Раздел 7. Гибкие методологии

Гибкая методология разработки (от англ. - Agile software development) - манифест, определяющий способ мышления и содержащий основные ценности и принципы, на которых базируется несколько подходов (фреймворков, от англ. framework — каркас, структура) к разработке программного обеспечения, подразумевающих под собой интерактивную разработку, периодического (динамического) предоставления (обновления) требований от Заказчика и их реализацию посредством самоорганизующихся рабочих групп, сформированных из экспертов различного профиля (разработчики, тестировщики, внедренцы и т.д.). Такой перевод Agile, как "гибкая методология разработки" не совсем корректен т.к. обычно Agile не называют методологией, а вот подходы на основе данного манифеста и есть методологии, но с точки зрения Agile их называют - фреймворки. На данный момент существует множество фреймворков (методологий), подходы которых базируются на гибкой методологии разработки, например, такие, как: Scrum, Extreme programming, FDD, DSDM и т.д.

В основе гибкой методологии разработки лежит либерально-демократический подход к управлению и организации труда команд, члены которой сконцентрированы на разработке конкретного программного обеспечения.

За счет того, что разработка программного обеспечения с применением гибкой методологии определяет серии коротких циклов (итераций), с длительностью 2-3 недели, достигается минимизация рисков т.к. по завершению каждой итерации Заказчик принимает результаты и выдает новые или корректирующие требования т.е. контролирует разработку и может на неё сразу влиять. Каждая итерация включает в себя этапы планирования, анализа требований, проектирование, разработку, тестирование и документирование. Обычно одной итерации недостаточно для выпуска полноценного программного продукта, но при этом по окончании каждого этапа разработки должен появляться "осязаемый" продукт или часть функционала, которую можно посмотреть, протестировать и выдать дополнительные или корректирующие меры. На основе проделанной работы, после каждого этапа, команда подводит итоги и собирает новые требования, на основании чего вносит корректировки в план разработки программного обеспечения.

Одной из основных идей Agile, является взаимодействие внутри команды и с заказчиком лицом к лицу, что позволяет быстро принимать решения и минимизирует

риски разработки программного обеспечения, поэтому команду размещают в одном месте, с географической точки зрения. Причем в команду входит представитель заказчика (англ. product owner - полномочный представитель заказчика или сам заказчик, представляющий требования к продукту; такую роль выполняет менеджер проекта от заказчика или бизнес-аналитик).

История выпуска Agile манифеста

«Манифест гибкой методологии разработки программного обеспечения» был выпущен и принят в феврале 2001 года (штат ЮТА США, лыжный курорт The Lodge at Snowbird) группой экспертов. Данный манифест определяет 4 основные ценности и 12 принципов для методологий, базирующихся на нем, а также дает альтернативное видение подхода к разработке программного обеспечения в отличие от крупных и известных методов и методологий, но не является сам по себе методологией. Обычно Agile сравнивают в первую очередь с "методом водопада" ("waterfall"), т.к. на момент выхода манифеста, именно "метод водопада" являлся основным при планировании разработки программного обеспечения. В разработке и выпуске Agile манифеста принимали участие представители следующих методологий:

- Adaptive software development (ASD)
- Crystal Clear
- Dynamic Systems Development Method (DSDM)
- Extreme Programming (XP)
- Feature driven development (FDD)
- Pragmatic Programming
- Scrum

Собственно, данные методологии гибкой разработки существовали и до выпуска манифеста. Сам же выпуск манифеста дал новый толчок к развитию гибких методологий, заложил основы, можно сказать конституцию гибкого подхода к разработке программного обеспечения.

Agile-манифест разработки программного обеспечения:

Основной метрикой agile-методов является рабочий продукт. Отдавая предпочтение непосредственному общению, agile-методы уменьшают объём письменной документации по сравнению с другими методами.

Основополагающие принципы Agile-манифеста:

1. Наивысшим приоритетом для нас является удовлетворение потребностей заказчика, благодаря регулярной и ранней поставке ценного программного обеспечения.

2. Изменение требований приветствуется, даже на поздних стадиях разработки. Agile-процессы позволяют использовать изменения для обеспечения заказчику конкурентного преимущества.

3. Работающий продукт следует выпускать как можно чаще, с периодичностью от пары недель до пары месяцев.

4. На протяжении всего проекта разработчики и представители бизнеса должны ежедневно работать вместе.

5. Над проектом должны работать мотивированные профессионалы. Чтобы работа была сделана, создайте условия, обеспечьте поддержку и полностью доверьтесь им.

6. Непосредственное общение является наиболее практичным и эффективным способом обмена информацией как с самой командой, так и внутри команды.

7. Работающий продукт — основной показатель прогресса.

8. Инвесторы, разработчики и пользователи должны иметь возможность поддерживать постоянный ритм бесконечно. Agile помогает наладить такой устойчивый процесс разработки.

9. Постоянное внимание к техническому совершенству и качеству проектирования повышает гибкость проекта.

10. Простота — искусство минимизации лишней работы — крайне необходима.

11. Самые лучшие требования, архитектурные и технические решения рождаются у самоорганизующихся команд.

12. Команда должна систематически анализировать возможные способы улучшения эффективности и соответственно корректировать стиль своей работы.

Критика Agile

Agile плохо описывает процессы управления требованиями, можно сказать, что такое понятие просто отсутствует т.к. гибкая методология разработки не подразумевает под собой долгосрочного планирования (планирование осуществляется на краткосрочную перспективу), как следствие пропущен шаг формирования плана развития продукта или другими словами дорожной карты продукта. Т.к. планирование краткосрочное (на ближайшую итерацию разработки), а Заказчик по окончанию каждой итерации принимает продукт и выставляет новые требования, сам продукт может поменяться в корни, а

выставляемые новые требования зачастую противоречат структуре и архитектуре продукта уже поставляемого клиентам. По большому счету, в случае, если Заказчик не до конца понимает, что хочет увидеть в итоге (конечный продукт), а понимание приходит во время разработки (это случается в 90% случаев), процесс разработки превращается в формализованную и легализованную бюрократию т.е. продукт дорабатывается бесконечно, пока не кончаются деньги, или заказчик не переключается на другой продукт. Справедливости ради, необходимо заметить, что Заказчик знает на что идет и сам решает, платить за разработку продукта или нет, по большому счету команда разработчиков просто выполняет требования заказчика. Однако, реально, в работе это приводит к хаосу, срыву сроков и авралам, что порождает новые требования, которые меняют не в лучшую сторону продукт. Более того, снижается качество разрабатываемого продукта, т.к. Agile определяет подход к разработке, в рамках которого необходимо быстро тушить пожары, наиболее простым и быстрым способом. Код пишется, не соблюдая требований платформы, на которой разрабатывается продукт, появляется множество обходных решений и дефектов, а такая конструкция не очень устойчива и не безопасна, растет негодование клиентов от частых сбоев в работе программного обеспечения. Бизнес на выходе получает потери, падает качество планирования.

Agile Modeling (AM) – это набор понятий, принципов и приемов (практик), позволяющих быстро и просто выполнять моделирование и документирование в проектах разработки программного обеспечения (ПО).

AM описывает стиль моделирования, который позволит повысить качество и сократить сроки. AM не является технологическим процессом. Это не детальная инструкция по проектированию, он не содержит описаний, как строить диаграммы на UML. AM сосредоточен на эффективном моделировании и документировании. Он не охватывает программирование и тестирование, хотя в нем и говорится о проверке модели кодом и рассматривается тестируемость моделей. AM также не включает вопросы управления проектом, развертывания и сопровождения системы.

AM должен рассматриваться как дополнение к существующим методам, а не самостоятельная технология. Этот метод должен использоваться для повышения эффективности труда разработчиков, использующих процессы eXtreme Programming (XP), Dynamic Systems Development Method (DSDM), или RUP.

AM декларирует максимальное участие заинтересованных лиц, создание моделей и документов, доступных для понимания заинтересованными лицами. Документация должна быть предельно простой и краткой. Она должна соответствовать основному назначению – описанию проектируемой системы и быть понятной заинтересованным лицам. Допускается использование любых инструментальных средств, упрощающих и ускоряющих проектирование.

Цели Agile Modeling

- Показать, как применять на практике набор понятий, принципов и приемов, позволяющих легко и просто выполнять моделирование. AM сосредоточен не на технике построения конкретных диаграмм, а на том, как их использовать.
- Показать, как использовать известные методики моделирования (XP, DSDM, RUP и др.) в гибком подходе к разработке проектов.
- Повысить эффективность моделирования в проектах на разных стадиях (бизнес-анализ, формирование требований, анализ и дизайн).

Основные принципы Agile Modeling

- Ключами к успеху проекта являются эффективное взаимодействие между его участниками; стремление применять наиболее простые решения, отвечающие требованиям; использование обратной связи как можно чаще и начиная с ранних стадий; критическая оценка собственных решений; понимание того, что все участники проекта могут вносить свой существенный вклад в проект.
- Предпочитайте простоту. Считайте простое решение наилучшим. Не вводите дополнительные свойства, не требующиеся сегодня. Стремитесь, чтобы модель была максимально простой. Нужно моделировать, основываясь на существующих требованиях, и пересматривать модель, если требования будут изменяться в будущем.
- Учитывайте изменения. Требования к системе эволюционируют, изменяются взгляды заинтересованных лиц и критерии их оценки ваших результатов. Это должно обязательно учитываться в проекте.
- Обеспечивайте будущую работу. Важно обеспечить возможность поддержки и выпуска следующих релизов ПО. Для этого требуется создать необходимую документацию по программному продукту и его поддержке. Надо определить, будет ли команда разработчиков ПО участвовать в следующем релизе, суть следующей работы,

важность следующего релиза для вашей организации. То есть, не следует закрывать глаза на будущее.

- Вносите изменения последовательно. Не следует стремиться к абсолютной правильности модели сразу с самого начала. Маловероятно, что удастся достигнуть этого даже ценой сверхусилий. Вместо этого следует сначала строить небольшую модель высокого уровня и далее развивать ее инкрементным образом (либо удалить ее, если в ней не будет необходимости).
- Привлекайте больше заинтересованных лиц. Наиболее эффективно включить в команду представителей заинтересованных лиц (заказчика пользователей).
- Определяйте цели моделирования. Моделирование должно помочь лучшему пониманию системы. Возможно, модель понадобится для того, чтобы обосновать проект перед руководством, возможно, потребуется ее включение в документацию для пользователей или тех, кто будет выполнять поддержку. Если вы не можете определить, для кого и зачем создается модель, зачем это делать вообще? Первым шагом должно быть определение точного назначения модели и для кого она предназначена. Затем, исходя из этого, строится модель с той степенью детализации, которая требуется для поставленной цели. После этого построение этой модели заканчивается и можно перейти к другой работе, например, написанию кода, который показывает, как модель работает. Этот принцип применяется и при внесении изменений в существующую модель. Должна быть существенная причина для внесения изменений (например, введение нового требования или необходимость уточнения). Нужно четко знать для кого предназначена модель и что нужно этому потребителю (например, программисту).
- Используйте много моделей. Для разработки ПО необходимо построить много моделей, каждая из которых описывает некоторый аспект проблемы. Нужно иметь комплекс методик проектирования на базе выбранного инструментального средства. Важно отметить, что не требуется всегда строить весь набор моделей для любой системы. В зависимости от природы ПО создаются то или иное подмножество возможного набора моделей.
- Обеспечивайте качество. Не должно быть неряшливо сделанных работ. Результаты такой работы трудно понимаемы, не говоря уже об их обновлении. В некачественном коде тяжело разбираться и его трудно сопровождать.

- Обеспечьте быструю обратную связь. Обычно вы работаете над моделью совместно с другими людьми, то есть осуществляется процесс разделенного моделирования. Здесь важна быстрая обратная связь. Используя техники совместного моделирования, вы получаете мгновенный ответ на свои идеи. Если речь идет о заказчике, то здесь важно обеспечить быстрое и правильное понимание им требований, и анализ пользовательского интерфейса.
- ПО – главная цель. Главная задача – создать ПО, эффективно отвечающее нуждам заказчика и других заинтересованных лиц. Избегайте создания излишней документации или моделей. Ни одна деятельность, напрямую не связанная с главной задачей, не должна выполняться.
- Минимизируйте количество создаваемых документов. Каждый создаваемый артефакт должен поддерживаться в дальнейшем. Если вы используете несколько моделей, то при внесении изменений необходимо оценить их влияние на все эти модели и выполнить их изменение. Чем больше поддерживаемых артефактов, тем сложнее внесение изменений. Точно также сложнее поддерживать более детальные модели. Решив поддерживать определенную модель, вы теряете в скорости, но получаете возможность предоставления дополнительной информации для разработчиков. Не следует недооценивать серьезность потерь времени. Поддерживая много моделей, вы можете заметить, что большой объем времени требуется на обновление этих документов, а не на написание кодов.

Дополнительные принципы

- Содержание важнее репрезентативности. Каждую модель можно представить многими способами. Модель не обязана быть выходным документом. Сложный набор диаграмм, построенных с помощью инструментальных средств, может рассматриваться как исходные данные для других артефактов, например, исходного кода и не являться официальной документацией. Таким образом, можно использовать преимущества моделирования без потерь времени и ресурсов на документирование.
- Учиться друг у друга. Технологии программирования меняются быстро, например, Java. Технология моделирования меняется медленнее, но тоже изменяется. Работа в команде дает возможность обмениваться знаниями, приемами навыками.
- Знайте ваши модели. Для эффективного использования моделей важно знать их сильные и слабые стороны.

- Знайте инструментальные средства. Вы должны хорошо знать средства моделирования, их свойства и как применять на практике.
- Адаптация к особенностям. Подход к разработке ПО должен учитывать особенности организации разработчика, лиц, заинтересованных в проекте и типа самого проекта. Следует иметь методики моделирования, использования инструментальных средств, описание процесса (технологию).
- Открытое взаимодействие. Все должны иметь возможность высказывать свое мнение и предложения по моделированию, требованиям, срокам и т. д. Это позволяет выбирать наилучшие решения, основываясь на более точной, качественной информации.
- Используйте интуицию. Если кто-то чувствует, что некоторые вещи в проекте не согласованы, то в большинстве случаев это так и есть. Если вам кажется, что требование не имеет смысла, следует это тщательно исследовать совместно с пользователем. Аналогично следует поступать при выборе архитектурных решений. Из двух равноценных вариантов выбирайте тот, который вам подсказывает интуиция.

Приемы гибкого моделирования (практики)

AM предлагает набор приемов работы (называемых практиками), отвечающих сформулированным принципам.

- Активное участие заинтересованных лиц. Необходимо иметь прямую связь с пользователями, которые имеют право и возможность предоставлять информацию, имеющую отношение к системе, и принимать решения относительно требований и их приоритетов. В число таких людей входят непосредственные пользователи, руководство, операторы, служба поддержки.
- Используйте подходящие артефакты. Каждый артефакт имеет собственное применение. Например, диаграммы деятельностей UML полезны для описания бизнес-процессов, в то время как статическая структура БД лучше представляет физические данные. Нужно знать достоинства и недостатки каждого вида представления, чтобы определить, когда какое нужно или не нужно.
- Коллективная собственность. Каждый участник проекта должен иметь возможность работать с любой моделью или другим артефактом.
- Учитывайте тестируемость. При моделировании целесообразно периодически задаваться вопросом «Как это тестировать?», поскольку нельзя создавать нетестируемые

приложения. Процессы тестирования и обеспечения качества должны охватывать весь жизненный цикл.

- Параллельное моделирование. Поскольку каждый тип модели имеет свои достоинства и недостатки, одной модели недостаточно. Например, при выявлении требований можно создавать варианты использования или пользовательские описания, прототип UI и бизнес правила. Одновременная работа с несколькими моделями дает больший эффект за счет влияния на другие артефакты.
- Переход к другому артефакту. Если вы, работая над некоторым артефактом (вариант использования, CRC-карта, диаграмма последовательности и т. д.), застряли, переключитесь на время на другой. В этом случае можно продвинуться дальше, работая над другим артефактом. Более того, меняя взгляд на систему, можно понять то, что не удалось в первом случае.
- Простота. Смысловое содержание моделей модели – требований, архитектурных, аналитических и др. – должно быть, как можно более простым и понятным для всех заинтересованных лиц. Не добавляйте излишних аспектов в модели, если это не оправдано. Если, например, у вас нет требования реализовать в системе функции контроля, не добавляйте их в модель.
- Используйте простую нотацию. Простая модель показывает ключевые свойства, например, ответственности классов и их отношения, поэтому обычно используется небольшое подмножество выразительных средств для каждого вида диаграмм. Не следует тратить время на модели, которые не потребуются для дальнейшей работы. Вряд ли понадобится создание модели для всего кода.
- Общедоступность модели. Все модели должны быть доступны всем заинтересованным лицам. Организуйте простой доступ ко всем моделям. Их можно разместить на доске, создать сайт для представления в электронном виде и т. д.
- Инкрементное развитие моделей. Последовательное развитие моделей позволяет быстрее передать ПО заказчику (от 2 недель до одного-двух месяцев).
- Групповое моделирование. Моделирование служит для понимания чего-то, для распространения идей для всех, для выработки единой точки зрения, единого видения проекта. Модель позволяет путем обсуждения согласовать точки зрения членов команды разработчиков, объединить их идеи самым простым способом.

- Проверка модели кодом. Модель – это абстракция, она должна точно отражать аспекты того, что вы создаете. Необходимо проверять модель кодом. Эскизы страниц пользовательского интерфейса следует показать заинтересованным лицам. Построив модель фрагмента бизнеса, напишите тесты и код программы и прогоните тесты, чтобы убедиться, что все правильно. Не забывайте, что для большинства проектов моделирование – только один из видов выполняемых работ.
- Используйте самые простые средства. Большинство моделей могут быть построены на бумаге или доске, поскольку впоследствии они все равно становятся ненужными. Построение модели служит для понимания. Инструментальные средства используются только, если модель должна быть показана заинтересованным лицам или в ней есть необходимость при программировании (автоматическая генерация кода).

Agile Unified Process (AUP) – упрощенная версия IBM Rational Unified Process, созданная Скоттом Амблером (Scott Ambler) и состоящая из семи методов.

1. Моделирование используется для понимания бизнес-требования и предметной области.

2. Реализация – преобразование модели в исполняемый код с модульными тестами.

3. Тестирование – способ поиска дефектов и верификации системы на предмет соответствия требованиям.

4. Размещение – доставка готовой системы пользователям.

5. Управление конфигурациями – управление доступом и версиями артефактов проекта.

6. Управление проектом – непосредственные активности, связанные с ходом проекта: управление и координация людей, управление рисками, финансами и т. д.

7. Среда – совокупность процессов, инструментов, стандартов и правил.

Crystal Clear – это легковесная гибкая методология, созданная Алистером Кокберном. Она предназначена для небольших команд из 6–8 человек для разработки некритических бизнес-приложений. Как и все гибкие методологии, Crystal Clear больше опирается на людей, чем на процессы и артефакты.

Crystal Clear использует семь методов/практик, три из которых являются обязательными.

1. Частая поставка продукта.

2. Улучшения через рефлексию.

3. Личные коммуникации.

4. Чувство безопасности.

5. Фокусировка.

6. Простой доступ к экспертам.

7. Качественное техническое окружение.

Как видите, все практики характерны для семейства Agile-методологий. В графическом виде практики Crystal Clear можно изобразить таким образом.

Agile Data Method — это серия подходов к обеспечению разработки программного обеспечения, которое предполагает итеративную разработку посредством включения в этот процесс самоорганизующихся рабочих групп, причем каждая из таких рабочих групп занимается исключительно своей профессиональной функцией.

Говоря простым языком, это означает ведение работы по разработке, при этом в процессе последней ведется беспрерывный анализ данных. Надо сказать, что такая система весьма сложна сама по себе и требует большого профессионализма. Более того, тут нужен как аналитический подход, так и сугубо технический, а сочетать человеку, который что-либо разрабатывает два этих практически противоположных качества, весьма и весьма за затруднительно.

Но интерактивная система в серии подходов разработки программного обеспечения становится все более и более популярной. Многие специалисты IT-отраслей прямо называют такую систему подходов ультрасовременной, указывая на то, что только те мировые компании, которые применяет ее, и будут успешны в дальнейшем, иные же просто напросто не выдержат такой конкуренции.

Методология DSDM (Dynamic Systems Development Method – метод разработки динамических систем) основана на подходе RAD (Rapid Application Development – быстрая разработка приложений) и включает в себя три стадии.

1. Предпроектная стадия, на которой авторизуется реализация проекта, определяются финансовые параметры и команда.

2. Жизненный цикл проекта представляет собой реализации проекта и включает в себя пять этапов.

3. Постпроектная стадия обеспечивает качественную эксплуатацию системы.

Жизненный цикл проекта включает в себя пять стадий (первые две фактически объединяются).

1. Определение реализуемости.

2. Экономическое обоснование.

3. Создание функциональной модели.

4. Проектирование и разработка.

5. Реализация.

Essential Unified Process (англ.) (EssUP) - «существенный унифицированный процесс для разработки ПО», разработанный Иваром Якобсоном как улучшение методологии RUP (Rational Unified Process). Он определеяет такие практики как use cases (диаграммы вариантов использования), итеративную разработку, архитектуру разработки на основе, командные и проектные практики, которые позаимствованы от RUP и CMMI. Идея состоит в том, что возможно применять эти все практики в подходящей ситуации и комбинировать их между собой в процессе.

Экстремальное программирование (XP) – это упрощенная методология организации разработки программ для небольших и средних по размеру команд разработчиков, занимающихся созданием программного продукта в условиях неясных или быстро меняющихся требований.

Основными целями XP являются повышение доверия заказчика к программному продукту путем предоставления реальных доказательств успешности развития процесса разработки и резкое сокращение сроков разработки продукта. При этом XP сосредоточено на минимизации ошибок на ранних стадиях разработки. Это позволяет добиться максимальной скорости выпуска готового продукта и даёт возможность говорить о прогнозируемости работы. Практически все приемы XP направлены на повышение качества программного продукта.

Принципы XP

1. Итеративность. Разработка ведется короткими итерациями при наличии активной взаимосвязи с заказчиком. Итерации как таковые предлагается делать короткими, рекомендуемая длительность – 2-3 недели и не более 1 месяца. За одну итерацию группа программистов обязана реализовать несколько свойств системы, каждое из которых описывается в пользовательской истории. Пользовательские истории (ПИ) в данном случае являются начальной информацией, на основании которой создается модуль. Они отличаются от вариантов использования (ВИ). Описание ПИ короткое – 1-2 абзаца, тогда как ВИ обычно описываются достаточно подробно, с основным и

альтернативными потоками, и дополняются моделью. ПИ пишутся самими пользователями, которые в XP являются частью команды, в отличие от ВИ, которые описывает системный аналитик. Отсутствие формализации описания входных данных проекта в XP стремятся компенсировать за счет активного включения в процесс разработки заказчика как полноправного члена команды.

2. Простота решений. Принимается первое простейшее рабочее решение. Экстремальность метода связана с высокой степенью риска решения, обусловленного поверхностностью анализа и жестким временным графиком. Реализуется минимальный набор главных функций системы на первой и каждой последующей итерации; функциональность расширяется на каждой итерации.

3. Интенсивная разработка малыми группами (не больше 10 человек) и парное программирование (когда два программиста вместе создают код на одном общем рабочем месте), активное общение в группе и между группами. Все это нацелено на как можно более раннее обнаружение проблем (как ошибок, так и срыва сроков). Парное программирование направлено на решение задачи стабилизации проекта. При применении XP методологии высок риск потери кода по причине ухода программиста, не выдержавшего интенсивного графика работы. В этом случае второй программист из пары играет роль «наследника» кода. Немаловажно и то, как именно распределены группы в рабочем пространстве – в XP используется открытое рабочее пространство, которое предполагает быстрый и свободный доступ всех ко всем.

4. Обратная связь с заказчиком, представитель которого фактически вовлечен в процесс разработки.

5. Достаточная степень смелости и желание идти на риск.

Приемы XP (практики)

Обычно XP характеризуют набором из 12 правил (практик), которые необходимо выполнять для достижения хорошего результата. Ни одна из практик не является принципиально новой, но в XP они собраны вместе.

1. Планирование процесса. Вся команда разработчиков собирается вместе, принимается коллективное решение о том, какие свойства системы будут реализованы в ближайшей итерации. Трудоемкость реализации каждого свойства определяется самими программистами.

2. Тесное взаимодействие с заказчиком. Представитель заказчика должен быть членом XP-команды. Он пишет ПИ, выбирает истории, которые будут реализованы в конкретной итерации, и отвечает на вопросы, касающиеся бизнеса. Представитель заказчика должен быть экспертом в автоматизируемой предметной области. Необходимо наличие постоянное обратной связи с представителем заказчика.

3. Общесистемные правила именования. Хорошие системные правила именования предполагают простоту именования классов и переменных. Команда разработчиков должна иметь единые правила именования.

4. Простая архитектура. Любое свойство системы должно быть реализовано как можно проще. Программисты в XP-команде работают под девизом: «Ничего лишнего!». Принимается первое простейшее работающее решение, реализуется необходимый уровень функциональности на данный момент. Тем самым экономится время программиста.

5. Рефакторинг. Это оптимизация существующего кода с целью его упрощения, Такая работа должна вестись постоянно. 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. Кроме того, нельзя совмещать рефакторинг с дизайном: если создается новый код, рефакторинг следует отложить.

6. Парное программирование. Все программисты должны работать в парах: один пишет код, другой смотрит. Таким образом, необходимо размещать группу программистов в одном месте. XP наиболее успешно работает в нераспределенных коллективах программистов и пользователей.

7. 40-часовая рабочая неделя. Программист не должен работать более 8 часов в день. Необходимость сверхурочной работы – это четкий индикатор проблемы на данном конкретном направлении разработки. Поиск причин сверхурочной работы и их скорейшее устранение – одно из основных правил.

8. Коллективное владение кодом. Каждый программист в коллективе должен иметь доступ к коду любой части системы и право вносить изменения в любой код. Обязательное правило: если программист внес изменения и система после этого работает некорректно, то именно этот программист должен исправить ошибки.

9. Единые стандарты кодирования. 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. Команда работает над проектом продолжительное время. Люди приходят и уходят. Никто не кодирует в одиночку и код принадлежит всем. Всегда будут моменты, когда необходимо будет понять и скорректировать чужой код. Разработчики будут удалять дублирующий код, анализировать и улучшать чужие классы и т. п. Со временем нельзя будет сказать, кто автор конкретного класса. Следовательно, все должны подчиняться общим стандартам кодирования – форматирование кода, именование классов, переменных, констант, стиль комментариев. Вышесказанное означает, что все члены команды должны договориться об общих стандартах кодирования. Неважно каких, но все обязаны им подчиняются.

10. Небольшие релизы. 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 ПИ. Поскольку пользователь включается в процесс разработки начиная с первого релиза, то он оценивает систему и выдает пользовательскую историю и замечания. На основании этого определяется следующая итерация, то есть, каким будет новый релиз. В XP все направлено на обеспечение непрерывной обратной связи с пользователями.

11. Непрерывная интеграция. 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.

12. Тестирование. В отличие от большинства остальных методологий тестирование в XP – одно из важнейших составляющих. Экстремальный подход предполагает, что тесты пишутся до написания кода. Каждый модуль обязан иметь unit

test – тест данного модуля. Таким образом, в XP осуществляется регрессионное тестирование, «неухудшение качества» при добавлении функциональности. Большинство ошибок исправляются на стадии кодирования. Тесты пишут сами программисты, любой из них имеет право написать тест для любого модуля. Еще один важный принцип: тест определяет код, а не наоборот (test-driven development), то есть кусок кода кладется в хранилище тогда и только тогда, когда все тесты прошли успешно, в противном случае данное изменение кода отвергается.

Процесс XP является неформальным, но требует высокого уровня самодисциплины. Если это правило не выполняется, то XP мгновенно превращается в хаотичный и неконтролируемый процесс. XP не требует от программистов написания множества отчетов и построения массы моделей. В XP каждый программист считается квалифицированным работником, который профессионально и с большой ответственностью относится к своим обязанностям. Если в команде этого нет, то внедрять XP абсолютно бессмысленно – лучше для начала заняться перестройкой команды. Риск разработки снижается только в команде, которой XP подходит идеально, во всех остальных случаях XP – это процесс разработки с наиболее высокой степенью риска, поскольку другие методы снижения коммерческих рисков, кроме человеческого фактора, в XP просто отсутствуют.

Feature driven development (**FDD**, *разработка, управляемая функциональностью*) — итеративная методология разработки программного обеспечения, одна из гибких методологий разработки (agile). FDD представляет собой попытку объединить наиболее признанные в индустрии разработки программного обеспечения методики, принимающие за основу важную для заказчика функциональность (свойства) разрабатываемого программного обеспечения. Основной целью данной методологии является разработка реального, работающего программного обеспечения систематически, в поставленные сроки.

FDD была изначально предложена Джеффом Де Люкой для проекта (рассчитанного на 15 месяцев и 50 человек) по разработке программного обеспечения для одного крупного сингапурского банка в 1997 году. Де Люка выделил набор из пяти процессов, охватывающий как разработку общей модели, так и ведение списка, планирование, проектирование и реализацию элементов функциональности.

Первое описание FDD появилось в 1996 году в главе 6 книги *Java Modeling in Color with UML*. В книге *A Practical Guide to Feature-Driven Development*(2002 год) описание FDD было обобщено, и в частности избавлено от привязок к конкретному языку программирования.

FDD включает в себя пять базовых видов деятельности:

1. разработка общей модели;
2. составление списка необходимых функций системы;
3. планирование работы над каждой функцией;
4. проектирование функции;
5. реализация функции.

Первые два процесса относятся к началу проекта. Последние три осуществляются для каждой функции. Разработчики в FDD делятся на «хозяев классов» и «главных программистов». Главные программисты привлекают хозяев задействованных классов к работе над очередным свойством. Работа над проектом предполагает частые сборки и делится на итерации, каждая из которых предполагает реализацию определенного набора функций.

Разработка общей модели

Разработка начинается с высокоуровневого сквозного анализа широты решаемого круга задач и контекста системы. Далее для каждой моделируемой области делается более детальный сквозной анализ. Сквозные описания составляются в небольших группах и выносятся на дальнейшее обсуждение и экспертную оценку. Одна из предлагаемых моделей или их объединение становится моделью для конкретной области. Модели каждой области задач объединяются в общую итоговую модель, которая изменяется в ходе работы.

Составление списка возможностей (функций)

Информация, собранная при построении общей модели, используется для составления списка функций. Это осуществляется разбиением областей на подобласти (предметные области) с точки зрения функциональности. Каждая отдельная подобласть соответствует какому-либо бизнес-процессу, шаги которого становятся списком функций (свойств). В данном случае функции — это маленькие части понимаемых пользователем функций, представленных в виде «<действие> <результат> <объект>», например, «проверка пароля пользователя». Разработка каждой функции должна занимать не более

2 недель, иначе задачу необходимо разбить на несколько подзадач, каждая из которых сможет быть завершена за установленный двухнедельный срок.

План по свойствам (функциям)

После составления списка основных функций, наступает черёд составления плана разработки программного обеспечения. Владение классами распределяется среди ведущих программистов путём упорядочивания и организации свойств (или наборов свойств) в классы.

Проектирование функций

Для каждого свойства создается проектировочный пакет. Ведущий программист выделяет небольшую группу свойств для разработки в течение двух недель. Вместе с разработчиками соответствующего класса ведущий программист составляет подробные диаграммы последовательности для каждого свойства, уточняя общую модель. Далее пишутся «болванки» классов и методов, и происходит критическое рассмотрение дизайна.

Реализация функции

После успешного рассмотрения дизайна, данная видимая клиенту функциональность реализуется до состояния готовности. Для каждого класса пишется программный код. После модульного тестирования каждого блока и проверки кода, завершенная функция включается в основной проект.

FDD построен на основе набора наилучших практик, признанных в отрасли и полученных из инженерии программного обеспечения. Эти практические методы строятся с точки зрения важного для клиента функционала.

- Объектное моделирование области. Объектное моделирование состоит из исследования и выяснения рамок предметной области решаемой задачи. Результатом является общий каркас, который можно в дальнейшем дополнять функциями.
- Разработка по функции. Любая функция, которая слишком сложна для разработки в течение двух недель, разбивается на меньшие подфункции до тех пор, пока каждая подзадача не может быть названа свойством (то есть, быть реализована за 2 недели). Это облегчает создание корректно работающих функций, расширение и модификацию системы.

- Индивидуальное владение классом (кодом). Означает, что каждый блок кода закреплён за конкретным владельцем-разработчиком. Владелец ответственен за согласованность, производительность и концептуальную целостность своих классов.
- Команда по разработке функций (свойств). Команда по разработке функций (свойств) — маленькая, динамически формируемая команда разработчиков, занимающаяся небольшой подзадачей. Позволяет нескольким разработчикам участвовать в дизайне свойства, а также оценивать дизайнерские решения перед выбором наилучшего.
- Проверка кода (англ. code review) Проверки обеспечивают хорошее качество кода, в первую очередь путём выявления ошибок.
- Конфигурационное управление. Помогает с идентификацией исходного кода для всех функций (свойств), разработка которых завершена на текущий момент, и с протоколированием изменений, сделанных разработчиками классов.
- Регулярная сборка. Регулярная сборка гарантирует, что всегда есть продукт (система), которая может быть представлена заказчику, и помогает находить ошибки при объединении частей исходного кода на ранних этапах.
- Обозримость хода работ и результатов. Частые и точные отчёты о ходе выполнения работ на всех уровнях внутри и за пределами проекта о выполненной работе помогают менеджерам правильно руководить проектом.

К самым последним новшествам в области методологий разработки программного обеспечения можно отнести вышедшую в 2006 году книгу компании 37signals «Getting Real» , в которой описывается одноименная методология, всецело направленная на минимизацию каких либо бюрократических издержек, вроде составления спецификации или штатного расписания. В данном подходе к разработке рекомендуется начинать разработку с дизайна интерфейса, затем переходить к самому интерфейсу и уже в конце к программированию. Таким образом, подразумевается в кратчайшие сроки выпустить минимальную версию продукта и развивать ее, что в наибольшей степени подходит для веб проектов.

Getting Real предлагается в первую очередь для использования в небольших компаний и командах, где различные организационные обязательства не являются необходимым условием выживания и лишь обременяют коллектив. В целом данная методология соответствует, либо способствует большей части принципов описанных в

экстремальном программирование и манифесте гибких методологий. Помимо этого, методология затрагивает такие области знаний, как:

- Управление приоритетами;
- Управление коммуникациями;
- Управление персоналом;
- Управление ценообразованием;
- Продвижение;
- Поддержка.

1. Делайте меньше

Сделайте меньше, чем ваши конкуренты. Решите базовые простые проблемы и оставьте решение сложных всем остальным.

2. Делайте ПО для себя

Если вы решаете собственные проблемы и становитесь для себя потенциальным клиентом, соответственно, и знание о том, что нужно у вас тоже есть. Знайте, что вы не одни: если у вас есть какая-то проблема, то такая же есть ещё у сотни или у тысячи человек. Когда вы решаете собственную проблему, вы делаете инструмент с душой. А это значит, что продукт точно будет успешен.

3. Финансируйте себя сами

Если вы привлекаете инвесторов, вы несёте перед ними ответственность. Чтобы уложиться в минимальный бюджет вам просто нужно начать работать не с 10 людьми, а с тремя. Расставьте приоритеты и подумайте, что вы можете сделать с тремя людьми? Это реально будет главный функционал, который, вообще-то и нужен.

4. Используйте FFF

«FFF» — fix time, fix budget, flex scope.

Чтобы успешно работать по ффф, вам потребуется расставить приоритеты и определить, что действительно важно для продукта. Ну и гибкость тоже пригодится.

5. Придумайте себе врага

Найдите монструозное ПО от конкурентов и сделайте противоположное. Когда 37signals решили работать над Basecamp, они выбрали в качестве «монстра» MS Project и постарались сделать намного меньше, но гораздо лучше.

6. Никакой рутины

Чем меньше в вашем приложении рутины, — тем лучше. Рутина — это неинтересные действия, которые вы выполняете только потому, что вынуждены.

7. Страсть

Делайте продукт с душой и это заметят. Ничто так не огорчает, как бездушный продукт. С другой стороны, когда вы видите, что разработчики делали продукт с душой и уделили достаточно внимания вашему пользовательскому счастью, вы будете довольны.

8. Оставайтесь маленькими

Для того, чтобы остаться маленькой компанией нужно избегать следующего:

- Долгосрочные контракты
- Большой штат сотрудников
- Неизменные решения
- Совещания о других совещаниях
- Долгий процесс
- Обширный инвентарь (физический или умственный)
- Привязка к оборудованию или программам
- Закрытые форматы данных
- Управление будущим на основании прошлого
- Долгосрочные цели
- Офисная политика

Уменьшить компанию позволяют:

- Решения, принимаемые по мере надобности
- Многозадачность членов команды
- Установка ограничений, вместо попыток их преодолеть
- Уменьшение программного обеспечения, меньше кода
- Меньшее количество функций продукта
- Маленькая команда
- Простота
- Открытые исходные коды
- Открытые форматы данных
- Свободная атмосфера, в которой легче признавать ошибки

Меньшие размеры позволяют быстрее сменить направление, слушать своих клиентов и отвечать им быстро.

9. Непредсказуемость

В вашем продукте могут внезапно появиться новые свойства. Многие методологии разработки ПО настолько всё формализуют, что не дают произойти случайности или внезапному появлению идей. Разумная система позволит продукту эволюционировать, дав возможность изменению. Чем больше жёстких установок, тем меньше места для творчества.

10. Три мушкетёра

Используйте команду из трёх человек в первой версии. Это достаточный ресурс, который позволяет быть быстрыми и проворными. Начинать можно с разработчиком, дизайнером и человеком, который разбирается и в том и в другом.

Недостаток людей создаст вам ограничения на первом же этапе, заставит вас раньше задуматься над приоритетами. Если вы не можете разработать продукт с тремя людьми, — значит вы или работаете не с теми людьми, или нужно снижать требования.

Пути взаимоотношений в маленькой команде намного проще. Если вы один человек в команде, — общение происходит только между вами и клиентом. Если количество людей увеличивается, количество необходимых связей увеличивается многократно.

11. Принимайте ограничения

Когда 37signals делали Basecamp у них было много ограничений. Они начинали, как дизайнерская фирма с текущей клиентской базой с 7-ми часовой разницей во времени с одним из разработчиков. У них была маленькая команда без внешнего финансирования. Но они приняли ограничения, взяли большие задачи и разделили их на маленькие куски, чтобы попытаться выполнить их по одному. Разница во времени стала серьёзным фактором коммуникации и вместо встречи они связывались почти исключительно через мессенджеры и электропочту.

12. Всегда и в любое время

Хорошее обслуживание клиентов обязательно должно быть. И не важно, в каком вы бизнесе. Сделайте так, чтобы клиенты могли связаться с вами по любому вопросу. Укажите на сайте ваши почты и телефоны. Сделайте упор на то, что клиенты могут добраться до вас и связаться в любое время.

13. Приоритеты

Ясно и точно определите видение для вашего приложения. Что оно должно делать? Почему оно должно существовать? В чём отличия от аналогов? Это видение будет вести вас по правильному пути и каждый раз, когда у вас будут сомнения, — посмотрите на видение и вы увидите, что делать.

14. Пренебрегайте деталями в начале

Успех и удовлетворение находятся в деталях, однако там же вы найдёте и застой, разногласия, встречи и задержки. Эти вещи могут уничтожить весь проект. Если вы сидите над одной строчкой кода в течение целого дня или работа, сделанная за целый день не дала никакого прогресса, значит, вы слишком рано сосредоточились на деталях. Позже вы сможете всё усовершенствовать.

15. Не тратьте время на проблемы, которых ещё нет

Вам просто нужно принимать решения вовремя, имея всю необходимую для этого информацию. Знайте, что в любом случае вам придётся переписать весь код рано или поздно.

16. Работайте с правильными клиентами

Нужно найти основной рынок для вашего продукта и сконцентрироваться исключительно на нём. Если вы ориентируетесь на всех, — вы не ориентируетесь ни на кого.

17. Выбор функций

Создайте половину продукта, но пусть это будет законченный продукт. Вы можете что-то не выпускать в релиз, но всё самое важное, сердце продукта, должно работать. Потом вы сможете добавлять функции.

Вы должны понимать скрытые затраты на новые функции. Подумайте, сколько эта небольшая функция принесёт головной боли.

Вместо того, чтобы спрашивать у людей, что они хотят, спросите у них что они не хотят. «Скажите, какими функциями вы не пользуетесь?» или «Если бы вы могли убрать одну особенность, чтобы это было?». Побольше «нет» в вопросах.

18. Создавайте ПО для общих решений

Не навязывайте людям свои решения. Предоставьте людям инструмент, с помощью которого будет достаточно просто решить их собственные проблемы. Сделайте решения базовых задач, а люди найдут собственные решения в пределах вашей общей структуры.

19. Работайте итерационно

Не ожидайте того, что будете понимать и делать всё правильно с первого раза. Проектируйте, используйте, анализируйте, переделывайте и запускайте снова. Вы будете принимать информированные обоснованные решения; у вас всегда есть обратная связь и реальное понимание того, что требует вашего внимания.

Если вы знаете, что собираетесь переделать всё снова, вам не нужно нацеливаться на совершенствования при первой попытке.

20. Избегайте настроек

Настройки — это уход от пути принятия жёстких решений. Вы тут для того, чтобы принимать в вашем продукте решения. Настройки — это зло, которое создаёт головную боль для клиента и раздувает код. А в реальности настройками никто и не пользуется. Настройки предполагают, что вы мало знаете о том, как всё должно работать.

Возможно, вы сделаете плохой выбор, но это не смертельно: люди будут жаловаться, и вы очень быстро об этом узнаете. Всегда можно подкорректировать и сделать правильно.

21. Разделяйте и объединяйте

Разбейте проект на десяток небольших кусков, если он требует 10 недель работы. Если вы не можете держать большую проблему в голове, — разбейте её на такие части, которые легко в вашей голове поместятся.

А вот организационно лучше объединять. Разделение по отделам и специализациям даёт то, что каждый специалист видит только свой аспект приложения и не видит общей картины. Как можно больше смешивайте команду, чтобы наладить диалог. Убедитесь, что разработчики знают, как работает поддержка; что поддержка знает, как дизайнеры принимают решения; что дизайнер знает, как работает фреймворк, на котором пишут программисты.

22. Единое время и никаких встреч

У вас должно быть единое время на работу. Особенно это актуально в распределённой по часовым поясам команде. Установите время, в которое вы будете работать, ни на что, не отвлекаясь. Сделайте половину рабочего дня временем, когда вы не будете отвечать на телефонные звонки, попусту говорить с коллегами, чтение почты и т.д. Избегайте любых перерывов в это время. Каждый перерыв отвлекает настолько, что потом необходимо опять вникать в работу.

Если вам понадобилась встреча, значит, есть какая-то неясность. Раз есть неясность, то у вас нет чёткого понимания задачи. Это вытекает из нечётких целей, нечёткой

постановки или «трудностей перевода». Встречи разбивают ваш рабочий день на куски и выбивают из технологического процесса.

23. Персонал. Нанимайте меньше и позже

Нет необходимости нанимать много людей слишком рано. Если у вас планируются продажи, — нет смысла нанимать аккаунт-менеджера, который понадобится только через 3-4 месяца. Если вы справляетесь, — справляйтесь.

Если у вас есть возможность быстро нанять много очень хороших людей, — это тоже плохая идея: вы не сможете быстро сделать из них хороший сплочённый коллектив. Будет много головной боли, а на ранних этапах развития это смертельно.

Если вы с чем-то не справляетесь, подумайте, нужно ли это делать вообще? Есть ли обходные пути? Только ели другого выхода нет, — нанимайте.

Ищите потенциальных сотрудников среди Open Source разработчиков. Так вы проследите, что делал этот человек длительное время и будет понятно, на что он способен. Вы оцените человека по сделанному, а не по сказанному. Посмотрите на его видение, которое можно определить по комментариям и переписке в проекте. Его видение не должно противоречить вашему.

24. Интерфейс до программирования

Создайте дизайн интерфейса до начала программирования. Начинать с программирования не слишком хорошо, потому что программирование — самая сложная и дорогая часть работы. Создав код вам будет сложно и дорого его изменить. Если же вы начинаете с рисунков интерфейса на бумаге, переделать их вам не стоит почти ничего.

С другой стороны, интерфейс — это и есть ваш продукт. Конечным пользователям вы продаёте именно его.

25. Три состояния программы

Делайте дизайн для обычного, пустого и ошибочного состояний программы. В большинстве случаев мы видим продукт, когда он заполнен. У нас есть тестовые данные, скрипты, которые заполняют БД автоматически и всё такое. Но зарегистрировавшийся только что пользователь видит совершенно иную картину. Для него ваш продукт — это чистый лист. Помогите ему, чтобы он разобрался быстро и усвоил, как работать, что добавить в первую очередь. Сделайте подсказки, которые будут направлять пользователей.

26. Сохраняйте код программы простым

Когда мы пишем в два раза больше кода, — это не значит, что программа станет в два раза сложнее. Сложность программы растёт экспоненциально. Переформулируйте задачу, которая требует много кода, так, чтобы можно было решить 80% требований, затратив 20% усилий. Это очень большой выигрыш.

27. Инструменты, дающие счастье

Счастливый программист — продуктивный программист. Выбирайте инструменты, основываясь не только на стандартах и на производительности, но и на том, даёт ли инструмент программистское счастье.

28. Слушайте свой код

Ваш код будет сопротивляться неэлегантным решениям. Возможно, он будет говорить вам, что существует более лёгкий способ решения. Выбирайте функцию, которую легко сделать. Возможно, она будут не в точности такой, какой вы её представляли в начале, но, если она работает достаточно хорошо и оставляет время на другие дела, — оставьте её.

29. Расплачивайтесь по долгам кода и дизайна

Написали блок кода, который работает, но неопрятен? —Это долг.

Набросали дизайн по принципу «и так сойдёт»? —Это ещё один долг.

На самом деле, это нормальный подход, когда вы делаете продукт к определённой дате, чтобы сделать пользователей счастливыми. Но долги нужно закрывать, так что регулярно выделяйте время, чтобы расплатиться по накопившимся долгам. Иначе, у вас будут скапливаться тонны костыльного кода, который будет очень сложно разобрать.

30. Откройте API. Свободно впускайте и выпускайте данные

Так вы облегчите всем жизнь. Дайте людям возможность обмениваться данными с вашим продуктом. API позволит множеству разработчиков расширять функционал вашего продукта. Не нужно недооценивать тот вклад, который могут внести другие разработчики в ваш продукт.

31. Нет функциональным спецификациям

Это только слова на бумаге. Лучше вместо спецификации написать короткий рассказ в одну страницу. Опишите, что именно должно делать приложение. Напишите простым языком и сделайте это быстро. Если вам не хватило одной страницы, — значит вы очень усложняете. Этот процесс не должен занимать более одного дня.

Как только вы напишете этот небольшой рассказ, — начните строить интерфейс. Он и будет альтернативой функциональной спецификации. На нём всё видно и всё понятно. Двусмысленность уходит, когда все начинают видеть на экране одно и то же. Постройте инстерфейс, который люди смогут потыкать мышкой. Сделайте это до того, как вы начнёте беспокоиться о внутреннем коде.

32. Нормальные слова, вместо lorem ipsum

Стандартная рыба позволит понять, как будет выглядеть дизайн. Но настоящие слова позволят понять, должен ли дизайн выглядеть именно так. Текстовое содержание — это не просто часть визуального дизайна, но это важная и значимая информация. Текст-пустышка — это завеса между вами и действительностью.

Возьмите настоящую информацию, чтобы посмотреть, как будет растягиваться таблица. Введите в форму настоящие данные для того, чтобы понять, насколько пользователям будет удобно. Когда вы проверяете форму отправки данных, — не вводите в поля что-то типа «asdf». Если вам самим лень заполнять каждый раз все эти поля, то что же будет с пользователями?

33. Очеловечьте ваш продукт

Определите тип личности вашего продукта. Он может быть строгим, шутливым, игривым, бесстрастным, требовательным и т. д. Когда вы принимаете решение о характере, — следуйте этому характеру и имейте в виду эти черты, во время разработки продукта. Всякий раз, когда вы вносите изменения, подумайте, насколько это изменение соответствует типу характера вашего приложения.

У вашего продукта есть голос и он разговаривает с вашими клиентами 24 часа в сутки.

34. Цена и регистрация

Дайте что-нибудь бесплатно. Вокруг очень много всего и вас не заметят, если вы не дадите ничего бесплатно. У умных компаний так принято: дать что-то бесплатно/в подарок/по акции, чтобы увеличить лояльность покупателей, а, вместе с этим, и прибыль.

35. Сделайте регистрацию и отказ безболезненными

Процесс регистрации должен быть «для чайников». Поставьте большую и ясную кнопку на каждой странице вашего маркетингового сайта. Расскажите, что это просто: от регистрации до использования 1 минута! Сама по себе регистрация всегда должна быть бесплатной, чтобы пользователь смог войти и понять, что это ему нужно. Не запрашивайте

при регистрации информацию, которая вам не нужна. Просто спросите email, в большинстве случаев этого достаточно.

Бывает так, что люди хотят закрыть свой аккаунт. Это неприятно, но нормально. Не препятствуйте этому. Просто покажите ссылку: «Закрыть мой аккаунт». Дайте человеку выгрузить свои данные, чтобы не оказалось, что он из-за этих данных навечно заточён в вашей системе. Это критический фактор, который напрямую влияет на доверие пользователей к вам.

36. Избегайте долгосрочных контрактов, платы за подключение и т. д.

Одно дело, когда с нас требуют денег на год вперёд, а другое дело, когда на месяц. Это сильно влияет на продажи. Просто сделайте оплаты небольшими порциями.

Плата за подключение — тоже зло. Это просто фокус, чтобы собрать с клиентов побольше денег. Заслужите их деньги и они будут пользоваться вашим продуктом.

37. Плохие новости

Если у вас есть плохие новости — как можно раньше расскажите об этом. Когда вы повышаете тарифы, подсластите пилюлю вашим стареньким пользователям. Расскажите, что вы их цените, поэтому они какое-то время могут пользоваться вашим продуктом по старым ценам.

38. Выпуск продукта

Выпускайте продукт в голливудском стиле. Если вы выпускаете продукт без предварительной подготовки, — о нём никто никогда не узнает. Голливуд делает так:

- Анонс

За несколько месяцев начинайте намекать. Расскажите людям, над чем вы работаете, покажите логотип, расскажите о разработке в своём блоге. Заведите сайт, где вы будете собирать адреса почты интересующейся публики. На этой стадии вы будете окучивать знатоков и экспертов, т. е. тех, кто находится на переднем крае. Предоставьте им эксклюзивное право раннего просмотра продукта.

- Рекламный показ отрывков

Голливуд показывает отрывки из фильма. Вы делаете предварительный показ некоторых возможностей. Дайте несколько скриншотов, покажите основные возможности продукта. Расскажите об идеях, принципах, заложенных в программу.

- Выпуск

В Голливуде в этот момент люди идут в кино. В вашем случае они будут работать с вашим продуктом и пытаться его использовать. Запустите полноценный маркетинговый сайт. На нём должен быть обзор, экскурсия по продукту, скриншоты и видеоролики, манифест, примеры, отзывы, форум.

Распространяйте проповедь как только можно. Пусть на вас ссылаются блоги, сообщайте о своём прогрессе: сколько у вас зарегистрированных пользователей, сколько обновлений/поправок вы сделали? Покажите ваше движение вперёд.

Когда придёт день выпуска, разошлите сообщения по вашему списку рассылки.

39. Заведите блог

Реклама стоит дорого. Блог стоит только ваше потраченное время и обычно оно окупается.

Начните с создания блога, который не только хвалит ваш продукт, но и даёт какие-то полезные советы, решения, ссылки и т. д.

40. Продвижение через обучение

Делитесь знаниями с другими. Таким образом вы получаете аудиторию. Обучение — это путь к тому, чтобы ваше имя и название вашего продукта появилось перед множеством людей. Вместо того, чтобы навязывать ваш продукт, вы получаете внимание за предоставление образовательной услуги. Размещайте на вашем сайте советы и решения, которыми читателям захочется поделиться с другими. Выступайте на конференциях и оставайтесь, чтобы в конце поговорить с участниками. Проводите семинары, пишите статьи и книги.

41. Пища функциональности

Добавление новых функций — хороший способ усилить разговоры о вашем приложении.

42. Следите за упоминаниями

Если о вас говорят, — это хорошо. Следите за тем, что конкретно о вас говорят и своевременно реагируйте. Если о вас начинают активно говорить на каком-то форуме, — включитесь в беседу и ответьте на заданные вопросы. Есть пару примеров из жизни: на рутрекере в ветке, посвящённой Intellij Idea, представитель JetBrains отвечал на вопросы, а так же разработчики игры The Uncertain в день открытия соответствующей ветки уже отслеживали положение дел. Очень мило с их стороны и это реально производит хорошее впечатление.

Цените любые отзывы, в т.ч. и отрицательные.

43. Продажи внутри приложения

Предложите переход на расширенные возможности в самом приложении. Суть в том, что у вас должно быть несколько тарифных планов для разных категорий пользователей. Кому-то достаточно бесплатной версии, а кому-то нужен полный функционал. Дайте людям возможность докупать функции.

44. Лёгкое название

Не делайте мудрёное название. Оно должно быть простым и легко запоминаемым. Выберите короткое, яркое название и вперёд. Не увлекайтесь фокус-группами и прочими ненужными штуками.

Не переживайте, если доменное имя занято. Подберите что-нибудь близкое.

45. Почувствуйте боль поддержки

Сокрушите стены между разработкой и поддержкой. Пусть поддержкой занимаются сами разработчики. Это позволяет напрямую общаться с клиентами и понимать, что происходит с продуктом. Не передавайте техподдержки колл-центру или сторонней организации, но осуществляйте её сами.

46. Нулевое обучение

Сделайте встроенную в продукт систему помощи. Чтобы пользоваться Яндексом или Гуглом вам не нужны справочники. Почему бы вам не сделать ваш продукт таким же простым в использовании? Начните с простой программы. Чем меньше сложность, — тем меньше усилий для заблудших пользователей. Хорошая система помощи и списки FAQ — это очень хороший способ помочь.

47. Первые пользователи

Вы сможете конкурировать с большими компаниями тогда и только тогда, когда вы будете уделять внимание каждому пользователю. Первые пользователи — это те, кто укажет вам на ошибки в программе и на те нужды, которым ваша программа не удовлетворяет. А ещё они расскажут о вашей программе другим.

Когда пользователи сообщают об ошибках, — обязательно ответьте им как можно быстрее. Они рассчитывают на то, что вы прислушаетесь к ним и что вы о них заботитесь.

Общайтесь с людьми на форуме. Сначала вы будете отвечать на все вопросы сами, однако вскоре вы заметите, как более опытные пользователи начнут давать советы менее

опытным. Вам всё меньше нужно будет вмешиваться в процесс, однако всегда нужно быть рядом и быть готовым оказать помощь.

48. Настройка через месяц

Быстрые обновления показывают движение и то, что вы прислушиваетесь к запросам пользователей. Быстрое обновление подкрепляет положительные впечатления, связанные с продуктом и даёт дополнительную пищу для обсуждения в блогах.

Покажите, что ваш продукт живой. Продолжайте вести блог продукта после его выхода в свет. Не переставайте писать. Включите туда частые вопросы, лайфхаки, эффективные способы работы с программой, необычные приёмы, советы и решения, новые функции, обновления, исправления.

49. Не бета, а лучше

Не используйте слово «бета». Это уже намозолило всем глаза. Когда вы говорите, что вы выпустили бета-версию, вы фактически признаётесь в том, что продукт сырой. Нормально давать возможность бета-тестирования узкому кругу людей по приглашениям. Однако, делать бету общедоступной нельзя. Гораздо лучше использовать другой подход: хороший продукт с минимальным функционалом в первой версии.

50. Делите ошибки по приоритетам

Определять приоритеты дел — это одно из важнейших качеств в жизни и это касается не только программирования, но и любых других дел. Узнайте, что действительно важно и действуйте. Когда к вам приходит отчёт об ошибке, — не спешите тут же исправлять. Возможно, у вас есть дела поважнее. Когда дело касается небольших косяков с вёрсткой в старых браузерах — это не такая уж и большая беда. Если же дело касается пользовательских данных, — это должно быть исправлено немедленно и восприниматься как опасность.

51. Переждите шторм

Если вы что-то изменили, — переждите этот момент и не обращайте какое-то время на реакцию пользователей. Многие люди будут ругаться, пока не привыкнут. Однако, те, кого обычно всё устраивает, просто молчат. Это нормальное поведение и вы всегда будете в ситуации, что отрицательных отзывов будет больше, чем положительных. Не делайте откатов на основании подобных отзывов и просто продолжайте свою линию, если вы в ней уверены. Расслабьтесь и не двигайтесь какое-то время.

52. Не отставайте от конкурентов

Подпишитесь на новости о конкурентах и всегда знайте, что они делают. Читайте RSS, форумы. Вы всегда должны знать, что делают конкуренты, какие функции они внедряют и что думают по этому поводу их пользователи.

53. Двигайтесь по течению

Всегда будьте готовы для новых путей и изменения направлений. Если что-то изменилось, то не бойтесь изменяться сами. Примите тот факт, что ваша первоначальная идея могла быть не самой лучшей и, если это обнаружится, вы всегда можете изменить направление.

OpenUP — это итеративно-инкрементальный метод разработки ПО. Позиционируется как легкий и гибкий вариант RUP.

В основу OpenUP положены следующие основные принципы:

- Совместная работа с целью согласования интересов и достижения общего понимания;
- Развитие с целью непрерывного обеспечения обратной связи и совершенствования проекта;
- Концентрация на архитектурных вопросах на ранних стадиях для минимизации рисков и организации разработки;
- Выравнивание конкурентных преимуществ для максимизации потребительской ценности для заинтересованных лиц.

OpenUP делит жизненный цикл проекта на четыре фазы: начальная фаза, фазы уточнения, конструирования и передачи. Жизненный цикл проекта обеспечивает предоставление заинтересованным лицам и членам коллектива точек ознакомления и принятия решений на протяжении всего проекта. Это позволяет эффективно контролировать ситуацию и вовремя принимать решения о приемлемости результатов. План проекта определяет жизненный цикл, а конечным результатом является окончательное приложение.

OpenUP делит проект на итерации: планируемые, ограниченные во времени интервалы, длительность которых обычно измеряется неделями. План итерации определяет, что именно должно быть сдано по окончании итерации, а результатом является работоспособная версия. Коллективы разработчиков OpenUP строятся по принципу самоорганизации, решая вопросы выполнения задач итераций и передачи

результатов. Для этого они сначала определяют, а затем решают хорошо детализированные задачи из списка элементов работ.

Базовый процесс OpenUP является независимым от инструментов, малорегламентированным процессом, который может быть расширен для удовлетворения потребностей широкого диапазона типов проектов.

Scrum — это набор принципов, на которых строится процесс разработки, позволяющий в жёстко фиксированные и небольшие по времени итерации, называемые спринтами (sprints), предоставлять конечному пользователю работающее ПО с новыми возможностями, для которых определён наибольший приоритет. Возможности ПО к реализации в очередном спринте определяются в начале спринта на этапе планирования и не могут изменяться на всём его протяжении. При этом строго фиксированная небольшая длительность спринта придаёт процессу разработки предсказуемость и гибкость.

Канбан.

Этот термин пришёл к нам из Японии благодаря широко известной в узких кругах производственной системе Тойота. Хотелось бы, чтобы как можно больше людей прочитало про эту систему и основные принципы, заложенные в неё — бережливое производство, постоянное развитие, ориентацию на клиента и т.п. Все эти принципы описаны в книге Тайити Оно Производственная система Тойоты, которая переведена на русский.

Термин Канбан имеет дословный перевод: “Кан” значит видимый, визуальный, и “бан” значит карточка или доска.

На заводах Тойота карточки Канбан используются повсеместно для того, чтобы не загромождать склады и рабочие места заранее созданными запчастями. Например, представьте, что вы ставите двери на Тойоты Короллы. У вас около рабочего места находится пачка из 10 дверей. Вы их ставите одну за другой на новые машины и, когда в пачке остается 5 дверей, то вы знаете, что пора заказать новые двери. Вы берете карточку Канбан, пишете на ней заказ на 10 дверей и относите ее тому, кто делает двери. Вы знаете, что он их сделает как раз к тому моменту, как у вас закончатся оставшиеся 5 дверей. И именно так и происходит — когда вы ставите последнюю дверь, прибывает пачка из 10 новых дверей. И так постоянно — вы заказываете новые двери только тогда, когда они вам нужны.

А теперь представьте, что такая система действует на всём заводе. Нигде нет складов, где запчасти лежат неделями и месяцами. Все работают только по запросу и производят именно столько запчастей, сколько запрошено. Если вдруг заказов стало больше или меньше — система сама легко подстраивается под изменения.

Основная задача карт Канбан в этой системе — это уменьшать количество «выполняющейся в данный момент работы» (work in progress).

Например, на всю производственную линию может быть выделено ровно 10 карточек для дверей. Это значит, что в каждый момент времени на линии не будет больше 10 готовых дверей. Когда заказывать новые двери и сколько — это задача для того, кто их устанавливает. Только он знает свои потребности, и только он может помещать заказы производителю дверей, но он всегда ограничен числом 10.

Этот метод Бережливого производства (Lean manufacturing) был придуман в Тойоте и сейчас многие производственные компании по всему миру его внедряют или уже внедрили.

Канбан разработка отличается от SCRUM в первую очередь ориентацией на задачи. Если в SCRUM основная ориентация команды — это успешное выполнение спринтов, то в Канбан на первом месте задачи.

Спринтов никаких нет, команда работает над задачей с самого начала и до завершения. Деплоймент задачи делается тогда, когда она готова. Презентация выполненной работы — тоже. Команда не должна оценивать время на выполнение задачи, ибо это имеет мало смысла и почти всегда ошибочно вначале.
Если менеджер верит команде, то зачем иметь оценку времени? Задача менеджера — это создать приоритизированный пул задач, а задача команды — выполнить как можно больше задач из этого пула. Всё. Никакого контроля не нужно. Всё, что нужно от менеджера — это добавлять задачи в этот пул или менять им приоритет. Именно так он управляет проектом.
Команда для работы использует Канбан-доску. Например, она может выглядеть так Столбцы слева направо:

Цели проекта:

Необязательный, но полезный столбец. Сюда можно поместить высокоуровневые цели проекта, чтобы команда их видела и все про них знали. Например, «Увеличить скорость работы на 20%» или «Добавить поддержку Windows 7».

Очередь задач:

Тут хранятся задачи, которые готовы к тому, чтобы начать их выполнять. Всегда для выполнения берется верхняя, самая приоритетная задача и ее карточка перемещается в следующий столбец.

Проработка дизайна:

этот и остальные столбцы до «Закончено» могут меняться, т.к. именно команда решает, какие шаги проходит задача до состояния «Закончено».

Например, в этом столбце могут находиться задачи, для которых дизайн кода или интерфейса еще не ясен и обсуждается. Когда обсуждения закончены, задача передвигается в следующий столбец.

Разработка:

Тут задача висит до тех пор, пока разработка фичи не завершена. После завершения она передвигается в следующий столбец.

Или, если архитектура не верна или не точна — задачу можно вернуть в предыдущий столбец.

Тестирование:

В этом столбце задача находится, пока она тестируется. Если найдены ошибки — возвращается в Разработку. Если нет — передвигается дальше.

Деплоймент:

У всех проектов свой деплоймент. У кого-то это значит выложить новую версию продукта на сервер, а у кого-то — просто закомитить код в репозиторий.

Закончено:

Сюда стикер попадает только тогда, когда все работы по задаче закончены полностью.

В любой работе случаются срочные задачи. Запланированные или нет, но такие, которые надо сделать прямо сейчас. Для таких можно выделить специальное место (на картинке отмечено, как «Expedite»). В Expedite можно поместить одну срочную задачу и команда должна начать ее выполнять немедленно и завершить как можно быстрее. Но может быть только одна такая задача! Если появляется еще одна — она должна быть добавлена в «Очередь задач».

А теперь самое важное. Цифры под каждым столбцом - это число задач, которые могут быть одновременно в этих столбцах. Цифры подбираются экспериментально, но считается, что они должны зависеть от числа разработчиков в команде.

Например, если вы имеете 8 программистов в команде, то в строку «Разработка» вы можете поместить цифру 4. Это значит, что одновременно программисты будут делать не более 4-х задач, а значит у них будет много причин для общения и обмена опытом. Если вы поставите туда цифру 2, то 8 программистов, занимающихся двумя задачами, могут заскучать или терять слишком много времени на обсуждениях. Если поставить 8, то каждый будет заниматься своей задачей и некоторые задачи будут задерживаться на доске надолго, а ведь главная задача Канбан — это уменьшение времени прохождения задачи от начала до стадии готовности.

Никто не даст точный ответ, какие должны быть эти лимиты, но стоит попробовать для начала разделить число разработчиков на 2 и посмотреть, как это работает в вашей команде. Потом эти числа можно подогнать под вашу команду.
Под «разработчиками» понимаются не только программисты, но и другие специалисты. Например, для столбца «Тестирование» разработчики — это тестеры, т.к. тестирование — это их обязанность.

Что нового и полезного дает такая доска с лимитами?

Во-первых, **уменьшение числа параллельно выполняемых задач сильно уменьшает время выполнения каждой отдельной задачи.** Нет нужды переключать контекст между задачами, отслеживать разные сущности, планировать их и т.д. — делается только то, что нужно. Нет нужды устраивать спринт планнинги и 5% воркшопы, т.к. планирование уже сделано в столбце «очередь задач», а детальная проработка задачи начинается ТОЛЬКО тогда, когда задача начинает выполняться.

Во-вторых, **сразу видны проблемные места.** Например, если тестеры не справляются с тестированием, то они очень скоро заполнят весь свой столбец и программисты, закончившие новую задачу, уже не смогут переместить ее в столбец тестирования, т.к. он заполнен. Что делать? Тут время вспомнить, что «мы — команда» и решить эту проблему. Например, программисты могут помочь тестерам завершить одну из задач тестирования и только тогда передвинуть новую задачу на освободившееся место. Это позволит выполнить обе задачи быстрее.

В-третьих, можно вычислить **время на выполнение усредненной задачи**. Мы можем помечать на карточке дату, когда она попала в очередь задач, потом дату, когда ее взяли в работу и дату, когда ее завершили. По этим трем точкам для хотя бы 10 задач можно уже посчитать среднее время ожидания в очередь задач и среднее время

выполнения задачи. А из этих цифр менеджер или product owner может уже рассчитывать всё, что ему угодно.

Весь Канбан можно описать всего тремя основными правилами:

1. **Визуализируйте производство**

Разделите работу на задачи, каждую задачу напишите на карточке и поместите на стену или доску.

Используйте названные столбцы, чтобы показать положение задачи в производстве.

2. **Ограничивайте WIP** (work in progress или работу, выполняемую одновременно) **на каждом** этапе производства.

3. **Измеряйте время цикла** (среднее время на выполнение одной задачи) и **оптимизируйте постоянно процесс**, чтобы уменьшить это время.