© Niall Ross, 2001
For use in XP training.
v2.0, 2Q01

XP-rience: eXtreme Programming Experience

Niall Ross nfr@bigwig.net

Confessions of a (not that) eXtreme Programmer leading a teleworking team building a Meta-Programming and Meta-Modelling Framework for Network & Service Management.

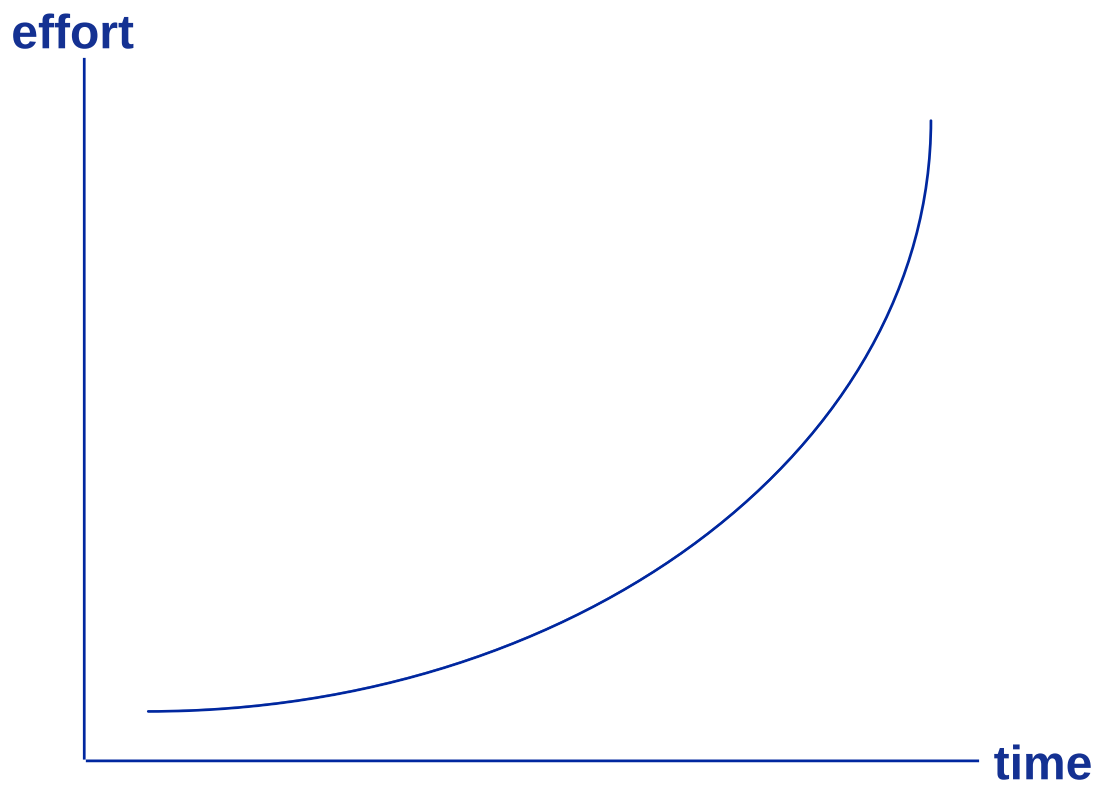
Outline of Talk

- The XP method
  - testing, refactoring and pair-programming
    - experience of each, how they combined
    - the role of architecture and design
    - pair programming experiment
- The XP process
  - what we thought it was
  - our experience of it
- Introducing XP to a team
  - Kent’s advice - pure XP from day 1
  - Niall’s practice - (comparatively) less strict
- XP FAQs
  - handover (unverified)
  - scaling up (plan based on others’ experience)

The Cost of Being Wrong

Each fault costs more to fix the later you fix it

Make decisions early; freeze them early

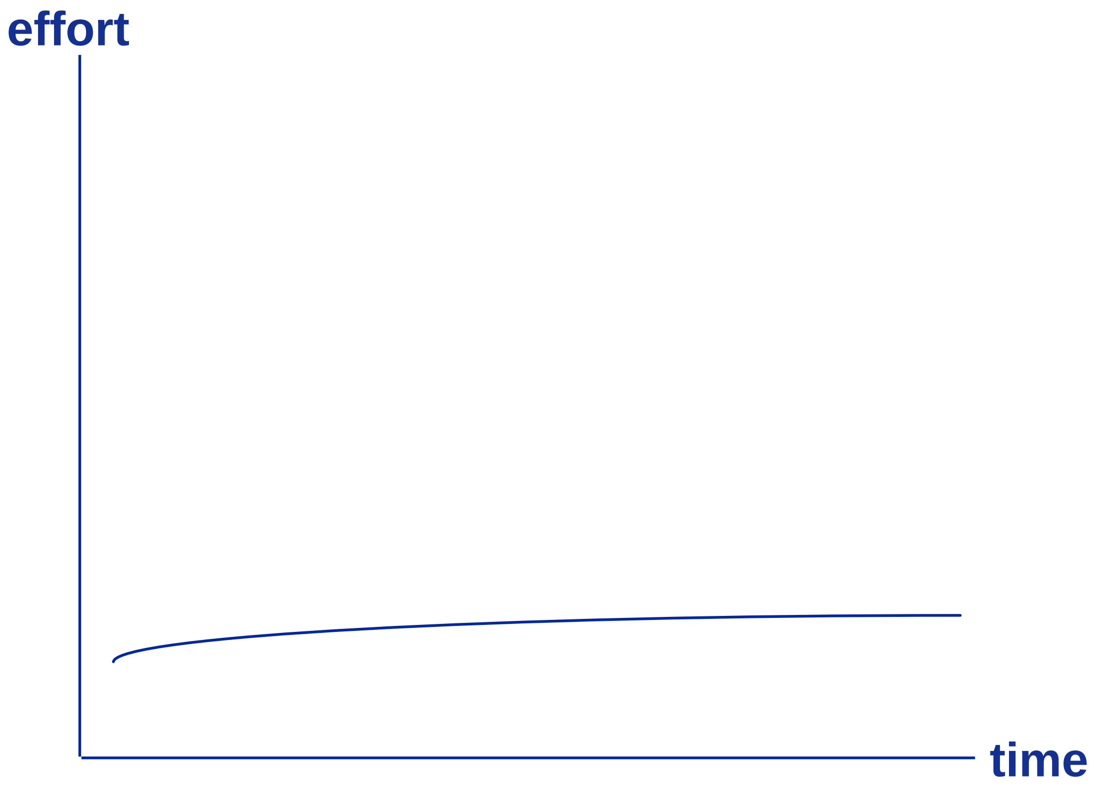


The Cost of Being Meta-Wrong

Software is not hardware

Software is malleable

The cost to fix can be steady



Experience of Test-before-Code (1)

- The need for speed
  - the current test / the whole test suite runs in
    - 10 seconds / 10 minutes: fine
    - 5 minutes / 1 hour: stop and refactor
  - ‘no time to redo test, must deliver’ WRONG!
- Beware ‘necessary but not sufficient’
  - starting with a necessary test is essence of XP
  - complete the test set
    - partition axes (e.g. lifecycle stages, config types)
  - complete the assertions
    - two wrongs don’t make a right ...
    - … but may pass a failure-reliant test
  - general domain truths: put in generic classes ?
    - e.g. tp realizer connector == tp connector realizer
- Write tests, not design docs or task lists

$$\begin{tikzcd} {} \arrow[r, "c"] \arrow[d, "r"'] & {} \arrow[d, "r"'] \\ {} \arrow[r, "c"] & {} \end{tikzcd}$$

Experience of Test-before-Code (2)

- Test case pattern only implicit in paper
  - formalizing it helped achieve common style
    - combine test cases via inheritance and delegation
- Needed flyweight pattern (SUnit 3.0 has this)
  - reuse/reset complex test configurations when
    - expensive to build or tear down
    - invariant or resettable under the tests
- Evolve hard-to-compute test results (J. Pelrine)
  - chose initial config to test simple service
  - computed deltas to config for complex service
- Testing the UI
  - ‘some XP-complete projects only test 50% code’
  - Kent says ‘O.K. to skip UI tests’ (paraphrase)
  - mea culpa !!! (and it sometimes hurt)

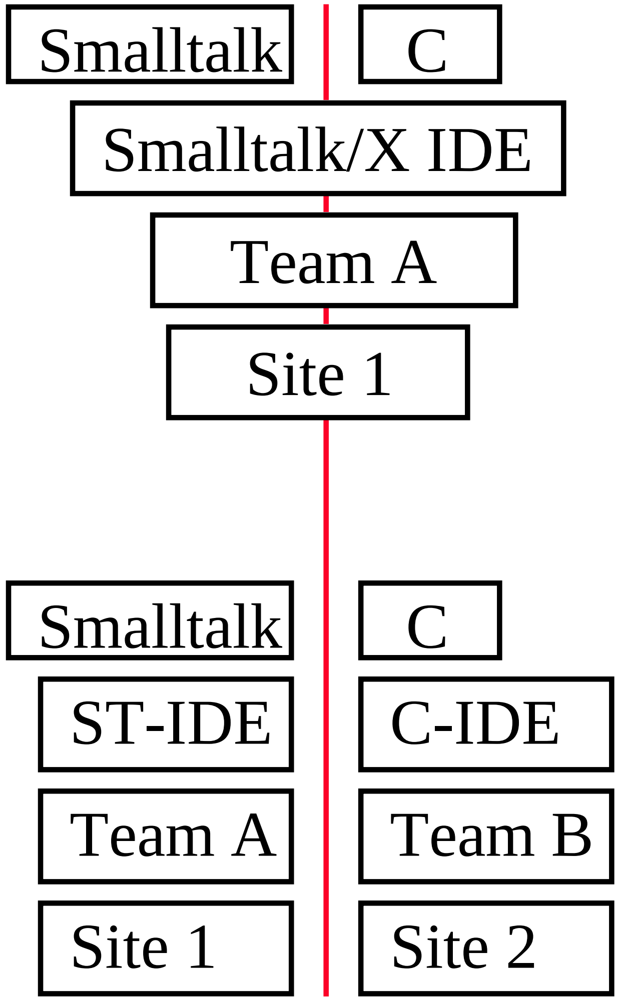
Experience of Refactoring (1)

- All coding is refactoring
  - theory: never write a line without a broken test
  - practice: after a while, we almost never did
- The need for elbow room
  - deliveries inside increments kill refactoring
    - must break to remake
    - customer understood (enforced cycles on his team)
      - 'We'll push on you, you push back on us.'
    - our new manager did not (survive :-)
      - 'Mustn't annoy them, let's do it this once.'
- What role for Architecture?
  - Refactoring demands fine-granularity
    - must be able to move behaviour incrementally
  - Barriers need architecture (or removal)
    - our ST-Corba-Java interface needed an architecture

Another Architecture Example

1) Low tax on refactoring;
use XP across boundary

2) High tax on refactoring;
architect boundary



Experience of Refactoring (2)

- What role for design?
  - XP says, ‘Let the code teach you’
    - pair-program sessions create design fragments
      - tests are primary, configure-controlled
      - models quickly sketched, useful in discussion
    - hard to reuse / refactor models due to poor tools
      - drawing tool: no understanding (bad)
      - modelling tool: wrong understanding (worse)
  - Documentation for new starters
    - most of our designs wrong / out-of-date
      - as XP says they will be
    - wrote some post-hoc designs
      - they were wrong / out-of-date too :-)
  - Design to find what tests to write
    - brief initial design helped a major refactor
    - impromptu sessions re non-code issues: helpful?

Notes for prior slide

Architecture and Design

(Slide not shown, only for notes.)

Experience of Pair-Programming

- The hardest part of XP?
  - ‘We don’t have time to pair-program’
    - you don’t have time not to
    - pays back surprisingly quickly
    - a useful pacing mechanism
  - ‘If only I hadn’t so many meetings’
    - stop meeting to talk and separating to work
    - start meeting to work and separating to think
- Pair-compatibility issues
  - powerful when both are system-experienced
  - used as training mechanism; mixed results
- Teleworking tool + handsfree phone ideal
  - better than sharing one mouse and keyboard
  - puts locals and teleworkers on even footing
  - collateral benefits: better split-site working

Notes for prior slide

Teleworking and Pair-Programming

(Slide not shown, only for notes.)

Experience of Combinations (1)

- Refactoring and Pairing
  - let the code (and talking about it) teach you
    - discover a better design than you could deduce
- example 1) object subClass: #class pattern
  - I built lightweight metaclass for special case
    - code told me, ‘metaclass wants class’ behaviour’
    - pair asked me, ‘Why not do that everywhere?’
    - suddenly we had a working system
- example 2) model-or-user-driven pattern
  - I wanted model-driven algorithms
    - but couldn’t solve every case
  - pair wanted user-driven, model-constrained
    - but couldn’t make UI comprehensible
  - union was ‘simplest thing that could work’
    - and result was tool for finding better algorithms

Experience of Combinations (2)

- Refactoring and Testing
  - basic XP: tests let you refactor
    - 100%: refactor broke feature => feature lacked test
    - 99%+: refactor broke feature, area lacked tests
- Pairing and Testing
  - pair-programming a test defines a task
    - especially good for new starters
  - test lets pair agree understanding of task
- Refactoring, Testing and Pairing
  - I several times experienced the sequence:
    - my refactor fails someone else’s tests
    - my pair worked with them earlier, so explains code
    - pair fixes, fails refactor test; I fix, get better refactor
  - discuss, and so defeat, ‘shy arrogance’
    - ‘I can’t pair till I’ve worked out how to do this task’

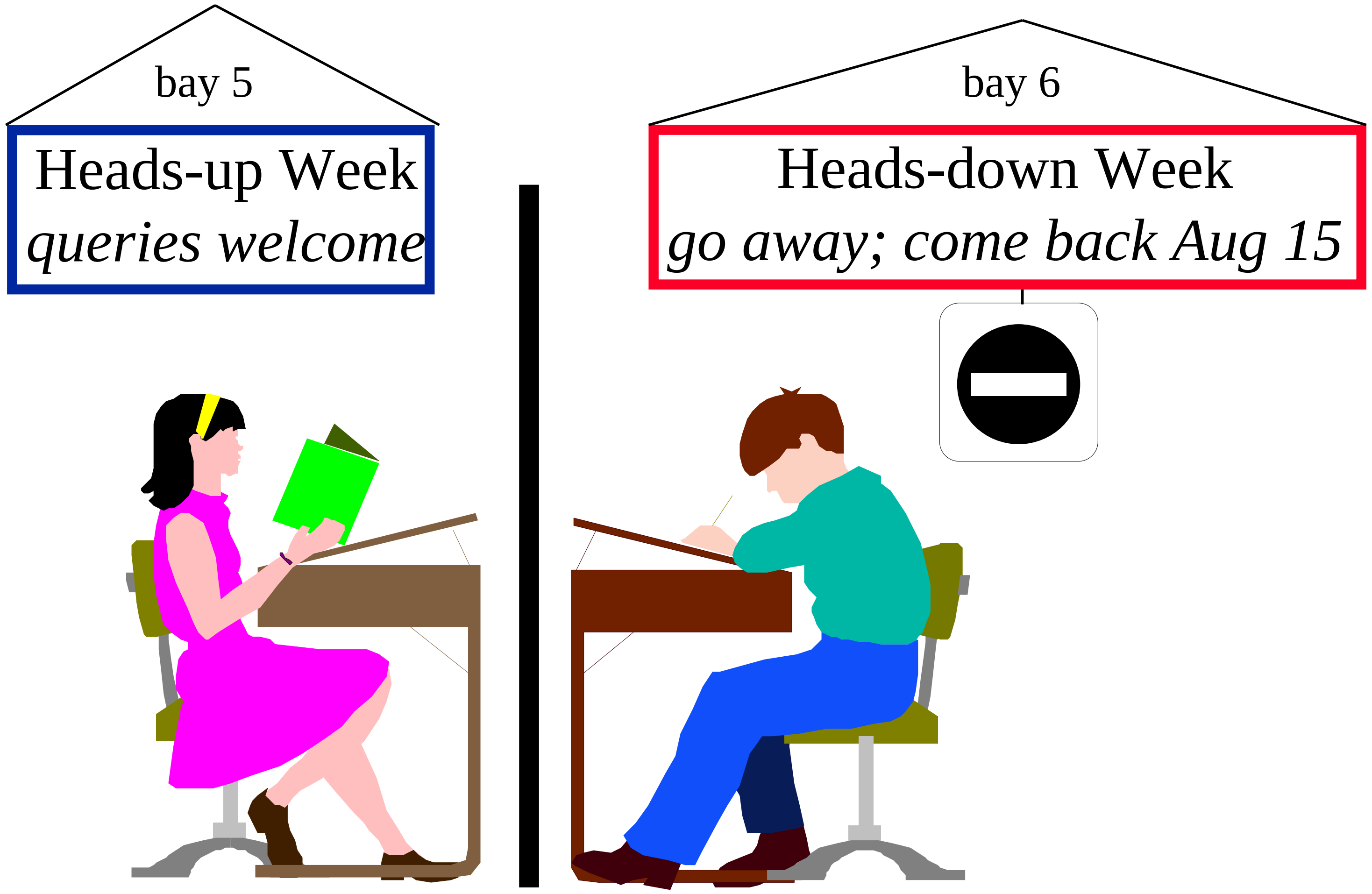
Pair-Programming Experiment

- Popularity
  - doubting start grew to 82% paired coding time
    - (industry: some always pair, some feel ‘burnt-out’)
    - (I browsed code alone, trying things out)
  - compatibility: rotate to ease clashes
    - 2 x expert great, 2 x novice good, expert-novice OK
    - extrovert-extrovert slow !!! (but they liked it :-)
    - introvert-introvert good training for v. introverted
- Effectiveness
  - paired code is of higher quality
    - pairs always write test cases
  - pairs take much the same effort
    - median: same effort, 50% elapsed time) due to 2
    - average: 115% effort, 58% elapsed time) outliers
    - pairs defeat parkinson’s law and ratholes

Our Experience: Process

- Discipline customers and managers with:
  - the planning game
    - customers always want everything yesterday
    - to get maximum value from technical synergy
      - customer maps stories to values
      - developer maps stories to iterations
  - iterative cycles
    - once upon a time we revectored every year
    - revector every month = process
    - revector anytime = no process
- Need to coach customers and managers
  - XP doesn't say how; our best customer
    - already believed in iterative cycles
    - was idea-rich but time-poor
    - was erratic in pushing XP to his team
  - XP says, 'refuse the ones who won't learn'

Interrupts: an eXtreme Solution :-)



Introducing XP: what Kent says

- Preach simple design
  - communicate in source code
    - no configure-controlled design docs
  - never duplicate logic
- Enforce XP’s rules on the team from day 1
  - if you don’t need it, don’t do it
    - no line of functional code written without broken test
  - pair program
    - no line of production code written unpaired
  - keep the feedback loop short
    - build system and run all test cases every day
  - work smarter not harder
    - concentrate work in middle of day
    - be exhausted by 16:00
    - 40-hour maximum working week (popular :-)

Introducing XP: what Niall did (1)

- Preached the theory: created champion(s)
  - talk (wanted workshop, summer school, ...)
- Started with Testing
  - chose test utility
    - ?Unit from www.XProgramming.com/software.htm
    - commercial testing tools that are XP-aware
  - for each team member, assigned starter task
    - champion pair programs test with them
    - lets them code task, running test often
  - gradually enforced rule
    - each task must have test(s)
    - all tests must be written paired or code-reviewed
  - once test suite built up, took time to
    - refactor tests for speed and common style
    - extend suite to a reasonably complete set

Introducing XP: what Niall did (2)

- Next came Refactoring
  - made first major one visible (team discussion)
    - the need to refactor arose naturally
    - waited till test suite was large enough for safety
  - got bolder as our test suite grew
- Grew into Pair Programming
  - started gently
    - thou shalt pair-program for 2 hours each week
    - thou shalt pair or review all tests
    - started by pairing equals
  - when team opinion leaders won over
    - upped weekly paired hours
    - rotated pairs
  - discussed, and so defeated, ‘shy arrogance’
    - ‘I can’t pair till I’ve worked out how to do this task’

Introducing XP: Up-front Costs

- Up-front costs: technical
  - finding, loading, learning the test utility
  - getting used to test-failure-drives-coding
    - cost is IDE-dependent
    - some benefit in all IDEs
- Up-front costs: non-technical
  - some customers / managers are keen
    - ‘the creative engagement of combative intellects’
    - ‘let’s pair-program tests to define use cases’
  - some give the uncommitted ‘yes’
    - put ‘heads-up / heads-down week’ on everything
  - some fight it
    - ‘XP doesn’t need those documents and meetings’
      - ‘those documents and meetings are what I’m about’
    - never give way, never give (avoidable) offence

FAQs (1): Handover ?

- Can you sleep / handover an XP project?
  - I followed VCAPS model
    - wrote 10 page document with pointers to tests
    - seemed O.K. to me
    - wished I could reuse design discussion diagrams
  - our handover process is not yet verified
    - recipients were reassigned
- Whether or no, should you worry about it?
  - XP's philosophy is about opportunity cost:
    - don't waste time preparing for unlikely events
    - you can refactor to handle them if they occur
  - (Almost) 'nothing that dies ever comes back'
    - spend your time on useful features => no handover
    - handover ifTrue: [self writeBetterDocNowThanBefore]

FAQ (2): Scaling Up ?

- Some say scale is an issue, others say not
  - Projects of 25 have used XP: ‘the tools creak’
  - I lack experience but have contacts
- Scaled-up extreme s/w eng process (eCom)
  - three teams with parallel increments
    - Xanalysis: 2 ‘relationship people’ model domain
    - Xreqts: 2 ‘lawyers’ write OCL use cases
    - XP: 8 programmers do standard XP
  - each team’s output feeds others’ next cycle
    - XP hate rate of business revectoring XA like
    - XA terrified by rate of system refactoring XP like
  - some programming styles collapse XR and XP
    - yes: Smalltalk, Prolog, Lisp, …
    - no: C++, Java, ...

Extreme Programming

Use it !!!

Acknowledgements

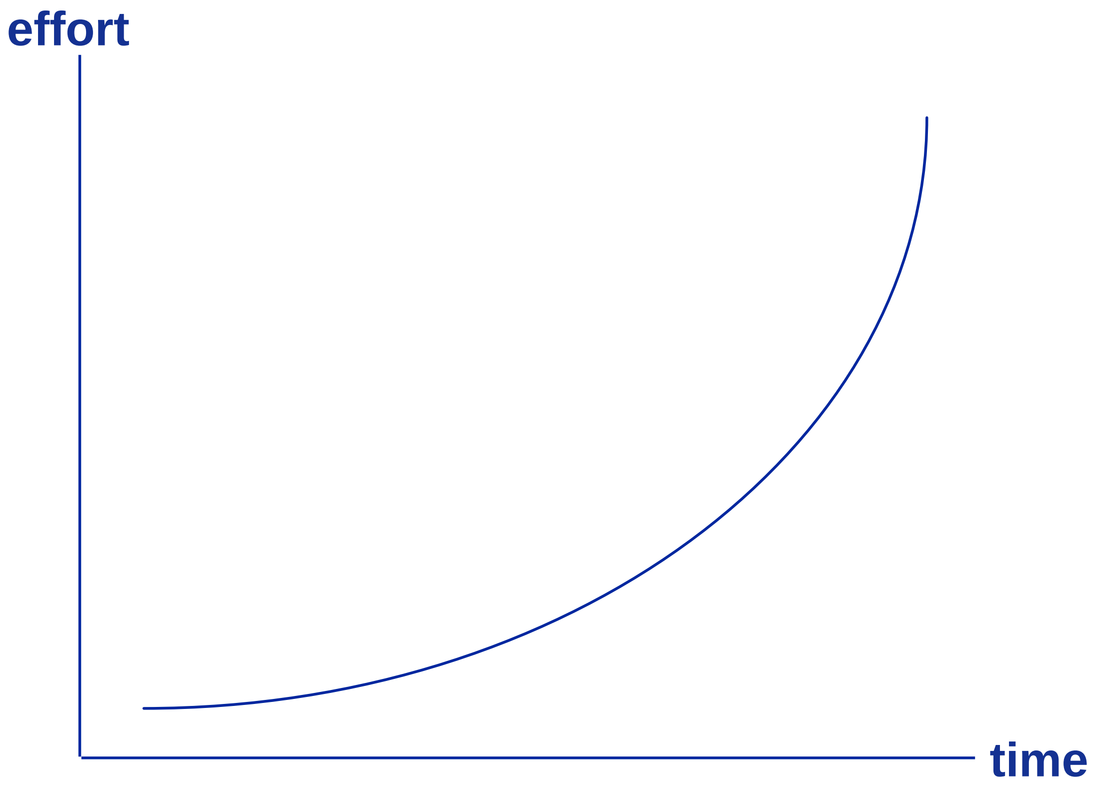
Thanks to my team (Gill Kendon, Steve Gaito, Jessica-Anne Hainey, Mike Hurd, Ian Corrie, Jonathan Durrant, Juan Barbieri, Fiona Davison, Bruno Buzzi) and to Joseph Pelrine, Steve Forgey, Laurie Williams, Stuart Kent and, of course, Kent Beck.

Backup slides

The Cost of Being Wrong

Each fault costs more to fix the later you fix it

Make decisions early; freeze them early

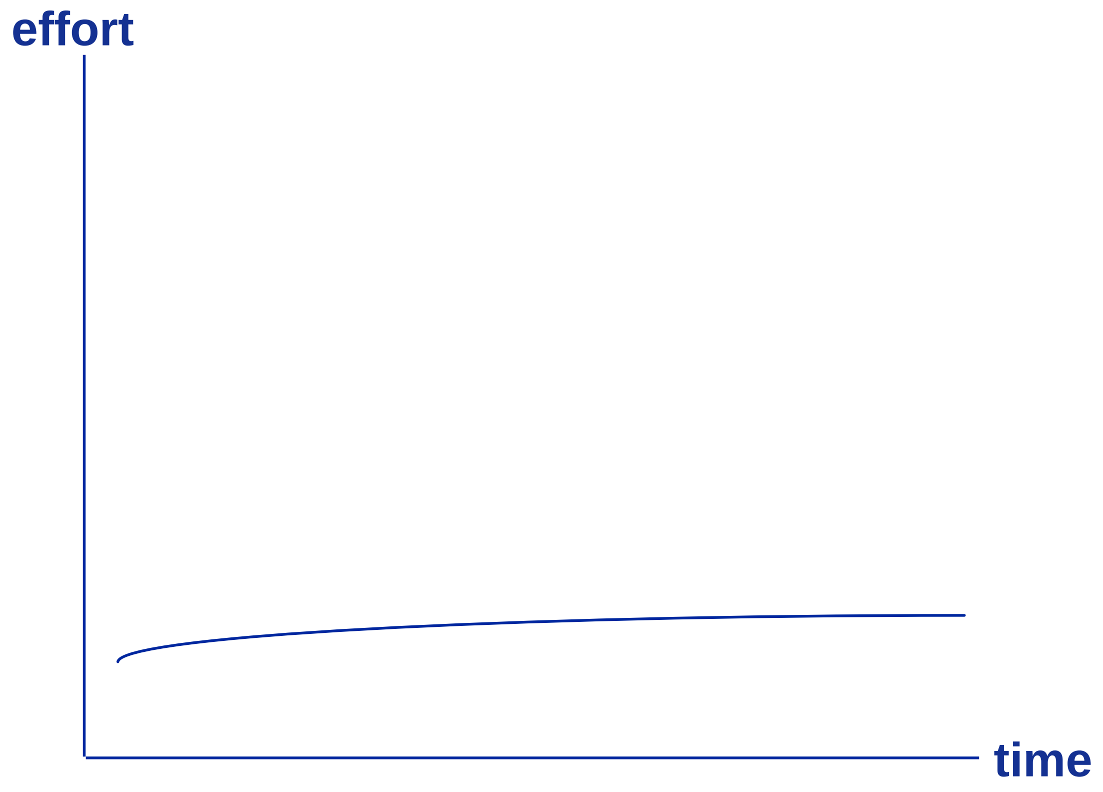


The Cost of Being Meta-Wrong

Software is not hardware

Software is malleable

The cost to fix can be steady



The Key Ingredients

test

- **Write the tests, then the code**
  - can't code what you don't know how to test
  - the test proves your function still works
    - next day, week, month, year
  - only write tests
    - that you know will fail, and/or
    - that capture domain knowledge

refactor mercilessly

- **Let the code teach you**
  - Learn how to do it as you do it
  - Code to learn
    - first make it run
    - then make it right
    - last make it fast

pair program

- **Frequent pair sessions**
  - force you to learn, explain and justify
  - force you to share system knowledge

The XP Process Philosophy

- Give customers what they want
  - Delivered used software is where it’s at
    - customers want code, not design documents
    - customers want features that add value
  - Building what turns out not to be wanted
    - costs effort and opportunity
    - when in doubt, just wait
  - An XP Story is the right, not obligation,
    - to build a feature at some future time
- Used software never dies
  - successful software is simultaneously
    - in production, being evolved
    - in use, being maintained
  - (almost) nothing that dies comes back
    - and VCAP did, via XP

The XP Process

- **Story: some testable features**
  - written by customer, estimated by designers
  - not what designers commit to
- **Iteration: ~ 4 weeks' worth of stories**
  - collectively, estimate iteration and list tasks
  - individually, sign-up for and estimate task
    - write test case for task
    - (re)write code till it passes test
    - either write another test or move to another task
  - customer reviews result between iterations
    - can't change story within iteration (can raise bugs)
- **Release: a set of iterations**
  - that make business sense together