# Extreme Programming (XP)

# What is XP?

- XP is a low-ceremony/low-tolerance software development methodology, which emphasises
  - Close cooperation with customers
  - Rapid evolution of design
  - Verbal communication over documents
  - High quality through extensive unit and acceptance testing
- Highly suited for situations where most requirements are not known until later in the project

# Four Values

- Communication
- Simplicity
- Feedback
- Courage

# Fifteen Principles (Part 1)

- Rapid feedback
- Assume simplicity
- Incremental change
- Embracing change
- Quality work
- Teach learning
- Small initial investment
- Play to win

# 15 Principles (Part 2)

- Concrete experiments
- Open honest communication
- Work with people’s instincts - not against them
- Accepting responsibility
- Local adaptation
- Travel light
- Honest measurement

# 12 Practices

- On-site customer
- Metaphor
- 40-hour week
- Planning game
- Refactoring
- Simple design
- Pair programming
- Testing
- Short releases
- Coding standards
- Collective ownership
- Continuous integration

# XP Ideas

- Yesterday’s Weather
  - Today's weather is more than likely similar to yesterday's
  - A developer's productivity today will more than likely be the same as yesterday's – the schedule should reflect that
  - A developer's optimism usually will not change the weather
- Velocity
  - Average of amount of work completed
- Ideal Time
  - Time spent on project with no interruptions

# Facilities

- Entire team truly working as a “team”
- High communication levels absolutely require team to be in the same room
  - XP is incompatible with private offices
- Whiteboards & shared spaces
  - Public visibility of status and critical information
- Two people at a table
  - Tables better than desks
  - Computer monitors must be visible to two users

# The Planning Game

- “Software development is always an evolving dialog between the possible and the desirable” - Beck
- XP project management
- Business people make business decisions
  - Scope, priority, composition of releases, dates of releases
- Technical people make technical decisions
  - Estimates, consequences, processes, detailed scheduling
- Iterations and releases
  - Developers should not give an estimate off the top of their head - It is well known that those developers who think more about estimates are more accurate

# Design

- Architecture with a little ‘a’
- CRC
- UML as a sketch
- Design today for today; refactor tomorrow
- No BDUF
  - Big design up front
- YAGNI
  - You ain't going to need it

# Test-First

- Written by programmers (unit tests) & customers (acceptance tests)
- Putting the concept into practice
- Test-driven development
  - Contrast with use case driven development and feature driven development
  - If requirements states what the system should do, and tests  really also state what the system should do, then what is the different

# Pair Programming

- “A driver and a navigator”
  - (think rally car racing) - not “a driver and a passenger”
- Duties of navigator
- Essentially this is continuous code review
- Helps exchange of tacit knowledge
  - XP does not rely on heavy documentation, so pair programming is the key
- All production code must be written by pair
  - With no exceptions to this rule

# Refactoring

- What happens when software matures?
- “Improve the design after the software has been written”
- When is a developer going to know most about how software should be designed?
  - Right at the beginning
  - Or after a number of months working on the project and communicating with the customer

# Business Issues

- Fixed price/fixed-scope contracts
  - XP is not suitable for fixed scope projects, because scope is variable
- Subscription service
- Responsibilities
- Outsourcing, etc.
- Time and materials basis

# Continuous Integration

- Waterfall model suffers from what is know as the late breakage problem
  - Different components are merged together near the end of the project and they do not work together
  - Big bang ain't good!
- Need to integrate much more frequently
- XP pushes it to the limit, with continuous integration

# Phases

- XP's phases
  - Exploration Phase
  - Commitment Phase
  - Steering Phase
- Do we need phases in software?
- Contrast with waterfall phases
  - Requirements, design, implement, test
- Unified process has iterations, but each iteration is a mini-waterfall

# Customer Role

- On-site customer is essential for XP
  - Need senior customer and it is important that they understand and are truly representative of userbase
- Can be overloaded
  - Very responsible job
- Budgetary issues
  - If customer agrees to extra stories, that in effect increases the budget for the project
- Having on-site customer eliminates much of the confusion that is the cause of many problems on other types of projects

# User Stories

- User stories are how XP handles requirements
- Promissory notes about having to talk to customer
  - Customer always present for details
  - Certain number completed each iteration
  - Basis for acceptance testing
- Excellent book on User Stories
  - “User Stories Applied”, Mike Cohn, ISBN: 0-321-20568-5, Addison-Wesley

# User Stories

- A user story is a one or two sentence identification of how a user may use the system
  - It is not a detailed description of this usage
  - Typically written on index cards (e.g. A5-sized)
  - Card can also store estimated time to implement, priority, and on back a list of acceptance tests
- Just enough information to distinguish from another story and to give a ballpark estimate about how long it will take
  - If it takes longer, not a problem, as XP is not scope-constrained

# The Three 'C's

- Card
  - One to two sentence promise to discuss a feature
- Conversation
  - Verbal discussion with customer
  - Done as the programmer is about to code the story
  - Hence what the customer says is fresh in the programmer's mind; as the project progresses, what the programmer knows about the rest of the project improves; and customer can ensure programmer has the very latest business information to implement
- Confirmation
  - Acceptance Tests

# INVEST Characteristics

- A good user story should have these characteristics (known as “INVEST”)
  - Independent
  - Negotiable
  - Valuable to customer
  - Estimatable
  - Small
  - Testable

# Roles and Personas

- There is seldom a single user, yet the needs of all users are not totally unrelated
- User roles help identify the general categories of users
- User role modelling helps create an organised model of groups of users who share the same characteristics
- Personas are vignettes that provide us with an imaginary description of a user

# Uses for User Stories

- User stories as scheduling tool
  - Each XP iteration should deliver user functionality and hence a certain number of XP user stories must be completed
  - A user story may be subdivided into smaller development tasks, each of which may be tackled by a different pair
- User stories as testing tool
  - Acceptance criteria are tied to user stories – clear customer definition of when the user story is “done” (often something that is difficult to pin down)
  - Some people call these acceptance tests, but usually only state general behaviour – the low level programmatic definition will usually be done by developer

# Epics

- A story should be one or two sentences in length, but how much functionality can that represent?
  - “A comprehensive online portal for an insurance company”
- Such huge stories are known as epics
  - Difficult to estimate
  - Provide top-level view of functionality – ok
- Later on, split into smaller stories
  - When it comes time to actually implement

# User Stories vs. …

- User Stories vs. Use Cases
  - User stories represent real users, not systems (note: user story, not use story!)
  - Use cases contain much more detail that user stories (which is both good and bad)
  - Some hugely complex use cases are being written – better to keep them short and focused (what Cockburn calls use case briefs)
- User Stories vs. IEEE 803
  - IEEE 803 is the famous list of “shall” statements
  - Not tied to users' needs
  - Generally not a good approach (user stories & use cases are superior)

# Cards – Paper or Bytes?

- Benefits of paper
  - Easy to handle
  - Can stick on wall; bring to desk
  - Very visible artefact
  - Just enough space to write what is needed – will not fit a novel (which is good)
- Benefits of bytes
  - Can archive
  - Can use for distributed team
  - Easier to integrate with electronic tools

# User Stories over time

- When completed, tear up and throw away
- There is no need to store over longer term
  - XP uses code as the long term repository of knowledge concerning the project
  - The acceptance tests will also live on, and these are a detailed representation of the user story

# Testing

- Prior to XP, testing was often the Cinderella of many popular software development processes
  - Not too much attention was paid to it initially
  - Realised too late that it was very important
- XP brings testing to the fore
  - It is an integral part of the process
  - You are not doing XP if you are not performing tests
- Two main groups of tests
  - Unit Tests – written by programmers, tests internals
  - Acceptance Tests – written by customers, tests externals

# Desirable Test Characteristics

- Automated
  - Want lots of tests, so need to click a button to run them
- Repeatable
  - Wish to run often (many times every day) so whatever test harness is needed must be provided
- Comprehensive
  - Wish to test many aspects of project
- Self-checking
  - With so many tests, do not wish to have to wade through log files – want clear indicator about errors
- Modular
  - When we have lots of tests, need ability just to run subset (tests should be independent of other tests)

# Test Driven Development (TDD)

- Role of testing changing
  - In the past, people wrote code first, and then wrote tests and debugged code – tests ensured code worked
  - New thinking is that tests play a more significant role as they help to unambiguously define what the code is meant to do
  - Obviously need to know this before starting writing code, so makes sense to write test first
  - Later write code in such a way as to make test run correctly
  - Tests are the driving force behind the development process
- Interesting book - “Test Driven Development”, Kent Beck, 0-321-14653-0, Addison Wesley

# Unit Tests

- Tests a unit of code
  - No standard definition of meaning of “unit”
  - A “unit” can be any block of code that the developers wants it to be (a method, a class, a group of classes, a namespace, a compilation unit - .NET assembly, DLL, JAR)
  - A unit is internal to the application
  - A unit is only of interest to developers (customers do not need to know about these)
- The concept of automated unit testing existed prior to XP, but fair to say XP popularised unit testing
  - There is uniform praise for unit testing
  - Now it is widely used outside of XP
  - Most developers agree it is plain silly to be writing code without the accompanying unit tests

# Mock Objects

- An object is often reliant on secondary objects to assist it in completing its work
- To effectively test such an object, we need these secondary objects also – or do we?
  - Such secondary objects may not be available, or may not be part of the unit we wish to test, or consume some resource, or for other reasons may used by a unit test
  - If instead of these secondary objects we use other objects that expose the same API, would that work?
- When to use mock objects
- Mock objects are not comparable to stub functions
  - Stubs are meant to evolve into production code, mock objects are not

# When Tests Take Too Long

- Large projects can easily have tens of thousands of tests
- Need to think about partitioning tests
  - Or get a very fast test machine
- Concept of test suites
- Might wish to partition tests according to different selection criteria
- Most definitely should run entire test suite once a day (and build entire code base)

# Notes about Unit Testing

- Unit testing is like insurance
- Excellent when you need it
- Need to target what is at risk and less at what will never fail
- Should not have to alter APIs just to accommodate testing
- Put tests in separate assembly
- Internal tests – could use conditional compile and for test compile put tests inside assembly

# Acceptance Tests

- Acceptance tests and requirements are two side of the one coin – stating what the system is supposed to do
  - So why not combine?
- Specifying requirements as acceptance tests
  - Highly controversial approach, but often makes sense
  - Traditionally, requirements came as the initial phase and acceptance testing at the final phase of a project
  - XP does not have requirements/design/coding/test phases
  - An acceptance test determines if the system does exactly as the customer wants – but to do that it really must state what the customer wants - surely that also is what a requirement is
  - Precise encoding of the requirement – in a testable form

# Acceptance Testing

- Unit testing is white box testing
  - Customers not involved at all
- Acceptance testing is black box testing
  - Customers are heavily involved
- Customers are meant to write tests
  - What “write” means depends on the technologies involved and the technical capabilities of the customer
  - Often customers create test assertions (English sentences describing test) and programmers code them up
  - The back of a user story card is an ideal location for customers to write acceptance tests

# Acceptance Test Tool Support

- FIT – Framework for Integrated Test
  - http://fit.c2.com/wiki.cgi?HostPlatforms
  - Available for Java, .NET, Python, Lisp, Perl
  - Spreadsheet-like approach to testing – customer enters values; developer extracts them and runs tests
- Fitnesse (fitnesse.org)
  - Wiki and acceptance testing framework that uses FIT
  - Does not need a web server

# Improved Estimating

- The more I know about a topic, the better I can estimate

- XP has significant advantages over waterfall model when it comes to estimating time needed for features
  - With waterfall model, need to estimate for all features needed for e.g. next year – at the beginning of this period
  - No feedback loop for estimating
  - For first few features, the estimation quality for XP is the same as for  waterfall (working off the same knowledge level)
  - Developers are constantly learning, and after a few (monthly) iterations, are much more knowledgeable about project)
  - As they learn more and more about the system, later stories can much more accurately be estimated

# Tackling the Unknown

- Picture the scene – meeting room full of development team discussing scheduling for the next project iteration
- Discussion moves to a new complicated feature
- Project Manager: “How long will it take?”
- Developer responds in one of three ways:
  - (speaking truthfully): “It will take as long as it takes”
  - (speaking truthfully): “I don't honestly know, but if I take half a day to investigate, I will be able to come back with an accurate estimate”
  - (wildly guessing): “Three days” (may end up being between three and ten days) [will never be less]
- If you were the project manager, which answer would you prefer?

# Knowing What We Don't Know

- Rumsfeldian comment
  - There are things we know we know [ok]
  - There are things we know we don't know [ok]
  - And there are things we don't know we don't know [yikes!]
- The last is extremely dangerous on software projects
  - and the primary reason for the failure of many projects
- If developers do not know how long a story will take, they should say clearly acknowledge this during planning
- XP tackles the known unknowns with spikes

# Spike

- A spike is aimed solely at estimating how long a user story will take to implement
  - It is whatever is needed to come up with the estimate
  - Could be some prototype code, research, test scenarios, talking to people, or anything else
  - Focus on coming up with accurate estimate
- A spike is a user story itself
  - Often a good idea to put the spike in one iteration and later, when the estimate is available, put the actual implementation in a later iteration
    (Or not – if the customer, upon learning how long it will take, says it is not that important)

# Why Some XP Projects Fail

- As happens with any software development process, some projects that use XP fail
  - XP is very good at completing the current project, but not so good at setting it up for the subsequent project
- The only artefact that remains at the end of an XP project is the source code
  - No architecture document; no UML model; no requirements document, nothing
- There are intermediate artefacts created during an XP project, but these are discarded
  - User story cards, UML sketches, CRC diagrams
- If customers wants doc, must add user story that explicitly requests it

# Where To Use XP

- Highly dynamic projects, where requirements change substantially
- Customers do not know what they want until they see it
- Highly disciplined development team, who will follow ALL of XP's interlinked practices
- Exploratory projects, that are little understood initially and significant experimentation needed
- Same team (at least most of) is likely to on the project for a long time
- Colocated team
- Customer is available who can speak with a single voice