http://experience.openquality.ru/software-configuration-management/

Управление изменениями и кессонная болезнь проектов

Юрий Удовиченко | 13.02.2011

Юрий Удовиченко делится практическим опытом внедрения систем управления конфигурациями ПО. Какие виды деятельности относятся к «управлению изменениями», как выбирать и внедрять соответствующие инструменты, каких ошибок следует избегать – темы сегодняшней публикации.

- Видишь суслика?
- Нет…
- И я не вижу. А он есть! (с) ДМБ

Вступление

В рамках любого проекта особняком стоит управление конфигурацией продукта (**Software Configuration Management**, SCM). С одной стороны, такое управление есть, с другой стороны, немалая часть разработчиков не выделяет его как отдельную «дисциплину». Иными словами, управление незримо присутствует и в то же время не существует. Между тем, практики SCM используются повсеместно. Иногда сообща, иногда разрозненно, но – используются. Присутствует ли SCM в вашем проекте? Попробуем предотвратить метания на тему «да или нет» и перечислим основные направления деятельности под общим названием «**Управление конфигурацией продукта**». К этому списку мы будем возвращаться на протяжении всей статьи.

1. Управление запросами на изменение (bug tracking)
2. Управление исходным кодом (другое название: «контроль версий»)
3. Работа по сборке продукта (и это не только ваш любимый компилятор)
4. Управление рабочей средой: настройка окружения, его переменных и установленных библиотек – мало кого обошла стороной эта деятельность.
5. Работа по выпуску продукта: строгий техпроцесс и учёт выпускаемых «наружу» артефактов. Это важный отрезок, на котором не имеет значения, кому выдается продукт – своим тестировщикам или отделу SQA заказчика.
6. Развёртывание продуктов: на стороне клиента может быть множество подводных камней, их надо уметь обойти.

Знакомые слова, не правда ли? Пусть вас не вводит в заблуждение слово «продукт». Продукт – это не только дистрибутив, за который компания-разработчик получит деньги. Это любой артефакт, создаваемый командой на всём протяжении жизненного цикла продукта. Файл с исходниками, документ со спецификацией требований, тесты, бинарники для установки у клиента – всё это продукты. А под **конфигурацией** понимается набор версий, то есть, “срез” состояния всего продукта.

В общем и целом, главное назначение SCM – это **управление изменениями**, поступающими в проект, забота о качестве приложений. При чём здесь качество? Если перечисленные функции свести воедино, то образуется «подложка», фундамент, лежащий под всеми видами деятельности проекта. Соответственно, если он будет непрочным, здание проекта рухнет. Хорошо, если фундамент просядет где-нибудь в начале строительства, когда мы не приступали к возведению стен. А если мы уже людей туда заселяем? Вот и получается, что обеспечить качество продукта без продуманной организации техпроцесса не получится.

Выбор инструмента

Как и в любом процессе, здесь важны две равновеликие составляющие – **методики работы и инструменты для поддержки этих методик**. Одно без другого не имеет смысла.
Скажем, многие разработчики стоят перед выбором: «Какую систему контроля версий использовать – git или Mercurial?» При детальном рассмотрении выясняется, что речь идёт о команде, в которой распределённая модель работы просто не нужна. К примеру, условия работы с кодом диктуют постоянную синхронизацию результатов работы между всей командой. Или же команда продолжает покупать лицензии на Perforce, хотя его использование выродилось в линейное складирование версий, без ветвления.
Порой разработчики начинают выбирать систему отслеживания ошибок, требуя гибкость в настройке жизненного цикла, максимум возможностей по разделению прав, встроенную Вики и т.п. А потом выясняется, что в команде два человека, заводятся только простые тикеты (tickets) из 3-х состояний, разделение прав ещё несколько лет не понадобится, а документированием своей работы они как не занимались, так и не занимаются (так что Вики ни к чему).

В общем, выбор порой надуман, особенно на первых этапах работы. Однако чаще всего бывает обратная ситуация – внедрение методик и инструментов просто не поспевает за потребностями.

При выборе инструментов и политик, а также правильного времени их внедрения, уместна аналогия с **кессонной болезнью** – *эффектом, который возникает при слишком быстром подъеме на поверхность с большой глубины. Из-за быстрой смены давления из крови и тканей ныряльщика начинают выделяться мелкие пузырьки газов, обычно в них растворённых. Это приводит к серьезным последствиям для всего организма. То есть, всплыть получится, однако человек рискует просто погибнуть. Известны только два решения – постепенный многочасовой подъем или быстрый подъём с последующей акклиматизацией на поверхности, но уже в барокамере*.
Поскольку обычные проекты не избалованы бюджетом для покупки “барокамер”, будем рассматривать только первый рецепт.
Предположим, некоторый проект с серверной и клиентской частями начинается «на коленке», в глубине гаража или квартиры основателя проекта. Выбираются простые инструменты: **багтрекером** (bug tracker) служит лист в Excel, контроль версий реализован в традиционном Subversion, сборка проекта идёт простым нажатием кнопки в любимой среде разработки, а дистрибутив выдается в виде архива или заливается на сервер по FTP.
Проходит время, проект зреет, появляются новые люди. Листик в Excel ещё удается расшарить по сетке, SVN (Subversion) пока хорошо справляется с **коммитами** (commit) в **транк** (trunk, по умолчанию – главная ветка в Subversion), ветки пока не нужны. Поскольку продукт теперь собирается под 2-3 разные операционные системы (клиентура

растёт), приходится использовать пару разных сред разработки и на каждой новой машине тратить немало времени на настройку всех библиотек. Да и сама компиляция и линковка теперь – задача непростая, каждый раз вылазят специфичные для среды ошибки. При наличии нескольких человек трудно понять, кто и когда залил на сервер новый релиз и кто готовит инсталляционный пакет для клиентов. Проблемы пока решаются, ибо команда всё ещё невелика, и все находятся рядом. Однако давление уже изменяется: проект "всплывает", последствия начального выбора процессов и инструментов скоро начнут «пузыриться».

Первый пузырёк появляется в системах контроля версий и багтрекинга. Всё чаще люди начинают непродуктивно тратить время на коммит новой версии на транк – ведь приходится по 2-3 раза делать update перед очередным коммитом. Классическая ситуация: три человека меняют один файл, первый делает commit, второй и третий, прежде чем сделать то же самое, вынуждены делать update. Далее кто-то из них двоих делает свой коммит, в результате чего третий вынужден опять упражняться в слиянии (merge) исходников. Начинается использование веток: сначала робко, для больших подсистем, потом всё чаще. При этом кто-то продолжает скидывать **дельту** (изменения в рабочем продукте) прямо в транк – ведь единой договорённости об использовании веток пока нет. Всё чаще возникают конфликты слияния, потеря дельты – сказывается разница в подходах между участниками команды. Всё больше ошибок при компиляции и тестировании появляется из-за несогласованности внесения изменений – слабо контролировалось вхождение новой дельты (новой функциональности или багфиксов). Листик Excel разросся, стало сложно управлять поступающими запросами, уже непонятно, кто за какие изменения отвечает. К примеру, кто-то решил вместе с очередным багфиксом сделать рефакторинг и «размазать» его на 4 последовательных коммита. Разумеется, это переплелось с плановыми изменениями – в итоге билд поломан, пытаемся выяснить, на каком основании что было сделано. Иными словами, "пузырёк" появляется там, где происходит наибольшая активность, а стало быть – наибольший перепад давлений.

Следующий пузырь появляется при сборке проекта и работе с настройками среды. Разные среды, компиляторы, настройки окружения – всё чаще появляются специфичные ошибки компиляции и линковки. Да тут ещё и одна из сторонних библиотек поменялась – всем разработчикам нужно ее установить и проверить на совместимость с их модулями. При этом текущая стабильная версия системы (она стоит у заказчика), использующая предыдущую версию библиотеки, требует сопровождения. А значит, приходится её переустанавливать и менять ключи компиляции каждый раз, когда делаются багфиксы. А если учесть, что теперь клиенту, работающую под Windows, надо устанавливать новую dll и регистрировать дополнительный COM-объект, становится понятен нервный смех того, кто делает инсталляционные пакеты. И вот уже очередной перепад давлений – политики и инструменты для сборки проекта и управления рабочей средой просто не справляются. Под словом "**политика**" (policy) в этом контексте обычно понимают свод договорённостей, правил и используемых практик, которые принимаются всей командой.

Худо-бедно, учась на своих ошибках, сменили часть инструментария (в первую очередь, систему отслеживания запросов), договорились о правильном взаимодействии при работе с ветками, написали скрипты для запуска отстройки, начали использовать CMake для поддержания кроссплатформенности. Но продукт не стоит на месте, он растёт, а значит продолжает увеличиваться перепад давлений. Вот уже к работе подключилась удаленная команда тестировщиков. Возникла необходимость разделения прав в системе отслеживания ошибок, нужны новые способы быстрой настройки рабочего окружения, надо куда-то и в каком-то порядке выкладывать релизы для прогона тестов разной интенсивности и глубины. И снова “пузыри”…

В нашем примере очень показательно и естественно добавление удаленной команды разработчиков. Это большое испытание для всего проекта, не говоря уже о его SCM-инфраструктуре. Надо решить вопрос распределенной работы с исходниками, понять кто, где и в какой момент интегрирует изменения, выпускает очередной релиз, куда его положить, как отдать на тесты и получить их результаты для дальнейшего закрытия «тикетов». В общем, очередной “подъём наверх” ставит всё новые задачи с точки зрения организации инфраструктуры. И редко кто бывает к ним заранее готов – всё меняется под нужны момента и редко кто заранее планирует все изменения.

Однако, предугадать это безобразие вполне возможно. Главное – нужно понимать, что все используемые инструменты и практики между собой увязаны, хотя и ортогональны друг другу. *Всё как в физике – есть векторы сил, которые действуют в разных направлениях и с разной величиной, однако при сложении получается конкретный вектор, который направлен вперёд*. Напомним, что есть 6 основных направлений работы (рассмотрены выше), с которыми совершенно точно предстоит столкнуться. Предупреждён – значит вооружён.
Нельзя отвергать и использование “барокамеры”, а именно – единовременных вложений ресурсов и времени на улучшение инфраструктуры. В этом случае команда целенаправленно вкладывается в то, чтобы улучшить свою работу. На это требуются деньги, однако результат можно получить относительно безболезненно и, если постараться, то быстро. Поэтому можно считать подобное решение альтернативой “постепенному подъёму”.

На что ориентироваться при выборе инструмента? Итак, начался проект. Первое, с чем придётся столкнуться ещё до начала написания кода – запросы на изменения. Это **пункт 1** в списке, рассмотренном в начале статьи. Даже когда единственным рабочим продуктом будет спецификация требований, её разработчикам уже нужен инструмент для отслеживания всего того, что происходит с требованиями, какие правки в них вносятся и зачем. Что уж говорить об исходном коде. Отслеживать изменения нужно научиться сразу – и научиться делать это качественно. Так что не экономим ресурсы, а постоянно оцениваем: хватает ли нам системы трекинга изменений и хватит ли на ближайшие пару месяцев.

Тут возникает проблема выбора, ведь система bug- или task-трекинга – это один из любимейших велосипедов программистов. За такие системы берутся многие, при этом доводят до презентабельного вида – сотни, и наибольшей популярностью пользуются десятки систем – от простых todo lists с галочками до сложных и безумно больших и гибких систем управления проектами. К слову сказать, разделение между bug-, task- и issue-tracking достаточно условно, однако task-трекеры ориентированы на более широкую аудиторию, не только разработчиков. Здесь очень велик риск внедрить одно, а через несколько месяцев захотеть уже сильно другое. Поэтому надо или быть заранее готовым сменить через год-два систему трекинга, или каким-то образом предвидеть направление роста команды и выбрать именно то, что не помешает маленькой команде и поможет – команде большой.

Не стоит забывать и о политиках использования. Ведь можно бесконтрольно плодить и закрывать запросы на изменения, а можно создать рабочую группу (change control board, группа контроля за изменениями), которая будет вести работу по приоритезации и отслеживанию выполнения этих запросов.

На данный момент мои фавориты – Redmine, eTraxis, Basecamp – каждый для своих классов задач. Basecamp, по сути, это сильно расширенный todo list. eTraxis – гибкая система трекинга чего угодно. В нём открытый код, и я, участвуя в его внедрении на крупном проекте, даже написал несколько улучшений и исправлений кода. Вообще, очень гибкая штука – горячо рекомендую. Ну а на Redmine смотрите, если нужна целая система управления проектами – для этих глобальных целей он хорошо подходит.
Следующий по значимости класс систем – контроль версий. Это **пункт 2** в нашем списке SCM-задач. Такая система также выбирается в самом начале, поскольку рабочие продукты начинают появляться практически с первого дня работы. Тут ситуация отличается от систем трекинга. С одной стороны, инструментов не так много. Из бесплатных (open source) систем лидируют Subversion (SVN), git, Mercurial, среди платных – MS TFS, Perforce, IBM Rational ClearCase (и его потомки). Небольшой выбор компенсируется тем, что этими инструментами можно пользоваться по-разному.

Можно использовать один и тот же SVN годами, однако в начале проекта работать только на транке, а уже через год вырастить увесистое дерево веток и меток. Так что с этим классом инструментов просто необходимо периодически пересматривать политики и практики их использования. При этом ничто не является догмой – вы можете сейчас активно ветвиться, а потом постепенно перейти к системе непрерывной интеграции (continuous integration), исповедующей частую интеграцию на транке. К примеру, немало копий, поломано при обсуждении того, как надо ветвиться (и надо ли вообще). Или священная война последних 3-4 лет – какие системы лучше, централизованные или распределенные. Что тут сказать, надо исходить из потребностей.

Моя слабость – это IBM Rational ClearCase, но только не UCM, а его базовый функционал. Мощнейшая штука, несправедливо загубленная маркетологами IBM. Кстати, не так давно появился неплохой продукт PlasticSCM, одновременно имеющий распределённую природу и частично использующий идеологию ClearCase. С интересом слежу за его развитием.

Следующую немаловажную нишу занимают системы поддержки построения продуктов – это **третий пункт** нашего "хит-парада". Ниша эта небольшая, однако без неё многие проекты немыслимы. Сюда входят как системы для описания кроссплатформенной компиляции и линковки (CMake), так и системы полного цикла сборки (Maven, ant). Общая цель – сделать построение продукта как можно более прозрачным. Зачастую такие системы специфичны для конкретного языка или технологии, а их выбор приходится на более поздние этапы развития проекта. Чаще всего их начинают внедрять когда ресурсы ручной сборки уже выработаны. Однако правильные архитекторы и CM-инженеры думают о них уже в самом начале проекта :)

Это же описание справедливо и для систем управления рабочим окружением, выпуском и развертыванием продуктов. В нашем списке они выделены под**пунктами 4, 5 и 6**, однако потребность в них возникает практически в одно и то же время. Зачастую продукт сильно эволюционирует до того как доходит очередь до этого класса систем. Это правильно – только реальные потребности покажут, что и когда надо внедрять для подобных задач. Мало научиться строить продукт – надо его правильно выдать пользователям (в том числе, тестировщикам), убедиться, что при использовании или тестировании будет всё настроено так, как задумывалось разработчиком, и нам не будут лететь сообщения об одних и тех же ошибках каждый раз после выхода нового релиза. Подобные инструменты эволюционируют обособленно и, как правило, по времени ближе к середине проекта, когда есть уже что показать тестировщикам. Обычно же начинают с того, что просто выкладывают бинарники в расшаренную папку в сети, приложив release notes, где описано, какие изменения внесены и как настраивать окружение. Тем и ограничиваются долгое время.

Несколько слов о литературе по SCM. Книг на русском языке практически нет. Есть масса онлайн-материалов по конкретным инструментам, много HowTo, руководств, но не книг. Обзор имеющейся литературы можно найти здесь.

В целом, важно не какой инструмент выберешь, а как его будешь использовать. То есть, первичны методы, практики, политики, а не инструменты. Инструмент выбирается под задачу, и не грех менять инструмент по мере необходимости.

Типичные ошибки и заблуждения

Первая типичная ошибка – не использовать контроль версий. Лечится быстро, но болезненно. Без контроля живут до первой ошибки, которую не смогли поправить откатом к предыдущей работающей версии системы. Никакие бэкапы не сравнятся с системой контроля версий.

Аналогичная ситуация – отслеживание запросов на изменение. Кто-то ограничивается перепиской по email, и долгое время тем и счаслив. Лечится не очень быстро. Скачок происходит, когда количество обозримых задач и контроль за их исполнением просто не умещается в голове, а поиск по почте мало помогает. Вот тогда и начинаются поиски подходящей системы. Впрочем, тут появляется другая ошибка – как я уже писал, начинают писать что-то своё, тогда как готового – просто навалом. Но тут уж трудно кого-то в чём-то убедить.

Ещё одно заблуждение (скорее, позиция в священной войне), состоит в том, что распределённая модель может полностью заменить модель централизованную. Рискую навлечь на себя гнев апологетов git и Mercurial, однако считаю, что это далеко не так. Будучи активистом антиброуновского движения :), полагаю, что централизованная модель может и даже должна использоваться в проектах, имеющих жёсткую иерархию работы с изменениями. Перед глазами собственный пример. Я имел удовольствие работать в крупном проекте, распределённом между странами и часовыми поясами. У нас, бывало, проходила не одна неделя, прежде чем изменения попадали от разработчика в продукт, отгруженный “наружу”. При этом участники команды должны были иметь возможность смотреть изменения от любого разработчика вне зависимости от его местонахождения, времени суток и доступности его копии изменений. У нас использовался ClearCase с надстройкой Multisite, позволявшей централизованным хранилищам в локальных командах реплицировать между собой все нужные изменения. Таким образом я, находясь в России или США, мог видеть код, написанный в Индии или Италии – он становился доступным в течение получаса (максимум). С децентрализованной моделью это было бы трудно.

К заблуждениям отнесу и уже упоминавшийся отказ от использования веток. Я не имею виду непрерывную интеграцию, которая принципиально строится на использовании единой ветки для разработки – есть аргументы как “за”, так и “против” этого метода. Речь идет о непонимании механизма работы. Кто-то считает, что наличие веток отнимает кучу дискового пространства – это неверно, так как внутреннее представление большинства систем достаточно оптимально и там, как правило, хранятся только изменения, но никак не полная копия исходников для каждой ветки. Кого-то пугает процесс слияния изменений после окончания работы на ветке. Страх этот идёт отчасти из-за того, что в самой популярной на сегодняшней день системе контроля версий – Subversion – ветвление и слияние действительно сделаны… скажем так, не совсем хорошо :) В других системах это делается значительно проще, так что страхи пора забывать.

Кстати, встречал упоминания о том, что кто-то просто удаляет ветки после того, как дельта использована для интеграции, и ветка уже не нужна. Объясняется это какими-то полурелигиозными соображениями вида “она будет мешать” или “она же занимает место”. Система контроля версий изначально создана для того, чтобы хранить все версии, сколько бы их не было – под это ветки “заточены”. Нередки случаи, когда вернуться к старым изменениям приходится и через пару месяцев. Если база разрастается совсем уж до неприличия и тормозит всю работу, можно слить старые изменения в архив, но ни в коем случае их не удалять.

Нельзя не сказать и о метках – незаслуженно мало используемой возможности систем контроля версий. Метки позволяют работать с именованными версиями. Например, интегратор проекта слил воедино все изменения пары команд, которые параллельно работали над изменениями, получил тот срез, который можно считать стабильным. Он вешает на него метку, и теперь любой участник команды может взять этот срез путём извлечения одной-единственной метки. Чем это отличается от обычного номера ревизии? Тем, что метку могут "перевесить", если вдруг выяснится, что произошел сбой и стабильной считается уже другая ревизия. Кроме того, не во всех системах контроля есть наборы изменений (changesets), и для них метки – это единственный способ каким-либо образом идентифицировать конфигурацию без перечисления всех версий всех элементов. В общем и целом, метки – полезная штука, от которой не надо отказываться.

Будущее SCM

Судя по англоязычным публикациям, будущее за системами класса Application Lifecycle Management (ALM). Это инструменты, соединяющие в себе функции контроля версий, поддержки билда, и, самое главное, сбора требований, отслеживания запросов на изменения, поддержки тестирования и оценки его результатов. Этакие швейцарские ножи. Уже сейчас существует некоторое количество таких продуктов. К примеру, Microsoft Team Foundation Server (MS TFS) – думаю, он в представлениях не нуждается. Чуть менее известно семейство IBM Rational Team Concert, в частности IBM Rational Jazz. Судя по описаниям и отзывам, команда IBM критически перетрясла разработки прошлых лет и оставила самое лучшее. Также на слуху AccuRev и Neuma.
Заметьте, все решения платные. Единственная альтернатива, которая приходит в голову – это сборка воедино бесплатных решений с открытым кодом. Многие системы багтрекинга имеют возможность интеграции с системами контроля версий. В общем, есть все предпосылки для объединения, однако, как правило, дальше объединения трекеров и контроля версий дело не заходит. Полные интегрированные системы если и существуют, то, как правило, это результат работы отдельных компаний или коллективов: и дальше внутреннего использования они не выходят. Думаю, здесь мы ещё увидим немало интересного.