ориентированный на разработку системы для конкретной ПрО. Мэри Шоу (Mary Shaw) и Дэвид Гарлан (David Garlan) предложили использовать каталог архитектурных стилей для их идентификации и документирования, а также облегчения использования в дальнейшем на практике.

Каждый архитектурный стиль обычно имеет определённое концептуальное представление — *архитектурный паттерн* (architectural pattern), и связанное с некоторой методологией разработки реализационное представление — *архитектурный каркас* (architectural framework).

При этом первый и третий подходы можно рассматривать как определённые разновидности второго подхода. Поэтому некоторые рассматривают эти подходы как соответствующие своей определённой методологии, используя для неё название *шаблонно-ориентированное программирование* (pattern-oriented programming).

Сборочное программирование

Сборочное программирование (assembly programming) — вид подходов, предполагающих сборку программы из уже разработанных фрагментов.

Сборка может осуществляться вручную, указываться на некотором языке сборки или извлечена полуавтоматическим образом из спецификации задачи. Фрагментами кода обычно выступают программные модули или иные части программы, представляемые в виде модулей определённого вида.

Выделяют следующие основные способы поддержки этого подхода:

1. Выработка стиля программирования, соответствующего принятым принципам модульности.
2. Повышение эффективности межмодульных интерфейсов. Обеспечение поддержки модульности на уровне программно-аппаратной платформы.
3. Ведение большой базы программных модулей. Решение проблемы идентификации модулей и проверки пригодности по описанию интерфейса.

Сборочное программирование тесно связано с методом повторного использования кода, причём как исходного, так и бинарного. Выделяют несколько разновидностей технологических подходов сборочного программирования, которые в значительной степени определяются базисной методологией:

1. Модульное сборочное программирование.
2. Объектное сборочное программирование.
3. Компонентное сборочное программирование.
4. Аспектное сборочное программирование.

Каждый из подходов (после модульного программирования) можно рассматривать как развитие предыдущего подхода в направлении решения различных проблем разработки. Особенно это важно для программирования, так как многие проблемы связаны с поддержкой и улучшением программного кода системы.

Рассмотрим кратко суть этих подходов.

Модульное сборочное программирование

Модульное сборочное программирование (module assembly programming) — исторически первый подход сборочного программирования, базирующийся на процедурах и функциях методологии структурного императивного программирования, точнее их объединении — программных модулях. В разных языках программные модули называются по-разному: модуль (module в Modula-2, unit в Pascal), пакет (package в Ada) или просто отдельный файл (в C/C++ и т.п.).

Некоторые рассматривают указанную методологию, дополненную концепцией модуля, как полноправную, самостоятельную, используя для неё название *модульно-ориентированное программирование* (module-oriented programming), под-

чёркивая этим выделение *модульности* (modularity) как отдельной специфики, а не части топологической специфики.

Развитием этого подхода является *расширяемое программирование* (extensible programming) — модульно-основанное программирование, при котором добавление новых модулей возможно без каких-либо изменений в существующих модулях. Данный подход предложен Н. Виртом и впервые реализован при проектировании Oberon System.

Фактически это компонентно-основанное программирование без ООП, так как модуль в Oberon является полноценным компонентом (т.е. выполняет соответствующие ему функции). Развитием Oberon стал Component Pascal, в своём названии отразивший своё происхождение от языка Pascal и свою нацеленность на компонентное программирование.

Объектное сборочное программирование

Объектное сборочное программирование (object assembly programming) — подход сборочного программирования, базирующийся на библиотеках классов ООП, поставляемых в виде исходного (source code, например, VCL в Delphi) или бинарного кода (binary code, например, DLL в C/C++ для MS Windows).

Как следует из названия, подход соответствует методологии ООП.

Компонентное сборочное программирование

Компонентное сборочное программирование (component assembly programming) — развитие предыдущего подхода, базирующееся на библиотеках компонентов. Данный подход связан с такими понятиями, как компонент и интерфейс.

Компонент (component) — класс, доступ к которому обеспечивается через строго определённые интерфейсы. Под *интерфейсом* (interface) понимается набор средств и правил для обеспечения единообразного взаимодействия.

Использование интерфейсов компонентов вместо непосредственного доступа к объектам позволяет снять проблему несовместимости компиляторов и обеспечивает смену версий компонентов без перекомпиляции основанного на них ПО.

Наиболее известными примерами реализации этого подхода являются COM (DCOM, COM+), CORBA, .NET.

Некоторые рассматривают этот подход как соответствующий своей определённой методологии, используя для неё название *компонентно-ориентированное программирование* (КОП, COP — component-oriented programming). Эта методология является дальнейшим развитием ООП. Её можно рассматривать как методологию, полученную применением специфики модульности к ООП.

Аспектное сборочное программирование

Аспектное сборочное программирование (aspect assembly programming) — развитие предыдущего подхода, базирующееся на библиотеках многоаспектных компонентов. Данный подход связан с такими понятиями, как аспект и значимость.

Аспект (aspect) — модуль, который содержит реализацию определённой значимости. *Значимость* (concern — букв. участие, отношение) — эксплуатационная часть системы, несущая определённую характеристику (в отличие от основной части, ориентированной на ПрО). Значимость связана с определёнными эксплуатационными требованиями, предъявляемыми к системе (надёжность, контролируемость, безопасность).

Примерами значимостей являются обработка контекстно-зависимых ошибок и исключений, ведение журналов работы с системой (протоколирование), выполнение контрольных проверок параметров, подтверждение подлинности (аутентификация).

Существенные проблемы связаны с реализацией пересекающих значимостей. *Пересекающая значимость* (cross-cutting concern, тж. сквозная, перекрёстная, секущая значимость) — это значимость, связанная с написанием кода, при обычном программировании сложно взаимосвязанного и рассредоточенного по всем модулям системы, т.е. «пересекающего» весь программный код.

Выделение пересекающей значимости в аспект позволяет:

— в полной мере использовать модульность при реализации всех требований,

— естественным образом разделить ответственность членов команды.

Некоторые рассматривают этот подход (аналогично КОП) как соответствующий своей определённой методологии, используя для неё название *аспектно-ориентированное программирование* (АОП, АОР — aspect-oriented programming). Эта методология является дальнейшим развитием ООП и КОП. Её можно рассматривать как методологию, полученную повторным применением специфики модульности к КОП на более высоком уровне организации модульности.

4.6. Формальные технологические подходы

Формальные подходы (formal approaches) основаны на применении математических принципов к разработке ПО.

Выделяют формальные подходы следующих трёх видов:

1. Формальные генетические подходы: формальное синтезирующее программирование, формальное конкретизирующее программирование и формальное сборочное программирование.
2. Подходы формальной разработки: подходы на основе языков формальной спецификации, исчисление процессов и т.п.
3. Смешанные формальные подходы: Инженерия стерильного цеха (CrSE).

Подходы формальной разработки фактически являются комбинацией и/или развитием идей и принципов формальных генетических подходов. Смешанные формальные подходы используют ряд принципов формальной разработки в рамках часто используемых моделей ЖЦ.

Высокая трудоёмкость формальных подходов в настоящее время существенно ограничивает область их применения на практике, поэтому для них справедливы те же рекомендации по классам систем, что и для каскадных подходов.

Эти виды подходов предназначены для обеспечения разработки путём применения математического формализма в спецификациях и процессах ЖЦ.

Формальные генетические подходы

Формальные генетические подходы (formal genetic approaches) основаны на аналогии разработки ПО и вывода математических утверждений.

Проблемами доказательств (правильности) программ занимались многие отечественные и зарубежные исследователи. Под правильностью ПО здесь понимается отсутствие в нём любых дефектов.

В 1975 г. Эдгар Вайб Дейкстра (Edgar W. Dijkstra) в работе «Охраняемые команды, недетерминированность и формальное порождение программ» («Guarded Commands, nondeterminancy and the formal derivation of programs») высказал идею и сформулировал свои положения по доказательству программ. Эти положения послужили основой его известной книги «Дисциплина программирования» («Discipline of Programming»), изданной в 1976 г. В 1981 г. Дэвид Грис (David Gries) опубликовал адаптацию этой книги к учебному процессу под названием «Наука программирования» («The Science of Programming»).

В 1984 г. А.П. Ершов (Андрей Петрович Ершов) в своём докладе (см. Труды Академии наук СССР) предложил термин «доказательное программирование». *Доказательное программирование* (proof programming) — разработка ПО, обладающая свойством доказательности правильности создаваемого продукта.

Кроме того, А.П. Ершов в этом же докладе указал три вида доказательного программирования, которые тесно связаны с соответствующими видами генетических подходов (см. §4.5). Поэтому эти виды получили в дальнейшем название формальных генетических подходов.

Выделяют следующие формальные генетические подходы:

1. Формальное синтезирующее программирование.
2. Формальное конкретизирующее программирование.
3. Формальное сборочное программирование.

Перечисленные подходы ориентированы в основном на решение проблем вычислительного характера и часто применяются в комбинации. Фактически генетические подходы являются упрощением этих подходов с целью существенного расширения ПрО и ускорения разработки ПО.

Приведём краткий обзор формальных генетических подходов.

Формальное синтезирующее программирование

Формальное синтезирующее программирование основывается на применении *математической спецификации* — совокупности логических формул, представляющих постановку проблемы.

Под *синтезом программы* понимается разработка этой программы по её математической спецификации. В случае явно существующего разделения постановки проблемы и метода её решения синтез заключается в правильном представлении метода решения в виде программы. Однако на практике метод решения проблемы присутствует лишь неявно — в постановке этой проблемы (что дано и что надо определить) и её ПрО (особенности проблемы в виде соотношений или условий). Тогда синтез заключается в извлечении метода решения с помощью некоторой систематической процедуры для его дальнейшего воплощения в виде программы.

В зависимости от трактовки спецификации выделяют два способа синтеза программы: логический (доказательный) и аналитический (трансформационный).

При логическом способе спецификация трактуется как формулировка теоремы, утверждающей существование решения проблемы. В этом случае синтез программы состоит в поиске доказательства этой теоремы существования в некотором конструктивном логическом исчислении. Если такое доказательство удаётся построить, то существуют формальные правила для преобразования этого доказательства в программу нахождения решения. При этом существует универсальная процедура для контроля правильности доказательства и правильности применения правил перехода от доказательства к программе.

При аналитическом способе спецификация трактуется как уравнение относительно программы. В этом случае синтез программы состоит в поэтапных символических преобразованиях спецификации в программу нахождения решения. Эти преобразования заключаются в постепенной замене частей уравнения с неизвестными символами программы на систему вспомогательных неизвестных. В свою очередь, полученные неизвестные заменяются другими неизвестными или конкретными программными фрагментами. При этом существует универсальная про-

цедура для контроля правильности выполнения символических преобразований на каждом этапе синтеза.

Следует отметить, что, несмотря на кажущуюся формальность, синтезирующее программирование является творческой задачей: в первом способе необходимо найти доказательство, во втором способе — правильное направление преобразований. Из-за сложности постановок проблем и творческого выполнения поиска оба способа часто применяются в комбинации.

Исходя из вышеописанного, синтез программы можно рассматривать как способ манипулирования следующими видами знания:

— специальное знание — знание, воплощённое в постановке проблемы,
— предметное знание — знание, характеризующее предметную область,
— универсальное знание — знание, отражающее общематематические закономерности и правила доказательного рассуждения.

Слово «манипулирование» подчёркивает, что здесь используется некоторое осязаемое, документированное и точное представление знания, позволяющее проследить и проконтролировать последствия его использования.

Манипулирование знанием позволяет объединить универсальную строгость математической логики с разнообразием ПрО, проницательность человеческой интуиции с безошибочной пунктуальностью вычислительной обработки сведений.

Таким образом, формальное синтезирующее программирование основывается на творческом поиске, совокупности доказываемых логических утверждений и разнообразной формальной манипуляционной технике и использует различные сведения, образующие определённую систему — базу знаний.

В настоящее время формальное синтезирующее программирование является редко используемым подходом доказательного программирования.

Формальное конкретизирующее программирование

Формальное конкретизирующее программирование основывается на применении *универсальных программ* — программ, реализующих некоторый обобщённый метод, пригодный для решения любых проблем некоторого класса из соответствующей ПрО.

Под *конкретизацией программы* понимается разработка этой программы по её математической спецификации в случае, когда существует универсальная программа для решения более общей проблемы. Фактически конкретизация в этом понимании является противоположностью синтезу программы.

Исследование ПрО приводит к созданию теории этой ПрО. Как правило, признаком полноты такой теории является создание обобщённого метода для решения любых проблем некоторого класса. Однако на практике применение такого метода ограничивается тем обстоятельством, что по отношению к конкретной проблеме он оказывается избыточным. Поэтому наиболее эффективным считается разработка специального частного метода, учитывающего особенности конкретной проблемы. В этом случае конкретизация заключается в систематическом построении (извлечении) специализированной программы из универсальной.

С теоретической точки зрения конкретизация означает поиск программы, рассматриваемой на некотором сужении области исходных данных и получающей на этом сужении такой же результат, что и универсальная программа, но делающей это более эффективно. Сужение области исходных данных — это фактически задание некоторой дополнительной информации об этих данных.

Одним из наиболее важных видов сужений является фиксация (или связывание) параметров — задание определённых значений некоторых данных. Использование зафиксированных (связанных) параметров позволяет упростить универсальную программу. На основе фиксации параметров возможно построение систематической процедуры для конкретизации программы. Математическим аппаратом для этой операции являются смешанные вычисления, в рамках которых выявлены преобразования, необходимые для такого упрощения.

Теория показывает, что существует некоторая дисциплина последовательного применения этих преобразований, которая:

— при полностью заданных исходных данных вычисляет результат программы,

— при отсутствии заданных исходных данных оставляет программу нетронутой,

— при частичном задании исходных данных редуцирует (упрощает) программу к так называемой остаточной программе, являющейся требуемой специализированной программой.

В настоящее время формальное конкретизирующее программирование является часто используемым подходом доказательного программирования.

Формальное сборочное программирование

Формальное сборочное программирование основывается на применении повторно используемых заранее разработанных фрагментов, в роли которых выступают *программные модули*.

Здесь используется более конкретное понимание этого понятия. Модуль обладает определённой структурной и функциональной целостностью и специально приспособлен для организации чётко определяемого и контролируемого информационно-логического взаимодействия с другими модулями. Указанное взаимодействие означает обмен информацией или соподчинённость выполнения.

Под *сборкой программы* понимается разработка этой программы по её математической спецификации в случае, когда отдельные блоки решения проблемы уже разработаны и оформлены в виде программных модулей. Фактически сборка в этом понимании является частным случаем синтеза программы.

Особенностью подхода является ориентация на некоторый класс проблем, связанных с определённой ПрО. Наличие небольшого числа заранее запрограммированных модулей позволяет скомбинировать их для решения конкретной проблемы этой области. В этом случае сборка заключается в извлечении из постановки проблемы схемы сборки модулей. Для некоторых классов проблем извлечение может быть выполнено по формальным правилам.

В настоящее время формальное сборочное программирование является наиболее продвинутым и часто используемым подходом доказательного программирования.

Подходы формальной разработки

Подходы формальной разработки (formal development approaches) основаны на применении формальных методов на некотором интервале или на всём протяжении ЖЦ.

Формальные методы (formal methods) представляют собой методы разработки с применением математического аппарата, направленные на обеспечение высокой надёжности и устойчивости дизайна создаваемых систем.

Однако пока ещё высокая стоимость использования этих методов приводит к тому, что они обычно применяются лишь при разработке критических систем, для которых приоритетными свойствами являются безопасность, безотказность и защищённость.

Модели ЖЦ для подходов формальной разработки являются модификацией и/или конкретизацией описанной ниже модели трансформации.

Жизненный цикл проекта

Трансформационная модель или *Модель трансформации* (transform model) аналогична каскадной модели, но ориентирована на использование формальных методов в процессах ЖЦ.

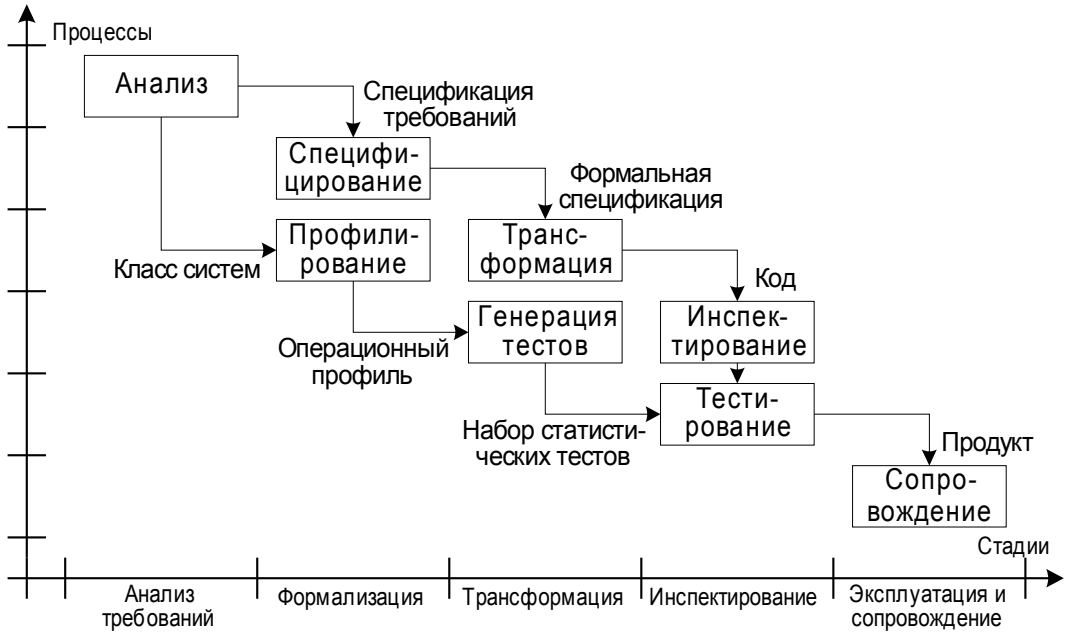


Рис.4.20. Схема трансформационной модели

Принцип модели этого подхода (рис.4.20) заключается создании формальной спецификации и её трансформации путём формальных преобразований в программный код системы.

При этом верификация и аттестация системы проводится с использованием не тестирования, а инспектирования. В отличие от тестирования инспектирование не требует выполнения кода системы. Но оно не исключает тестирования. В частности, для оценивания таких характеристик, как производительность и надёжность, применяется так называемое статистическое тестирование.

Выделенные процессы могут выполняться параллельно (перекрываться) и итерационно (повторяться), возможен возврат на любую предыдущую стадию.

Процесс «Анализ» предназначен для определения требований к системе и их представления в виде (обычной) спецификации требований. Кроме этого в этот процесс входит отнесение создаваемой системы к определённому классу систем.

Процесс «Специфицирование» заключается в формализации полученной спецификации требований в виде формальной спецификации. *Формальная спецификация* (formal specification) — это спецификация на систему, записанная на некотором формальном языке, т.е. математическое описание системы. Кроме этого, в рамках этого процесса (или отдельного параллельного процесса «Проектирование») обычно разрабатывается архитектура системы.

Процесс «Трансформация» позволяет итерационно — с помощью формальных преобразований — получить код системы из формальной спецификации.

Процесс «Инспектирование» представляет собой статическую верификацию исходного представления системы, к которому относятся спецификация требований, формальная спецификация, архитектура системы, программный код.

Кроме указанной последовательности процессов выделяют другую, параллельную ей последовательность процессов (рис.4.20).

Процесс «Профилирование» заключается в построении операционного профиля системы на основе класса этой системы. *Операционный профиль* (operational profile) — это спецификация, включающая в себя классы входных данных и вероятность их появления при нормальных режимах работы системы. Операционный

профиль обычно строится на основе анализа существующих систем данного класса с учётом новых возможностей новой системы или предположения об использовании этой системы различными группами пользователей.

Процесс «Генерация тестов» предполагает формирование набора статистических тестов, соответствующих операционному профилю системы.

Процесс «Тестирование» означает выполнение статистического тестирования системы и определения специальных характеристик этой системы по полученным результатам.

Процесс «Сопровождение» имеет стандартное назначение.

Представления системы в ЯФС

Для составления формальных спецификаций используются языки формальной спецификации (ЯФС). Они различаются по используемым представлениям для отображения структуры и поведения создаваемой системы.

Применяются следующие два представления структуры системы.

1. Алгебраическое представление (algebraic view): структура системы представляется в виде совокупности наборов операций над данными и отношений между этими операциями.
2. Модельно-теоретическое представление (model-theoretic view): структура системы представляется в виде совокупности наборов данных вместе с функциями над этими наборами.

Алгебраическое представление позволяет описать интерфейсы между подсистемами, модельно-теоретическое — сами подсистемы создаваемой системы.

Эти представления структуры системы тесно связаны со следующими представлениями поведения этой системы:

1. Свойственно-ориентированное представление (property-oriented view): поведение системы представляется в виде логической системы аксиом, свойствам которой должны удовлетворять функции системы или взаимосвязанная совокупность функций.

2. Модельно-ориентированное представление (model-oriented view): поведение системы представляется в виде модели состояний, в которой функции системы задаются изменением состояний этой системы.

Кроме этого ЯФС различают и по ряду других признаков.

Подходы формальной разработки включают в себя ЯФС, либо самостоятельные, либо созданные специально для этих подходов.

Обзор используемых подходов

Рассмотрим кратко наиболее часто используемые языки и подходы формальной разработки.

Нотация Z (Z notation) — ЯФС, названный по теории множеств Цермело — Френкеля (Zermelo — Fränkel set theory) и основанный на стандартной математической системе обозначений, используемой аксиоматической теорией множеств, лямбда-исчислением и логикой предикатов первого порядка. Предложен в 1977 г. Жаном—Реймондом Абриалем (Jean—Raymond Abrial) совместно со Стивом Шуманом (Steve Schuman) и Бертрандом Мейером (Bertrand Meyer). В дальнейшем развивался Исследовательской группой по программированию (Programming Research Group) Оксфордского университета, в которой Ж.-Р. Абриаль работал в начале 1980-х гг.

Этот язык послужил основой для многих других ЯФС (Z++, Object-Z, Alloy), подходов (B-Method) и средств (проект CZT, среда ZETA).

Язык общей алгебраической спецификации (CASL — Common Algebraic Specification Language) — общецелевой ЯФС, основанный на логике первого порядка с индукцией и поддерживающий некоторые дополнительные возможности. Разработан группой Инициатива по созданию общих каркасов (CoFI — Common Framework Initiative), целью которой является категоризация большинства существующих языков спецификации.

Существует множество разнообразных расширений языка CASL (в частности, HasCASL, CoCASL, ModalCASL, CASL-LTL, HetCASL).

B-Метод (B-Method) — подход формальной разработки, использующий формальный метод B. Этот метод базируется на языке *Абстрактно-машинная нота-*

ция (AMN — Abstract Machine Notation) — ЯФС для спецификации абстрактных машин, основанный на математической теории обобщённых подстановок. Метод B разработан Ж.—Р. Абриалем как развитие нотации Z, но является более низкоуровневым и более акцентированным на усовершенствовании кода.

На основе B реализовано множество инструментальных средств (ABTools, Atelier B, B4free, jbtools, ProB, Batcave) и платформ (BRILLIANT, Rodin).

Венский метод разработки (VDM — Vienna Development Method) — подход формальной разработки с использованием ЯФС VDM-SL (VDM Specification Language) для моделирования систем на высоком уровне абстракции. Разработан в Венской лаборатории фирмы IBM (IBM's Vienna Laboratory, откуда и название подхода) в 1970-х гг. Первоначально в виде Венского языка определения (VDL — Vienna Definition Language), а затем метаязыка Meta-IV, использовался для разработки компиляторов для языков программирования из определений этих языков.

В дальнейшем в VDM были включены различные методики и средства, разработанные на VDM-SL. В частности расширение подхода под названием VDM++ поддерживает объектно-ориентированные и параллельные системы. Для VDM и VDM++ существуют и средства разработки (VDMTools, Overture).

Исчисление процессов (Process calculi, Process calculus) или *Алгебры процессов* (Process algebras) — разнородное семейство связанных языков и подходов формального моделирования параллельных систем. Оно обеспечивает возможность высокоуровневого описания взаимодействия, коммуникации и синхронизации набора независимых процессов, в также манипуляции описаниями процессов. Обзор подходов исчисления процессов приведён ниже.

Подходы исчисления процессов

Несмотря на чрезвычайное разнообразие подходов исчисления процессов, можно выделить следующие три общие особенности:

1. Представление взаимодействия между независимыми процессами как коммуникации (передачи сообщений — message-passing), а не как модификации разделяемых переменных.

2. Описание процессов и систем с использованием небольшого набора примитивов, а также операторов для комбинирования этих примитивов.
3. Определение алгебраических законов для процессных операторов, которые позволяют манипулировать процессными выражениями с помощью эквациональных рассуждений.

Самыми известными до сих пор подходами исчисления процессов являются CCS, CSP и ACP.

Исчисление взаимодействующих систем (CCS — Calculus of Communicating Systems) — подход формальной разработки в рамках исчисления процессов, тесно связанный с исследованием теоретических положений и практических методов в области организации, взаимодействия и эквивалентности процессов. Создан Робином Милнером (Robin Milner, A.J.R.G. Milner) между 1973 г. и 1980 г. (изложен в книге в 1980 г.). Под взаимным влиянием CCS и CSP для обоих подходов был разработан ряд усовершенствований.

На CCS реализовано несколько языков (CBS, LOTOS), кроме того для изучения CCS-подобных систем применяются такие модели, как моноид истории (History monoid) и модель актёра (Actor model).

Взаимодействующие последовательные процессы (CSP — Communicating Sequential Processes) — подход для описания образцов взаимодействия в параллельных системах. Предложен (в виде параллельного языка программирования) Тони Хоаром (Tony Hoare, C.A.R. Hoare) в статье 1978 г. В дальнейшем автор совместно со Стивеном Бруксом (Stephen Brookes) и А.В. Роско (A.W. Roscoe) развили этот язык в форму исчисления процессов и изложили статье в 1984 г. Т. Хоар изложил современную форму CSP в своей книге в 1985 г. Теория CSP до сих пор остаётся предметом активных исследований.

Ряд языков и подходов являются производными CSP (Timed CSP, RPT, CSPP, HCSP) или интегрируют его возможности с другими языками и подходами (TCOZ, Circus, CspCASL).

Наиболее известным CSP—средством является программа проверки модели (model checking) FDR (Failures / Divergence Refinement — букв. Очищение

от сбоев / отклонений) — коммерческий продукт, разработанный фирмой Formal Systems Ltd. Другие CSP—средства: ProBE, ARC и Casper.

Алгебра взаимодействующих процессов (ACP — Algebra of Communicating Processes) — алгебраический подход к обоснованию параллельных систем. Разработан Жаном Бергстрой (Jan Bergstra) и Жаном Уиллемом Клопом (Jan Willem Klop) в 1982 г. Разработка ACP связана с созданием абстрактной обобщённой аксиоматической системы для процессов (отсюда и понятие «алгебра процессов»).

Подход ACP послужил основой для разработки нескольких подходов для описания и анализа параллельных систем (μCRL, mCRL2, HyPA).

Современными подходами исчисления процессов являются π-исчисление (π-calculus), исчисление среды (ambient calculus), PEPA (Performance Evaluation Process Algebra — Алгебра процессов для оценки производительности), исчисление слияния (fusion calculus).

Инженерия стерильного цеха (CrSE)

Инженерия стерильного цеха или *Стерильно-цеховая инженерия ПО* (СцИП, CrSE, Cleanroom SE — Cleanroom Software Engineering) — подход формальной разработки, предложенный сотрудником фирмы IBM Харланом Д. Миллзом (Harlan D. Mills), в его разработке приняли участие и некоторые другие сотрудники фирмы.

Обзор подхода

В общем случае под *стерильным цехом* (cleanroom, букв. чистая комната) понимают и метод, и технологию разработки продукта, а также помещение, предназначенное для использования этого метода или технологии. Само понятие используется в научно-исследовательской работе (например, в области биотехнологии) и промышленности (в частности, в полупроводниковой).

Стерильный цех — это производственное помещение, в котором специальными мерами обеспечивается низкий контролируемый уровень загрязнителей (пыль, микробы в воздухе, аэрозольные частицы, химические пары и т.п.) в окружающей среде. Поэтому это понятие отражает главную идею подхода — переход от устранения дефектов к их предотвращению.

Разработка программной системы (ПС) в виде стерильного цеха требует использования следующих особенностей — правил стерильного цеха:

1. Разработчики могут и должны производить ПО, которое почти свободно от ошибок уже перед выполнением тестирования.
2. Целью тестирования является измерение качества, а не его обеспечение.

Для учёта этих особенностей в такой разработке ПС необходимо использовать формальные методы. СцИП как раз и является развитием подхода IID на основе применения формальных методов.

С середины 1980-х гг. подход СцИП использовался подразделением FSD фирмы IBM в рамках проектов военного назначения. В 1987 г. Х. Миллз, М. Дайер (M. Dyer) и Р. Лингер (R. Linger) опубликовали статью «Инженерия стерильного

цеха» («Cleanroom Software Engineering»). После этого подход стали использовать и для некоторых коммерческих критически важных проектов.

СцИП обладает особенностями в следующих областях разработки:

1. Командная разработка (Team-based development).
2. Распределение времени по фазам ЖЦ (Time allocation across life cycle phases).
3. Существующие организационные практики (Existing organizational practices).

В области командной разработки считается, что команда проекта должна быть небольшой (обычно 6 - 8 человек) и работать в рамках определённой дисциплины для обеспечения разумного контроля над продвижением проекта. Предусматривается парный обзор (peer review) индивидуальной работы, но без вытеснения творческой деятельности отдельного члена команды. Как только будет разработана архитектура ПС и определены интерфейсы между компонентами этой системы, разработку компонентов можно выполнять индивидуально. Индивидуальные результаты считаются рабочими проектами и подвергаются командному обзору. Для крупных проектов для параллельной разработки отдельных подсистем ПС (после получения архитектуры) используется ряд небольших команд.

В области распределения времени по фазам ЖЦ считается, что для предотвращения дефектов следует выделять больше времени под фазу проектирования. В СцИП качество достигается именно через проектирование и верификацию, а не через тестирование, как в других неформальных подходах. В связи с таким акцентом подхода, в календарном плане для фаз проектирования и верификации необходимо отводить существенную часть времени ЖЦ. Практика применения СцИП и каскадных подходов для крупных проектов показывает, что затраты на разработку для СцИП сопоставимы и даже обычно ниже соответствующих затрат для традиционных каскадных подходов.

В области существующих организационных практик СцИП позволяет применять сложившиеся у разработчиков методики и практики, если они не противоречат принципам подхода. Переход к СцИП может происходить постепенно путём отработки принципов и методов подхода на небольшом некритическом проекте (пилотном проекте, pilot) и их приспособлении к принятой организации труда.

Основные принципы

Подход СцИП рассматривает процесс разработки ПО не как ремесло, а как инженерную деятельность, используя вместо традиционных методов разработки строгие точные методы.

Разработка в рамках подхода выражается в виде трёх принципов:

1. Инкрементная разработка под статистическим контролем качества.
2. Разработка ПО на основе математических принципов.
3. Тестирование ПО на основе статистических принципов.

Инкрементная разработка под статистическим контролем качества означает разработку ПО с использованием инкрементной стратегии, но на основе статистического контроля качества (SQC — statistical quality control). Инкремент представляет собой полное повторение процесса ЖЦ. Измерения производительности сравниваются с предустановленными стандартами для определения контролируемости процесса. Если стандарты качества не удовлетворяются, тестирование инкремента прекращается и происходит возврат на стадию проектирования.

Разработка ПО на основе математических принципов использует представление ПС как выражения математической функции (подробнее см. ниже). Для спецификирования и проектирования используется специальная методика (также см. ниже), а для подтверждения того, что дизайн представляет собой корректную реализацию спецификации, применяется функциональная верификация.

Поэтому формальная спецификация должна представляться требуемой математической функцией. Верификация заключается в командном обзоре ПС, основанном на вопросах корректности. Следовательно, до тестирования не происходит никакого исполнения кода программы.

Тестирование ПС на основе статистических принципов связано с рассмотрением тестирования как статистического эксперимента. Формируется представительная выборка всех возможных использований ПС. Производительность ПС на полученной выборке служит основой для заключения об общей эксплуатационной производительности. По протоколам тестирования, удовлетворяющим принципам прикладной статистики, может быть сформулировано математически обос-

нованное утверждение об ожидаемой эксплуатационной производительности ПС в терминах полноты тестирования и надёжности этой системы.

Таким образом, СцИП позволяет получить ПО, корректное математически спецификационным проектированием и заверенное статистически обоснованным тестированием. Уменьшение времени разработки в СцИП связано с инкрементной стратегией и исключением доработки.

При использовании этого подхода выявление и предотвращение дефектов во время проектирования позволяет уменьшить стоимость всего цикла разработки по сравнению с традиционными подходами и существенно ограничить область отказов ПО во время эксплуатации.

Жизненный цикл проекта

Основой модели ЖЦ для подхода служит модель трансформации, ориентированная на использование формальных методов (рис.4.20).

Особенностью модели ЖЦ для СцИП (рис.4.21) является использование специальной методики (см. ниже) вместо процесса трансформации. Это существенно снижает затраты на разработку ПС по сравнению с подходами формальной разработки, но обеспечивает высокий уровень приемлемого качества ПС. Поэтому СцИП оказывается сопоставимым по стоимости с другими современными подходами, превосходя их в качестве ПС.

В СцИП можно (условно) выделить следующие фазы:

1. Формализация.
2. Проектирование.
3. Верификация.
4. Сертификация.

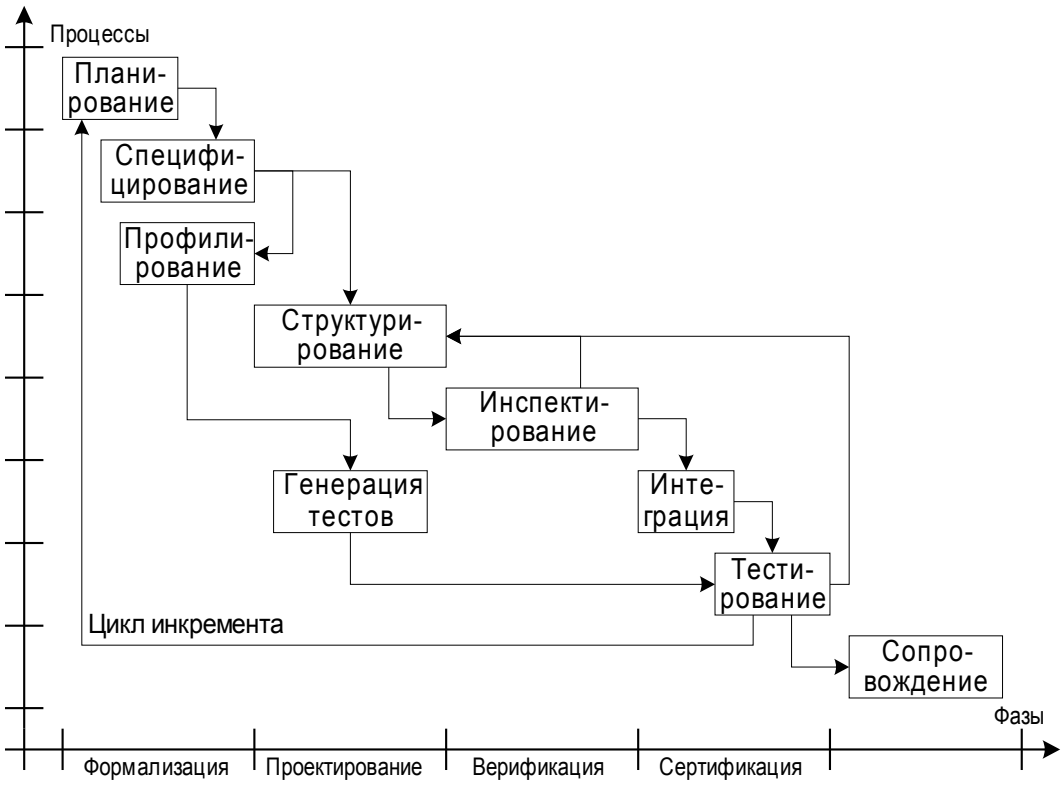


Рис.4.21. Схема модели ЖЦ для СцИП

При этом для сложных систем используется постепенное наращивание функциональности ПС на основе цикла инкремента путём реализации выделяемых заказчиком подмножеств требований на следующий инкремент разработки.

На фазе 1 «Формализация» выполняются три процесса:

1. Планирование — построение плана разработки инкремента ПС, в том числе выделение подмножества реализуемых в инкременте требований.
2. Специфицирование — преобразование неформальной спецификации требований в формальную спецификацию ПС.
3. Профилирование — разработка операционного профиля ПС на основе полученной формальной спецификации ПС.

На фазе 2 «Проектирование» выполняются два процесса:

1. Структурирование — проектирование ПС как структурированное преобразование формальной спецификации в программный код системы. В данном подходе дизайн ПС (в частности, архитектура системы) является промежуточным результатом структурирования.

2. Генерация тестов — разработка статистических тестов на основе операционного профиля системы.

Для сложных систем используется *пошаговое улучшение* (stepwise refinement) — постепенное определение структуры ПС на основе специальных правил, аналогичных правилам структурного программирования.

На фазе 3 «Верификация» выполняется один процесс — Инспектирование — формальная проверка соответствия кода ПС формальной спецификации (без запуска программы!). На этой фазе возможно продолжение Генерации тестов.

На фазе 4 «Сертификация» выполняются два процесса:

1. Интеграция — сборка кода компонентов ПС в единый код ПС.
2. Тестирование — статистическое тестирование кода ПС.

При обнаружении проблем при Инспектировании и Тестировании выполняется возврат к Структурированию.

Методика подхода

В рамках подхода ПС рассматривается как система, в которой входные данные называются *стимулами* (stimulus), получаемые результаты — *ответами* (response). На практике ответ системы определяется некоторой *последовательностью* (sequence) стимулов. Таким образом, ПС представляется как преобразователь стимулов в ответы.

Для разработки ПС используется специальная методика, основанная на следующих двух методах:

1. *Метод специфицирования на основе последовательностей* (МСОП, Sequence-Based Specification Method) — метод представления спецификации ПС в виде последовательностей.
2. *Метод структурирования на основе ящиков* (МСОЯ, Box Structure Method) — метод представления структуры ПС в виде ящиков.

МСОП позволяет получить формальную спецификацию ПС на основе неформальной спецификации требований. МСОЯ предназначен для проектирования ПС на основе её формальной спецификации.

Рассмотрим эти методы подробнее.

Метод специфицирования на основе последовательностей

Целью МСОП является разработка формальной спецификации ПС в виде математической функции — функции чёрного ящика (Black Box function). Областью её определения является множество всех последовательностей стимулов, входящих в ПС, областью значений — множество возможных ответов системы, выходящих из неё.

Исходным положением для МСОП является неформальная спецификация — описание требований к системе, написанное на естественном языке с использованием терминов ПрО. Метод обеспечивает разработку формальной спецификации с сохранением полной отслеживаемости (traceability) по исходной неформальной спецификации. Это позволяет заинтересованным лицам с помощью обследования установить, что формальная спецификация ПС определяет ту же систему, что и неформальная спецификация требований. Для облегчения отслеживаемости требованиям необходимо назначать уникальные идентификаторы.

Во время выполнения формализации могут возникнуть проблемы с тем, что некоторые описания в неформальной спецификации являются умалчиваемыми, неоднозначными и непоследовательными. В таких случаях необходимо разрешение этих проблем совместно с заинтересованными лицами и внесение соответствующих изменений в неформальную спецификацию.

Перечисление последовательностей стимулов. МСОП требует *перечисления* (enumeration) *последовательностей стимулов* — упорядочения по длительности всех последовательностей стимулов (включая пустую, недопустимые и невозможные последовательности) и назначения на каждую из них правильного ответа системы. Такое перечисление позволяет задать полностью определённую функцию и исключить таким образом некорректное поведение системы.

Теоретически область определения функции оказывается бесконечной из-за повторяемости стимулов и отсутствия ограничения на длину их последовательности. Практически же в большинстве систем неограниченность последовательностей стимулов не означает их бесконечности, так как системы имеют ограниченное поведение. Это поведение описывается внешне наблюдаемыми состоя-

ниями системы, соответствующими определённым последовательностям стимулов. Повторяемость поведения позволяет определить последовательности стимулов рекурсивным образом, что исключает бесконечное перечисление.

Классы эквивалентности последовательностей. МСОП разбивает бесконечное множество последовательностей стимулов на конечное множество *классов эквивалентности* (equivalence class). Две последовательности считаются *эквивалентными* (equivalent), если их всевозможные расширения (extensions) приводят к одному и тому же поведению (состоянию) системы.

Каждый класс эквивалентности характеризуется последовательностью минимальной длины — *канонической* (canonical) *последовательностью*. Если таких последовательностей оказывается несколько, то выбирается по перечислению первая из них. Пустая последовательность по определению также является канонической последовательностью.

Таблица перечисления последовательностей. Результаты перечисления представляются в виде набора, включающего по одной *таблице перечисления последовательностей* (Sequence Enumeration Table) на каждый класс эквивалентности (с соответствующей ему канонической последовательностью).

Для каждого стимула в каждой таблице существует одна строка, которая формируется по правилу чёрного ящика. *Правило Чёрного ящика* (Black Box rule) задаёт алгоритм перечисления последовательностей в виде формирования классов эквивалентности. Этот алгоритм включает следующие шаги:

1. Выбрать пустую (являющуюся канонической) последовательность.
2. Сформировать новый набор последовательностей путём расширения канонической последовательности соответствующими стимулами.
3. Для каждой новой последовательности определить ответ системы.
4. Для каждой новой последовательности определить класс эквивалентности.
5. Для каждой новой последовательности проанализировать результат:

—Если класс эквивалентности уже существует, исключить эту последовательность из рассмотрения. (Дальнейшие расширения уже определены и поведение системы в этом случае проанализировано.)

—Если класса эквивалентности не существует, то задать новый класс эквивалентности с этой последовательностью в качестве канонической и создать для этого класса свою таблицу перечисления последовательностей.

В таблицу с исходной канонической последовательностью заносится строка со стимулом, использованным для расширения, и указанием новой таблицы перечисления последовательностей.

6. Если рассмотрены все классы эквивалентности, то перейти к шагу 8.
7. Выбрать очередную каноническую последовательность и перейти к шагу 2.
8. Завершить перечисление последовательностей.

Совокупность таблиц перечисления последовательностей определяет формальную спецификацию как требуемую математическую функцию, заданную в табличном виде.

Метод структурирования на основе ящиков

Целью МСОЯ является постепенное преобразование формальной спецификации ПС в программный код системы. Для выполнения этого МСОЯ определяет представления ПО в виде так называемых *ящиков* (box) — моделей системы, характеризующих определённый уровень информации о неизвестной системе.

В МСОЯ выделены три ящика:

1. Чёрный ящик (Black Box).
2. Ящик состояний (State Box, букв. ящик с состоянием).
3. Прозрачный ящик (Clear Box, но это не White Box — белый ящик).

Чёрный ящик определяет видимое извне поведение системы или компонента в терминах её / его видимых взаимодействий с внешней средой; ответ формируется по стимулам. Ящик состояний задаёт поведение системы / компонента в виде состояний и переходов между ними; ответ формируется по текущему стимулу и состоянию. Прозрачный ящик представляет собой реализацию требуемого поведения — код для реализации переходов между состояниями и формирования ответов на стимулы.

Эти представления образуют иерархию абстракции, которая учитывает:

— пошаговое уточнение, так как может применяться как к системе в целом, так и к отдельным её компонентам;

— верификацию, так как каждое следующее представление системы / компонента получается из предыдущего.

Это не означает, что уточнение выполняется ровно в три этапа (независимо от размера и сложности ПС). На самом деле это означает, что во время каждого уточнения требуется использовать все три представления системы (рис.4.22): определить чёрный ящик, на его основе разработать ящик состояний, по которому реализовать прозрачный ящик.

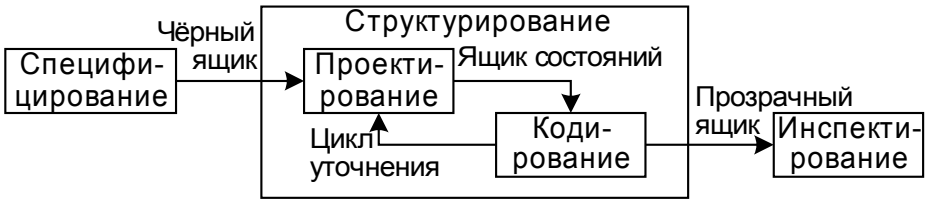


Рис.4.22. Схема уточнения на основе ящиков

Для ПС формальная спецификация является чёрным ящиком, программный код — прозрачным ящиком, а ящиком состояний служит детальный дизайн ПС.

Чёрный ящик получает на вход стимулы *S* и выдаёт на выход ответ *R*. Генерация каждого ответа чёрного ящика определяется текущим стимулом *S* и (возможно) *историей стимулов SH* — предыдущей последовательностью стимулов. Тогда функция преобразования чёрного ящика имеет следующий вид:

$$(S, SH) \rightarrow (R).$$

Ящик состояний получается из чёрного ящика путём выделения состояний, инкапсулирующих части истории стимулов (инварианты состояний), на основе таблиц перечисления последовательностей. Влияние истории стимулов *SH* на ответ заменяется изменением старого состояния S_o на новое состояние S_n ПС. Следовательно, функция преобразования ящика состояний имеет следующий вид:

$$(S, S_o) \rightarrow (R, S_n).$$

Если существует несколько ящиков состояний, получаемых из чёрного ящика, осуществляется выбор одного из них.

Прозрачный ящик получается из ящика состояний реализацией требуемого преобразования в виде программного кода. Поэтому функция преобразования прозрачного ящика имеет следующий вид:

$$(S, S_o) \text{ — КОД} \rightarrow (R, S_n).$$

Данная реализация выполняется постепенным определением логики изменения состояний и метода генерации ответов ПС. При этом прозрачный ящик часто выражается как композиция новых заданных чёрных ящиков. В подходе определены правила композиции чёрных ящиков аналогично правилам для конструкций, используемым в структурном программировании.

Если существует несколько прозрачных ящиков, получаемых из ящика состояний, осуществляется выбор одного из них.

Цикл уточнения (рис.4.22) завершается, когда все чёрные ящики преобразованы в прозрачные. В итоге получается полный прозрачный ящик — код ПС.

Контрольные вопросы

Вопросы к §4.1

1. Охарактеризуйте каскадные технологические подходы.
2. Перечислите виды каскадных подходов и примеры подходов каждого вида.

Простые каскадные подходы

3. В чём суть классического каскадного подхода?
4. В чём суть модифицированного каскадного подхода?

Развитые каскадные подходы

5. В чём суть каскадно-возвратного подхода? Приведите графическое представление модели для этого подхода.
6. В чём суть каскадно-итерационного подхода? Приведите графическое представление модели для этого подхода.
7. В чём суть каскадно-перекрывающегося подхода? Приведите графическое представление модели для этого подхода.
8. В чём суть каскадно-декомпозиционного подхода? Приведите графическое представление модели для этого подхода.

Вопросы к §4.2

9. Охарактеризуйте каркасные технологические подходы.
10. Перечислите виды каркасных подходов и примеры подходов каждого вида.

Унифицированный процесс (УП, UP)

11. Что представляет собой подход УП?
12. Перечислите и поясните особенности УП.
13. Приведите графическое представление модели ЖЦ для УП.
14. Перечислите фазы ЖЦ проекта для УП.
15. Перечислите потоки работ ЖЦ проекта для УП.
16. Перечислите вехи ЖЦ проекта для УП.

17. Перечислите модификации УП.

Рациональный унифицированный процесс (РУП, RUP)

18. Что представляет собой подход РУП?
19. Что представляет собой Rational Unified Process как продукт?
20. Перечислите первопричины провала проекта.
21. Перечислите признаки (проявления первопричин) провала проекта.
22. Дайте определение понятию «лучшая практика».
23. Перечислите лучшие практики, используемые в РУП.
24. Поясните лучшую практику «Итеративная разработка».
25. Поясните лучшую практику «Управление требованиями».
26. Поясните лучшую практику «Использование компонентной архитектуры».
27. Поясните лучшую практику «Визуальное моделирование».
28. Поясните лучшую практику «Проверка качества».
29. Поясните лучшую практику «Контроль изменений».
30. Перечислите и поясните ключевые принципы бизнес-управляемой разработки.
31. Приведите графическое представление модели ЖЦ для РУП.
32. Перечислите и поясните фазы ЖЦ проекта для РУП.
33. Перечислите вехи ЖЦ проекта для РУП.
34. Перечислите и поясните основные потоки работ ЖЦ проекта для РУП.
35. Перечислите и поясните вспомогательные потоки работ ЖЦ проекта для РУП.
36. Как распределяются фазы ЖЦ для РУП по трудоёмкости и затратам времени? Приведите графическое представление распределения фаз.
37. Как распределяется нагрузка потоков работ РУП по фазам ЖЦ?
38. Приведите графическое представление итеративности разработки для РУП.

Каркас решений Microsoft (МСФ, MSF)

39. Что представляет собой подход МСФ?

40. Что представляет собой Microsoft Solutions Framework как продукт? Охарактеризуйте пакет руководств МСФ 4.0.
41. Перечислите и поясните компоненты и шаблоны МСФ 4.0.
42. Перечислите основополагающие принципы МСФ.
43. Перечислите ключевые концепции МСФ.
44. Перечислите и поясните особенности модели руководства МСФ.
45. Поясните образование модели ЖЦ из модели руководства МСФ.
46. Приведите графическое представление модели ЖЦ для МСФ.
47. Перечислите и поясните фазы ЖЦ проекта для МСФ.
48. Перечислите и поясните вехи ЖЦ проекта для МСФ.
49. Охарактеризуйте фазу «Представление» подхода МСФ. Перечислите промежуточные вехи и результаты этой фазы ЖЦ.
50. Охарактеризуйте фазу «Планирование» подхода МСФ. Перечислите промежуточные вехи и результаты этой фазы ЖЦ.
51. Охарактеризуйте фазу «Разработка» подхода МСФ. Перечислите промежуточные вехи и результаты этой фазы ЖЦ.
52. Охарактеризуйте фазу «Стабилизация» подхода МСФ. Перечислите промежуточные вехи и результаты этой фазы ЖЦ.
53. Охарактеризуйте фазу «Развёртывание» подхода МСФ. Перечислите промежуточные вехи и результаты этой фазы ЖЦ.

Процесс ICONIX

54. Что представляет собой Процесс ICONIX?
55. Как связан Процесс ICONIX с подходами RUP и XP?
56. Перечислите и поясните основные особенности ICONIX.
57. Перечислите условия построения хороших моделей объектов в ICONIX.
58. Перечислите и поясните ключевые принципы ICONIX.
59. Приведите графическое представление модели ЖЦ для ICONIX.

60. Перечислите и поясните этапы ЖЦ проекта для ICONIX.
61. Перечислите и поясните вехи ЖЦ проекта для ICONIX.
62. Охарактеризуйте этап «Анализ требований» Процесса ICONIX. Какие действия выполняются на этом этапе? Что позволяет установить соответствующая данному этапу ЖЦ веха?
63. Охарактеризуйте этап «Предварительное проектирование» Процесса ICONIX. Какие действия выполняются на этом этапе? Что позволяет установить соответствующая данному этапу ЖЦ веха?
64. Охарактеризуйте этап «Детализированное проектирование» Процесса ICONIX. Какие действия выполняются на этом этапе? Что позволяет установить соответствующая данному этапу ЖЦ веха?
65. Охарактеризуйте этап «Реализация» Процесса ICONIX. Какие действия выполняются на этом этапе? Что позволяет установить соответствующая данному этапу ЖЦ веха?

Вопросы к §4.3

66. Охарактеризуйте эволюционные технологические подходы.
67. Перечислите виды эволюционных подходов и примеры подходов каждого вида.
68. Что представляет собой непланируемый подход? Охарактеризуйте подход.

Подходы прототипирования

69. Что представляют собой подходы прототипирования?
70. Перечислите основные подходы прототипирования.
71. Дайте определение понятиям, связанным с прототипом.
72. Охарактеризуйте подход Эволюционная доставка. В чём суть модели ЖЦ для этого подхода. Приведите графическое представление данной модели.
73. Охарактеризуйте подход Итеративная доставка. В чём суть модели ЖЦ для этого подхода. Приведите графическое представление данной модели.

74. Охарактеризуйте подход Постадийная доставка. В чём суть модели ЖЦ для этого подхода. Приведите графическое представление данной модели.

Итеративная инкрементная разработка (ИИР, IID)

75. Что представляет собой подход ИИР?
76. В каких подходах ИИР является одним из основных компонентов?

Быстрая разработка приложений (БРП, RAD)

77. Что представляет собой подход БРП?
78. Какие подходы созданы на основе БРП?
79. Перечислите и поясните особенности БРП.
80. Перечислите и поясните основные принципы БРП.
81. Каким образом в БРП определяется оценка размера приложения? Что понимается под функциональным элементом?
82. Приведите графическое представление модели ЖЦ для БРП.
83. Перечислите фазы ЖЦ проекта для БРП.
84. Охарактеризуйте фазу «Планирование и Анализ требований» подхода БРП. Перечислите и поясните деятельности, выполняемые на этой фазе. Перечислите результаты этой фазы ЖЦ.
85. Охарактеризуйте фазу «Проектирование» подхода БРП. Перечислите и поясните деятельности, выполняемые на этой фазе. Перечислите результаты этой фазы ЖЦ.
86. Охарактеризуйте фазу «Построение» подхода БРП. Перечислите и поясните деятельности, выполняемые на этой фазе. Укажите результат этой фазы ЖЦ.
87. Охарактеризуйте фазу «Внедрение» подхода БРП. Перечислите и поясните деятельности, выполняемые на этой фазе ЖЦ.

Вопросы к §4.4

88. Охарактеризуйте адаптивные технологические подходы.
89. Перечислите виды адаптивных подходов и примеры подходов каждого вида.

Особенности адаптивных подходов

90. Что представляет собой Живая разработка ПО?
91. Перечислите основные положения Живого манифеста.
92. Перечислите принципы Живой разработки ПО.

Адаптивная разработка ПО (АРП, ASD)

93. Что представляет собой подход АРП?
94. Сформулируйте модель сложной адаптивной системы, используемой в АРП. Охарактеризуйте процесс разработки как сложную адаптивную систему.
95. В чём состоит особенность модели ЖЦ для АРП? Приведите графическое представление схемы этой модели.
96. Перечислите и поясните свойства АРП.
97. Приведите графическое представление модели ЖЦ для АРП.
98. Перечислите фазы ЖЦ проекта для АРП.
99. Охарактеризуйте фазу «Обдумывание» подхода АРП. Перечислите и поясните процессы и действия процессов, выполняемые на этой фазе ЖЦ.
100. Охарактеризуйте фазу «Сотрудничество» подхода АРП. Укажите и поясните процесс, выполняемый на этой фазе ЖЦ.
101. Охарактеризуйте фазу «Обучение» подхода АРП. Перечислите и поясните процессы, выполняемые на этой фазе ЖЦ.
102. Сформулируйте и поясните суть адаптивности подхода АРП.

Экстремальное программирование (ЭП, XP)

103. Что представляет собой подход ЭП?
104. Перечислите категории ЭП. Поясните их взаимосвязь.
105. Перечислите ценности ЭП. Поясните их значение.
106. Перечислите и поясните основные принципы ЭП(1).
107. Перечислите и поясните вспомогательные принципы ЭП(1).
108. Перечислите и поясните принципы ЭП(2).

109. Перечислите группы практик ЭП.
110. Перечислите и поясните практики ЭП(1).
111. Перечислите и поясните основные практики ЭП(2).
112. Перечислите и поясните дополнительные практики ЭП(2).
113. Как взаимосвязаны практики ЭП(1) и ЭП(2)? Приведите графическое представление связи практик. Поясните направление развития ЭП.
114. Приведите графическое представление схемы модели ЖЦ для ЭП.
115. Перечислите фазы ЖЦ проекта для ЭП.
116. Охарактеризуйте фазу «Исследование» подхода ЭП. Поясните операции, выполняемые на этой фазе ЖЦ.
117. Охарактеризуйте фазу «Планирование» подхода ЭП. Поясните операции, выполняемые на этой фазе ЖЦ.
118. Охарактеризуйте фазу «Реализация» подхода ЭП. Поясните операции, выполняемые на этой фазе ЖЦ.
119. Охарактеризуйте фазу «Продуцирование» подхода ЭП. Поясните операции, выполняемые на этой фазе ЖЦ.
120. Охарактеризуйте фазу «Смерть» подхода ЭП. Перечислите причины «наступления» этой фазы ЖЦ.
121. Перечислите и поясните деятельности ЭП.

Вопросы к §4.5

122. Охарактеризуйте генетические технологические подходы.
123. Перечислите виды генетических подходов. Как эти виды связаны с методологиями разработки ПО?
124. Какая существует связь между генетическими и формальными генетическими подходами?

Синтезирующее программирование

125. В чём суть синтезирующего программирования?
126. Какие задачи необходимо решить при синтезирующем программировании?

127. Перечислите и поясните языки спецификации, часто применяемые в подходах синтезирующего программирования.

Конкретизирующее программирование

128. В чём суть конкретизирующего программирования?
129. Перечислите подходы конкретизирующего программирования.
130. Охарактеризуйте обобщённое программирование.
131. Охарактеризуйте подход на основе паттернов и анти-паттернов.
132. Охарактеризуйте подход на основе архитектурных стилей.
133. Что такое шаблонно-ориентированное программирование?

Сборочное программирование

134. В чём суть сборочного программирования?
135. Какие существуют способы поддержки сборочного программирования?
136. Перечислите подходы сборочного программирования.
137. Охарактеризуйте модульное сборочное программирование.
138. Что такое модульно-ориентированное программирование?
139. Охарактеризуйте расширяемое программирование.
140. Охарактеризуйте объектное сборочное программирование.
141. Охарактеризуйте компонентное сборочное программирование.
142. Что такое компонентно-ориентированное программирование?
143. Охарактеризуйте аспектное сборочное программирование.
144. Что такое аспектно-ориентированное программирование?

Вопросы к §4.6

145. Охарактеризуйте формальные технологические подходы.
146. Перечислите виды формальных подходов и примеры подходов каждого вида.

Формальные генетические подходы

147. Охарактеризуйте формальные генетические подходы.
148. Что такое доказательное программирование?

149. Перечислите формальные генетические подходы.
150. Как определяются генетические технологические подходы по формальным генетическим подходам?
151. В чём суть формального синтезирующего программирования?
152. Что такое математическая спецификация?
153. Что понимается под синтезом программы?
154. Перечислите способы синтеза программы.
155. Поясните логический способ синтеза программы.
156. Поясните аналитический способ синтеза программы.
157. Как проявляется творчество в синтезирующем программировании?
158. Как связан синтез программы с манипулированием знанием?
159. В чём суть формального конкретизирующего программирования?
160. Что такое универсальная программа?
161. Что понимается под конкретизацией программы?
162. Какой математический аппарат используется в конкретизирующем программировании? Как он используется для конкретизации программы?
163. В чём суть формального сборочного программирования?
164. Что такое программный модуль?
165. Что понимается под сборкой программы?

Подходы формальной разработки

166. Охарактеризуйте подходы формальной разработки.
167. Что такое формальные методы?
168. В чём суть трансформационной модели? Приведите графическое представление схемы этой модели.
169. Перечислите и поясните процессы ЖЦ для формальных подходов.
170. Дайте определение понятию «формальная спецификация».
171. Дайте определение понятию «операционный профиль».

172. Перечислите и поясните представления структуры системы для ЯФС.
173. Перечислите и поясните представления поведения системы для ЯФС.
174. Охарактеризуйте язык формальной спецификации Z notation.
175. Охарактеризуйте язык формальной спецификации CASL.
176. Охарактеризуйте семейство подходов Исчисление процессов.
177. Охарактеризуйте подход B-Метод (B-Method).
178. Охарактеризуйте подход Венский метод разработки (VDM).
179. Перечислите особенности подходов Исчисления процессов.
180. Охарактеризуйте подход CCS Исчисления процессов.
181. Охарактеризуйте подход CSP Исчисления процессов.
182. Охарактеризуйте подход ACP Исчисления процессов.

Инженерия стерильного цеха (СцИП, CrSE)

183. Что представляет собой подход СцИП?
184. Дайте определение понятию «стерильный цех».
185. Перечислите правила стерильного цеха.
186. Перечислите области разработки, в которых СцИП имеет особенности. Охарактеризуйте эти особенности.
187. Перечислите и поясните основные принципы разработки в рамках СцИП.
188. Приведите графическое представление схемы модели ЖЦ для СцИП.
189. Перечислите фазы ЖЦ проекта для СцИП.
190. Охарактеризуйте фазу «Формализация» подхода СцИП. Перечислите и поясните процессы, выполняемые на этой фазе ЖЦ.
191. Охарактеризуйте фазу «Проектирование» подхода СцИП. Перечислите и поясните процессы, выполняемые на этой фазе ЖЦ. Что такое пошаговое улучшение?
192. Охарактеризуйте фазу «Верификация» подхода СцИП. Укажите и поясните процесс, выполняемый на этой фазе ЖЦ.

193. Охарактеризуйте фазу «Сертификация» подхода СцИП. Перечислите и поясните процессы, выполняемые на этой фазе ЖЦ.
194. В чём суть специальной методики, используемой в рамках СцИП.
195. Охарактеризуйте метод специфицирования на основе последовательностей (МСОП) подхода СцИП. Что такое последовательность стимулов?
196. Что такое перечисление последовательностей стимулов?
197. Что такое класс эквивалентности для последовательностей стимулов?
198. Что такое каноническая последовательность стимулов?
199. Что такое таблица перечисления последовательностей?
200. Сформулируйте правило Чёрного ящика.
201. Охарактеризуйте метод структурирования на основе ящиков (МСОЯ) подхода СцИП. Что понимается под ящиком?
202. Перечислите и поясните ящики, используемые в МСОЯ.
203. Как связаны между собой ящики? Приведите графическое представление схемы уточнения на основе ящиков. Как связаны представления ПС и ящики?
204. Сформулируйте функцию преобразования для чёрного ящика.
205. Сформулируйте функцию преобразования для ящика состояний.
206. Сформулируйте функцию преобразования для прозрачного ящика.

Раздел 5. Инженерия и инструментарий ПО

В данном разделе рассматривается ряд вопросов инженерии и инструментария ПО, тесно связанных с методологией и технологией разработки ПО.

5.1. Инженерия ПО

Ряд направлений инженерии ПО представляет интерес с методологической и технологической точек зрения.

Стиль программирования

Стиль программирования (programming style) — набор приёмов, методик и практик кодирования, которые используются, чтобы получить правильные, эффективные, простые для понимания и удобные для применения программы (также см. §3.4, «Процесс 5. Кодирование»).

Брайан У. Керниган (Brian W. Kernighan) и Роб Пайк (Rob Pike) уточняют, что код должен быть прост и понятен, т.е. обладать следующими свойствами:

— очевидная логика;
— естественные выражения;
— использование соглашений, принятых в языке;
— осмысленные имена;
— аккуратное форматирование;
— развёрнутые комментарии;
— отсутствие хитрых трюков и необычных конструкций.

Обычно стиль программирования формируется как результат здравого смысла, исходящего из опыта. В то же время он не является постоянным, так как во многом определяется используемым языком конструирования, командой разработчиков и стандартами различного уровня.

Во многом стремление к хорошему стилю при императивном программировании привело к разработке структурного программирования (см. §2.4) и способствовало развитию (обычного и формального) модульного сборочного программирования (см. §4.5 и §4.6 соответственно).

Защитное программирование

Защитное программирование (defensive programming) — подход к программированию, при котором появляющиеся ошибки легко обнаруживаются и идентифицируются программистом (также см. §3.4, «Процесс 5. Кодирование» и «Процесс 6. Тестирование»).

Основные принципы и механизмы

Существуют три основных принципа защитного программирования:

1. Общее недоверие: Для каждого модуля входные данные должны тщательно анализироваться в предположении, что они могут быть ошибочными.
2. Немедленное обнаружение: Каждая ошибка должна быть выявлена как можно раньше, это упрощает установление её причины.
3. Изолирование ошибки: Ошибки в одном модуле должны быть изолированы так, чтобы не допустить их влияние на другие модули.

Выделяют ряд общих рекомендаций по защитному программированию:

— Проверка области значений переменных.
— Контроль правдоподобности значений переменных.
— Проверка результатов вычислений.
— Автоматические проверки (например, контроль переполнения).
— Проверка длины элементов информации.
— Проверка кодов возврата функций.

К механизмам защитного программирования относят:

— директивы проверки (checking directives) — команды компилятору для использования встроенных средств генерации элементарных (но важных) проверок перед использованием некоторых операций;
— утверждения (assertions) — логические операторы или подпрограммы для проверки правильности выполнения программы с помощью некоторых ограничений, наложенных на программу;
— исключения (exceptions) — представление ошибок или некорректных ситуаций для управления их обработкой в нужном месте программы;

— баррикады (barricades) — изоляция повреждений путём организации проверок при передаче данных между определёнными программными модулями;
— отладочный код (debug code) — специальный код для выполнения дополнительных проверок и облегчения поиска ошибок.

Недостатком защитного программирования является усложнение программного кода, однако он устраняется применением аспектного сборочного программирования (см. §4.5, «Сборочное программирование»). Достоинством этого подхода как раз и является простая реализация пересекающих значимостей, в качестве которых выступает большинство перечисленных выше механизмов.

Одним из самых известных подходов, на основе которых можно эффективное защитное программирование, является проектирование по контракту.

Проектирование по контракту

Проектирование по контракту (DbC — Design by Contract) — подход к проектированию и программированию, основанный на документировании прав и обязанностей подпрограмм и программных модулей для обеспечения корректности программы. Он предложен Бертраном Мейером (Bertrand Meyer) и изложен им в 1997 г. в книге «Конструирование объектно-ориентированного ПО» («Object-Oriented Software Construction»).

Подход основывается на методологии ООП, хотя может быть построен и на основе других методологий. В подходе на основе методологии ООП модулем считается класс, подпрограммой — операция класса (метод).

Проектирование по контракту использует утверждения трёх видов: предусловия, постусловия и инварианты.

Предусловие (precondition) — утверждение о требовании для выполнения подпрограммы. Предусловие проверяется перед выполнением подпрограммы для обеспечения правильности исходных данных этой подпрограммы.

Постусловие (postcondition) — утверждение о требовании к выполнению подпрограммы. Постусловие проверяется после выполнения подпрограммы для обеспечения правильности результатов этой подпрограммы.

Инвариант (invariant) — утверждение о требовании по выполнению программного модуля. Инвариант является истинным при взаимодействии модуля с другими модулями, но может быть ложным при выполнении подпрограммы этого модуля.

Перечисленные утверждения позволяют при грамотном использовании контролировать не только выполнение программы, но и её разработку. Проектирование по контракту оказывается эффективным при применении аспектного сборочного программирования (см. §4.5, «Сборочное программирование»).

5.2. Инструментарий ПО

Ряд направлений инструментария ПО позволяет оценить инженерно-практическую сторону методологии и технологии разработки ПО.

Автоматизация разработки

Методологические и технологические подходы разработки становятся эффективными и экономически выгодными при их автоматизации. Системы автоматизации программной / системной разработки получили название «CASE-средства».

CASE-средства

CASE-средство (CASE — Computer-Aided Software / System Engineering — букв. компьютерная автоматизированная программная / системная инженерия) — система автоматизированной разработки ПО / систем с помощью компьютеров.

Обычно CASE-средством считается программное средство, автоматизирующее некоторую совокупность ЖЦ и обладающее следующими особенностями:

1. Визуальные возможности для описания и документирования систем — обеспечивает удобный интерфейс с разработчиком и реализацию его творческих возможностей, избавляя от рутинной деятельности.
2. Интеграция компонент этого средства — позволяет управлять процессом разработки с помощью элементарной передачи данных между компонентами.
3. Использование репозитория — единого хранилища информации о проекте.

Интегрированное CASE-средство включает в себя следующие компоненты:

1. Репозиторий — основа CASE-средства: база данных со специальными возможностями по хранению и управлению информацией о проекте.
2. Компоненты разработки: бизнес-моделирование с использованием различных методологий и технологий, анализ и проектирование.
3. Компоненты программирования: кодирование и тестирование / инспектирование, а также интеграция и сопровождение.
4. Компоненты поддержки: документирование и управление конфигурацией, верификация и аттестация, обзор и аудит.
5. Компоненты организации: управление проектом, инфраструктура.

Классификация CASE-средств

CASE-средства обычно классифицируются по типам и категориям.

Тип CASE-средства отражает его функциональное назначение:

1. Анализ и проектирование: анализ требований и проектирование, в том числе определение требований, специфицирование и построение архитектуры системы различного уровня детализации.
2. Проектирование баз данных: моделирование данных, преобразования моделей данных, генерация схем баз данных и описаний форматов файлов.
3. Программирование (разработка приложений): автоматизированное кодирование, тестирование и/или инспектирование, интеграция.
4. Сопровождение и поддержка: сопровождение всех категорий, документирование и другие связанные действия.
5. Управление проектом: руководство, планирование, контроль.
6. Инфраструктура: создание и управление инфраструктурой.

Таким образом, классификация по типам определяется компонентным составом CASE-средств.

Категория CASE-средства связана со степенью взаимодействия его компонентов в рамках охватываемых им стадий ЖЦ:

1. Инструментальное средство (tool, букв. инструмент) — вспомогательное средство для решения относительно самостоятельных задач.

2. Инструментальный пакет (toolkit, букв. набор инструментов) — связанная совокупность инструментальных средств для решения класса задач обычно в рамках одной стадии ЖЦ.
3. Инструментарий (workbench, букв. верстак) — организованная совокупность инструментальных средств для решения класса задач в рамках всего ЖЦ.

Таким образом, классификация по категориям определяется степенью интеграции компонентов CASE-средств в рамках выполняемых функций.

Дополнительная классификация связана с выделением уровней.

Уровень CASE-средства выражает область его действия в рамках ЖЦ:

1. Верхний (Upper) уровень: организация, управление.
2. Средний (Middle) уровень: моделирование, анализ и проектирование.
3. Нижний (Lower) уровень: программирование и поддержка.

Таким образом, классификация по уровням определяется ориентацией на конкретные группы пользователей и связана с типом CASE-средств.

Кроме этого, существуют и другие классификации CASE-средств.

Системы автоматизации

Для инженерии ПО интерес представляют инструментарии (средства и платформы) для поддержки конкретных технологий. Кроме этого часто используются и отдельные среды разработки для поддержки программирования.

Большинство технологий разработки и соответствующих им CASE-средств ориентировано на одну из двух наиболее популярных методологий — структурную или объектно-ориентированную (также см. §3.5).

Для бизнес-моделирования, анализа и проектирования CASE-средства на основе структурной методологии используют подходы на основе DFD, ERD, STD и IDEF0 (SADT) с применением при необходимости других моделей и методов, а CASE-средства на основе объектно-ориентированной методологии применяют подход на основе UML.

Контрольные вопросы

Вопросы к §5.1

1. Дайте определение понятию «стиль программирования».
2. Перечислите свойства хорошего стиля программирования.
3. Как формируется стиль программирования?
4. Как связан стиль программирования с методологиями разработки?
5. Дайте определение понятию «защитное программирование».
6. Перечислите основные принципы защитного программирования.
7. Перечислите общие рекомендации по защитному программированию.
8. Перечислите и поясните механизмы защитного программирования.
9. Как защитное программирование связано с аспектным сборочным программированием?
10. Что представляет собой подход Проектирование по контракту?
11. Поясните механизм, используемый Проектированием по контракту?

Вопросы к §5.2

12. Что такое CASE-средство?
13. Перечислите особенности CASE-средств.
14. Перечислите компоненты CASE-средств.
15. Перечислите основные признаки классификации CASE-средств.
16. Приведите классификацию CASE-средств по типам.
17. Приведите классификацию CASE-средств по категориям.
18. Приведите классификацию CASE-средств по уровням.
19. Кратко охарактеризуйте системы автоматизации.

Литература

Основная литература

1. Одинцов И.О. Профессиональное программирование: Системный подход.— СПб.: BHV-Санкт-Петербург, 2002.— 512 с. (Мастер).— Эл. версия.— URL: http://lib.aswl.ru/books/methodology/programming/ . См. также: Одинцов И.О. Профессиональное программирование: Системный подход.— 2-е изд., перераб. и доп.— СПб.: BHV-Санкт-Петербург, 2004.— 624 с.— (Мастер).
2. Иванова Г.С. Технология программирования: Учеб. для вузов.— М.: Изд-во МГТУ им. Н.Э. Баумана, 2002.— 320 с.— (Информатика в техн. ун-те).
3. Орлов С.А. Технологии разработки программного обеспечения: Разработка сложных программных систем: Учеб. пособие.— 2-е изд.— СПб.: Питер, 2003.— 480 с.
4. Воройский Ф.С. Информатика: Новый систематизированный толковый словарь-справочник. (Введение в современные информационные и телекоммуникационные технологии в терминах и фактах).— 3-е изд., перераб. и доп.— М.: ФИЗМАТЛИТ, 2003.— 760 с.
5. Орлик С. Введение в программную инженерию и управление жизненным циклом ПО / При уч. Ю. Булуя.— Эл. версия.— URL: http://sorlik.blogspot.com/ .
6. Непейвода Н.Н., Скопин И.Н. Основания программирования.— М.; Ижевск: Изд-во РХД, 2003.— 880 с.— Эл. версия от 11.09.2003. 2+iv+914 с.— URL: http://ulm.udsu.ru/~nnn/fp.zip .
7. Соммервилл И. Инженерия программного обеспечения. 6-е изд. / Пер. с англ. под ред.А.А. Минько.— М.: Вильямс, 2002.— 624 с.
8. Калянов Г.Н. CASE-технологии: Консалтинг в автоматизации бизнес-процессов.— 3-е изд.— М.: Горячая линия — Телеком, 2002.— 320 с.
9. Кватрани Т. Rational Rose 2000 и UML: Визуальное моделирование / Пер. с англ.— М.: ДМК Пресс, 2001.— 176 с.— (Объектно-ориентированные технологии в программировании).

Дополнительная литература

10. Системный анализ и принятие решений: Словарь-справочник / Под общ. ред. В.Н. Волковой, В.Н. Козлова.— М.: Высш. шк., 2004.— 616 с.
11. Непейвода Н.Н. Стили и методы программирования // Интернет-университет информационных технологий, 2005.— Проект Изд-ва «Открытые системы».— Эл. изд.— URL: http://www.intuit.ru/department/se/progstyles/ .
12. Технология разработки программного обеспечения: Конспект лекций / Автор неизвестен.— Эл. изд.— URL: http://www.solomil.ru/ .
13. Берлинский К. Набор серебряных пуль: Справочник удачных проектных решений при разработке ПО.— 2004.— 99 с.— Эл. изд., v.1.37, 20.06.2004.
14. Марков Е. Архитектура, управляемая моделью // CIT City. 2005.— Эл. изд.: 15.12.2005.— URL: http://citcity.ru/ .
15. Безуглый Д.Л. Технология разработки программного обеспечения // Корпоративные Информационные Системы.— 2001.— №2.— С. 24—30.
16. IBM Corp. Методология и инструментальные средства IBM Rational для разработки программных систем. 26.04.2007 / Пер. с англ.— Эл. изд.— URL: http://1050049.ru/iservices/files.asp?artId=2186&file=0 .
17. Microsoft Corp. Microsoft Solutions Framework: Белая книга. Модель процессов MSF. Ver. 3.1, 06.2002 / Пер. с англ. под ред. В. Павлова.— Эл. изд.— URL: http://www.microsoft.com/rus/docs/msdn/msf/MSF_process_model_rus.doc .
18. Коуберн А. Четвёртое измерение, или Как обмануть Железный Треугольник / Пер. с англ. К. Максимова, А. Максимова.— 08.08.2004.— Эл. изд., оригинал от 04.10.2003.— URL: http://www.maxkir.com/ .
19. Коуберн А. Создание программного обеспечения как коллективная игра (По материалам статей А. Коуберна за 1997 — 2004 гг.) / Пер. с англ. К. Максимова, А. Максимова.— 11.09.2004.— Эл. изд.— URL: http://www.maxkir.-com/ .
20. Фаулер М. Новые методологии программирования / Пер. с англ. К. Максимова, А. Максимова.— 13.10.2001.— Эл. изд.— URL: http://www.maxkir.com/ .

21. Хайсмит Дж. Устаревшие методологии — на пенсию! / Пер. с англ. К. Максимова, А. Максимова.— 20.03.2002.— Эл. изд., оригинал от 07 — 08.2000.— URL: http://www.maxkir.com/ .

22. Бек К. Экстремальное программирование.— СПб.: Питер.— 216 с.— (Б-ка программиста). См. также: Бек К. Экстремальное программирование: Разработка через тестирование.— СПб.: Питер, 2003.— 224 с.— (Б-ка программиста).

23. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приёмы объектно-ориентированного программирования: Паттерны проектирования / Пер. с англ. А. Слинкина.— СПб.: Питер, 2001.— 368 с.— (Б-ка программиста).

24. Дубина О. Обзор паттернов проектирования // CIT Forum.— 2005.— Эл. изд. — URL: http://citforum.ru/SE/project/pattern/ .

25. Ершов А.П. Научные основы доказательного программирования.— 1984.— 15 с.— Эл. версия.— URL: http://www.europrog.ru/paper/ae1984-02r.pdf .

26. Ершов А.П. Отношения методологии и технологии программирования // Технология программирования. Информ. материалы и тез. плен. докл. II Всесоюз. конф. (Киев, 18 — 21.11.1986.) / Ин-т кибернетики им. В.М. Глушкова, АН УССР.— Киев, 1986.— С. 10 — 13.— Эл. версия.— URL: http://ershov.iis.nsk.su/ .

27. Дейкстра Э.В. Дисциплина программирования / Пер. с англ. под ред. Э.З. Любимского.— М.: Мир, 1978.— 290 с.— (Матем. обеспечение ЭВМ).

28. Грис Д. Наука программирования / Пер. с англ. под ред. А.П. Ершова.— М.: Мир, 1984.— 416 с.

29. Вендров А.М. CASE-технологии: Современные методы и средства проектирования информационных систем // CIT Forum.— 1997.— Эл. изд.— URL: http://citforum.ru/ .

30. Норенков И.П. Основы автоматизированного проектирования: Учеб. для вузов.— 2-е изд., перераб. и доп.— М.: Изд-во МГТУ им. Н.Э. Баумана, 2002. — 336 с.— (Информатика в техн. ун-те).

31. Дубова Н. В круге разработки // Открытые системы.— 2003.— № 9.— CIT City.— Эл. изд.: 18.09.2003.— URL: http://www.osp.ru/os/ .

Документация

Основные стандарты, связанные с жизненным циклом ПО и систем:

32. ISO/IEC 12207:1995 «Information Technology — Software Life Cycle Processes». Рус. версия: ГОСТ Р ИСО/МЭК 12207:1999 «Информационная технология. Процессы жизненного цикла программных средств».

33. ISO/IEC 15288:2002 «Systems Engineering — System Life Cycle Processes». Рус. версия: ГОСТ Р ИСО/МЭК 15288:2005 «Системная инженерия. Процессы жизненного цикла систем».

34. ISO/IEC TR 15271:1998 «Information Technology — Guide for ISO/IEC 12207 (Software Life Cycle Processes)». Рус. версия: ГОСТ Р ИСО/МЭК ТО 15271:2002 «Информационная технология. Руководство по применению ГОСТ Р ИСО/МЭК 12207 (Процессы жизненного цикла программных средств)».

35. ISO/IEC TR 16326:1999 «Software Engineering — Guide for the Application of ISO/IEC 12207 to Project Management». Рус. версия: ГОСТ Р ИСО/МЭК ТО 16326:2002 «Программная инженерия. Руководство по применению ГОСТ Р ИСО/МЭК 12207 при управлении проектом».

36. ISO/IEC TR 19760:2003 «Systems Engineering — A Guide for the Application of ISO/IEC 15288 (System Life Cycle Processes)». Рус. версия: ГОСТ Р ИСО/МЭК ТО 19760:? «Системная инженерия. Руководство по применению ГОСТ Р ИСО/МЭК 15288 (Процессы жизненного цикла систем)».

Другая документация по разработке ПО и систем:

37. IEEE Guide to the Software Engineering Body of Knowledge (SWEBOK).— 2004 Version.— Los Alamos, CA: IEEE Computer Society, 2004.— 204 p.

Интернет — источники

38. Объектно-ориентированный анализ и проектирование (обозначение сайта — ООА&П / OOA&D).— URL: http://ooad.asf.ru/ .
39. Клуб разработчиков программных систем.— URL: http://www.caseclub.ru/ .
40. Сайт MAXKIR.com — переводы зарубежных публикаций К. Максимова, А. Максимова.— URL: http://www.maxkir.com/ .
41. Wikipedia.org — The Free Encyclopedia.— URL: http://en.wikipedia.org/ .— Сайт полезен для получения ссылок на материалы по нужной тематике. Не используйте информацию без их проверки по оригинальным источникам.
42. Европейский центр программирования.— URL: http://www.europrog.ru/ .
43. Agile Manifesto — Живой манифест.— URL: http://www.agilemanifesto.org/ .
44. PraxOS — Организационная система PraxOS.— URL: http://praxos.ru/ .
45. ИНТУИТ — Интернет-университет информационных технологий.— URL: http://www.intuit.ru/ .
46. Open Systems — Проект «Открытые системы».— URL: http://www.osp.ru/ .
47. CIT — Центр информационных технологий.— URL: http://www.citforum.ru/, http://www.citcity.ru/ и другие сайты.