Экстремальное программирование

XP

eXtreme Programming

Экстремальное программирование – это упрощенная методика организации производства для небольших и средних по размеру команд специалистов, занимающихся разработкой программного продукта в условиях неясных и быстро меняющихся требований.

XP

Признаки применения XP:

- Короткие циклы;
- Планирование по нарастающей;
- Гибкий график реализации функциональности;
- XP базируется на автоматических тестах, разработанных и программистами, и заказчиками;
- Обмен сведениями через общение, тесты и исходный код;
- Эволюционирующий дизайн.

XP

Основная проблема разработки ПО - РИСК

Виды рисков:

- ✔ Смещение графиков;
- ✔ Закрытие проекта;
- ✔ Система теряет полезность;
- ✔ Велико количество дефектов и недочетов системы;
- ✔ Несоответствие системы решаемой проблеме;
- ✔ Изменение характера бизнеса;
- ✔ Недостаток возможностей системы;
- ✔ Текучка кадров.

XP

Модель контроля переменных

4 переменные:

- ✔ Затраты;
- ✔ Время;
- ✔ Качество;
- ✔ Объем работ.

Внешние силы (заказчики, менеджеры) должны определить значения для любых трех переменных, а команда разработчиков выбирает результирующее значение для четвертой переменной (управляет четвертой переменной).

XP

Стоимость внесения изменений в систему

Обычная стратегия разработки ПО предусматривает стадии:

- ✔ Формулировка требований;
- ✔ Анализ требований;
- ✔ Проектирование системы;
- ✔ Реализация системы;
- ✔ Тестирование системы;
- ✔ Внедрение системы.

Стоимость внесения изменений растет экспоненциально в зависимости от времени.

XP

Стоимость внесения изменений в систему

Основное предположение XP:

Сегодня требуется реализовать только то, без чего сегодня не обойтись.

Стоимость внесения изменений в систему растет пропорционально $\sqrt{t}$, где t – время работы над системой.

XP

Три ключевых фактора XP:

- Простой дизайн без лишних элементов;
- Автоматические тесты;
- Постоянная практика в деле модификации дизайна системы.

XP

Четыре ценности XP:

- Коммуникация
- Простота
- Обратная связь
- Храбрость

Четыре основных вида деятельности XP:

- Кодирование;
- Тестирование;
- Общение;
- Проектирование.

XP

Базовые принципы XP:

- ✔ Быстрая обратная связь;
- ✔ Приемлемая простота;
- ✔ Постепенное изменение;
- ✔ Приемлемые изменения;
- ✔ Качественная работа.

Менее важные принципы:

- ✔ Обучение обучению;
- ✔ небольшие начальные инвестиции;
- ✔ надежное экспериментирование;
- ✔ открытая честная коммуникация;
- ✔ работа в соответствии с человеческими инстинктами;
- ✔ принимаемая ответственность;
- ✔ локальная адаптация;

XP

Ограничения применения XP:

- Бизнес-культура;
- Обычный стиль работы разработчиков, настроенный на тщательное планирование;
- Крупномасштабные проекты, требующие большой команды программистов;
- Рабочая среда, препятствующая легкости обратной связи.

Практики XP

1. Игра в планирование
2. Тестирование
3. Парное программирование
4. Рефакторинг
5. Простой дизайн
6. Коллективное владение кодом
7. Постоянная интеграция
8. Заказчик на месте с разработчиками
9. Частые выпуски версий
10. 40-часовая рабочая неделя
11. Стандарты кодирования
12. Метафора системы

Игра в планирование (Planning Game)

- Задачи команды
  1. Оценка времени для реализаций каждого из пожеланий;
  2. Оценка времени связанной с выбором технологии
  3. Распределение задачи между командами;
  4. Оценка рисков, связанных с каждым из пожеланий;
  5. Определение порядка в котором будет реализованы задачи.
- Задачи заказчика
  1. Определяет набор пожеланий и конкретизирует каждую итерацию;
  2. Определяет дату завершения работы(Realize)
  3. Определяет приоритеты задач

Планирование должно выполняться как можно чаше

Первичное обследование

- В начале работы над проектом разработчики и заказчики обсуждают новую систему, чтобы выявить наиболее существенные функции.
- Однако они не пытаются идентифицировать *все вообще* функции. По мере развертывания работ заказчики будут обнаруживать все новые и новые функции. Поток функций не иссякнет вплоть до момента завершения проекта. Вновь выявленная функция разбивается на одну или несколько *пользовательских историй*, которые записываются на учетной карточке или чем-то подобном.
- Пример, «Вход в систему», «Добавление пользователя», «Удаление пользователя», «Изменение пароля»
- Разработчики совместно оценивают истории в баллах. Эти оценки относительны, а не абсолютны.

Объединение, разбиение и скорость

- Если история слишком велика, то ее следует разбить на меньшие части.
- Если история слишком мала, ее следует объединить с другими
- История «Пользователи могут безопасно переводить деньги на свой счет, со своего счета или с одного счета на другой» может быть разбита на истории:
- · Пользователь может войти в систему
- · Пользователь может выйти из системы
- · Пользователь может положить деньги на свой счет
- · Пользователь может снять деньги со своего счета
- · Пользователь может перевести деньги с любого из своих счетов на другой

Планирование выпуска

- Каждую неделю завершатся реализация нескольких историй. Сумма оценок завершенных историй дает *скорость*.
- Зная скорость, заказчик может получить представление о стоимости каждой истории, а также о ее значимости для бизнеса и о приоритете. Это позволяет заказчику решить, какие истории реализовывать первыми.
- Разработчики и заказчик согласуют дату первого выпуска проекта. Обычно это занимает 2–4 месяца. Заказчик указывает, какие истории реализовать в этом выпуске и примерный порядок реализации. Не разрешается выбирать больше историй, чем укладывается в текущую скорость.

Планирование итерации

- Разработчики и заказчик определяют длительность итерации: обычно 1 или 2 недели. И снова заказчик решает, какие истории реализовать на первой итерации, но не может выбрать больше историй, чем позволяет текущая скорость.
- Порядок реализации историй в пределах итерации – вопрос решаемый самими разработчиками.
- Заказчик не может изменять истории после начала итерации. Он вправе изменять или переупорядочивать любые истории, кроме тех, над которыми разработчики уже трудятся.
- Итерация заканчивается в заранее оговоренный день, даже если не все истории реализованы. Оценки всех завершенных историй суммируются, и вычисляется скорость на данной итерации. Эта величина ис-пользуется для планирования следующей итерации.

Определения понятия ≪готово≫

- История не считается реализованной, пока не пройдут *все* приемочные тесты. Эти тесты автоматизированы.
- Пишут их заказчики, бизнес аналитики, специалисты по контролю качества, тестировщики и даже программисты в начале каждой итерации.
- Тесты определяют детали истории и являются неоспоримым авторитетом в вопросе о поведении историй.

Планирование задач (1)

- В начале новой итерации разработчики и заказчики вырабатывают план. Разработчики разбивают истории на задачи.
- *Задачей* называется объем работы, с которым один разработчик может справиться за 4–16 часов.
- Истории анализируются совместно с заказчиком, и задачи формулируются максимально полно.
- Список задач записывается в лекционном блокноте, на доске или на любом другом носителе.
- Затем разработчики выбирают себе задачи, которые хотели бы реализовать, присваивая каждой задаче произвольную оценку в баллах.

Планирование задач (2)

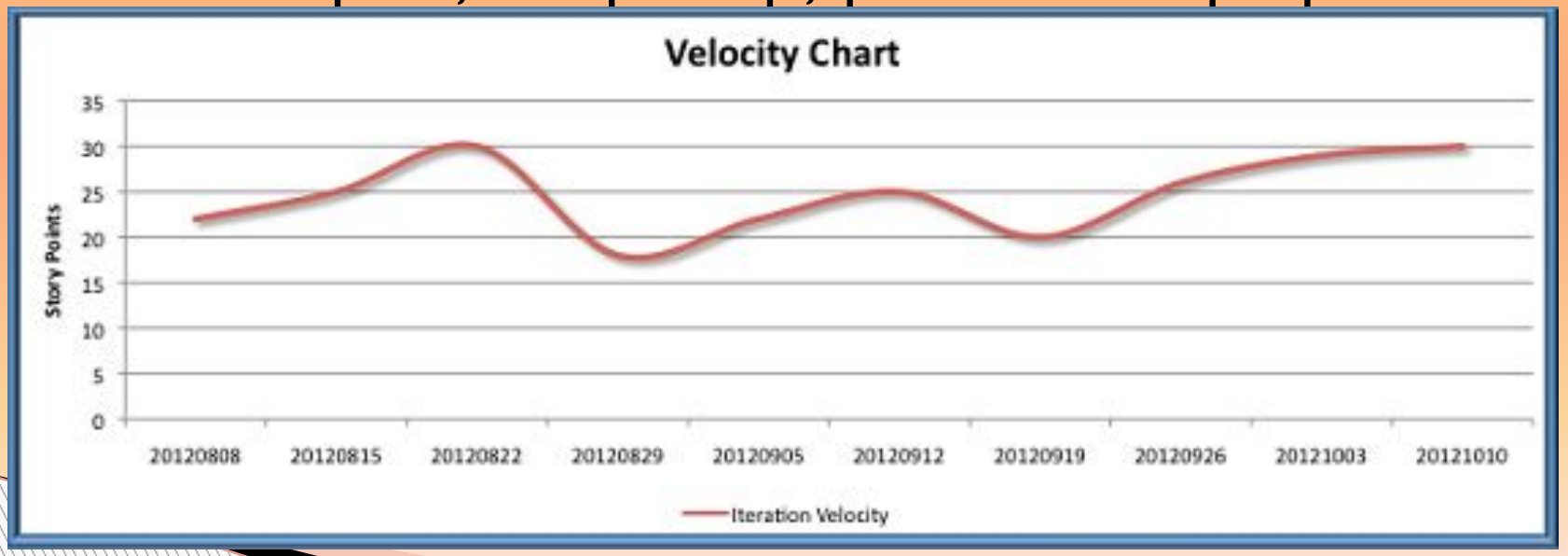
- Разработчик может выбрать любую задачу вне зависимости от специализации.
- Каждый разработчик знает, сколько баллов он заработал на задачах на предыдущей итерации; это число составляет *бюджет* разработчика.
- Никто не берет себе задач на большую сумму, чем его бюджет.
- В середине итерации команда устраивает рабочее совещание. В этот момент должна быть реализована половина *историй*, запланированных на данную итерацию. Если это не так, то команда пытается перераспределить задачи и ответственность таким образом, чтобы к концу итерации все истории были реализованы.

Подведение итогов итераций

- Раз в две недели текущая итерация заканчивается и начинается новая.
- В конце каждой итерации заказчику демонстрируется текущая версия системы.
- Заказчика просят оценить внешний вид и функциональность проекта. Заказчик выражает свое мнение в виде набора новых пользовательских историй.
- Заказчика часто информируют о ходе работы над проектом.
- Он может измерить скорость.
- Он может оценить, как быстро продвигается команда, и запланировать высокоприоритетные истории на ранней стадии.

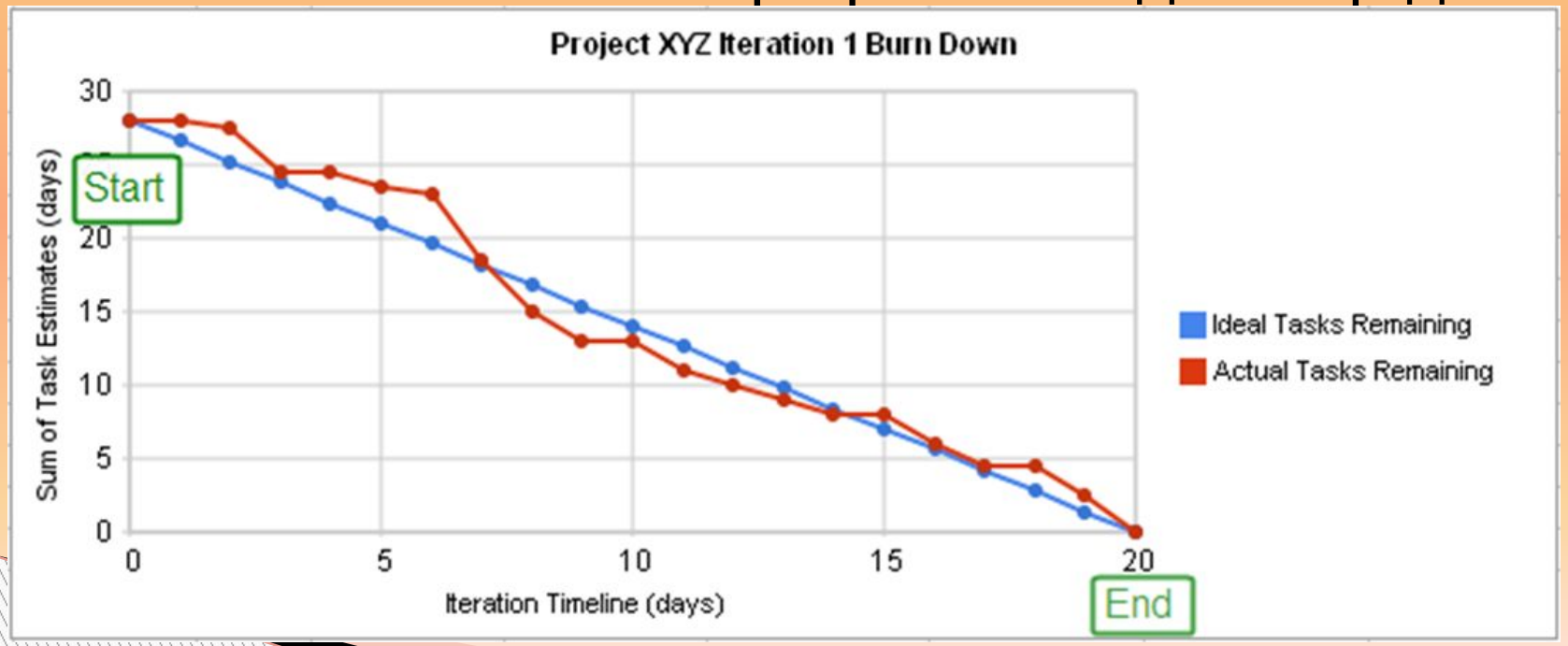
Мониторинг (1)

- Мониторинг и управление XP-проектом сводятся к записи результатов каждой итерации и использованию этих данных для прогнозирования следующих итераций. Рассмотрим, например, рис.. Этот график



Мониторинг (2)

- *Диаграмма выгорания* показывает, сколько баллов оставалось набрать в конце каждой недели до следующей важной контрольной точки. Наклон этого графика – индикатор даты



Тестирование (Testing)

Тестирование применяемое в XP

- Тестирование модулей (unit testing)
- Приемочное тестирование (acceptance testing)

- ***Разработка через тестирование*** *(англ. test-drivendevelopment)* - техника программирования, при которой модульные тесты для программы или её фрагмента пишутся до самой программы (англ. test-firstdevelopment) и, по существу, управляют её разработкой. Является одной из основных практик экстремального программирования.

Тест - это процедура, которая позволяет либо подтвердить, либо опровергнуть работоспособность кода.

Методика разработки через тестирование (Test-DrivenDevelopment, TDD) позволяет получить ответы на вопросы об организации автоматических тестов и выработке определенных навыков тестирования.

«Чистый код, который работает» - в этой короткой, но содержательной фразе, кроется весь смысл методики разработки приложений через тестирование.

Чистый код, который работает, - это цель, к которой стоит стремиться, и этому есть причины:

- Это предсказуемый способ разработки программ. Разработчик знает, когда работу следует считать законченной, и можете не беспокоиться о длинной череде ошибок.
- У разработчика появляется шанс усвоить уроки, которые преподносит ему код. Если он воспользуется первой же идеей, которая пришла ему в голову, у него не будет шанса реализовать вторую, лучшую идею.
- Коллеги по команде могут рассчитывать на разработчика, а он, в, свою очередь, на них.
- Разработчику приятнее писать такой код.

Чтобы избавиться от множества проблем, разрабатывают код, исходя из автоматических тестов. Такой стиль программирования называется разработкой через тестирование.
В рамках этой методики:

- Пишем новый код только тогда, когда автоматический код не сработал.
- Удаляем дублирование.

Два столь простых правила на самом деле генерируют сложное индивидуальное и групповое поведение со множеством технических последствий:

- Проектируя код, мы постоянно запускаем его и получаем представление о том, как он работает, это помогает нам принимать правильные решения.
- Мы самостоятельно пишем свои собственные тесты, так как мы не можем ждать, что кто-то другой напишет тесты для нас.
- Наша среда разработки должна быстро реагировать на небольшие модификации кода.
- Архитектура программы должна базироваться на использовании множества сильно связанных компонентов, которые слабо сцеплены друг с другом, благодаря чему тестирование кода упрощается.

Два упомянутых правила определяют порядок этапов программирования:

- **Красный** - напишите небольшой тест, который не работает, а возможно, даже не компилируется.
- **Зеленый** - заставьте тест работать как можно быстрее, при этом не думайте о правильности дизайна и чистоте кода. Напишите ровно столько кода, чтобы тест сработал.
- **Рефакторинг** - удалите из написанного вами кода любое дублирование.

Разработка через тестирование

1. Не писать код, пока не будет написан автономный тест, который не проходит.
2. Не писать автономный тест большего объема, чем необходимо для неудачного завершения или неудачной компиляции.
3. Не писать больше кода, чем необходимо для того, чтобы ранее не проходивший тест завершился успешно.

- Написание теста до кода способствует проектированию кода, который было бы *удобно вызывать*.
- Предварительное написание тестов *заставляет писать слабо связанные программы*!
- Тесты могут выступать как чрезвычайно полезная документация. Она всегда актуальна. Она не лжет.

Приемочные тесты – 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.

Приемочное тестирование **выполняется на основании набора типичных тестовых случаев и сценариев,** разработанных на основании требований к данному приложению.

Решение о проведении приемочного тестирования принимается, когда:

- продукт достиг необходимого уровня качества;
- заказчик ознакомлен с **Планом Приемочных Работ** (**ProductAcceptancePlan**) или иным документом, где описан набор действий, связанных с проведением приемочного тестирования, дата проведения, ответственные и т.д.

Фаза приемочного тестирования длится до тех пор, пока заказчик не выносит решение об отправлении приложения на доработку или выдаче приложения.

Заказчик описывает подлежащие тестированию сценарии после того, как история пользователя (userstory) была корректно реализована.

История может содержать один или несколько приемочных тестов – столько, сколько необходимо для проверки работоспособности системы.

Традиционно приемочные тесты состоят из ряда тестовых сценариев – последовательности действий, выполняемых тестировщиком (или поданных на вход инструмента тестирования), которые проверяют правильность работы приложения.

Наличие автоматизированных приемочных тестов позволяет сосредоточиться на порученной задаче и поддерживать продуктивность на оптимальном уровне. В них применимы те же принципы разработки, управляемой тестами, но на более высоком концептуальном уровне, чем позволяют автономные тесты.

- **Парное программирование** — техника программирования, при которой исходный код создаётся парами людей, программирующих одну задачу, сидя за одним рабочим местом.

- Один из них работает непосредственно с текстом программы, другой просматривает его работу и следит за общей картиной происходящего. При необходимости клавиатура свободно передается от одного к другому. В течение работы над проектом пары не фиксированы: рекомендуется их перемешивать, чтобы каждый программист в команде имел хорошее представление о всей системе. *Таким образом, парное программирование усиливает взаимодействие внутри команды.*

Программирование парами (Pair Programming)

Преимущества парного программирования

- Любые решения принимаются не одним программистов а двумя;
- Какую бы часть системы не взять, в ней будут хорошо разбираться как минимум два человека;
- В паре партнеры контролируют друг друга – снижение вероятности написания не корректного, не аккуратного кода;
- Партнеры в парах постоянно меняются, это дает возможность всем членам команды знать функционал системы

Недостатки:

- *Отсутствует возможность сосредоточиться.* Непрерывно отвлекают.

Рефакторинг (Refactoring)

Refactoring – процесс изменения программной системы таким образом, что ее внешнее поведение не изменяется, а внутренняя структура улучшается.

- У каждого программного модуля есть три функции:

1) та функция, которую он реализует во время выполнения
2) модуль должен допускать изменение. Почти все модули на протяжении своей жизни изменяются, и задача разработчика – обеспечить возможность и простоту внесения изменения. Модуль, который трудно изменять, следует считать непригодным и нуждающимся в исправлении, даже если он работает.
3) модуль должен быть доступным для понимания. Разработчики, не знакомые с модулем, должны иметь возможность прочитать и понять его, не прилагая сверхъестественных умственных усилий. Модуль, который не информативен, также непригоден и нуждается в исправлении.

Простой дизайн (Simple Design)

Цели Simple Design:

- Спроектировать систему один раз и на всегда невозможно. В XP – проектирование происходит постоянно.

Simple Design :

- Обеспечение корректного срабатывания всех тестов;
- Отсутствие дублирующего кода;
- Хорошо выраженные намерения программиста для конкретного участка кода;
- Минимальное количество классов и методов;

Коллективное владение кодом (Collective Code Ownership)

- Каждый член команды может вносить изменения в любой код системы;
- Разработчик может выбрать любую задачу. Специалисты по базе данных не обязаны ограничиваться только задачами, связанными с базой.
- Специалисты по разработке ГИП могут при желании выбрать задачу, касающуюся базы данных. Хотя на первый взгляд это может показаться неэффективным, но выгода очевидна: чем больше каждый разработчик знает о проекте *в целом*, тем успешнее и информированнее вся команда.
- Добиваемся, чтобы за проект отвечала команда целиком, независимо от специализации.

Улучшение дизайна

- Плохой код не должен доживать до заката. Код все время должен оставаться максимально чистым и выразительным.

Постоянная интеграция (Continuous Integration)

- Интеграция модулей в систему выполняется постоянно, несколько раз в день.
- Обеспечение интеграции осуществляется за счет систем совместной работы.

Заказчик в команде (On-Site Customer)

- Постоянная доступность заказчика, который может ответить на возникшие вопросы, пускай даже по телефону;
- Наличие заказчика в команде позволяет работать с оптимальной скоростью.

Частые выпуски версий (Small Realize)

Версии должны выпускаться в эксплуатацию, как
можно чаще. Работа над каждой версией должна
занимать как можно меньше времени. При этом
каждая версия должна быть осмысленной.

40 – часовая рабочая неделя (40-hour Week)

Стандарты кодирования (Coding Standards)

- Весь код выглядит так, будто его писал один – очень квалифицированный – человек.

Благодаря стандартам:

- Члены команды не тратят время на глупые споры о вещах, которые фактически никак не влияют на скорость работы над проектом;
- Обеспечивается эффективное выполнение остальных практик.

Метафора системы (System Metaphor)

- Команда поддерживает общее видение работы программы.

Метафора системы - это архитектура системы.

Метафора системы дает команде представление о том, каким образом система работает в настоящее время, в какие места добавляются новые компоненты и какую фору они должны принять.