Extreme Programming in Practice

The Why and the How of it, from the front line.

Personal Introduction

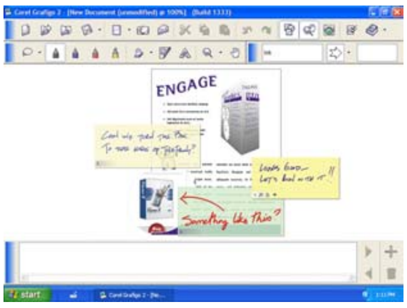
Dana Smith

- Graduated from Systems Design Engineering at Waterloo in 1995
- Developed PHOTO-PAINT until 1999
- Managed teams until January of this year
- Lead Bitmap FX, KPT, Painter, Grafigo, XMP, and SVG Viewer teams
- Involved in process definition and improvement from the beginning
- Helped design and implement the process used across engineering
- Used Extreme Programming for Grafigo, XMP, and Viewer

Agenda

- Detailed description of Extreme Programming
- The details in context: Grafigo and Viewer
- Putting it in practice
- Personal Observations

Corel Grafigo


Corel Grafigo 2 - [New Document (unmodified) @ 100%] (Build 1333)
ENGAGE
Looks Good—
Let's Build on it!!
Something like this?
start

Corel Grafigo

- July to October 2002
- New code base
- 5 full time and 2 co-op development staff
- Experienced team
- A User Experience Designer (designer)
- A project manager (PGM)

Corel SVG Viewer

- SVG - Scalable Vector Graphics
- W3C XML-based Standard
- The Viewer renders SVG files with a browser window
- January 2002 - January 2004
- Multiple teams, no clear customer

Cross Media Publishing

- Never made it past the sandbox stage
- An XML based language for abstracting presentation from date in publications
- Completely Extreme
- Too short to make much of an example
- Highlights:
  - 200 lines of code per person day
  - That's about 10x better than average
  - Extraordinarily malleable code base

What is Extreme Programming?

The Agile Manifesto

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

That is, while there is value in the items on the right, we value the items on the left more

The 12 Practices

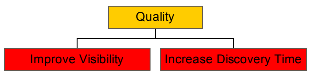
- Planning Game
- Frequent releases
- Constant Integration
- Sustainable Pace
- Collective Code Ownership
- Whole Team
- Pair Programming
- System Metaphor
- Coding Conventions
- Simple Design
- Design Improvement
- Constant Testing

What is the goal?

- Become more reactive to the needs of the customer.
- Extreme Programming is a strategy for creating high quality software

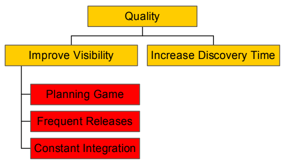
The Strategy

- Improve project visibility
- Increase discovery time



Improving Project Visibility

- Ensures status and progress information
- Three part strategy:
  - Planning Game
  - Frequent Small Releases
  - Constant Integrations



Planning Game

- Clearly sets out everyone’s role
- The customer defines and orders the work
- Developers provide time estimates
  - Gives the customer a sense of cost
- Estimates have associated risk
  - Developers negotiate priority based on risk
- The outcome is a project plan
  - A complete list of tasks is ideal

Frequent Small Releases

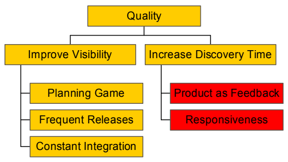
- Provides measure of progress
  - The most valuable functionality is added
- Work breakdown has to be functional
  - User Stories
  - Iterations

Constant Integration

- Visibility to other developers
  - Hourly check-ins
  - Ensures that everyone knows what everyone else is doing
- Project must remain fully functional
  - Leads to small grained evolution
- Continuous Tracking
  - A rolling tally of the amount of work done
  - Velocity and Load

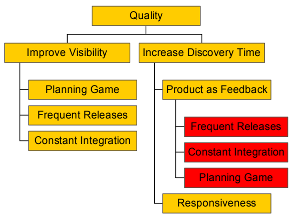
Increasing Discovery Time

- The plan changes with new information
  - Changes in the environment
  - Using the product
- Support a reaction by the customer
- Two part strategy:
  - Using the product as feedback
  - Responsive development



Use the Product as Feedback

- What the customer can do with the product determines the quality of their feedback
  - Imagine, See, Handle, Use
- Second purpose of the practices for Improving Visibility



Frequent Releases Revisited

- Satisfies the need to have something useful to evaluate
- Ensures frequent feedback, saving time

Constant Integration Revisited

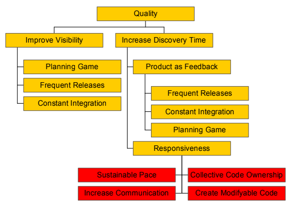
- Elicits feedback from other developers
- Ensures that you don't have integration problems

Planning Game Revisited

- Completes the feedback loop
  - Developers deliver the most important things first
  - The customer uses them
  - The schedule is adapted according to findings

Responsive Development

- Responding to feedback has a cost
- Strategy to mitigate the cost:
  - Sustainable Pace
  - Collective Code Ownership
  - Increase Communication
  - Create Modifiable Code



Sustainable Pace

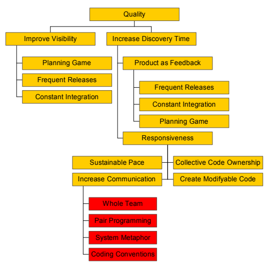
- Each member needs to be at their peak
- Everyone has a limit, beyond which they are counterproductive
  - Make a negative contribution
- Sprints are effective and necessary
  - They can't be relied upon
  - They can't be the norm

Collective Code Ownership

- Specialists vs. Generalists
- Defensive mechanism
  - Truck number
- Reactive mechanism
  - Working in priority order
  - Fewer mistakes
- Favours seasoned developers

Increase Communication

- To respond to the customer, you have to communicate with the her.
- Strategy to ensure proper communication
  - Whole Team
  - Pair Programming
  - System Metaphor
  - Coding Standards



Whole Team

- Lack of up-front detail results in questions
  - If the developer can't get answers, he must make assumptions
  - Stopping is the worst thing he could do
- Having an accessible customer mitigates the problem
- Customer proxies have a difficult task

Pair Programming

- A "driver" and a "navigator"
- Increases collective code ownership
- Increases level of communication
  - Learning happens
- Constant code review
- Saves time in the long run

System Metaphor

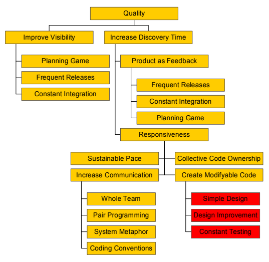
- A way of communicating the vision
- Illuminates the customer's implications
- *Very* difficult to come up with

Coding Standards

- Einstein's suits
- You don't want to waste time trying to figure out how it should look
- Ensures a common look to the code
  - Makes it easier to understand

Create Modifiable Code

- When the requirements change, so must the code
- Traditionally, nothing explicit to avoid brittle code
- Writing malleable code:
  - Simple Design
  - Design Improvement
  - Constant Testing



Simple Design

- Do the absolute minimum required to get the job done
  - You Aren't Going to Need It
  - Don't anticipate - react
- Plan for everything you know about, ignore everything else
- Laser-point focus on functionality

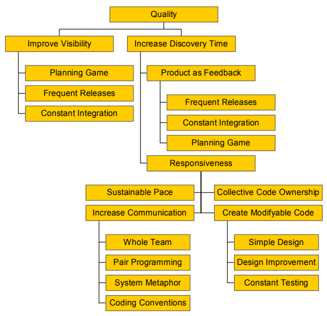
Design Improvement

- Simple Design starts things off right
- Refactoring keeps things clean from there
  - Avoid compromises by changing bad code
  - As developers improve, so does the code

Constant Testing

- The key reactive practice
  - Fearless response to change requests
  - Fearless response to bad code
- Two types:
  - Acceptance tests for customer's requirements
  - Unit tests for developer's intent
- Should all be automated

The Strategy



Putting it into context

- How these concepts were integrated into the Grafigo and Viewer projects

Grafigo

- Team was open to anything
- Started with a specification
  - Wrote stories
- Whole Team meeting for estimates
  - Zope, MS Project
- Me, PGM and UED met regularly to prioritize/add stories
- Built iterations 2 weeks in advance

Grafigo

- We had a coach who held us to:
  - Constant Integration
  - Simple Design
  - Test Driven development
  - Design Improvement
- We stopped writing tests for a while
  - People got scared and we started again
  - Test first design helps avoid this

Grafigo

- Tried Peer Programming a few times
- We had *constant* communication and support from the customer proxies
  - Conversations revealed willingness to negotiate
- Other stakeholders weren't so involved
  - The focus split
  - We needed an emergency course correction

Grafigo

- Transitioned to another team
  - Couldn't just move the code and ideas
- We transitioned the culture
  - Worked through 2 iterations with members of the other team
  - Introduced them to our way of working from a position of majority
- They're still doing things this way, on other projects

Grafigo: Practice Summary

- Things we did:
  - Worked from stories
  - Tracked Velocity
  - Frequent Releases
  - Constant Integration
  - Simple Design
  - Unit test
  - Design Improvement
  - Sustainable Pace
  - Coding Conventions
  - System Metaphor
- Things we didn't do:
  - Plan regularly
  - Pair Programming
  - Include QA
  - Collective Code Ownership
  - Automated Acceptance Tests

SVG Viewer

- Team was very closed minded about XP
  - They had a very ad-hoc approach
- There was no clear customer
- Product had never successfully shipped
- The code was a mess of patches
- It was painfully slow
- Tasks were not traceable

SVG Viewer

- Spent first five months clearing out pending work
  - Bug fixing
  - Performance tuning
  - A small number of new features
- Had a successful release

SVG Viewer

- It was very difficult to use the XP practices
  - There was no customer
  - Work was defined in large chunks
  - People resisted changes their practices
  - They were a team of specialists
  - All the tests had been commented out
- Fantastic regression suite

SVG Viewer

- Contracted to create a C++ SDK
- Worked with the customer to generate a list of requirements
- Architecture wasn't meant to be exposed
  - Much of the work was refactoring infrastructure

SVG Viewer

- Writing stories was very hard
  - The functionality had a small footprint
  - The work had a large footprint
- We planned iterations and tracked velocity

SVG Viewer

- Refactoring requires tests
- To refactor the architecture we:
  - had the regression suite
  - started building firewalls
  - put the old unit tests back in
- We had a good release

SVG Viewer

- Universal Maps was our next customer
- The Viewer couldn't render all their maps
- Correcting that was a huge undertaking
  - This gave us terrific focus
  - Even when things got *really* bad, this pulled us through

SVG Viewer

- Like the SDK, stories were hard to write
- We had developer, not user stories
- We underestimated by a factor of 3
- We had to integrate a month and a half of work at once
- New automated test suites were written to validate that work
- Before we were done, the department was shut down

SVG Viewer

- We made QA part of the Whole Team
- They wrote acceptance tests based on information from the customer
  - Time for their execution was included in the story estimates
- We treated bugs like any other story
  - They were integrated into our list of stories and dealt with at planning meetings
- Development saw QA as a customer
  - Stories were marked "done" only when the tests passed
  - We had stories for doing work to make automated testing easier

Viewer: Practice Summary

- We really used all the practices
- Our failure was in writing the stories
  - It was catastrophic for our ability to plan
- We succeeded from a technological standpoint
  - The code was much less brittle
  - More efficient
  - Fewer bugs
- We were poised to race ahead

Implementing Extreme

- When is it Appropriate?
  - In particular:
    - When the customers don't really know what they want
    - When the discovery time leaves insufficient time to implement
  - It's valuable for any project
  - The more experienced the team is, the easier it is
  - Certain aspects of XP rely on good judgement
- What I've learned about making it work
  - Practices that met no resistance
  - Contentious practices

Easily Accepted Practices

- Planning Meeting
- Constant Integration
- System Metaphor
- Coding Conventions

Planning Meeting

- The meetings are invaluable
- They:
  - increase communication dramatically
  - give everyone a sense of their role
  - put the roles in contrast
- Nobody every questioned their value

Constant Integration

- This follows naturally from Frequent Releases
  - Any arguments can be handled with the same responses

System Metaphor

- It's a really good communication tool
- My own experience is that it was either totally obvious or nearly impossible to generate

Coding Conventions

- Doesn’t *everyone* have coding standards by now?

Contentious Practices

- Stories
- Iterations
- Sustainable Pace
- Collective Code Ownership
- Whole Team
- Pair Programming
- Simple Design
- Design Improvement
- Constant Testing

Stories

- Stories can be difficult to write
  - Yes - it takes practice
- The value is not apparent
  - Everything is expressed in the customer’s terms
  - Enables frequent releases and feedback without prototyping
  - You don’t waste time specifying things that change
- It’s not clear how this is different from other types of specification
  - The customer can test each story for completeness

Iterations

- Takes too much time to keep everything releasable
  - It does take longer, but you save time too:
    - There’s no integration at the end, and the big problems always surface during integration
    - You get the necessary feedback from the customer
- It’s not enough time to get things done
  - Break the work down into pieces that fit into an iteration
  - Each iteration is a successive approximation of the requirements

Sustainable Pace

- It’s counter-intuitive that 60 hours of work doesn’t generate 60 hours of output
  - People are not machines; their output varies with:
    - Stress
    - Time of day
    - Environment
  - It’s not about limiting commitment, it’s about maximizing efficiency.
  - This one comes down to a matter of opinion. Go with personal experience.

Collective Code Ownership

- Many developers are threatened by this
  - It’s a chance for them to learn more
  - I’ve yet to find a developer who doesn’t want to learn
- You need specialists to get the really tough stuff done
  - Working to be generalists doesn’t mean you loose specialization
  - Everyone still has areas of affinity and are recognized experts in some

Whole Team

- This is hard because it goes beyond development
  - If you don't have authority over Project Management and QA, find a sponsor who does
- At Corel, we really didn't have anyone who was a customer proxy.
  - We used people to this end, but it wasn't their defined role
  - In most cases, they weren't qualified
- No surprise that access to customer or proxy is needed
- Implications are overlooked
  - The proxy needs industry expertise
  - The customer needs to commit a great deal of time

Whole Team - QA

- QA Team should be writing acceptance tests/specs
  - If the customer hasn't said it's important, it isn't a bug
- It's harder to get acceptance for this
- The QA team itself is resistant
  - They believe it's necessary to eliminate all defects
  - Leads to excessive system testing
- Relies *heavily* on automated testing

Pair Programming

- Everyone thinks this is a great idea
- Nobody wants to do it
- After enforcing this for an iteration or two, I've been making it elective

Simple Design

- Reusable code saves money
  - Absolutely, but you can't predict the need
  - Better to generalize the code when you actually need to reuse it
  - You don't waste time doing things that aren't useful
- Beware of extra work disguised as 'Good Practices'
  - Good Practices keep the code from getting brittle
  - Anything else is work you don't need to do

Design Improvement

- Retouching code creates bugs
  - That's why we do unit, acceptance and regression testing
- Don't waste time on something that's working
  - The time was saved when the work wasn't done to begin with
  - You'll save time without the problems caused by bad code

Constant Testing

- We don't have time
  - No practice I've ever seen saves more time
  - Short term: avoids bugs
  - Long term: code stays malleable
- Once the pressure's on, this looses priority
  - Measure the coverage and have a minimum level
- Test First design actually works best.

Personal Observations

- This analysis has allowed me to realize a few things:
  - Why bug blitzes typically work well
  - How to motivate
  - The value of testing
  - The value of planning

Bug Blitzes

- Bugs are small, user-facing pieces of work
- They typically take a very short amount of time to do
- They are usually prioritized
- Bug Fixing blitzes are usually highly focused and productive
- Progress is very measurable

Motivating people

- Motivation is intrinsically self serving
- Staff know that they have a job to do
  - They want to do it well
  - Managers have an obligation to protect and support this
- They need to be given clear expectations
  - The goal and motivation for the project need to be crystal clear
- This gives them a context in which to find motivation
  - The goal and motivation don't even need to be good!

The Value of Planning

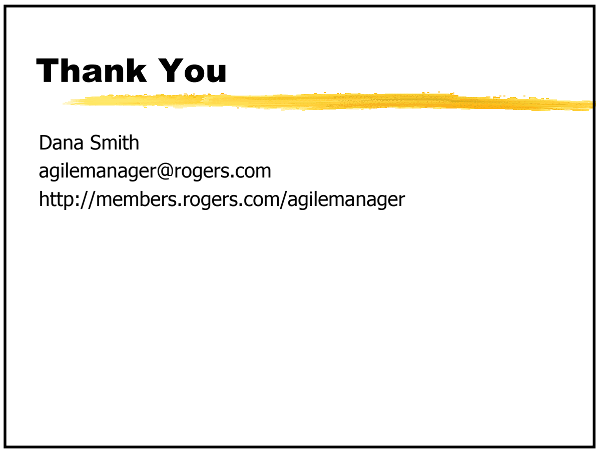
- The value of a plan is in breaking it
  - Someone's doing unplanned work
  - Something takes longer than planned
  - A task is started before something of higher priority
- Focusing on keeping the plan is counter productive
  - People hide the fact that they're behind until it's too late
- Should focus on meeting the customer's needs

The Value of Testing

- How do you know when you're done?
  - When everything on 'the list' is finished?
  - How do you the tasks themselves are done?
- Doneness is a matter of *opinion*
- The only opinion that counts is the customer's
- Acceptance tests express the customer's opinion
  - Implies that you need at least one for every story
- Until the customer accepts the work, it's not done
  - You have no idea how close you are - only a guess
  - Progress reporting should be binary

Emergent Behaviours

- The 12 Practices interact to create higher order behaviours
- Generates a *culture* reinforced through peer pressure
- What I see:
  - Unity
  - Courage
  - Focus
  - Control
  - Productivity



Thank You

Dana Smith
agilemanager@rogers.com
http://members.rogers.com/agilemanager