Раздел 4. Организация процесса разработки программного обеспечения

Что такое CI?

Процесс интеграции программного обеспечения — далеко не новая проблема. В проекте, выполняемом одним человеком с немногими внешними зависимостями, интеграция программного обеспечения — не слишком существенная проблема, но при увеличении сложности проекта (даже если в него просто добавлен еще один человек) возникает насущная потребность в интеграции и проверке слаженной работы компонентов программного обеспечения, причем заранее и ***часто***. Дожидаться конца проекта для проведения интеграции и выявления всего спектра возможных ошибок — неразумно и к тому же не способствует качеству программного обеспечения, а зачастую даже приводит к удорожанию и задержке сдачи проекта. Непрерывная интеграция снижает подобные риски.

В своей популярной статье “Continuous Integration”

Мартин Фаулер описывает CI так:

... практика разработки программного обеспечения, когда участники группы осуществляют частую интеграцию своих работ. Обычно каждый человек проводит интеграцию по крайней мере ежедневно, что приводит к нескольким интеграциям в день. Для максимально быстрого обнаружения ошибок каждая интеграция осуществляется автоматизированно (вместе с проверкой). Многие группы находят, что данный подход позволяет значительно уменьшить проблемы интеграции и способствует более быстрой разработке программного обеспечения.

А это означает следующее:

- ✓ все разработчики выполняют закрытое построение на собственных рабочих станциях перед передачей кода в хранилище с контролем версий, для гарантии того, что внесенные изменения не приведут к ошибке при интеграционном построении;
- ✓ разработчики обновляют свой код в хранилище с контролем версий ***по крайней мере*** один раз в день;

- ✓ интеграционное построение осуществляется несколько раз в день на выделенной для этого машине;
- ✓ при каждом построении проводится 100 % проверок;
- ✓ создаваемый продукт (например, файл WAR, сборка, исполняемый файл и т.д.) пригоден для функциональной проверки;
- ✓ исправление ошибок имеет самый высокий приоритет; часть разработчиков просматривают отчеты, созданные в ходе построения, стандарты программирования и отчеты анализа зависимостей, в поисках областей для усовершенствования.

Регулярный запуск процесса «непрерывной интеграции» позволяет получить работоспособные выпуски, функциональные возможности которых наращиваются при каждом выпуске... Получаемые при этом промежуточные отчеты позволяют руководству отслеживать прогресс и качество, а следовательно, упреждать, выявлять и своевременно устранять риски на постоянной основе. С появлением XP и других гибких (Agile) методологий, а также практик, рекомендуемых CI, люди начали воспринимать концепции не только ежедневного, но и “непрерывного” построения. Практика CI продолжает развиваться. Вы найдете ее элементы практически в каждой книге по XP. Зачастую при обсуждении практики CI ссылаются на оригинальную статью Мартина Фаулера "Continuous:Integration”. Поскольку аппаратные средства и ресурсы программного обеспечения продолжают совершенствоваться, все больше процессов становится частью того, что составляет непрерывную интеграцию

Назначение CI

На высоком уровне CI имеет следующие преимущества:

- ✓ снижение риска;
- ✓ уменьшение количества повторяемых процессов, выполняемых вручную;
- ✓ построение развертываемого программного обеспечения в любой момент, в любом
- ✓ месте; обеспечение лучшего контроля проекта;
- ✓ повышение доверия к программному продукту со стороны группы разработки

Преимущества

- ✓ проблемы интеграции выявляются и исправляются быстро, что оказывается дешевле;
- ✓ немедленный прогон модульных тестов для свежих изменений;
- ✓ постоянное наличие текущей стабильной версии вместе с продуктами сборок — для тестирования, демонстрации, и т. п.
- ✓ немедленный эффект от неполного или неработающего кода приучает разработчиков к работе в итеративном режиме с более коротким циклом.

Недостатки

- ✓ затраты на поддержку работы непрерывной интеграции;
- ✓ потенциальная необходимость в выделенном сервере под нужды непрерывной интеграции;
- ✓ немедленный эффект от неполного или неработающего кода отучает разработчиков от выполнения периодических резервных включений кода в репозиторий.
- ✓ в случае использования системы управления версиями исходного кода с поддержкой ветвления, эта проблема может решаться созданием отдельной «ветки» (англ. branch) проекта для внесения крупных изменений (код, разработка которого до работоспособного варианта займет несколько дней, но желательно более частое резервное копирование в репозиторий). По окончании разработки и индивидуального тестирования такой ветки, она может быть объединена (англ. merge) с основным кодом или «стволом» (англ. trunk) проекта.

Сборка по расписанию

В случае сборки по расписанию, которые проводятся каждой ночью в автоматическом режиме — ночные сборки (чтобы к началу рабочего дня были готовы результаты тестирования). Для различия дополнительно вводится система нумерации сборок — обычно, каждая сборка нумеруется натуральным числом, которое увеличивается с каждой новой сборкой. Исходные тексты и другие исходные данные при взятии их из репозитория (хранилища) системы контроля версий помечаются номером сборки. Благодаря этому, точно такая же сборка может быть точно воспроизведена в будущем — достаточно взять исходные данные по нужной метке и запустить процесс

снова. Это даёт возможность повторно выпускать даже очень старые версии программы с небольшими исправлениями.

Принятые термины:

Автоматизированный (automated) — процесс без “ручного вмешательства”. Как только *полностью автоматизированный* (fully automated) процесс запускается, никакого вмешательства пользователя уже не нужно. Системные администраторы называют его “безголовым” (headless) процессом.

Построение (build) — набор действий, выполняемых при компиляции, проверке, инспекции и развертывании программного обеспечения.

Непрерывный (continuous). Технически термин *непрерывный* означает нечто запущенное один раз и никогда не останавливающееся. Это означало бы, что построение выполняется все время; но это не так. Термин *непрерывный* в контексте CI и в случае серверов CI — это скорее циклически повторяемый процесс опроса хранилища с контролем версий на предмет изменений. Если сервер CI обнаруживает изменения, он запускает сценарий построения.

Непрерывная интеграция (Continuous Integration — CI) — это способ разработки программного обеспечения, при котором все участники группы осуществляют частую интеграцию результатов своей работы. Обычно каждый человек выполняет интеграцию ежедневно, что приводит к нескольким интеграциям в день. Результат каждой интеграции автоматически проверяется на предмет ошибок по возможности быстрее. Большинство групп находит, что данный подход ведет к значительному уменьшению количества проблем интеграции и позволяет ускорить разработку связного программного обеспечения.

Среда разработки (development environment) — система, в которой разрабатывается программное обеспечение. Сюда может относиться IDE, сценарии построения, инструменты, библиотеки сторонних производителей, серверы и файлы конфигурации.

Инспекция (inspection) — анализ исходного и бинарного кодов по внутренним критериям качества. В контексте этой книги под *инспекцией программного обеспечения* (software inspection) подразумеваются автоматизированные аспекты (статический анализ и анализ времени выполнения).

Интеграция *(integration) — действие по объединению отдельных артефактов исходного кода вместе, позволяющее проверить их совместную работу.*

Интеграционное построение *(integration build) — действие по объединению компонентов (программы и файлов) в систему программного обеспечения. В больших проектах —b это объединение нескольких составляющих или только низкоуровневых откомпилированных файлов исходного кода, как в меньших проектах. В повседневной жизни термины* построение *(build) и* интеграционное построение *(integration build) используются как синонимы, но в этой книге мы их различаем, подразумевая, что интеграционное построение осуществляется на отдельной машине интеграционного построения.*

Закрытое (системное) построение *(private (system) build). Построение, выполняемое локально на рабочей станции разработчика перед передачей изменений в хранилище с контролем версий для уменьшения вероятности того, что последние изменения нарушат интеграционное построение.*

Качество *(quality). Бесплатный сетевой словарь компьютерной лексики определяет качество как "существенный и специфический атрибут чего-либо ..." и "высшая степень"*

Финальное построение *(release build) — подготовка программного обеспечения для передачи пользователям. Может осуществляться в конце итерации или некоторого другого промежуточного этапа и должно включать все приемочные испытания, а возможно, и* другие проверки, например производительности и загруженности.

Риск *(risk) — потенциальная возможность возникновения проблемы. Риск, который оправдался, называется* проблемой *(problem).*

Проверка *(testing) — общий процесс проверки работоспособности разработанного программного обеспечения. Кроме того, мы разделяем проверки на несколько категорий, включая проверку модуля, проверку компонента и проверку системы. Все, что проверяется (объекты, пакеты, модули и система), и составляет программное обеспечение. Существует множество других типов проверок, например проверка функций и нагрузки, но с точки зрения CI все проверки модуля, написанные разработчиками, выполняются как минимум при построении (хотя процесс построения может быть организован так, что сначала осуществляется быстрая предварительная проверка, сопровождаемая более продолжительными проверками).*

Хранилище с контролем версий

Для непрерывной интеграции необходимо использовать хранилище с контролем версий. Фактически, даже если вы не применяете CI, хранилище с контролем версий, как правило, используется в проекте. Его задача заключается в контроле изменений исходного кода и других элементов программного обеспечения (например, документации), а также в управлении доступом. Это предоставляет "единый источник" для всего исходного кода, где он будет доступен из любого места. Хранилище с контролем версий позволяет возвращаться к прежним версиям исходного кода и других файлов. Вы запускаете CI как основную линию хранилища с контролем версий (например,Head/Trunk в таких системах, как CVS и Subversion). Существуют различные типы систем контроля версий, которые вы также можете использовать

Сервер CI

Сервер CI выполняет интеграционное построение всякий раз, когда в хранилище с контролем версий передаются изменения. Как правило, вы настраиваете сервер CI, чтобы проверять изменения в хранилище с контролем версий каждые несколько минут или около того. Сервер CI получает файлы исходного кода и запускает сценарий или сценарии построения. Серверы CI допускают планирование, обеспечивая построение с обычной частотой, например каждый час (однако обратите внимание, это не непрерывная интеграция). Кроме того, серверы CI обычно предоставляют удобную панель, где отображаются результаты построения. Хотя это и рекомендуется, сервер CI не обязан выполнять непрерывную интеграцию. Но вы можете писать свои собственные специальные сценарии и, кроме того, запускать вручную интеграционное построение всякий раз, когда в хранилище происходит изменение. Использование сервера CI уменьшает количество специальных сценариев, которые в противном случае вам пришлось бы написать.

Сценарий построения

Сценарий построения (build script) — это единый сценарий, или набор сценариев, используемый для компиляции, проверки инспекции и развертывания программного обеспечения. Вы можете применять сценарий построения без реализации системы. К инструментам, позволяющим автоматизировать цикл построения программного обеспечения, относятся Ant, NAnt, make, MSBuild и Rake, но сами по себе они не обеспечивают CI. Для построения программного обеспечения некоторые используют

IDE, однако с тех пор как процесс CI перестал быть “ручным” исключительно с применением средств построения, обладающих IDE, не будем сбрасывать его со счетов. Если быть честным, то используя IDE для построения, вы можете сделать то же самое, что и *без его применения*.

Сделайте тесты

Тесты просто необходимо включать в continuous integration процесс, в противном случае вы не можете быть уверены в качестве и работоспособности своего проекта. Чем тестов больше, тем лучше, в разумных пределах конечно.

Основными двумя ограничителями на количество тестов будет:

1. время интеграции — сборка по-прежнему должна оставаться быстрой, основное тестирование можно перенести «на ночь»,
2. наличие автоматизированных тестов — не все тесты требуют автоматизации, нет смысла делать автоматизированные тесты только для самих тестов, они должны быть целесообразны.

Чем лучше ваши тесты, тем раньше находятся ошибки и раньше исправляются. Как известно, чем раньше ошибка исправлена, тем дешевле ее исправление. Это одно из основных преимуществ практики непрерывной интеграции — снижение стоимости исправления ошибок (не всех конечно). Попутно наличие хорошего набора тестов в процессе интеграции дает больше уверенности в том, что проект работает правильно.

Именнно присутствие тестов одно из отличий интеграции от нажатия кнопки Build в вашей любимой IDE.

Механизм обратной связи

Одной из ключевых задач CI является обратная связь при интеграции, поскольку вы, естественно, хотите как можно скорее узнать, не было ли проблем с последним построением. Оперативное получение такой информации позволяет быстро устранить проблему. К механизмам обратной связи относится служба коротких сообщений *(Short Message Service — SMS) и RSS (Really Simple Syndication)*

Как добиться непрерывной интеграции?

Эти этапы могут быть последовательно применены практически к каждому действию, осуществляемому в проекте.

Идентификация (identify).— это процесс, который требует автоматизации. Он может осуществляться при компиляции, проверке, инспекции, развертывании, интеграции базы данных и т.д.

Построение (build). Применение сценария построения делает автоматизацию воспроизводимой и однозначной. Сценарии построения можно написать при помощи NAnt для платформ .NET, Ant для платформы Java и Rake для Ruby (это лишь несколько из примеров).

Совместное использование (share). Используя такую систему контроля версий, как Subversion, вы позволяете применять эти сценарии и программы другим. Теперь значение сценария распространяется на весь проект.

Непрерывность (continuous). Следует гарантировать, что автоматизированный

процесс будет запущен при внесении любого изменения. Для этого используется

сервер CI. Если в вашей группе соблюдается жесткая дисциплина, вы конечно можете решить выполнять построение вручную при каждом изменении, передаваемом системе контроля версий

Снижение **риска** с использованием CI

Проект никогда не проходит без проблем. Занимаясь CI на практике, вы неизбежно выясняете, что все этапы процесса разработки рано или поздно повторяются. CI помогает выявить и снизить возможные риски, упрощая оценку и оповещение о состоянии проекта на основании конкретных доказательств.

Сколько программного обеспечения мы реализовали?

Ответ: посмотрите последнее построение.

Каково покрытия кода проверками?

Ответ: посмотрите последнее построение. Кто отличился в последнем коде? Ответ: посмотрите последнее построение.

Все проекты начинаются с благих намерений, но завершаются благополучно далеко не все. **Погубившие** их проблемы — ***результат потери контроля над рисками***. Как уже упоминалось ранее, мы нередко слышим от участников группы разработки: “На мой взгляд, проверки и обзоры кода (объединенные или нет) — плохие практики”. А когда сроки сдачи проекта начинают поджимать, группа обычно отказывается от этих практик в первую очередь. Когда после каждого изменения вы нажимаете “кнопку Integrate”, вы закладываете фундамент снижения рисков заранее.

Снижение рисков

Отсутствие развертываемого программного обеспеченияИспользуйте систему CI для построения развертываемого программного обеспечения в любое время. Организуйте воспроизводимый процесс построения, применяющий все элементы программного обеспечения из хранилища с контролем версий

Позднее выявление дефектов

Выполняйте построение, подразумевающее проверки разработчика при каждом изменении, это позволит обнаруживать дефекты программного обеспечения на раннем этапе разработки

Плохой контроль проекта

Выполняя построение регулярно, постоянно контролируйте состояние вашего программного обеспечения. При эффективном применении практик CI состояние проекта перестанет быть проблемой

Низкокачественное программное обеспечение

Запускайте проверки и инспекции при каждом изменении, так вы сможете обнаружить потенциальные дефекты базового кода, включая его чрезмерную сложность, дублирование, недостатки проекта, покрытие кода проверками и другие факторы

Автоматизация построений

Что такое построение (build) программного обеспечения? Только ли это компиляция (compiling) его компонентов? Или построение — это компиляция

компонентов и запуск автоматизированных проверок? Должно ли построение включать инспекции (inspection)? Построение может включать любой из этих процессов, что эффективно снижает риски; однако чем больше процессов добавлено в построение, тем медленней обратная связь.

Существует множество инструментов построения. К наиболее популярным относятся Ant для Java и NAnt для .NET. Использование инструментов выполнения сценариев, разработанных специально для построения программного обеспечения, вместо собственного набора пакетных файлов оболочки, является более эффективным способом создания однозначного и воспроизводимого решения построения. Помните: построение должно осуществляться нажатием одной кнопки. Когда вы нажимаете кнопку <Integrate>, как показано на рис. 4.1, на сборочном конвейере запускается процесс, который создает работоспособное программное обеспечение. Иногда организации не способны задействовать CI потому, что они не могут реально автоматизировать свое

В общих чертах построение программного обеспечения подразумевает примерно следующие этапы.

1. Осуществляйте построение с использованием таких инструментов сценариев построения, как NAnt, Rake, Ant или Maven. Сначала оставьте сценарии простыми; впоследствии в них можно будет добавить больше процессов.

2. Добавьте кнопке <Integrate> внутри сценария построения другие процессы (очистка, компиляция и т.д.).

3. Чтобы построить программное обеспечение, запустите сценарий из IDE или командной строки.
NAnt для платформы .NET; но вы вполне можете получить тот же результат при помощи других средств сценариев построения, таких как Ant или Maven для Java, MSBuild для .NET, Rake для Ruby и т.д. Проблема не в том, какой инструмент вы выбираете, главное, использовать существующий инструмент, а не создавать собственное решение.

Отделяйте сценарии построения от IDE

Вы должны избегать применения IDE в сценариях построения. IDE может зависеть от сценария построения, но сценарий построения не должен зависеть от IDE Например, IDE способен облегчить создание сценария построения, но при этом он может помещать файлы построения и зависимые элементы в структуре того каталога, в котором

установлен IDE. Чтобы проверить возможность многократного использования сценария построения, созданного IDE, возьмите сценарий построения и запустите его на новой машине с только что установленной ОС.

Создание отдельного сценария построения важно по двум причинам.

1. Все разработчики могут использовать разные IDE, поэтому может оказаться весьма сложно соотнести конфигурационные особенности каждого IDE.

2. Сервер CI должен работать автоматически, осуществляя построение без вмешательства людей. Следовательно, тот же автоматизированный сценарий построения, используемый разработчиками, может и должен быть использован сервером CI.

Типы и механизмы построения

Существуют разные типы построения, предназначенные для различных целей. Построе ние может быть запущено разными механизмами, например пользователем, по расписанию, в связи с внесением изменений или в результате некоего события.

Закрытое построение

Разработчик выполняет закрытое построение (private build) перед передачей своего кода в хранилище. Осуществляя закрытое построение, вы интегрируете свои изменения с последними изменениями, доступными в хранилище с контролем версий. Это может предотвратить сбой построения. При закрытом построении выполняются следующие шаги.

Проверьте код, полученный из хранилища.

Внесите в него изменения

Получите последние системные изменения из хранилища

Запустите построение, включающее выполнение всех ваших проверок модуля.

Передайте свои изменения кода в хранилище.

Интеграционное построение

Интеграционное построение (integration build) интегрирует изменения, внесенные в хранилище группой с общей линией (mainline) (называемой также головой (head) или магистралью (trunk)). В идеале оно должно осуществляться на выделенной машине.

Финальное построение

Финальное построение (release build) готовит программное обеспечение к выпуску для пользователей. Одной из задач CI является создание развертываемого программного обеспечения. Финальное построение, происходящее в конце итерации или

некоторого другого промежуточного этапа, может включать более обширные проверки, в том числе проверку производительности, загруженности и все приемочные испытания. Кроме того, при большинстве финальных построений создается установочная среда, запускаемая в системе пользователя. Финальное построение может также применяться для проверки готовности к QA, если используется отдельный, поэтапный процесс и группа

Механизмы построения

Не все построения запускаются одинаково. Для вызова построения в соответствующий момент необходимо учесть его назначение и частоту. В некоторых ситуациях сценарии могут оказаться настолько большими или иметь так много зависимостей, что запускать их автоматически не стоит; лучше делать это по требованию. В других случаях автоматический запуск может выполняться под управлением CI. Типы механизмов построения описаны ниже.

По требованию (on-demand). **Это управляемый пользователем процесс, в котором некто вручную инициализирует интеграционное построение.**

По расписанию (scheduled). Процессы управляются по времени, например, можно ежечасно проверять, не произошли ли изменения. Расписание действий может пригодиться для повторяемых процессов, таких как запуск исчерпывающего набора проверок защиты или загруженности программного обеспечения. Для расписания можно использовать задачу cron, хотя большинство серверов CI самостоятельно поддерживают расписание.

Опрос изменений (poll for changes). Регулярно инициируемый процесс проверки хранилища с контролем версий на предмет изменений. Если таковые обнаружены, он запускает интеграционное построение. Все серверы CI поддерживают некий механизм “опроса изменений”.

Управляемое событиями (event-driven). Управление событиями подобно опросу изменений, но вместо инструмента CI построение запускает хранилище с контролем версий на основании предопределенного события (изменения). Если хранилище с контролем версий обнаруживает изменение, оно инициализирует сценарий построения.

Инструменты CI

Непрерывная интеграция является неотъемлемой частью гибких методологий разработки (scrum, kanban и т.п.). От спринта к спринту команда стремиться «не сломать

сборку» добавляя новые фичи в проект. Однако, разработчики полностью сосредоточены на добавление нового функционала, случается, что и ошибки попадают в код, чем ломают сборку. Что бы остановить такие ошибки, поступающие из системы контроля версий, сервер CI (Continius integration) проверяет качество кода. Если в какой-то момент, в код закрадывается ошибка, сервер CI очень быстро сообщит об этом.

Существует ряд платных продуктов от именитых компаний, которые специализируются на продуктах для разработчиков. Bamboo от Atlassian и Teamcity от Jetbrains — наиболее популярные серверы непрерывной интеграции среди платных продуктов. Однако, существует несколько приличных заменителей, кратких обзор которых, представлен ниже.

Jenkins

Наиболее популярный из известный в среде CI серверов с открытым исходным кодом, стал ответвлением от своего предшественника — CI сервера Hudson. Так же можно использовать установку в контейнере сервлета. Jenkins поддерживает ряд систем контроля версий, таких как Git, Mercurial, Subversion, Clearcase и многие другие. Возможно использование систем сборок Apache Ant, Apache Maven и выполнение других shell скриптов или windows скриптов для действий до и после сборки.

Buildbot

Написаный в основном на Python и основаный на Twisted фреймворке. Начинался как более легковесная альтернатива Mozilla Tinderbox и сейчас используется в таких проектах как Mozilla, Chromium, Webkit и многими другими. Buildbot устанавливается в виде основного сервера (master) и подчиненных (slave). С основного сервера осуществляется мониторинг изменений в репозитории, координация работы подчиненных серверов и рассылка отчетов пользователям и разработчикам. Подчиненный сервер может запускаться под различными операционными системами. Настраивается при помощи конфигурационных скриптов Python на основном сервере. Эти скрипты для настройки компонентов сборки очень просты в понимании, однако обладают всей мощью питона.

Travis CI

Travis CI вероятно самый простой CI сервер, для того что бы начать. Кроме того, что он распростроняется с открытым исходным кодом и соответственно бесплатен при установке на свой сервер, Travis CI имеет SaaS версию, которая бесплатна для проектов с

открытым кодом. При регистрации и настройке просто нужно закрепить свой GitHub аккаунт, получить необходимые права и добавить .travis.yaml файл в своем проекте. Travis CI соберет новый билд после добавления изменений на GitHub.

CruiseControl

имеет следующие возможности:

работает по расписанию или следит за изменениями:

в репозитории CVS и многих других систем контроля версий в файлах и директориях в сборке других проектов

сам вытащит новую версию из CVS(и других)

соберет используя Ant, Maven, скрипт

опубликует результаты:

в зависамости от результата сборки на web-сервере входящем в комплект разошлет почту

выполнит скрипт (shell, bat)

выполнит скрипт Ant

много других способов конфигурируется одним XML файлом — его тоже можно хранить под CVS

Используйте выделенную машину для интеграционного построения

Когда вы выделяете машину для интеграционного построения, вы решительно ограничиваете предположения о среде и конфигурации, а также способствуете предотвращению слишком позднего проявления проблемы “а на моей машине это работает”. Любая локальная рабочая станция обычно имеет несколько разных конфигураций и множество зависимостей, зачастую отсутствующих в среде развертывания. Если разработчик вносит локальные изменения и забывает передать несколько файлов в хранилище с контролем версий, то система CI, выполняющаяся на отдельной машине, запустит интеграционное построение и обнаружит их отсутствие. Кроме того, вы можете устанавливать серверы приложений и баз данных в определенное состояние каждый раз, когда происходит интеграционное построение. Это также позволит не только уменьшить количество предположений, но и существенно быстрей обнаруживать и решать проблемы. Когда сотрудники узнают, что последнее интеграционное построение потерпело неудачу, они могут избежать получения

При создании машины интеграционного построения следует учитывать несколько факторов. Сосредоточившись на них, вы извлечете максимум преимуществ. Рекомендуемые системные ресурсы. Многие из них можно получить при помощи инструментальных средств. Чем лучше аппаратные ресурсы, тем меньше продолжительность построения (обсуждается далее в этой главе). Как правило, цена увеличения аппаратных ресурсов машины интеграционного построения окупается за счет экономии времени. В хранилище с контролем версий находятся все элементы программного обеспечения. Все, что имеет отношение к его разработке, должно быть передано в хранилище с контролем версий. Сюда относится исходный код, сценарии построения, файлы конфигурации, инструменты (сервер приложений, сервер баз данных и инструменты статического анализа), сценарии проверки кода и файлы базы данных

Чистая среда. Перед выполнением интеграционного построения сценарий CI должен удалять любые зависимости кода от среды интеграции. Следует гарантировать удаление всего исходного кода и бинарных файлов предыдущего интеграционного построения. Удостоверитесь также, что система CI устанавливает проверочные данные и любые другие элементы конфигурации в определенное состояние. Этот подход уменьшает количество предположений, ликвидирует зависимости и организует построение программного обеспечения как будто на новой машине.

Используйте сервер CI

При реализации непрерывной интеграции имеет смысл использовать сервер CI. Безусловно, вы можете создать свой собственный инструмент или выполнять интеграцию вручную; но сейчас на рынке доступно множество превосходных инструментов, которые предоставляют ценнейшие возможности, а также позволяют расширять их. Следовательно, необходимость в создании собственного сервера CI отпадает. Но если вам все же придется делать это, то вы, вероятно, захотели бы включить в него большинство следующих возможностей.

Периодический опрос хранилища с контролем версий на предмет изменений.

Выполнение неких действий по расписанию, например, ежечасно или ежедневно.

Выявление “периодов затишья”, в течение которых никаких интеграционных построений не выполняется.

Поддержку для различных инструментов сценариев построения, включая утилиты командной строки, такие как Rake, make, Ant или NAnt.

Отправку электронной почты заинтересованным сторонам.

Отображение истории предыдущего построения.

Отображение панели управления, доступной через Web, чтобы каждый мог просматривать информацию интеграционного построения.

Поддержку нескольких систем контроля версий для разных проектов.

На этом список не заканчивается. Большинство серверов CI уже реализовало эти средства. Можно без проблем подобрать инструмент, который полностью удовлетворяет вашим потребностям и подходит к среде разработки. CruiseControl, Luntbuild, Continuum, Pulse, и Gauntlet — вот лишь некоторые из инструментов, которые можно использовать для реализации CI.

Функциональные возможности

Безусловно, важнейшим критерием при выборе инструмента является его функциональность. В этом разделе описаны основные и дополнительные функциональные возможности, предоставляемые инструментами построения и планирования.

Инструменты построения — основные функциональные возможности

Ниже приведены основные функциональные возможности инструментов построения. Компиляция кода. Здесь никаких сюрпризов: компиляция исходного кода — это основной компонент построения программного обеспечения. Для повышения эффективности она должна осуществляться при условии изменения либо исходного кода, либо зависимостей.

Упаковка компонентов. После компиляции исходного кода и оформления любых других артефактов, которые должны быть включены в программное обеспечение, их обычно необходимо связать в развертываемые компоненты, такие как файлы JAR для Java или файлы EXE для Windows. Инструмент построения, который вы выберете, должен уметь добавлять в пакет необходимые компоненты вашей системы, и делать это только при изменении содержимого пакета.

Выполнение программы. Инструмент построения должен обладать хорошими возможностями по запуску программ на целевой платформе, а также вызова любых программ, которые имеют интерфейс командной строки.

Манипулирование файлами. Функциональные возможности создания, копирования, а также удаления файлов и каталогов присущи практически всем инструментам построения.

Инструменты построения — дополнительные функциональные возможности

К дополнительным функциональным возможностям инструментов построения относятся следующие.

Интеграция инструментов контроля версий.

Если ваш инструмент планирования построений делегирует действия по контролю версий инструменту построения или если вам необходимы некие другие действия по контролю версий, связанные с автоматизацией, то желательно подобрать систему с контролем версий, имеющую внутренний инструмент построения. Напомним, что в случае необходимости вы всегда можете прибегнуть в качестве аварийного варианта к интеграции инструментов при помощи командной строки.

Создание документации.

Если вы работаете с языком программирования, который имеет встроенные средства документирования, например C# или Java, то очень полезно иметь инструмент построения, обладающий API автоматического создания документации в ходе построения.

Функциональные возможности развертывания.

Если при автоматизирован ном построении вы планируете запускать проверки функций или модулей “в контейнере”, то построенное приложение необходимо сначала развернуть на проверочном сервере. Эти функциональные возможности могут быть предоставлены инструментом построения или производителем сервера (либо сообществом пользователей сервера) в качестве дополнения.

Анализ качества кода.

Как упоминалось в главе 7, прекрасное представление о стабильности и ремонтопригодности кода можно получить в результате запуска различных типов автоматизированных инспекторов. Выясните, какие средства анализа встроены в ваш инструмент построения или доступны как дополнения.

Расширяемость**.**

Как правило, писать собственные дополнения для инструмента построения не приходится; большинство проблем, с которыми вы столкнетесь, типичны

и кем-то уже решались. Но в некоторых случаях вы можете захотеть усовершенствовать собственно инструмент построения; например, при желании полностью интегрировать новую проверку или средство составления отчетов. Хорошо задокументированные API расширения окажутся в этом случае весьма полезны. Только не забудьте пожертвовать свое дополнение сообществу пользователей. Вы, ваше дополнение и сообщество существенно выиграют от этого.

Многоплатформенное построение.

Большинство серверов CI разработано так, чтобы выполняться на одной машине построения. Это, безусловно, означает, что все действия построения будут происходить на платформе сервера построения. Для большинства приложений это подойдет. Но если вы разрабатываете программное обеспечение для нескольких платформ, все значительно усложняется. Наилучшим выходом в данном случае может стать приобретение одного из коммерческих инструментов, способных осуществлять процесс построения на нескольких серверах.

Ускоренное построение**.**

Ключевым преимуществом CI является возможность быстрого запуска полного цикла построения. По этой причине некоторые эксперты советуют удерживать полный цикл построения в пределах десяти минут1. Если же ваш цикл построения занимает несколько часов в связи с большим объемом кода (хотя это редкость), вы можете рассмотреть возможность применения некоторых инструментальных средств, которые способны распределить этапы построения на

Инструменты планирования построения — дополнительные функциональные возможности

К дополнительным функциональным возможностям инструментов планирования построения относятся следующие. Зависимости между проектами. В соответствии с вашей стратегией управления конфигурацией, при наличии зависимости между проектами можно выполнять построение зависимого проекта одновременно с построением основного.

Пользовательский интерфейс**.**

Строго говоря, особой причины требовать наличия пользовательского интерфейса для инструмента планирования построения нет. Основные функциональные возможности выполняются как демон, проверяющий изменения в системе контроля

версий и запускающий построение, посылая сообщение обратной связи. Тем не менее весьма полезно иметь пользовательский интерфейс, который позволяет изменять конфигурацию, проверять текущее состояние построения и загружать артефакты. Все инструменты обеспечивают это в той или иной форме, обычно в качестве интерфейса Web-приложения. Некоторые инструменты, такие как Luntbuild, распространяются как Web-приложения. Другие инструменты, такие как CruiseControl, применяют другой пользовательский интерфейс, предоставляемый как необязательный элемент. Любой хорошо проработанный пользовательский интерфейс сэкономит ваше время и силы при работе с инструментом.

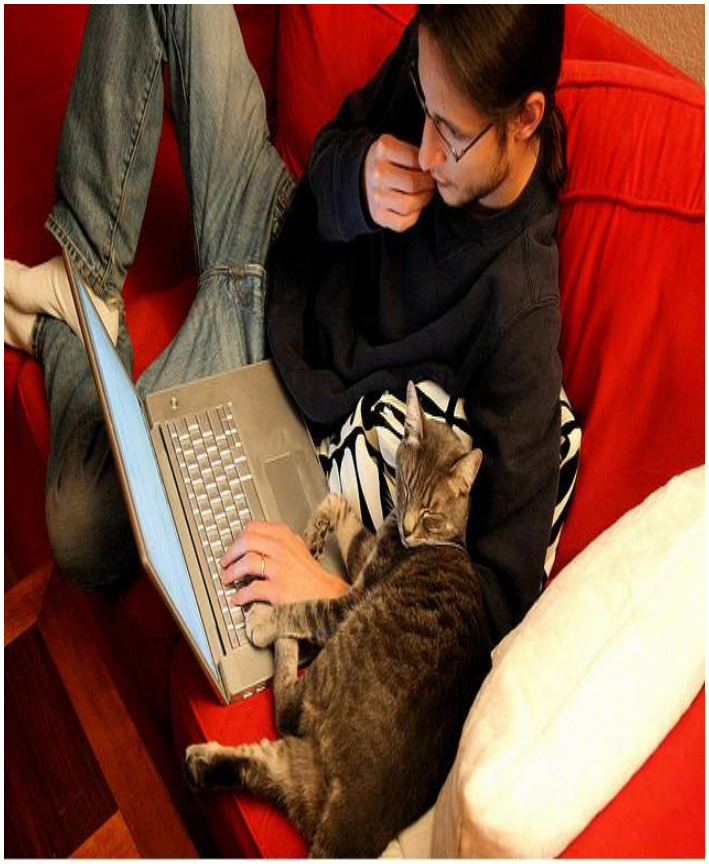
Публикация артефактов. В конце концов, итог успешного построения — это развертываемый компонент. Если вы используете реальную мощь CI, то результат будет также включать документирование, проверку результатов, анализ их качества и другие показатели. Все инструменты обеспечивают некоторый уровень функций публикации, предоставляя каталог для содержания публикуемых артефактов. Более сложные инструменты автоматически оформляют результаты проверки разработчика и другие отчеты для удобства просмотра.

Защита. И наконец, некоторые инструменты обеспечивают аутентификацию и авторизацию, позволяя определять, кто именно может просматривать результаты и вносить изменения в конфигурацию. Обычно, учитывая корпоративный дух CI, в этом нет необходимости, но если вы работаете несколькими группами или имеете специфические требования безопасности, то это может оказаться важным. Тем не менее помните, что обеспечение защиты увеличивает затраты на поддержку. Каждый раз, когда некто входит в состав группы или покидает ее, вам придется

Заключение

CI становится господствующей тенденцией, имеет инструментальные средства и сообщество пользователей, доказывающих это. Теперь, когда вы готовы присоединиться к тем, кто извлекает выгоду из подходов CI, вы можете выбирать инструменты, обеспечивающие наилучшее соответствие задачам вашего проекта и вашей группы

Парное программирование – вид так называемого экстремального программирования, которое относится к методологии гибкой разработки Agile. У этого метода есть как плюсы, так и минусы.



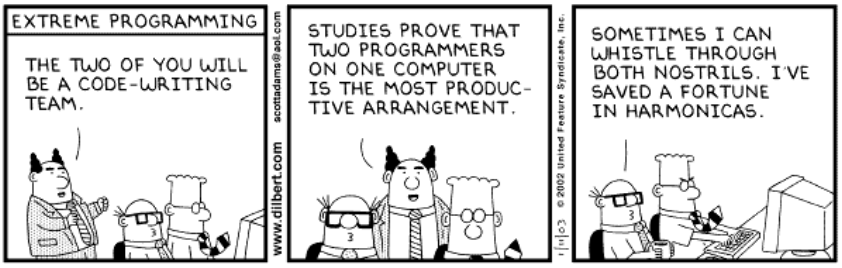
Плюсы:

- как правило, в этом случае не требуется проводить code review;
- даже самый опытный разработчик не может знать всего. Наличие двух программистов помогает быстрее и эффективнее решать задачи;
- уменьшается количество ошибок. Кроме того, они чаще всего могут быть обнаружены именно в процессе работы;
- разработчики лучше концентрируются и меньше отвлекаются на посторонние дела. Этот пункт можно также отнести и к минусам: постоянные перерывы все же необходимы, поэтому в случае парного программирования рекомендуется делать паузы в 10-20 минут каждые 40-60 минут;
- разработчики знают бОльшую часть кода, чем если бы они писали только свою часть. В этом случае они смогут более эффективно вносить изменения в случае надобности;
- если нужно сделать слияние двух фрагментов кода, ПП быстрее и эффективнее, чем если бы один разработчик сначала писал код, а потом отправлял его на отправку коллеге, попутно давая объяснения;
- разработчики учатся коллективно решать проблемы, обсуждать вопросы, находить компромисс.

Минусы:

- не все хотят работать в паре. Кому-то психологически удобнее работать одному, кто-то может быть недоволен партнером. Кроме того, код каждого разработчика индивидуален в той или иной степени;

- многие руководители могут посчитать, что это невыгодно – сажать за одно рабочее место двух человек для выполнения одной задачи. Стоимость разработки может возрасти, однако это коменсируется еще одним плюсом – улучшением внутренней архитектуры и меньшим количеством ошибок. В 1999 году было даже проведено исследование на тему временных затрат. Согласно эксперименту, затраченное время выросло на 15%, но при этом ошибок в коде было на 15% меньше. Однако не стоит забывать, что исправление ошибок еще на стадии разработки экономит большое количество времени на поддержке;
- необходимо согласовывать график разработчиков;
- человеческий фактор. Для ПП необходимо иметь терпение и желание работать в паре. Если один из участников будет просто сидеть молча и смотреть, как второй пишет код, или если один из программистов станет продавливать свое мнение, не прислушиваясь к коллеге, то такой вариант разработки не принесет ни удовольствия, ни эффективности.



Виды ПП

Существует несколько **стилей** парного программирования:

- **ведущий-ведомый**. В этом случае на втором месте после разработки часто стоит обучение менее опытного программиста, как правило, именно он непосредственно пишет код. Второй программист более опытный, он подсказывает, направляет, дает рекомендации;
- **на равных**. В этом случае работают два примерно одинаковых по опыту разработчиков, время от времени меняющихся местами;
- **водитель-штурман**. В этом случае программисты выбирают разные роли. Один пишет код, разбирается в деталях, а второй занимается архитектурой кода, решением логических задач, рисует схемы;

- **пинг-понг**. Один программист пишет тест, а второй – реализацию под него. После происходит смена ролей;
- **удаленное ПП**. Единственным минусом такого стиля является то, что нельзя тыкнуть пальцем в экран. А если серьезно, то в этом случае можно использовать разные инструменты: например, давать партнеру возможность одновременно с вами писать код или же только видеть ваш экран. Но многое также зависит от от интернет-канала: в случае неполадок на линии работать не получится. Для такого стиля работы рекомендуют сократить итерации по времени и почаще коммитить код.

Метод ПП можно использовать не только в написании кода. Для отрисовки дизайна он, скорее всего, не подойдет. В сети также есть упоминания о тройном программировании, однако этот метод вряд ли может претендовать на эффективность разработки.

Область применения

Кент Бек, «отец» экстремального программирования, писал о ПП в своей книге Extreme Programming Explained: Embrace Change. В ней он не дает четких советов о том, для каких проектов подойдет или не подойдет данный метод, но считает, что помехой в ПП могут быть различия в бизнес-культуре участников разработки. По его словам, ПП – не для внеурочной работы, не для разработки в режиме аврала, когда участники уже устали. Кроме того, ПП вряд ли подойдет для проекта, над которым работает сотня разработчиков, а также для рабочей среды, в которой для обратной связи требуется длительный период времени. Кент скептически относится к удаленному ПП. Процитируем еще одно исключение:

«Вы не можете использовать экстремальное программирование в случае, если имеете дело с технологией, которая подразумевает экспоненциальный рост затрат, связанных с внесением в систему изменений. Например, если вы имеете дело с мэйнфреймом, планируете использовать установленную на нем реляционную базу данных и не уверены в том, что схема реляционной базы данных в точности соответствует тому, что вам нужно. В подобной ситуации вы не должны использовать ЭП. ЭП основывается на чистом и простом коде. Если вы усложняете код для того, чтобы избежать модификации 200 существующих приложений, в самом скором времени вы потеряете гибкость, ради которой вы, собственно, и решили использовать ЭП».

Что же думают о ПП программисты?

Александр Фомин, lead разработчик C# из Taucraft, делится впечатлениями:

– Для каких задач вы использовали ПП?

При знакомстве с проектом практически всегда первое задание делалось в паре – вникнуть в суть, понять, «как здесь принято», познакомиться с cross-edge functionality. В начале разработки новой фичи, когда общего решения еще не придумано. В паре архитектура вырисовывается гораздо проще. В сложных местах. Был реальный случай, когда одну штуку я не смог сделать за неделю, а в паре написали за полдня. Правда, возможно, здесь сыграло свою роль то, что над этой штукой я таки неделю думал. Редко, но бывало, когда человек внедряется к уже наполовину сделанной фиче. Правда, сессия получалась недолгой, но это лучший способ вникнуть в процесс.

– Как долго использовали ПП?

– Зависит от таска, конечно. Одно время – примерно три дня в неделю примерно три месяца, когда делали REST API, – большой кусок полностью с нуля. Сейчас реже, не более трех дней подряд где-то раз в месяц.

– Какие у вас остались впечатления от использования данной практики?

– Это достаточно сильно изматывает. Из 8 часов в паре удается поработать часов пять-шесть, дальше становится сложно. Но, вместе с этим, время, отработанное в паре, летит незаметно, и к концу дня частенько кажется, что прошла только половина.

Многое зависит от второго человека и от стиля работы в паре. Образно говоря, «ведущий-ведомый» лично мне нравится больше, чем «на равных», но это, скорее, черта моего характера. Часто бывает, что сделанное в паре кто-то один потом правит самостоятельно – сильно раздражает, когда это не ты, и очень нравится, когда делаешь

сам. Еще важным фактором является скорость работы – иногда бывает, что просто не успеваешь за соседом, тогда приходится терять много времни, чтобы «въехать» или объяснить этот момент.

Выводы

Как видим, ПП не зря относится к экстремальному программированию. Почему-то большинство статей не записывает в минусы повышенную интенсивность разработки и как следствие накопление усталости. С другой стороны, родственники британских ученых считают, что человек способен продуктивно работать 4-5 часов в день, остальное уходит на коммуникацию, околорабочие вопросы и посторонние дела. И в этом случае ПП как раз может быть интересным вариантом использования продуктивного рабочего времени.

Исследование Simula Research Laboratory учитывало и психологические аспекты ПП. В нем участвовало почти 300 разработчиков Java. Авторы исследования сделали выводы: **ПП показывает наибольшую эффективность при работе в новых областях и при обучении джуниоров**. А вот в журнале International Journal of Human Computer Studies были опубликованы результаты другого эксперимента на тему ПП, и выводы оказались также интересными: **эффективность пары разработчиков снижается, если один или два участника уже имели опыт решения похожих задач. Получается, что программистов мобилизуют нерешенные вопросы?**