

There Must Be 50 Ways to Speed Up Software Development

George Dinwiddie
iDIA Computing, LLC
<http://idiacomputing.com/>

Copyright 2008
by George Dinwiddie
All rights reserved



The problem is all inside your head
She said to me
The answer is easy if you
Take it logically
I'd like to help you in your struggle
To be free
There must be fifty ways...

-- Paul Simon
50 Ways to Leave Your Lover



Type Faster

Work Harder

Work Longer Hours

Hire More People

(The Usual Suspects)



Have a Clear Goal

“Our project will leverage a new paradigm of synergy with our customers to enable leading-edge performance with a best-of-breed strategic enterprise solution.”

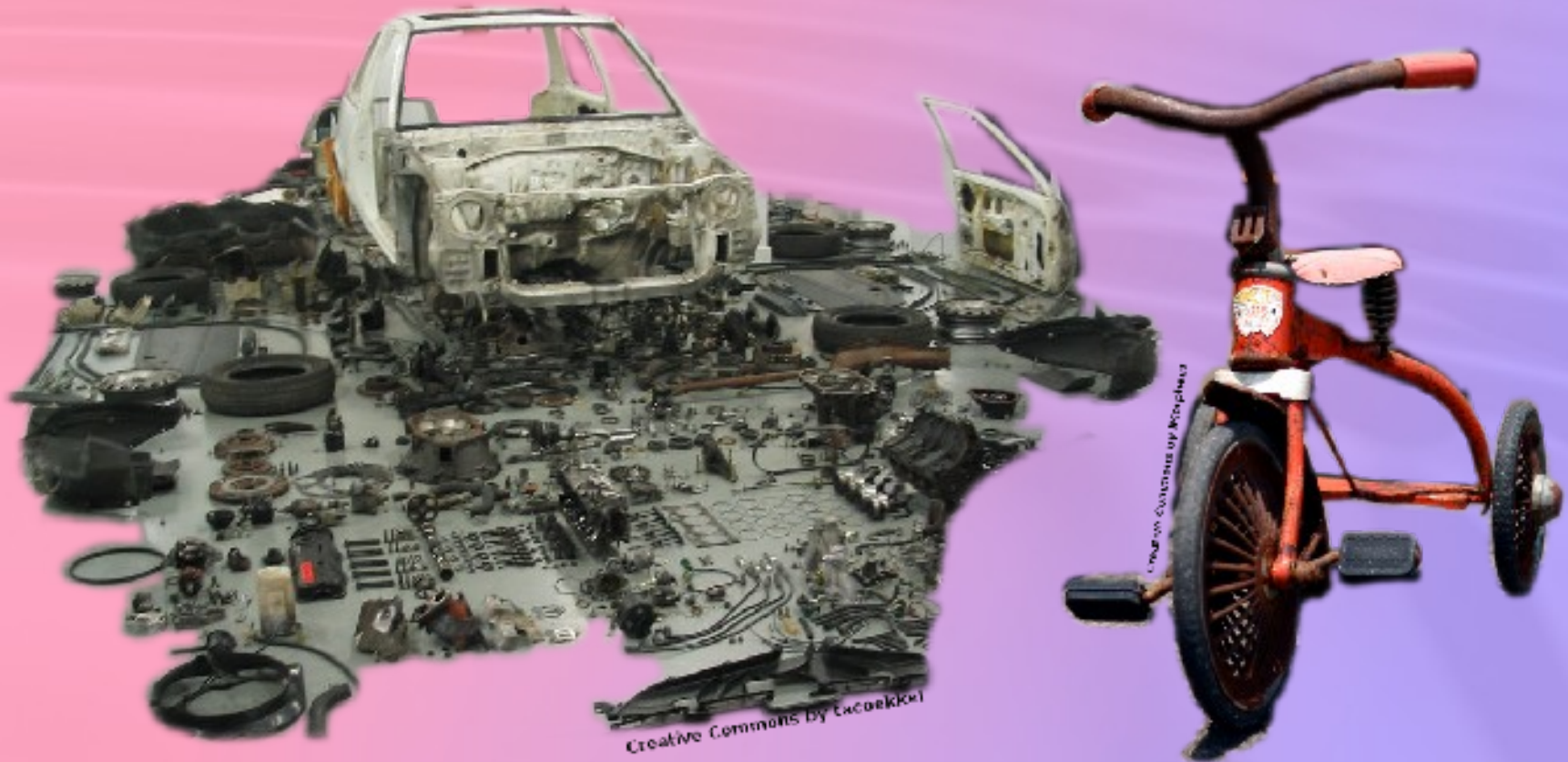


Keep Learning Throughout the Project

- Don't expect to know everything at the beginning.
- Make use of what you learn to improve the way you work.



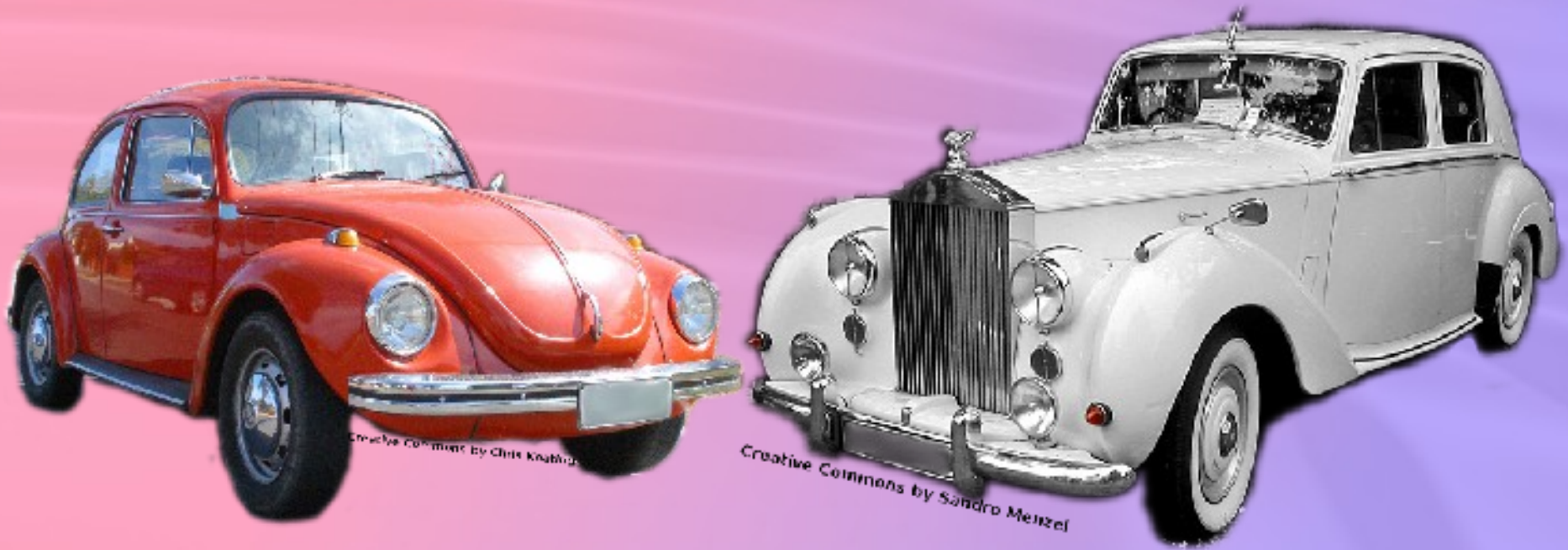
Build Complete Working Vertical Slices



It may not do much, but it works



Keep the system working as you add new functionality



Allows stopping when sufficient value has been achieved



Build Complete Working Vertical Slices



Avoid hidden undoneness

“We're 90% done; we just have to do the 10% that takes 90% of the time.”



Have a Shared Goal

- Understood by All
- Agreed to by All
- *Requires listening to contributions of All*



Creative Commons by Whiteller



Have Information Radiators

- that keep people aligned and working in the same direction



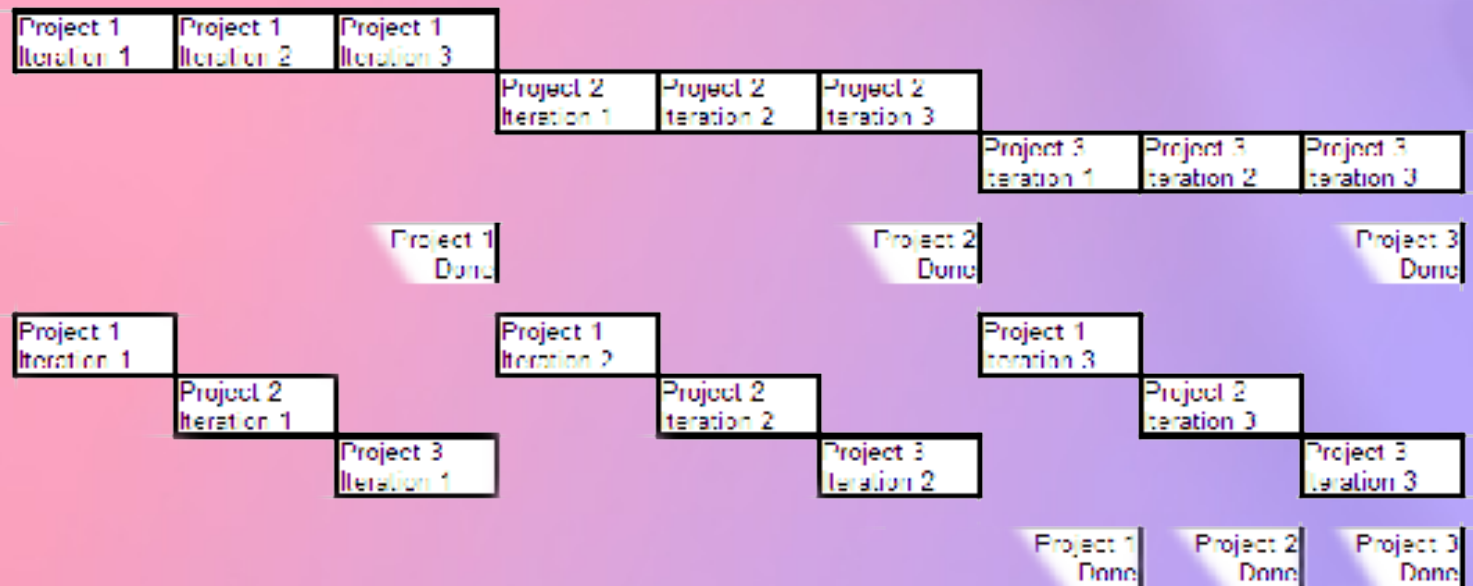
Use Short Iterations

- It's hard to write all the requirements up front
- AND it takes a lot of time
- Get started *now* with the Big Picture
- Refine the requirements as you go



Do One Project At A Time

- Multi-tasking takes extra time for context switching.
- Even if switching time is ignored, multitasking doesn't get anything done quicker—it just delays the earlier projects.



Don't build features that no one wants or will use

- Choose the right stories
- Test the choice early & often



Involve Testers at the Start

- *“How will we know that this feature works as intended?”*
- It's hard to get DONE without knowing this.

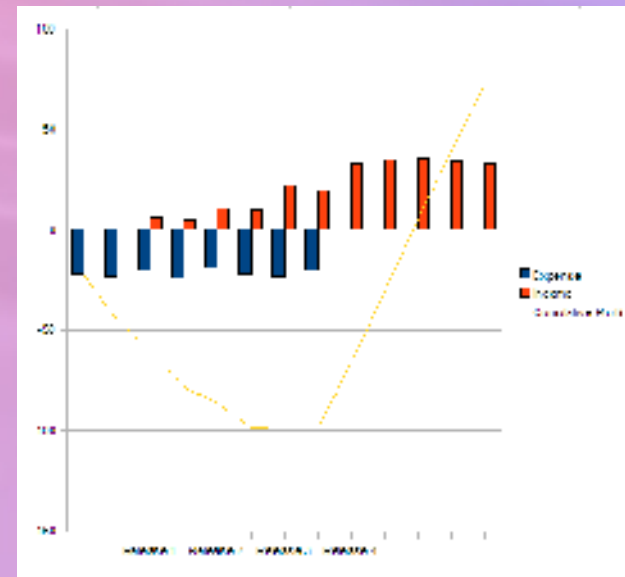
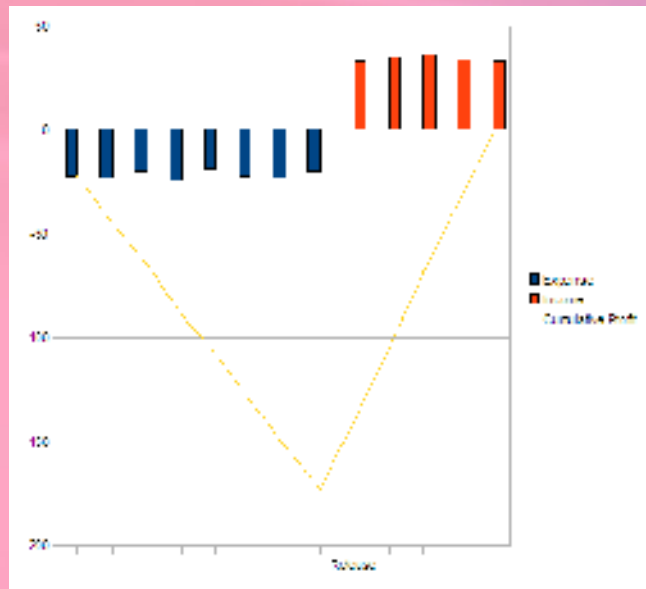


Avoid Gold-Plating

- No extra features beyond what the business wants
 - *But if you have a good idea, bring it up and consider it from the business perspective.*



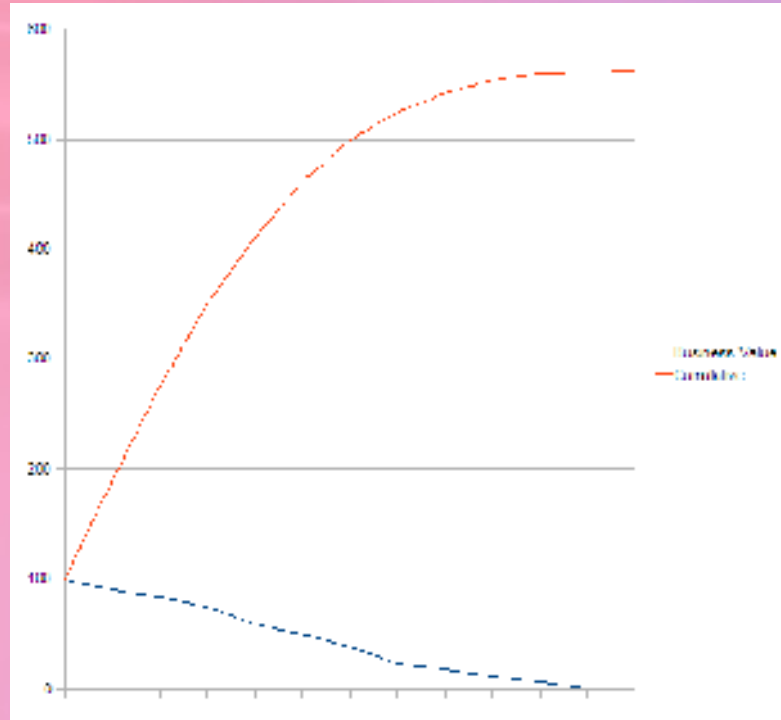
Early Delivery of Partial Functionality



- Earns early Return on Investment
- Enabled by working in vertical slices



Put Highest Value First



- You may find that the low-value items have little or no ROI, and can be dropped



Optimize Globally, not Locally



“Where is the bottleneck?”



Slow Down to go Faster

- *“Haste makes waste.”*
- Get things right before trying to speed things up.



Slow Down to go Faster

- It's hard to improve when you're rushing.
 - “People under time pressure don't *think* faster.” --
Tim Lister
- Build slack into the plan.



Retrospectives

- Take time to look around
 - *Where are we?*
 - *Where are we headed?*
 - *What's working?*
 - *What needs work?*

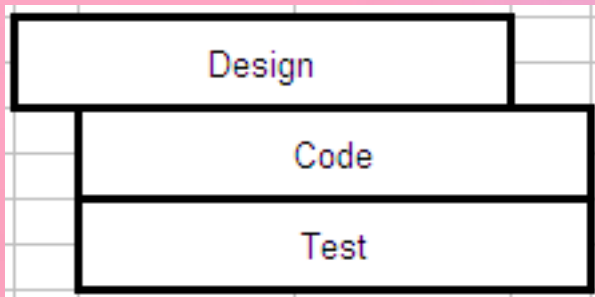
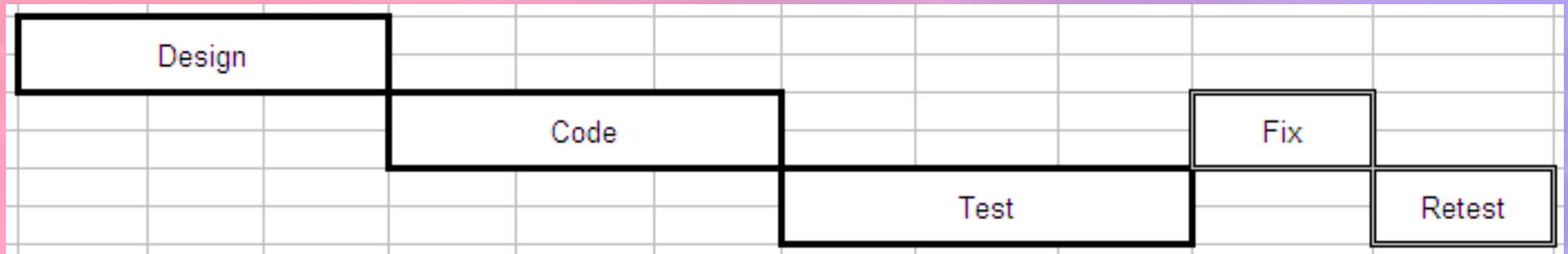


De-Phase the Process

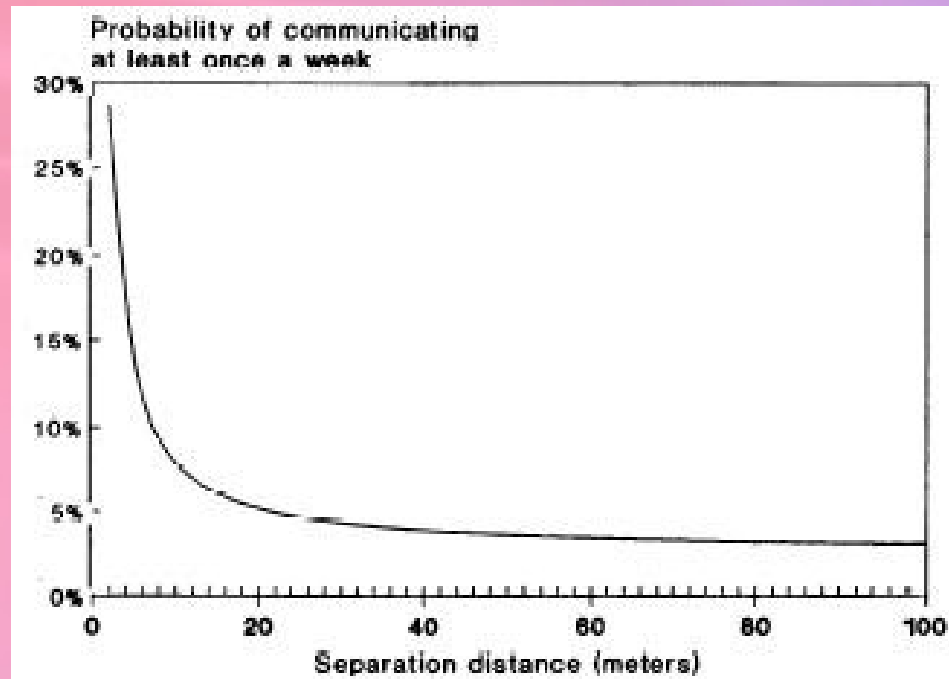
- Don't try to do all the design before coding
- Or to do all the coding before test
- The things you learn will inform future work
 - *“I'm glad we caught this problem early, before we built a lot more like this.”*



De-Phase the Process



Reduce Physical Distance Between People



Or they're unlikely to ask the questions they have

(chart from <http://www.npd-solutions.com/collocation.html>)



Use Direct Communication over Written Communication

- faster
- easier for most people
- allows for quick detection and correction of misunderstandings



Reduce Handoffs

- lost context
- imperfect communications
- It's better to work on things together



Work Together in One Room

- Team & Customer/Product Owner
- Osmotic communication
- Alignment of effort



Make Good Use of Meetings

- and Spend less time in them
- Use working meetings
 - not Status
 - not general communications



Use Short Iterations

- Make tangible progress
- Focus on getting discrete steps done



Use Short Iterations

- Give quick feedback to the business
- *“Does this do what you want?”*



Automate Regression Tests

- Run all of the tests, all of the time
- *Have you ever resisted making a change because it would mean retesting everything?*



Reduce Waiting and Delays

- Handoffs are a big source
- Look around and ask around
 - Delays are all over the place
 - and the all add up.



Use Small Stories

- Gives quick feedback to the developer
 - *“Does it work the way I intended it to work?”*



Use Small Stories

- and work to completion
- Partially done work is depreciating inventory



Automated User Acceptance Tests

- Let the developers know when they might have completed the user story as intended
- Unambiguous representation of the requirements
- *May not be comprehensive, though.*
 - *Use common sense*
 - *Exploratory testing*



Test Immediately

- It's not **DONE** until it's shown to work



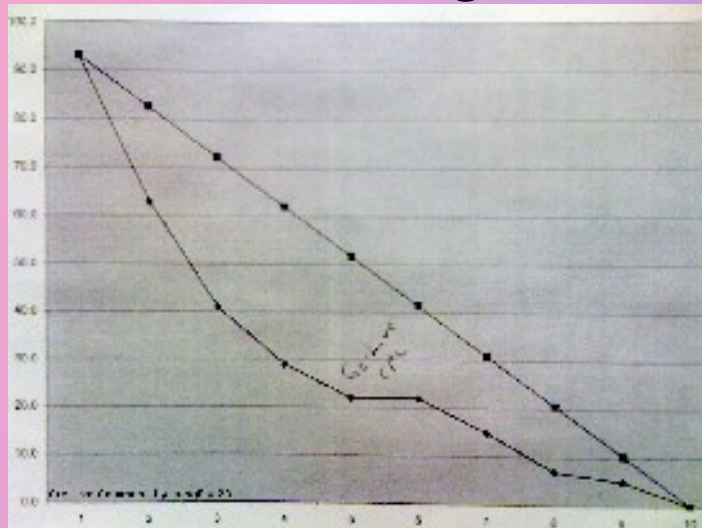
Use Small Stories

- A small risk of time and effort
 - if implemented wrong, or
 - if implementing the wrong thing



Use Information Radiators

- that keep people informed about progress and highlight trends
 - Burn charts
 - Regression test results
 - Whatever looks like it might be an issue *in your context*



Use Pair Programming

- to reduce incidents of programmers getting “stuck” on a problem



Creative Commons by Nic McPhee



Test Driven Development

- Heads off “Analysis Paralysis”
 - Express the next goal as a test
 - “What is the simplest thing that could possibly work”
 - Work in baby steps
- Get back in the “flow” quickly
 - Leave a failing test at the end of the day



Use Pair Programming

- to catch mistakes early
 - sometimes before they happen
 - always in time to improve the future



Creative Commons by Svein Halvor Halvorsen



Test Driven Development

- Checks each line as it goes in
- Lose little time if it's wrong
- Stay out of the debugger



Continuous Integration

- Checks each developer's work as it goes in
- Avoid unpleasant surprises (undone integration work) at the end of the project
- Best if done in small increments
- Includes smoke tests to detect problems immediately



Non-locking Version Control

- It's quicker to merge changes than to wait for access
- Conflicting changes should be few
 - Helped by working in small increments
 - Check in working code frequently
 - *If conflicts occur frequently, it may be telling you something about the code design or the process*



Use Promiscuous Pair Programming

- to spread beneficial knowledge



Creative Commons by Ade Oshineye



Shared Code Ownership

- Don't wait for “the expert” to be available
- When a change ripples through multiple places, take care of them all
 - without handoffs
 - without waiting
- *Enabled by and enables continuous learning*



Test Driven Development

- Let's you know when you've built enough to satisfy the requirement
- Avoid unnecessary complexity in the code



Refactoring

- Keep the code clean
- Lets you add new things later
 - without overbuilding today
- Enabled by “safety net” of unit tests



Avoid Technical “Hooks”

- Think about future needs
- But build what you need now
 - With a clean code base, you can add the hooks when you need them
- Evolve your frameworks from actual needs, not from speculation on future needs



Make Problems Visible

- Hidden problems don't get solved
- Hidden problems mean that you're further from the end than it looks
- Problem reports are a gift
 - Thank the bearer of bad news
 - Blaming sweeps problems under the rug



You can't always get what you want
You can't always get what you want
You can't always get what you want
But if you try sometimes you might find
You get what you need

-- Rolling Stones
You Can't Always Get What You Want



iDIA Computing, LLC George Dinwiddie

*Helping teams find
more effective
patterns of software
development.*

<http://idiacomputing.com/>

