# КОНСТРУИРОВАНИЕ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

**Учебное пособие**

# Оглавление

# Введение

В данном пособии представлено базовое описание процедур создания программного обеспечения с использованием методов верификации, кодирования и тестирования компонентов.

В главе 1 выделяются основные понятия и положения обеспечения процесса конструирования. В главе 2 приводятся процедуры по управлению процессом конструирования от планирования до выбора модели жизненного цикла программного обеспечения. В 3-й главе содержится материал, включающий в себя ряд практик и принципов конструирования, применение методологий разработки, тестирования программного обеспечения.

Пособие предназначено для студентов направления 09.03.04 «Программная инженерия», изучающих дисциплину «Конструирование программного обеспечения».

# 1. Введение в конструирование программного обеспечения

В настоящее время вычислительные системы находят все более и более широкое применение. При этом программное обеспечение (ПО) является неотъемлемой частью таких систем. Программные системы весьма сложны, например, операционные системы и системы автоматизированного проектирования, другие программы, как системы домашней бухгалтерии, наоборот, ясны и понятны широкому кругу пользователей.

При всем многообразии программ и программных комплексов у них есть одна общая черта – технологии разработки. В 1969 г. фирма IBM разделила аппаратную и программную части вычислительной системы, положив начало индустрии программного обеспечения, а также подходам, методам, средствам и технологиям разработки программ.

Большие программные средства, как правило, обладают всеми свойствами сложных систем. Они содержат большое количество (сотни и тысячи) компонент-модулей, тесно взаимосвязанных в процессе решения общей целевой задачи.

Для обеспечения взаимодействия компонент в едином комплексе широко используются иерархические структуры с несколькими уровнями группирования и подчиненности модулей.

Каждый модуль имеет свою целевую задачу и специфический частный критерий качества, как правило, не совпадающий с целевой задачей и критерием качества программного изделия в целом.

Особенно сложно в программном изделии, содержащем сотни модулей, обеспечить наилучшее использование ресурсов ЭВМ с точки зрения основного критерия эффективности при сохранении ряда частных показателей качества в допустимых пределах.

Многочисленность и сложность путей исполнения программ требует их высокой устойчивости как по отношению к ошибкам во входной информации, так и по отношению к внутренним сбоям ЭВМ, выполняющей программу, что определяется понятием «Надежность программного изделия».

Создание больших программных изделий с заданными характеристиками при ограниченных ресурсах требует проведения определенного комплекса мероприятий для достижения поставленной цели, который получил название «проект».

## 1.1 Понятие конструирования

Термин «конструирование программного обеспечения» описывает детальное создание рабочей программной системы посредством комбинации кодирования, верификации (проверки), модульного тестирования, интеграционного тестирования и отладки.

Данная область знаний связана с другими областями. Наиболее сильная связь существует с проектированием (Software Design) и тестированием (Software Testing). Причиной этого является то, что сам по себе процесс конструирования программного обеспечения затрагивает важные аспекты деятельности по проектированию и тестированию. Кроме того, конструирование отталкивается от результатов проектирования, а тестирование (в любой своей форме) предполагает работу с результатами конструирования. Достаточно сложно определить границы между проектированием, конструированием и тестированием, так как все они связаны в единый комплекс процессов жизненного цикла и, в зависимости от выбранной модели жизненного цикла и применяемых методов (методологии), такое разделение может выглядеть по-разному.

## 1.2 Связь конструирования с прочими стадиями жизненного цикла

Хотя ряд операций по проектированию детального дизайна может происходить до стадии конструирования, большой объем такого рода проектных работ происходит параллельно с конструированием или как его часть. Это есть суть связи с областью знаний «Проектирование программного обеспечения».

В свою очередь на протяжении всей деятельности по конструированию, инженеры используют модульное и интеграционное тестирование. Таким образом, данная область знаний связана с «Тестированием программного обеспечения».

В процессе конструирования обычно создается большая часть активов программного проекта – конфигурационных элементов. Поэтому в реальных проектах просто невозможно рассматривать деятельность по конструированию в отрыве от области знаний «Конфигурационного управления» (Software Configuration Management).

Так как конструирование невозможно без использования соответствующего инструментария и, вероятно, данная деятельность является наиболее инструментально-насыщенной, важную роль в конструировании играет область знаний «Инструменты и методы программной инженерии» (Software Engineering Tools and Methods).

Безусловно, вопросы обеспечения качества значимы для всех областей знаний и этапов жизненного цикла. В то же время код является основным результирующим элементом программного проекта. Таким образом, явно напрашивается и присутствует связь обсуждаемых вопросов с областью знаний «Качество программного обеспечения» (Software Quality).

Из связанных дисциплин программной наиболее тесная и естественная связь данной области знаний существует с

компьютерными науками. Именно в них обычно рассматриваются вопросы построения и использования алгоритмов и практик кодирования. Наконец, конструирование касается и управления проектами, причем в той степени, насколько деятельность по управлению конструированием важна для достижения результатов конструирования.

*Технология конструирования программного обеспечения* (ТКПО) – система инженерных принципов для создания экономичного ПО, которое надежно и эффективно работает в реальных компьютерах.

Различают методы, средства и процедуры ТКПО. Методы обеспечивают решение следующих задач:

- планирование и оценка проекта;
- анализ системных и программных требований;
- проектирование алгоритмов, структур данных и программных структур;
- кодирование;
- тестирование;
- сопровождение.

Средства (утилиты) ТКПО обеспечивают автоматическую поддержку методов. В целях совместного применения утилиты могут объединяться в системы автоматизированного конструирования ПО. Такие системы принято называть CASE-системами (Computer Aided Software Engineering).

Процесс конструирования программного обеспечения состоит из последовательности шагов, использующих методы, утилиты и процедуры. Эти последовательности шагов часто называют парадигмами ТКПО.

Применение парадигм ТКПО гарантирует систематический, упорядоченный подход к промышленной разработке, использованию

и сопровождению ПО. Фактически, парадигмы вносят в процесс создания ПО организующее инженерное начало, необходимость которого трудно переоценить.

Проектирование программного обеспечения начинается, собственно, с его конструирования, которое определяет стратегию для его внутреннего проектирования – для этапа программирования. Заметим, что этот этап выполняется без использования языка программирования, но с ориентацией на определенный программный инструмент разработки ПО.

В процессе конструирования программного изделия осуществляют:

- функциональную декомпозицию решаемой задачи, на основе которой определяется архитектура системы, представляющей задачу;
- внешнее проектирование программного обеспечения, выражающееся в форме его внешнего взаимодействия с пользователем;
- проектирование базы данных, если это необходимо;
- проектирование архитектуры программного обеспечения, т. е. определение множества объектов или модулей, функционально связанных с решаемой задачей, включая сопряжения между ними и требования к ним.

Одной из наиболее опасных болезней жизненного цикла разработки программного изделия является синдром ползущего проекта, или «оползня». Он проявляется, когда конструирование программного изделия проведено неполно и недостаточно; неверно сконструированы отдельные аспекты проекта.

В этом случае, по мере создания программного обеспечения, пользователи, рассматривая работу отдельных готовых ветвей программы, будут просить внести некоторые усовершенствования, ссылаясь на неясные описания этого участка проекта во внешней

спецификации. Глобальные изменения уже разработанных частей программы производить будет нельзя, а изменения и усовершенствования в нее внести надо.

Данная ситуация в первую очередь приведет к перерасходу временного лимита на создание отдельных частей проекта и нестабильности работы программного обеспечения из-за иска-жения или выпадения отдельных функциональных конструкций из общей строгой схемы всего проекта.

Основные принципы проектирования программного обеспече-ния можно представить в виде следующей схемы (рис 1):


Рис. 1. Принципы проектирования

Предварительный внешний проект высокого уровня предполагает определение взаимодействия будущего программного продукта с внешним миром (обычно с пользователем), но не рассматривает многие его мельчащие детали, такие как форматы ввода-вывода. Последнее уточняется в детальном внешнем проектировании.

## 1.3 Структура жизненного цикла программы

Комплексы программ создаются, эксплуатируются и развиваются во времени. Жизненный цикл программных средств (ПС) включает в себя все этапы развития: от возникновения потребности в программе, определения целевого назначения, до полного прекращения использования этого программного средства, вследствие его морального старения или потери необходимости решения соответствующих задач.

По длительности жизненного цикла ПС можно разделить на два класса:

- с малым временем жизни (гибкий (мягкий) подход к их созданию и использованию);

- с большим временем жизни (жесткий промышленный подход регламентированного проектирования и эксплуатации промышленных изделий).

**Программы с малой длительностью эксплуатации**

Создаются в основном для решения научных и инженерных задач, для получения конкретных результатов вычислений. Разрабатываются одним специалистом или маленькой группой, не предназначены для тиражирования и передачи для последующего использования в другие коллективы. ЖЦ таких программ состоит из длительного интервала системного анализа и формализации проблемы, значительного этапа проектирования программ и относительно небольшого времени эксплуатации и получения результатов.

Требования, предъявляемые к функциональным и конструктивным характеристикам, не формализуются, отсутствуют оформленные испытания программ, и показатели их качества контролируются только разработчиками, в соответствии с неформальными представлениями.

Сопровождение и модификация таких программ не нужны, и их ЖЦ завершается после получения результатов вычислений. Основные затраты в ЖЦ таких программ приходятся на этапы системного анализа и проектирования, которые продолжаются от месяца до одного или двух лет. В результате ЖЦ редко превышает три года, т. е. основное время идет на анализ и проектирование.

**Программы с большой длительностью эксплуатации**

Создаются для регулярной обработки информации и управления в процессе функционирования сложных вычислительных систем. Программы такого класса допускают тиражирование. Они сопровождаются документацией как промышленные изделия и представляют собой отчуждаемый программный продукт.

Проектированием и эксплуатацией программного средства могут заниматься большие коллективы специалистов, для чего необходима формализация требуемых технических характеристик комплекса программ и их компонент. А также формализованные испытания и определение достигнутых показателей качества программных средств.

ЖЦ таких программных средств составляет 10–20 лет, из которых 70–90% приходиться на эксплуатацию и сопровождение. Вследствие массового тиражирования и длительного сопровождения совокупные затраты в процессе эксплуатации и сопровождения могут значительно превышать затраты на системный анализ и проектирование.

## 1.4 Стандарты в конструировании

Стандарты, которые напрямую применяются при конструировании, включают:

- коммуникационные методы (например, стандарты форматов документов и оформления содержания);

- языки программирования и соответствующие стили кодирования (например, Java Language Specification, являющийся частью стандартной документации JDK – Java Development Kit и Java Style Guide, предлагающий общий стиль кодирования для языка Java);

- платформы (например, стандарты программных интерфейсов для вызовов функций операционной среды, такие как прикладные программные интерфейсы платформы Windows — Win 32 API, Application Programming Interface или .NET Framework SDK);

- инструменты (не в терминах сред разработки, но возможных средств конструирования, например, UML как один из стандартов для определения нотаций для диаграмм, представляющих структуру кода и его элементов или некоторые аспекты поведения кода).

Применяемые стандарты можно условно разделить на две большие группы: внешние и внутренние.

**Использование внешних стандартов.** Конструирование зависит от внешних стандартов, связанных с языками программирования, используемым инструментальным обеспечением, техническими интерфейсами и взаимным влиянием Конструирования программного обеспечения и других областей знаний программной инженерии (в том числе, связанных дисциплин, например, управления проектами). Стандарты создаются разными источниками, например, консорциумом OMG – Object Management Group (в частности, стандарты CORBA, UML, MDA), международными организациями по стандартизации, такими как ISO/IEC, IEEE, TMF, производителями платформ, операционных сред и т. д. (например, Microsoft, Sun

Microsystems, CISCO, NOKIA), производителями инструментов, систем управления базами данных и т. п. (Borland, IBM, Microsoft, Sun, Oracle). Понимание этого факта позволяет определить достаточный и полный набор стандартов, применяемых в проектной команде или организации в целом.

**Использование внутренних стандартов**. Определенные стандарты, соглашения и процедуры могут быть также созданы внутри организации или даже проектной команды. Эти стандарты поддерживают координацию между определенными видами деятельности, группами операций, минимизируют сложность (в том числе при взаимодействии членов проектной группы и за ее пределами), могут быть связаны с вопросами ожидания и обработки изменений, рисков и вопросами конструирования для проверки и дальнейшего тестирования. В сочетании с внешними стандартами, внутренние стандарты призваны определить общие правила игры для всех членов проектной команды, договорившись о терминах, процедурах и других значимых соглашениях, вне зависимости от степени формализации процессов конструирования, в частности, и процессов жизненного цикла в общем случае.

## Список контрольных вопросов

1. Понятие конструирования программных средств.
2. Место конструирования в жизненном цикле программного обеспечения.
3. Стандарт ГОСТ 34.601-90.
4. Стандарт ISO/IEC 12207:1995.

# 2. Управление конструированием

## 2.1 Планирование в конструировании

### 2.1.1 Метод оценки и обзора программы

Согласно PMBoK оценка длительности операции может выполняться следующими методами и инструментами:

1. Экспертная оценка

«Экспертные оценки, основанные на исторической информации, могут предоставить информацию об оценке длительности или о рекомендованной максимальной длительности операций из предыдущих подобных проектов».

2. Оценка по аналогам

«Оценка по аналогам подразумевает использование таких параметров, как длительность, бюджет, размер, вес и сложность из предыдущих подобных проектов в качестве основы для оценки тех же параметров или измерений будущего проекта».

3. Параметрическая оценка

«Параметрическая оценка использует статистические взаимосвязи между историческими данными и прочими переменными (например, площадью в квадратных метрах в строительстве) для численной оценки параметров операции, таких как стоимость, бюджет и длительность.

Длительность операций может быть количественно определена путем умножения количества работ, которые необходимо выполнить, на количество рабочего времени, затрачиваемое на производство единицы работы».

4. Оценки по трем точкам

PERT – это аббревиатура от английских Program Evaluation and Review Technique (метод оценки и обзора программы), некая

технология оценки и пересмотра программы, которая базируется на идее сетевого планирования. Для того чтобы рассчитать для каждой работы дату начала и дату окончания, нужно оценить длительности всех работ с оценкой того, сколько они займут времени. Только после этого эту модель можно рассчитать и понять сроки окончания работ. Здесь есть достаточно важный момент. Менеджер самостоятельно сделать расчет по данному методу не в состоянии. У него, вполне естественно, не хватает компетенций. Он спрашивает исполнителей, которые, мягко говоря, не заинтересованы говорить правду, потому что им выгодно заложить свои временные резервы. При разработке данной техники предложили запрашивать не одну оценку, а три:

- оптимистическую;
- пессимистическую;
- наиболее вероятностную.

Запрос ответственному ресурсу по задаче на оценку длительности производится по указанной выше последовательности. Дальше, исходя из полученных ответов, с заранее заданной вероятностью производится расчет соответствующих сроков. При этом рекомендуется применять технику скользящего пересмотра сроков с учетом достигнутых результатов.

Допустим, команда отработала по проекту полгода, получены фактические данные по продолжительности выполненных этапов. Менеджер получает более точные оценки на оставшиеся невыполненными работы. Это называется еще «планированием по методу бегущей волны». Так постепенно выявляется все более точная продолжительность всей проектной реализации еще задолго до ее окончания. В результате достигается существенная экономия на стыках: заканчивается одна работа – начинается другая, а поэлементные ошибки нивелируются.

Таким образом, PERT предназначен для оптимизации длительности проектов за счет интересных логико-управленческих решений и, в большей степени, благодаря применению статистических методов. Техника позволяет при расчете продолжительности работ применить вероятностный подход с использованием так называемого среднего значения β-распределения.

Как уже было отмечено выше, техника PERT предполагает использование эффектов аппроксимации β-распределения. Установлен факт, что такое распределение гибко реагирует на численный ряд и позволяет анализировать эмпирические данные, не следующие за нормальным распределением.

Длительность каждой работы в проекте имеет ограничения от наилучшего до наихудшего значения, а средний показатель подлежит расчету. Отставания от графика, вызванные ошибкой в момент установления длительности или субъективными причинами, имеют свойство продолжаться и далее. Причем время операций может отклоняться либо в сторону нижнего, либо в сторону верхнего предела.

Ряд распределенных показателей длительности доступен к анализу как сумма средневзвешенных значений для операций, включенных в состав критического пути. Получение такого средневзвешенного показателя вместе с отклонением дает PM возможность произвести расчет вероятности возможной продолжительности как операций, так и проекта. В методе применяется следующая формула расчета средневзвешенного времени операции:

$$tE = (tO + 4tM + tP) / 6,$$

где tM – наиболее вероятное время операции; tO – оптимистичное время операции; tP – пессимистичное время операции.

Будучи зависимой от предполагаемого распределения значений в диапазоне трех оценок, ожидаемая длительность tE рассчитывается по представленной формуле. Две наиболее распространенные формулы, применяемые в технике, – треугольное распределение и уже рассмотренное β-распределение.

Треугольное распределение, в свою очередь, выглядит следующим образом:

$$tE = (tO + tM + tP) / 3.$$

Важные особенности методологии PERT

Представленные выше логика и математика метода дают достаточные основания для начала его применения. Вместе с тем, как у каждой методики, у техники PERT имеются тонкости применения на практике, которые обязательно нужно учитывать для успешного ее внедрения. Эти особенности заключаются в следующих аспектах:

- метод пока еще используется на масштабных проектах, активно применяется в строительстве, оборонной промышленности, в военной инженерии и логистике, самый яркий пример – строительство космодрома «Восточный», а классикой считается разработка ракетной системы Polaris (США, 1958 г.) – в момент собственно зарождения данного метода;

- для получения лучших результатов от применения техники PERT помимо участников проектной команды целесообразно привлекать также и опытных в предмете экспертов, это позволит снизить отклонения оптимистичной, пессимистичной и наиболее вероятностной оценок;

- важно помнить, что техника по своему методу занижает предполагаемую продолжительность исполнения проектной задачи, увеличение числа одновременно выполняемых работ увеличивает размер ошибки;

- разброс вероятности по критическому пути увеличивается;

- техника не воспринимает существующие ограничения на ресурсы и действия PM, который стремится обеспечить проектный результат в назначенные сроки, необходимо допустить, что все случайные величины продолжительностей работ критического пути независимы, тогда применение инструмента даст лучший результат.

5. Анализ резервов

Оценки длительности могут включать в себя резервы на возможные потери (иногда называемые «временными резервами», или «буферами») в рамках общего расписания проекта для устранения неопределенности расписания. Резерв на возможные потери может выражаться в процентах от оценочной длительности операции, в фиксированном числе рабочих периодов или может быть рассчитан с помощью методов количественного анализа.

По мере поступления более точной информации о проекте резервы на возможные потери могут быть использованы, сокращены или устранены.

### 2.1.2 Покер планирования

Покер планирования (от англ. *planning poker*, а также от англ. *scrum poker*) – техника оценки, основанная на достижении договоренности, главным образом используемая для оценки сложности предстоящей работы или относительного объема решаемых задач при разработке программного обеспечения. Это разновидность метода Wideband Delphi.

Она обычно используется в гибкой методологии разработки, в частности, в методологии экстремального программирования.

Метод впервые был описан Джеймсом Греннингом (James Grenning) в 2002 году и позднее популяризован Майком Коном (Mike Cohn) в книге «Agile Estimating and Planning».

Описание процесса оценки:

**Подготовка**

Для проведения покера планирования необходимо подготовить список обсуждаемых функций и несколько колод пронумерованных карт. Список функций либо пользовательские истории описывают разрабатываемое программное обеспечение. Карты в колодах должны быть пронумерованы. Обычно колода содержит карты, содержащие числа Фибоначчи, включая ноль: 0, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89; другие разновидности колод могут использовать аналогичные последовательности.


Рис. 2. Колода карт для покера планирования

Аргумент в пользу применения последовательности Фибоначчи – отражение типичной неопределенности в обсуждении самых важных и больших пунктов.

Одна из имеющихся в продаже колод содержит такую последовательность: 0, ½, 1, 2, 3, 5, 8, 13, 20, 40,100 и иногда знак вопроса «?», означающий неуверенность, и чашку кофе, означающую требование перерыва. Некоторыми организациями используются обычные игровые карты, включающие туз, 2, 3, 5, 8 и короля. Король буквально означает: «Данный пункт слишком большой или его слишком сложно оценить». Выбрасывание короля завершает обсуждение пункта в текущем круге (англ. *sprint*).

По желанию может использоваться таймер, чтобы устанавливать лимит времени одного круга.

**Процедура проведения**

Каждому участнику обсуждения выдается по одной колоде карт. Все колоды идентичны друг другу. Обсуждение проводится следующим образом.

Ведущий (англ. *moderator*), не участвующий в обсуждении, ведет собрание. Менеджер проекта (англ. *product manager*) представляет краткие обзоры каждого из пунктов. Команда может задавать вопросы и вести обсуждение предложений и рисков. Итог обсуждения записывается менеджером проекта.

Участники выбирают по одной карте и кладут их рубашкой вверх, показывая таким образом, что выбор сделан. Числовые достоинства карт могут использоваться по-разному: они могут означать количество дней, наиболее подходящие дни или относительные единицы сложности (англ. *story points*). Во время обсуждения достоинствам не должны приписываться новые значения в зависимости от размера функций с целью избегания эффекта привязки. Каждый участник называет свою карту и переворачивает ее. Участникам с высокими и низкими оценками дается возможность высказаться и обосновать свою оценку.

Процесс обсуждения продолжается до тех пор, пока не будет достигнут консенсус. Голос участника, который, скорее всего, будет владеть разработкой, имеет больший вес в «голосовании на основе консенсуса».

Таймер используется для обеспечения структурированности обсуждения; ведущий или менеджер проекта может в любое время перезапустить таймер, по истечении времени все обсуждения должны быть прекращены, затем начинается новый круг покера.

Выступления участников повторяются вновь и вновь. Карты пронумерованы так, что чем больше цифра, тем больше неопределенность. Так, если разработчик желает выбрать 6, но он не до конца уверен, он выберет 5, либо может предусмотрительно выбрать 8.

Достоинства метода:

- покер планирования – это средство оценки проектов по разработке программного обеспечения. Эта техника минимизирует эффект привязки путем опроса каждого из участников команды таким образом, что никто не знает чужого решения до одновременного оглашения выбора каждого из участников. Исследование К. Молеккен-Эствольда и Н. Хаугена показало, что оценки, полученные с помощью покера планирования, были менее оптимистичными и более точными, чем оценки, полученные с помощью простого сложения отдельных оценок аналогичных задач;

- избегание эффекта привязки. Эффект привязки возникает, когда команда открыто обсуждает оценки. Команда обычно имеет в своем составе как сдержанных, так и импульсивных участников, могут быть участники, у которых есть определенные планы; разработчики, вероятно, захотят как можно больше времени заниматься работой над проектом, а владелец продукта или заказчик, вероятно, захочет, чтобы работа была закончена как можно скорее. Оценка становится подверженной эффекту привязки, когда владелец продукта говорит нечто подобное: «Я думаю, это несложная работа, вряд ли это займет больше пары недель». Либо когда разработчик говорит: «Думаю, нам нужно лучше стараться; решение проблем с бэк-эндом, которые у нас были, могло затянуться на месяцы». Если начинающий обсуждение говорит: «Думаю, это займет 50 дней», – он сразу устанавливает рамки мышления остальных участников; возникает эффект привязки, т. е. число 50 подсознательно будет

отправной точкой для всех участников. Те, кто хотел назвать число 100, захотят уменьшить свою оценку, а те, кто задумал число 10, захотят увеличить ее. Это становится серьезной проблемой, если число 50 произносится влиятельным участником в то время, когда остальная команда преимущественно останавливает свой выбор на больших или меньших значениях. Из-за эффекта привязки остальные участники могут – сознательно или нет – отказаться от своей точки зрения, чтобы выразить их начальное единство; на самом деле, они могут это сделать, даже чтобы просто показать, что они думают так же. Влияние различных мнений, не сосредоточенных на качественном выполнении работы, может быть опасным для оценки.

Покер планирования выявляет потенциально влиятельного участника команды, изолируя его мнение от других участников группы. Затем необходимо, чтобы участник аргументировал свой выбор, если он не совпадает с превалирующим мнением. Если участники группы могут выражать свою сплоченность таким образом, они более склонны верить в свои первоначальные оценки.

Если у влиятельного участника есть хорошие аргументы для спора, все остальные будут видеть смысл и прислушиваться, но, по крайней мере, остальные участники не будут подвержены эффекту привязки; вместо этого они должны будут исходить только из разумных соображений.

## 2.2 Стратегии конструирования программного обеспечения

Существуют три стратегии конструирования ПО:

1. Однократный проход (водопадная стратегия) – линейная последовательность этапов конструирования.
2. Инкрементная стратегия. В начале процесса определяются все пользовательские и системные требования, оставшаяся часть конструирования выполняется в виде последовательности

версий. Первая версия реализует часть запланированных возможностей, следующая версия реализует дополнительные возможности и т. д., пока не будет получена полная система.

3. Эволюционная стратегия. Система также строится в виде последовательности версий, но в начале процесса определены не все требования. Требования уточняются в результате разработки версий.

Характеристики стратегий конструирования ПО в соответствии с требованиями стандарта IEEE/EIA 12207.2 приведены в табл. 1.

Таблица 1

Стратегии конструирования

| Стратегия конструирования | В начале процесса определены все требования | Множество циклов конструирования | Промежуточное ПО распространяется? |
|---|---|---|---|
| Однократный проход | Да | Нет | Нет |
| Инкрементная (запланированное улучшение продукта) | Да | Да | Может быть |
| Эволюционная | Нет | Да | Да |

Легко обнаружить, что в разное время и в разных источниках приводится разный список моделей и их интерпретация. Например, ранее инкрементальная модель понималась как построение системы в виде последовательности сборок (релизов), определенной в соответствии с заранее подготовленным планом и заданными (уже сформулированными) и неизменными требованиями. Сегодня об инкрементальном подходе чаще всего говорят в контексте постепенного наращивания функциональности создаваемого продукта.

Может показаться, что индустрия пришла, наконец, к общей «правильной» модели. Однако каскадная модель, многократно «убитая» и теорией, и практикой, продолжает встречаться в реальной

жизни. Спиральная модель является ярким представителем эволюционного взгляда, но в то же время представляет собой единственную модель, которая уделяет явное внимание анализу и предупреждению рисков. Коротко рассмотрим каждую из моделей жизненного цикла.

## 2.3 Классический жизненный цикл

Старейшей парадигмой процесса разработки ПО является классический жизненный цикл.

Очень часто классический жизненный цикл называют каскадной, или водопадной, моделью, подчеркивая, что разработка рассматривается как последовательность этапов, причем переход на следующий, иерархически нижний этап наступает только после полного завершения работ на текущем этапе.

Охарактеризуем содержание основных этапов. Подразумевается, что разработка начинается на системном уровне и проходит через анализ, проектирование, кодирование, тестирование и сопровождение. При этом моделируются действия стандартного инженерного цикла.

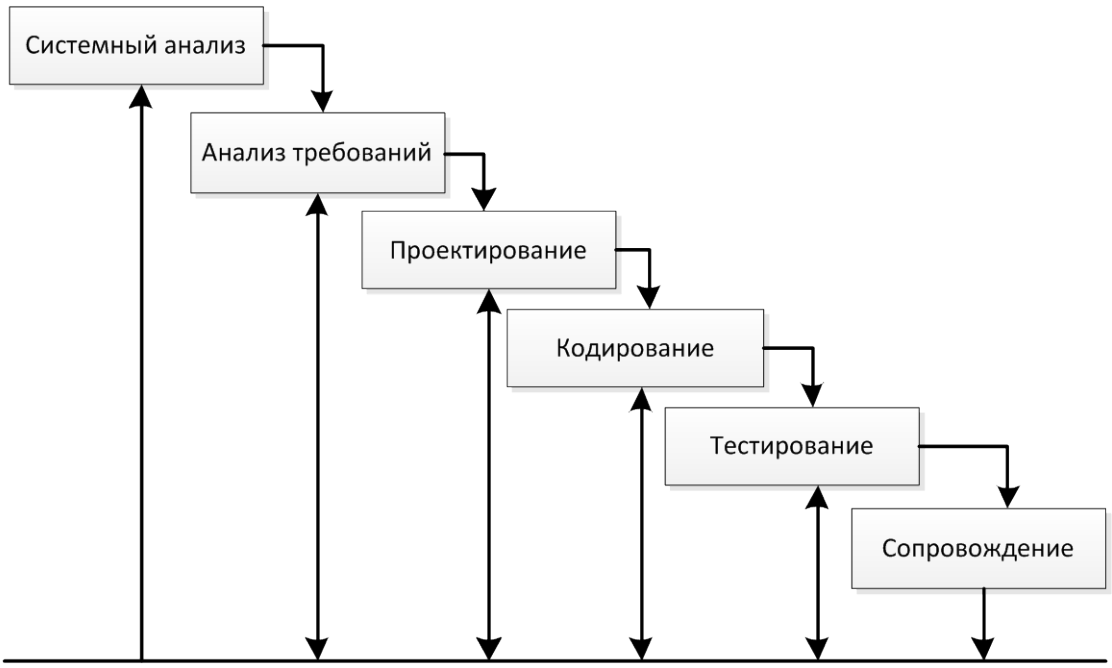


Рис. 3. Классический жизненный цикл разработки ПО

Системный анализ задает роль каждого элемента в компьютерной системе, взаимодействие элементов друг с другом. Поскольку ПО является лишь частью большой системы, то анализ начинается с определения требований ко всем системным элементам и назначения подмножества этих требований программному «элементу». Необходимость системного подхода явно проявляется, когда формируется интерфейс ПО с другими элементами (аппаратурой, людьми, базами данных). На этом же этапе начинается решение задачи планирования проекта ПО. В ходе планирования проекта определяются объем проектных работ и их риск, необходимые трудозатраты, формируются рабочие задачи и план-график работ.

Анализ требований относится к программному элементу – программному обеспечению. Уточняются и детализируются его функции, характеристики и интерфейс.

Все определения документируются в спецификации анализа. Здесь же завершается решение задачи планирования проекта.

Проектирование состоит в создании представлений:

- архитектуры ПО;
- модульной структуры ПО;
- алгоритмической структуры ПО;
- структуры данных;
- входного и выходного интерфейса (входных и выходных форм данных).

Исходные данные для проектирования содержатся в спецификации анализа, т. е. в ходе проектирования выполняется трансляция требований к ПО во множество проектных представлений. При решении задач проектирования основное внимание уделяется качеству будущего программного продукта.

Кодирование состоит в переводе результатов проектирования в текст на языке программирования.

Тестирование – выполнение программы для выявления дефектов в функциях, логике и форме реализации программного продукта.

Сопровождение – это внесение изменений в эксплуатируемое ПО. Цели изменений:

- исправление ошибок;

- адаптация к изменениям внешней для ПО среды;

- усовершенствование ПО по требованиям заказчика.

Сопровождение ПО состоит в повторном применении каждого из предшествующих шагов (этапов) жизненного цикла к существующей программе, но не в разработке новой программы.

Как и любая инженерная схема, классический жизненный цикл имеет достоинства и недостатки.

Достоинства классического жизненного цикла:

- дает план и временной график по всем этапам проекта, упорядочивает ход конструирования.

Недостатки классического жизненного цикла:

- реальные проекты часто требуют отклонения от стандартной последовательности шагов;

- цикл основан на точной формулировке исходных требований к ПО (реально в начале проекта требования заказчика определены лишь частично);

- результаты проекта доступны заказчику только в конце работы.

## 2.4 Инкрементная модель

Инкрементная модель является классическим примером инкрементной стратегии конструирования. Она объединяет элементы последовательной водопадной модели с итерационной философией макетирования.

Каждая линейная последовательность здесь вырабатывает поставляемый инкремент ПО. Например, ПО для обработки слов в 1-м инкременте реализует функции базовой обработки файлов, функции редактирования и документирования; во 2-м инкременте – более сложные возможности редактирования и документирования; в 3-м инкременте – проверку орфографии и грамматики; в 4-м инкременте – возможности компоновки страницы.

Первый инкремент приводит к получению базового продукта, реализующего базовые требования (правда, многие вспомогательные требования остаются нереализованными).

План следующего инкремента предусматривает модификацию базового продукта, обеспечивающую дополнительные характеристики и функциональность.

По своей природе инкрементный процесс итеративен, но в отличие от макетирования, инкрементная модель обеспечивает на каждом инкременте работающий продукт.

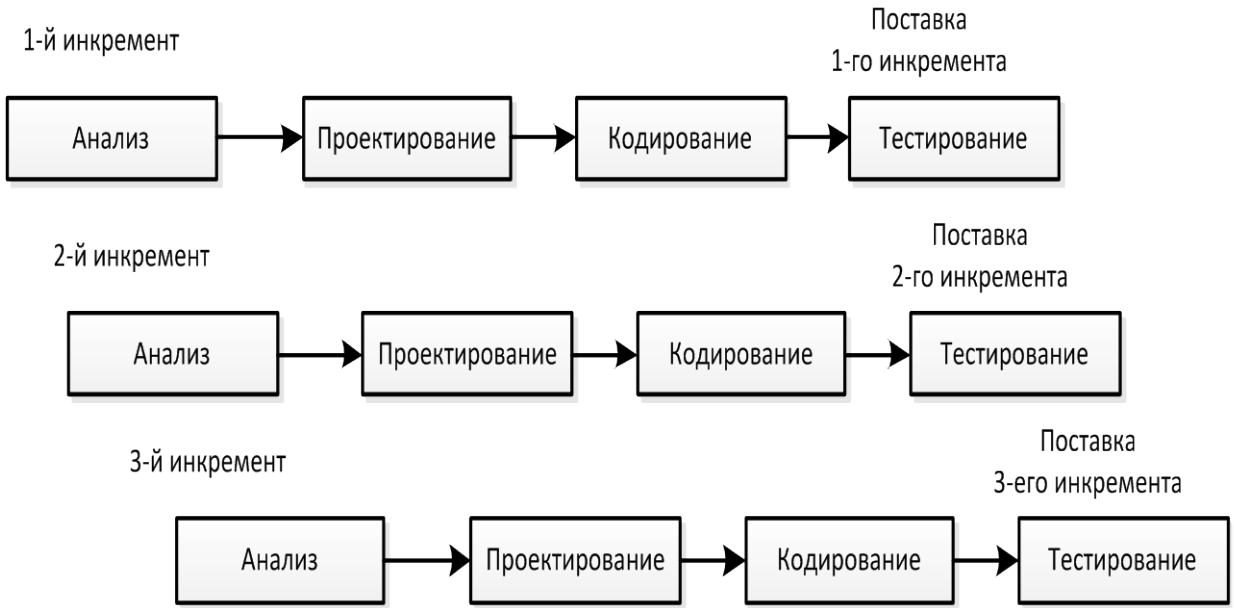


Рис. 4. Инкрементная модель

### 2.4.1 Быстрая разработка приложений

Модель быстрой разработки приложений RAD (Rapid Application Development) – второй пример применения инкрементной стратегии конструирования.

RAD-модель обеспечивает экстремально короткий цикл разработки. RAD – высокоскоростная адаптация линейной последовательной модели, в которой быстрая разработка достигается за счет использования компонентно-ориентированного конструирования. Если требования полностью определены, а проектная область ограничена, RAD-процесс позволяет группе создать полностью функциональную систему за очень короткое время (60–90 дней). RAD-подход ориентирован на разработку информационных систем и выделяет следующие этапы:

- бизнес-моделирование. Моделируется информационный поток между бизнес-функциями. Ищется ответ на следующие вопросы: какая информация руководит бизнес-процессом? Какая информация генерируется? Кто генерирует ее? Где информация применяется? Кто обрабатывает ее?;

- моделирование данных. Информационный поток, определенный на этапе бизнес-моделирования, отображается в набор объектов данных, которые требуются для поддержки бизнеса. Идентифицируются характеристики (свойства, атрибуты) каждого объекта, определяются отношения между объектами;

- моделирование обработки. Определяются преобразования объектов данных, обеспечивающие реализацию бизнес функций. Создаются описания обработки для добавления, модификации, удаления или нахождения (исправления) объектов данных;

- генерация приложения. Предполагается использование методов, ориентированных на языки программирования 4-го поколения. Вместо создания ПО с помощью языков программирования 3-го

поколения, RAD-процесс работает с повторно используемыми программными компонентами или создает повторно используемые компоненты. Для обеспечения конструирования используются утилиты автоматизации;

- тестирование и объединение. Поскольку применяются повторно используемые компоненты, многие программные элементы уже протестированы. Это уменьшает время тестирования (хотя все новые элементы должны быть протестированы).

Применение RAD возможно в том случае, когда каждая главная функция может быть завершена за 3 месяца. Каждая главная функция адресуется отдельной группе разработчиков, а затем интегрируется в целую систему.

Применение RAD имеет и недостатки, и ограничения:

- для больших проектов в RAD требуются существенные людские ресурсы (необходимо создать достаточное количество групп);

- RAD применима только для таких приложений, которые могут декомпозироваться на отдельные модули и в которых производительность не является критической величиной;

- RAD не применима в условиях высоких технических рисков (т. е. при использовании новой технологии).

## 2.5 Спиральная модель

Спиральная модель – классический пример применения эволюционной стратегии конструирования.

Спиральная модель (автор Барри Боэм, 1988) базируется на лучших свойствах классического жизненного цикла и макетирования, к которым добавляется новый элемент – анализ риска, отсутствующий в этих парадигмах.

Модель определяет четыре действия, представляемые четырьмя квадрантами спирали (рис. 5).

1. Планирование – определение целей, вариантов и ограничений.
2. Анализ риска – анализ вариантов и распознавание/выбор риска.
3. Конструирование – разработка продукта следующего уровня.
4. Оценивание – оценка заказчиком текущих результатов конструирования.

Интегрирующий аспект спиральной модели очевиден при учете радиального измерения спирали. С каждой итерацией по спирали (продвижением от центра к периферии) строятся все более полные версии ПО.

В первом витке спирали определяются начальные цели, варианты и ограничения, распознается и анализируется риск. Если анализ риска показывает неопределенность требований, на помощь разработчику и заказчику приходит макетирование.


Рис. 5. Спиральная модель:
1 – начальный сбор требований и планирование проекта; 2 – та же работа, но на основе рекомендаций заказчика; 3 – анализ риска на основе начальных требований; 4 – анализ риска на основе реакции заказчика; 5 – переход к комплексной системе; 6 – начальный макет системы; 7 – следующий уровень макета; 8 – сконструированная система; 9 – оценивание заказчиком

Для дальнейшего определения проблемных и уточненных требований может быть использовано моделирование. Заказчик оценивает инженерную (конструкторскую) работу и вносит предложения по модификации (квадрант оценки заказчиком). Следующая фаза планирования и анализа риска базируется на предложениях заказчика. В каждом цикле по спирали результаты анализа риска формируются в виде «продолжать, не продолжать». Если риск слишком велик, проект может быть остановлен.

В большинстве случаев движение по спирали продолжается, с каждым шагом продвигая разработчиков к более общей модели системы. В каждом цикле по спирали требуется конструирование (нижний правый квадрант), которое может быть реализовано классическим жизненным циклом или макетированием. Заметим, что количество действий по разработке (происходящих в правом нижнем квадранте) возрастает по мере продвижения от центра спирали.

Достоинства спиральной модели:

1) наиболее реально (в виде эволюции) отображает разработку программного обеспечения;
2) позволяет явно учитывать риск на каждом витке эволюции разработки;
3) включает шаг системного подхода в итерационную структуру разработки;
4) использует моделирование для уменьшения риска и совершенствования программного изделия.

Недостатки спиральной модели:

1) новизна (отсутствует достаточная статистика эффективности модели);
2) повышенные требования к заказчику;
3) трудности контроля и управления временем разработки.

## 2.6 Компонентно-ориентированная модель

Компонентно-ориентированная модель является развитием спиральной модели и тоже основывается на эволюционной стратегии конструирования.

В этой модели конкретизируется содержание квадранта конструирования – оно отражает тот факт, что в современных условиях новая разработка должна основываться на повторном использовании существующих программных компонентов.

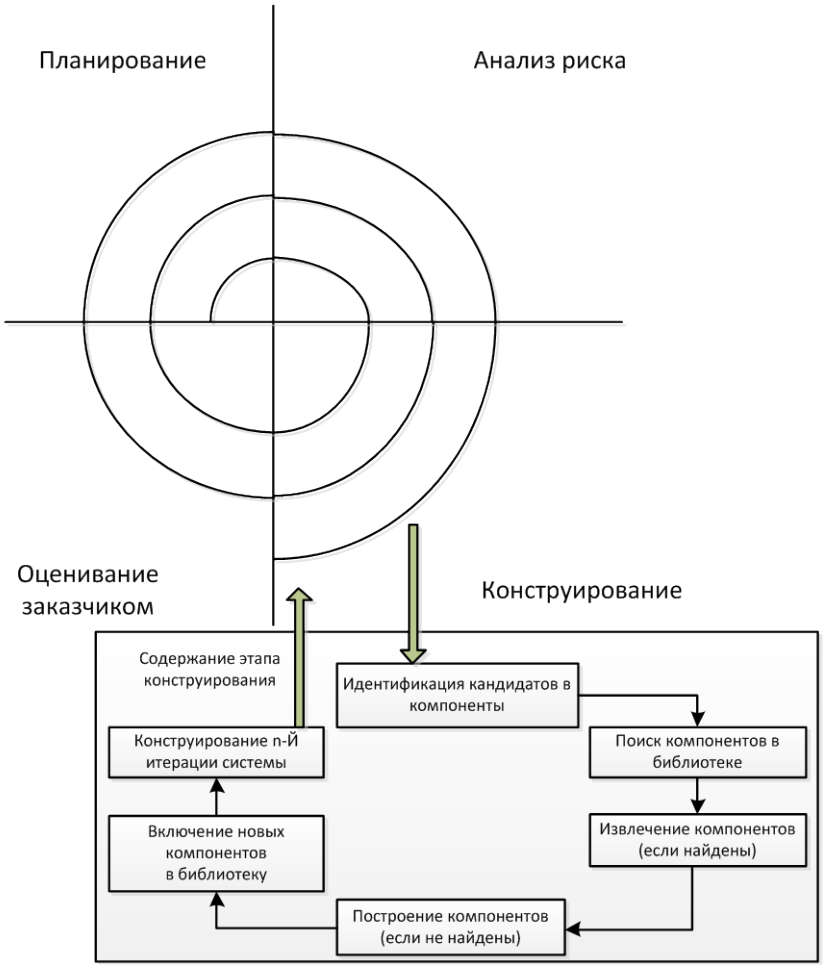


Рис. 6. Компонентно-ориентированная модель

Программные компоненты, созданные в реализованных программных проектах, хранятся в библиотеке. В новом программном

проекте, исходя из требований заказчика, выявляются кандидаты в компоненты. Далее проверяется наличие этих кандидатов в библиотеке. Если они найдены, то компоненты извлекаются из библиотеки и используются повторно. В противном случае создаются новые компоненты, они применяются в проекте и включаются в библиотеку.

Достоинства компонентно-ориентированной модели:

1) уменьшает на 30% время разработки программного продукта;
2) уменьшает стоимость программной разработки до 70%;
3) увеличивает в полтора раза производительность разработки.

## Список контрольных вопросов

1. Стандарты и модели конструирования.
2. Модели жизненного цикла.
3. Классический (каскадный) жизненный цикл.
4. Основные принципы макетирования.
5. Какие этапы содержит инкрементная модель?
6. Быстрая разработка приложений.
7. Какие квадранты представлены в спиральной модели?
8. Компонентно-ориентированная модель.
9. Анализ требований и определение спецификаций программного обеспечения.
10. Планирование конструирования.
11. Измерения в конструировании.
12. Проектирование в конструировании.

# 3. Практика использования

## 3.1 Модульность

Модуль – фрагмент программного текста, являющийся строительным блоком для физической структуры системы. Как правило, модуль состоит из интерфейсной части и части-реализации.

Модульность – свойство системы, которая может подвергаться декомпозиции на ряд внутренне связанных и слабо зависящих друг от друга модулей.

По определению Г. Майерса, модульность – свойство ПО, обеспечивающее интеллектуальную возможность создания сколь угодно сложной программы. Проиллюстрируем эту точку зрения.

Пусть $C(x)$ – функция сложности решения проблемы $x$, $T(x)$ – функция затрат времени на решение проблемы $x$. Для двух проблем $p_1$и $p_2$из соотношения $C(p_1) > C(p_2)$ следует, что $T(p_1) > T(p_2)$. Этот вывод интуитивно ясен: решение сложной проблемы требует большего времени.

Из практики решения проблем человеком следует

$C(p_1 + p_2) > C(p_1) + C(p_2)$.

Отсюда запишем вывод:

$T(p_1 + p_2) > T(p_1) + C(p_2)$.

Выведенное соотношение – это обоснование модульности. Оно приводит к заключению «разделяй и властвуй» – сложную проблему легче решить, разделив ее на управляемые части. Результат, выраженный неравенством, имеет важное значение для модульности и ПО. Фактически, это аргумент в пользу модульности.

Однако здесь отражена лишь часть реальности, ведь здесь не учитываются затраты на межмодульный интерфейс. Как показано на рис. 7, с увеличением количества модулей (и уменьшением их размера) эти затраты также растут.

Таким образом, существует оптимальное количество модулей Opt, которое приводит к минимальной стоимости разработки. У нас нет необходимого опыта для гарантированного предсказания Opt. Впрочем, разработчики знают, что оптимальный модуль должен удовлетворять двум критериям:

- снаружи он проще, чем внутри;
- его проще использовать, чем построить.

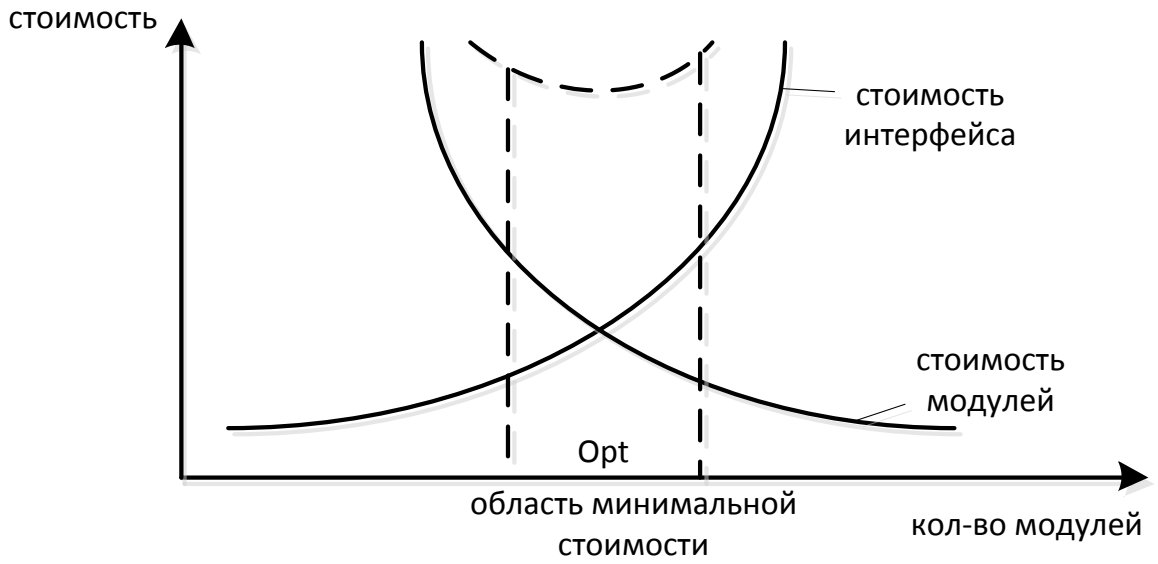


Рис. 7. Зависимость стоимости от количества модулей

**Информационная закрытость**

Принцип информационной закрытости утверждает: содержание модулей должно быть скрыто друг от друга. Как показано на рис. 8, модуль должен определяться и проектироваться так, чтобы его содержимое (процедуры и данные) было недоступно тем модулям, которые не нуждаются в такой информации (клиентам).

Информационная закрытость означает следующее:

1) все модули независимы, обмениваются только информацией, необходимой для работы;
2) доступ к операциям и структурам данных модуля ограничен.

Достоинства информационной закрытости:

3) обеспечивается возможность разработки модулей различными независимыми коллективами;
4) обеспечивается легкая модификация системы (вероятность распространения ошибок очень мала, так как большинство данных и процедур скрыто от других частей системы).

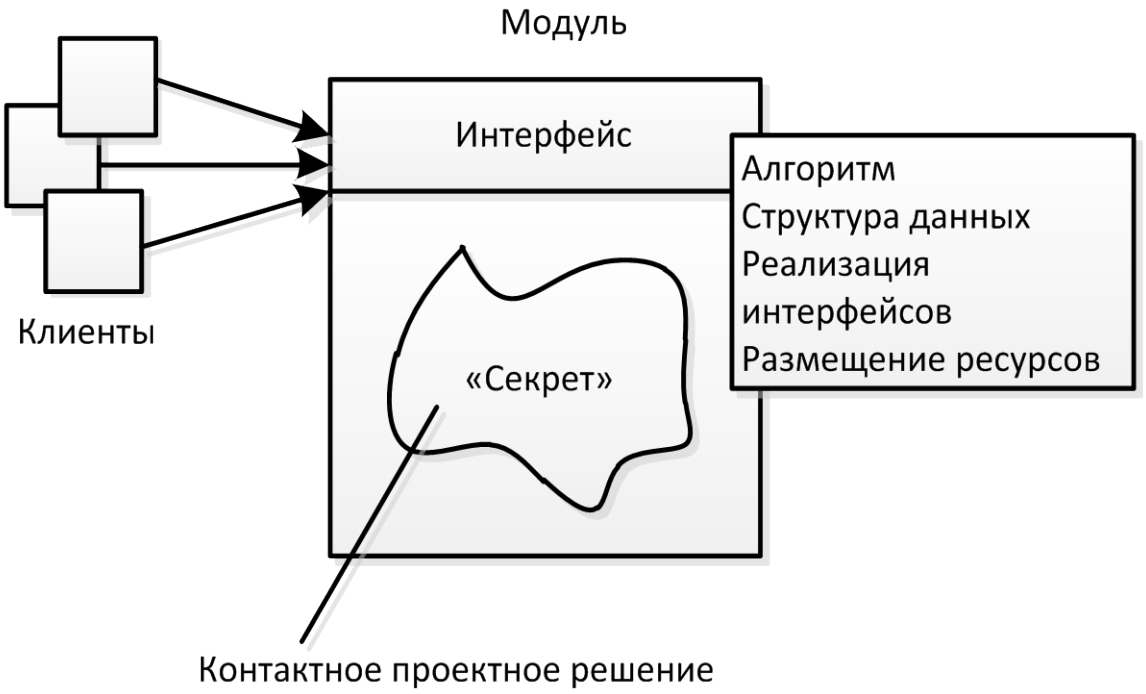


Рис. 8. Информационная закрытость модуля

Идеальный модуль играет роль «черного ящика», содержимое которого невидимо клиентам. Он прост в использовании – количество «ручек и органов управления» им невелико (аналогия с эксплуатацией телевизора). Его легко развивать и корректировать в процессе сопровождения программной системы. Для обеспечения таких возможностей система внутренних и внешних связей модуля должна отвечать особым требованиям.

### 3.1.1 Связность модуля

Связность модуля – это мера зависимости его частей. Связность – внутренняя характеристика модуля. Чем выше связность модуля, тем лучше результат проектирования, т. е. тем «черней» его ящик (капсула, защитная оболочка модуля), тем меньше «ручек управления» на нем находится и тем проще эти «ручки».

Для измерения связности используют понятие силы связности (СС). Существуют 7 типов связности:

1) Связность по совпадению (СС = 0). В модуле отсутствуют явно выраженные внутренние связи.
2) Логическая связность (СС = 1). Части модуля объединены по принципу функционального подобия. Например, модуль состоит из разных подпрограмм обработки ошибок. При использовании такого модуля клиент выбирает только одну из подпрограмм.

   Недостатки:

   - сложное сопряжение;

   - большая вероятность внесения ошибок при изменении сопряжения ради одной из функций.
3) Временная связность (СС = 3). Части модуля не связаны, но необходимы в один и тот же период работы системы.

   Недостаток: сильная взаимная связь с другими модулями, отсюда сильная чувствительность внесению изменений.
4) Процедурная связность (СС = 5). Части модуля связаны порядком выполняемых ими действий, реализующих некоторый сценарий по ведения.
5) Коммуникативная связность (СС = 7). Части модуля связаны по данным (работают с одной и той же структурой данных).
6) Информационная (последовательная) связность (СС = 9). Выходные данные одной части используются как входные данные в другой части модуля.

7) Функциональная связность (CC = 10). Части модуля вместе реализуют одну функцию.

Отметим, что типы связности 1, 2, 3 – результат неправильного планирования архитектуры, а тип связности 4 – результат небрежного планирования архитектуры приложения.

Общая характеристика типов связности представлена в табл. 2.

Таблица 2

Характеристика связности модуля

| Тип связности | Сопровождаемость | Роль модуля |
|---|---|---|
| Функциональная | Лучшая сопровождаемость | «Черный ящик» |
| Информационная (последовательная) | | Не совсем «черный ящик» |
| Коммуникативная | | «Серый ящик» |
| Процедурная | Худшая сопровождаемость | «Белый» или «просвечивающий ящик» |
| Временная | | «Белый ящик» |
| Логическая | | |
| По совпадению | | |

**Функциональная связность**

Функционально связный модуль содержит элементы, участвующие в выполнении одной и только одной проблемной задачи. Примеры функционально связных модулей:

- вычислять синус угла;
- проверять орфографию;
- читать запись файла;
- вычислять координаты цели;
- вычислять зарплату сотрудника;
- определять место пассажира.

Каждый из этих модулей имеет единичное назначение. Когда клиент вызывает модуль, выполняется только одна работа, без привлечения внешних обработчиков. Например, модуль «Определять

место пассажира» должен делать только это; он не должен распечатывать заголовки страницы.

Некоторые из функционально связных модулей очень просты (например, «Вычислять синус угла» или «Читать запись файла»), другие сложны (например, «Вычислять координаты цели»). Модуль «Вычислять синус угла», очевидно, реализует единичную функцию, но как может модуль «Вычислять зарплату сотрудника» выполнять только одно действие? Ведь каждый знает, что приходится определять начисленную сумму, вычеты по рассрочкам, подоходный налог, социальный налог, алименты и т. д. Дело в том, что, несмотря на сложность модуля и на то, что его обязанность исполняют несколько подфункций, если его действия можно представить как единую проблемную функцию (с точки зрения клиента), тогда считают, что модуль функционально связен.

Приложения, построенные из функционально связных модулей, легче всего сопровождать. Напрасно думать, что любой модуль можно рассматривать как однофункциональный. Существует много разновидностей модулей, которые выполняют для клиентов перечень различных работ, и этот перечень нельзя рассматривать как единую проблемную функцию. Критерий при определении уровня связности этих нефункциональных модулей – как связаны друг с другом различные действия, которые они исполняют.

**Информационная связность**

При информационной (последовательной) связности элементы – обработчики модуля образуют конвейер для обработки данных – результаты одного обработчика используются как исходные данные для следующего обработчика. Приведем *пример*:

модуль «Прием и проверка записи»

прочитать запись из файла;

проверить контрольные данные в записи;

удалить контрольные поля в записи;

вернуть обработанную запись;

конец модуля.

В этом модуле 3 элемента. Результаты первого элемента (прочитать запись из файла) используются как входные данные для второго элемента (проверить контрольные данные в записи) и т. д.

Сопровождать модули с информационной связностью почти так же легко, как и функционально связные модули. Правда, возможности повторного использования здесь ниже, чем в случае функциональной связности. Причина – совместное применение действий модуля с информационной связностью полезно далеко не всегда.

**Коммуникативная связность**

При коммуникативной связности элементы-обработчики модуля используют одни и те же данные, например, внешние данные. *Пример* коммуникативно связного модуля:

модуль «Отчет и средняя зарплата»

используется «Таблица зарплаты служащих»;

сгенерировать «Отчет по зарплате»;

вычислить параметр «Средняя зарплата»;

вернуть «Отчет по зарплате. Средняя зарплата»;

конец модуля.

Здесь все элементы модуля работают со структурой «Таблица зарплаты служащих».

С точки зрения клиента, проблема применения коммуникативно связного модуля состоит в избыточности получаемых результатов. Например, клиенту требуется только отчет по зарплате, он не нуждается в значении средней зарплаты. Такой клиент будет вынужден выполнять избыточную работу – выделение в полученных данных материала отчета. Почти всегда разбиение коммуникативно

связного модуля на отдельные функционально связные модули улучшает сопровождаемость системы.

Попытаемся провести аналогию между информационной и коммуникативной связностью.

Модули с коммуникативной и информационной связностью схожи в том, что содержат элементы, связанные по данным. Их удобно использовать, потому что лишь немногие элементы в этих модулях связаны с внешней средой. Главное различие между ними – информационно связный модуль работает подобно сборочной линии; его обработчики действуют в определенном порядке; в коммуникативно связном модуле порядок выполнения действий безразличен. В нашем примере не имеет значения, когда генерируется отчет (до, после или одновременно с вычислением средней зарплаты).

**Процедурная связность**

При достижении процедурной связности мы попадаем в пограничную область между хорошей сопровождаемостью (для модулей с более высокими уровнями связности) и плохой сопровождаемостью (для модулей с более низкими уровнями связности). Процедурно связный модуль состоит из элементов, реализующих независимые действия, для которых задан порядок работы, т. е. порядок передачи управления. Зависимости по данным между элементами нет. *Например*:

модуль «Вычисление средних значений»

используется Таблица-А. Таблица-В;

вычислить среднее по Таблица-А;

вычислить среднее по Таблица-В;

вернуть среднее Таблица-А. Таблица-В;

конец модуля.

Этот модуль вычисляет средние значения для двух, полностью несвязанных таблиц «Таблица-А» и «Таблица-В», каждая из которых имеет по 300 элементов.

Теперь представим себе программиста, которому поручили реализовать данный модуль. Соблазнившись возможностью минимизации кода (использовать один цикл в интересах двух обработчиков, ведь они находятся внутри единого модуля!), программист пишет:

```
модуль «Вычисление средних значений»
    используется Таблица-А. Таблица-В;
    сумма Таблица-А := 0;
    сумма Таблица-В := 0;
    для i := 1 до 300;
        сумма Таблица-А := сумма Таблица-А + Таблица-А(i);
        сумма Таблица-В : = сумма Таблица-В + Таблица-В(i);
    конец для
    среднее Таблица-А := сумма Таблица-А / 300;
    среднее Таблица-В := сумма Таблица-В / 300;
    вернуть среднее Таблица-А, среднее Таблица-В;
конец модуля.
```

Для процедурной связности этот случай типичен – независимый (на уровне проблемы) код стал зависимым (на уровне реализации). Прошли годы, продукт сдали заказчику. И вдруг возникла задача сопровождения – модифицировать модуль под уменьшение размера таблицы В. Оцените, насколько удобно ее решать.

**Временная связность**

При связности по времени элементы-обработчики модуля привязаны к конкретному периоду времени (из жизни программной системы).

Классическим примером временной связности является модуль инициализации:

модуль «Инициализировать систему»

перемотать магнитную ленту 1;

Счетчик магнитной ленты 1 := 0;

перемотать магнитную ленту 2;

Счетчик магнитной ленты 2 := 0;

Таблица текущих записей : = пробел..пробел;

Таблица количества записей := 0..0;

Переключатель 1 : = выкл;

Переключатель 2 := вкл;

конец модуля.

Элементы данного модуля почти не связаны друг с другом (за исключением того, что должны выполняться в определенное время). Они все – часть программы запуска системы. Зато элементы более тесно взаимодействуют с другими модулями, что приводит к сложным внешним связям.

Модуль со связностью по времени испытывает те же трудности, что и процедурно связный модуль. Программист соблазняется возможностью совместного использования кода (действиями, которые связаны только по времени), модуль становится трудно использовать повторно.

Так, при желании инициализировать магнитную ленту 2 в другое время, вы столкнетесь с неудобствами. Чтобы не сбрасывать всю систему, придется или ввести флажки, указывающие инициализируемую часть, или написать другой код для работы с лентой 2. Оба решения ухудшают сопровождаемость.

Процедурно связные модули и модули с временной связностью очень похожи. Степень их непрозрачности изменяется от темно-серого до светло-серого цвета, так как трудно объявить функцию

такого модуля без перечисления ее внутренних деталей. Различие между ними подобно различию между информационной и коммуникативной связностью. Порядок выполнения действий более важен в процедурно связных модулях. Кроме того, процедурные модули имеют тенденцию к совместному использованию циклов и ветвлений, а модули с временной связностью чаще содержат более линейный код.

**Логическая связность**

Элементы логически связного модуля принадлежат к действиям одной категории, и из этой категории клиент выбирает выполняемое действие. Рассмотрим следующий *пример*:

модуль «Пересылка сообщения»

переслать по электронной почте;

переслать по факсу;

послать в телеконференцию;

переслать по ftp-протоколу;

конец модуля.

Как видим, логически связный модуль – мешок доступных действий. Действия вынуждены совместно использовать один и тот же интерфейс модуля. В строке вызова модуля значение каждого параметра зависит от используемого действия. При вызове отдельных действий некоторые параметры должны иметь значение пробела, нулевые значения и т. д. (хотя клиент все же должен использовать их и знать их типы).

Действия в логически связном модуле попадают в одну категорию, хотя имеют не только сходства, но и различия. К сожалению, это заставляет программиста «завязывать код действий в узел», ориентируясь на то, что действия совместно используют общие строки кода. Поэтому логически связный модуль имеет:

- уродливый внешний вид с различными параметрами, обеспечивающими, например, четыре вида доступа;

- запутанную внутреннюю структуру со множеством переходов, похожую на волшебный лабиринт.

В итоге модуль становится сложным как для понимания, так и для сопровождения.

**Связность по совпадению**

Элементы связного по совпадению модуля вообще не имеют никаких отношений друг с другом:

модуль «Разные функции» (какие-то параметры)

поздравить с Новым годом (...);

проверить исправность аппаратуры (...);

заполнить анкету героя (...);

измерить температуру (...);

вывести собаку на прогулку (...);

запастись продуктами (...);

приобрести «ягуар» (...);

конец модуля.

Связный по совпадению модуль похож на логически связный модуль. Его элементы-действия не связаны ни потоком данных, ни потоком управления. Но в логически связном модуле действия, по крайней мере, относятся к одной категории; в связном по совпадению модуле даже это не так. Словом, связные по совпадению модули имеют все недостатки логически связных модулей и даже усиливают их.

Чтобы клиент мог воспользоваться модулем «Разные функции», этот модуль (подобно всем связным по совпадению модулям) должен быть «белым ящиком», чья реализация полностью видима. Такие модули делают системы менее понятными и труднее сопровождаемыми, чем системы без модульности вообще.

К счастью, связность по совпадению встречается редко. Среди ее причин можно назвать:

- бездумный перевод существующего монолитного кода в модули;

- необоснованные изменения модулей с плохой (обычно временной) связностью, приводящие к добавлению флажков.

### 3.1.2 Определение связности модуля

Приведем алгоритм определения уровня связности модуля.

1) Если модуль – единичная проблемно-ориентированная функция, то уровень связности – функциональный; конец алгоритма. В противном случае перейти к пункту 2.
2) Если действия внутри модуля связаны, то перейти к пункту 3. Если действия внутри модуля никак не связаны, то перейти к пункту 6.
3) Если действия внутри модуля связаны данными, то перейти к пункту 4. Если действия внутри модуля связаны потоком управления, перейти к пункту 5.
4) Если порядок действий внутри модуля важен, то уровень связности – информационный. В противном случае уровень связности – коммуникативный. Конец алгоритма.
5) Если порядок действий внутри модуля важен, то уровень связности – процедурный. В противном случае уровень связности – временной. Конец алгоритма.
6) Если действия внутри модуля принадлежат к одной категории, то уровень связности – логический. Если действия внутри модуля не принадлежат к одной категории, то уровень связности – по совпадению. Конец алгоритма.

Возможны более сложные случаи, когда с модулем ассоциируются несколько уровней связности. В этих случаях следует применять одно из двух правил:

- правило параллельной цепи. Если все действия модуля имеют несколько уровней связности, то модулю присваивают самый сильный уровень связности;

- правило последовательной цепи. Если действия в модуле имеют разные уровни связности, то модулю присваивают самый слабый уровень связности.

Например, модуль может содержать некоторые действия, которые связаны процедурно, а также другие действия, связные по совпадению. В этом случае применяют правило последовательной цепи, и в целом, модуль считают связным по совпадению.

### 3.1.3 Сцепление модулей

Сцепление (Coupling) – мера взаимозависимости модулей по данным. Сцепление – внешняя характеристика модуля, которую желательно уменьшать.

Количественно сцепление измеряется степенью сцепления (СЦ). Выделяют шесть типов сцепления.

1. Сцепление по данным (СЦ = 1). Модуль А вызывает модуль В. Все входные и выходные параметры вызываемого модуля – простые элементы данных (рис. 9).
2. Сцепление по образцу (СЦ = 3). В качестве параметров используются структуры данных (рис. 10).

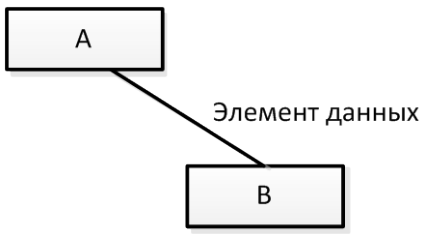


Рис. 9. Сцепление по данным

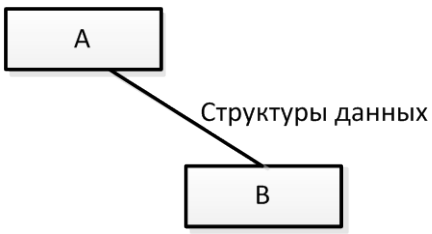


Рис. 10. Сцепление по образцу

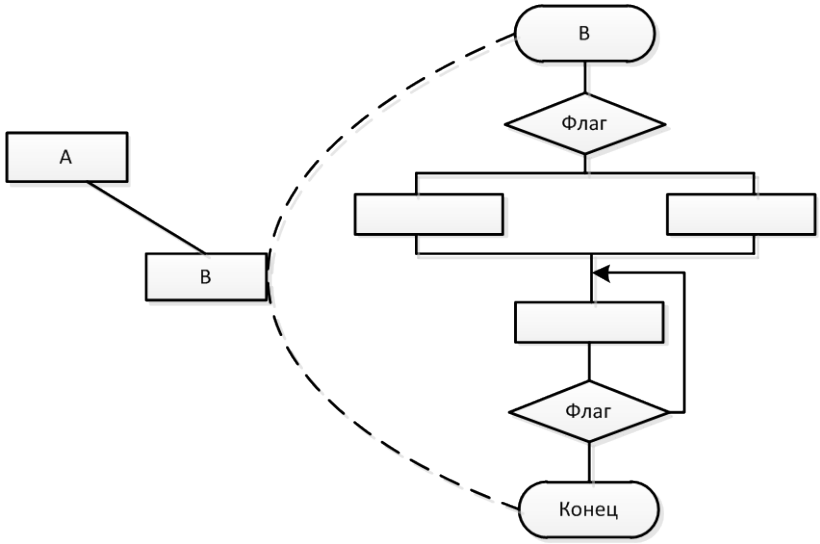


Рис. 11. Сцепление по образцу

3. Сцепление по управлению (СЦ = 4). Модуль А явно управляет функционированием модуля В (с помощью флагов или переключателей), посылая ему управляющие данные (рис. 11).
4. Сцепление по внешним ссылкам (СЦ = 5). Модули А и В ссылаются на один и тот же глобальный элемент данных.
5. Сцепление по общей области (СЦ = 7). Модули разделяют одну и ту же глобальную структуру данных (рис. 12).
6. Сцепление по содержанию (СЦ = 9). Один модуль прямо ссылается на содержание другого модуля (не через его точку входа). Например, коды их команд перемежаются друг с другом (рис. 12).

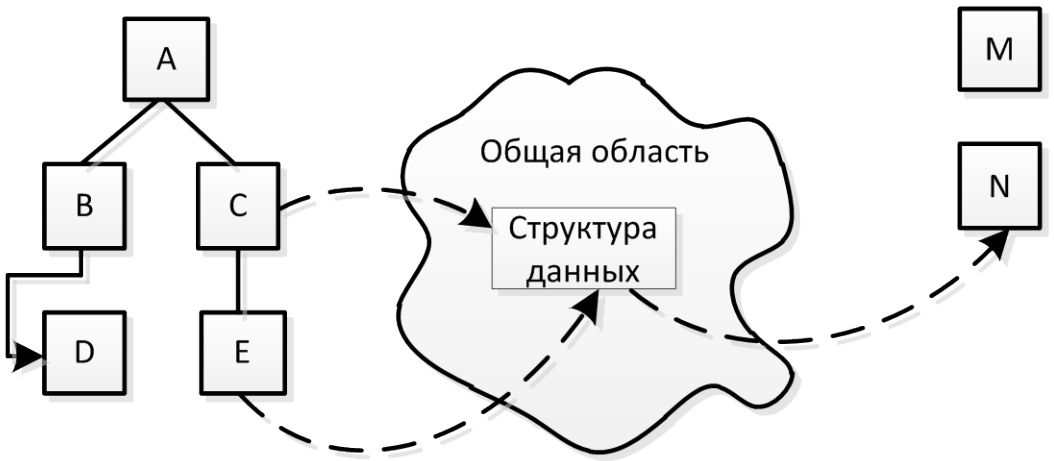


Рис. 12. Сцепление по данным

На рисунке 12 видим, что модули B и D сцеплены по содержанию, а модули C, E и N сцеплены по общей области.

### 3.1.4 Сложность программной системы

В простейшем случае сложность системы определяется как сумма мер сложности ее модулей. Сложность модуля может вычисляться различными способами.

Например, в [6] предложена мера длины $N$ модуля:

$$N \approx n_1 \log_2(n_1) + n_2 log_2(n_2),$$

где $n_1$ – число различных операторов; $n_2$ – число различных операндов.

В качестве второй метрики М. Холстед рассматривал объем V модуля (количество символов для записи всех операторов и операндов текста программы):

$$V = N \times log_2(n1 + n2),$$

Вместе с тем известно, что любая сложная система состоит из элементов и системы связей между элементами и что игнорировать внутрисистемные связи неразумно.

При оценке сложности ПС предложено исходить из топологии внутренних связей. Для этой цели он разработал метрику цикломатической сложности:

$$V(G) = E - N + 2,$$

где $E-$ количество дуг; $N$ – количество вершин в управляющем графе ПС.

Это был шаг в нужном направлении. Дальнейшее уточнение оценок сложности потребовало, чтобы каждый модуль мог представляться как локальная структура, состоящая из элементов и связей между ними.

Таким образом, при комплексной оценке сложности ПС необходимо рассматривать меру сложности модулей, меру сложности

внешних связей (между модулями) и меру сложности внутренних связей (внутри модулей). Традиционно с внешними связями сопоставляют характеристику «сцепление», а с внутренними связями – характеристику «связность».

## 3.2 Методологии

### 3.2.1 Методология, созданная компанией «Rational Software»

На сегодняшний день практически все ведущие компании – разработчики технологий и программных продуктов (IBM, Oracle, Borland) располагают развитыми технологиями создания ПО, которые создавались как собственными силами, так и за счет приобретения продуктов и технологий, созданных небольшими специализированными компаниями.

Одна из наиболее совершенных технологий, претендующих на роль мирового корпоративного стандарта – Rational Unified Process (RUP). RUP представляет собой программный продукт, разработанный компанией Rational Software (www.rational.com), которая в настоящее время входит в состав IBM.

RUP в значительной степени соответствует стандартам и нормативным документам, связанным с процессами ЖЦ ПО и оценкой технологической зрелости организаций-разработчиков (ISO 12207, ISO 9000, CMM и др.). Основным принципом RUP является итерационный и инкрементный (наращиваемый) подход к созданию ПО. В соответствии с ним разработка системы выполняется в виде нескольких краткосрочных мини-проектов фиксированной длительности (от 2 до 6 недель), называемых *итерациями*. Каждая итерация включает свои собственные этапы анализа требований, проектирования, реализации, тестирования, интеграции и завершается созданием работающей системы.

Итерационный цикл основывается на постоянном расширении и дополнении системы в процессе нескольких итераций с периодической обратной связью и адаптацией добавляемых модулей к существующему ядру системы. Система постоянно разрастается шаг за шагом, поэтому такой подход называют итерационным или инкрементным.

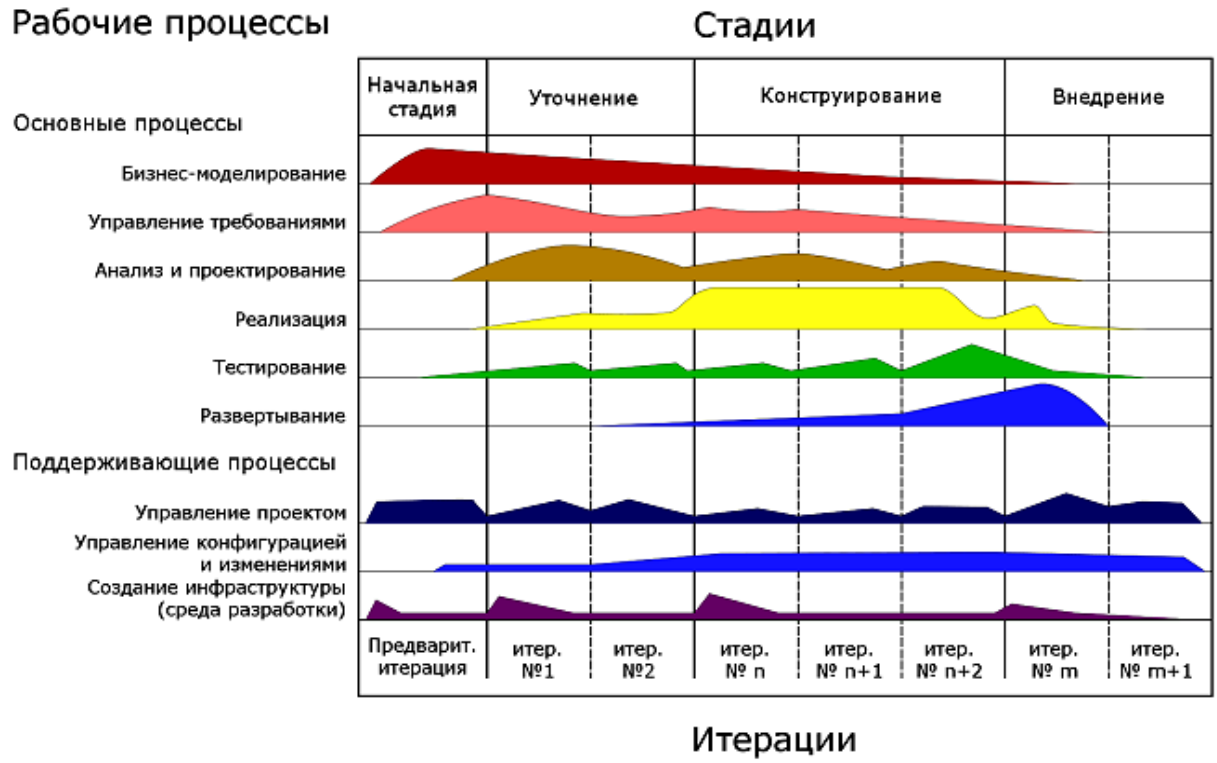


Рис. 13. Технология, разработанная компанией Rational Software (Rational Unified Process)

На рисунке 13 показано общее представление RUP в двух измерениях. Горизонтальное измерение представляет время, отражает динамические аспекты процессов и оперирует такими понятиями, как стадии, итерации и контрольные точки. Вертикальное измерение отражает статические аспекты процессов и оперирует такими понятиями, как виды деятельности (технологические операции), рабочие продукты, исполнители и технологические процессы.

Согласно RUP, ЖЦ ПО разбивается на отдельные циклы, в каждом из которых создается новое поколение продукта. Каждый цикл, в свою очередь, разбивается на четыре последовательные стадии:

- начальная стадия (inception);
- стадия разработки (elaboration);
- стадия конструирования (construction);
- стадия ввода в действие (transition).

Каждая стадия завершается в четко определенной контрольной точке (milestone). В этот момент времени должны достигаться важные результаты и приниматься критически важные решения о дальнейшей разработке.

Начальная стадия может принимать множество разных форм. Для крупных проектов начальная стадия может вылиться во всестороннее изучение всех возможностей реализации проекта, которое займет месяцы. Во время начальной стадии вырабатывается бизнес-план проекта – определяется, сколько приблизительно он будет стоить и какой доход принесет. Определяются также границы проекта, и выполняется некоторый начальный анализ для оценки размеров проекта.

- Результатами начальной стадии являются:
- общее описание системы: основные требования к проекту, его характеристики и ограничения;
- начальная модель вариантов использования (степень готовности – 10–20%);
- начальный проектный глоссарий (словарь терминов);
- начальный бизнес-план;
- план проекта, отражающий стадии и итерации;
- один или несколько прототипов.

На стадии разработки выявляются более детальные требования к системе, выполняется высокоуровневый анализ предметной области и проектирование для построения базовой архитектуры системы, создается план конструирования и устраняются наиболее рискованные элементы проекта.

Результатами стадии разработки являются:

- модель вариантов использования (завершенная, по крайней мере, на 80%), определяющая функциональные требования к системе;

перечень дополнительных требований, включая требования нефункционального характера и требования, не связанные с конкретными вариантами использования;

- описание базовой архитектуры будущей системы;

- работающий прототип;

- уточненный бизнес-план;

- план разработки всего проекта, отражающий итерации и критерии оценки для каждой итерации.

Самым важным результатом стадии разработки является описание базовой архитектуры будущей системы. Эта архитектура включает:

- модель предметной области, которая отражает понимание бизнеса и служит отправным пунктом для формирования основных классов предметной области;

- технологическую платформу, определяющую основные элементы технологии реализации системы и их взаимодействие.

Эта архитектура является основой всей дальнейшей разработки, она служит своего рода проектом для последующих стадий. В дальнейшем неизбежны незначительные изменения в деталях архитектуры, однако серьезные изменения маловероятны.

Стадия разработки занимает около пятой части общей продолжительности проекта. Основными признаками завершения стадии разработки являются два события:

- разработчики в состоянии оценить с достаточно высокой точностью, сколько времени потребуется на реализацию каждого варианта использования;

- идентифицированы все наиболее серьезные риски, и степень понимания наиболее важных из них такова, что известно, как справиться с ними.

Стадия конструирования заключается в определении последовательности итераций конструирования вариантов использования, реализуемых на каждой итерации. Итерации на стадии конструирования являются одновременно инкрементными и повторяющимися:

- итерации являются инкрементными в соответствии с той функцией, которую они выполняют. Каждая итерация добавляет очередные конструкции к вариантам использования, реализованным во время предыдущих итераций;

- итерации являются повторяющимися по отношению к разрабатываемому коду. На каждой итерации некоторая часть существующего кода переписывается с целью сделать его более гибким.

Результатом стадии конструирования является продукт, готовый к передаче конечным пользователям. Как минимум, он содержит следующее:

- ПО, интегрированное на требуемые платформы;

- руководства пользователя;

- описание текущей реализации.

Стадия ввода в действие предназначена для передачи готового продукта в распоряжение пользователей. Данная стадия включает:

- β-тестирование, позволяющее убедиться, что новая система соответствует ожиданиям пользователей;

- параллельное функционирование с существующей системой, которая подлежит постепенной замене;

- конвертирование баз данных;

- оптимизацию производительности;

обучение пользователей и специалистов службы сопровождения.

Статический аспект RUP представлен четырьмя основными элементами:

- роли;

- виды деятельности;

- рабочие продукты;

- дисциплины.

Понятие «роль» (role) определяет поведение и ответственность личности или группы личностей, составляющих проектную команду. Одна личность может играть в проекте много различных ролей.

Под видом деятельности конкретного исполнителя понимается единица выполняемой им работы. Вид деятельности (activity) соответствует понятию технологической операции. Он имеет четко определенную цель, обычно выражаемую в терминах получения или модификации некоторых рабочих продуктов (artifacts), таких, как модель, элемент модели, документ, исходный код или план. Каждый вид деятельности связан с конкретной ролью. Продолжительность вида деятельности составляет от нескольких часов до нескольких дней, он обычно выполняется одним исполнителем и порождает только один или весьма небольшое количество рабочих продуктов. Примерами видов деятельности могут быть планирование итерации, определение вариантов использования и действующих лиц, выполнение теста на производительность. Каждый вид деятельности сопровождается набором руководств (guidelines), представляющих собой методики выполнения технологических операций.

Дисциплина (discipline) соответствует понятию технологического процесса и представляет собой последовательность действий, приводящую к получению значимого результата.

В рамках RUP определены шесть основных дисциплин:

- построение бизнес-моделей;
- определение требований;
- анализ и проектирование;
- реализация;
- тестирование;
- развертывание;

и три вспомогательных:

- управление конфигурацией и изменениями;
- управление проектом;
- создание инфраструктуры.

RUP как продукт входит в состав комплекса Rational Suite, причем каждая из перечисленных выше дисциплин поддерживается определенным инструментальным средством комплекса. Физическая реализация RUP представляет собой Web-сайт, включающий следующие компоненты:

- описание всех элементов динамического и статического аспекта RUP;
- навигатор по всем элементам RUP, глоссарий и средство быстрого обучения технологии;
- руководства для всех участников проектной команды, охватывающие весь жизненный цикл ПО. Руководства представлены в двух видах: для осмысления процесса на верхнем уровне и в виде подробных рекомендаций по повседневной деятельности;
- рекомендации по использованию инструментальных средств, входящих в состав Rational Suite;
- примеры и шаблоны проектных решений для Rational Rose;

- шаблоны проектной документации для SoDa;

- шаблоны в формате Microsoft Word, предназначенные для поддержки документации по всем процессам и действиям жизненного цикла ПО;

- планы в формате Microsoft Project, отражающие итерационный характер разработки ПО.

RUP опирается на интегрированный комплекс инструментальных средств Rational Suite. Он существует в следующих вариантах:

- Rational Suite AnalystStudio - предназначен для определения и управления полным набором требований к разрабатываемой системе;

- Rational Suite DevelopmentStudio – предназначен для проектирования и реализации ПО;

- Rational Suite TestStudio – представляет собой набор продуктов, предназначенных для автоматического тестирования приложений;

- Rational Suite Enterprise – обеспечивает поддержку полного жизненного цикла ПО и предназначен как для менеджеров проекта, так и отдельных разработчиков, выполняющих несколько функциональных ролей в команде разработчиков.

В состав Rational Suite, кроме самой технологии RUP как продукта, входят следующие компоненты:

- Rational Rose – средство визуального моделирования (анализа и проектирования), использующее язык UML;

- Rational XDE – средство анализа и проектирования, интегрируемое с платформами MS Visual Studio .NET и IBM WebSphere Studio Application Developer;

- Rational Requisite Pro – средство управления требованиями, предназначенное для организации совместной работы группы разработчиков. Оно позволяет команде разработчиков создавать,

структурировать, устанавливать приоритеты, отслеживать, контролировать изменения требований, возникающих на любом этапе разработки компонентов приложения;

- Rational Rapid Developer – средство быстрой разработки приложений на платформе Java 2 Enterprise Edition;

- Rational ClearCase – средство управления конфигурацией ПО;

- Rational SoDA – средство автоматической генерации проектной документации;

- Rational ClearQuest – средство для управления изменениями и отслеживания дефектов в проекте на основе средств e-mail и Web;

- Rational Quantify – средство количественного определения узких мест, влияющих на общую эффективность работы программы;

- Rational Purify – средство для локализации трудно обнаруживаемых ошибок времени выполнения программы;

- Rational PureCoverage – средство идентификации участков кода, пропущенных при тестировании;

- Rational TestManager – средство планирования функционального и нагрузочного тестирования;

- Rational Robot – средство записи и воспроизведения тестовых сценариев;

- Rational TestFactory – средство тестирования надежности;

- Rational Quality Architect – средство генерации кода для тестирования.

Одно из основных инструментальных средств комплекса Rational Rose представляет собой семейство объектно-ориентированных CASE-средств и предназначено для автоматизации процессов анализа и проектирования ПО, а также для генерации кодов на различных языках и выпуска проектной документации. Rational Rose реализует процесс объектно-ориентированного анализа и проектирования ПО, описанный в RUP. В основе работы Rational

Rose лежит построение диаграмм и спецификаций UML, определяющих архитектуру системы, ее статические и динамические аспекты. В составе Rational Rose можно выделить шесть основных структурных компонентов: репозиторий, графический интерфейс пользователя, средства просмотра проекта (браузер), средства контроля проекта, средства сбора статистики и генератор документов. К ним добавляются генераторы кодов для каждого поддерживаемого языка, состав которых меняется от версии к версии.

Репозиторий представляет собой базу данных проекта. Браузер обеспечивает «навигацию» по проекту, в том числе перемещение по иерархиям классов и подсистем, переключение от одного вида диаграмм к другому и т. д. Средства контроля и сбора статистики дают возможность находить и устранять ошибки по мере развития проекта, а не после завершения его описания. Генератор отчетов формирует тексты выходных документов на основе содержащейся в репозитории информации.

Средства автоматической генерации кода, используя информацию, содержащуюся в диаграммах классов и компонентов, формируют файлы описаний классов. Создаваемый таким образом скелет программы может быть уточнен путем прямого программирования на соответствующем языке (основные языки, поддерживаемые Rational Rose - C++ и Java).

В результате разработки проекта с помощью Rational Rose формируются следующие документы:

- диаграммы UML, в совокупности представляющие собой модель разрабатываемой программной системы;

- спецификации классов, объектов, атрибутов и операций;

- заготовки текстов программ.

Тексты программ являются заготовками для последующей работы программистов. Состав информации, включаемой в

программные файлы, определяется либо по умолчанию, либо по усмотрению пользователя. В дальнейшем эти исходные тексты развиваются программистами в полноценные программы.

### 3.2.2 Экстремальное программирование

Экстремальное программирование (Extreme Programming), часто обозначаемое аббревиатурой XP, – это дисциплина разработки программного обеспечения и ведения бизнеса в области создания программных продуктов, которая фокусирует усилия обеих сторон (программистов и бизнесменов) на общих, вполне достижимых целях. Команды, использующие XP, производят качественное программное обеспечение с весьма большой скоростью.

Четыре ценности для XP – это:

- коммуникация (communication);
- простота (simplicity);
- обратная связь (feedback);
- храбрость (courage).

Для начала перечислим все методики.

- Игра в планирование (planning game) – быстро определяет перечень задач (объем работ), которые необходимо реализовать в следующей версии продукта. Для этого рассматриваются бизнес-приоритеты и технические оценки. Если со временем план перестает соответствовать действительности, происходит обновление плана.

- Небольшие версии (small releases) – самая первая упрощенная версия системы быстро вводится в эксплуатацию, после этого через относительно короткие промежутки времени происходит выпуск версии за версией.

- Метафора (metaphor) – эта простая общедоступная и общеизвестная история, которая коротко описывает, как работает вся система. Эта история управляет всем процессом разработки.

- Простой дизайн (simple design) – в каждый момент времени система должна быть спроектирована так просто, как это возможно. Чрезмерная сложность устраняется, как только ее обнаруживают.

- Тестирование (testing) – программисты постоянно пишут тесты для модулей. Для того чтобы разработка продолжалась, все тесты должны срабатывать. Заказчики пишут тесты, которые демонстрируют работоспособность и завершенность той или иной возможности системы.

- Переработка (refactoring) – программисты реструктурируют систему, не изменяя при этом ее поведения. При этом они устраняют дублирование кода, улучшают коммуникацию, упрощают код и повышают его гибкость.

- Программирование парами (pair programming) – весь разрабатываемый код пишется двумя программистами на одном компьютере.

- Коллективное владение (collective ownership) – в любой момент времени любой член команды может изменить любой код в любом месте системы.

- Непрерывная интеграция (continuous integration) – система интегрируется и собирается множество раз в день. Это происходит каждый раз, когда завершается решение очередной задачи.

- 40-часовая неделя (40-hour week) – программисты работают не более 40 часов в неделю. Это правило. Никогда нельзя работать сверхурочно две недели подряд.

- Заказчик на месте разработки (on-site customer) – в состав команды входит реальный живой пользователь системы. Он доступен в течение всего рабочего дня и способен отвечать на вопросы о системе.

- Стандарты кодирования (coding standards) – программисты пишут весь код в соответствии с правилами, которые обеспечивают коммуникацию при помощи кода.

В данной главе кратко рассказывается о том, что подразумевается под каждой из этих методик. В следующей главе (Как это работает?) мы рассмотрим взаимосвязи между методиками, благодаря которым слабость одной методики нейтрализуется силой другой методики.

**Игра в планирование**

Ни соображения бизнеса, ни технические соотношения не должны превалировать. Разработка программного обеспечения – это всегда эволюционирующий диалог между желаемым и возможным. Природа этого диалога состоит в том, что в его процессе изменяется как то, что кажется возможным, так и то, что кажется желаемым.

Представители бизнеса должны принимать решения в следующих областях.

- Объем работ – какую часть проблемы достаточно решить для того, чтобы систему можно было эксплуатировать в реальных производственных условиях? Представитель бизнеса должен быть способен определить, какого объема будет недостаточно и какой объем будет чрезмерным.

- Приоритет – если с самого начала вы можете реализовать только возможности A или B, то какую из них следует реализовать в первую очередь? Ответ на этот вопрос должен быть определен в первую очередь представителем бизнеса, а не программистом.

- Композиция версий – как много или как мало должно быть сделано для того, чтобы бизнес лучше шел с программным обеспечением, чем без него? Программистская интуиция зачастую ошибается в отношении данного вопроса.

- Сроки выпуска версий – в какие важные даты очередные версии программного продукта должны появляться в производстве?

Бизнес не может принимать решения в вакууме. Разработчики должны сформировать набор технических решений, которые должны стать исходным материалом при формировании бизнес-решений.

Разработчики должны принимать решения в следующих областях.

- Оценка – как много времени потребуется для того, чтобы реализовать ту или иную возможность?

- Последствия – существует набор бизнес-решений, которые следует формировать, только ознакомившись с технологическими последствиями. Хорошим примером является выбор той или иной технологии управления базой данных. Бизнесу лучше иметь дело с крупной компанией, а не с новичками, однако и здесь существует набор факторов, которые необходимо тщательно изучить, так как, возможно, сотрудничество с компанией, появившейся на рынке недавно, оправданно по тем или иным причинам. Возможно, свежая, недавно разработанная система управления базой данных дает двукратный рост производительности. Возможно, разработка решения на основе этой базы данных обойдется бизнесу в два раза дешевле. Таким образом, риск, связанный с использованием новой технологии управления базой данных, вполне оправдан. А возможно, и нет. Разработчики должны объяснить последствия того или иного решения.

- Процесс – как будет организована команда разработчиков и как будет организована работа этой команды? Команда должна соответствовать культуре, в рамках которой будет осуществляться разработка, однако при этом не следует забывать, что ваша основная задача – получить качественный программный продукт, а не следить за чистотой культуры разработки.

- Подробный график работ – какие требования заказчика должны быть реализованы в первую очередь в рамках работы над

очередной версией продукта? Программисты должны обладать свободой при решении вопроса о том, какие самые рискованные сегменты разработки должны быть реализованы в первую очередь. Благодаря этому снижается общий риск разработки. В рамках этого ограничения по-прежнему следует в первую очередь работать над наиболее приоритетными для бизнеса задачами. Благодаря этому снижается вероятность того, что реализация чрезвычайно важных для бизнеса возможностей системы будет отложена до следующей версии.

**Небольшие версии**

Каждая версия должна быть настолько маленькой, насколько это возможно. В ней должны быть реализованы наиболее ценные для бизнеса требования. В целом версия должна быть логически завершенной и работоспособной, то есть вы не можете реализовать некоторую возможность только наполовину и внедрить ее в производство только для того, чтобы сократить время работы над версией.

Лучше планировать на месяц или два вперед, чем планировать на полгода или год вперед. Компания, которая за один раз передает в руки заказчика достаточно крупный программный продукт, не может выпускать очередные версии этого продукта достаточно часто. Эта компания должна сократить время работы над очередной версией настолько, насколько это возможно.

**Метафора**

Каждый программный проект XP направляется при помощи единой всеобъемлющей метафоры. Иногда эта метафора выглядит наивной, как, например, система управления контрактами, о которой рассказывается в терминах контрактов, заказчиков и индоссаментов. Иногда метафора требует дополнительных разъяснений, например, требуется отметить, что компьютер должен рассматриваться как

рабочий стол, или вычисление пенсии должно выглядеть как электронная таблица. Все это метафоры, так как на самом деле мы не говорим буквально, что система – это электронная таблица. Метафора просто помогает каждому участнику проекта понять базовые элементы программы и то, как они взаимосвязаны.

Слова, которые используются для идентификации технических элементов системы, должны постоянно заимствоваться из выбранной метафоры. По мере того как идет работа над проектом, метафора развивается, и вся команда продолжает ее анализировать, получая при этом новые источники вдохновения.

В XP метафора во многом заменяет собой то, что другие люди называют термином «архитектура». Проблема состоит в том, что архитектура – это, как правило, огромная по размерам схема системы, которая не дает представления о ее целостности. Архитектура – это большие прямоугольники и соединения между ними.

Конечно же, вы можете сказать: плохо сформированная архитектура – это плохо. Однако нам требуется подчеркнуть саму цель, для которой формируется архитектура, а это значит, что мы должны предоставить каждому участнику проекта связную историю о строении и функционировании системы. Эта история должна быть изложена на языке, понятном как технарям, так и бизнесменам. В рамках этой истории будет осуществляться работа над проектом. Спросив о метафоре, мы получаем в ответ сведения об архитектуре, причем эти сведения передаются нам в такой форме, которая удобна для общения и обдумывания.

**Простой дизайн**

Правильным является дизайн системы, в рамках которого постоянно выполняются следующие условия:

1. Выполняются все тесты.

2. Нет дублирующейся логики. (Опасайтесь скрытого дублирования, например, применения параллельных иерархий классов).

3. Выражается каждая из идей, важных для программистов.

4. Существует наименьшее возможное количество классов и методов.

Этот совет противоположен тому, что обычно приходится слышать программистам: реализуйте для сегодняшнего дня, а проектируйте – для завтрашнего. Однако если вы уверены, что будущее – в неопределенности, если вы верите в то, что завтра вы можете сменить направление своих мыслей и не платить за это слишком дорого, значит, включение в дизайн функциональности только на основании абстрактных размышлений – это безумство. Добавляйте в дизайн то, что вам нужно, только тогда, когда это действительно вам нужно.

**Тестирование**

Любая возможность программы, для которой нет автоматических тестов, просто не существует. Программисты пишут тесты модулей, благодаря чему их уверенность в правильности функционирования программы становится частью самой программы. Заказчики пишут функциональные тесты, благодаря чему их уверенность в функционировании программы также становится частью программы. В результате всеобщая уверенность в работоспособности программы со временем все возрастает. Эта уверенность выражена в наборе тестов, количество которых увеличивается и которые продолжают функционировать по мере продолжения работы над программой. Благодаря этому со временем программа становится не менее, а более приспособленной для внесения в нее изменений.

Нет необходимости писать тесты для каждого разрабатываемого вами метода, проверять надо только производственные методы, которые могут не сработать. Иногда вы тратите усилия только на то, чтобы понять, возможно ли в процессе функционирования кода

возникновение той или иной ситуации. В течение получаса вы анализируете код. Теперь вы отбрасываете код и начинаете писать его заново, начиная с тестов.

**Переработка**

Когда программисты приступают к реализации некоторой возможности программы, они всегда задаются вопросом: существует ли способ изменения имеющейся программы для того, чтобы упростить добавление в нее требуемой новой возможности? После того как возможность добавлена, программисты спрашивают себя: можно ли теперь упростить программу и при этом обеспечить выполнение всех тестов? Это и называется переработкой кода (refactoring).

Обратите внимание, это означает, что иногда вы должны сделать больше работы, чем это на самом деле необходимо для того, чтобы добавить в программу новую возможность. Однако, работая в этом ключе, вы обеспечиваете снижение затрат, связанных с добавлением в программу последующих возможностей. Переработав код, вы упрощаете его дальнейшую модернизацию. Вы не выполняете переработку исходя из предварительных нечетких размышлений, напротив, вы перерабатываете код тогда, когда система нуждается в этом. Когда при добавлении новой возможности вы дублируете код, это означает, что система просит вас выполнить переработку.

Если программист видит некоторый черновой способ обеспечить выполнение теста, для реализации которого потребуется одна минута, и, помимо этого, он также видит другой, более качественный способ обеспечить выполнение теста, предусматривающий также упрощение дизайна, но для реализации которого требуется десять минут, программист должен предпочесть второй способ, то есть потратить больше времени, но в результате получить более простой и более удобный дизайн. В рамках XP вы получаете

возможность вносить в дизайн системы любые, даже самые радикальные изменения, делая это небольшими, малорискованными шагами.

**Программирование парами**

Весь код системы, вплоть до ее внедрения в производство, пишется парами программистов, каждая из которых работает на одном компьютере, оснащенном одной клавиатурой и одной мышью.

В каждой паре существуют две роли. Один партнер, в руках которого находится клавиатура и мышь, думает о том, как прямо сейчас реализовать некоторый метод самым лучшим образом. Второй партнер думает более стратегически:

- Сработает ли используемый подход в целом?

- Какими могут быть другие, еще не рассмотренные тестовые случаи?

- Существуют ли какие-либо способы упростить всю систему таким образом, что текущая проблема просто исчезнет?

Состав пар меняется динамически. Партнеры, которые утром входили в состав одной пары, днем могут стать членами совершенно других пар. Если вы отвечаете за решение задачи в области, которая является для вас малознакомой, вы можете попросить составить вам компанию кого-либо, кто недавно работал в этой области. Скорее всего, вы побываете в паре с каждым из членов вашей команды.

**Коллективное владение**

Любой член команды, который видит возможность добавить что-либо в любой раздел кода системы, может сделать это в любой подходящий для этого момент времени.

Сравните это с двумя другими моделями владения кода – полное отсутствие владения и индивидуальное владение. В давние времена различными кусками кода программы никто не владел. Если

кто-либо желал изменить какой-либо код, он мог сделать это в соответствии со своими собственными пожеланиями. Результатом был хаос, в особенности если приходилось иметь дело с объектами, в которых взаимосвязь между строкой кода в одном месте и строкой кода в другом месте нельзя было в точности установить статически. Код разрастался очень быстро, и с такой же скоростью он стремительно терял стабильность.

Чтобы подвести ситуацию под контроль, программисты стали использовать индивидуальное владение кодом. Единственным человеком, кто обладал правом внесения в некоторый фрагмент кода изменений, являлся официальный владелец этого кода. Если кто-либо, не являющийся владельцем, видел, что код необходимо изменить, он должен был обратиться с соответствующей просьбой к владельцу. В результате такой практики действительный код системы начинал расходиться с тем, каким его хотели бы видеть работающие в рамках проекта программисты. Изменение кода в рамках подобного подхода превращалось в своего рода бюрократическую процедуру – люди начинали избегать обращаться к владельцу кода для того, чтобы внести в код желаемые изменения, вместо этого они предпочитали работать с тем, что есть. В конце концов внести изменение требуется прямо сейчас, а не спустя некоторое время. Таким образом, код оставался относительно стабильным, однако он не эволюционировал с достаточно большой скоростью. А когда владелец кода находил другую работу и уходил из команды... возникали серьезные проблемы.

В рамках XP ответственность за весь код системы лежит на всех членах команды. Нельзя сказать, что каждый член команды хорошо знает каждую часть кода, однако можно сказать, что каждый член команды знает, по крайней мере, что-то о каждой части. Если пара программистов работает над решением некоторой

задачи и видит, что для упрощения работы требуется внести модификации в некоторую часть кода, тем самым улучшив этот код, изменения вносятся немедленно, благодаря чему решаемая этой парой задача упрощается.

**Постоянно продолжающаяся интеграция**

Код интегрируется и тестируется каждые несколько часов, минимум один раз в день. Для того чтобы обеспечить это, проще всего выделить для этой цели один специально предназначенный для этого компьютер. Когда этот компьютер освобождается, пара, у которой имеется код, подлежащий интеграции, садится за интеграционный компьютер, загружает текущую версию системы, добавляет в нее свои собственные изменения (проверяя и устраняя любые несоответствия и конфликты) и запускает тесты до тех пор, пока все они не сработают (все 100% тестов).

Интеграция одного набора изменений за один раз отлично срабатывает, так как становится очевидным, кто именно должен исправить тест, который не сработал, – мы должны, так как, должно быть, именно мы его сломали. Это связано с тем, что предыдущая пара, которая выполняла интеграцию, добилась срабатывания всех 100% тестов. И если мы не добьемся срабатывания всех 100% тестов, мы должны выкинуть из системы все, что мы написали, и начать решать задачу заново, так как очевидно, что в этом случае, приступая к решению, мы просто не знали всего того, что требуется для разработки требуемого кода (возможно, мы не знаем всего необходимого и сейчас).

**40-часовая рабочая неделя**

Конечно же, для этого необязательно, чтобы рабочих часов в неделе было бы ровно 40. Разные люди способны эффективно работать в течение различного времени. Один человек не может концентрировать свое внимание дольше, чем в течение 35 рабочих

часов, другой способен успешно действовать в течение 45 часов в неделю. Но никто не способен работать по 60 часов в неделю на протяжении многих недель и при этом оставаться свежим, творческим, внимательным и уверенным в своих силах.

Работа во внеурочное время – это признак серьезных проблем в проекте. В рамках XP действует очень простое правило – нельзя работать во внеурочное время две недели подряд. В течение одной недели можно поработать несколько лишних часов. Но если в очередной понедельник вы приходите на работу и объявляете: чтобы достичь поставленных целей, мы должны снова работать допоздна, это означает, что у вас возникли проблемы, которые вы не сможете решить простым увеличением рабочего времени.

**Заказчик на месте разработки**

Для ответов на возникающие вопросы, решения споров и установки мелкомасштабных приоритетов рядом с командой разработчиков должен постоянно находиться реальный заказчик. Под термином «реальный заказчик» подразумевают человека, который действительно пользуется системой, когда эта система работает в производственных условиях. Например, если вы разрабатываете систему обслуживания клиентов, заказчиком будет служащий по работе с клиентами, пользующийся этой системой. Если вы разрабатываете систему обмена долговыми обязательствами, заказчиком будет биржевой брокер, работающий с этой системой.

Основным препятствием для воплощения этого правила является то обстоятельство, что реальные пользователи разрабатываемой системы обходятся для заказчика слишком дорого в рамках рабочего времени. Иногда менеджерам заказчика жалко жертвовать одним из реальных служащих компании только для того, чтобы он постоянно находился вместе с разработчиками и

отвечал на их вопросы. Менеджеры должны решить, что является для них более важным – ускорение и повышение качества разработки или рабочее время одного или двух сотрудников. Если программная система не приносит бизнесу больше, чем приносит ему один из сотрудников предприятия, скорее всего, такая система не стоит того, чтобы ее разрабатывать и внедрять на производстве.

Кроме того, утверждение, что представитель заказчика, работающий вместе с командой, не может заниматься некоторыми своими повседневными делами, на самом деле не верно. Даже такие творческие люди, как программисты, не могут генерировать вопросы непрерывно в течение 40 часов каждую неделю. Конечно, сотрудник предприятия-заказчика обладает тем недостатком, что он физически удален от тех людей, с которыми он должен взаимодействовать, однако при этом у него будет достаточно времени для выполнения некоторой своей обычной работы.

Недостатком такого подхода является то, что в случае, если проект в конце концов закрывают, то сотни часов, которые сотрудник предприятия-заказчика потратил на помощь и консультации разработчиков, оказываются временем, потраченным впустую. Получается, что заказчик потерял работу, которую его сотрудник делал вместе с разработчиками, а также он потерял работу, которую его сотрудник мог бы сделать в соответствии со своими прямыми обязанностями, если бы его рабочим временем не пожертвовали ради поддержки разработки проекта. XP делает все возможное для того, чтобы проект не закрыли.

**Стандарты кодирования**

Если мы собираемся перебрасывать программистов с разработки одной части системы на разработку другой части системы, обеспечивать постоянную смену партнеров в парах по несколько раз на дню, обеспечивать постоянную переработку кода программистами,

которые этот код не писали, мы просто не можем использовать в рамках одного проекта применение нескольких разных стилей кодирования. Спустя некоторое время работы над системой уже нельзя будет сказать точно, какой именно член команды написал тот или иной код.

Стандарт должен требовать от разработчиков приложения минимально возможных усилий для реализации той или иной возможности. Он должен способствовать воплощению на практике правила *трех О – Once and Only Once* (запрет на дублирование кода). Стандарт должен способствовать коммуникации. Наконец, стандарт должен быть добровольно воспринят всей командой разработчиков.

### 3.2.3 Скрам и Канбан

Попробуем описать Scrum и Kanban так, чтобы вложиться в сотню слов.

Scrum:

- Разделите вашу организацию на маленькие, кросс-функциональные, самоорганизующиеся команды.

- Разделите вашу работу на маленькие, конкретные компоненты. Отсортируйте этот список по приоритетам и оцените относительный объем работы по каждому элементу.

- Разделите время на короткие итерации фиксированной длины (обычно 1–4 недели) так, чтобы после каждой итерации проводилась демонстрация потенциально готового к использованию кода.

- Оптимизируйте план релиза и корректируйте приоритеты совместно с клиентом, основываясь на данных, получаемых при рассмотрении релиза после каждой итерации.

- Оптимизируйте процесс с помощью проведения ретроспективы после каждой итерации.

Итак, вместо большой команды, которая долго работает над чем-то большим, у нас получается небольшая команда, которая короткими итерациями работает над небольшими кусочками. Но с регулярной интеграцией, чтоб видеть целостность картины.

Kanban:

- Визуализируйте поток работ − разбейте работу на части, выпишите каждый из пунктов на карточку и прикрепите на стену, подпишите столбцы, чтобы видеть, на какой стадии находится каждое задание.

**- Ограничьте НЗР (WIP) –** (от англ. work-in-progress – незавершенная работа) – определите возможное количество незавершенных пунктов на каждой стадии рабочего процесса.

**- Измеряйте время выполнения задачи (lead time)** (среднюю продолжительность времени для завершения одного пункта, иногда называемую «оперативным временем» (cycle time)), оптимизируйте процесс, чтобы свести время выполнения задачи к минимуму и сделать его настолько прогнозируемым, насколько это возможно.

Так как же Scrum и Kanban связаны друг с другом?

Scrum и Kanban – инструменты процесса.

*Инструмент* – это то, что используется в качестве средства для выполнения задачи или достижения цели. *Процесс* – это то, как вы работаете.

Scrum и Kanban – это *инструменты процесса*, они помогают вам работать более эффективно, до определенной степени подсказывая, что делать. Java – тоже инструмент, он упрощает программирование компьютера.

Не бывает универсальных инструментов, не бывает и идеальных. Как и любые инструменты, Scrum и Kanban не совершенны и не универсальны. Они дают определенные правила, устанавливают ограничения и предписания. Например, Scrum предписывает

фиксированные по времени итерации и кросс-функциональные команды, а Kanban – использование визуализации процесса на доске задач и ограничение НЗР.

Достаточно любопытно, что ценность инструмента в том, что он *ограничивает ваш выбор*. Инструмент процесса, который позволяет вам делать что угодно, не очень полезен. Мы могли бы назвать такой процесс «Делайте что хочешь», а как насчет «Делайте все правильно»? Процесс «Делайте все правильно» гарантированно будет работать.

Правильные инструменты помогут вам достичь успеха, но не обязательно его обеспечат. Легко спутать успех или провал *проекта* с успешным или неудачным *использованием инструмента.*

- проект может быть успешным из-за очень хорошего инструмента;
- проект может быть успешным, несмотря на плохой инструмент;
- проект может провалиться из-за плохого инструмента;
- проект может провалиться, несмотря на очень хороший инструмент.

Scrum более директивный, чем Kanban.

Можно сравнить инструменты по количеству предписываемых правил, которое в них содержится. *Директивный* означает «содержащий больше правил, которым надо следовать», а *адаптивный* – «содержащий меньше правил». При 100%-м директивном подходе вам будет легче работать, ведь есть правила для всего. А при 100%-й адаптивности – «Делай что хочешь», не существует никаких правил и ограничений. Как видите, обе крайности выглядят весьма нелепо.

Agile-методы иногда называют *легковесными*, в частности, потому что они менее директивны, чем традиционные методы. В самом деле, первый принцип Agile-Манифеста гласит «Люди и взаимодействие важнее процессов и инструментов».

Scrum и Kanban являются высокоадаптивными, но, *если их сравнить*, то Scrum более директивен, нежели Kanban. Scrum предоставляет вам больше ограничений, и, таким образом, оставляет

меньше вариантов для выбора. Например, Scrum предписывает использование фиксированных по времени итераций, Kanban – нет.

Давайте сравним несколько инструментов процесса на шкале «предписание/адаптация»:

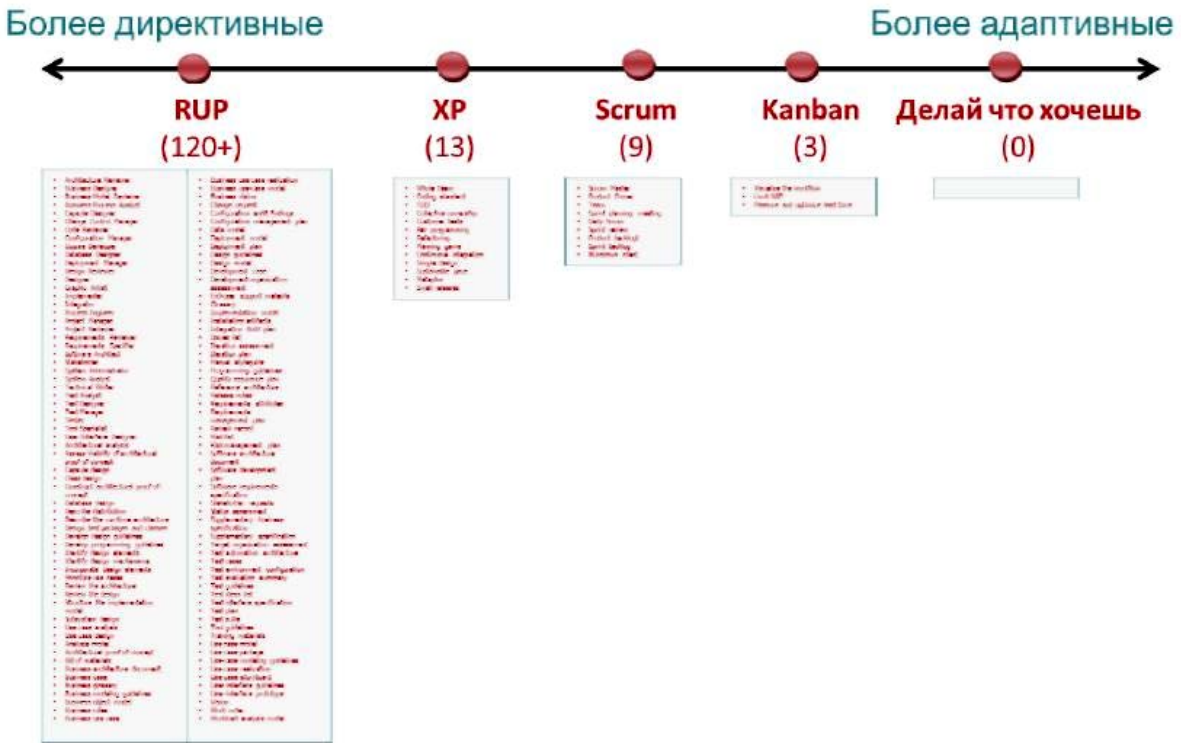


Рис. 14. Сравнение методологий

RUP – директивный метод, насчитывает более 30 ролей, более 20 мероприятий и более 70 артефактов. Вы, конечно, не должны использовать все, предполагается выбор какого-то подмножества для конкретного проекта. К сожалению, на практике это не просто. XP (eXtreme Programming) – более директивен по сравнению со Scrum. Эта методология включает в себя большую часть Scrum + конкретные инженерные практики, таких как TDD (разработка через тестирование) и парное программирование.

Scrum менее директивный, чем XP, так как он не навязывает использование никакой конкретной инженерной практики. Scrum

более директивный, чем Kanban, поскольку он предусматривает такие вещи, как итерации и кросс-функциональные команды.

Одно из основных различий между Scrum и RUP в том, что в RUP вы получаете сразу много всего, но надо избавиться от лишнего. А в Scrum – слишком мало, и надо добавить недостающее.

Kanban оставляет почти все на ваше усмотрение. Единственные ограничения – визуализировать ваш ход работы и ограничить незавершенную работу (НЗР). Не ограничивайте себя одним инструментом!

Сложно представить успешную Scrum-команду, которая не использует большинство практик XP. Многие Kanban-команды используют ежедневные пятиминутки, а это Scrum-практика. Некоторые Scrum-команды описывают задачи из backlog-а в виде Use Case – сценариев использования (практика RUP), или ограничивают размер очередей задач (практика Kanban).

Внимательно относитесь к ограничениям каждого из инструментов. Например, если вы используете Scrum и решили отказаться от использования ограниченных во времени итераций (или любой другой основной практики Scrum), то не говорите, что вы используете Scrum. Scrum сам по себе и так достаточно минимизирован, если вы удалите что-то и по-прежнему будете называть это Scrum, то слово станет бессмысленным и непонятным. Назовите это как-нибудь вроде «по мотивам Scrum» или «подмножество Scrum», или вообще «Scrum»

Роли, определенные в Scrum.

Scrum предписывает наличие трех ролей: Product Owner (отвечает за видение продукта и приоритеты), Команда (отвечает за реализацию продукта) и Scrum Master (устраняет препятствия в работе и руководит Scrum-процессом).

Kanban же не предписывает вообще никаких ролей.

Но это, однако, не значит, что вам нельзя или не стоит иметь роль Product Owner при применении Kanban! Это просто означает, что *эта роль необязательна.* Применяя как Scrum, так и Kanban, вы вольны добавлять какие угодно дополнительные роли.

Однако при добавлении ролей следует сохранять осторожность – убедитесь, что дополнительные роли действительно добавляют некую ценность и не конфликтуют с другими элементами выбранного процесса. Основным посылом как в Scrum, так и в Kanban является «Чем меньше, тем лучше». Поэтому в случае сомнений начинайте с меньшего.

Scrum основан на ограниченных по времени итерациях. Длину итерации вы можете выбрать любую, однако главная идея состоит в том, чтобы сохранить эту длину неизменной на протяжении определенного периода времени, тем самым задав *ритм (cadence).*

- **Начало итерации:** создается план итерации, т. е. команда выбирает определенное количество историй из Product Backlog, исходя из приоритетов Product Owner-а и количества работы, которое команда надеется выполнить за спринт.

- **В ходе итерации:** команда фокусируется на выполнении задач, которые она обязалась закончить. Объем работы на итерацию фиксирован.

- **Конец итерации:** команда демонстрирует работающий код заинтересованным сторонам, в идеале этот код должен быть *готовым к релизу* (т. е. протестированным и готовым к использованию). После этого команда проводит ретроспективу, чтобы обсудить и улучшить свой процесс.

Таким образом, итерации в Scrum задают единый ритм, объединяющий три различных вида деятельности: планирование, улучшение процесса, и (в идеале) релиз.

В Kanban ограниченные по времени итерации не обязательны. Вы можете сами выбирать время, когда заниматься планированием, улучшать процесс и делать релиз. Вы можете выполнять эти действия на систематически («релиз каждый понедельник») или по требованию («выпускаем релиз, как только у нас будет что-то полезное, что можно выпустить»).

**Lean и Agile.**

В целом и Scrum, и Kanban достаточно хорошо вписываются в их ценности и принципы.

И Scrum, и Kanban используют «вытягивающие» системы планирования, которые соответствуют принципу управления запасами «Точно в срок» (TBC, JIT -Just In Time). Это означает, что именно команда выбирает, когда и сколько работы взять на себя, они «вытягивают» работу, как только они готовы, а не работа «проталкивается» к ним извне. Так же, как принтер вытягивает следующую страницу только тогда, когда он готов к печати на ней.

- и Scrum, и Kanban основываются на непрерывной и эмпирической оптимизации процесса, что соответствует принципу Kaizen в Lean;

- и Scrum, и Kanban считают, что важнее реагировать на изменения, а не следовать плану (хотя Kanban обычно позволяет отреагировать быстрее, чем Scrum), что соответствует одной из четырех ценностей Agile-манифеста.

С одной стороны, может показаться, что Scrum, который предписывает объединять задачи в ограниченные по времени итераций. Но это зависит от выбранной длины итерации и от того, с чем сравнивать.

Если сравнивать с более традиционным процессом, когда мы, возможно, собираем и выпускаем что-то 2-4 раза в год, то Scrum-

команда, которая выпускает рабочий код каждые 2 недели – это уже самый настоящий Lean.

А когда вы делаете итерации все короче и короче, вы, по сути, приближаетесь к Kanban. Когда начинаются предложения о том, чтобы сделать итерации меньше одной недели, тогда можно вообще отказаться от ограниченных по времени итераций.

Незначительные различия. Есть еще несколько отличий, о которых хорошо бы знать, несмотря на то, что они кажутся менее важными, чем упомянутые выше.

В Scrum расстановка приоритетов всегда производится путем сортировки элементов Product Backlog, и изменения приоритетов вступают в силу со следующего (а не текущего) спринта. В Kanban же вы можете выбрать любой способ расстановки приоритетов (или даже вообще никакого), и он заработает, как только будет закончена какая-то задача (а не в определенное время). У команды может быть, а может и не быть Product Backlog, и он, в свою очередь, может быть приоритезированным или нет.

Scrum-команда проводит короткие встречи (не более 15 минут) каждый день, в одно и то же время в одном и том же месте. Цель этого собрания – обмен информацией о происходящем, планирование работы на текущий день и определение существенных проблем. Иногда его называют ежедневным standup.

Ежедневные встречи не обязательны в Kanban, но, тем не менее, большинство Kanban-команд все равно их устраивают. Это отличная практика, независимо от того, какой процесс вы используете.

В Scrum формат встречи ориентирован на людей – все отчитываются один за другим. Большинство Kanban-команд использует формат, более ориентированный на доску, фокусируясь на узких местах и других видимых проблемах. Этот подход более масштабируем. Если у вас 4 команды с общей доской, и они вместе проводят ежедневные

собрания, может не быть необходимости слушать каждого, так как можно сфокусироваться на отдельных проблемах.

В Scrum обязательны burndown диаграммы (рис. 15).

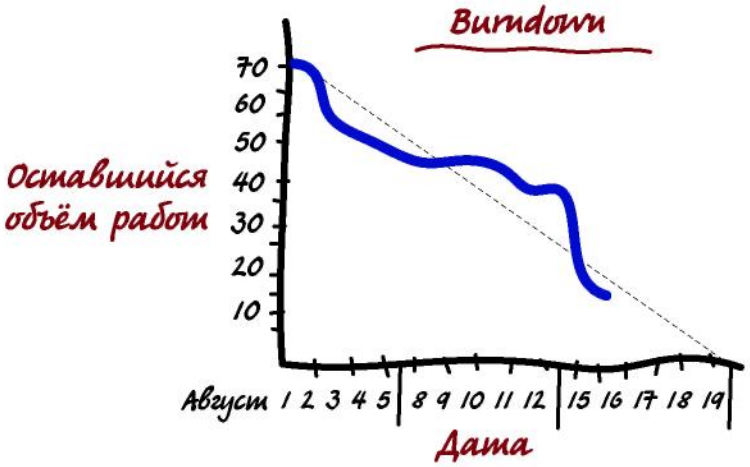


Рис. 15. Burndown диаграмма

Burndown диаграмма спринта ежедневно показывает, сколько работы осталось в текущей итерации.

Единица измерения Y-оси та же, что используется для оценки задач спринта. Обычно это часы или дни (если команда разбивает элементы backlog на задачи) или story points (если нет). Хотя, вообще-то вариантов масса. В Scrum burndown диаграммы спринта являются одним из основных инструментов отслеживания состояния итерации.

Некоторые команды также используют burndown диаграммы такого же формата, но на уровне релиза – обычно они показывают, сколько story points остается в Product Backlog после каждого спринта.

Основное назначение burndown диаграммы – как можно раньше определить, отстаем мы от графика или опережаем его, чтобы иметь возможность отреагировать.

В Kanban burndown диаграммы не обязательны. Да и вообще никаких обязательных диаграмм нет. Но, конечно же, можно использовать любые из них (в том числе и burndown).

Рассмотрим пример диаграммы суммарного потока. Этот тип диаграмм хорошо иллюстрирует, насколько гладко идет процесс и как количество незавершенной работы влияет на время выполнения задач.

Давайте посмотрим, как это работает. Каждый день подсчитывается количество элементов в каждой колонке Kanban-доски и отмечается на оси Y. Так, на 4-й день на доске было 9 элементов. Начиная с самой правой колонки: 1 элемент – готов к использованию, 1 – в тестировании, 2 – в разработке, и 5 элементов в backlog. Если мы будем ставить такие точки каждый день, а потом соединим их, то мы получим диаграмму, наподобие приведенной. Вертикальная и горизонтальная стрелки показывают взаимосвязь между количеством НЗР и временем выполнения задач (рис. 16).

Горизонтальная стрелка показывает нам, что элементам, добавленным в backlog на 4-й день, понадобилось в среднем 6 дней, чтобы стать готовыми к использованию. Примерно половину этого времени заняло тестирование. Мы можем отметить, что если ограничить количество НЗР в тестировании и backlog-е, то можно существенно сократить общее время выполнения задач.

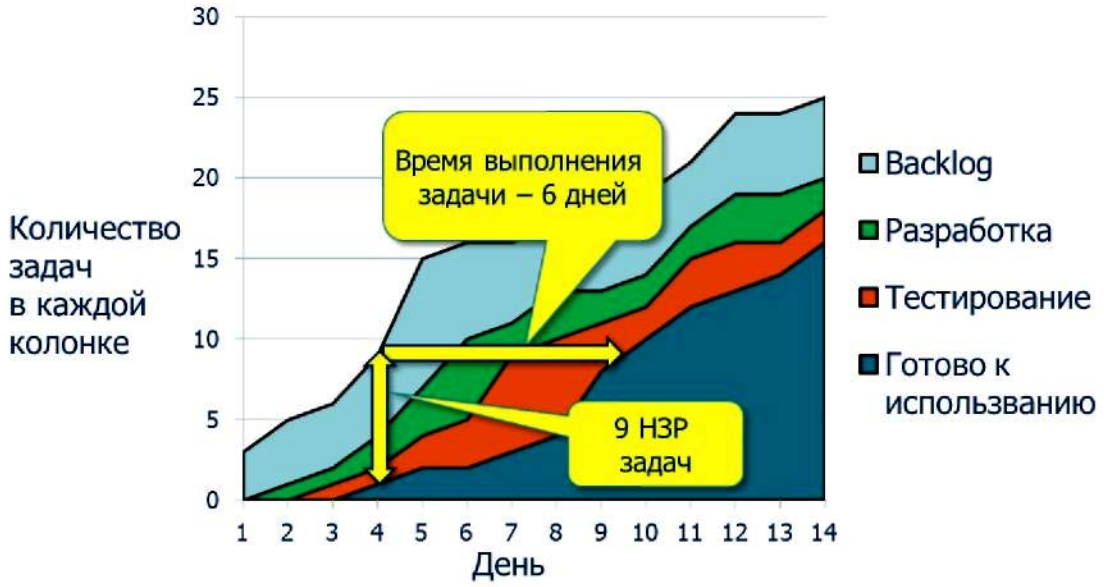


Рис. 16. Диаграмма суммарного потока

Наклон нижней области отображает производительность (т. е. количество завершенных элементов в день). Со временем мы сможем увидеть, как повышение производительности уменьшает время выполнения задачи, в то время как увеличение количества НЗР увеличивает это время.

Сходства Scrum против Kanban:

- и Lean, и Agile;
- используют вытягивающие системы планирования;
- ограничивают НЗР;
- используют прозрачность для обеспечения улучшения процесса;
- ориентированы на ранние и частые поставки продукта;
- полагаются на самоорганизующиеся команды;
- требуют деления задач на более мелкие.

В обоих случаях план релиза постоянно оптимизируется на основе эмпирических данных (производительности/ времени выполнения задачи).

Отличия:

| **Scrum** | **Kanban** |
|---|---|
| Обязательны ограниченные по времени итерации | Ограниченные по времени итерации необязательны. Могут быть отдельные ритмы для планирования, выпуска и усовершенствования процессов. Также могут быть событийно-управляемые итерации вместо ограниченных по времени |
| Команда обязуется выполнить конкретный объем работы за эту итерацию | Обязательства опциональны |
| Как основная метрика для планирования и улучшения процессов используется производительность | Как основная метрика для планирования и улучшения процессов используется время выполнения задачи |

Окончание

| Scrum | Kanban |
|---|---|
| Кросс-функциональные команды обязательны | Кросс-функциональные команды, опциональны<br>Допустимы узкопрофильные команды |
| Задачи должны быть разбиты на более мелкие так, чтобы они были завершены в течение одного спринта | Нет каких-либо определенных размеров задач |
| Наличие burndown-диаграммы обязательно | Наличие каких-либо обязательных диаграмм не требуется |
| НЗР ограничивается косвенно (за спринт) | НЗР ограничивается явно (по статусам) |
| Оценки задач обязательны | Оценки задач опциональны |
| Нельзя двлять задачи в текущую итерацию | Можно двлять новые задачи, когда это возможно |
| За backlog спринта отвечает только одна конкретная команда | Kanban-доска может совместно использоваться несколькими группами или отдельными лицами |
| Предписаны 3 роли (Product Owner / Scrum Master / Команда) | Нет предписанных ролей |
| Scrum-доска очищается между спринтами | Kanban-доска является неизменной |
| Приоритезированный Product Backlog обязателен | Приоритезация не является обязательной |

## 3.3 Языки конструирования

Языки конструирования включают все формы коммуникаций, с помощью которых человек может задать решение проблемы, выполняемое на компьютере.

Простейший тип языков конструирования – конфигурационный язык (configuration language), позволяющий задавать параметры выполнения программной системы.

Инструментальный язык (toolkit language) – язык конструирования из повторно-используемых элементов; обычно строится как сценарный язык (script), выполняемый в соответствующей среде.

Язык программирования (programming language) – наиболее гибкий тип языков конструирования. Содержит минимальный объем информации о конкретных областях приложения и процессе разработки, требуя больше всего (по сравнению с другими типами языков конструирования) усилий на изучение и наработку опыта для эффективного применения при решении конкретных задач.

Существуют три основных вида нотаций используемых при определении языков программирования:

- лингвистическая
- формальная
- визуальная

Лингвистические нотации характеризуются, в частности, использованием строк текста, содержащих специализированные «слова», представляющие сложные программные конструкции, и комбинируемые в шаблоны, напоминающие предложения, построенные в соответствии с определенным синтаксисом. В случае корректного использования таких нотаций, каждая получаемая строка обладает строгой смысловой нагрузкой (семантикой), обеспечивающей интуитивное понимание того, что будет происходить, когда будет выполняться программное обеспечение, построенное с использованием такого языка конструирования.

Формальные нотации являются менее интуитивными, чем лингвистические, и часто базируются на точных формальных (математических) определениях. Формальные нотации конструкций и формальные методы являются ядром практически всех форм системного программирования, точнее, поведения систем во времени. Такие нотации обеспечивают наибольшую готовность получаемого

кода к тестированию, что намного важнее, чем просто возможность отображения на обычный человеческий язык. Формальные конструкции также используют точный метод определения комбинаций применяемых символов, что позволяет избежать неоднозначностей, присущих конструкциям естественных языков.

Визуальные нотации наименее связаны с текстово-ориентированными подходами, предполагая непосредственную интерпретацию визуальных конструкций в процессе исполнения описываемой логики. При этом логика в визуальных нотациях задается расположением соответствующих визуальных сущностей, ответственных за те или иные операции и структуры данных.

Использование визуальных конструкций ограничено сложностью визуального представления сложных выражений и утверждений только за счет перемещения визуальных сущностей на диаграмме (визуальном представлении). Однако визуальная нотация может играть роль достаточно мощного инструмента, когда применяется в тех задачах программирования, где необходимо построение пользовательского интерфейса для программ, чья логика. Детализированное поведение определено ранее.

Сегодняшние работы (и их состояние) в области архитектур и приложений, управляемых моделями, в первую очередь OMG MDA (Model-Driven Architecture www.omg.org/mda)/UML (Unified Modeling Language www.omg.org/uml), Microsoft DSL (Domain-Specific Language), направлены на то, чтобы использовать ту или иную визуальную нотацию, базирующуюся на мета-моделях, в качестве инструмента, применяемого и для определения функциональной логики системы. Можно ли считать эти нотации именно визуальными – вопрос спорный. Но они предполагают однозначную интерпретацию визуального представления в виде текста и наоборот. Кроме того, исторически эти нотации определялись изначально как

нотации визуального представления функциональности и уже в дальнейшем эти визуальные представления были отражены на уровне соответствующих мета-моделей (хотя это в большей степени верно для UML, чем DSL, но DSL можно рассматривать и как аналог UML, предполагающий большую свободу применений и интегрированность с конкретной платформой Microsoft).

Другая область стандартов, направленных на применение визуальных нотаций для описания функциональности – Business Process Management Notation (BPMN, www.omg.org/bpmn) и связанный с ней язык Business Process Execution Language, построенный на базе XML. Таким образом, область обоснованного применения визуальных нотаций для конструирования программных систем качественно расшириться и, не исключено, мы увидим de-facto формирования новой категории нотаций, соглашений и смешанных типов языковых средств, предназначенных для конструирования программного обеспечения как естественного продолжения проектирования.

**BPEL**

«Корпоративная Информационная Система (КИС)», как правило, представляет собой набор программного обеспечения разной степени однородности. То есть в лучшем случае все необходимое ПО (для организации бухгалтерского, оперативного и складского учетов, документооборота, сервисных служб, call-центра) закупается у одного вендора.

В то же время понятно, что пользователи (сотрудники предприятия) всегда должны иметь возможность получать любую необходимую им информацию, независимо от того, в какой системе она хранится и/или обрабатывается.

Для этого необходимо интегрировать все приложения, предоставляющие и обрабатывающие информацию в единую

информационную систему. Одной из концептуальных возможностей решения такой задачи является использование SOA.

Сервисно-ориентированная архитектура (COA, SOA) – понятие, которое в последнее время становится мэйнстримом в области разработки корпоративных систем. Суть SOA в том, что информационная система разбивается на несколько функционально законченных узлов – сервисов, которые взаимодействуют друг с другом. Собственно, это и есть концепция SOA, остальное (устройство сервисов, обеспечение связей между ними) – детали.

Сервис в терминах SOA и веб-сервис – разные вещи. Веб-сервисы – это один из наиболее распространенных способов реализации сервисов в рамках SOA. В рамках SOA сервисы можно заставить взаимодействовать через RMI, CORBA, OSGi, а самим сервисом может быть та же 1С Бухгалтерия. Точно также веб-сервисы могут взаимодействовать без всякой SOA.

Примерами сервисов могут являться такие узлы, как менеджер задач (всем сотрудникам в рамках КИС будут создаваться задачи и будет контролироваться их выполнение), сервис расчета зарплаты, сервис отображения начисленной зарплаты в интерфейсе КИС, сервис взаимодействия с пенсионным фондом, с налоговой инспекцией, сервис складского учета и т. д. Главное, что сервис — это некий черный ящик, который умеет получать, обрабатывать и выдавать информацию.

Раз сервисы обмениваются информацией, то возникает вопрос: «как организовать этот обмен». Дело в том, что сервисы потребляют, обрабатывают и выдают самую разную (как по семантике, так и по формату) информацию (бухгалтерской системе абсолютно безразлично, распилены в цехе доски или нет, но не безразлично, сколько часов их пилили и сколько рабочего времени потратили, его ведь надо оплатить). Более того, следует учитывать большое количество уже

имеющихся legacy-систем, которые имеют свой формат представления информации.

Таким образом, первая проблема взаимодействие сервисов – согласование информации и форматов ее передачи.

Решение данной задачи напрямую зависит от того, как соединим сервисы. Мы можем соединить сервисы друг с другом напрямую. Тогда получим огромное количество связей, в которых со временем будет трудно разобраться. Плюс к этому нам необходимо будет писать конвертеры данных из формата одного сервиса в форматы каждого сервиса, с которым он взаимодействует.

Другим, более правильным, решением является использование сервисной шины (ESB – Enterprise Service Bus). Сервисная шина является посредником между сервисами. Соответственно, она обеспечивает унификацию формата обмена (теперь нужно всего лишь написать для каждого сервиса один конвертер – в формат шины), а также синхронизацию и управление обменом между сервисами.

В мире Java существуют такие реализации ESB, как:

- Sun Microsystems – OpenESB;
- IBM – IBM WebSphere ESB;
- BEA Systems (куплена Oracle) – ALSB (AquaLogic Service Bus);
- Oracle – Application Server 10g + WebService Manager;
- JBoss – JBoss ESB + JBoss Web Services.

**BPM**

Итак, первым вопросом было «как заставить сервисы взаимодействовать?». Вторым вопросом является: «как управлять вызовом сервисов?».

Деятельность любого предприятия представляет собой множество параллельно выполняющихся последовательностей создания-выполнения-контроля-завершения задач, т. е. множество бизнес-процессов.

Примеры бизнес-процессов: обработка входящего письма, подготовка исходящего документа, составление инструкции, создание шестеренки, расчет заработной платы для сотрудника, оформление отпуска, оформление командировки и т. д.

Раз любой бизнес – это ни что иное, как выполнение бизнес-процессов, то и управление бизнесом сводится к управлению бизнес-процессами. В то же время управление бизнес-процессами включает в себя и их автоматизацию. Именно для автоматизации бизнес-процессов и для контроля за их исполнением и заказывают дорогостоящие КИС.

Таким образом, мы вплотную подошли к рассмотрению класса систем, которые называются BPMS (Business Processes Management Systems – системы управления бизнес-процессами). В чем же особенности разработки таких систем? BPMS очень удобно разрабатывать в рамках концепции SOA.

Фактически, реализация любого бизнес-процесса в рамках SOA является лишь последовательностью (зачастую очень хитрой и запутанной) вызовов тех или иных сервисов. То есть у нас есть набор строительных блоков – сервисов, задаем последовательность вызовов этих блоков – получаем автоматизацию бизнес-процесса. Меняем последовательность вызовов – получаем автоматизацию другого бизнес-процесса.

Несомненным плюсом такого подхода является высочайшая масштабируемость, причем как техническая, так и функциональная. Двляем новые сервисы – реализуем новые бизнес-процессы. Разворачиваем каждый сервис на своей машине – решаем проблемы производительности.

Теперь посмотрим, что же предлагают нам вендоры для работы с BPM:

- Sun Microsystems – Application Server GlassFish + java.sun.com/javaee/community/;

- IBM – IBM WebSphere Business Modeller, IBM WebSphere Integration Developer, IBM WebSphere Process Server;
- BEA Systems – BPM (Business Process Management), DSP (Data Services Platform);
- Oracle – Application Server 10g + WebService Manager;
- JBoss – JBoss Web Services + JBoss BPM;
- Active Endpoints – ActiveBPEL Engine + ActiveBPEL Designer.

BPEL.

Вообще, некоторые вендоры (например, JBoss) предлагают свой формат описания, однако существует стандарт, и этим стандартом является язык Business Process Modeling and Execution (BPEL). BPEL – XML-based язык, что позволяет довольно легко его распознать машиной и в то же время читать и даже слегка править человеком. Также сохраняются все преимущества текстовых форматов, такие как легкость в передаче по сети, читабельность, работа с системами контроля версий и т. д.

Как и в любом языке программирования, в BPEL есть набор конструкций (можно рассматривать его как набор операторов). Операторов, которыми задается непосредственно действие, всего два: присваивание и вызов веб-сервиса (есть реализации, позволяющие вызывать EJB). Остальные конструкции являются управляющими — их задача обеспечить правильную последовательность присваиваний и вызовов внешних сервисов. В теории предполагается, что бизнес-аналитик будет рисовать бизнес-процесс, задавая порядок вызова сервисов, а программисты – разрабатывать эти сервисы.

Также любой BPEL-процесс имеет точку входа и точку выхода. Правда, это не совсем соответствует понятию «начало» и «конец» на схеме алгоритма. Точка входа представляет собой уведомление BPEL-машине о том, что надо создать и запустить экземпляр бизнес-процесса. Точка выхода – уведомление вызвавшей процесс системе о

том, что процесс запущен. Также можно передать некоторые данные в вызвавшую систему, например pid процесса. После того как мы уведомили вызвавшую нас систему о том, что процесс запущен, можем продолжать его исполнение.

## 3.4 Тестирование в модели жизненного цикла разработки ПО

### 3.4.1 Определения

«Тестирование программ может служить доказательством наличия ошибок, но никогда не докажет их отсутствие!» Эд. Дейкстра, 1972 г.

С точки зрения ISO 9126, качество программных средств можно определить как совокупную характеристику исследуемого ПО с учетом следующих составляющих:

- надежность;
- сопровождаемость;
- практичность;
- эффективность;
- мобильность;
- функциональность.

Более полный список атрибутов и критериев можно найти в стандарте ISO 9126 Международной организации по стандартизации. Состав и содержание документации, сопутствующей процессу тестирования, определяется стандартом IEEE 829-1998 Standard for Software Test Documentation.

В первую очередь нужно отметить, что вопросы тестирования следует рассматривать в контексте всего жизненного цикла ПО, начиная от разработки ТЗ и заканчивая сопровождением приложений. Как известно, тестирование – это процедура обнаружения дефектов (ошибок) ПО до его промышленного использования. Очевидно, что

трудоемкость такой работы связана с количеством самих ошибок, в связи с чем надо четко выделить основные причины их появления:

- неудовлетворительное организационное, методическое и техническое обеспечение всего процесса разработки;

- сжатые сроки исполнения проекта;

- сложность проекта, большое число требований и их изменений по ходу работы;

- недостаточная квалификация разработчиков.

Тестирование является лишь составляющей частью отладки – процесса доводки ПО после его написания до эксплуатационного состояния. Процесс этот включает две основные процедуры: обнаружение ошибок (тестирование) и поиск и устранение их причин. Однако, даже учитывая все возможные взаимосвязи этих работ (например, поиск причин ошибок требует проведения специального дополнительного тестирования), нужно подчеркнуть, что тестирование является достаточно автономным, независимым этапом жизненного цикла ПО.

Тестирование пронизывает весь жизненный цикл ПО, начиная от проектирования и заканчивая неопределенно долгим этапом эксплуатации. Эти работы напрямую связаны с задачами управления требованиями и изменениями, ведь целью тестирования является как раз возможность убедиться в соответствии программ заявленным требованиям.

Тестирование – процесс пошаговый. Наверное, имеет смысл разделить проверку работоспособности программ в ходе непосредственного написания кода (самим программистом) и после завершения основного этапа кодирования (скорее всего, специальными тестировщиками). Тут можно вспомнить о *правиле программирования*: написание каждых 20–30 строк кода (тем более законченных процедур, функций) должно сопровождаться проверкой

их работоспособности, хотя бы в каком-то основном режиме. В то же время нужно подчеркнуть и важное различие в проведении тестирования в ходе кодирования и по его завершении: в первом случае продолжать написание программы (а также запуск других тестовых примеров) желательно только после устранения ошибки, во втором осуществляется пакетное выполнение серии текстов с простой фиксацией их результатов.

Тестирование – процесс также итерационный. После обнаружения и исправления каждой ошибки обязательно следует повторение тестов, чтобы убедиться в работоспособности программы. Более того, для идентификации причины обнаруженной проблемы может потребоваться проведение специального дополнительного тестирования.

Одна из ключевых проблем кроется в правильном определении понятия тестирования, так как это далеко не тривиальная и не однозначная задача. Для того, чтобы убедиться в этом, обратимся к рассуждениям основоположника теории тестирования Гленфорда Майерса (Glenford Myers).

Но, прежде всего, приведем *два главных закона теории тестирования программных продуктов*:

1. Невозможно отыскать абсолютно все ошибки в программном продукте. Ошибки остаются всегда.

2. Построение исчерпывающего входного теста невозможно.

Иными словами, невозможно полностью протестировать программу: даже для самой простейшей программы это займет настолько большое количество времени, которое никогда не будет восполнено выгодой от производства «идеально оттестированной» программы. При тестировании современных сложных программных систем исчерпывающее тестирование становится невыполнимой задачей.

Майерс приводит следующие наиболее распространенные определения тестирования с комментариями к ним:

1. Тестирование – это процесс, позволяющий убедиться в том, что в программе нет ошибок.

Данный результат недостижим, исходя из первого закона теории тестирования, приведенного выше. Но, если следовать данному определению и преследовать именно такую цель в тестировании, то можно искусственно показать, что ошибок нет. Что заранее будет неверно, поэтому проведение тестирование с целью демонстрации отсутствия ошибок приведет лишь к провалу проекта.

2. Цель тестирования – показать, что программа корректно выполняет предусмотренные функции, т. е. программа соответствует спецификации. Или, более детально, цель тестирования – показать, в каких ситуациях программа не соответствует спецификации, в то время как тестовые данные используются в соответствии со спецификацией программы.

Приведенное определение является определением одного из критериев тестирования программы по методу «черного ящика». Но, как мы уже определили ранее, у тестирования есть набор методов и критериев, а метод «черного ящика» – лишь частный случай. В общем случае, обнаружение всех ошибок в программе является критерием «исчерпывающего входного тестирования», но построение исчерпывающего входного теста невозможно (см. второй закон теории тестирования, приведенный выше).

3.Тестирование – это процесс, позволяющий убедиться в том, что программа выполняет свое назначение.

Данное определение звучит логично: если программа не делает того, что от нее требуется, то ясно, что она содержит ошибки. Но как быть с тем, что она дополнительно делает еще и то, чего от нее не требуется. Появляется необходимость проводить так называемое

«негативное» тестирование. Следовательно, данное определение тестирования не совсем корректно.

Далее Майерс дает собственное определение тестирования: Тестирование ПО – это процесс выполнения программы с целью обнаружения ошибок.

Майерс считает тест удачным, если в процессе его выполнения были обнаружены ошибки. Именно в этом и состоит задача тестирования.

Немного модифицируем определение тестирование, данное Майерсом, которое на сегодняшний день является слегка устаревшим.

Тестирование ПО (software testing) – это процесс анализа и эксплуатации программного обеспечения с целью выявления дефектов. Под дефектом, в соответствии с RUP, будем понимать невыполнение требования, связанного с предполагаемым или установленным использованием. В определении стоит обратить внимание на ключевое слово «процесс». Тестирование – это плановая и упорядоченная деятельность. Этот момент очень важен, поскольку в условиях зачастую очень ограниченного времени, выделенного на разработку и тестирование, хорошо продуманный и систематический подход быстрее приводит к обнаружению программных ошибок, чем плохо спланированное тестирование, проводимое в спешке.

Определение тестирования, выведенное Майерсом, следует помнить при выполнении тестирования. Но все же понимать современное тестирование следует несколько шире. Поэтому при описании технологии тестирования будем придерживаться более официального определения тестирования, приведенного в международных стандартах по тестированию.

В 1990 году стандартом ISO принято следующее определение тестирования:

Тестирование – это наблюдение за функционированием ПО в специфических условиях с целью определения степени соответствия ПО требованиям к нему.

При переходе к современным методологиям разработки ПО наблюдается более системный подход в определении тестирования ПО. В соответствии с RUP, тестирование – одна из дисциплин RUP. Она ориентирована в первую очередь на оценку качества с помощью следующих методов:

- поиск и документирование дефектов качества;
- общие рекомендации относительно качества;
- проверка выполнения основных предположений и требований на конкретных примерах;
- проверка, что продукт функционирует так, как было запроектировано;
- проверка, что требования выполнены соответствующим образом.

Обобщенная модель жизненного цикла тестирования ПО может быть представлена в виде буквы V, как показано на рис. 17. Такая концепция получила соответствующее название – V-модель.

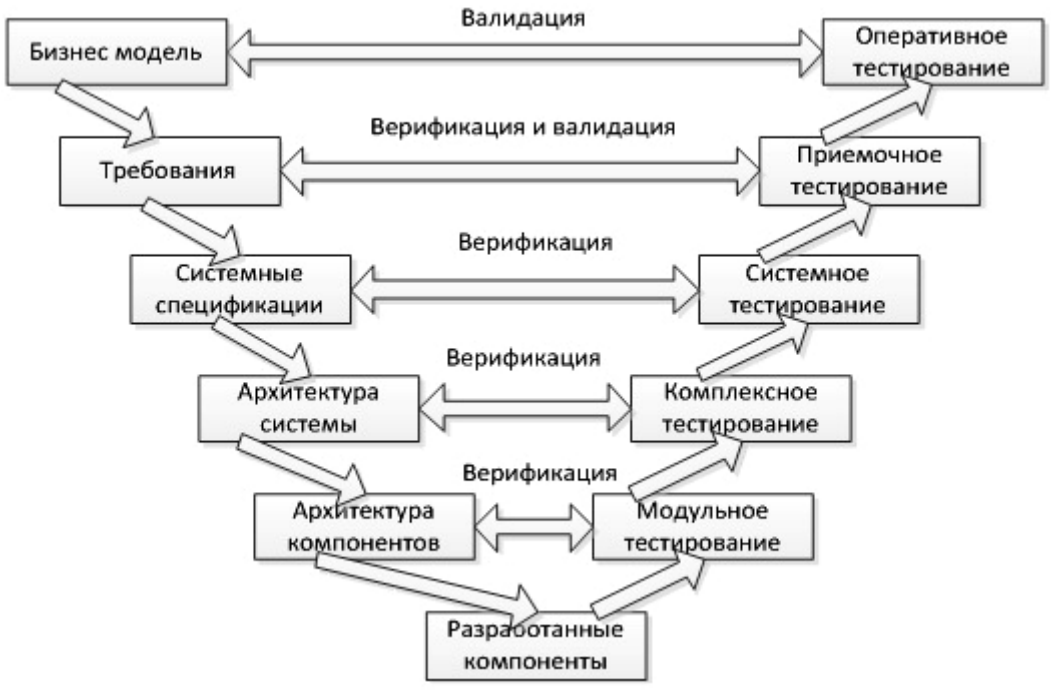


Рис. 17. V-образная модель обобщенного ЖЦ ТП

Это один из способов продемонстрировать, как соотносятся процессы тестирования с основными процессами проектирования и разработки. В V-модели основные этапы ЖЦ ПО образуют левую сторону «V», кодирование находится в самой нижней точке диаграммы, а тестирование образует ее правую сторону. Для простоты изложения вид деятельности, подобный сопровождению, на диаграмме не показан.

Следует уточнить, что, точно так же как обобщенная модель ЖЦ ПО не определяет строгой последовательности этапов разработки и не исключает цикличность и пересмотр всех или некоторых из них, так и «V»-модель – лишь показывает основные этапы тестирования ПО и их связь с этапами разработки. Обобщенная модель ЖЦ тестирования ПО может быть преобразована к виду конкретной модели ЖЦ ПО. Так, при работе по итеративной или спиральной модели, «V»-модель тестирования приобретает итеративную природу, как и другие процессы разработки. Это означает, что обозначенные виды (уровни) тестирования могут неоднократно повторятся в процессе ЖЦ ПО в каждой итерации разработки, а также – в качестве регрессионного тестирования уже разработанных и протестированных на предыдущих итерациях элементов системы.

Таким образом, на основе современных методологий, стандартов качества и стандартов разработки ПО мы систематизировали общее понимание тестирования и определили обобщенную модель ЖЦ ТП, которая обладает следующими качествами:

1. ЖЦ ТП систематически связана с ЖЦ ПО;
2. ЖЦ ТП закрывает все уровни проверки качества ПО;
3. ЖЦ ТП используется для валидации и верификации, которые являются крайне важными процессами по обеспечению качества ПО; даже название обобщенной модели ЖЦ ТП «V-модель» отражает свой главный смысл – Верификация и Валидация

(Verification and Validation) качества ПО, как неотъемлемая составляющая любой современной разработки;

4. ЖЦ ТП позволяет понять, что, как и для чего должно быть подвержено тестированию в каждой точке ЖЦ ПО;
5. Обобщенная модель ЖЦ ТП может быть преобразована под любую модель ЖЦ ПО. В частности, она может быть преобразована под методологию RUP, приобретая цикличность в двух направлениях – горизонтальном (верификация на каждом уровне) и вертикальном (валидация по всем уровням).

Таким образом, тестирование вправе называться самостоятельной дисциплиной, которая имеет свои цели, задачи, роли, виды, стратегии, методы, критерии, свою методологию и технологию.

### 3.4.2 Циклы тестирования

Как уже было определено, обобщенная модель ЖЦ ТП приобретает итеративную природу при итеративной разработке. Помимо этого тестирование обычно проводится циклами, каждый из которых имеет конкретный список задач и целей. Поэтому можно сделать вывод о двойной цикличности процесса тестирования, если разработка ведется по итеративной или спиральной модели ЖЦ ПО.

Можно выделить *два вида циклов тестирования* (полный цикл тестирования и частный цикл тестирования):

1. Полный цикл тестирования обычно совпадает с итерацией разработки или соответствует ее определенной части. Очевидно, что, в случае разработки программного продукта по каскадной модели, полный цикл тестирования скорее всего будет иметь только одну итерацию.

2. Частный цикл тестирования, как правило, проводится для конкретной сборки объекта тестирования (системы, подсистемы или отдельного компонента).

Краткие описания задач, входящих в **частный цикл тестирования:**

1. Определить цели тестирования. Включает выбор тестируемых фрагментов и формулирование задач тестирования (например, определить пригодность архитектуры, проверить реализацию основной функциональности конкретного Сценария использования или проверить выполнение требований Заказчика в полном объеме).

2. Верифицировать метод тестирования. Настройка среды и инструментов тестирования, выполнение отдельных тестов, подтверждение возможности реализовать задачи и цели тестирования.

3. Подтвердить правильность сборки. Прежде чем приступить к детальному тестированию выбранной сборки, проводятся ее тесты «на дым». Эти тесты должны показать, что сборка не содержит явных ошибок, делающих ее дальнейшее тестирование просто нецелесообразным. Для «проходных» сборок, в которых не реализован достаточный объем новой функциональности, тестирование может на этом и заканчиваться.

4. Тестировать и оценивать. Разрабатываются (уточняются) необходимые тесты, после чего тесты выполняются в ручном или автоматическом режиме, и проводится оценка результатов. Достичь приемлемого уровня достижения целей тестирования. Оценивается, с одной стороны, качество и эффективность тестирования, а, с другой стороны, качество тестируемой системы и ее соответствие требованиям, предъявляемым на данном этапе разработки проекта.

5. Улучшить набор тестов и другие активы для дальнейшего использования. Описать и сохранить тесты, наборы тестовых данных, настройки среды и инструментальных средств, которые можно использовать в последующих тестовых циклах.

Частный цикл тестирования, проводимый для отдельной сборки объекта тестирования на текущем этапе ЖЦ ТП, представлен на рис. 18.

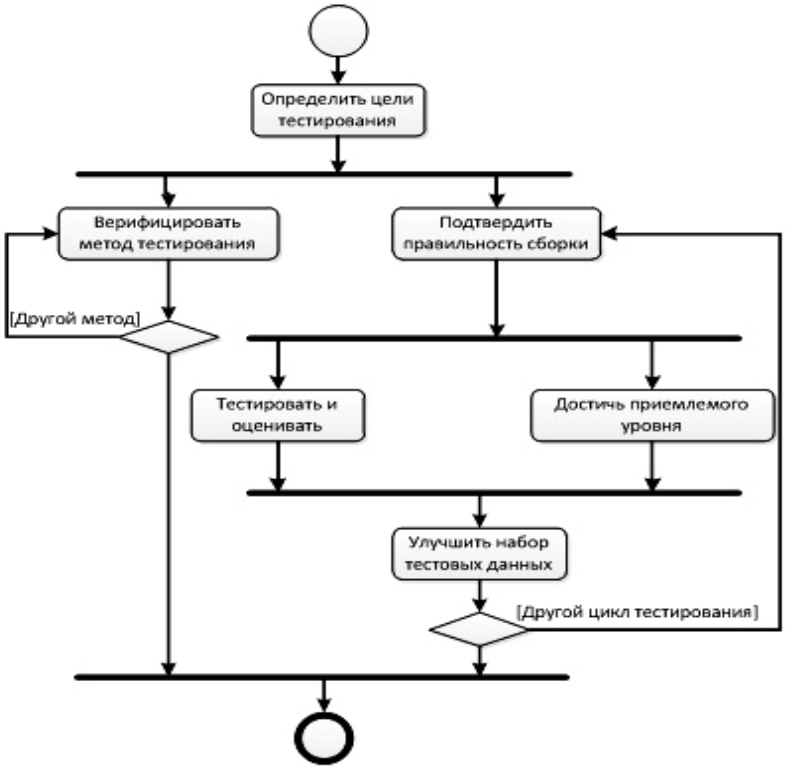


Рис. 18. Частный цикл тестирования, проводимый для отдельной сборки объекта тестирования

Выполнение задач жизненного цикла сопровождается разработкой различных артефактов (документов, моделей и других материалов проекта). Как обычно в RUP, разработка артефактов может проводиться в разной форме с разными требованиями к способу выполнения, рецензированию и качеству оформления.

Прежде чем описывать полный цикл тестирования, определим основные артефакты этого процесса. Ниже представлены основные рабочие артефакты тестировщиков, в той или иной форме связанные со сценариями использования. Эти документы, если это не оговорено особо, стоит готовить в достаточно аккуратном виде, поскольку, скорее всего, вам придется неоднократно к ним возвращаться самим, а часто еще и передавать их Заказчику или группе сопровождения и технической поддержки системы.

**Полный цикл тестирования и его задачи**

Рассмотрим более подробно существующие активности/задачи связанные с тестированием:

1) Планирование тестов: определение требований к тестам; оценка рисков; выбор стратегии тестирования; определение ресурсов; создание расписания/последовательностей; разработка Плана тестирования.

2) Дизайн тестов: анализ объема работ; определение и описание тестовых случаев; определение и структурирование тестовых процедур; обзор и оценка тестового покрытия.

3) Разработка тестов: запись или программирование тестовых скриптов; определение тесто-критичной функциональности в Дизайне и Модели реализации; создание/подготовка внешних наборов данных.

4) Выполнение тестов: выполнение тестовых процедур; оценка выполнения тестов; восстановление после сбойных тестов; проверка результатов; исследование неожиданных результатов; запись ошибок.

5) Оценка тестов: оценка покрытия тестовыми случаями; оценка покрытия кода; анализ дефектов; определение критериев завершения и успешности тестирования.

На основе перечисленных задач и активностей можно определить полный цикл активностей тестирования.

Таким образом, помимо уже определенной итеративности V-модели жизненного цикла ТП, она приобретает двойную цикличность за счет того, что общие и/или частные циклы тестирования могут происходить конечное число раз в пределах итерации.

### 3.4.3 Основные артефакты тестирования

**План тестирования.** Основной документ, определяет стратегию тестирования на каждой итерации. В него входят описание целей и задач тестирования в текущей итерации, перечни тестов,

которые должны и не должны использоваться, формируемые начальные и конечные метрики и критерии тестирования.

**Сценарий тестирования** (тест кейс или тестовый случай). Это один из основных документов, с которыми имеет дело тестировщик. По сути, упрощенное описание теста, то есть входной информации, условий и последовательности выполнения действий и ожидаемого выходного результата. Учитывая, что даже успешно прошедшие тесты в RUP выполняются неоднократно в ходе регрессионного тестирования, наличие таких описаний необходимо. Однако уровень формальных требований к их оформлению может меняться в очень широких пределах.

**Тестовые данные**. Призваны определять наборы (обычно формальных) входных данных для тестов, а также ожидаемые результаты, с которыми полученные результаты выполнения тестов должны сравниваться как с эталонными. Тестовые данные должны храниться в одном месте, желательно в центральном хранилище данных. Рекомендуется собирать вместе данные для каждой определенной группы тестов.

**Тестовый скрипт.** Обычно говорят о программной реализации теста, хотя скрипт может описывать и ручные действия, необходимые для выполнения конкретного тест кейса.

**Набор тестов.** Как правило, сценарии тестирования объединяются в пакеты или наборы. Во-первых, это просто способ группирования тестов со сходными задачами, а, во-вторых, в такой набор можно включать зависимые тесты, которые должны выполняться в определенном порядке (поскольку последующие тесты используют данные, сформированные в ходе выполнения предыдущих).

**Список идей тестов.** Использование в RUP для анализа и проектирования Системы Сценариев существенно упрощает задачу

разработки необходимого набора тестов. Основной объем тестов строится как проверка различных вариантов выполнения каждого сценария использования. Однако тесты не сводятся к Сценариям использования, как и задачи тестирования не сводятся только лишь к проверке функциональных требований к системе. Проверка нефункциональных требований может потребовать использования специальных приемов и подходов. Соответствующие тесты не всегда очевидны. Для таких ситуаций и создается Список идей тестов. В него все желающие могут записать Что и/или Как стоит еще проверить. Этот список является внутренним рабочим документом группы тестирования.

**Результаты тестирования.** Представляют собой суммарную информацию о прохождении тестов, на основе анализа которых и сравнения с ожидаемыми результатами выполняется детальная оценка качества тестируемого продукта и текущего статуса процесса тестирования. Рекомендуется записывать и сохранять результаты тестирования для каждого этапа как один из важнейших артефактов тестирования.

**Дефекты.** Основополагающие артефакты процесса тестирования – описывают обнаруженные факты несоответствия системы предъявляемым требованиям. Являются одним из подтипов запросов на изменение, описывающих найденную ошибку или несоответствие на всех этапах тестирования. Хотя базу данных дефектов можно вести в текстовом файле или Excel таблице, предпочтительным является использования специализированного инструментального средства, которое позволяет передавать информацию об обнаруженных дефектах от тестировщиков к разработчикам, а в обратную сторону – сведения об устранении дефектов. А также формировать необходимые отчеты о тенденциях изменения количества обнаруживаемых и устраняемых дефектов.

### 3.4.4 Стратегии тестирования

Различие задач и целей тестирования на протяжении жизненного цикла продукта приводит к необходимости разрабатывать и реализовывать различные стратегии тестирования. Каждая такая стратегия определяет:

1) итерации, на которых используются стратегия тестирования и цели тестирования на каждой итерации;

2) стадии тестирования для каждой итерации;

3) критерий успешного завершения тестирования;

4) типы используемых тестов;

5) набор методов и инструментальных средств, необходимых для проведения тестирования и оценки качества;

6) критерии оценки тестов.

Стратегии тестирования должны разрабатываться на этапе планирования тестирования.

**Тестирование «белого ящика» и «черного ящика»**

В терминологии профессионалов тестирования фразы «тестирование белого ящика» и «тестирование черного ящика» относятся к тому, имеет ли разработчик тестов доступ к исходному коду тестируемого ПО, или же тестирование выполняется через пользовательский интерфейс либо прикладной программный интерфейс, предоставленный тестируемым модулем.

При тестировании «белого ящика» (англ. *white-box testing*, также говорят – прозрачного ящика), разработчик теста имеет доступ к исходному коду программ (см. открытое программное обеспечение) и может писать код, который связан с библиотеками тестируемого ПО. Это типично для юнит-тестирования (англ. *unit testing*), при котором тестируются только отдельные части системы. Оно обеспечивает то, что компоненты конструкции – работоспособны и устойчивы, до

определенной степени. При тестировании «белого ящика» используются метрики покрытия кода.

При тестировании «черного ящика», тестировщик имеет доступ к ПО только через те же интерфейсы (например, при интеграции приложений), что и заказчик или пользователь, либо через внешние интерфейсы, позволяющие другому компьютеру либо другому процессу подключиться к системе для тестирования. Например, тестирующий модуль может виртуально нажимать клавиши или кнопки мыши в тестируемой программе с помощью механизма взаимодействия процессов, с уверенностью в том, все ли идет правильно, что эти события вызывают тот же отклик, что и реальные нажатия клавиш и кнопок мыши. Тестирование «черного ящика» ведется с использованием спецификаций или иных документов, описывающих требования к системе. В данном виде тестирования критерий покрытия складывается из покрытия структуры входных данных, покрытия требований и покрытия модели (в тестировании на основе моделей).

При тестировании «серого ящика» разработчик теста имеет доступ к исходному коду, но при непосредственном выполнении тестов доступ к коду не требуется.

### 3.4.5 Метрики и критерии тестирования

При проведении тестирования необходимо определить критерии окончания процесса тестирования. Ведь недостаток тестирования может вести к выпуску продукта с существенными недостатками. А «лишнее» тестирование может стоить достаточно дорого, задерживать выпуск продукта и отвлекать тестировщиков от других работ.

Чтобы принять решение о прекращении тестирования, выбрать оптимальный набор тестов и для многих других целей, используются

*метрики тестирования и качества*. Они позволяют оценить покрытие кода продукта тестами, спрогнозировать число ненайденных дефектов, оценить характеристики тестируемой системы.

Еще одним важным понятием в теории тестирования является понятие критериев покрытия тестирования. Не следует путать *метрики тестирования* и *критерии покрытия тестирования*. Последние позволяют определить степень покрытия разрабатываемого продукта тестами. Поэтому часто критерии покрытия используются для определения метрик тестирования.

Приведем примеры самых распространенных критериев покрытия при тестировании функциональных требований в соответствии с методологией RUP.

При тестировании функциональных требований могут быть выделены, по крайней мере, два типа покрытия: покрытие, основанное на спецификации, и покрытие, основанное на коде.

Покрытие, основанное на спецификации или на требованиях (Specification-Based Coverage or Requirements-based Test Coverage).

Этот критерий оценивает степень покрытия, принимая во внимание требования Заказчика или системные спецификации. Основой может быть, например, таблица требований, use case модель и диаграмма состояний-переходов. Набор тестов должен покрывать все или конкретно определенные функциональные требования. На практике это чаще всего реализуется следующим образом: Заказчик (или системный аналитик) составляет набор требований, которые могут быть переведены в тестовые сценарии. После чего эти сценарии могут быть проверены на правильность и полноту.

Таким образом, данный критерий показывает в процентном отношении количество покрытых тестами требований. Чаще всего

данный критерий используется при тестировании методом «черного ящика».

Покрытие, основанное на коде (Code-Based Coverage), имеет отношение к потоку управления и потоку данных программы. Чаще всего данный критерий используется при тестировании методом «белого ящика». Основные критерии покрытия тестирования кода следующие:

- Покрытие строк (Line Coverage) – мера измерения покрытия кода, указывающая процентное отношение строк программы, затронутых тестами, к общему числу строк. Это очень неточная метрика, потому что даже стопроцентное покрытие по ней пропускает много ошибок.
- Покрытие ветвей (Branch Coverage). Это мера измерения покрытия кода указывает в процентном отношении, сколько ветвей потока управления было протестировано во время теста. Она надежнее предыдущей метрики, но снова стопроцентное покрытие не гарантирует отсутствие ошибок.
- Покрытие путей (Path Coverage). Эта единица измерения характеризует процент всевозможных путей (и/или комбинаций ветвей), которые покрываются тестами. Однако, даже не смотря на 100-процентное покрытие (достичь которого практически нереально в коммерческих системах), все еще могут присутствовать скрытые ошибки.

Метрики и критерии тестирования определяются в стратегии тестирования наряду с остальными составляющими процесса.

### 3.4.6 Основные технологии и методы тестирования

Технологий тестирования существует целое множество. Условно их можно отнести к статическим или к динамическим.

Необходимо разобраться в том, что же такое динамическое тестирование, а что такое статическое, и какие технологии они используют.

**Статическое тестирование** – это процесс, который обычно ассоциируют с анализом ПО. Статическим тестированием пользуются для верификации практически любого артефакта разработки: программного кода компонент, требований, системных спецификаций, функциональных спецификаций, документов проектирования и архитектуры программных систем и их компонентов, и т. д. Использование статических методов тестирования – один из наиболее эффективных способов обнаружения дефектов на ранних стадиях разработки ПО. Действительно, статическое тестирование – это единственный способ тестирования без запуска программного кода приложения.

**Динамическое тестирование** – процесс тестирования, производимый над работающей системой или подсистемой. Оно не может быть осуществлено без запуска программного кода приложения. Если быть более точным, динамическое тестирование состоит из:

1) запуска системы или подсистемы;

2) вызова необходимых функциональных элементов или модулей;

3) сравнения через графический интерфейс пользователя поведения системы с ожидаемым результатом поведения.

Технологии тестирования используются при применении тех или иных методов тестирования. Среди методов тестирования обычно выделяют два самых распространенных:

- метод «черного ящика» («black-box» testing);

- метод «белого ящика» («white-box» or «glass-box» testing).

Различие между тестированием «белого ящика» и «черного ящика» имеет место на любом уровне. Как может показаться на первый взгляд, тестирование внутренних компонент есть

тестирование «белого ящика». В то же время, с точки зрения разработчика, сам компонент может быть протестирован как методом «черного», так и «белого ящика».

### 3.4.7 Классификация в тестировании

Тесты существенно различаются по задачам, которые с их помощью решаются, и по используемой технике. Различие задач тестирования приводит, естественным образом, к необходимости использовать весьма разнообразные типы (виды) тестирования. Принято подразделять тестирование на виды по следующим категориям:

- по объектам (элементам) тестирования, часто разделение на виды тестов по данному критерию называют разделением тестирования на уровни;

- по глубине тестирования, то есть разделение тестовых испытаний на типы проводится в зависимости от количества времени и объема тестируемых компонент программного продукта.

Основная классификация тестов на виды производится в соответствии с традиционными показателями качества, которые проверяются с их помощью.

**Уровни тестирования**

**Модульное тестирование** (Автономное или Unit-тестирование)

Входные требования – Архитектура компонентов или модель «нижнего уровня» системы (Component Design или Low Level Design)

Объект тестирования – Разработанные компоненты

Определение: На данном уровне тестируются по отдельности небольшие элементы системы, максимально отделенные от других элементов и, в то же время, пригодные для тестирования. Такое

тестирование обычно проводится сразу же вслед за разработкой каждого из элементов и направлено на оценку соответствия функциональности каждого из компонентов спроектированной «модели компонентов».

**Комплексное тестирование** (Сборочное тестирование, integration testing или interface testing)

Входные требования – Архитектура системы или модель «верхнего уровня» системы (System Design или High Level Design)

Объект тестирования – Собранная из компонентов система или подсистема

Определение: На данном уровне тестируются объединенные элементы (компоненты или подсистемы) общей системы, чаще всего некоторая взаимодействующая между собой группа элементов.

Комплексное тестирование направлено не на проверку функционирования каждого из компонентов, а на проверку взаимодействия компонентов в соответствии с «Архитектурой системы».

Тесты данного уровня обычно проверяют все интерфейсы взаимодействия между компонентами, определенные в системной архитектуре, до тех пор, пока все компоненты не будут разработаны, отлажены и проинтегрированы друг с другом в единую систему.

**Системное тестирование** (system testing)

Входные требования – Системные спецификации (System Specification)

Объект тестирования – Разработанная система

Определение: После того как система собрана из составляющих компонентов, она должна быть протестирована на соответствие

«Системным спецификациям» – реализованы ли все функциональные и нефункциональные требования к разрабатываемой системе.

На данном уровне тестируется приложение или система (одно или более приложений) целиком.

**Приемочное тестирование** (Приемо-сдаточное тестирование или acceptance testing)

Входные требования – Требования (Requirements)

Объект тестирования – Разработанная система

Определение: На данном уровне завершенное приложение (система) тестируется Заказчиком, конечными пользователями или соответствующими уполномоченными с целью определения соответствия системы «Требованиям Заказчика» и готовности системы к внедрению. Приемосдаточные испытания оформляют процесс передачи продукта от Разработчика Заказчику. В зависимости от особенностей продукта и от требований Заказчика они могут проводиться в различной форме. Например, в виде альфа- или бета-тестирования.

Приемочное тестирование схоже с системным тестированием, но со следующим различием:

- системное тестирование проверяет, что разработанная система соответствует специфицированным требованиям;
- приемочное тестирование проверяет, что разработанная система удовлетворяет запрошенным Заказчиком требованиям с упором на нужды конечных пользователей в данной предметной области.

**Операционное тестирование** (Release Testing)

Входные требования – Бизнес-модель (Business Case или Business Model)

Объект тестирования – Разработанная система

Определение: Даже если система удовлетворяет всем требованиям, важно убедиться в том, что она удовлетворяет нуждам

пользователя и выполняет свою роль в среде своей эксплуатации, как это было определено в бизнес-модели системы. Следует учесть, что и Бизнес-модель может содержать ошибки. Поэтому так важно провести операционное тестирование как финальный шаг валидации.

Кроме этого, тестирование в среде эксплуатации позволяет выявить и нефункциональные проблемы, такие как: конфликт с другими системами, смежными в области бизнеса или в программных и электронных окружениях; недостаточная производительность системы в среде эксплуатации и др.

Очевидно, что нахождение подобных вещей на стадии внедрения – критичная и дорогостоящая проблема. Поэтому так важно проведение не только верификации, но и валидации, с самых ранних этапов разработки ПО.

Основное разделение тестов на виды по объектам тестирования, или, точнее, на уровни тестирования, было произведено нами при определении обобщенной модели ЖЦ ТП. Уровни тестирования приведены ниже. Для каждого уровня тестирования могут использоваться различные виды тестирования, для каждого из которых, в свою очередь, могут использоваться различные типы тестовых испытаний.

**Виды тестирования**

**Инсталляционное тестирование** (Installation testing)

Определение: В процессе инсталляционного тестирования проверяется корректность установки и деинсталляции программного продукта в среде, максимально приближенной к эксплуатационной. Проверка правильности установки программного продукта должна быть обязательным элементом проекта по тестированию любого продукта.

Цель: Основная цель состоит в том, чтобы убедиться, что продукт может быть установлен/деинсталлирован при различных

условиях – таких как: новая инсталляция, усовершенствование системы (upgrade), установка по умолчанию, полная установка, установка по выбору.

**Дымное тестирование** (проверка на дым, Smoke testing)

Определение: Первый прогон программы (после написания или после внесения существенных изменений). Как правило, используется для определения, готова ли программа для проведения более обширного тестирования.

Цель: Выявление проблем, «лежащих на поверхности», – тестируется чаще всего основная бизнес-логика программы

**Функциональное тестирование** (Functional testing)

Определение: Проверка соответствия продукта функциональным требованиям и спецификациям

Цель: Проверка соответствия продукта функциональным требованиям и спецификациям

**Регрессионное тестирование** (Regression testing)

Определение: Повторное тестирование после внесения изменений в программное обеспечение или в его окружение (в новой версии приложения), чтобы убедиться в том, что функции, которые работали в предыдущей версии системы, по-прежнему работают так, как ожидалось, а найденные дефекты успешно исправлены (все протестированное ранее тестируется повторно)

Цель: Выявление потенциальных проблем, которые могли возникнуть в результате изменений. Проверка исправления найденных ранее дефектов.

**Интеграционное тестирование** (Integration testing)

Определение: Проверка скомбинированных компонентов прикладной программы с целью определения корректности их совместного функционирования

Цель: Выявление потенциальных проблем в совместном функционировании компонент

**Тестирование графического интерфейса пользователя** (User Interface testing)

Определение: Тестирование интерфейса – экранов, кнопок и т. д. Большая часть функциональности ПО реализуется, как правило, через пользовательский интерфейс.

Цель: Обнаружение ошибок в интерфейсе и поиск ошибок в функциональности посредством интерфейса

**Тестирование производительности** (Performance testing)

Определение: Проверка скорости работы системы (время отклика, частота транзакций и другие зависящие от времени) в имитационной и реальной средах

Цель: Установить реальную производительность программного продукта

**Нагрузочное тестирование** (Load testing)

Определение: Это те же тесты производительности, при которых система подвергается различным нагрузкам; при этом цель этого тестирования – оценить способность системы правильно функционировать при некотором превышении планируемых нагрузок при реальной эксплуатации (система имеет некоторый «запас прочности»)

Цель: Убедиться в том, что система работает соответственно ожидаемым рабочим нагрузочным параметрам (какой предел работоспособности)

**Стресс тестирование** (Stress testing)

Определение: Является одним из разновидностей тестирования на производительность. Проверяется поведение системы при недостатке ресурсов (дискового пространства, обрывов сети и т. д.).

Цель: Проверка того, что система адекватно реагирует на те или иные стрессовые ситуации

**Конфигурационное тестирование** (Configuration testing)

Определение: Конфигурационное тестирование – тестирование работы на различных платформах. Различные варианты аппаратной конфигурации, версии операционной системы и окружения.

Цель: Проверить работоспособность системы при различных конфигурациях

**Тестирование интернационализации** (Internationalization testing)

Определение: Этот вид тестирует, насколько продукт готов к тому, чтобы быть адаптированном для работы в других локалях с другим языком пользовательского интерфейса, отличном от языка по умолчанию (как правило, это английский)

Цель: Проверить способность продукта быть быстро локализованным под необходимую локаль потенциальных пользователей системы

**Локализационное тестирование** (Localization testing)

Определение: Локализационное тестирование, в свою очередь, проверяет, правильно ли локализован продукт. То есть переведен на другой язык и корректно работает с учетом национальных особенностией страны или региона, в котором будет продаваться и использоваться продукт.

Цель: Проверить, правильно ли локализован продукт

Классификация тестов на виды производится в соответствии с традиционными показателями качества, которые проверяются с их помощью. Иными словами, разделение тестирования на виды происходит в зависимости от типа требований (функциональные, нефункциональные), проверяемых с помощью тестов.

Для проверки функциональности (functionality) ПО необходимо испытать приложение на выполнение функциональных требований к нему (сценариев использования и др.). Для этого используются собственно функциональные тесты, а также тесты безопасности, объема и другие.

**Тестирование надежности** (reliability) ПО производится с целью проверки нефункциональных требований, что приложение работает, как и ожидалось, устойчиво к падениям и т. п. Здесь применяются интеграционные тесты, тесты структуры, стрессовые тесты и другие.

Тестирование удобства использования (usability) ПО (нефункциональные требования) производится с целью удостовериться в том, что приложение удобно для использования его конечным пользователям. Включает в себя тесты на человеческий фактор, эстетику интерфейса и его непротиворечивость, наличие и качество оперативной и контекстной помощи, руководств и учебных материалов.

**Тестирование производительности** (performance) ПО выполняется с целью удостовериться, что функционирование приложения обеспечивается в то время, когда выполняются нефункциональные требования к приложению по работе в реальных условиях. Включает в себя оценку временных профилей, времени отклика, операционной надежности и некоторых других характеристик. Основные виды тестирования приведены ниже.

**Типы тестовых испытаний по глубине тестирования**

**Приемочный тест** (Smoke test) – первый и самый короткий тест, призванный проводить проверку основных элементов программного продукта и его работоспособности в целом. В случае функционального тестирования – проверяется основной функционал приложения. Тест занимает 1–4 часа в зависимости от сложности

тестируемого продукта. На основе результатов данного теста принимается решение о приемке версии программного продукта и продолжении тестирования текущей версии продукта более серьезными тестовыми испытаниями.

**Критический тест** (Critical path test) – основной тип тестовых испытаний, во время которого значимые элементы и функции приложения проверяются на предмет правильности работы при стандартном их использовании. Как правило, на данном уровне тестирования проверяется основная масса требований к продукту.

**Расширенный тест** (Extended test) – вид углубленного тестирования, при котором проверяется нестандартное использование программного продукта, границы переполнения массивов данных, ввод специальных символов и т. п.

## Список контрольных вопросов

1. Методология SWEBOK.
2. Методология Custom Development Method (CDM).
3. Методология Rational Unified Process (RUP).
4. Методология Microsoft Solutions Framework (MSF).
5. Понятие тестирования ПО.
6. Виды тестов.
7. Понятие надежности ПО.
8. Регрессионное тестирование.
9. Модульное тестирование.
10. Интеграционное тестирование.
11. Ручное тестирование.
12. Системное тестирование.
13. Каковы основные критерии отбора тестов?
14. Метрики и методики оценки качества тестирования.

# Заключение

Данное пособие предназначено для формирования у студентов знаний о последних практических предложениях в разработке программного обеспечения. Представленный материал степени содержит ключевые позиции по обеспечению современного процесса конструирования.

В пособии отражаются не только общие теоретические сведения, но и дополнительно приведены ряд практических приемов и подходов к конструированию программного обеспечения, которые имеют широкое применение в настоящее время.

Более подробную информацию по рассматриваемым разделам можно найти в источниках из библиографического списка.

## Основные использованные понятия (глоссарий)

Модуль – фрагмент программного текста, являющийся строительным блоком для физической структуры системы. Как правило, модуль состоит из интерфейсной части и части-реализации.

Модульность – свойство системы, которая может подвергаться декомпозиции на ряд внутренне связанных и слабо зависящих друг от друга модулей.

Покер планирования (от англ. *planning poker*, а также от англ. *scrum poker*) – техника оценки, основанная на достижении договоренности, главным образом используемая для оценки сложности предстоящей работы или относительного объема решаемых задач при разработке программного обеспечения. Это разновидность метода Wideband Delphi.

Связность модуля – мера зависимости его частей.

Сопровождение – внесение изменений в эксплуатируемое ПО.

Спиральная модель – модель, которая базируется на лучших свойствах классического жизненного цикла и макетирования, к которым добавляется новый элемент – анализ риска, отсутствующий в этих парадигмах.

Сцепление (*Coupling*) – мера взаимозависимости модулей по данным.

Тестирование – наблюдение за функционированием ПО в специфических условиях с целью определения степени соответствия ПО требованиям к нему.

Технология конструирования программного обеспечения (ТКПО) – система инженерных принципов для создания экономичного ПО, которое надежно и эффективно работает в реальных компьютерах.

Экстремальное программирование (*Extreme Programming XP*) – дисциплина разработки программного обеспечения и ведения бизнеса в области создания программных продуктов, которая фокусирует усилия обеих сторон (программистов и бизнесменов) на общих, вполне достижимых целях.

PERT (от англ. *Program Evaluation and Review Technique* – метод оценки и обзора программы) – некая технология оценки и пересмотра программы, которая базируется на идее сетевого планирования.

## Сокращения

ЖЦ – жизненный цикл;

ИС – информационные технологии;

КИС – корпоративная информационная система;

НЗР – незавершенная работа;

ПО – программное обеспечение;

ПС – программные средства;

СОА – сервисно-ориентированная архитектура;

СС – сила связности модулей;

СЦ – степень сцепления;

ТКПО – технология конструирования программного обеспечения;

ТП – технический проект;

ЭВМ – электронно-вычислительная машина.

## Библиографический список


1. Гагарина, Л. Г. Технология разработки программного обеспечения : учебное пособие для вузов / Л. Г. Гагарина. – Москва : ФОРУМ: ИНФРА-М, 2008,2011. – 399 с. – (Высшее образование).
2. Соммервилл, И. Инженерия программного обеспечения / И. Соммервилл ; пер. с англ. – 6-е изд. – Москва : Вильямс, 2002. – 623 с.
3. Разработка программного обеспечения : пер. с англ. – Санкт-Петербург : Питер, 2004. – 592 с. – (Классика computer science»).
4. Отладка и тестирование приложений в среде Visual Studio 2005 : учебное пособие / сост. О. Н.Евсеева, А. Б. Шамшев. – Ульяновск : УлГТУ, 2008. – 88 с.
5. Beizer, B. Software Testing Techniques / B. Beizer. – ITP, 1990. – 550 pp.
6. Boehm, B. Software Engineering Economic Prentice-Hall, Inc. / B. Beizer. – N. J., 1981. – 767 pp.
7. Макгрегор, Дж. Тестирование объектно-ориентированного программного обеспечения / Дж. Макгрегор. – Киев : Диасофт, 2002. – 432 с.
8. Брукс, Ф. Мифический человеко-месяц, или Как создаются программные системы / Ф. Брукс. – Санкт-Петербург : Символ-Плюс, 1999. – 304 с.
9. Липаев, В. В. Тестирование программ / В. В. Липаев. – Москва : Радио и связь, 1986. – 296 с.
10. Канер, С. Тестирование программного обеспечения / С. Канер. – Киев : ДиаСофт, 2000. – 544 с.
11. IEEE Software Engineering Standards Collection 1997 Edition.

12. IEEE Standard Glossary of Software Engineering Technology IEEE Std 610.12-1990.
13. Шимаров, В. А. Тестирование программ: цели и особенности инструментальной поддержки / В. А. Шимаров // Программное обеспечение ЭВМ / АН БССР. Институт математики. – Минск, 1994. – Вып. 100. – С. 19–43.
14. Разработка по гибким методологиям [Электронный ресурс]. – Режим доступа: http://agilerussia.ru/ (дата обращения: 01.07.2016).
15. Статьи по тестированию [Электронный ресурс]. – Режим доступа: http://software-testing.ru/ (дата обращения: 01.07.2016).
16. Базовые понятия тестирования [Электронный ресурс]. – Режим доступа: http://www.protesting.ru/ (дата обращения: 01.07.2016).
17. Разработка документации по ГОСТ [Электронный ресурс]. – Режим доступа: http://www.rugost.com/ (дата обращения: 01.07.2016).
18. Основы программной инженерии (по SWEBOK). Конструирование [Электронный ресурс]. – Режим доступа: http://swebok.sorlik.ru/3_software_construction.html (дата обращения: 01.07.2016).
19. Технологии разработки программного обеспечения : учебник / С. Орлов. – Санкт-Петербург : Питер, 2002. – 464 с.: ил. – Режим доступа: http://forcoder.ru/developing/tehnologii-razrabotki-programmnogo-obespecheniya-93 (дата обращения: 01.07.2016).