Оглавление

К оглавлению

Словечки, поставь свечку

Прототипирование— это упрощенная версия конечного продукта, которая позволяет выявить возможные недостатки, фактически даже не приступая к процессу его создания. Таким способом удается устранить возможные проблемы заранее, что в конечном счете сокращает затраты. Ведь переделывать с нуля конечный продукт значительно сложнее и дороже, чем использовать прототипирование. Это возможность с минимальными затратами создать образец. На основании созданного прототипа принимают решения относительно дальнейшей возможности и целесообразности реализации проекта.

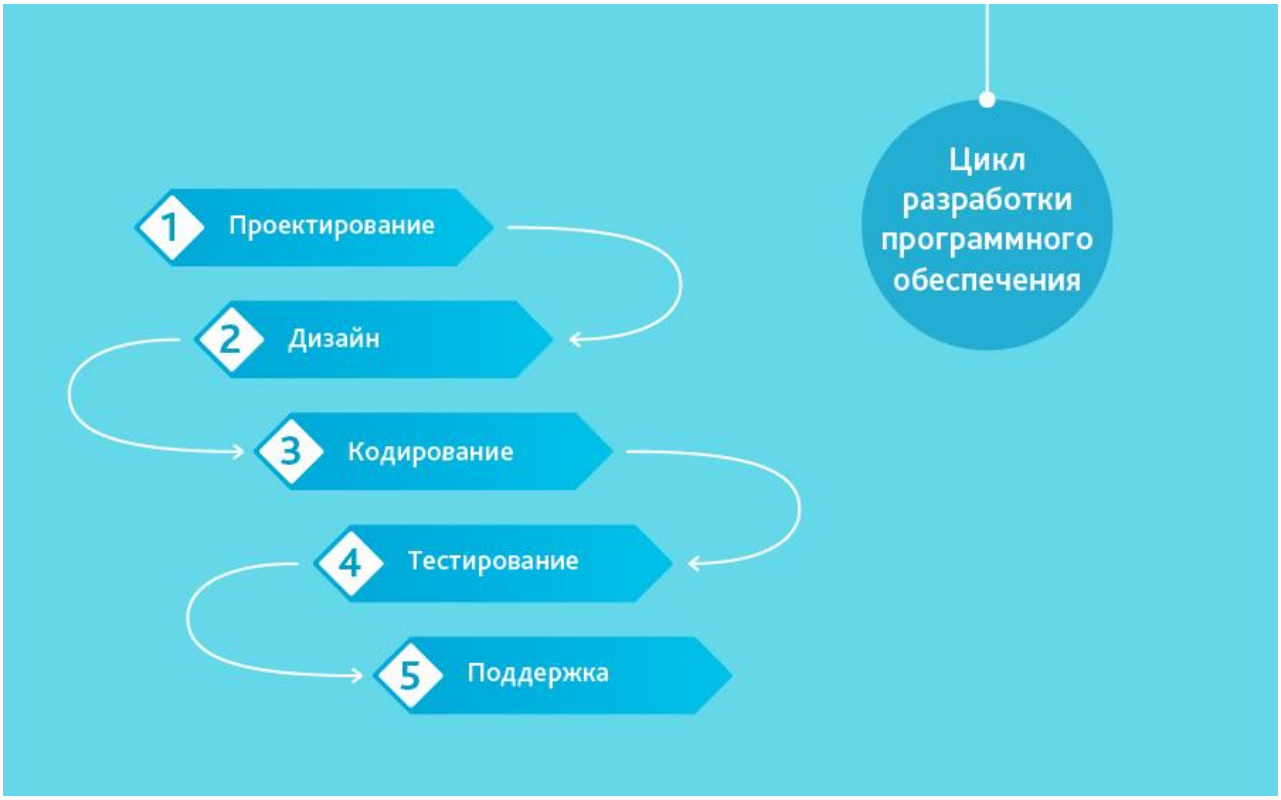
Методология — это наука, представляющая собой учение об организации разных видов деятельности человека, а именно - научной, практической, учебной, игровой и художественной. В методологии используют различные способы, стратегии и методы построения определенного вида деятельности, разработанные специалистами из этой области знаний на основании глубокого изучения принципов деятельности и процессов работы.

Итеративный подход (англ. iteration - «повторение») в разработке программного обеспечения — это выполнение работ параллельно, с непрерывным анализом полученных результатов и корректировкой предыдущих этапов работы в выбранном направлении.

1. Каскадная стратегия разработки: достоинства и недостатки

Каскадная модель или «водопад»

К оглавлению



Одна из самых старых, подразумевает **последовательное прохождение стадий**, каждая из которых **должна завершиться** полностью **до начала следующей**. Данная стратегия основана **на полном определении** всех **требований** к разрабатываемому программному средству в начале процесса разработки. В модели Waterfall **легко управлять проектом**. Благодаря её жесткости, **разработка проходит быстро**, **стоимость и срок заранее определены**.

Основными достоинствами каскадной стратегии, проявляемыми при разработке **соответствующего** ей проекта, являются:

1. стабильность требований в течение всего жизненного цикла разработки;
2. простота применения стратегии (необходимость только одного прохода этапов разработки;
3. простота планирования, контроля и управления проектом;
4. возможность достижения высоких требований к качеству проекта в случае отсутствия жестких ограничений затрат и графика работ;
5. доступность для понимания заказчиками.

Но это палка о двух концах. Каскадная модель **будет давать отличный результат только в проектах с четко и заранее определенными требованиями и способами их реализации.**

К оглавлению

К **недостаткам каскадной стратегии**, проявляемым при ее выборе к **несоответствующему** проекту, следует отнести:

1. сложность четкого формулирования требований в начале жизненного цикла ПС и невозможность их динамического изменения на протяжении ЖЦ ПС;
2. последовательность линейной структуры процесса разработки; на практике разрабатываемые ПС (или система) обычно слишком велики, чтобы все работы по их созданию выполнить однократно; в результате возврат к предыдущим шагам для решения возникающих проблем приводит увеличению затрат и нарушению графика работ;
3. проблемность финансирования проекта, связанная со сложностью единовременного распределения больших денежных средств;
4. непригодность промежуточного продукта для использования;
5. недостаточное участие пользователя в разработке системы или ПС - только в самом начале (при разработке требований) и в конце (во время приемочных испытаний), что приводит к невозможности предварительной оценки пользователем качества ПС или системы.

Нет возможности сделать шаг назад, **тестирование начинается** только после того, **как разработка завершена** или почти завершена. Продукты, разработанные по данной модели без обоснованного ее выбора, могут иметь **недочеты** (**список требований нельзя скорректировать в любой момент**), о которых становится известно лишь в конце из-за строгой последовательности действий. **Стоимость внесения изменений высока**, так как для ее инициализации приходится ждать завершения всего проекта. **Тем не менее, фиксированная стоимость часто перевешивает минусы подхода.**

Области применения каскадной стратегии ограничены недостатками данной стратегии. Ее использование наиболее эффективно в следующих случаях:

1. при разработке проектов с четкими, неизменяемыми в течение ЖЦ требованиями, понятными реализацией и техническими методиками;
2. при разработке проекта, ориентированного на построение системы или продукта такого же типа, как уже разрабатывались разработчиками ранее;
3. при разработке проекта, связанного с созданием и выпуском новой версии уже существующего продукта или системы;
4. при разработке проекта, связанного с переносом уже существующего продукта на новую платформу;
5. при выполнении больших проектов в качестве составной части моделей ЖЦ, реализующих другие стратегии разработки

К оглавлению

2. Инкрементальная стратегия разработки: достоинства и недостатки

Инкрементная стратегия представляет собой многократный проход этапов разработки с запланированным улучшением результата. Данная **стратегия основана на полном определении всех требований** к разрабатываемому ПС **в начале процесса** разработки. **Однако полный набор требований реализуется постепенно при последовательных циклах разработки**.

В инкрементной модели полные требования к системе делятся на различные сборки. **Имеют место несколько циклов разработки**, и вместе они составляют жизненный цикл «мульти-водопад». **Цикл разделен на** более мелкие легко создаваемые **модули**. Каждый **модуль проходит через фазы определения требований, проектирования, кодирования, внедрения и тестирования.** Результат каждого цикла называется ***инкрементом***.

Процедура разработки по инкрементной модели **предполагает выпуск** на первом большом этапе **продукта в базовой функциональности**, **а затем** уже **последовательное добавление новых функций**, так называемых «инкрементов».

Процесс продолжается *до тех пор*, пока не будет создана полная система.

К оглавлению

Особенностью инкрементной стратегии разработки является большое количество циклов разработки незначительной продолжительности цикла и небольших различиях между инкрементами соседних циклов.

Инкрементная стратегия обычно основана на объединении элементов каскадной модели и прототипирования. При этом использование прототипирования позволяет существенно сократить продолжительность разработки каждого инкремента и всего проекта в целом. Под прототипом понимается легко поддающаяся модификации и расширению рабочая модель предполагаемой системы (или программного средства), позволяющая пользователю получить представление о ключевых свойствах системы (или ПС) до ее полной реализации.

Преимущества инкрементной модели

1. не требуется заранее тратить средства, необходимые для разработки всего проекта, поскольку сначала выполняется разработка и реализация основной функции или функции из группы высокого риска;
2. в результате выполнения каждого инкремента получается функциональный продукт;
3. в конце каждой инкрементной поставки существует возможность пересмотреть риски, связанные с затратами и соблюдением установленного графика;
4. требования стабилизируются (посредством включения в процесс пользователей) на момент создания определенного инкремента;
5. короткая продолжительность создания инкремента, что приводит к сокращению сроков начальной поставки, позволяет снизить затраты на первоначальную и последующие поставки программного продукта;
6. предотвращение реализации громоздких спецификаций требований; стабильность требований во время создания определенного инкремента; возможность учета изменившихся требований;
7. снижение рисков по сравнению с каскадной стратегией;
8. включение в процесс пользователей, что позволяет оценить функциональные возможности продукта на более ранних этапах разработки и в конечном итоге приводит к повышению качества программного продукта, снижению затрат и времени на его разработку

Недостатки инкрементной модели

1. в модели не предусмотрены итерации в рамках каждого инкремента;
2. необходимость полного функционального определения системы или программного средства в начале ЖЦ для обеспечения планирования инкрементов и управления проектом;

К оглавлению

3. возможность текущего изменения требований к системе или программному средству, которые уже реализованы в предыдущих инкрементах;
4. сложность планирования и распределения работ;
5. проявление человеческого фактора, связанного с тенденцией к оттягиванию решения трудных проблем на поздние инкременты, что может нарушить график работ или снизить качество программного продукта.

Область применения инкрементной модели

1. когда основные требования к системе четко определены и понятны. В то же время некоторые детали могут дорабатываться с течением времени.
2. требуется ранний вывод продукта на рынок.
3. есть несколько рисковых фич или целей.
4. при разработке программ, связанных с низкой или средней степенью риска;
5. при выполнении проекта с применением новой технологии;
6. при разработке сложных проектов с заранее сформулированными требованиями; для них разработка системы или программного средства за один цикл связана с большими трудностями;

3. Эволюционная стратегия разработки: достоинства и недостатки

Эволюционная стратегия представляет собой **многократный проход этапов разработки**. Данная стратегия основана на **частичном определении требований** к разрабатываемому программному средству или системе **в начале процесса** разработки. **Требования постепенно уточняются** в последовательных циклах разработки. **Результат каждого цикла** разработки обычно представляет собой очередную **поставляемую версию программного средства** или системы.

Следует отметить, что в общем случае для эволюционной стратегии **характерно** существенно **меньшее количество циклов** разработки **при большей продолжительности цикла по сравнению с инкрементной** стратегией. При этом **результат каждого цикла** разработки (очередная версия программного средства или системы) гораздо **сильнее отличается от результата предыдущего цикла**.

Как и при инкрементной стратегии, при реализации эволюционной стратегии зачастую **используется прототипирование**. В данном случае основной **целью прототипирования** является **обеспечение полного понимания требований**. Оно позволяет итеративно уточнять требования к продукт. Использование прототипирования наиболее эффективно в тех случаях, когда в проекте применяются новые концепции или новые технологии, так как в этих случаях достаточно сложно полностью и корректно

К оглавлению

разработать детальные технические требования к системе или программному средству на ранних стадиях цикла разработки.

Основными **достоинствами эволюционной стратегии**, проявляемыми при разработке **соответствующего** ей проекта, являются:

1. возможность уточнения и внесения новых требований в процессе разработки;
2. пригодность промежуточного продукта для использования;
3. возможность управления рисками;
4. обеспечение широкого участия пользователя в проекте, начиная с ранних этапов, что минимизирует возможность разногласий между заказчиками и разработчиками и обеспечивает создание продукта высокого качества;
5. реализация преимуществ каскадной и инкрементной стратегий.

К **недостаткам эволюционной стратегии**, проявляемым при ее **несоответствующем** выборе, следует отнести:

1. неизвестность точного количества необходимых итераций и сложность определения критериев для продолжения процесса разработки на следующей итерации, это может вызвать задержку реализации конечной версии системы или программного средства;
2. сложность планирования и управления проектом;
3. необходимость активного участия пользователей в проекте, что реально не всегда осуществимо;
4. необходимость в мощных инструментальных средствах и методах прототипирования;
5. возможность отодвигания решения трудных проблем на последующие циклы, что может привести к несоответствию полученных продуктов требованиям заказчиков.

Очевидно, что ряд недостатков эволюционной характерны и для инкрементной стратегии.

Области применения эволюционной стратегии определяются ее достоинствами и ограничены ее недостатками. **Использование данной стратегии наиболее эффективно в следующих случаях:**

1. при разработке проектов, для которых требования слишком сложны, неизвестны заранее, непостоянны или требуют уточнения;
2. при разработке сложных проектов, в том числе:
   - больших долгосрочных проектов;
   - проектов по созданию новых, не имеющих аналогов ПС или систем;

К оглавлению

- проектов со средней и высокой степенью рисков;
- проектов, для которых нужна проверка концепции, демонстрация технической осуществимости или промежуточных продуктов;

3. при разработке проектов, использующих новые технологии.

4. Базовые стратегии разработки ПО, выбор модели

На начальном этапе развития вычислительной техники ПС разрабатывались по принципу **«кодирование – устранение ошибок».** Очевидно, что недостатками данной модели являются:

- неструктурированность процесса разработки ПС;
- ориентация на индивидуальные знания и умения программиста;
- сложность управления и планирования проекта;
- большая длительность и стоимость разработки;
- низкое качество программных продуктов;
- высокий уровень рисков проекта;

Для устранения или сокращения вышеназванных недостатков к ***настоящему времени*** созданы и **широко используются три базовые стратегии разработки ПО**:

- каскадная;
- инкрементная;
- эволюционная.

Выбор той или иной стратегии определяется характеристиками:

- проекта;
- требований к продукту;
- команды разработчиков;
- команды пользователей.

Каждая из стратегий разработки имеет как достоинства, так и недостатки, определяемые правильностью выбора данной стратегии по отношению к конкретному проекту. Следует подчеркнуть, что одни и те же свойства стратегии могут проявлять себя как достоинства при выборе стратегии к соответствующему ей проекту и как ее недостатки, если стратегия выбрана неверно.

К оглавлению

5. Гибкие технологии разработки ПО: Agile манифест

Гибкая методология разработки (англ. **Agile** software development, agile-методы) — одна из методологий **итеративной** и **пошаговой разработки** ПО. **Методология гибкой разработки определяет систему методов проектирования, разработки и тестирования на протяжении всего жизненного цикла ПО**. Методы гибкой разработки основаны на **оперативном реагировании на изменения** за счет применения адаптивного планирования, совместной выработки требований, рационализации самоорганизующихся кроссфункциональных групп разработчиков, а также пошаговой разработки ПО с четкими временными рамками.

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

Большинство гибких методологий **нацелены на минимизацию рисков путём сведения разработки к серии коротких циклов**, называемых *итерациями*, которые обычно длятся две-три недели. Т.к. **по завершению каждой итерации заказчик принимает результаты и выдает новые или корректирующие требования,** т.е. контролирует разработку и может на неё сразу влиять.

Каждая итерация сама по себе выглядит как **программный проект в миниатюре** и **включает все задачи, необходимые для выдачи мини-прироста по функциональности**: планирование, анализ требований, проектирование, программирование, тестирование и документирование. **По окончании каждого этапа** разработки **должен появляться "осязаемый" продукт** или **часть функционала**, которую можно посмотреть, протестировать и выдать дополнительные или корректирующие меры.

На основе проделанной работы, **после каждого этапа, команда подводит итоги и собирает новые требования**, **на основании чего вносит корректировки в план разработки программного обеспечения.**

Одной из основных идей Agile, **является взаимодействие внутри команды и с заказчиком лицом к лицу**, что позволяет быстро принимать решения и минимизирует риски разработки программного обеспечения. **Причем в команду входит представитель заказчика** (англ. product owner - полномочный представитель заказчика или сам заказчик, представляющий требования к продукту).

МАНИФЕСТ

К оглавлению

«Манифест гибкой методологии разработки программного обеспечения» **был выпущен и принят в феврале 2001 года** (штат ЮТА США, лыжный курорт The Lodge at Snowbird) на встрече 17 независимых практиков нескольких методик программирования, именующих себя «Agile Alliance».

Данный манифест определяет **4 основные ценности и 12 принципов** для методологий, базирующихся на нем, а также **дает альтернативное видение подхода к разработке программного обеспечения** в отличие от крупных и известных методов и методологий, **но не является сам по себе методологией**.

Выпуск манифеста дал новый толчок к развитию гибких методологий, заложил основы, можно сказать конституцию **гибкого подхода** к разработке программного обеспечения.

Agile-манифест разработки программного обеспечения:

Основной метрикой agile-методов **является рабочий продукт**. Отдавая предпочтение непосредственному общению, agile-методы **уменьшают объём письменной документации** по сравнению с другими методами. Это привело к критике этих методов как недисциплинированных.

Мы постоянно открываем для себя более совершенные методы разработки программного обеспечения, занимаясь разработкой непосредственно и помогая в этом другим. Благодаря проделанной работе мы смогли **осознать, что**:

- **Люди и взаимодействие важнее процессов и инструментов**
- **Работающий продукт важнее исчерпывающей документации**
- **Сотрудничество с заказчиком важнее согласования условий контракта**
- **Готовность к изменениям важнее следования первоначальному плану**

То есть, не отрицая важности того, что справа, мы всё-таки больше ценим то, что слева.

Основополагающие принципы Agile-манифеста:

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

К оглавлению

2. Изменение требований приветствуется, даже на поздних стадиях разработки. Agile-процессы позволяют использовать изменения для обеспечения заказчику конкурентного преимущества.
3. Работающий продукт следует выпускать как можно чаще, с периодичностью от пары недель до пары месяцев.
4. На протяжении всего проекта разработчики и представители бизнеса должны ежедневно работать вместе.
5. Над проектом должны работать мотивированные профессионалы. Чтобы работа была сделана, создайте условия, обеспечьте поддержку и полностью доверьтесь им.
6. Непосредственное общение является наиболее практичным и эффективным способом обмена информацией, как с самой командой, так и внутри команды.
7. Работающий продукт — основной показатель прогресса.
8. Инвесторы, разработчики и пользователи должны иметь возможность поддерживать постоянный ритм бесконечно. Agile помогает наладить такой устойчивый процесс разработки.
9. Постоянное внимание к техническому совершенству и качеству проектирования повышает гибкость проекта.
10. Простота — искусство минимизации лишней работы — крайне необходима.
11. Самые лучшие требования, архитектурные и технические решения рождаются у самоорганизующихся команд.
12. Команда должна систематически анализировать возможные способы улучшения эффективности и соответственно корректировать стиль своей работы.

6. Экстремальное программирование (XP): тестирование и рефакторинг

Экстремальное программирование или XP, **eXtreme Programming — гибкая методология** разработки программного обеспечения. **Автор** методики — **американский разработчик Кент Бек. В конце 90-х годов он руководил проектом** Chrysler Comprehensive Compensation System и там **впервые применил практики экстремального программирования**. **Автор XP не придумал ничего нового, а взял лучшие практики гибкой разработки и усилил до максимума**. Поэтому программирование и зовется экстремальным.

XP отличается от других гибких методологий тем, что **применимо только в области разработки программного обеспечения**. Оно не может быть использовано в другом бизнесе или повседневной жизни, как scrum, kanban или lean.

К оглавлению

Цель методики **XP**— **справиться с постоянно меняющимися требованиями** к программному продукту **и повысить качество разработки**. **Поэтому XP хорошо подходит для сложных и неопределенных проектов.**

Преимущества экстремального программирования имеют смысл, когда команда полноценно использует хотя бы одну из практик XP. Итак, ради чего стоит стараться:

- заказчик получает именно тот продукт, который ему нужен, даже если в начале разработки сам точно не представляет его конечный вид
- команда быстро вносит изменения в код и добавляет новую функциональность за счет простого дизайна кода, частого планирования и релизов
- код всегда работает за счет постоянного тестирования и непрерывной интеграции
- команда легко поддерживает код, т.к. он написан по единому стандарту и постоянно рефакторится
- быстрый темп разработки за счет парного программирования, отсутствия переработок, присутствия заказчика в команде
- высокое качество кода
- снижаются риски, связанные с разработкой, т.к. ответственность за проект распределяется равномерно и уход/приход члена команды не разрушит процесс
- затраты на разработку ниже, т.к. команда ориентирована на код, а не на документацию и собрания

Несмотря на все плюсы, XP не всегда работает и имеет ряд слабых мест. Итак, экстремальное программирование — **недостатки**:

- успех проекта зависит от вовлеченности заказчика, которой не так просто добиться
- трудно предугадать затраты времени на проект, т.к. в начале никто не знает полного списка требований
- успех XP сильно зависит от уровня программистов, методология работает только с senior специалистами
- менеджмент негативно относится к парному программированию, не понимая, почему он должен оплачивать двух программистов вместо одного
- регулярные встречи с программистами дорого обходятся заказчикам
- требует слишком сильных культурных изменений
- из-за недостатка структуры и документации не подходит для крупных проектов
- т.к. гибкие методологии функционально-ориентированные, нефункциональные требования к качеству продукта сложно описать в виде пользовательских историй.

К оглавлению

Методология XP строится вокруг четырех процессов: кодирования, тестирования, дизайна и слушания. Кроме того, экстремальное программирование имеет ценности: простоту, коммуникацию, обратную связь, смелость и уважение.

ТЕСТИРОВАНИЕ И РЕФАКТОРИНГ

Тестирование

В отличие от большинства остальных методологий **тестирование в XP – одно из важнейших составляющих**. **Экстремальный подход предполагает**, что **ТЕСТЫ ПИШУТСЯ ДО НАПИСАНИЯ КОДА**. **Каждый модуль обязан иметь unit test**– тест данного модуля.

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

Для старого кода тесты пишутся по мере необходимости: когда нужно добавить новую функциональность, исправить ошибку или переработать часть старого кода.

Рефакторинг

Рефакторинг — это **оптимизация существующего кода** в сторону упрощения, процесс постоянного **улучшения дизайна системы**, чтобы привести его в соответствие новым требованиям. Рефакторинг **включает удаление дублей кода, повышение связности и снижение сопряжения**. Сохраняя код прозрачным и определяя его элементы всего один раз, программисты сокращают число ошибок, которые впоследствии придется устранять.

При реализации каждого нового свойства системы программист должен подумать над тем, можно ли упростить существующий код и как это поможет реализовать новое свойство. Кроме того, **нельзя совмещать рефакторинг с дизайном**: если создается новый код, рефакторинг надо отложить.

XP предполагает постоянные рефакторинги, поэтому дизайн кода всегда остается простым.

К оглавлению

7. Экстремальное программирование (XP): code style и метафора системы

Code-style

Все члены команды в ходе работы **должны соблюдать требования общих стандартов оформления кода**. **Благодаря этому**: члены команды **не тратят время на споры** о вещах, которые фактически никак не влияют на скорость работы над проектом; **обеспечивается эффективное выполнение остальных практик**.

Если в команде **не используются единые стандарты** оформления кода, разработчикам **становится сложнее выполнять рефакторинг**; **при смене партнёров** в парах **возникает больше затруднений**; в общем и целом, **продвижение проекта затрудняется**.

В рамках XP **необходимо добиться того, чтобы было сложно понять, кто является автором того или иного участка кода**, — вся команда работает унифицировано, как один человек.

Команда должна сформировать набор правил, а затем **каждый член команды должен следовать этим правилам** в процессе написания кода. **Перечень правил не должен быть исчерпывающим или слишком объёмным**. Задача состоит в том, чтобы **сформулировать общие указания**, благодаря которым код станет понятным для каждого из членов команды.

Метафора системы

Метафора системы даёт команде представление о том, каким образом система работает в настоящее время, в каких местах добавляются новые компоненты, и какую форму они должны принять.

Разработчикам необходимо проводить анализ программного обеспечения для того, **чтобы понять, в каком месте системы необходимо добавить новую функциональность, и с чем будет взаимодействовать новый компонент.**

Подбор хорошей метафоры облегчает для группы разработчиков **понимание того, каким образом устроена система.** Иногда сделать это непросто. **Хорошая метафора системы означает простоту именования классов и переменных.**

К оглавлению

8. Экстремальное программирование (XP): парное программирование и continuous integration

Парное программирование

Парное программирование **предполагает**, что **весь код создается парами программистов, работающих за одним компьютером.** Один из них работает непосредственно с текстом программы, другой просматривает его работу и следит за общей картиной происходящего. При необходимости клавиатура свободно передаётся от одного к другому.

В течение работы над проектом **пары не фиксированы**: рекомендуется их перемешивать, **чтобы каждый** программист в команде **имел хорошее представление обо всей системе.** Таким образом, парное программирование **усиливает взаимодействие внутри команды.**

Преимущества:

- **Повышение дисциплины** (В паре чаще «делают то, что нужно» и реже устраивают длинные перерывы)
- **Лучший код** (В паре менее склонны к неудачным решениям и производят более качественный код.)
- **Гибкий поток работы**
- **Высокий боевой дух**
- **Коллективное владение кодом** (Каждый несёт ответственность за весь код. Таким образом, каждый вправе вносить изменения в любой участок. Важное преимущество коллективного владения кодом заключается в том, что оно ускоряет процесс разработки, поскольку, при появлении ошибки, её может устранить любой программист)
- **Наставничество** (Каждый, даже начинающий программист, знает что-то, чего не знают другие)
- **Командный дух**
- **Меньше прерываний**
- **Экономическая обоснованность**
- **Высокое качество дизайна**
- **Обратная связь**
- **Непрерывность проверки кода**
- **Обучение** (Программисты постоянно обмениваются знаниями)

Недостатки:

К оглавлению

- **Отсутствует возможность сосредоточиться** (Непрерывно отвлекают)

Continuous integration

Непрерывная интеграция (CI, англ. **Continuous Integration**) — **практика разработки программного обеспечения**, которая заключается **в постоянном слиянии рабочих копий в общую основную ветвь разработки** (до нескольких раз в день) и выполнении частых автоматизированных сборок проекта для скорейшего выявления потенциальных дефектов и решения интеграционных проблем.

Если выполнять интеграцию разрабатываемой системы достаточно часто, то можно избежать большей части связанных с ней проблем. В традиционных методиках интеграция, как правило, **выполняется в самом конце** работы над продуктом, когда считается, что все составные части разрабатываемой системы полностью готовы. **В XP интеграция кода всей системы выполняется несколько раз в день**, **после того, как разработчики убедились в том, что все тесты модулей корректно срабатывают.**

Это значит, что **новые части кода сразу же встраиваются в систему** — команды XP заливают новый билд **каждые несколько часов и чаще**.

Во-первых, **сразу видно, как последние изменения влияют на систему**. Если новый кусок кода что-то сломал, то ошибку найти и исправить в разы проще, чем спустя неделю.

Во-вторых, команда всегда работает с последней версией системы.

9. Экстремальное программирование (XP): соглашение о кодировании и коллективное владение кодом

Соглашение о кодировании

НЕ ВИЖУ РАЗНИЦЫ С CODE-STYLE

Стандарт оформления кода описывает различные аспекты создания и сопровождения исходных текстов программ. В качестве примера, можно перечислить следующие аспекты: правила именования переменных, стиль отступов, способ расстановки скобок, использование пробелов при оформлении арифметических выражений, стиль комментариев и так далее.

Все должны подчиняться общим стандартам кодирования — форматирование кода, именование классов, переменных, констант, стиль комментариев.

К оглавлению

Вышесказанное означает, что все члены команды должны договориться об общих стандартах кодирования. Неважно каких. Правило заключается в том, что все им подчиняются. Те, кто не желает их соблюдать покидает команду.

Коллективное владение кодом(collective code ownership)

Коллективное владение означает, что **каждый член команды несёт ответственность за весь исходный код.** Таким образом, **каждый вправе вносить изменения в любой участок программы. Обязательное правило**: если программист внес изменения, и система после этого работает некорректно, то именно этот программист должен исправить ошибки.

Преимущества:

- **Любой разработчик может изменять любой код** для **расширения функциональности и исправления ошибок**.
- Коллективное владение кодом **стимулирует разработчиков подавать идеи для всех частей проекта**, а не только для своих модулей.
- Коллективное владение кодом значительно **упрощает внесение изменений** и **снижает риск, связанный с высокой специализацией** того или иного члена команды.
- **ускоряет процесс разработки**, поскольку **при появлении ошибки** её может **устранить любой** программист.

10. Scrum: основные характеристики

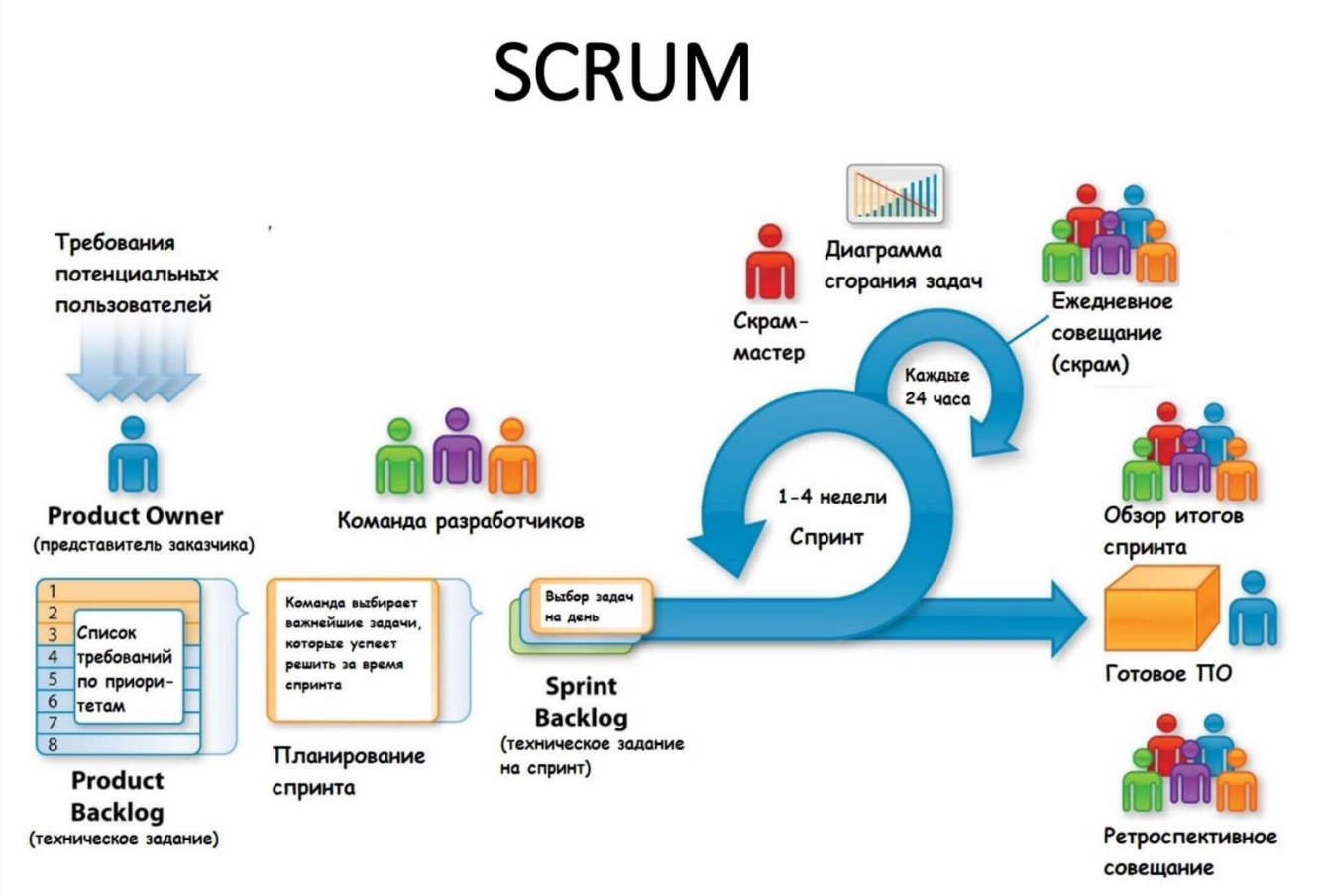
Простое мультяшное объяснение scrum'а:
https://www.youtube.com/watch?v=BHhr1aMgKPk

ScrumGuides_official

Неплохие статьи:

- https://xbsoftware.ru/blog/zhiznennyj-tsykl-po-skram-po-shagam/
- http://www.alexcouncil.com/scrum/

К оглавлению



Scrum (skrʌm «схватка») — это **методология управления** проектами. Основной **акцент** использования данной методологии делается на **качественном контроле процесса разработки**.

Впервые методология SCRUM была представлена на общее обозрение **задокументированной, четко сформированной** и описанной совместно Швабером и Сазерлендом на OOPSLA'**95** в Остине. Швабер и Сазерленд на протяжении следующих лет работали вместе, чтобы обработать и описать весь их опыт и лучшие практические образцы для индустрии в одно целое, в ту методологию, что известна сегодня как SCRUM.

Scrum является одним из возможных способов **реализации гибкой (Agile) методологии** разработки. В отличие от каскадной модели жизненного цикла ПО, отличительной **особенностью Scrum является итеративность**.

Процесс разработки **разбивается на отдельные этапы, результатом каждого** из которых является **готовый продукт. В конце каждого этапа** (в терминологии Scrum — спринта) готовый продукт **предоставляется заказчику**. Полученный от заказчика отзыв позволяет выявить возможные проблемы или пересмотреть некоторые аспекты первоначального плана. Таким образом, **Scrum позволяет наилучшим образом следовать принципам Agile-разработки.**

Основой Scrum является Sprint, в течении которого выполняется работа над продуктом**. По окончанию Sprint** должна быть получена **новая рабочая версия продукта**. **Sprint всегда ограничен по времени** (1-4 недели) и имеет одинаковую продолжительность на протяжении все жизни продукта.

Перед началом каждого Sprint производится Sprint Planning, на котором производится оценка содержимого Product Backlog и **формирование Sprint Backlog**, который содержит задачи, которые должны быть выполнены в текущем спринте. Каждый спринт должен иметь цель, которая является мотивирующим фактором и достигается с помощью вы полнения задач из Sprint Backlog.

Каждый день производится Daily Scrum, на котором каждый член команды отвечает на вопросы «что я сделал вчера?», «что я планирую сделать сегодня?», «какие препятствия на своей работе я встретил?». **Задача Daily Scrum — определение статуса и прогресса работы над Sprint**, раннее обнаружение возникших препятствий, выработка решений по изменению стратегии, необходимых для достижения целей Sprint'a.

По окончанию Sprint'a производятся Sprint Review(Обзор) и Sprint Retrospective, **задача которых оценить эффективность (производительность) команды в прошедшем Sprint'e, спрогнозировать ожидаемую эффективность (производительность) в следующем спринте, выявлении имеющихся проблем, оценки вероятности завершения всех необходимых работ по продукту и другое.**

Scrum: основные характеристики

- Самоорганизующиеся команды
- Продукт разрабатывается серией “спринтов”, каждый не больше месяца
- Все требования записываются в виде единого списка “бэклога продукта”
- Инженерные практики не являются частью Скрам
- Использует простые правила для создания гибкой среды разработки проектов
- Один из “Agile процессов”

Нюансы

- Нет ориентации на клиента. Не все заказчики будут готовы к определенным стандартам Scrum.

К оглавлению

- Не учитывается система реагирования на риски. Команда может заложить какое-то доп.время на выполнение задач, но при сильных отклонениях от плана, система встанет.
- Команда и человеческие качества. Так как упор сделан на самоорганизующуюся команду, то все участники должны обладать высоким уровнем ответственности и соответствующей мотивацией. Создание такой команды, очень сложная задача.
- Скрам-мастер. Человек отвечающий за процессы и мотивацию команды, должен чувствовать всех участников и связи внутри группы. Это редкий специалист, которого также тяжело найти на рынке

11. Scrum: структура + роли

Структура scrum:

- Роли
  - **Владелец продукта (Product Owner)**-ставит задачи, определяет приоритеты по задачам, взаимодействует с заказчиком.
  - **Скрам-мастер (Scrum Master)**– человек, который отвечает за процессы внутри команды, координирует работу, следит за внутренней атмосферой. Планирует спринт, организует скрам митинг, участвует в демонстрации результатов в конце каждого спринта.
  - **Команда разработчиков (Delivery Team)**– 7±2 человек(оптимально), которые реализуют требования владельца продукта.
- Артефакты
- Процессы (ритуалы)

12. Scrum: структура + ритуалы

Структура scrum:

- Роли
- Артефакты
- Процессы (ритуалы)
  - **Планирование спринта.** Команда со скрам-мастером планирует план работ на будущий спринт, то есть составляет беклог спринта (список) задач.
  - **Обзор спринта.** Демонстрация инкремента продукта после каждого спринта. Команда показывает рабочую функциональность владельцу продукта (и заказчику по запросу), а тот, в свою очередь, вносит изменения в требования, если они необходимы.

К оглавлению

- **Ретроспектива.** Обзор прошедшего спринта с целью улучшения процессов. Команда, скрам-мастер и владелец продукта обсуждают прошедший спринт, делают выводы, думают над тем, что можно было бы улучшить.
- **Скрам митинг**– ежедневная планерка, летучка, где разбирается ход работы спринта. Что сделали, есть ли проблемы, что планируется сделать. Не более 15 минут на собрание. Все участники команды должны высказаться. Скрам-мастер следит за таймингом и выступлением каждого.
- **Спринт.** Как правило двухнедельный этап времени, в течении которого команда успевает разработать готовый для демонстрации функционал.

13. Scrum: структура + артефакты

Структура scrum:

- Роли
- Артефакты
  - **Беклог продукта.** Список требований с расставленными приоритетами и трудозатратами.
  - **Беклог спринта.** Часть беклога продукта, то есть несколько задач, которые реально уместить в один спринт.
  - **Инкремент продукта**. Готовая часть продукта для демонстрации. В digital проектах, это может быть функциональность. К примеру, рабочая форма регистрации на сайте, которую можно показать.
  - **Burndown Chart — диаграмма сгорания.** Эта диаграмма показывает, сколько задач осталось до завершения спринта на временной шкале (и сколько уже сделано). По вертикали — количество задач, по горизонтали — время. Цель команды: «сжечь» все задачи до того, как приблизится дедлайн. На графике показаны: «идеальная кривая», если бы задачи выполнялись по N штук в день каждый день; «фактическая кривая», которая показывает реальное количество выполненных в конкретный день задач. Вы смотрите, насколько фактическая кривая отличается от идеальной, по ситуации корректируете действия команды.
- Процессы (ритуалы)

14. Scrum: масштабируемость

Что важно знать, прежде чем задумываться о масштабировании

К оглавлению

Когда у Scrum-Master'а возникают мысли о масштабировании, это говорит о том, что команда столкнулась с тремя ключевыми проблемами, которые нужно как можно скорее решить:

1. Стало сложно планировать спринт: Product Owner не справляется с составлением требований для всех, поскольку команда слишком большая. Также он слишком длинный.
2. Ретроспектива теперь больше похожа на зомби-апокалипсис: одна половина команды пытается перекричать другую, а Scrum Master изо всех сил старается контролировать ситуацию, но получается плохо.
3. Постоянно что-то «впихивается» в текущий спринт, хотя, на самом деле, с дополнительным объемом задач справиться невозможно.

6 советов по масштабированию небольших команд.

1. Разбить ScrumTeam на маленькие команды с полным набором скиллов (минимальная команда может состоять из двух человек, хоть Scrum Guide и советует начинать с трех).
2. Настроить полностью автоматизированный workflow(рабочий процесс) в Jira (или любом другом трекере).
3. Команда должна слаженно работать даже в отсутствие Scrum Master-а. Главная цель любого Scrum-Master'а — сделать так, чтобы команда отлично работала без его активного участия впроцессе, особенно когда дело касается масштабируемых проектов.
4. Обратить внимание на Self Selection process — процесс, когда участники команд сами выбирают, в какой части продукта они хотят работать. Эта практика хороша для поднятия боевого духа и мотивации сотрудников.
5. Разделить команду на самодостаточные блоки, по которым они закрывают те или иные проблемы. Можете разбить большой продукт на функциональные куски и сформировать команды, которые будут работать в разном окружении. Здесь как раз и пригодится SelfSelection process.
6. Не давайте сотрудникам потерять ощущение, что без их работы мир рухнет! Нет более или менее важных команд. Все крутые и все равны!

Небольшие команды чаще показывают хорошие результаты, чем большие, поэтому необходимо по возможности вести разработку компактными командами.

К сожалению, часто бывает, что объем проекта и сроки его реализации просто не позволяют вести разработку 5–9 людьми и приходится задействовать несколько команд.

Привлечение нескольких команд:

к оглавлению

Для организации разработки больших проектов или портфеля проектов необходимо масштабировать Scrum на следующий уровень. Со стороны команд разработки это выливается в проведение **Scrum of Scrum**.

На этот митинг собираются Scrum-Master'а (SM) в качестве представителей конкретных команд. Организует собрание руководитель программы (Program Manager, PM). При использовании дивизионной организационной структуры на данном уровне он также может являться руководителем соответствующего дивизиона (подразделения).

В качестве базовой структуры митинга можно предложить **каждому Scrum-Master'а** ответить наследующие вопросы.

1. Что было сделано с прошлого Scrum of Scrum?
2. Какие были проблемы?
3. Что будет сделано к следующему Scrum of Scrum?

При этом **акцент нужно сделать на проблемах**, которые команда не может решить сама и вынуждена передавать выше

15. Тестирование ПО: разработка теста для функции по ее спецификации

Может здесь - https://www.intuit.ru/studies/courses/48/48/lecture/1440?page=4

Ту полный брЭд

Тестирование программного обеспечения (Software Testing) – проверка соответствия между реальным и ожидаемым поведением программы, осуществляемая на конечном наборе тестов, выбранном определенным образом. [IEEE Guide to Software Engineering Body of Knowledge, SWEBOK, 2004]

В более широком смысле, тестирование **— это одна из техник контроля качества**, **включающая** в себя активности по **планированию** работ (Test Management), **проектированию** тестов (Test Design), **выполнению тестирования** (Test Execution) **и анализу полученных результатов** (Test Analysis)

Тест — это выполнение определенных условий и действий, необходимых для проверки работы тестируемой функции или её части.

Спецификация — это текстовый файл с описанием того, что нужно протестировать в тестовых данных. В ней указывается какие результаты должна получить программа.

К оглавлению

Тестовый код находит реальные, вычисленные на живом коде результаты. А тестовый движок производит сверку спецификации и вычисленных результатов.

Такой подход позволяет декларативно создавать тесты. Спецификации легко читаются и дополняются при изменении требований. Тестовый код получается компактным. Его легко поддерживать и расширять.

Файл состоит из:

- Комментариев в начале файла. Комментарии могут состоять из любых символов кроме $.
- Проверяемых свойств. Имя каждого свойства начинается с символа $. Конец значения свойства определяется либо началом следующего свойства, либо концом файла.

Уровни тестирования:

1. Модульное тестирование (Unit Testing)

Компонентное (модульное) тестирование проверяет функциональность и ищет дефекты в частях приложения, которые доступны и могут быть протестированы по-отдельности (модули программ, объекты, классы, функции и т.д.).

2. Интеграционное тестирование (Integration Testing)

Проверяется взаимодействие между компонентами системы после проведения компонентного тестирования.

3. Системное тестирование (System Testing)

Основной задачей системного тестирования является проверка как функциональных, так и не функциональных требований в системе в целом. При этом выявляются дефекты, такие как неверное использование ресурсов системы, непредусмотренные комбинации данных пользовательского уровня, несовместимость с окружением, непредусмотренные сценарии использования, отсутствующая или неверная функциональность, неудобство использования и т.д.

4. Операционное тестирование (Release Testing).

Даже если система удовлетворяет всем требованиям, важно убедиться в том, что она удовлетворяет нуждам пользователя и выполняет свою роль в среде своей эксплуатации.

К оглавлению

5. Приемочное тестирование (Acceptance Testing)

Формальный процесс тестирования, который проверяет соответствие системы требованиям и проводится с целью:

- определения удовлетворяет ли система приемочным критериям;
- вынесения решения заказчиком или другим уполномоченным лицом принимается приложение или нет.

Функциональные виды тестирования

- Функциональное тестирование (Functional testing)
  рассматривает заранее указанное поведение и основывается на анализе спецификаций функциональности компонента или системы в целом.
- Тестирование пользовательского интерфейса (GUI Testing)
  функциональная проверка интерфейса на соответствие требованиям — размер, шрифт, цвет и тп
- Тестирование безопасности (Security and Access Control Testing)
  стратегия тестирования, используемая для проверки безопасности системы, а также для анализа рисков, связанных с обеспечением целостного подхода к защите приложения, атак хакеров, вирусов, несанкционированного доступа к конфиденциальным данным.
- Тестирование взаимодействия (Interoperability Testing)
  функциональное тестирование, проверяющее способность приложения взаимодействовать с одним и более компонентами или системами и включающее в себя тестирование совместимости (compatibility testing) и интеграционное тестирование

Нефункциональные виды тестирования:

- Все виды тестирования производительности:
  - нагрузочное тестирование (Performance and Load Testing)
  - стрессовое тестирование (Stress Testing)
  - тестирование стабильности или надежности (Stability / Reliability Testing)
  - объемное тестирование (Volume Testing)
    задачей является получение оценки производительности при увеличении объемов данных в базе данных приложения
- Тестирование установки (Installation testing)
- Тестирование удобства пользования (Usability Testing)
- Тестирование на отказ и восстановление (Failover and Recovery Testing)
- Конфигурационное тестирование (Configuration Testing)

К оглавлению

Этапы тестирования:

1. Программа и требования поступают к тестировщику.
2. Он совершает необходимые операции, ведет наблюдение за тем, как ПО выполняет поставленные задачи.
3. По результатам проверки формируется список соответствий и несоответствий.
4. Используя полученную информацию, можно либо улучшить существующий софт, либо обновить требования к разрабатываемой программе.

/*

1. Юзабилити-тестирование (проверка эргономичности)помогает определить: удобен ли сайт или пользовательский интерфейс для его предполагаемого применения.
2. Создание чек-листа — подготовка набора тестов, внесение необходимых предложений в разрабатываемые требования (с точки зрения качества).
3. Тестирование. Получив готовую для проверки программу или её часть, специалист проверяет её соответствие требованиям на выбранном наборе тестов. В случае обнаружения дефектов — передаёт разработчикам набор задач, необходимых для улучшения продукта до состояния соответствия требованиям.
4. Верификация — проверка, которая показывает: были ли исправлены ошибки, обнаруженные в результате тестов.
5. Тест производительности (performance testing) проводится на стендах, где в дальнейшем будет эксплуатироваться софт. Цель — выявление проблем стенда, имитация работы пользователей, проверка на стрессоустойчивость. Позволяет убедиться, что приложение/система справится с реальной нагрузкой в будущем.

*/

16. Тестирование ПО: статическое, динамическое тестирование

Статическое тестирование – тип тестирования, который предполагает, что программный код во время тестирования не будет выполняться. При этом само тестирование может быть как ручным, так и автоматизированным.

К оглавлению

Мы проверяем не работу программы, а сам код. Он вычитывается либо вручную, либо с помощью программ, которые анализируют код. На этом этапе можно найти неверные конструкции, неверные отношения объектов программы.

Статическое тестирование начинается на ранних этапах жизненного цикла ПО и является, соответственно, частью процесса верификации(процесс оценки системы или её компонентов с целью определения удовлетворяют ли результаты текущего этапа разработки условиям, сформированным в начале этого этапа). Для этого типа тестирования в некоторых случаях даже не нужен компьютер – например, при проверке требований.

Большинство статических техник могут быть использованы для «тестирования» любых форм документации, включая вычитку кода, инспекцию проектной документации, функциональной спецификации и требований.

Даже статическое тестирование может быть автоматизировано – например, можно использовать автоматические средства

Виды статического тестирования:

- вычитка исходного кода программы;
- проверка требований.

Динамическое тестирование – тип тестирования, который предполагает запуск программного кода. Таким образом, анализируется поведение программы во время ее работы.

Для выполнения динамического тестирования необходимо чтобы тестируемый программный код был написан, скомпилирован и запущен. При этом, может выполняться проверка внешних параметров работы программы: загрузка процессора, использование памяти, время отклика и т.д. – то есть, ее производительность.

Динамическое тестирование является частью процесса валидации(определение соответствия разрабатываемого ПО ожиданиям и потребностям пользователя, требованиям к системе) программного обеспечения.

Кроме того динамическое тестирование может включать разные подвиды, каждый из которых зависит от:

- Доступа к коду (тестирование черным, белым и серым ящиками).
- Уровня тестирования (модульное интеграционное, системное, и приемочное тестирование).

К оглавлению

- Сферы использования приложения (функциональное, нагрузочное, тестирование безопасности и пр.)

17. Тестирование ПО: методы белого и черного ящика

В зависимости от доступа разработчика тестов к исходному коду тестируемой программы различают «тестирование (по стратегии) белого ящика» и «тестирование (по стратегии) чёрного ящика».

Белый ящик

При тестировании белого ящика (также говорят — прозрачного ящика), разработчик теста имеет доступ к исходному коду программ и может писать код, который связан с библиотеками тестируемого программного обеспечения. Это типично для модульного тестирования, при котором тестируются только отдельные части системы. Оно обеспечивает то, что компоненты конструкции — работоспособны и устойчивы, до определённой степени. При тестировании белого ящика используются метрики покрытия кода или мутационное тестирование.

Тестирование по стратегии белого ящика — тестирование кода на предмет логики работы программы и корректности её работы с точки зрения компилятора того языка, на котором она писалась.Тестирование по стратегии белого ящика, также называемое техникой тестирования, управляемой логикой программы, позволяет проверить внутреннюю структуру программы. Исходя из этой стратегии , тестировщик получает тестовые данные путем анализа логики работы программы.

Техника Белого ящика включает в себя следующие методы тестирования:

1. покрытие решений
2. покрытие условий
3. покрытие решений и условий
4. комбинаторное покрытие условий

Черный ящик

При тестировании чёрного ящика, тестировщик имеет доступ к программе только через те же интерфейсы, что и заказчик или пользователь, либо через внешние интерфейсы, позволяющие другому компьютеру либо другому процессу подключиться к системе для тестирования. Например, тестирующий модуль может виртуально нажимать клавиши или кнопки мыши в тестируемой программе с помощью механизма взаимодействия

К оглавлению

процессов, с уверенностью в том, все ли идёт правильно, что эти события вызывают тот же отклик, что и реальные нажатия клавиш и кнопок мыши.

Как правило, тестирование чёрного ящика ведётся с использованием спецификаций или иных документов, описывающих требования к системе. Обычно в данном виде тестирования критерий покрытия складывается из покрытия структуры входных данных, покрытия требований и покрытия модели (в тестировании на основе моделей).

Тестирование чёрного ящика или поведенческое тестирование — стратегия (метод) тестирования функционального поведения объекта (программы, системы) с точки зрения внешнего мира, при котором не используется знание о внутреннем устройстве тестируемого объекта. Под стратегией понимаются систематические методы отбора и создания тестов для тестового набора. Стратегия поведенческого теста исходит из технических требований и их спецификаций.

18. Системы контроля версий: история развития (RCS, CVS, SVN и git), локальные, централизованные и распределенные системы

Система управления версиями (от англ. Version Control System, VCS или Revision Control System) — программное обеспечение для облегчения работы с изменяющейся информацией. Система управления версиями позволяет хранить несколько версий одного и того же документа, при необходимости возвращаться к более ранним версиям, определять, кто и когда сделал то или иное изменение, и многое другое.

Локальные системы контроля версий

RCS (Revision Control System, **Система контроля ревизий**) была разработана **в начале 1980-х** годов **Вальтером Тичи**. Система **позволяет хранить версии только одного файла**, таким образом управлять несколькими файлами приходится вручную.

Недостатки:

- Работа только с одним файлом, каждый файл должен контролироваться отдельно;
- Неудобный механизм одновременной работы нескольких пользователей с системой

Централизованные системы контроля версий

CVS

К оглавлению

CVS (Concurrent Versions System, **Система совместных версий**) **Дик Грун** разработал CVS в **середине 1980-х**.

Для хранения индивидуальных файлов CVS (также как и RCS) **использует файлы в RCS формате**, но **позволяет управлять группами файлов расположенных в директориях**. Также CVS **использует клиент-сервер** архитектуру в которой **вся информация о версиях хранится на сервере.**

Недостатки:

- Так как версии хранятся в файлах RCS нет возможности сохранять версии директорий.
- Перемещение, или переименование файлов не подвержено контролю версий

SVN

SVN (Subversion) **свободная централизованная система управления версиями**, официально выпущенная в **2004 году** компанией **CollabNet.**

Из наиболее **значительных изменений по сравнению с С**VS можно отметить:

- Атомарное внесение изменений (commit). В случае если обработка коммита была прервана не будет внесено никаких изменений.
- Переименование, копирование и перемещение файлов сохраняет всю историю изменений.
- Директории, символические ссылки и мета-данные подвержены контролю версий.
- Эффективное хранение изменений для бинарных файлов.

Распределенные системы контроля версий

В таких системах как Git, Mercurial, Bazaar или Darcs клиенты не просто выгружают последние версии файлов, а **полностью копируют весь репозиторий**. Поэтому в случае, когда "умирает" сервер, через который шла работа, **любой клиентский репозиторий может быть скопирован обратно на сервер, чтобы восстановить базу данных**. Каждый раз, **когда клиент забирает свежую версию** файлов, **он создаёт себе полную копию всех данных**.

К оглавлению

Кроме того, в большей части этих систем можно работать с несколькими удалёнными репозиториями, таким образом, можно одновременно работать по-разному с разными группами людей в рамках одного проекта.

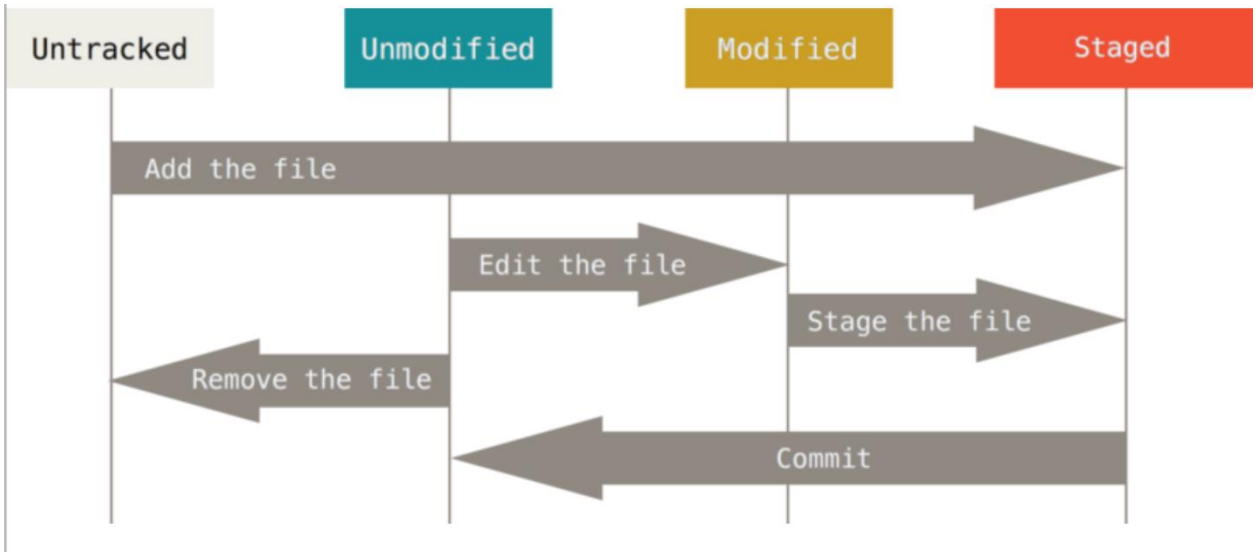
GIT

Git — распределённая система управления версиями. Проект был создан **Линусом Торвальдсом** для управления разработкой ядра Linux, первая версия **выпущена 7 апреля 2005 года.**

Основные требования к новой системе были следующими:

- Скорость
- Простота дизайна
- Поддержка нелинейной разработки (тысячи параллельных веток)
- Полная распределённость
- Возможность эффективной работы с такими большими проектами, как ядро Linux (как по скорости, так и по размеру данных)

19. Git: создание репозитория + фиксация изменений, стадии процесса



Для создания Git-репозитория вы можете использовать два основных подхода. Во-первых, импорт в Git уже существующего проекта или директории. Во-вторых, клонирование существующего репозитория с другого сервера.

Создание репозитория в существующей директории

К оглавлению

Если вы собираетесь начать использовать Git для существующего проекта, то вам необходимо перейти в директорию проекта и в командной строке ввести **$ git init**

Эта команда создаёт в текущей директории новую поддиректорию с именем.git, содержащую все необходимые файлы репозитория — основу Git-репозитория. На этом этапе ваш проект ещё **не находится** под версионным контролем.

Если вы хотите добавить под версионный контроль существующие файлы (в отличие от пустого каталога), вам стоит добавить их в индекс и осуществить первый коммит изменений. Добиться этого вы сможете запустив команду **$git add** несколько раз, указав индексируемые файлы, а затем выполнить **$git commit**

Клонирование существующего репозитория

Для получения копии существующего Git-репозитория, например, проекта, в который вы хотите внести свой вклад, необходимо использовать команду **$ git clone**. Вместо того, чтобы просто получить рабочую копию, Git получает копию практически всех данных, которые есть на сервере. При выполнении git clone с сервера забирается (pulled) каждая версия каждого файла из истории проекта. Фактически, если серверный диск выйдет из строя, **вы можете использовать любой из клонов на любом из клиентов, для того, чтобы вернуть сервер в то состояние, в котором он находился в момент клонирования**(вы можете потерять часть серверных перехватчиков (server-side hooks) и т.п., но все данные, помещённые под версионный контроль, будут сохранены)

Клонирование репозитория осуществляется командой **$ git clone [url]**. Например:

```
$ git clone https://github.com/libgit2/libgit2
```

Эта команда создаёт директорию “libgit2”, инициализирует в ней поддиректорию .git, скачивает все данные для этого репозитория и создаёт (checks out) рабочую копию последней версии. Если вы зайдёте в новую директорию libgit2, то увидите в ней файлы проекта, готовые для работы или использования.

Для того, чтобы клонировать репозиторий в директорию с именем, отличающимся от “libgit2”, необходимо указать желаемое имя, как параметр командной строки:

```
$ git clone https://github.com/libgit2/libgit2  mylibgit
```

Эта команда делает всё то же самое, что и предыдущая, только результирующий каталог будет назван mylibgit.

К оглавлению

20. Git: создание ветки (branch)

Ветка (branch) в Git — это легко перемещаемый указатель на один из этих коммитов. Имя основной ветки по умолчанию в Git — master.

Когда вы делаете коммиты, то получаете основную ветку, указывающую на ваш последний коммит. Каждый коммит автоматически двигает этот указатель вперед.

Создание новой ветки

Что же на самом деле происходит, когда вы создаете ветку? Всего лишь создается новый указатель для дальнейшего перемещения. Допустим вы хотите создать новую ветку с именем “testing” Вы можете это сделать командой git branch :

```
$ git branch testing
```

В результате создается новый указатель на тот же самый коммит, в котором вы находитесь.

Чтобы создать ветку и сразу переключиться на нее, можно выполнить команду git checkout с параметром -b:

```
$ git checkout -b branch2
```

Переключение веток

Чтобы переключиться на существующую ветку, выполните команду git checkout. Давайте переключимся на ветку “testing”:

```
$ git checkout testing
```

21. Git: объединение веток (merge)

Слияние — обычная практика для разработчиков, использующих системы контроля версий. Независимо от того, созданы ли ветки для тестирования, исправления ошибок или по другим причинам, слияние фиксирует изменения в другом месте. Слияние принимает содержимое ветки источника и объединяет их с целевой веткой. В этом процессе изменяется только целевая ветка. История исходных веток остается неизменной.

К оглавлению

Плюсы:

- простота;
- сохраняет полную историю и хронологический порядок;
- поддерживает контекст ветки.

Минусы:

- история коммитов может быть заполнена (загрязнена) множеством коммитов
- Git merge создает новый «Merge commit», который содержит историю обеих веток

Как это сделать

Слияние feature в ветку master:

$ git checkout master
$ git merge feature
or
$ git merge master feature

Это создаст новый «Merge commit» в ветке master, который содержит историю обеих веток.

22. Git: конфликты и способы их разрешения

Иногда процесс не проходит гладко**. Если вы изменили одну и ту же часть одного и того же файла поразному в двух объединяемых ветках, Git не сможет их чисто объединить**. Если ваше решение проблемы “name2” изменяет ту же часть файла, что и “name1”, вы получите конфликт слияния.

Git не создал коммит слияния автоматически. **Он остановил процесс до тех пор, пока вы не разрешите конфликт.** Если вы хотите посмотреть, какие файлы не прошли слияние (на любом этапе после возникновения конфликта), выполните команду **$ git status.**

Всё, что имеет отношение к конфликту слияния и что не было разрешено, отмечено как unmerged(неслитое). Git добавляет стандартные маркеры к файлам, которые имеют конфликт, так что вы можете открыть их вручную и разрешить эти конфликты.

В верхней части блока (всё что выше =======) это версия из HEAD (вашей ветки master, так как именно на неё вы перешли перед выполнением команды merge), всё, что находится в нижней части — версия в “name2”. **Чтобы разрешить конфликт, вы**

К оглавлению

должны либо выбрать одну из этих частей, либо как-то объединить содержимое по своему усмотрению.

Разрешив каждый **конфликт** во всех файлах, **запустите** **$ git add** для каждого файла, **чтобы отметить конфликт как решенный**. Подготовка (staging) файла помечает его для Git как разрешенный конфликт.

Выбрать одну из версий файла

Можно явным образом указать: какой файл выбирать. Подходит, только если одна из версий не нужна.

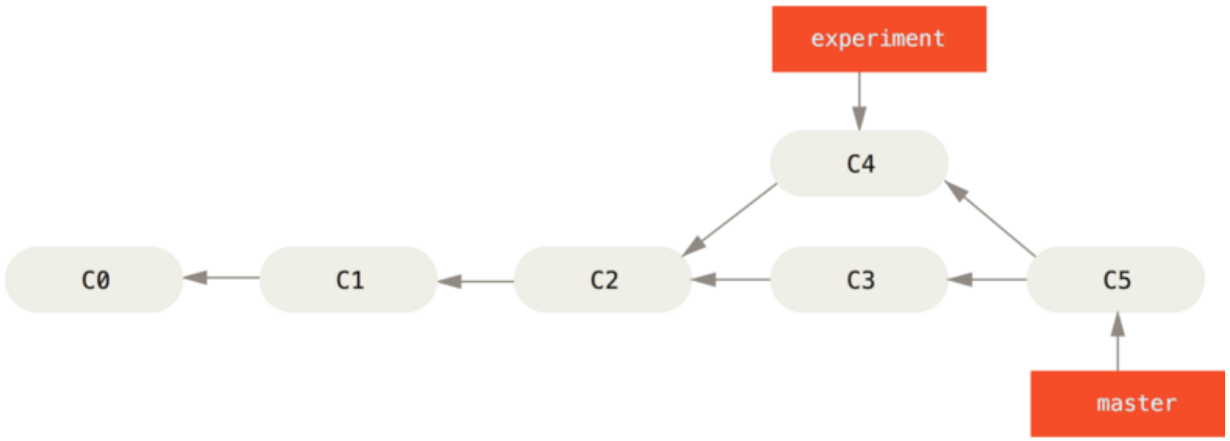
```
$ git checkout –ours(-2) a.txt
$ git checkout --theirs(-3)  a.txt
$ git add a.txt
```

Прерывание слияния

```
$ git merge --abort
```

Когда происходит слияние веток посредством merge, то все комиты сохраняются — сохраняются комментарии комита, его hash + как правило добавляется еще один искусственный комит.

Она осуществляет трехстороннее слияние между двумя последними снимками (snapshot) сливаемых веток (C3 и C4) и самого недавнего общего для этих веток родительского снимка (C2), создавая новый снимок (и коммит).



23. Git: работа с удаленным репозиторием (git fetch & git pull)

Удалённые репозитории представляют собой версии вашего проекта, сохранённые в интернете или ещё где-то в сети. У вас может быть несколько удалённых репозиториев, каждый из которых может быть доступен для чтения или для чтения-записи. Взаимодействие с другими пользователями предполагает управление удалёнными

К оглавлению

репозиториями, а также отправку и получение данных из них. Управление репозиториями включает в себя как умение добавлять новые, так и умение удалять устаревшие репозитории, а также умение управлять различными удалёнными ветками, объявлять их отслеживаемыми или нет и так далее.

Просмотр удалённых репозиториев

Для того, чтобы просмотреть список настроенных удалённых репозиториев, вы можете запустить команду **$ git remote**. Она выведет названия доступных удалённых репозиториев.

Просмотр удаленного репозитория

Если хотите получить побольше информации об одном из удалённых репозиториев, вы можете использовать команду **$ git remote show [remote-name]**

Добавление удалённых репозиториев

В предыдущих разделах мы уже упоминали и приводили примеры добавления удалённых репозиториев, сейчас рассмотрим эту операцию подробнее. Для того, чтобы добавить удалённый репозиторий и присвоить ему имя (shortname), просто выполните команду **$ git remote add [shortname] [url]**

Получение изменений из удалённого репозитория - Fetch

$ git fetch [remote-name]

Данная команда связывается с указанным удалённым проектом и забирает все те данные проекта, которых у вас ещё нет. После того как вы выполнили команду, у вас должны появиться ссылки на все ветки из этого удалённого проекта. Теперь эти ветки в любой момент могут быть просмотрены или слиты.

Важно отметить, что *команда git fetch забирает данные в ваш локальный репозиторий, но не сливает их с какими-либо вашими наработками и не модифицирует то, над чем вы работаете в данный момент*. Вам необходимо вручную слить эти данные с вашими, когда вы будете готовы.

Получение изменений из удалённого репозитория - Pull

Если у вас есть ветка, настроенная на отслеживание удалённой ветки, то вы можете использовать команду **$ git pull** чтобы автоматически получить изменения из удалённой ветви и слить их со своей текущей ветвью.

К оглавлению

Выполнение **$ git pull**, как правило, *извлекает (fetch) данные с сервера, с которого вы изначально склонировали, и автоматически пытается слить (merge) их с кодом, над которым вы в данный момент работаете*.

Удаление веток на удалённом сервере

Скажем, вы и ваши соавторы закончили с нововведением и слили его в ветку master на удалённом сервере (или в какую-то другую ветку, где хранится стабильный код). Вы можете удалить ветку на удалённом сервере:

```
$ git push origin --delete branchname
```

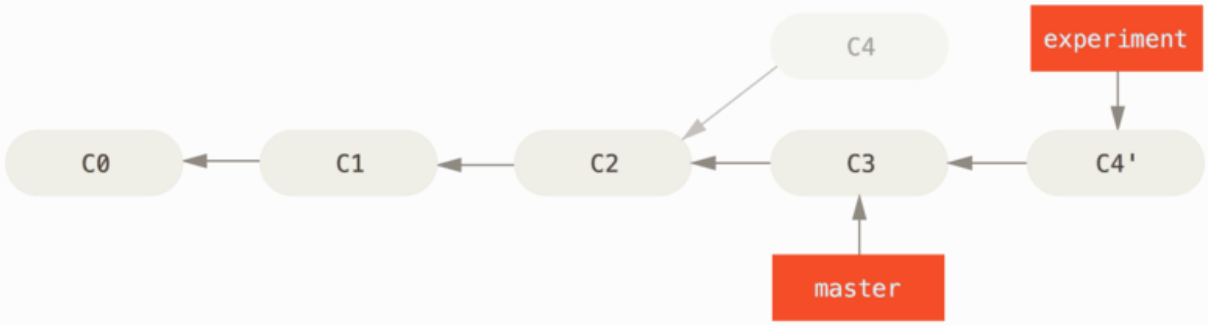
24. Git: работа с удаленным репозиторием (git rebase)

Вы можете взять те изменения, что были представлены в C4 и применить их поверх C3. В Git это называется **перебазированием** (rebasing). С помощью команды **rebase** вы можете взять все изменения, которые были зафиксированы (commited) в одной ветке и применить их к другой ветке.

В данном примере для этого необходимо выполнить следующее:

```
$ git checkout experiment
$ git rebase master
```

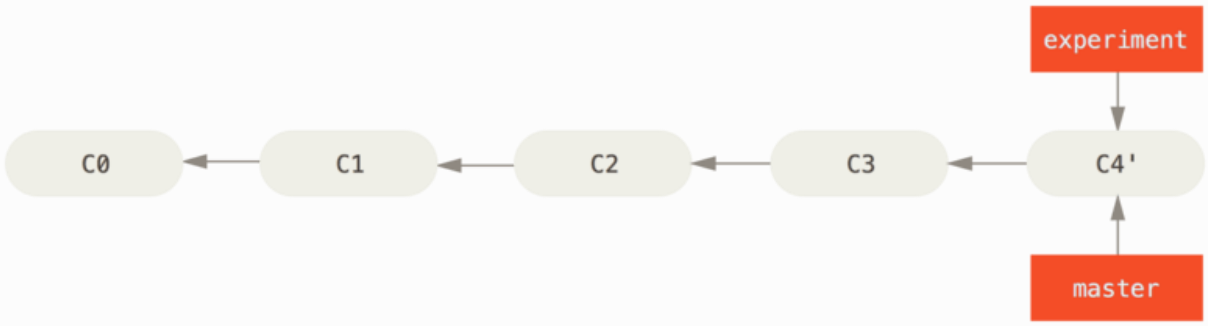
Это работает следующим образом: берется общий родительский снимок (snapshot) двух веток (той, в которой вы находитесь, и той, поверх которой вы выполняете перебазирование); берется дельта (diff) каждого коммита той ветки, на который вы находитесь, эти дельты сохраняются во временные файлы; текущая ветка устанавливается на тот же коммит, что и ветка, поверх которой вы выполняете перебазирование; и, наконец, ранее сохраненные дельты применяются по очереди.



К оглавлению

На этом моменте вы можете переключиться обратно на ветку master и выполнить слияние перемоткой.

```
$ git checkout master
$ git merge experiment
```



Теперь снимок (snapshot), на который указывает C4' абсолютно такой же, как тот, на который указывал C5 в примере с трехсторонним слиянием. Нет абсолютно никакой разницы в конечном результате между двумя показанными примерами, но перебазирование делает историю коммитов чище**. Если вы взглянете на историю перебазированной ветки, то увидите, что она выглядит абсолютно линейной**: будто все операции были выполнены последовательно, даже если изначально они совершались параллельно.

Перебазирование повторяет изменения из одной ветки поверх другой в порядке, в котором эти изменения были представлены, в то время как слияние берет две конечные точки и сливает их вместе.

Плюсы:

- Упрощает потенциально сложную историю
- Упрощение манипуляций с единственным коммитом
- Избежание слияния коммитов в занятых репозиториях и ветках
- Очищает промежуточные коммиты, делая их одним коммитом, что полезно для DevOps команд

Минусы:

- Сжатие фич до нескольких коммитов может скрыть контекст
- Перемещение публичных репозиториев может быть опасным при работе в команде
- Появляется больше работы

К оглавлению

- Для восстановления с удаленными ветками требуется принудительный пуш. Это приводит к обновлению всех веток, имеющих одно и то же имя, как локально, так и удаленно, и это ужасно.

25. Git: работа с удаленным репозиторием (git push)

Когда вы хотите поделиться своими наработками, вам необходимо отправить (push) их в главный репозиторий. Команда для этого действия простая: **$ git push [remote-name] [branch-name]**. Чтобы отправить вашу ветку master на сервер origin (повторимся, что клонирование, как правило, настраивает оба этих имени автоматически), вы можете выполнить следующую команду для отправки наработок на сервер:

$ git push origin master

Эта команда срабатывает только в случае, если вы клонировали с сервера, на котором у вас есть права на запись, и если никто другой с тех пор не выполнял команду push. Если вы и кто-то ещё одновременно клонируете, затем *он выполняет команду push, а затем команду push выполняете вы, то ваш push точно будет отклонё*н. *Вам придётся сначала вытянуть (pull) их изменения и объединить с вашими. Только после этого вам будет позволено выполнить push*

26. Git: модели ветвления

Почти каждая СКВ имеет в какой-то форме поддержку ветвления. Ветвление означает, что вы отклоняетесь от основной линии разработки и продолжаете работу, не вмешиваясь в основную линию. Во многих СКВ это в некотором роде дорогостоящий процесс, зачастую требующий от вас создания новой копии каталога с исходным кодом, что может занять продолжительное время для больших проектов.

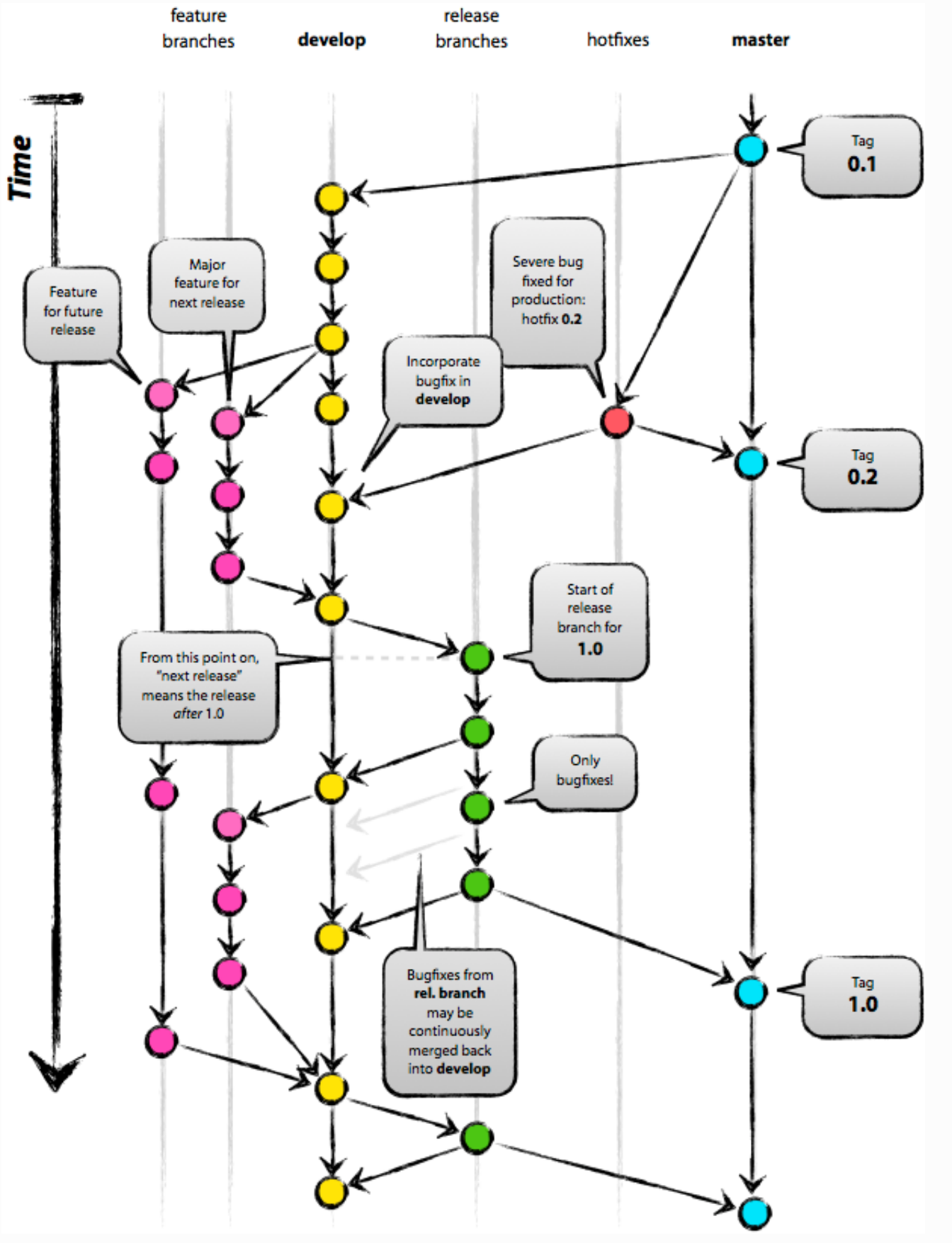
Некоторые говорят, что модель ветвления Git'a это его “killer feature“ и она безусловно выделяет Git в СКВ-сообществе. Что же в ней такого особенного? Способ ветвления в Git'е чрезвычайно легковесен, что делает операции ветвления практически мгновенными и переключение туда-сюда между ветками обычно так же быстрым. В отличие от многих других СКВ, Git поощряет процесс работы, при котором ветвление и слияние осуществляется часто, даже по несколько раз в день. Понимание и владение этой функциональностью даёт вам уникальный мощный инструмент и может буквально изменить то, как вы ведёте разработку.

Простая модель ветвления git (a simple git branching model)

New Фича→создал ветку→сделал фичу→слил в главВетку (Изи-изи, рил ток, синк эбаут ит)

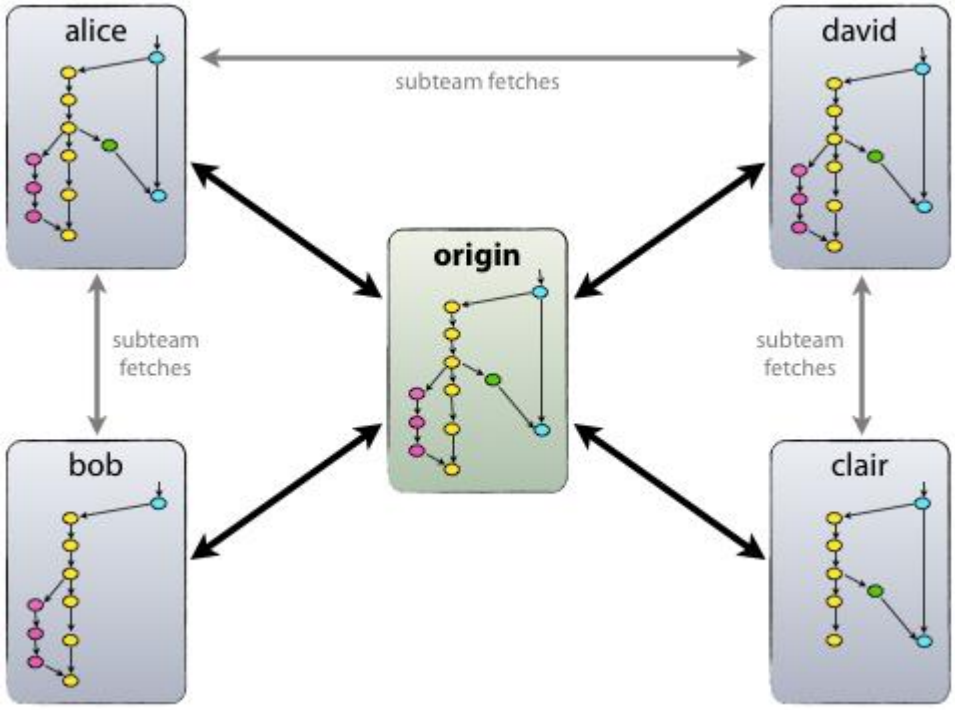
Статья

Удачная модель ветвления для Git(A successful Git branching model)



К оглавлению

Предлагаемая модель ветвления опирается на конфигурацию проекта, содержащую один центральный «истинный» репозиторий.



Каждый разработчик забирает и публикует изменения (pull & push) в origin. Но, помимо централизованных отношений push-pull, каждый разработчик также может забирать изменения от остальных коллег внутри своей микро-команды. Например, этот способ может быть удобен в ситуации, когда двое или более разработчиков работают вместе над большой новой фичей, но не могут издать незавершённую работу в origin раньше времени.

Технически это реализуется несложно: Алиса создаёт удалённую ветку Git под названием bob, которая указывает на репозиторий Боба, а Боб делает то же самое с её репозиторием.

Главные ветки

Представленная модель в значительной степени вдохновлена уже существующими моделями. Центральное хранилище предоставляет две ветки с бесконечным жизненным циклом:

- **master**
- **develop**

Мы считаем ветку **origin/master главной**. То есть, **исходный код в ней должен находиться в состоянии production-ready в любой произвольный момент времени**.

К оглавлению

Ветвь **origin/develop** мы **считаем главной ветвью для разработки**. Хранящийся в ней код в любой момент времени **отражает состояние исходного кода**, на момент последних изменений, **подготовленных к следующему релизу**. Некоторые называют эту ветку integration branch. Например, HEAD этой ветки используется для ночных сборок. Она служит источником для сборки автоматических ночных билдов.

Когда исходный код в ветке develop достигает стабильности, и содержит уровень функциональности близкий к желаемому, все изменения из develop должны быть внесены обратно в master, и отмечены тегом соответствующего релиза.

Следовательно, каждый раз, когда изменения вливаются в главную ветвь (master), мы по определению получаем новый релиз.

Вспомогательные ветви

Помимо главных ветвей master и develop, наша модель разработки содержит некоторое количество типов вспомогательных ветвей, которые используются для распараллеливания разработки между членами команды, для упрощения внедрения нового функционала (features), для подготовки релизов и для быстрого исправления проблем в производственной версии приложения.

В отличие от главный ветвей, эти ветви **всегда имеют ограниченный срок жизни**. Каждая из них в конечном итоге рано или поздно удаляется.

Мы используем следующие типы ветвей:

- Ветви функциональностей (Feature branches)
- Ветви релизов (Release branches)
- Ветви исправлений (Hotfix branches)

У каждого типа ветвей есть своё специфическое назначение и **строгий набор правил, от каких ветвей они могут порождаться, и в какие должны вливаться.**

Ветви функциональностей (feature branches)

Могут порождаться от: develop
Должны вливаться в: develop
Соглашение о наименовании: всё, за исключением master, develop, release-* или hotfix-*

Ветви функциональностей (feature branches), также называемые иногда тематическими ветвями (topic branches), используются для разработки новых функций, которые должны появиться в текущем или будущем релизах. При начале работы над функциональностью

К оглавлению

(фичей) может быть ещё неизвестно, в какой именно релиз она будет добавлена. Смысл существования ветви функциональности (feature branch) состоит в том, что она живёт так долго, сколько продолжается разработка данной функциональности (фичи). Когда работа в ветви завершена, последняя вливается обратно в главную ветвь разработки (что означает, что функциональность будет добавлена в грядущий релиз) или же удаляется (в случае неудачного эксперимента).

Ветви функциональностей (feature branches) обычно существуют в репозиториях разработчиков, но не в главном репозитории (origin).

Создание ветви функциональности (feature branch)

При начале работы над новой функциональностью делается ответвление от ветви разработки (develop).

Добавление завершённой функциональности в develop

Завершённая функциональность (фича) вливается обратно в ветвь разработки (develop) и попадает в следующий релиз.

Ветви релизов (release branches)

Могут порождаться от: develop
Должны вливаться в: develop и master
Соглашение о наименовании: release-*

Ветви релизов (release branches) используются для подготовки к выпуску новых версий продукта. Они позволяют расставить финальные точки над i перед выпуском новой версии. Кроме того, они позволяют вносить исправления мелких ошибок, и подготовить метаданные для релиза (номер версии, дата создания и т.д.). **Делая все эти действия в ветках release, ветка develop останется чиста, и будет готова к добавлению нового кода, отвечающего непосредственно за расширение функционала разрабатываемого продукта.**

Новую ветку релиза (release branch) надо порождать в тот момент, когда состояние ветви разработки полностью или почти полностью соответствует требованиям, соответствующим новому релизу. По крайней мере, вся необходимая функциональность, предназначенная к этому релизу, уже влита в ветвь разработки (develop). Функциональность, предназначенная к следующим релизам, может быть и не влита. Даже лучше, если ветки для этих функциональностей подождут, пока текущая ветвь релиза не отпочкуется от ветви разработки (develop).

Очередной релиз получает свой номер версии только в тот момент, когда для него создаётся новая ветвь, но ни в коем случае не раньше. Вплоть до этого момента ветвь разработки содержит изменения для «нового релиза», но пока ветка релиза не отделилась, точно неизвестно, будет ли этот релиз иметь версию 0.3, или 1.0, или какую-то другую. Решение принимается при создании новой ветви релиза и зависит от принятых на проекте правил нумерации версий проекта.

Создание ветви релиза (release branch)

Ветвь релиза создаётся из ветви разработки (develop). Пускай, например, текущий изданный релиз имеет версию 1.1.5, а на подходе новый большой релиз, полный изменений. Ветвь разработки (develop) готова к «следующему релизу», и мы решаем, что этот релиз будет иметь версию 1.2 (а не 1.1.6 или 2.0). В таком случае мы создаём новую ветвь и даём ей имя, соответствующее новой версии проекта

Мы создали новую ветку, переключились в неё, а затем выставили номер версии (bump version number). В нашем примере bump-version.sh — это вымышленный скрипт, который изменяет некоторые файлы в рабочей копии, записывая в них новую версию. (Разумеется, эти изменения можно внести и вручную; я просто обращаю Ваше внимание на то, что некоторые файлы изменяются.) Затем мы делаем коммит с указанием новой версии проекта.

Эта новая ветвь может существовать ещё некоторое время, до тех пор, пока новый релиз окончательно не будет готов к выпуску. В течение этого времени к этой ветви (а не к develop) могут быть добавлены исправления найденных багов. Но добавление крупных новых изменений в эту ветвь строго запрещено. Они всегда должны вливаться в ветвь разработки (develop) и ждать следующего большого релиза.

Закрытие ветви релиза

Когда мы решаем, что ветвь релиза (release branch) окончательно готова для выпуска, нужно проделать несколько действий. В первую очередь ветвь релиза вливается в главную ветвь (напоминаю, каждый коммит в master — это по определению новый релиз). Далее, этот коммит в master должен быть помечен тегом, чтобы в дальнейшем можно было легко обратиться к любой существовавшей версии продукта. И наконец, изменения, сделанные в ветви релиза (release branch), должны быть добавлены обратно в разработку (ветвь develop), чтобы будущие релизы также содержали внесённые исправления багов.

Ветви исправлений (hotfix branches)

Могут порождаться от: master

К оглавлению

Должны вливаться в: develop и master
Соглашение о наименовании: hotfix-*

Ветви для исправлений (hotfix branches) весьма похожи на ветви релизов (release branches), так как они тоже используются для подготовки новых выпусков продукта, разве лишь незапланированных.

Они порождаются необходимостью немедленно исправить нежелательное поведение производственной версии продукта. Когда в производственной версии находится баг, требующий немедленного исправления, из соответствующего данной версии тега главной ветви (master) порождается новая ветвь для работы над исправлением.

Смысл её существования состоит в том, что работа команды над ветвью разработки (develop) может спокойно продолжаться, в то время как кто-то один готовит быстрое исправление производственной версии.

Создание ветви исправлений (hotfix branch)

Ветви исправлений (hotfix branches) создаются из главной (master) ветви. Пускай, например, текущий производственный релиз имеет версию 1.2, и в нём (внезапно!) обнаруживается серьёзный баг. А изменения в ветви разработки (develop) ещё недостаточно стабильны, чтобы их издавать в новый релиз. Но мы можем создать новую ветвь исправлений и начать работать над решением проблемы:

Не забывайте обновлять номер версии после создания ветви

Теперь можно исправлять баг, а изменения издавать хоть одним коммитом, хоть несколькими.

Закрытие ветви исправлений

Когда баг исправлен, изменения надо влить обратно в главную ветвь (master), а также в ветвь разработки (develop), чтобы гарантировать, что это исправление окажется и в следующем релизе. Это очень похоже на то, как закрывается ветвь релиза (release branch).

Прежде всего надо обновить главную ветвь (master) и пометить новую версию тегом.

Следующим шагом переносим исправление в ветвь разработки (develop).

У этого правила есть одно исключение: если в данный момент существует ветвь релиза (release branch), то ветвь исправления (hotfix branch) должна вливаться в неё, а не в ветвь разработки (develop). В этом случае исправления войдут в ветвь разработки

К оглавлению

вместе со всей ветвью релиза, когда та будет закрыта. (Хотя, если работа в develop требует немедленного исправления бага и не может ждать, пока будет завершено издание текущего релиза, Вы всё же можете влить исправления (bugfix) в ветвь разработки (develop), и это будет вполне безопасно).

Подведем итоги

Автор объединил существующие подходы, очевидные вещи в единую логичную системы и подвел черту. Данная модель не является серебрянной пулей, но она проста, наглядна и легка в применении. Данная модель помогает развить понимание ветвления и процесса выпуска релизов.

27. Системы автоматизации сборки: цели, задачи, примеры

Автоматизация сборки — этап процесса разработки программного обеспечения, заключающийся в автоматизации широкого спектра задач, решаемых программистами в их повседневной деятельности.

Включает в себя такие действия, как:

- компиляция исходного кода в Объектный модуль (файл с промежуточным представлением отдельного модуля программы, полученный в результате обработки исходного кода компилятором)
- сборка бинарного кода в исполняемый файл
- выполнение тестов
- разворачивание программы на производственной платформе
- написание сопроводительной документации или описание изменений новой версии

Преимущества

- Ускорение процесса компиляции и линковки
- Избавление от излишних действий
- Минимизация «плохих (некорректных) сборок»
- Избавление от привязки к конкретному человеку
- Ведение истории сборок и релизов для разбора выпусков
- Нивелируются человеческие ошибки при сборе системы.
- Упрощается перенос системы в новую тестовую среду.
- Растет качество продукта, потому что не нужно ждать окончания разработки, чтобы убрать баги и внести изменения в функционал.
- Экономия времени и денег благодаря причинам, указанным выше

К оглавлению

Требования к системам сборки

Базовые требования:

- Частые или ночные сборки для своевременного выявления проблем.
- Поддержка управления зависимостями исходного кода (Source Code Dependency Management)
- Обработка разностной сборки
- Уведомление при совпадении исходного кода (после сборки) с имеющимися бинарными файлами.
- Ускорение сборки
- Отчет о результатах компиляции и линковки.

Дополнительные требования:

- Создание описания изменений (release notes) и другой сопутствующей документации (например, руководства).
- Отчет о статусе сборки
- Отчет об успешном/неуспешном прохождении тестов.
- Суммирование добавленных/измененных/удаленных особенностей в каждой новой сборке

Типы

- Автоматизация по запросу (On-Demandautomation): запуск пользователем скрипта в командной строке.
- Запланированная автоматизация (Scheduledautomation): непрерывная интеграция, происходящая в виде ночных сборок.
- Условная автоматизация (Triggeredautomation): непрерывная интеграция, выполняющая сборку при каждом подтверждении изменения кода (commit) в системе управления версиями.

Таким образом, можно сказать следующее: автоматическая сборка позволяет снять огромный пласт работ с разработчиков и тестеров системы на себя, причем выполняться все это будет без участия пользователя в фоновом режиме. Можно сдать свои исправления в центральное хранилище и знать, что через какое-то время будет готово все для передачи версии пользователю или на окончательное тестирование и не беспокоиться обо всех промежуточных шагах – или же, по крайней мере, узнать о том, что правки были некорректными, как можно скорее.

К оглавлению

Makefile- одна из форм автоматизации сборки

Построение Makefile'а:

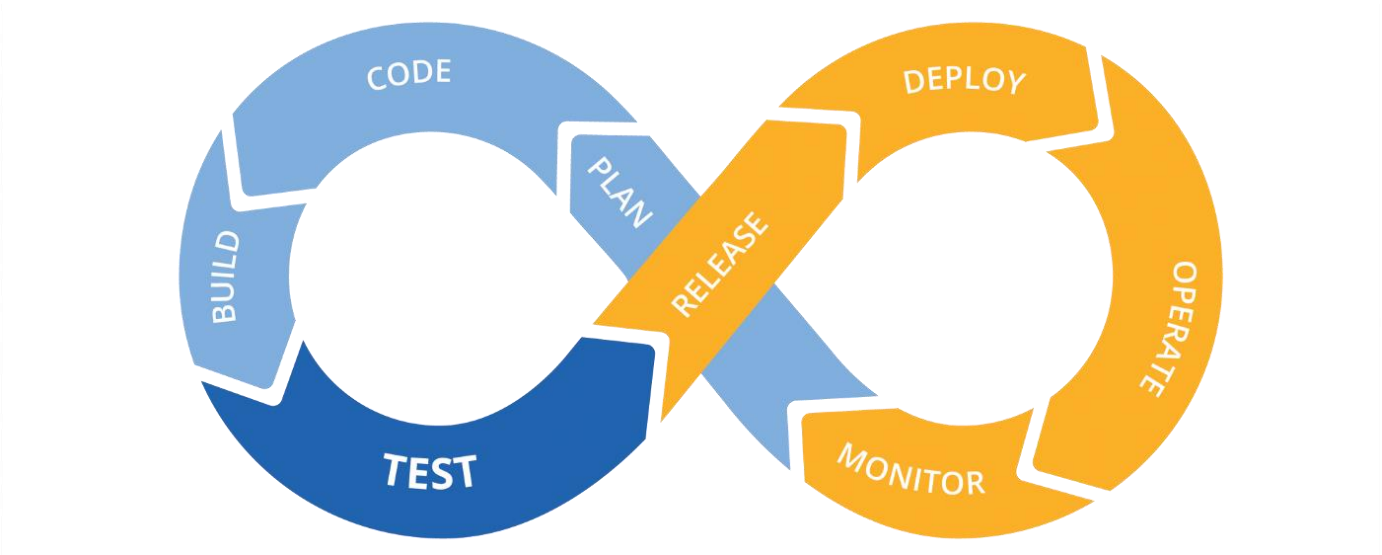
цель: зависимости
[tab] команды

Пример фиктивных целей
all — является стандартной целью по умолчанию. При вызове make ее можно явно не указывать.
clean — очистить каталог от всех файлов полученных в результате компиляции.
install — произвести инсталляцию
uninstall — и деинсталляцию соответственно.
Чтобы make не искал файлы с такими именами, их следует определить припомощи директивы .PHONY.
.PHONY: all clean install uninstall
Пример makefile'а:

```
COMPILER = g++
FLAGS = -MMD -c -Wall -Werror -std=c++14 -o

.PHONY: clean run all

all: create bin/chessviz bin/chessviz-test

-include build/src/*.d

bin/chessviz: build/src/main.o build/src/checks.o build/src/board_read.o build/src/board.o build/src/board_print_html.o build/src/board_print_plain.o
    $(COMPILER) $^ -o $@

build/src/main.o: src/main.cpp
    $(COMPILER) $(FLAGS) $@ $<

build/src/board.o: src/board.cpp
    $(COMPILER) $(FLAGS) $@  $<

build/src/checks.o: src/checks.cpp
    $(COMPILER) $(FLAGS) $@  $<

build/src/board_read.o: src/board_read.cpp
    $(COMPILER) $(FLAGS) $@  $<

build/src/board_print_html.o: src/board_print_html.cpp
    $(COMPILER) $(FLAGS) $@  $<

build/src/board_print_plain.o: src/board_print_plain.cpp
    $(COMPILER) $(FLAGS) $@  $<
```

28. Непрерывная интеграция: цели, основные принципы, инструментарий

К оглавлению



"Что такое Continuous Integration?"

Посмотрим что нам на этот вопрос скажет Wiki:

Непрерывная интеграция (CI, англ. Continuous Integration) — это **практика разработки программного обеспечения**, которая **заключается в слиянии рабочих копий в общую основную ветвь разработки** (до нескольких раз в день**) и выполнении частых автоматизированных сборок проекта для скорейшего выявления потенциальных дефектов и решения интеграционных проблем.**

Страшно, не правда ли? Давайте попробуем объяснить этот термин простыми словами:

Непрерывная интеграция - это система для сборки и автоматизированного тестирования программного обеспечения с определенными конфигами на определенных машинах с целью обнаружения багов и несовместимостей.

Впервые концептуализирована и предложена Гради Бучем в 1991 году

Непрерывная интеграция (Continuous Integration, CI) — это набор практик и принципов разработки программного обеспечения, в которой члены команды проводят интеграцию как можно чаще. Каждая такая операция должна приводить к ряду автоматических проверок, позволяющих еще на ранних этапах разработки обнаруживать проблемы. Непрерывную интеграцию в основном используют в проектах с большим числом разработчиков или несколькими командами разработки, потому что она позволяет снизить трудоемкость процесса и сделать его более предсказуемым за счет наиболее раннего обнаружения и устранения ошибок и противоречий.

Зачем нам CI?

К оглавлению

При каждом push на какой-либо ресурс, CI будет собирать ваш проект с нуля, запускать ВСЕ тесты, и только если все тесты пройдут и проект соберется, сборка получит статус passing(проходящий). В противном случае у вас будет возможность сделать comeback(возврат), и посмотреть, что пошло не так.

Семь составляющих непрерывной интеграции

Continuous Integration порой относят к agile-практикам. Но если в Agile во главу угла ставят изменение мышления и практики (за которыми идут организационные изменения), **то CI — это в первую очередь автоматизация процессов, которая позволяет избежать многократного выполнения рутинных операций**. Проще говоря, непрерывная интеграция — это **практика частых сборок кода** в сочетании с его **инспекцией** и постоянным **тестированием**. И поскольку вручную все это делать сложно, процесс максимально автоматизирован.

Единый репозиторий исходного кода

Репозиторий необходим, чтобы снизить риски потери контроля над развитием кода или даже потери самого кода. Когда над проектом трудятся несколько разработчиков, и каждый из них хранит код по-своему, велика вероятность, что какой-то фрагмент будет потерян. Нельзя исключать конфликтов при интеграции кода разными разработчиками — тогда понадобится организовать резервное копирование. Кроме того, единый для проекта репозиторий позволяет удобно организовать в проекте процесс Code Review (рецензирование кода) и в случае ошибки позволяют откатиться обратно, до работающей версии.

Преимущества

- Вся команда получает единую входную точку.
- Проще контролировать версии.
- Есть возможность получить список отличий между двумя версиями.
- К коду получает доступ вся команда.
- Проще посчитать вклад каждого.
- Можно посмотреть историю изменений.
- Код можно разбивать на ветки.

Автоматизированный процесс сборки

Преобразование исходного кода в исполняемые файлы (они же – «артефакты») не всегда проходит просто. И чтобы превратить исходники в работающую систему, нужно компилировать, постоянно перемещать файлы и загружать схемы в базы данных.

Преимущества

- Нивелируются человеческие ошибки при сборе системы.
- Растет взаимозаменяемость — любой разработчик или тестировщик может развернуть систему.
- Не нужно заниматься ручным развертыванием системы, больше времени для разработки.

К оглавлению

- Упрощается перенос системы в новую тестовую среду.
- Растет качество продукта, потому что не нужно ждать окончания разработки, чтобы убрать баги и внести изменения в функционал.

Автоматизация тестирования сборки

Традиционно сборкой решения считают компиляцию проекта и получение исполняемой части программы. Конечно, после сборки программа может исполняться, но это не означает, что она будет работать правильно. Один из способов отловить дефекты быстро (а чем раньше ошибка найдена, тем дешевле ее исправление) и эффективно — автоматизация тестирования.

Преимущества

- Автотесты могут выполняться 24/7, они значительно дешевле.
- Скорость намного выше ручного тестирования.
- Они надежнее, потому что нельзя забыть или намеренно пропустить какой-то тест.
- Снижаются человеческие и технические риски, нет угрозы попасть в зависимость от команды разработки и тестирования.
- Автотест позволяет обойти ограничения пользовательского интерфейса.

Запуск на интеграционном сервере

Если изменения в основную ветку разработки добавляют ежедневно, команда чаще получает протестированные сборки. Но в любой момент все может пойти наперекосяк.

Во-первых, разработчики не собирают решение каждый раз при вливании изменений. Во-вторых, среда разработчика может отличаться от среды тестирования или продуктивной среды.

Поэтому необходимо, чтобы после каждого изменения решение собиралось в некой эталонной среде. Такой средой выступает сервер непрерывной интеграции, который мониторит все изменения на сервере исходного кода и, как только их обнаруживает, приступает к сборке проекта. И только после того как все успешно собрано, изменения становятся доступны каждому.

Преимущества

- Проще интегрировать различные части проекта.
- Снижается риск, связанный с конфигурацией программного и аппаратного обеспечения у членов команды

Тестирование в клоне рабочей среды

Основная идея в том, чтобы проводить тестирование и развертывание продукта в среде, аналогичной продуктивной, чтобы не тратить время на выяснение конфигурационных проблем. Эти проблемы можно будет найти уже во время тестирования. Есть и ограничения: если разрабатывать десктопное приложение, не будет финансовой

К оглавлению

возможности повторить все многообразие конфигураций. Но все, что касается серверной части, решается в рамках этой практики. К тому же можно сократить затраты, построив виртуальную многомашинную среду для тестирования и имитации различных окружений.

Одна из самых малораспространенных практик CI, потому что не каждый заказчик готов описать конфигурацию его серверов, есть только абстрактные схемы. К тому же этот инструмент сложно применять в банкинге, потому что там, чтобы создать клон рабочей среды, нужны настоящие данные пользователей банка.

Преимущества

- Позволяет предупредить ошибки продукта, которые могут возникнуть в связи с особенностями окружения.

Доступность сборки для всей команды

Каждый раз в процессе разработки, когда нужно убедиться в работоспособности ПО, необходимо запустить приложение и проверить его. При этом важно, чтобы каждый член команды в любой момент мог посмотреть на это программное обеспечение и проверить его. Поэтому должно быть единое, удобное хранилище со стабильными сборками. А все участники проекта должны знать, как попасть в это хранилище и как с ним работать.

Преимущества

- Обеспечивает стабильность и высокую надежность системы.
- Минимизирует затраты на управление и хранение исполняемых файлов в разных средах — файловая система хранит только одну копию любого двоичного файла.
- Контроль доступа и история изменений помогают отследить, кто и какие изменения вносил в систему.
- Доступ к артефактам есть у всех участников процесса разработки.
- Обеспечивает расширенное управление каждым этапом: загрузкой, развертыванием, перемещением, копированием.

Автоматизированное развертывание целевого решения

В процессе разработки и использования программного обеспечения требуется несколько сред: тестовая, разработчиков, продуктивная. И вместо того чтобы тратить время на ручное развертывание при обновлении, есть смысл автоматизировать этот процесс.

Преимущества

- Автоматизированная сборка более устойчива к появлению ошибок и проще в тиражировании.
- Собрать ее может любой человек из команды.
- Разработчики больше времени занимаются непосредственно разработкой.
- Установить сборку на новый стенд несложно, достаточно изменить имя машины.
- Появляется возможность выпускаться чаще.

К оглавлению

Инструментарий

- Локальные
  - GitLab CI
  - TeamCity
  - Bamboo
  - GoCD Jenkins
  - Circle CI.
- Облачные
  - BitBucket Pipelines
  - Heroku CI
  - Travis
  - Codeship
  - Buddy CI
  - AWS CodeBuild.

К оглавлению