“You’ve got to be very careful if you don’t know where you’re going, because you might not get there.”

Тестировщику необходимо понимание всего процесса разработки ПО. Создание ПО вовлекает десятки и сотни членов команды, каждый из которых играет свою роль и работают вместе. Особенности того, что эти люди делают, как они взамодействуют, и как они принимают решения это все влияет на процесс разработки ПО. Знания о процессе разработки ПО помогут вам лучше понимать, как наилучшим образом применять знания тестирования.

Процесс создания ПО от его идеи до релиза называется **software development life cycle model.**

**Waterfall model**

Названа так потому, что процесс разработки организован систематически переходами от одной фазы к другой как водопад. Наиболее распространенными фазами являются следющие:

* Definition Study/Analysis
* Requirements
* Construction
* Testing
* Deployment
* Support

До этой модели процесс разработки представлял собой беспорядочный набор (scope) задач. Этот метод позволил внести упорядоченность в проекты. По этому методу проект разделяется на несколько фаз, таким образом охватывается весь процесс. Например, вы начинаете с фазы 1, и переход к следующей фазе начинается только после завершения предыдущей. Таким образом, достигнув следующей фазы, вы не можете вернуться назад.

* **Definition Study/Analysis:**В течении этой фазы происходит мозговой штурмо ПО: что приложение должно делать, какие цели должны быть достигнуты, продумывается план разработки.
* **Requirements:** Следующий шаг состоит в разработке требований к приложению и разработке базовой конструкции программного обеспечения, продумывается архитектура.

Далее производится более подробное техническое проектирование.

* **Construction / Implementation:**В этой фазе пишется код.
* **Testing:**На этом этапе весь дизайн и конструкция тестируется для проверки его функциональности. Если есть ошибки, то они выйдут на поверхность в этой точке процесса.
* **Deployment:**в фазе внедрения, приложение отдают заказчику, после того, как система была успешно протестирована.
* **Support:**Техническое обслуживание и управление необходимо, чтобы убедиться, что система будет продолжать работать правильно.

**Приемущества:**

* Easy to understand, easy to use
* Provides structure to inexperienced staff
* Milestones are well understood
* Sets requirements stability
* Good for management control
* Works well when quality is more important than cost or schedule

**Недостатки:**

* All requirements must be known upfront
* Integration is one big band at the end
* Little opportunity to customer to preview the system
* Does not maintain iterations
* Testing is performed of the final phases (Тк тестирование происходит лишь в конце, основная проблема может заключаться в ошибках на начальных этапах, а также непосредственно перед релизом. Но стоимость таких ошибок, как известно, будет весомой.)

**Когда лучше использовать данную модель:**

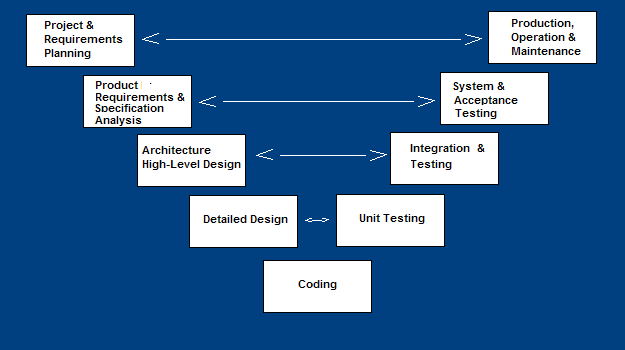
* Требования очень хорошо известны
* Продукт стабилен
* Технология ясна и понятна

Есть модели, в которых возможен переход на предыдущий шаг.

На первый взгляд, такая модель кажется очень ограниченной. Но она работает для тех проектов, где хорошо понятно определение продукта, его назначение, а также где высококвалифицированная и сплоченная команда. Поэтому целью является выяснение всех деталей до того, как будет написана первая строчка кода.

С точки зрения тестирования, waterfall модель имеет большие выгоды по сравнению с другими моделями. Все тщательно и внимательно проверено. К тому времени, как ПО будет передано команде тестировщиков, каждая деталь будет выяснена и отражена в коде. Группа тестировщиков может создать четкий план и расписание.

**V-Shaped SDLC Model**



Это разновидность модели водопад. Стадии тестировнаия соотностятся с соответствующими стадиями разработки. В результате поиск дефектов происходит на более ранних стадиях процесса разработки. Это позволяет уменьшить затраты на поиск и управление дефектов. Горизонтальные пунктирные линии определяют отношение между видом тестовой деятельности и видом проектной деятельности. Например, верхняя пунктирная линия со стрелками показывает, что цель приемочного тестирования заключается в подтверждении требований, а само приемочное тестирование основано на требованиях.

**Приемущества**:

Особое значение придается планированию, направленному на верификацию и аттестацию разрабатываемого продукта на ранних стадиях его разработки.

В модели предусмотрены аттестация и верификация всех внешних и внутренних полученных данных, а не только самого программного продукта.

Благодаря модели менеджеры проекта могут отслеживать ход процесса разработки, так как в данном случае вполне возможно воспользоваться временной шкалой, а завершение каждой фазы является контрольной точкой.

Проста в использовании.

**Недостатки**:

Модель не предусматривает работу с параллельными событиями.

Невозможно управлять итерацией или фазой

В модели не предусмотрено внесение требования динамических изменений на разных этапах жизненного цикла.

В модель не входят действия, направленные на анализ рисков.

**Prototyping Model**

Основная идея: вместо замораживания требований перед стадией разработки архитектуры или кодирования строится прототип программы, чтобы наилучшим образом понять требования. Создание прототипа также подразумевает создание архитектуры, кодирования и тестирования. Но каждый из этих этапов делается очень поверхностно. Используя этот прототип, клиент может получить «фактическое ощущение» системы. С прототипом он может лучше понять требования желаемой системы.

Основная причина того, что данная модель не часто используется – расходы на написание системы дважды. Но есть утверждения, что прототип не должен быть очень дорогостоящим, и может реально снизить общую стоимость разработки.

**Приемущества.**

Пользователь активно участвует в разработке. У него появляется лучшее понимание системы.

Он может изменять свое мнение о требованиях системы.

Ошибки могут обнаруживаться значительно раньше.

Из-за наличия быстрой обратной связи возможны лучшие реализации системы.

**Недостатки.**

Приводит к переделываению системы из-за изменения способа реализации.

Практически, эта методика может повысить сложность системы, так как область применения системы может выйти за рамки первоначальных планов.

**Spiral Model**

Делает упор на начальные этапы жизненного цикла: **анализ и проектирование**. Отличительной особенностью этой модели является специальное внимание рискам, влияющим на организацию жизненного цикла. Каждый виток спирали соответствует созданию фрагмента или версии программного обеспечения, на нем уточняются цели и характеристики проекта, определяется его качество и планируются работы следующего витка спирали. Таким образом углубляются и последовательно конкретизируются детали проекта и в результате выбирается обоснованный вариант, который доводится до реализации. Каждый виток разбит на 4 сектора:

* оценка и разрешение рисков,
* определение целей,
* разработка и тестирование,
* планирование.

На каждом витке спирали могут применяться разные модели процесса разработки ПО. В конечном итоге на выходе получается готовый продукт. Неполное завершение работ на каждом этапе позволяет переходить на следующий этап, не дожидаясь полного завершения работы на текущем. Главная задача — как можно быстрее показать пользователям системы работоспособный продукт, тем самым активизируя процесс уточнения и дополнения требований. Переход осуществляется в соответствии с планом, даже если не вся запланированная работа закончена. План составляется на основе статистических данных, полученных в предыдущих проектах, и личного опыта разработчиков.

Выводы: Спиральная модель Боэма сфокусирована на проектировании. Собственно разработка ПО происходит лишь на последнем витке спирали по обычной каскадной модели, однако этому предшествует несколько итераций проектирования на основе создания прототипов – при этом каждая итерация включает стадию выявления и анализа рисков и наиболее сложных задач.

Поскольку спиральная модель в основном охватывает именно проектирование, то в первоначальном виде она не получила широкого распространения в качестве метода управления всем жизненным циклом создания ПО. Однако главная ее идея, заключающаяся в том, что процесс работы над проектом может состоять из циклов, проходящих одни и те же этапы, послужила исходным пунктом для дальнейших исследований и стала основой большинства современных моделей процесса разработки ПО.

**Agile**

Manifesto:

* **Individuals** and **interactions** over process and tools
* **Working Software** over comprehensive documentation
* **Customer collaboration** over contract
* **Responding to change** over following a plan

Principles of Agile:

* **Customer satisfaction** by rapid (быстрый, скоростной), continuous delivery of useful software;
* **Working Software** is delivered frequently (weeks rather than month)
* Working software is the principal **measure of progress**
* Even late changes in requirements are welcomed
* Close, daily cooperation between people and developers
* **Face-to-face** conversation is the best form of communication
* Projects are built are built around motivated **individuals,** who should be trusted
* Continuous attention to **technical excellence** and good design
* **Simplicity**
* **Self-organizing** teams
* Regular **adaptation** to changing circumstances

<http://www.slideshare.net/cspag67/introduction-to-agile-practices>

**RUP**

Один из самых известных процессов, использующих итеративную модель разработки – RationalUnifiedProcess (RUP). Он был создан во второй половине 1990-х годов в компании RationalSoftware. Основными разработчиками были Филипп Крачтен (PhilippeKruchten), Грейди Буч (GradyBooch), Джеймс Рамбо (JamesRumbaugh) и Айвар Якобсон (IvarJacobson). Кстати, последние трое являются также создателями нотации UML.

Методология RUP описывает абстрактный общий процесс, на основе которого организация или проектная команда должна создать специализированный процесс, ориентированный на ее потребности.

**Жизненный цикл проекта**

Жизненный цикл проекта RUP состоит из четырех фаз. Последовательность этих фаз фиксирована, но число итераций, необходимых для завершения каждой фазы, определяется индивидуально для каждого конкретного проекта.

**Начало (Inception)**

Фаза Начало обычно состоит из одной итерации. В ходе выполнения этой фазы необходимо:

* Определить видение и границы проекта.
* Создать экономическое обоснование (businesscase).
* Идентифицировать большую часть прецедентов использования и подробно описать несколько ключевых прецедентов.
* Найти хотя бы одно возможное архитектурное решение.
* Оценить бюджет, график и риски проекта.

Если после завершения первой итерации заинтересованные лица приходят к выводу о целесообразности выполнения проекта, проект переходит в следующую фазу. В противном случае проект может быть отменен или проведена еще одна итерация фазы Начало.

**Проектирование (Elaboration)**

В результате выполнения этой фазы на основе требований и рисков проекта создается основа архитектуры системы. Проектирование может занимать до двух-трех итераций или быть полностью пропущенным (если в проекте используется архитектура существующей системы без изменений). Цели:

* Детальное описание большей части прецедентов использования.
* Создание оттестированной (при помощи архитектурно значимых прецедентов использования) базовой архитектуры.
* Снижение основных рисков и уточнение бюджета и графика проекта.

В отличие от модели водопада, основным результатом этой фазы является не множество документов со спецификациями, а действующая система с 20-30% реализованных прецедентов использования.

**Построение (Construction)**

В этой фазе (длящейся от двух до четырех итераций) происходит разработка окончательного продукта. Во время ее выполнения создается основная часть исходного кода системы и выпускаются промежуточные демонстрационные прототипы.

**Внедрение (Transition)**

Целями фазы Внедрения являются проведение бета-тестирования и тренингов пользователей, исправление обнаруженных дефектов, развертывание системы на рабочей площадке, при необходимости – миграция данных. Кроме того, на этой фазе выполняются задачи, необходимые для проведения маркетинга и продаж.

Фаза внедрения занимает от одной до трех итераций. После ее завершения проводится анализ результатов выполнения всего проекта: что можно изменить для улучшения эффективности в будущих проектах?

**Рабочий процесс**

В терминах RUP участники проектной команды создают так называемые артефакты (workproducts), выполняя задачи (tasks) в рамках определенных ролей (roles). Артефактами являются спецификации, модели, исходный код и т.п. Задачи разделяются по девяти процессным областям, называемым дисциплинами (discipline). В RUP определены шесть инженерных и три вспомогательные дисциплины. В них входят:

* Бизнес-моделирование (BusinessModeling) – исследование и описание существующих бизнес-процессов заказчика, а также поиск их возможных улучшений.
* Управление требованиями (RequirementsManagement) – определение границ проекта, разработка функционального дизайна будущей системы и его согласование с заказчиком.
* Анализ и проектирование (AnalysisandDesign) – проектирование архитектуры системы на основе функциональных требований и ее развитие на протяжении всего проекта.
* Реализация (Implementation) – разработка, юнит-тестирование и интеграция компонентов системы.
* Тестирование (Test) – поиск и отслеживание дефектов в системе, проверка корректности реализации требований.
* Развертывание (Deployment) – создание дистрибутива, установка системы, обучение пользователей.
* Управление конфигурациями и изменениями (Configuration and Change Management) – управление версиями исходного кода и документации, процесс обработки запросов на изменение (change requests).
* Управление проектом (Project Management) – создание проектной команды, планирование фаз и итераций, управление бюджетом и рисками.
* Среда (Environment) – создание инфраструктуры для выполнения проекта, включая организацию и настройку процесса разработки.

В ходе жизненного цикла проекта распределение усилий проектной команды между дисциплинами постоянно меняется. Например, как правило, в начале проекта большая часть усилий затрачивается на анализ и дизайн, а ближе к завершению – на реализацию и тестирование системы. Однако в общем случае задачи из всех девяти дисциплин выполняются параллельно.

**Разработка требований**

Для описания требований в RUP используются прецеденты использования (usecases). Это не слишком удивительно, учитывая, что один из создателей RUP, Айвар Якобсон, является также автором концепции прецедента использования. Полный набор прецедентов использования системы вместе с логическими отношениями между ними (прецеденты могут включать и расширять другие прецеденты) называется моделью прецедентов использования.

Каждый прецедент использования – это описание сценария взаимодействия пользователя с системой, полностью выполняющего конкретную пользовательскую задачу. Разумеется, не имеет смысла документировать в виде прецедентов нефункциональные требования (к производительности, качеству т.д.). Однако согласно RUP все функциональные требования должны быть представлены в виде прецедентов использования. Считается, что модель прецедентов дает более целостное представление о функциональности системы по сравнению с традиционным описанием требований (перечислением функций, которыми должна обладать система).

**Итеративная разработка**

Проект в RUP состоит из последовательности итераций с рекомендованной продолжительностью от 2 до 6 недель. Основной единицей планирования итераций является прецедент использования. Перед началом очередной итерации определяется набор прецедентов использования, которые будут реализованы к ее завершению.

Итеративная модель, подробно описанная выше, позволяет вносить необходимые изменения в требования, проектные решения и реализацию в ходе проекта.

**Архитектура**

Можно сказать что RUP – ориентированная на архитектуру методология. Считается, что реализация и тестирование архитектуры системы должны начинаться на самых ранних стадиях проекта. RUP использует понятие исполняемой архитектуры (executablearchitecture) – основы приложения, позволяющей реализовать архитектурно значимые прецеденты использования. Основы исполняемой архитектуры должны быть реализованы как можно раньше. Это позволяет оценить адекватность принятых архитектурных решений и внести необходимые коррективы еще в начале проекта. Таким образом, для первых нескольких итераций необходимо выбирать прецеденты, которые требуют реализации большей части архитектурных компонентов.

RUP поощряет использование визуальных средств для анализа и проектирования. Как правило, используется нотация и, соответственно, средства моделирования UML (такие как RationalRose). Модель предметной области документируется в виде диаграммы классов, модель прецедентов использования – при помощи диаграммы прецедентов, взаимодействие компонентов системы между собой описывается диаграммой последовательности и т д.

**Scrum**

Продукт описывается списком фич: the **backlog**.

Фичи описаны с помощью **user stories**.

The scrum team **estimates** the **work** associated with each story.

Фичи в backlog ранжированы по порядку важности.

В результате: отранжированный список фич продукта - **roadmap**.

Product owner **owns** the product backlog.

Roles:

**The scrum team:** представляет собой команду тех людей, разрабатывают и внедряют решение тех или иных проблем. Обычно команда состоит из 5-9 человек. Это оптимальный размер для скрам команды. Члены команды решают, как настроить работу и как работа будет распределяться между ее членами. Нет специально установленных ролей для членов команды – любой ее член может поменяться такой с другим членом команды. Естественно, это не препятствует отдельным членам команды быть экспертами в их области.

**ProductOwner:** он представляет интересы заказчика и отвечает за то, чтоcточки зрения бизнес перспективы ScrumTeam работает правильно. ProductOwner управляет Product Backlog-ом, который виден любому из членов команды, так что все осведомлены что ожидается в будущем релизе продукта. просмотривает все changes, которые запланированы для продукта и расставляет приоритеты для всех функциональностей. Результатом работы **ProductOwner**-а является **ProductBacklog** – список “to-do” фич, который строго приоритезирован. Пееред каждой итерацией, наиболее важные фичи перемещаются в **SprintBacklog**. Часто Product Owner является заказчиком, но он также может быть члено внутренней организации. Его задачи требуют глубоких знаний в энжинеринге, маркетинге и бизнес процессах.

**ScrumMaster:** тренерует команду разработчиков, устраняет всевозможные препятствия и работает постоянно над тем, чтобы команда имела наилучшие условия для достижения целей данного спринта. Он встречается с командой каждый день на митингах, DailyScrum. После каждого спринта ScrumMaster проводит EvaluationMeeting со скрам-командой – **Sprint Retrospective**, во время которых просматривается итоги сделанной работы. Целью является также улучшить уровень знаний команды и повысить мотивацию к следующей итерации.

Things:

**ProductBacklog –** главный список всей желаемой функциональности продукта с назначенными приоритетами.

**Stories –** краткое описание функциональности с точки зрения пользователя, ценной для использователя или команды заказчика. Истории традиционно пишутся на карточках, обычно содержит однострочное описание средства. Пример: «Как покупатель, я могу помещать элементы в совю корзину покупок, чтобы приобрести их».

Scrum Sprint Cycle

Каждый спринт усиливает (увеличивает) значимость, ценность продукта и добавляет больше функциональностей и совершенствует то, что будет доставлено пользователю.

**Creating a Backlog:** ProductOwner собирает все реквесты и спецификации, на которых основываются изменения в продукте. Этими изменениями могут быть новые функции, а также фиксы багов. После того, как цели будут опеределены, все задачи делятся на сегменты. Каждый сегмент должен иметь собственную ценность и должен иметь возможность быть самостоятельно доставленным для пользователя. Далее Продакт Оунер составяет список приоритетов: в каком порядке измененеия должны быть выполнены и доставлены. В результате получается список фич, которые необходимо сделать, в порядке того, насколько они востребованы заказчиком и насколько этого требует рынок. Когда приходит время начинать новый спринт, Продакт Оунер «замораживает» самые важные фичи и на митинге их предостваляет.

**The Sprint Phase:** длится 30 календарных дней, первые из которых используется для того, чтобы создать Sprint Backlog. Таски и необходимое время определены и дальше Скрам Команда работает в соответствии своих обязанностей. Если группа была правильно составлена, то она будет сомоорганизованной.

**DailyScrum:** каждый день, в одно и то же время ScrumMaster и ScrumTeam проводят митинги. Их целью является опеределить все препятствия, которые тормозят процесс. Каждый из приглашенных должен ответить на вопрос:

- что было сделано со времени прошлого митинга?

- что будет делаться на протяжении времени между настоящим и следующим митингом?

- есть ли что-то, что мешает делать то, что вами было запланировано.

Перые 2 вопроса дают возможность сформировать мнение о прогрессе проекта. Третий вопрос дает основание для решения проблем – начиная с того, кому надо заменить мышку и заканчивая организаторскими изменениями в компании. Любой может посещать митинги, но только Скрам Мастер и члены команды имеют правл говорить.

**Demonstration And Evaluation:** каждый спринт заканчивается демонстрацией, во время которой работающее приложение запускается большой группой (кроме Продакт Оунера) пользователей. Этот запуск является начальным блоком для следующего Спринта.

**Burn-down chart** используется для того, чтобы оценить остаток работы, которую необходимо выполнить до внедрения части приложения. Диаграмма показывает уровень оставшихся часов до того как Спринт «burndown».

(<http://www.slideshare.net/guest035e0d/learn-scrum-engineering-in-5-minutes>)

Benefits:

* + it’s simple;
  + provide clear measures;
  + each story is estimated;
  + over time we can improve estimates and notice trends;
  + keeps team focused;
  + maintains flexibility;

**XP**

В основе экстремального программирования — очень короткий, постоянно повторяющийся цикл разработки, составляющий 1-3 недели. К концу каждого цикла должен быть полностью рабочий, функциональный и протестированный релиз приложения.

Двенадцать основных приёмов экстремального программирования:

* Короткий цикл обратной связи (Fine Scale Feedback)
  + Разработка через тестирование (Test Driven Development)
  + Игра в планирование (Planning Game)
  + Заказчик всегда рядом (Whole Team, Onsite Customer)
  + Парное программирование (Pair Programming)
* Непрерывный, а не пакетный процесс
  + Непрерывная интеграция (Continuous Integration)
  + Рефакторинг (Design Improvement, Refactoring)
  + Частые небольшие релизы (Small Releases)
* Понимание, разделяемое всеми
  + Простота (Simple Design)
  + Метафора системы (System Metaphor)
  + Коллективное владение кодом (Collective Code Ownership)
  + Стандарт кодирования (Coding Standard or Coding conventions)
* Социальная защищенность программиста (Programmer Welfare):
  + 40-часовая рабочая неделя (Sustainable Pace, 40-hour week)

**TDD**

**Test-drivendevelopment, TDD** — техника [разработки программного обеспечения](http://ru.wikipedia.org/wiki/%D0%A0%D0%B0%D0%B7%D1%80%D0%B0%D0%B1%D0%BE%D1%82%D0%BA%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F), которая основывается на повторении очень коротких циклов разработки: сначала пишется тест, покрывающий желаемое изменение, затем пишется код, который позволит пройти тест и под конец проводится рефакторинг нового кода к соответствующим стандартам.

В XP особое внимание уделяется двум разновидностям тестирования:

* Тестирование модулей (unit testing);
* Приемочное тестирование (acceptance testing).

<http://ademidov.net/articles/Extreme%20Programming%201.htm>

Отцом-идеологом экстремального программирования считают Кента Бека (Kent Beck). XP является достаточно молодой методологией, оценки которой весьма противоречивы — от восторженных до резко негативных. Основными принципами являются:

* Простота решений (simplicity).
* Интенсивная разработка малыми группами (не больше 10 человек), активное общение в группе и между группами (communication).
* Обратная связь с клиентом (feedback), который фактически вовлечен в процесс разработки.
* Достаточная степень смелости (courage) и желание идти на риск.
* Планирование процесса (**Planning game**). Вся команда собирается вместе, принимается коллективное решение о том, какие свойства системы будут реализованы в ближайшей итерации. Набор свойств определяется пользовательскими историями. Трудоемкость каждого свойства определяется самими программистами.
* Тесное взаимодействие с заказчиком (**feed-back**, **on-site customer**). Заказчик должен быть членом XP-команды (on-site customer). Он пишет пользовательские истории, выбирает истории, которые будут реализованы в конкретной итерации, и отвечает на вопросы, касающиеся бизнеса. Заказчик должен быть экспертом в автоматизируемой предметной области. Необходимо постоянное наличие обратной связи с заказчиком (feed-back).
* Парное программирование (**pair programming**) — одна из самых известных XP-практик. Все программисты должны работать в парах: один пишет код, другой смотрит. Таким образом, необходимо размещать группу программистов в одном месте, что легче всего сделать на территории заказчика (все необходимые члены команды географически находятся в одном месте); XP наиболее успешно работает в нераспределенных коллективах программистов и пользователей.
* Непрерывная интеграция (**continuous integration**). Интеграция новых частей системы должна происходить как можно чаще, как минимум раз в несколько часов. Основное правило интеграции следующее: интеграцию можно производить, если все тесты проходят успешно. Если тесты не проходят, то программист должен либо внести исправления и тогда интегрировать составные части системы, либо вообще не интегрировать их. Правило это жесткое и однозначное — если в созданной части системы имеется хотя бы одна ошибка, то интеграцию производить нельзя. Частая интеграция позволяет быстрее получить готовую систему, вместо того чтобы тратить на сборку неделю.
* Рефакторинг (**refactoring**). Рефакторинг — это оптимизация существующего кода в сторону упрощения, что предусматривает постоянную работу по упрощению кода. Сохраняя код прозрачным и определяя его элементы всего один раз, программисты сокращают число ошибок, которые впоследствии придется устранять. При реализации каждого нового свойства системы программист должен подумать над тем, можно ли упростить существующий код и как это поможет реализовать новое свойство.
* Частая смена версий (**small releases**). Минимальная итерация — один день, максимальная — месяц; чем чаще осуществляются релизы, тем больше недостатков системы будет выявлено. Первые релизы помогают выявить недостатки на самых ранних стадиях, далее функциональность системы расширяется (на основании тех же пользовательских историй). Поскольку пользователь включается в процесс разработки начиная с первого релиза, то он оценивает систему и выдает пользовательскую историю плюс feedback. На основании этого определяется следующая итерация: каким будет новый релиз. В XP все направлено на обеспечение непрерывной обратной связи с пользователями.
* Простая архитектура (**simple design**). Любое свойство системы должно быть реализовано как можно проще. Программисты в XP-команде работают под девизом: «Ничего лишнего!». Принимается первое наипростейшее работающее решение, реализуется необходимый уровень функциональности на данный момент. Тем самым экономится время программиста.
* Стандарты кодирования (**coding standard**). Стандарты кодирования нужны для обеспечения других практик: коллективного владения кодом, парного программирования и рефакторинга. Без единого стандарта выполнять эти практики как минимум сложнее, а в реальности вообще невозможно: группа будет работать в режиме постоянной нехватки времени. Детальные стандарты не требуются, необходимо стандартизировать только важные вещи. Определение наиболее важных объектов стандартизации в XP субъективно.
* Коллективное владение кодом (**collective code ownership**). Каждый программист в коллективе XP должен иметь доступ к коду любой части системы и вносить изменения в любой код. Обязательное правило: если программист внес изменения и система после этого работает некорректно, то именно этот программист должен исправить ошибки. В противном случае работа системы уподобится тотальному хаосу.

**FDD**

FDD is a model-driven short-iteration process that consists of five basic activities.FDD was first introduced to the world in 1999 via the book [**Java Modeling In Color with UML**](http://www.amazon.com/exec/obidos/ASIN/013011510X/ambysoftinc/), a combination of the software process followed by Jeff DeLuca’s (ДжефдеЛюка) company and Peter Coad’s concept of features. FDD was first applied on a 15 month, 50-person project for a large Singapore bank in 1997, which was immediately followed by a second, 18-month long 250-person project.

As the name implies, [**features**](http://www.agilemodeling.com/artifacts/feature.htm) are an important aspect of FDD.  A feature is a small, client-valued function expressed in the form

<action><result><object>

For example, “Calculate the total of a sale”,

“Validate the password of a user”, and

“Authorize the sales transaction of a customer”.

Features are to FDD as [**use cases**](http://www.agilemodeling.com/artifacts/systemUseCase.htm) are to the [**Rational Unified Process (RUP)**](http://www.agilemodeling.com/essays/agileModelingRUP.htm) and [**user stories**](http://www.agilemodeling.com/artifacts/userStory.htm) are to [**XP**](http://www.agilemodeling.com/essays/agileModelingXP.htm) – they’re a primary (исходный) source of requirements and the primary input into your planning efforts.

As you see in [**Figure 1**](http://www.agilemodeling.com/essays/fdd.htm#Figure1) there are five main activities in FDD that are performed iteratively:

**Develop An Overall Model**. В процессе участвуют:

- главный архитектор;

- ведущие разработчики;

- эксперты в предметной области;

**Подготовка к созданию**: выслушивают эксперта в предметной области, изучают документы, обсуждают между собой.

**Создание модели**:обязательно разбиваются на небольшие группы, каждая группа создает свою модель (диаграмму классов), после этого собираются все вместе, обсуждают различия, выбирают/формируют итоговык решения. В результате: диграммы классов (без детализации), комментируют, почему выбрано такое решение, альтернативные решения.

At the start of a project your goal is to identify and understand the fundamentals of the domain (области) that your system is addressing, and throughout the project you will flesh this model out to reflect what you’re building (напротяжениивсегопроекта вы плоти этоймодели из отразить то, что выстроите).

**Build а**[**Features List**](http://www.agilemodeling.com/artifacts/feature.htm#Figure1)**.** В результате: FeatureList, сгруппированыыйвнаборы (FeatureSet), разделенныйпообластям (SubjectArea).

**PlanByFeature**.

Основания для планирования: наборы функций; последовантельность итераций определяется исходя из:

- взаимоствязи функций с точки зрения реализации;

- сложности набора функций;

- различия рисков при реализации функций;

- равномерное распределение загрузки разработчиков.

В результате: график реализации наборов функций (Набор - Дата);

Ведущий программисты, закрепленные за наборами (набор - ответственный)

**DesignByFeature**

Задачи:

- формирование команд на итерацию;

- обзор затрагиваемой предметной области;

- изучение необходимых документов;

- составление диаграмм взаимодействия;

- уточнение и детализация соответствующей части диаграммы классов.

В результате:

- краткое описание набора функций, дополнительные комментарии;

- диаграммы взаимодействия;

- объектная модель, заголовки классов;

- задания (ToDo) в системе ведения дел для каждого участника команды

**BuildByFeature**:

Задачи:

- реализация классов и методов;

- Code Inspection (Code Review)

- unit-тестирование;

- integration (PromotetoBuild)

- уточнение и детализация соответствующей жиаграммы классов.

В результате:

- проинспектированные классы и методы;

- реализованная функциональность, понятная заказчику. (client-valuedfeatures).

**Замечания**:

- итерация не заканчивается, пока все взятые в нее feature не доделают, т.е следующие недоделанные feature не перетекают с следующую итерацию – их дожимают в текущей, растягивая ее.

The majority of the effort on an FDD project, roughly 85%, is comprised of the fourth and fifth steps: Design By Feature and Build By Feature. These two activities are exactly what you’d expect, they include tasks such as detailed modeling, programming, testing, and packaging of the system.

An FDD project starts by performing the first three steps in the equivalent of the RUP’s Inception phase or XP’s “iteration 0”, the goal being to identify the scope of the effort, the [**initial architecture**](http://www.agilemodeling.com/essays/initialArchitectureModeling.htm), and the initial high-level plan.  Construction efforts occur in two-week (or less) iterations, similar to XP although a little extreme for most RUP teams, with the team iteratively working through all five steps as needed.  As with other agile software development processes, systems are delivered incrementally by FDD teams.

FDD’s five steps are supported by several best practices.  The first is domain object modeling, the creation of a high-level class diagram and supporting artifacts that describes the problem domain.  Developing by feature and individual class ownership are also best practices, as is having developers work together in feature teams.  Inspections are an important aspect of FDD, as they are with the RUP.  FDD also insists on regular builds, similar to XP, and configuration management.  Finally, FDD promotes a best practice called reporting/visibility of results, similar to XP and AM’s philosophy of [open and honest communication](http://www.agilemodeling.com/principles.htm#OpenAndHonestCommunication).

How would Agile Modeling (AM) be applied on an FDD project?  The principles and practices can be clearly applied to FDD’s two modeling-oriented steps – develop an overall model and design by feature.  The only apparent mismatch between the two processes is FDD’s practice of class ownership and AM’s practice of [collective ownership](http://www.agilemodeling.com/practices.htm#CollectiveOwnership), but I would argue that this isn’t the case.  FDD’s practice pertains to coding but does not to modeling, on a FDD project people work together in teams to model, along the lines of AM’s model with others practice, and therefore several people will be working on your shared collection of modeling artifacts.

(<http://www.agilemodeling.com/essays/fdd.htm>, <http://www.slideshare.net/biBIGine/fdd-and-ddd>)

http://cppbuilder.ru/articles/0043.php