Introduction to Continuous Integration or Stone Soup

Andrey Satarin
Customized InformSystems
www.custis.ru
email: asatarin@custis.ru

Abstract

Continuous integration in software development process promises many advantages: fast defect discover, integration problems remove, fewer numbers of defects [1,2]. With more detailed consideration its turned out that this practice depends on others, such as unit testing, coding standard etc. Many of expected advantages are not realized without using such additional practices. Paradox situation occurs, where its unknown does continuous integration have independent value or all value comes from "foreign" methods. Isn't where any fraud, when on the pretext of continuous integration adoption one tries to use other engineering practices advantages? Perhaps, continuous integration is "stone soup", all ingredients are well-knows, but now served together with another title. In this article we try to show that this is not true, and continuous integration has its own value. This value considerably lower than synergetic effect from several practices, but adoption and use costs are much lower too. Adoption of "bare" continuous integration could also be the first step to many others effective development technologies.

***Keywords**: continuous integration; quality, agile development.*

Введение в непрерывную интеграцию или каша из топора

Андрей Сатарин
Заказные ИнформСистемы
www.custis.ru
email: asatarin@custis.ru

Аннотация

Использование непрерывной интеграции в процессе разработки программного обеспечения обещает много преимуществ: быстрое обнаружение ошибок, устранение проблем интеграции, меньшее число дефектов [1,2]. При более подробном рассмотрении, оказывается, что эта практика сильно зависит от других, таких как модульное тестирование, стандарт кодирования и т.д. Множество ожидаемых преимуществ не реализуются без использования этих дополнительных практик. Складывается парадоксальная ситуация, когда не ясно, имеет ли непрерывная интеграция независимую ценность или вся ценность обусловлена только «сторонними» методиками. Нет ли здесь обмана, когда под предлогом внедрения непрерывной интеграции пытаются использовать преимущества других инженерных практик? Возможно, непрерывная интеграция представляет собой «кашу из топора» , все ингредиенты которой давно известны, но теперь поданы вместе под другим названием. В данной статье мы пытаемся показать, что это не так, и непрерывная интеграция имеет свою ценность. Эта ценность существенно ниже, чем синергетический эффект от нескольких практик, но и затраты на внедрение и использование существенно ниже. К тому же, внедрение «голой» непрерывной интеграции может служить и первым шагом к многим другим технологиям эффективной разработки.

***Ключевые слова**: непрерывная интеграция; качество, гибкая разработка.*

1. Введение

Непрерывная интеграция (*continuous integration*, далее НИ) первоначально была создана как одна из практик экстремального программирования. Моментом создания считается приблизительно 2000 г., когда была написана первая версия статьи Мартина Фаулера [1]. С того времени она развивалась и изменялось понимание того, как она должна взаимодействовать с другими практиками гибкой разработки.

Образцом современного понимания НИ можно считать недавно вышедшую книгу Пола Дюваля [2]. В этом достаточно большом и подробном труде выделяется несколько составных частей НИ:

- НИ баз данных;
- непрерывное тестирование;
- непрерывную инспекцию кода;
- непрерывное развертывание;
- непрерывная обратная связь.

Все эти части подробно описаны и показано как они влияют на процесс разработки. В таком понимании практики НИ можно указать один недостаток — она не является самостоятельной и основная часть выгод получаемых от нее происходит не из нее самой. Выгоды появляются от удачного комбинирования данной практики с другими, такими как: модульное тестирование, автоматическое приемочное тестирование, автоматизированные инспекции кода (рис. 1). С одной стороны хорошо, когда много разнообразных практик вместе дают синергетический эффект, но с другой стороны, это поднимает входной порог использования этой технологии. Т.е. для использования НИ нужно уже иметь модульные тесты, стандарт кодирования и систему его автоматической проверки и т.д. Эта ситуация схожа с описанной в русской народной сказке «Каша из топора» [3]. Постулируется огромная польза от внедрения и использования НИ в проектах разработки ПО, но не уточняется, что большая часть этой пользы происходит не из самой практики, а из ее удачной комбинации с другими сильными методиками. На самом деле, описанные выше условия не являются необходимыми для использования НИ, можно получить выгоду от самостоятельного внедрения НИ, с минимальным привлечением сторонних активностей. В данной статье обсуждается именно такой «упрощенный» способ.

2. Зачем это нужно?

Может показаться что «урезанный» вариант НИ, не соответствующий последним достижениям в данной области, бесполезен и только будет отнимать время на внедрение и поддержку. Но это не так, есть круг проблем, которых можно просто и эффективно решить при помощи НИ, не отягощенной другими практиками. Также внедрение НИ, отдельно от других практик, может быть приемлемо для организаций, процесс разработки которых далек от гибких (agile) методологий или для организаций, которые только встают на этот путь. Проблемы, которые на наш взгляд, можно решить внедрением «голой» НИ, описаны в следующем разделе.

2.1. Проблемы

В целом эти проблемы можно разделить на два вида, те, с которыми сталкиваются тестировщики и те, с которыми сталкиваются разработчики. Для других участников процесса разработки практика НИ также является полезной, но это не так явно выражено.

Любое тестирование начинается с получения и развертывания одной из версий ПО. Тестировщики, как правило, не могут собрать проект самостоятельно, для этого у них нет знаний, инструментальных средств (IDE), да и просто навыков. Но любое тестирование начинается с получения готовой к использованию бинарной сборки ПО и его развертывания на тестовом стенде. Для разработчиков, наоборот, собрать проект не составляет труда, эту операцию они проделывают часто и много. Кажется, это и есть решение проблемы. Нет. При передаче версии от разработчиков могут возникнуть сомнения в целостности передаваемой сборки:

- Из какой версии исходных кодов она была собрана?
- Какие ключи компиляции и настройки среды окружения применялись?
- Не были ли внесены в код изменения, не отраженные в системе контроля версий?
- Были ли использованы правильные версии внешних библиотек при сборке?

На эти вопросы ответить не так просто, а проконтролировать каждого разработчика — еще сложнее. Многие скажут, что для сборки, развертывания и передачи ПО в тестирование должна существовать особая роль — инженер по сборке (*build engineer*), или даже целое подразделение сотрудников, но в небольших проектах такое просто невозможно. Конечно, эту

роль может исполнять кто-то из членов команды, и, на первый взгляд, это кажется разумным. На самом деле, это лишь способ скрыть проблемы, но не избавиться от них. Возникнет консервация уникального знания и навыка в голове одного человека и связанные с этим риски:

- Что будет, если работник, ответственный за сборку и развертывание, уволится или заболеет?
- Какие есть гарантии, что он не сделает ошибки при очередной сборке или развертывании?

Можно идти путем регламентов, которые контролируют окружение и процедуру сборки, но такие регламенты сложно исполнять и еще более сложно поддерживать. Программисты не могут заниматься исполнением желаний тестировщиков: собирать проект по требованию, жестко контролировать окружение, в котором происходит сборка. Это тупиковый путь.

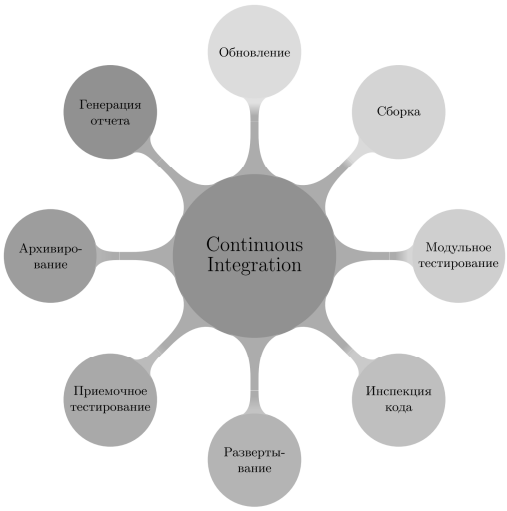

Рисунок 1. Составные части полного процесса непрерывной интеграции.

Сами разработчики тоже часто являются источником проблем. Такие проблемы могут быть охарактеризованы следующими фразами:

- «На моей машине все собирается и работает!»
- «На последней сборке, ушедшей в тестирование, используется устаревшая версия библиотеки superlib».
- «Я не могу работать, потому что проект не собирается. Я не знаю, когда все сломалось».

Особенно хороша первая проблема, вокруг этой фразы («It works on my machine!») даже возникла субкультура [4], но разработчики все равно продолжают повторять ее снова и снова. В борьбе с этим злом НИ особенно хороша.

Кому-то эти проблемы могут показаться смешными или надуманными, тем не менее, в разработке такое часто случается. Раннее обнаружение этих проблем экономит много времени и денег.

В итоге получается, что есть некий объем работ (сборка, развертывание и т.д.), который должен выполняться для того, чтобы контролировать (в некоторой степени) состоянии производимого ПО. Практика НИ призывает избавиться от этих рутинных операций посредством их автоматизации и передаче машине.

3. Решение

Автоматизация здесь — это в первую очередь автоматизация сборки, развертывания, обратной связи. Все эти автоматические процедуры связываются воедино на специальной интеграционной машине, а управление процедурами осуществляет сервер интеграции. Базовая схема организации процесса интеграции показана на рис. 2. На схеме выделены следующие этапы:

- Обновление исходного кода;
- Сборка;
- Развертывание проекта;
- Архивирование бинарных файлов;
- Генерация и публикация отчета.

Это практически минимальный набор этапов, при котором можно говорить о процессе НИ. В некоторых случаях можно убрать этап развертывания, если для программы этого не требуется. Остальные этапы, безусловно, необходимы. Рассмотрим подробно, что происходит на каждом из указанных этапов. Вначале сервер интеграции обновляет дерево исходных кодов проекта до последней свежей версии, это можно делать как по расписанию, так и при каждом изменении исходных кодов. Далее идет сборка. Сборка на интеграционном сервере отличается от сборки на локальной машине разработчика. Чем они отличаются и зачем? Современные сервера интеграции позволяют передавать метку (номер версии) в качестве параметра сборки. Рекомендуется включать этот параметр на видном месте в результирующий продукт, например, в информации о программе, которую может посмотреть пользователь. Кроме того, окружение на сборочной машине контролируется более жестко. После этого, готовый продукт разворачивается в окружении. Важным моментом является, какое именно окружение использовать для развертывания в данном случае. Один из авторов практики НИ Мартин Фаулер советует делать это в промышленном окружении [2].

После развертывания идет этап архивирования, на котором происходит сохранение результатов сборки для дальнейшего использования.
Таким использованием может быть тестирование, демонстрация, воспроизведение проблем в ранних версиях. Кроме того на данном этапе может происходить пометка кода тегом в репозитории, для того, чтобы можно было всегда повторить данную сборку. Обычно тег совпадает с номером версии, ранее сгенерированным сборочным сервером.
После того, как все сделано, наступает этап генерации отчета и сервер оповещает разработчиков о результатах. Оповещение можно разделить на два типа: активное и пассивное. К первому относится email, системы мгновенных сообщений, мониторы. Второй тип — это публикация на веб или файловом сервере.

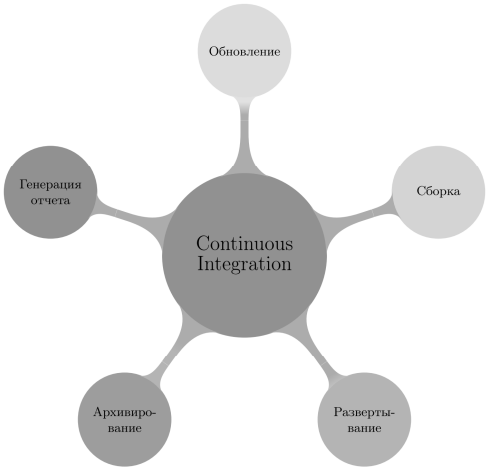

Рисунок 2. Составные части *базового* процесса непрерывной интеграции.

4. Пример

Приведем пример использования описанного подхода «голой» непрерывной интеграции. Примером будет служить проект разработки веб приложения на платформе java.
В качестве средства управления версиями использовался Subversion, который поддерживается большинством широко распространенных серверов сборки. Сами средства сборки стандартны для платформы java — это ant и maven. Такая комбинация с одной стороны предоставляют большие возможности при управлении сборкой проекта в целом, а с другой стороны позволяет просто автоматизировать смежные с компиляцией этапы.
Более нестандартной является реализация этапа развертывания. В первую очередь из-за того, что стандартного механизма развертывания для различных серверов приложений java не существует. Мы выбрали достаточно простой способ развертывания. Развертывание происходит на том же сервере, что и сборка, поэтому возможно простая подмена файлов приложения. После такой подмены сервер приложений (в нашем случае Resin) сам производит развертывание приложения.
К сожалению это не все что нужно сделать для запуска веб приложения, необходимо так же развернуть свежую версию базы данных. Свежую, значит ту, которая находится в репозитории, а где ее еще хранить? Для этой проблемы так же нет стандартного решения. Изначально мы пошли по очень простому и неправильному пути — стали хранить полный дамп базы в репозитории. Недостатки данного подхода очевидны: невозможность простого сравнения версий, неудобство работы с бинарным файлом. Гораздо более прогрессивным оказался метод хранения описания базы в SQL. Все современные средства работы с базой позволяют просто и удобно проводить экспорт таких скриптов. Такой подход к хранению структуры базы оказался существенно более удобным и не имеет указанных выше недостатков.
Последние этапы процесса непрерывной интеграции: архивирование и генерация отчета легко реализуются средствами практически любого специализированного сервера.

5. Преимущества и недостатки подхода

Выгода первая — избавление от рутины. Не надо объяснять, что профессионалы не любят рутину, кроме того, время людей становится все дороже, а время машин все дешевле.
Выгода вторая — простота и повторяемость. Кто угодно — любой участник проекта: тестировщик, аналитик и т.д. может собрать и запустить проект, потому что все эти операции автоматизированы. Совершить ошибку при этом невозможно.
Выгода третья — незаметность. На первых этапах использования большинству участников разработки нет необходимости что-то менять в их работе. Сравните это с такими «тяжелыми» практиками как разработка через тестирование (TDD), где разработчику надо практически поменять мировоззрение или внедрение автоматического приемочного тестирования, где тестировщику предлагается писать код. Если изменения в работе людей практически отсутствуют, они не будут сопротивляться внедрению и использованию НИ.

Как уже отмечалось, при комбинировании НИ с другими практиками получается синергетический эффект, но этот случай за пределами рассмотрения данной статьи.
Есть ли недостатки у практики НИ? Безусловно, но они не так значительны как преимущества. Например, к недостаткам относят необходимость иметь выделенный сервер, тратить время на поддержку работы этого сервера. Первый аргумент с каждым днем все слабее и слабее, железо стоит все меньше и меньше по сравнению с временем разработчика. Относительно второго аргумента можно сказать о нашем опыте. Время на поддержание сервера совершенно незначительно по сравнению со временем на обнаружение проблем другим способом и запоздалое их устранение.

6. Заключение

Практика НИ тесно переплетена с другими практиками гибкой (agile) разработки: модульное тестирование, приемочное тестирование, стандарт кодирования, но она может применяться и без них. Мы не отрицаем преимуществ этих практик, особенно в комбинации с НИ, но их внедрение — отдельная задача. Внедрить несколько таких «ресурсоемких» технологий сразу невозможно, как бы ни призывали различные гуру. Это можно сделать только по частям. Например, сначала внедрить «голую» НИ, а затем к ней постепенно присоединять другие методики. При этом необходимо последовательное «наращивание» шагов включаемых в сборку.
Т.о. непрерывная интеграция объединит вокруг себя несколько других инженерных практик, и благодаря этому может стать центром вашего процесса разработки.

7. Литература

[1] Мартин Фаулер (Martin Fowler) “Continuous Integration”
http://www.martinfowler.com/articles/continuousIntegration.html
[2] Duvall, Paul and Matyas, Steve and Glover, Andrew “Continuous Integration. Improving Software Quality and Reducing Risk”, Addison-Wesley, 2007.
[3] «Каша из топора», русская народная сказка http://www.skazki.org.ru/view.php?id=7073 см. так же http://en.wikipedia.org/wiki/Stone_Soup
[4] “It works on my machine!”
http://www.google.com/search?q=It+works+on+my+machine!