Innovation

FOSTERING THROUGH THE AGILE METHOD

Who Am I?

- Former Technical Instructor for Fortune 10 companies
- Wrote first computer program in 1981 at age 11 (36 years, 23 prof)
- Presently Architect over new product development at a Medicare/Medicaid/ACA claims processing company

What is this about?

- This presentation is based on three lectures I gave up through 2007 consolidated into one much shorter presentation
  - “Why We Code” – Profiles of programmers
  - “The Joy of Not Coding” – Leveraging Design Patterns to write as little code as possible (i.e. The Best Code Is No Code)
  - “Fostering Innovation through Continuous Process Improvement”, now specifically “Fostering Innovation through the Agile Method”

Why Innovate?

Unless your product is “perfect”, or has no competitors… you will need to innovate… or die.

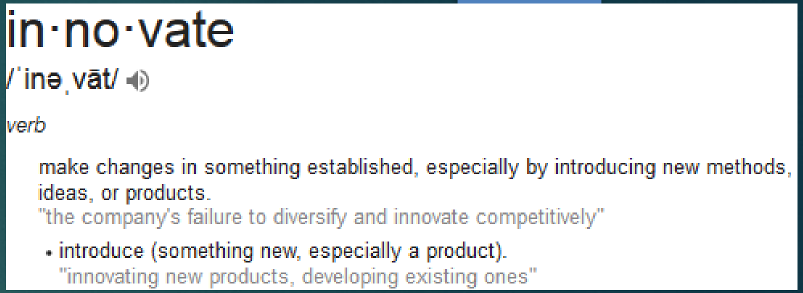
Here we go down the rabbit hole…

What’s the difference between imitation, and innovation?

An innovation in your product might just be an imitation of a feature in a competitors product

Frederich Hegel created the idea of a three part process to innovative thought, that of “Thesis”, “Antithesis”, “Sythesis” (a.k.a. “The Dialectic Triad”).

Basically there’s what you know (thesis), what you don’t know and that you are presented with (antithesis), and if you can meld thesis and antithesis together, you get synthesis, which is something new (at least to you)



in·no·vate

/ˈinəˌvāt/

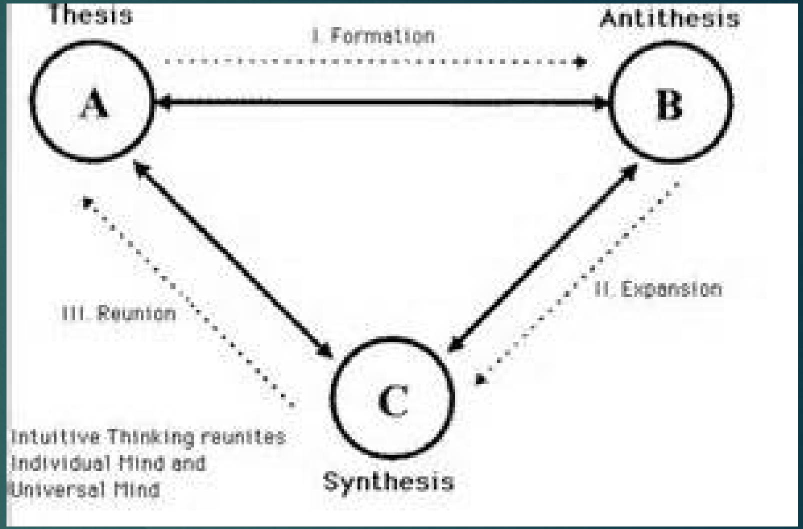
verb

make changes in something established, especially by introducing new methods, ideas, or products.

"the company's failure to diversify and innovate competitively"

- introduce (something new, especially a product).

  "innovating new products, developing existing ones"

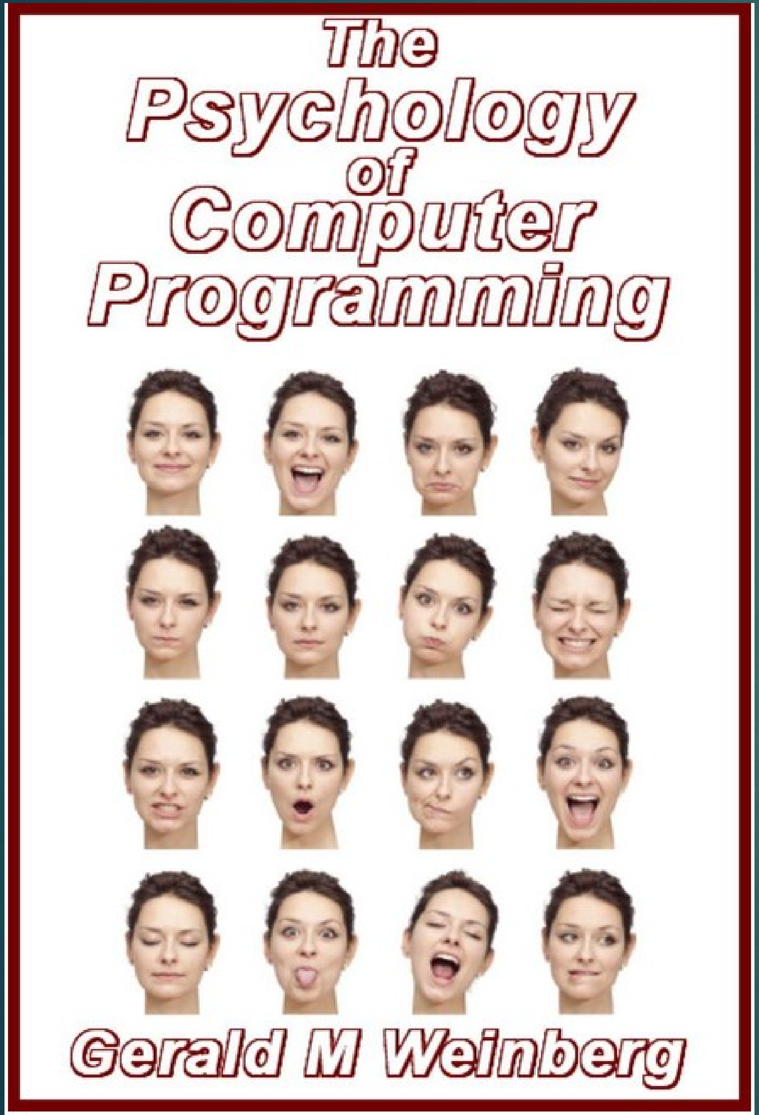


Influences

Nothing happens in a vacuum… These sources played an oversize role in this presentation…

- Books
  - “The Psychology of Computer Programming” – Gerald M. Weinberg
  - “The Pragmatic Programmer” – Andrew Hunt & David Thomas
  - “Design Patterns” – Multiple Authors
  - “Anti-Patterns” – Multiple Authors
- Heroes of Computer Science
  - Edsgar Djikstra
  - Grace Hopper
  - Doug Crockford
  - Martin Fowler

Egoless Programming



Arguably one of the most influential books of all time

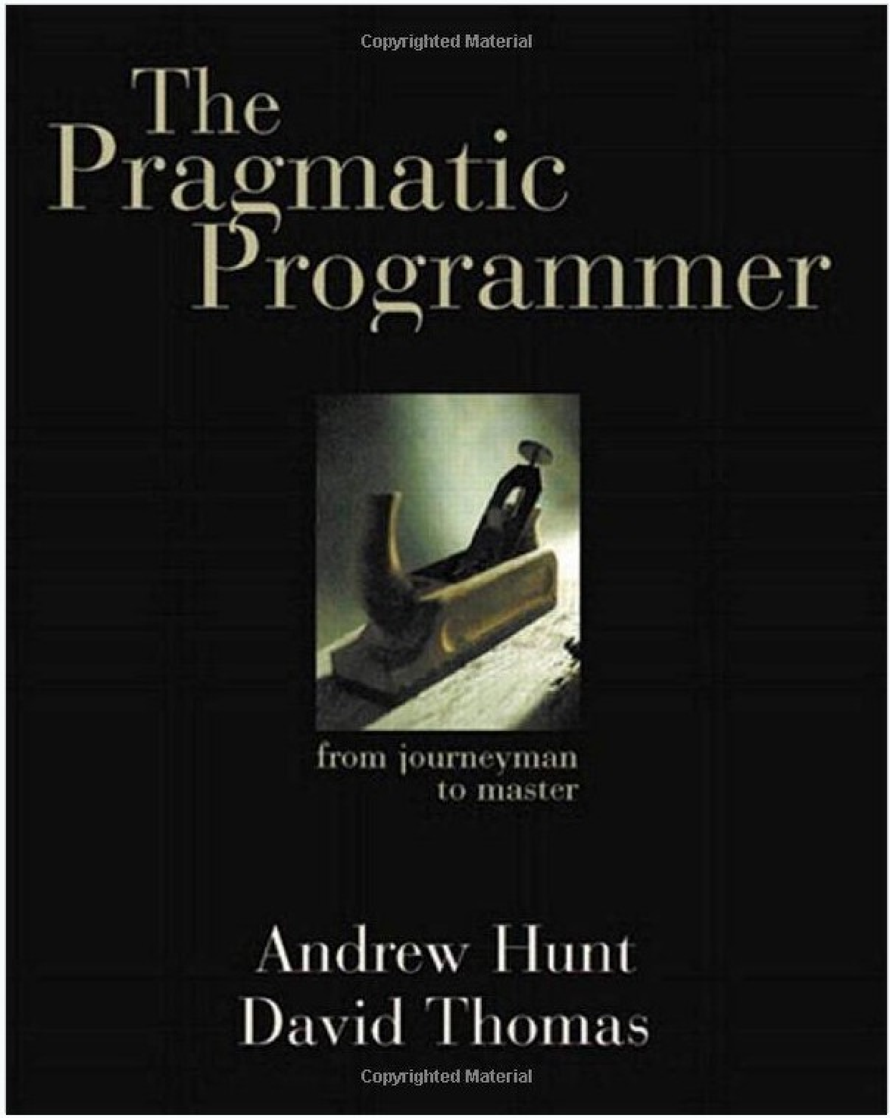
Longest book about the CS field continually in print (1971)

Revised 1996, the “Silver Anniversary Edition”

The 10 Commandments of Egoless Programming

1. **Understand and accept that you will make mistakes.**
2. **You are not your code.**
3. **No matter how much "karate" you know, someone else will always know more.**
4. **Don't rewrite code without consultation.**
5. **Treat people who know less than you with respect, deference, and patience.**
6. **The only constant in the world is change.**
7. **The only true authority stems from knowledge, not from position.**
8. **Fight for what you believe, but gracefully accept defeat.**
9. **Don't be "the guy in the room.“**
10. **Critique code instead of people – be kind to the coder, not to the code.**

Software Craftsmanship



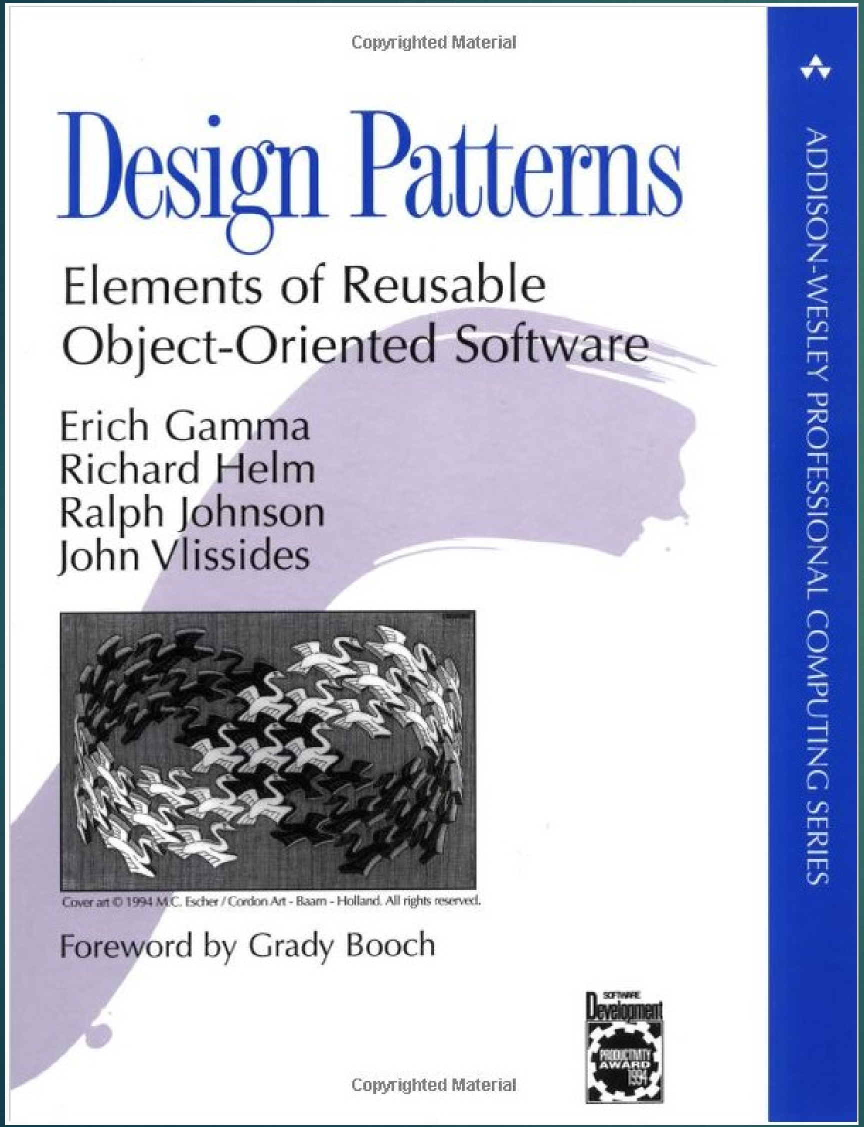
Began the “Pragmatic” Movement

The Software Craftsmanship Manifesto

1) **Care About Your Craft**
Why spend your life developing software unless you care about doing it well?
4) **Don't Live with Broken Windows**
Fix bad designs, wrong decisions, and poor code when you see them.
8) **Invest Regularly in Your Knowledge Portfolio**
Make learning a habit.
13) **Eliminate Effects Between Unrelated Things**
Design components that are self-contained. independent, and have a single, well-defined purpose.
24) **Fix the Problem, Not the Blame**
It doesn't really matter whether the bug is your fault or someone else's – ⇝it is still your problem, and it still needs to be fixed.
29) **Write Code That Writes Code**
Code generators increase your productivity and help avoid duplication.
31) **Design with Contracts**
Use contracts to document and verify that code does no more and no less than it claims to do.
38) **Put Abstractions in Code, Details in Metadata**
Program for the general case, and put the specifics outside the compiled code base.
47) **Refactor Early, Refactor Often**
Just as you might weed and rearrange a garden, rewrite, rework, and re-architect code when it needs it. Fix the root of the problem.
68) **Build Documentation In, Don't Bolt It On**
Documentation created separately from code is less likely to be correct and up to date.

AND THERE ARE DOZENS MORE!

Programmers Tool Chest



Copyrighted Material

Design Patterns

Elements of Reusable Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides

Cover art © 1994 M.C. Escher / Cordon Art - Baarn - Holland. All rights reserved.

Foreword by Grady Booch

ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

Software Development Productivity Award 1994

Copyrighted Material

Common solutions to common problems

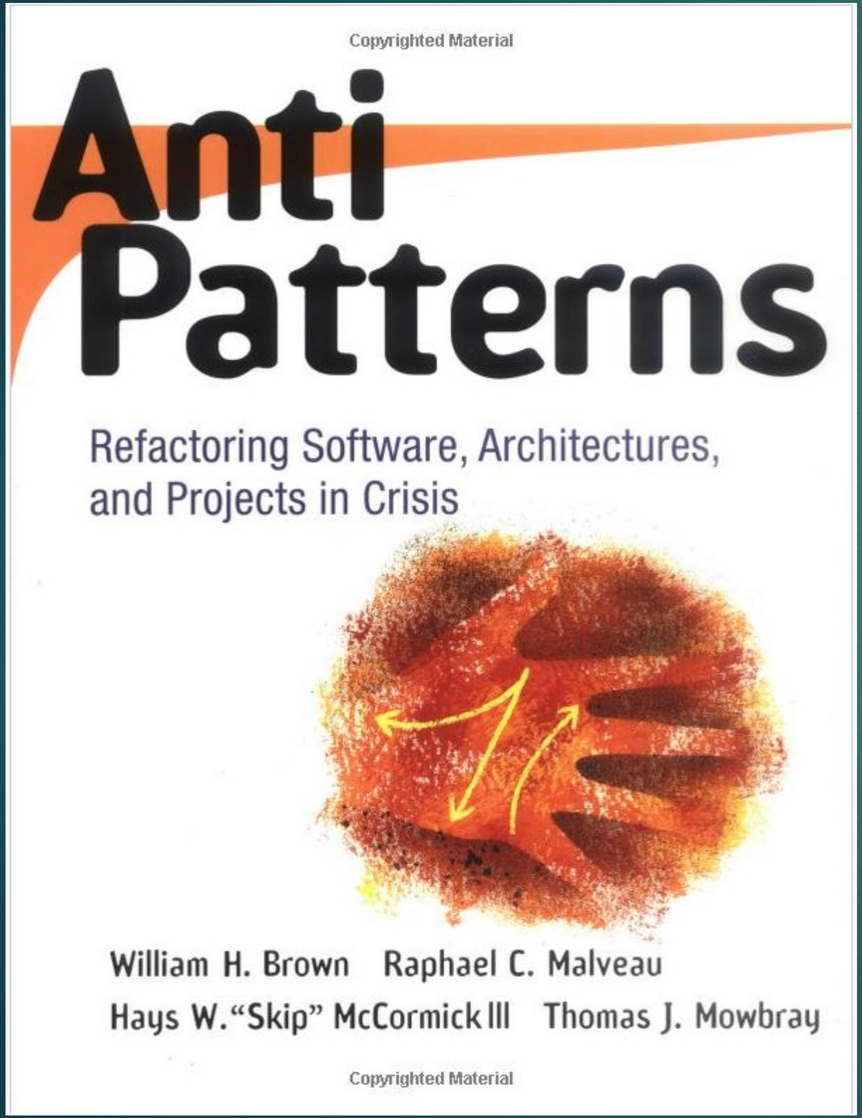
Concept originally imported from Structural Architecture

An emphasis on identifying common challenges that developers face

To be a Design Pattern, the problem must have the following characteristics

- A Definition
- A Name
- A Categorization
- An Abstracted Solution

What to avoid



If "Design Patterns" are what you do to solve problems, Anti-patterns tells you what to avoid. If you find yourself in these situations, it is time for drastic change

There are 23 primary Design Patterns, but over 100 anti-patterns. In a sense, it is more important to recognize what not to do, than what to do

Anti-patterns are broken out into Social, Business, Project Management, Software Engineering, Programming, Methodology, and Configuration Management… basically, if it's something people do, there are anti-patterns that you should avoid while doing whatever it is

Samples:

Organizational:
"Escalation of Commitment" -- Failing to revoke a decision when it proves wrong

Project Management:
"Cadillac approach" -- Wrongly assuming that the most expensive tools, staff, and other resources are the best guarantee of a project's success
Software Design:
"Big ball of mud" -- A system with no recognizable structure

Programming:
"God object" -- Concentrating too many functions in a single part of the design (class)

Methodology:
"Tester Driven Development" -- Software projects in which new requirements are specified in bug reports

It is good to have Heroes…

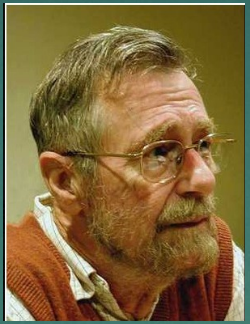
Gil Gerard
(Buck Rogers)

Kermit (Muppets)

Superman

When I wrote my first program at age 11, these were my heroes…

Today, these are my Heroes…



Edsgar Djikstra

Grace Hopper

Doug Crockford

Martin Fowler

Each of these people brought a qualitative assessment of programs, programmers, and the processes that govern them

Quotes

- Edsgar Djikstra
  - *Computer Science is no more about computers than astronomy is about telescopes.*
  - *The quality of a programmer is inversely proportional to their use of the "GOTO" statement.*
  - *The competent programmer is fully aware of the limited size of his own skull. He therefore approaches his task with full humility, and avoids clever tricks like the plague.*
  - *Program testing can be used to show the presence of bugs, but never to show their absence!*
  - *When you program you must be careful because the program also programs you!*
- Grace Hopper
  - *No computer is ever going to ask a new, reasonable question. It takes trained people to do that.*
  - *The worst reason to continue to do something is "Because we've always done it that way".*
  - *It is often easier to ask for forgiveness than to ask for permission.*
- Doug Crockford
  - *The long-term value of software to an organization is in direct proportion to the quality of the codebase*
  - *The best thing about JavaScript is its implementation of functions. It got almost everything right. But, as you should expect with JavaScript, it didn't get everything right*
  - *Obsolete comments are worse than no comments.*

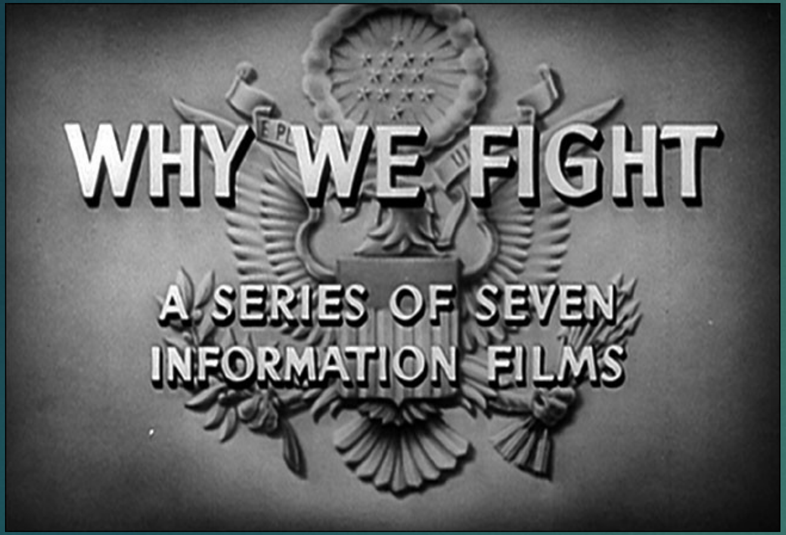
A brief summary of what is to come

“Why we code” – Machines don’t innovate, people do. It is somewhat ironic that to maximize your innovative potential, you must assess and manage your people appropriately. Recognize skill and encourage growth in your staff. Build relationships and not fixed roles

“The joy of not coding” – Innovation is hampered by bad code or systems that require excessive maintenance. Through the leveraging of technique and design patterns, you can create applications that require less refactoring and maintenance, thus allowing for more innovation

“Fostering Innovation” – You might be an agile shop, but are you allowing for the members of your team to grow to become incubators of innovation? Let’s pull it all together to show how through the Agile process, you can get the most out of the resources you have

Why we code…



A lecture loosely based on the purpose behind the WWII information movies that attempted to explain how the U.S. ended up in a world wide war

Attempts to describe the motivations and mindset of the developer to the manager

Helps bridge the gap between the developer's mindset and the mindset of the manager

Our goal is to maximize the individual developer's potential

"Grow your developers before you grow your development team" – Rick Myers

Why We Code: The developer

- When asked why a developer chose their career, various answers often are often given
    - “I liked computers when I was in high school”
    - “I played video games, and wanted to know how they worked”
    - “I wanted a stable job that was in demand”
    - “I thought it was a good way to make money”
    - “I liked solving problems”

Let’s focus on the last point…

Problem Solving: Instant Gratification

- Most developers enjoy solving problems
- They enjoy the feeling of accomplishment when something gets done
- There’s a certain “selfish” quality to gratification, particularly when something is “new” work
- They like it when people react positively to the work they have done
- The developer is different from other employees because in their role a certain amount of innovation is expected
- A word of encouragement goes a long way, but don’t pile it on… words are cheap
- An **investment** by managers and team leaders goes much further

The Developer: Their own worst enemy

- As developers solve problems, they run the risk of their egos growing
- A developer with a giant ego is not only a problem to an organization, they are going to limit their own growth
- “The biggest impediment to achieving greatness is believing that one is already great”
- See “The Humble Programmer”, by Edsgar Djikstra
- See “Egoless Programming”
- When a developer’s ego exceeds their skill, they run the risk of encountering “Cognitive Dissonance”, and provoking a crisis in themselves and the team
- Managing egos within a team is as much the responsibility of the manager as is product delivery

Assessing Developers Personality

- A developer is not measured by physical strength, running speed, or how high they can jump
- The developer's muscle is their brain
- To get a feel for a developer's potential, there are numerous psychological profile tools available
- Myers-Briggs Type Indicator (MBTI)
  - *S*J vs *N*P
  - *N*J vs *F*P
- Karen Horney's "Three categories of needs"
  - Compliant, Aggressive, Detached
- The Enneagram Spectrum

Assessing Developer Skills

The **Dunning–Kruger effect** is a cognitive bias in which relatively unskilled individuals suffer from illusory superiority, mistakenly assessing their ability to be much higher than it really is. Dunning and Kruger attributed this bias to a metacognitive inability of the unskilled to recognize their own ineptitude and evaluate their own ability accurately.

-- Wikipedia

Tips when assessing skills:

- Avoid "Gotcha" questions
- Never ask "on a scale from 1 to 10, how do you rate yourself on..."
- Avoid complicated/ridiculous code challenges
- General programming challenges are OK
- Assess personality along with skills… skills are easier to develop than personalities
- Conversation about technologies can reveal competency

It is OK to be incompetent at something… it is **not** OK to stay incompetent

Lazy Programmers are the Best

- According to Larry Wall[1], the original author of the Perl programming language, there are **three great virtues of a programmer**; Laziness, Impatience and Hubris
  - **Laziness**: The quality that makes you go to great effort to reduce overall energy expenditure. It makes you write labor-saving programs that other people will find useful and document what you wrote so you don't have to answer so many questions about it.
  - **Impatience**: The anger you feel when the computer is being lazy. This makes you write programs that don't just react to your needs, but actually anticipate them. Or at least pretend to.
  - **Hubris**: The quality that makes you write (and maintain) programs that other people won't want to say bad things about.

East Coast vs West Coast

What kind of an environment is right for you?

East Coast (Frenetic)	West Coast (Innovative)
Compliance	Consensus
Enforcement	Empowerment
Process	Relationships
Standards	Flexibility
Values obedience	Values opinions
Stability & Continuation	Change and renewal
Defined Roles & Hierarchy	Egalitarian
And the list goes on…	

In an East Coast environment, activity is often mistaken for progress

See "The Five Factors" for more on finding workplace bliss

The Joy Of Not Coding

- No matter how many “ninja rock-stars” are on your team, that might not be enough to get over…
  - Poorly written code
  - Poorly constructed systems
  - Inconsistencies
  - Too much Technical Debt
  - Poor/No documentation
  - Over-reliance on one or a few team members

The best code is no code at all! (🡐 Google that…)

Enemy #1: Software Entropy

A work on software engineering by Ivar Jacobson et al. describes **software entropy** as follows:

> The second law of thermodynamics, in principle, states that a closed system's disorder cannot be reduced, it can only remain unchanged or increased. A measure of this disorder is entropy. This law also seems plausible for software systems; as a system is modified, its disorder, or entropy, always increases. This is known as **software entropy**.

1. A computer program that is used will be modified
2. When a program is modified, its complexity will increase, provided that one does not actively work against this.

Enemy #2: Technical Debt

Technical debt (also known as **design debt**[1] or **code debt**) is "a concept in programming that reflects the extra development work that arises when code that is easy to implement in the short run is used instead of applying the best overall solution[2]".

For a great primer, google “Technical Debt 101”…

Common causes of technical debt include (a combination of):

- **Business pressures**, where the business considers getting something released sooner before all of the necessary changes are complete, builds up technical debt comprising those uncompleted changes.
- **Lack of process or understanding**, where businesses are blind to the concept of technical debt, and make decisions without considering the implications.
- **Lack of building loosely coupled components**, where functions are not modular, the software is not flexible enough to adapt to changes in business needs.
- **Lack of a test suite**, which encourages quick and risky band-aids to fix bugs.
- **Lack of documentation**, where code is created without necessary supporting documentation. That work to create the supporting documentation represents a debt that must be paid.
- **Lack of collaboration**, where knowledge isn't shared around the organization and business efficiency suffers, or junior developers are not properly mentored.
- **Parallel development** at the same time on two or more branches can cause the buildup of technical debt because of the work that will eventually be required to merge the changes into a single source base. The more changes that are done in isolation, the more debt that is piled up.
- **Delayed refactoring** – As the requirements for a project evolve, it may become clear that parts of the code have become unwieldy and must be refactored in order to support future requirements. The longer that refactoring is delayed, and the more code is written to use the current form, the more debt that piles up that must be paid at the time the refactoring is finally done.
- **Lack of alignment to standards**, where industry standard features, frameworks, technologies are ignored. Eventually, integration with standards will come, doing sooner will cost less (similar to 'delayed refactoring').
- **Lack of knowledge**, when the developer simply doesn't know how to write elegant code.
- **Lack of ownership**, when outsourced software efforts result in in-house engineering being required to refactor or rewrite outsourced code.
- **Poor technological leadership** where poorly thought out commands handed down the chain of command increases the technical debt rather than reduce it
- **Last minute specification changes**; these have potential to percolate throughout a project but no time or budget to see them through with documentation and checks
- **Scope Doping**, unmanaged scope reduction or shedding of scope by project managers, or indeed technical staff. Subtly different to **Business pressures** in that, while business pressures may be a cause, Scope Doping will also cause business pressure due to unmanaged work scope that has been shed.

What’s that smell?

Know when to think about Refactoring.

According to Martin Fowler, "a **code smell** is a surface indication that usually corresponds to a deeper problem in the system".

Application-level smells:

- *Duplicated code*: identical or very similar code exists in more than one location.
- *Contrived complexity*: forced usage of overcomplicated design patterns where simpler design would suffice.

Class-level smells:

- *Large class*: a class that has grown too large. See God object.
- *Feature envy*: a class that uses methods of another class excessively.
- *Inappropriate intimacy*: a class that has dependencies on implementation details of another class.
- *Refused bequest*: a class that overrides a method of a base class in such a way that the contract of the base class is not honored by the derived class. See Liskov substitution principle.
- *Lazy class / Freeloader*: a class that does too little.
- *Excessive use of literals*: these should be coded as named constants, to improve readability and to avoid programming errors. Additionally, literals can and should be externalized into resource files/scripts where possible, to facilitate localization of software if it is intended to be deployed in different regions.
- *Cyclomatic complexity*: too many branches or loops; this may indicate a function needs to be broken up into smaller functions, or that it has potential for simplification.
- *Downcasting*: a type cast which breaks the abstraction model; the abstraction may have to be refactored or eliminated.[7]
- *Orphan Variable or Constant class*: a class that typically has a collection of constants which belong elsewhere (typically a problem when using a Constants class) where those constants should be owned by one of the other member classes.

Method-level smells:

- *Too many parameters*: a long list of parameters is hard to read, and makes calling and testing the function complicated. It may indicate that the purpose of the function is ill-conceived and that the code should be refactored so responsibility is assigned in a more clean-cut way.
- *Long method*: a method, function, or procedure that has grown too large.
- *Excessively long identifiers*: in particular, the use of naming conventions to provide disambiguation that should be implicit in the software architecture.
- *Excessively short identifiers*: the name of a variable should reflect its function unless the function is obvious.
- *Excessive return of data*: a function or method that returns more than what each of its callers needs.

Know your “friends”

It is possible to write code that is “resistant” to Software Entropy

These three approaches can help stave off Software Entropy

- “Separation of Concerns”
- “Convention over Configuration”
- “Inversion of Control”

Managing “Technical Debt” is something that needs to be done at the managerial level

Separation of Concerns (SoC)

In computer science, **separation of concerns** (SoC) is a design principle for **separating** a computer program into distinct sections, such that each section addresses a separate **concern**. A **concern** is a set of information that affects the code of a computer program – Wikipedia

SoC applies at the macro level and at the micro level. MVC is an example of SoC at the macro level

At the micro level, make sure that each method does only one thing, and that it is the "canonical" source for whatever that is. Basically, it does only one thing, and it's the only thing in your system that does it. Do not allow for duplication!

	Model	**View**	**Controller**
Java/Struts/Webwork	Java POJO	JSP, Freemarker, HTML	XML
PHP/Symfony/Zend/YII	PHP	Twig, Smarty, HTML	PHP

Use Database Access Objects

Do not use in-line queries

You should avoid mixing model logic with persistence logic

In computer software, a **data access object** (DAO) is an **object** that provides an abstract interface to some type of **database** or other persistence mechanism. By mapping application calls to the persistence layer, DAO provide some specific **data** operations without exposing details of the **database**.

It can also provide some low-level documentation value

Composition over Inheritance

Classes should achieve polymorphic behavior and code reuse by their composition (by containing instances of other classes that implement the desired functionality) rather than inheritance from a base or parent class

Inheritance

```
class Super {

    public function __construct() {
    }

    public function method_1() {
    }

    public function method_2() {
    }

    public function method_3() {
    }
}

class Child extends Super {
    public function method_4() {
    }
}

$child = new Child();
$child->method_1();
$child->method_4();
```

Composition

```
class Super {

    public function method_1() {
    }

    public function method_2() {
    }

    public function method_3() {
    }
}

class Child {
    public $methods = false;

    public function load() {
        $this->methods = new Super();
    }

    public function method_4() {
    }
}

$child = new Child();
$child->load();
$child->methods->method_1();
```

Convention over Configuration

Is a design paradigm that attempts to decrease the number of decisions that
a developer using the framework is required to make without necessarily losing flexibility.

A developer only needs to specify unconventional aspects of the application. For example, if there is a class Sales in the model, the corresponding table in the database is called "sales" by default. This is an example of a naming convention

We will take a look at a directory/filename convention in a few slides

Sample #1: Getters/Setters

The Setter/Getter convention (borrowed from Java) is one of the most common conventions, and powerful. The downside is that your classes can be cluttered up with trivial “accessors/mutators”. Using PHP Magic Methods, we can solve this in a clean and elegant way, meaning less actual code needs to be written.

```
public function __get($name) {
       $retval = null;
             if (isset($this->_data[$name])) {
          $retval = $this->_data[$name];
       }
       return $retval;
}

public function __set($name,$value)   {
       $this->_data[$name] = $value;
       return $this;
}

public function __call($name, $arguments)  {
       $token = substr($name,3);
       $token{0} = strtolower($token{0});
       if (substr($name,0,3)=='set') {
           return $this->__set($token,$arguments[0]);
       } elseif (substr($name,0,3)=='get') {
           return $this->__get($token);
       }
}
```

Sample #2: Auto CRUD

The JavaScript frameworks pioneered a technique for managing basic CRUD actions from JavaScript, even though the actual persistence tier exists on the server. To do so, they established a ***convention*** where by the shape of the URI, and the method used to send/request data, actions are performed on the server on your behalf.

In this way, those who work on the front end can perform the majority of common tasks with no involvement from the back end developers

URI: */namespace/table/id**Example**: */acme/user/37*

Method	Database Action
PUT	Create/Add (will get a new ID in the process)
GET	Read/Fetch (can return single row or cursor)
POST	Update/Save
DELETE	Delete

Sample #3: Factory/Files

A “Factory” is a fundamental Design Pattern. The factory takes in a resource and outputs a product. In our case, the resource is the criteria for instantiation, and the product is the instantiated class

Use factories to give a chance to determine class instantiation at run time…

“Just say ‘No’ to ‘new’”….

Sample #3: Factory/Files (cont)

```
$customer = ‘ACME’;
$engine   = Factory::instantiate($customer.‘/billing/engine’);
```

File Structure:

```
/billing/Engine.php
/ACME/billing/Engine.php
```

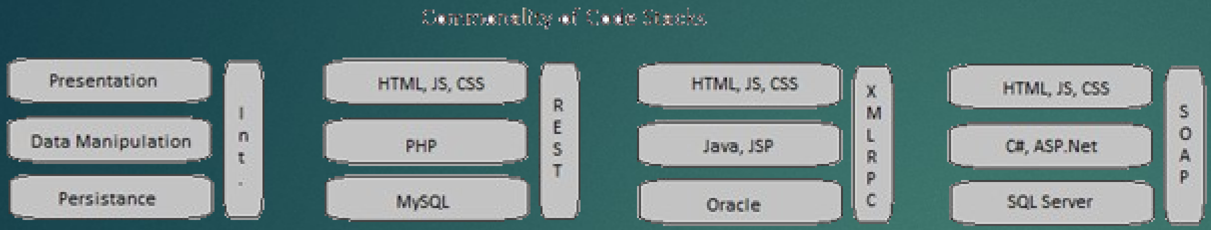
Factory Code:

```
public static function instantiate($criteria) {
          $comps    = explode(‘/’,$criteria);
          $class    = ucfirst($criteria[2]);
          $default  = ‘/’.$criteria[1].’/’..’.php’;
          $specific = ‘/’.$criteria[0].‘/’.$criteria[1].’/’.ucfirst($criteria[2]).’.php’;
          if (file_exists($specific) {
                    require_once($specific);
          } else {
                    require_once($default);
          }
          return $class();
}
```

Inversion of Control

- IoC is a design principle in which custom-written portions of a computer program receive the flow of control from a generic framework or other construct
- It often comprises these design patterns: Dependency Injection, Factory, Interpreter, Template, Strategy, and some include Observer
- Seeks to separate execution from implementation
- Is a natural progression from Separation of Concerns (SoC)

Technology: Not all code is created equal


Commonality of Code Stacks

Presentation
Data Manipulation
Persistance
Int.

HTML, JS, CSS
PHP
MySQL
REST

HTML, JS, CSS
Java, JSP
Oracle
XMLRPC

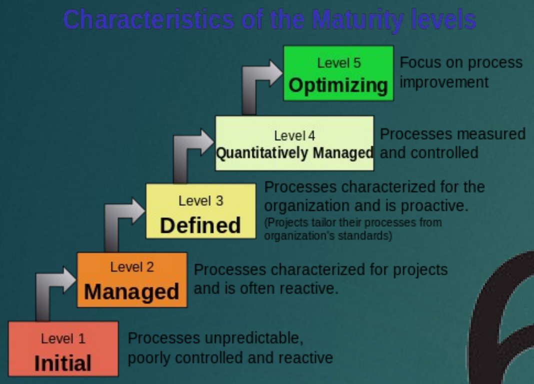
HTML, JS, CSS
C#, ASP.Net
SQL Server
SOAP

Generally speaking, all application stacks consist of some form of presentation tier, a data Manipulation tier, and a data storage tier, a.k.a. Persistence tier.

Not all code has the same cost in development and maintenance. Above you can see that the only consistent technology across these respective code stacks is the presentation tier, which is written in JS, CSS, and HTML. The more we can do in that tier, the less maintenance work we will incur later, and the less chance of impact on other unrelated components.

The important thing here is that we are trying to minimize the writing of custom code particularly on the backend (the model)

The Road to Agile



Characteristics of the Maturity levels
Level 5
Optimizing
Focus on process improvement
Level 4
Quantitatively Managed
Processes measured and controlled
Level 3
Defined
Processes characterized for the organization and is proactive.
(Projects tailor their processes from organization's standards)
Level 2
Managed
Processes characterized for projects and is often reactive.
Level 1
Initial
Processes unpredictable, poorly controlled and reactive

SIX SIGMA

A process that works at all levels of the business is not a "nice to have"… It is a requirement.

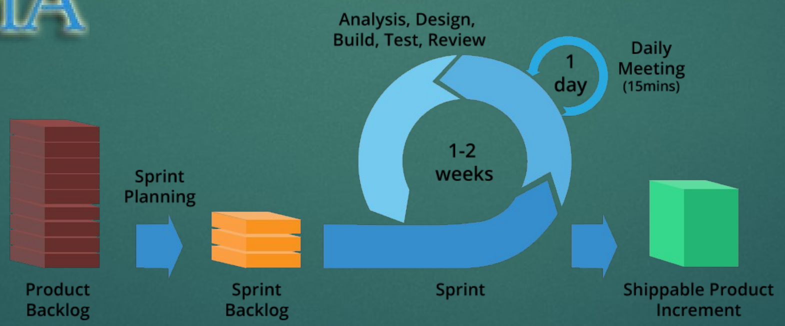
At different levels of the enterprise, different "paradigms" are in effect.

Developers should focus on agile, while the higher business functions should focus on SEI/CMM inspired processes.

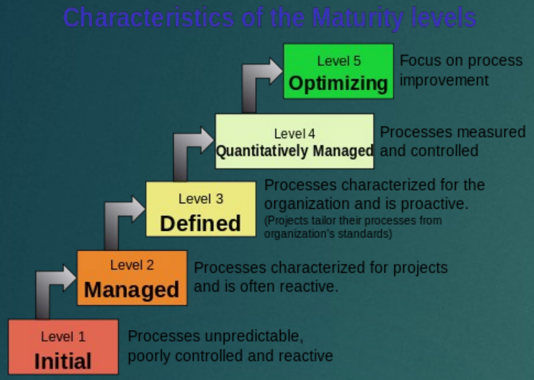
Six Sigma has some great ideas on how to produce better software through incentives

The process of innovation is not linear. A properly implemented process can be the catalyst to truly innovative software, but it is not easy.

Agile Software Development



SEI/CMMi



Characteristics of the Maturity levels

Level 5 **Optimizing** Focus on process improvement

Level 4 **Quantitatively Managed** Processes measured and controlled

Level 3 **Defined** Processes characterized for the organization and is proactive. (Projects tailor their processes from organization's standards)

Level 2 **Managed** Processes characterized for projects and is often reactive.

Level 1 **Initial** Processes unpredictable, poorly controlled and reactive

Initially released in 1993, and then re-assessed circa 1999 for "web"

More focused on qualitative results, though later levels also offer a certain amount of quantitative prediction… (how many widgets will we be able to make per year)

Level 1: Initial -- Reactive. Minimal planning. Success depends on *individual heroics*

Level 2: Managed – Basics of project management are in place, initial standards are created

Level 3: Defined – Standards are expanded to include other areas, including education, contract management, and other topics. At the end of Level 3, you are now producing quality

Level 4: Quantitatively Managed – Proper metrics gathering can now help us understand not only the quality of the products we are producing, but how many we can produce

Level 5: Optimizing – continuous improvement… improving a well-oiled machine

Process: Six Sigma

Define
Measure
Analyze
Improve
Control

An approach adapted to IT from manufacturing

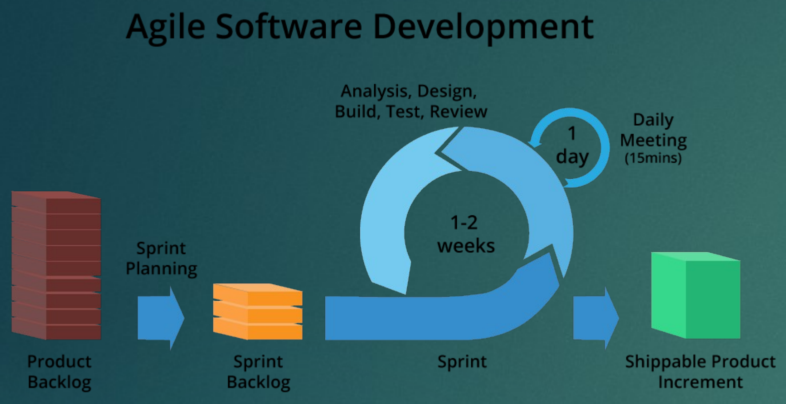
Intended to lower the number of defects in code

Very focused on “quantifying” the development process

Summed up by what Sun Tzu said: “Know Thyself”

Has some good ideas for incentivizing software developers to be more thorough in testing and
Producing code with fewer defects, however, a code base can be defect free and still be wrong,
So ultimately six sigma has dubious value vis-à-vis SEI/CMM and Agile .

Process: Agile


Agile Software Development
Analysis, Design, Build, Test, Review
1 day
Daily Meeting (15mins)
1-2 weeks
Sprint Planning
Product Backlog
Sprint Backlog
Sprint
Shippable Product Increment

This part of the process is developer focused

SEI/CMM can help you get the right work in the backlog

Six Sigma can help the developers improve their code over time

And a properly executed Agile/SCRUM env. can foster innovation

It is important to understand **where innovation doesn't come from**: It doesn't come from a blog, a book or an article. It doesn't come from inspirational quotes and stories. It doesn't come from LinkedIn Influencers or anyone you follow on Twitter. It doesn't come from motivational speakers. And it most certainly doesn't come from any kind of self-improvement or personal productivity

Innovation comes from inside you. Ideas, inspiration and innovation only seem to come from outside you, but they don't. They always come from inside you. The only exception is small teams...but only intimate groups in real time in the real world, never large-scale or online collaborations

Agile: Background

- First suggested in 1985
- Responds better to changing requirements
- Developer centered
- Especially good about innovation
- Holistic and Heuristic
- One of the iterative approaches

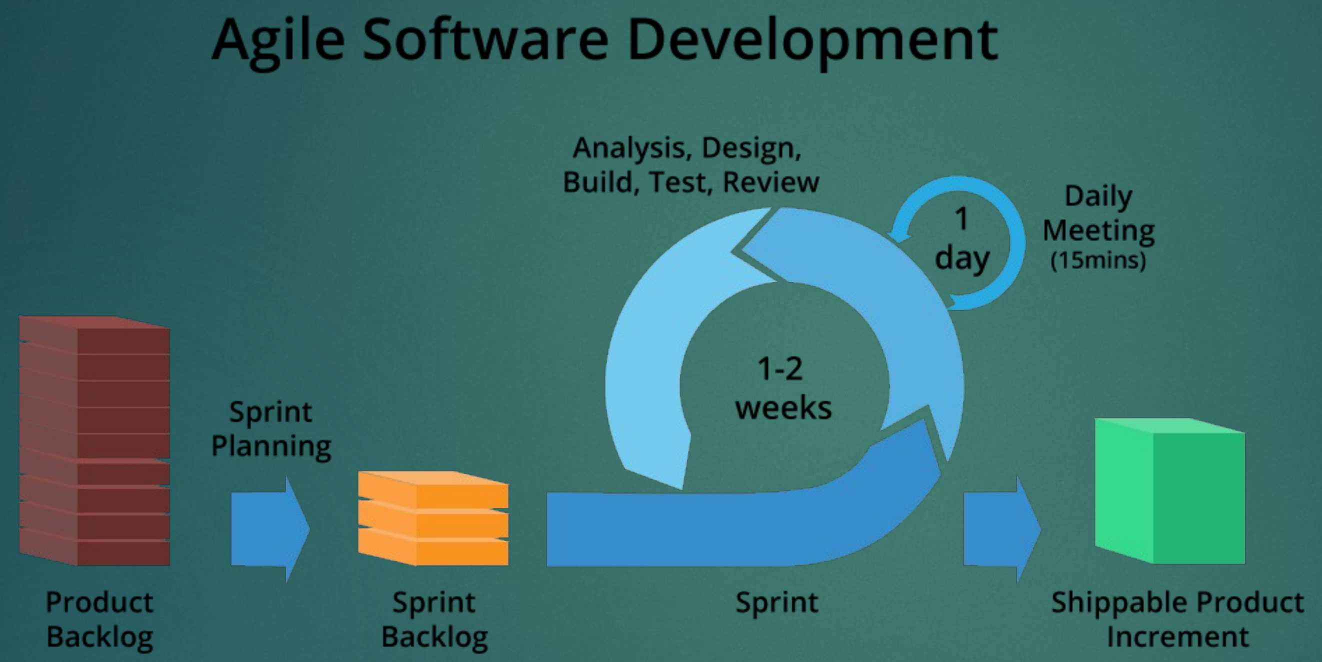
Agile Manifesto

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

12 principles behind the Agile Manifesto

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity--the art of maximizing the amount of work not done--is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behavior accordingly.

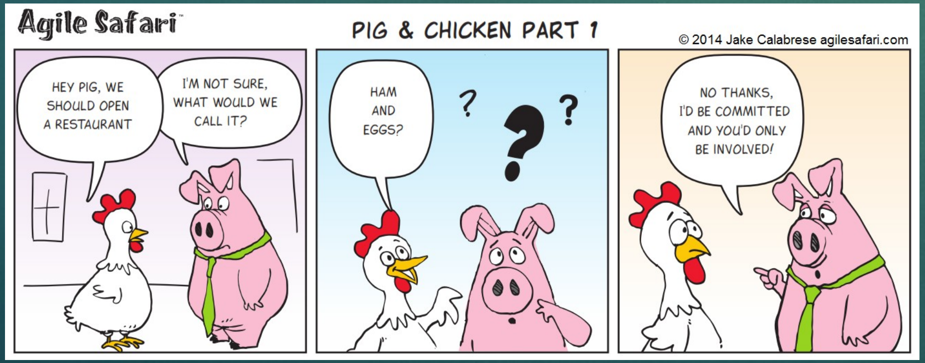
Let’s talk Agile…


Agile Software Development
Analysis, Design, Build, Test, Review
1 day
Daily Meeting (15mins)
1-2 weeks
Sprint Planning
Product Backlog
Sprint Backlog
Sprint
Shippable Product Increment

The first goal is to have a working product for review at the end of each “Sprint”

Define Who You Are

- Define your product…What is it? What does it do? Why is it different?
- Define who your customer(s) are
- Define roles
  - Product Owner
  - Stakeholders
  - Team Members
  - Scrum Master (If that's your thing)…
- Know the difference between a pig and a chicken!


© 2014 Jake Calabrese agilesafari.com

Agile Safari
PIG & CHICKEN PART 1
HEY PIG, WE SHOULD OPEN A RESTAURANT
I'M NOT SURE, WHAT WOULD WE CALL IT?
HAM AND EGGS?
? ? ?
NO THANKS, I'D BE COMMITTED AND YOU'D ONLY BE INVOLVED!

Agile Basics

- Agile teams are always an odd number, the ideal is 5 (you can't build a consensus with an even number)
- Sprints are measured in weeks, usually 2-3, though as along as 4. Avoid overly long sprints as planning beyond 3 weeks is very difficult
- Ideally, team members aren't "assigned" work but take it upon themselves (see "Self-Directed Work Teams")
- The Agile cycle involves Sprint Planning, Sprint, and Sprint review (we are going add a fourth…
- Sprint Planning reviews only what is in the backlog…
- Sprint Review looks at unfinished work and what we have learned
- All unfinished work is added back to the backlog

The Backlog

The Backlog contains the complete set of work needing to be done at that time

It contains

- Epics - Which contain multiple stories
- Stories - Which contain multiple tasks
- Task - Which are individual “things” needed to get done, and are “pointed”
- Spike Tasks – One-off tasks unrelated to a current story

All requirements changes or new work are added to the Backlog for review by the team

During Sprint Planning, tasks are taken off of the Backlog and added to the current Sprint queue (a.k.a. Sprint Backlog) to be worked on

Epics/Stories

When specifying Epics and Stories, please note role, the reason, and justification.

You can follow this approach:

As a [*role*], I would like a [*feature*], because of [*justification*].

Samples:

Epic:

As a Product Owner, I would like a web-based billing system, because it is easier to collect payments through the web.

Story:

As a user, I would like a search field, so that I may find related products faster

Agile Tasks

- The “Unit of Work” in Agile is the Task
- It is a granular piece of work. Stories contain 1 or more tasks
- Tasks get assigned a value based on effort of work
- The value normally is assigned an element from the Fibonacci series (1,2,3,5,8,13…)
- Anything 13 or over is normally elevated to a story to be tasked and pointed out separately
- POINT VALUE != HOURS!!!!
- The total of all tasks points add up to the effort level of a story, which adds up to the effort level of the epic.

Velocity

- Over several sprints, a mean value in terms of points will become apparent on a per developer basis
- This mean value is called the “Velocity”
- There’s a velocity value per developer, and per team
- Used to achieve effective and efficient sprint planning
- When planning, never try to “fill” velocity… always stop short

Sprint Planning

- Product owner and team members should be present, along with relevant stakeholders
- “Groom” the backlog, make sure everything is still relevant
- Mentally visualize where you want to be at the end of the sprint
- Talk about what it will take to be a “complete” sprint
- Cultivate team members, foster ownership of issues, then get out of the way
- Focus less on controlling, and more on empowering
- Pull relevant tasks from the backlog and add to current sprint queue
- Don’t fill to capacity, leave time for collaboration
- Velocity is hardly ever consistent

Sprint Review

- Open to anyone interested in reviewing
- Review is not to be used as a "signoff" for user acceptance
- Asks the question
  - What went well?
  - What could use some improvement?
  - What could be done to improve future sprints?
- Demonstrates the state of the current product, and the stories completed during the last sprint
- Just say no to PowerPoint (our goal is a working product)
- Product Owner should lead the review
- Questions from stakeholders can be taken about product set and future sprint activities

A tale of two Lumberjacks

Fostering Innovation

1) Work personal and professional growth into each sprint cycle

2) Manage relationships between developers, between developers and stakeholders, and between the team and the Product Owner

3) Always “Under-commit and over deliver!”, you may want to promise the world, but it is better to surprise with what got done, rather than what was missed

End Sprint with an "Un-Meeting"

- The Un-Meeting is not a meeting, it is a chance for the team to "sharpen their axes" and nip any personality conflicts at the bud
- Manage team members relationships by getting them to function together in a non-work setting (go play a game together!), have lunch, do something outside of work
- Typically lasting ½ day, and comprised of two different presentations
- The first presentation is given by a team member, and can cover any topic relevant to the team. This can be an opportunity for someone with specialized knowledge to share with the team ("flatten" the teams skills), or can be an early code review
- The second presentation can be videos or other sources, and may or may not be relevant to the existing product. You can look at technologies, techniques, competitors products, etc… anything that might stimulate creative thought

Self-Directed Work Teams

- Seriously, just google it… http://www.joe.org/joe/2004april/tt1.php
- Works extremely well with Agile/Scrum… they go hand-in-hand
- The developers direct the development… business job is to get the work into the queue, then handoff to developers
- Management’s job is to help the developers grow
- Success is no longer based on “Heroic Suicide”
- I mean it… just google it… :-D