УПРАВЛЕНИЕ ПРОЕКТАМИ С ПРИМЕНЕНИЕМ ГИБКОЙ ТЕХНОЛОГИИ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

А. В.Истомин
Кыргызско-Российский Славянский университет

В современном скоротечном и быстро изменяющемся мире для достижения успеха очень важно быть гибким и уметь адекватно реагировать на часто меняющиеся обстоятельства. Данная тенденция не могла не коснуться и сферы коммерческой разработки программного обеспечения. Зачастую бывает так, что при большом промежутке времени между формулировкой требований и внедрением, к моменту получения продукта заказчиком система уже не соответствует реалиям его бизнеса.

В данный момент существует множество подходов, базирующихся на так называемой гибкой (быстрой) парадигме в процессе разработки программного обеспечения (Agile Software Development) [1]. Причем часто бывает сложно отделить, чем же принципиально отличаются те или иные подходы друг от друга, и, что еще более важно, выделить их специфические черты. Наиболее ясный и формализованный подход – так называемое экстремальное программирование (XP – eXtreme programming) [2]. Однако, для того чтобы применять все практики данного процесса, необходимо иметь набор разработчиков высокого уровня, что в подавляющем большинстве случаев, особенно в условиях рынка труда Кыргызской Республики, невыполнимо.

В предисловии к своей книге Хенрик Книберг пишет: «есть и хорошая новость – я расскажу, как именно я практикую Scrum[1]... очень подробно и со всеми деталями. Однако и без плохой новости не обойдётся – то, что я расскажу – это всего лишь то, как практикую Scrum я. Это значит, что вам не обязательно делать всё точно так же. На самом деле, в зависимости от ситуации, я и сам могу делать что-то по-другому» [3]. Целью данной статьи также не ставилось пространное рассуждение об имеющихся трактовках быстрой разработки ПО, и, уж тем более, автор не пытается выдвинуть свою концепцию процесса. Задача данной статьи – поделиться опытом управления проектами с применением быстрой разработки программного обеспечения (ПО). Предприятие, на котором применялся процесс и о котором будет рассказано ниже, вело параллельную разработку нескольких коммерческих проектов по созданию ПО. Основной профиль предприятия – платежные системы с терминалами быстрой оплаты.

Итак, первый вопрос, на который необходимо ответить прежде, чем заниматься постановкой того или иного процесса разработки – чего мы хотим добиться? Для нас ответ был следующим: «Добиться выполнения тех задач, которые нужны заказчику, в срок и с максимальным пользовательским качеством».

Однако, сразу видно, что данный ответ порождает несколько вопросов:

1) Каким образом выяснить требования заказчика?
2) Каким образом убедиться в том, что выясненные требования соответствуют тому, что ожидает заказчик?
3) Откуда появляются сроки?
4) Каким образом добиться качества?

Попытаемся найти ответы на поставленные вопросы.

Каким образом выяснить требования заказчика? Опираясь на идеи, изложенные в [3], в каждом проекте была введена роль *куратора*. Куратор - это человек, сотрудник предприятия-подрядчика, который тесно взаимодействует с заказчиком, осуществляет сбор и анализ требований и при этом отстаивает интересы команды разработчиков. Этот человек является первым фильтром, через которые проходят требования заказчика и в последствии

[1] Scrum – одна из разновидностей процессов гибкой разработки ПО.

им же формализуются. Далее требования направляются команде разработчиков. Структура команды разработчиков представлена на рис. 1.

Рис 1. Структура команды.

От *куратора* требования попадают к *капитану* команды. Как правило, капитан – это более опытный разработчик, который совместно с куратором формирует бэклог (backlog) [3] итерации. Под бэклогом понимается конкретный набор задач для команды на итерацию. Чаще всего задачи формализуются в виде тикетов (tickets). Под тикетом понимается формат представления задачи, при которой она имеет обязательные поля, формат тикетов – понятие обсуждаемое, и, как правило, утверждается в конкретной команде, исходя из ее опыта и традиций (более подробно о процессе формирования тикетов, или же User Story можно почитать в [3]). Капитан совместно с куратором заносят тикеты в Issue-Tracker – автоматизированную систему для операций с тикетами[2], формируют и приоритезируют бэклог [3]. Приоритезация бэклога крайне важна. Что такое приоритезация? Это процесс, в результате которого куратор каждому тикету ставит условный коэффициент важности. О том, какие подходы в выборе коэффициентов существуют – можно посмотреть в [3]. Далее, когда бэклог сформирован и итерация началась, капитан занимается организацией процесса внутри команды, отслеживанием выполнения плана, ведением Burndown Chart'a (BDC) [3]. Burndown Chart – это диаграмма, на которой представлены два графика: накопления емкости команды, и решенных задач (по оси X откладываются дни – от 0 до n, где n – количество дней итерации, а по оси Y – затраченные и выполненные часы). Например, имеем емкость команды – 16 часов (2 разработчика при 8 часовом рабочем дне) в день. По окончании первого дня, предположим, не было закрыто ни одного тикета – соответственно график capacity (та емкость, которая была израсходована) перешел из точки (0,0) в точку (1, 16), а график burndown (то, что сделано) - в точку (1, 0). Во второй день был закрыт тикет, оцененный в 8 часов и тикет на 20 часов. В итоге capacity оказался в точке (2, 32) – так как в итерации уже на разработку потрачено 32 часа, а burndown «прыгнул» в точку (2, 28). Данная диаграмма приведена на рисунке 2.

[2] После создания тикет отдается на разработку, тогда разработчик закрепляет его за собой и статус у тикета становится «In Progress». Когда разработка по тикету заканчивается, разработчик ставит ему статус «Resolved». Далее, ответственный за тестирование переводит его в состояние «Reopened», если были обнаружены багги, или же в состояние «Closed», если тикет успешно прошел тестирование.

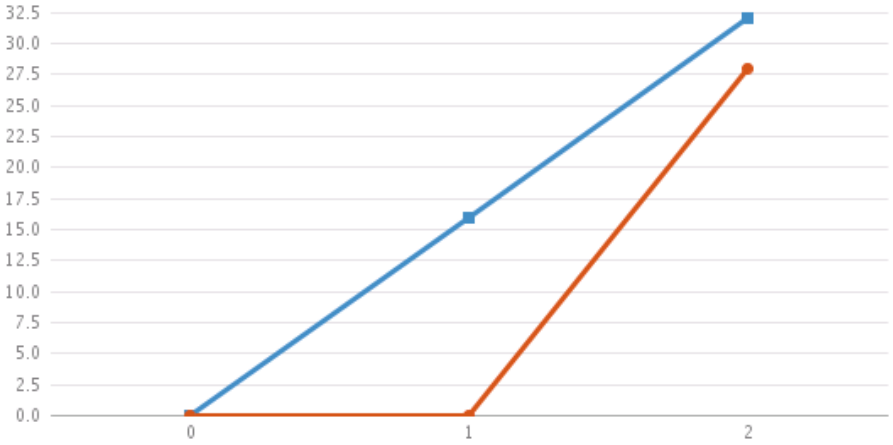

Рис 2. Пример BDC. Синим показан график capacity, оранжевым - burndown.

Таким образом, ясно, что если график burndown постоянно находится ниже capacity, то разработчики отстают от намеченного плана, в противном случае – опережают план. В нормальных условиях эти графики должны стремиться друг к другу (пример реального BDC см. на рис. 3).

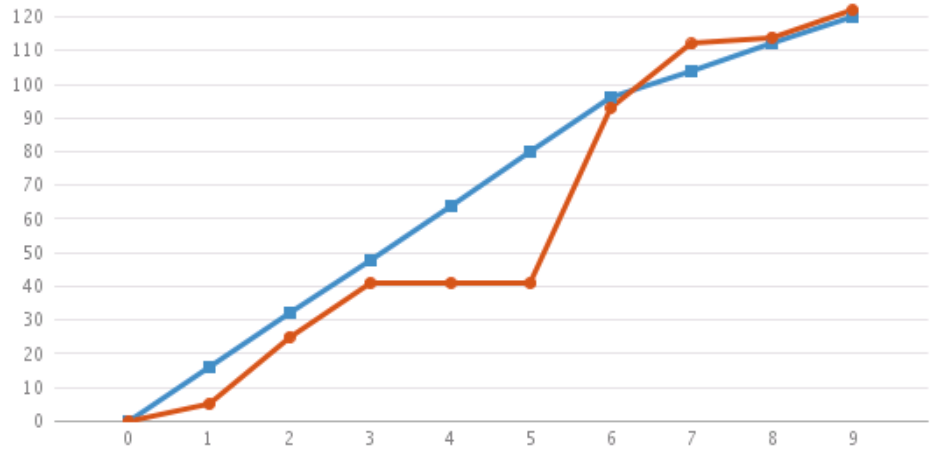

Рис 3. Пример реальной BDC одной из итераций проекта.

Опыт показал, что на капитана не следует планировать задач по разработке и включать его в емкость команды, он должен быть активной единицей, деятельность которой направлена на ограждение разработчиков от внешних воздействий, чтобы те, в свою очередь, могли быть полностью сконцентрированы на своих, разработческих[3], задачах в течение всей итерации [3]. В противном случае, на капитана запланирован ряд тикетов, по которым он может не уложиться в срок, выполняя свои задачи по организации работы в команде. Получается, что сложно отследить производительность капитана по написанию кода, и, поэтому, мы для себя решили в планах не учитывать разработческие задачи капитана. Помогает капитану *ОТК* – ответственный за качество – роль, которая была введена за неимением необходимости содержать отдел качества. Этот человек занимается тестированием закрытых разработчиками тикетов, а в свободное от тестирования время занимается внутрикомандными работами (создает билд-план, разрабатывает планы регрессионного тестирования и автоматизированного тестирования) или занимается написанием кода в парах[4] с разработчиками. ОТК также не учитывается в емкости по тем же

[3] Внимательный читатель обратит внимание, что на данной диаграмме график capacity не есть прямая, а это, в свою очередь, означает, что емкость команды в течение итерации менялась.

[4] Стоит отметить, что как таковое «парное программирование» у нас не практиковалось, в парах была возможность работать у ОТК и капитана в свободное от решения своих прямых задач время. За счет таких сеансов совместной работы достигалось усреднение опыта разработчиков и распределение знаний среди команды.

причинам, что и капитан. На самом деле то, что у команды, скажем, из трех разработчиков, есть два человека, занимающихся обслуживанием процесса написания кода, не принимая в разработке кода непосредственного участия – не проблема [4]. По опыту компании и по заметкам более опытных апологетов гибкой разработки [3] в команде для эффективной работы должно быть от 2 до 4 разработчиков при одном ОТК и капитане. Когда команда становится больше, видится целесообразным, что ее нужно раздробить.

При закрытии тикетов очень важно осознавать, что то, что понимают в формулировке задачи капитан и разработчик может быть совсем не то, что подразумевал под этим куратор, а учитывая то, что, в нашем случае заказчик вместе с куратором имели местом дислокации другую страну, а связь происходила по ICQ или Skype – такие случаи были вовсе не редки.

Итак, *каким образом убедиться в том, что выясненные требования соответствуют тому, что ожидает заказчик?* Практика показывает, что этому способствует ранний фидбэк (feedback) - отзывы заказчика по работе заказанного функционала системы [3]. Была придумана схема поставок разрабатываемых систем в тестовое окружение заказчика. После каждого закрытого и протестированного *ОТК* тикета осуществлялась так называемая альфа-поставка [6] – поставка, которая идет в тестовое окружение заказчика, но не может идти в production (то есть в «полевые» условия) и несет функцию ознакомительную, цель поставки - на раннем этапе выяснить погрешность в понимании требований. После того, как все тикеты итерации закрыты, протестированы и прошли альфа-поставку, происходит полное тестирование версии. Под полным тестированием версии понимается прогон всех тестовых планов тикетов, входящих в итерацию и приемочное тестирование функционала итерации.

Здесь необходимо отметить один ньюанс. В случае, если на интеграционной машине не работает автоматический запуск актуальных модульных тестов [5], сильно увеличивается риск «поломать» функционал, прямо не касающийся функционала итерации. Для этого было принято решение составлять и поддерживать планы регрессионного тестирования [6]. От итерации к итерации план пополняется новыми тестовыми планами тикетов итерации и планами воспроизведения уже отловленных багов[6]. В случае, когда планы регрессионного тестирования достаточно полные и покрывают большую часть функционала, мы решаем проблему: *каким образом добиться качества?* Однако, стоит отметить, что избежать появления багов невозможно! В связи с этим хотелось бы отметить вопрос: как строить процесс, если в течении итерации, которая должна быть закрыта для внесения дополнительных задач [3], выявляются баги? Для этого стоит ввести следующую классификацию бага:

- неверсионный[7] баг, некритичный – он отправляется в бэклог следующей итерации и исправляется в порядке своей приоритетности,
- неверсионный баг, критичный – если вызывает не сильно большие изменения в коде, то его решает ОТК, в противном случае, текущая итерация под угрозой срыва – вполне вероятно целесообразно ее остановить и бросить все силы на устранение бага,
- версионный баг – исправляется разработчиком, который его создал или же командой в сверхурочное время – тщательно выявляется путь его появления в production'e, с целью избежать подобного в будущем. Есть еще вариант – не ставить версию в production, если баг был замечен на этапе тестирования на стороне заказчика, однако, это тоже не лучший путь, так как, все же, чем чаще обновляется production, тем стабильнее и управляемее процесс разработки ПО [1].

Итак, мы добрались до самого, пожалуй, интересного и обсуждаемого в нашей стране [5] вопроса - *откуда появляются сроки?*

---

[5] Стоит отметить, что в нашем случае был не сильно высокий уровень подготовки разработчиков и TDD[5] особенно никто не владел, а потому говорить об автоматизированном юнит-тестировании сборки было абсурдно.

[6] Баг (англ. bug) – дефект, ошибка в программе.

[7] Под неверсионным багом понимается баг, сделанный раньше, чем в n-1 итерации, то есть не в текущей итерации(версии).

За формат итерации мы приняли трехнедельные сроки. На наш взгляд показалось, что это тот самый срок, за который команда успевает набрать приличную скорость, но при этом не успевает устать. В соответствии с этим требованием капитан с куратором до начала итерации кладут в бэклог самые приоритетные тикеты, суммарная оценка которых укладывается в 3 недели[8].

Но все же, спросит внимательный читатель, откуда же берутся оценки в тикетах? Однажды мы попробовали сыграть в игру Planning Poker, правила которой хорошо описываются в [3] – и мы приняли такой формат процесса оценки за правило. На наш взгляд такое игровое планирование позволяет наиболее лучшим образом распространить понимание задач итерации между членами команды и вывести наиболее точную оценку, учитывая даже разный уровень подготовки разработчиков в команде, что также особенно актуально [5].

Последнее, о чем хотелось бы сказать, это о расширяемости подобной схемы. В случае прихода нового сотрудника появляется новая роль: *стажер*. В этой роли сотрудник в зависимоти от своего опыта и потенциала может пребывать от нескольких дней до целой итерации. В это время сотрудник осваивается в команде, устанавливает и настраивает необходимое окружение (IDE разработки, сервер баз данных и т. п.) Также имелся успешный опыт дробления команды, когда количество разработчиков стало больше, чем допустимое в команде – команда поделилась на 2, причем *ОТК* занял роль второго капитана, стажеры стали полноценными разработчиками, а наиболее опытные разработчики заняли позиции ОТК.

Подводя итог всему вышесказанному, хотелось бы отметить те плюсы, которые дает применение гибких технологий в разработке программного обеспечения.

Во-первых, мы получаем программный продукт, отвечающий требованиям заказчика на данный момент времени. То есть заказчик в кратчайшее время получает тот функционал, который ему необходим.

Во-вторых, мы постоянно отслеживаем продвижение команды по намеченному графику и, так как итерации достаточно коротки – всегда можем узнать о проблемах раньше, чем произошла катастрофа.

В-третьих, мы тесно взаимодействуем с заказчиком, и потеря понимания между ним и разработчиками практически невозможна.

И, наконец, практика показывает, что командам, практикующим гибкий процесс разработки удается приблизиться к идеальному процессу, а именно, делать то, что нужно заказчику, в срок и с максимальным качеством [7].

Литература

1. http://agilemanifesto.org
2. Кент Бек. .Экстремальное программирование. Разработка через тестирование. - СПб: «Питер», 2003.- 380 с.
3. Хенрик Книберг. «Scrum и XP: заметки с передовой».- Enterprise Software Development Series, 2006.- 102с.
4. Фредерик Брукс. Мифический человеко-месяц или как создаются программные системы. Forbidden Reality, 1975.- 327с.
5. http://it.potcast.kg
6. Jones, Capers. *Software Quality: Analysis and Guidelines for Success.* Boston: International Thompson Computer Press, 1997.
7. Роберт Мартин. Быстрая разработка программ: принципы, примеры, практика: Пер. с англ. – М.: Издательский дом «Вильямс», 2004. –752 с.

[8] Более подробно о процессе формирования бэклога можно посмотреть в [3].