Lean Software Development: One Step at a Time

Curt Hibbs
Boeing Defense, Space & Security

Steve Jewett
Boeing Research & Technology

Systems & Software Technology Conference 2010

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
APR 2010		**00-00-2010 to 00-00-2010**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Lean Software Development: One Step at a Time	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Boeing Defense, Space & Security,P. O. Box 516,St. Louis,MO,63166**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES
Presented at the 22nd Systems and Software Technology Conference (SSTC), 26-29 April 2010, Salt Lake City, UT. Sponsored in part by the USAF. U.S. Government or Federal Rights License

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**84**	

Agenda

Part 1 – Introduction to Lean Software Development

- The Problem with Software Development
  - Research Statistics
- Lean & Agile Software
  - Principles
  - Differences, Similarities

Part 2 – The Core Practices

- Universally Recommend Practices
  - Common across all Lean & Agile methodologies

Part 1

Introduction to Lean Software Development

The Problem

- **Have you ever…**
  - Gone over budget?
  - Missed a deadline?
  - Had your project cancelled?
  - Delivered software that didn't really meet the needs of the customer?

- **You're not alone**

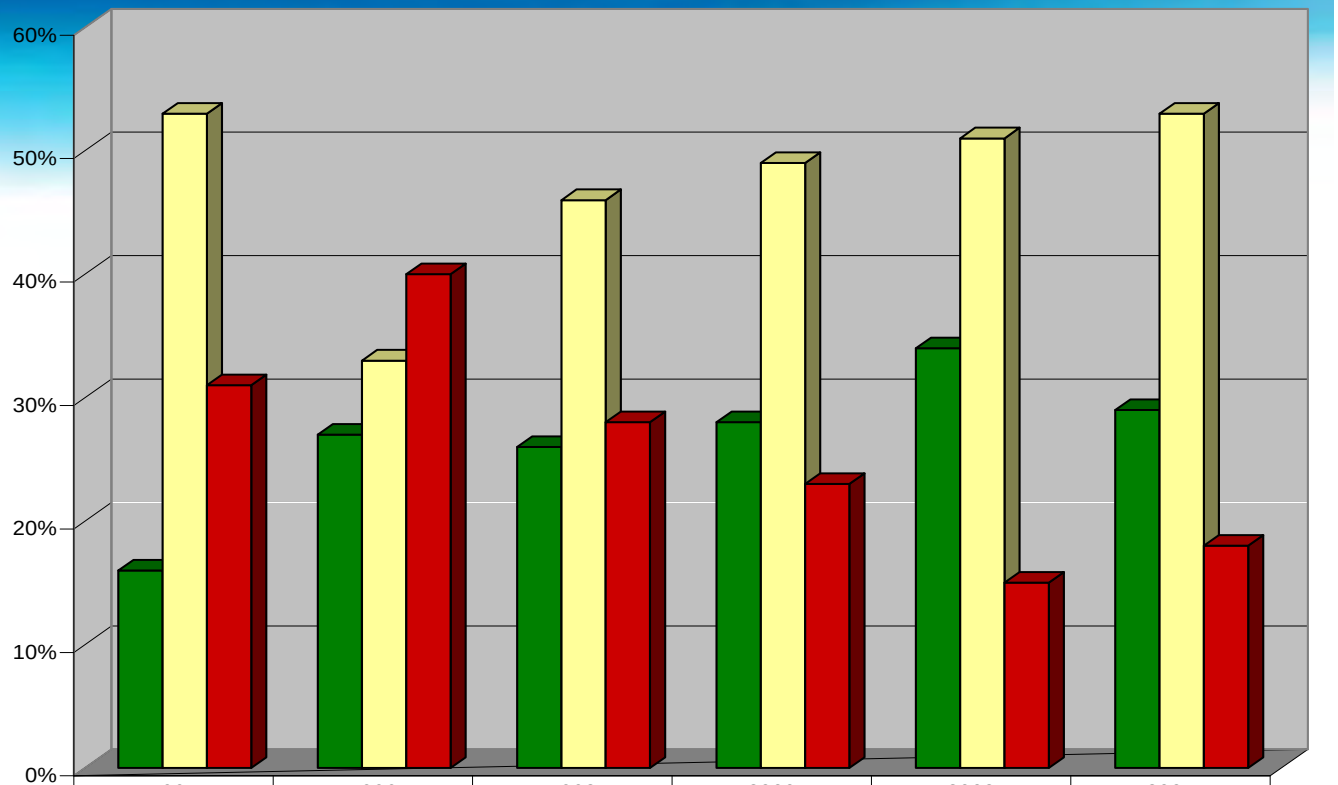
Some Statistics

- **The CHAOS Study[1]**
  - The landmark study of software project failures
  - Multiple reports issued over a period of more than ten years
  - Analyzed more than 40,000 projects

- **The original 1994 CHAOS report found**
  - Only 16% of software development projects were successful
    - Out of 8,000 projects studied
      - 31% failed outright
      - 53% were challenged (failed to meet schedule or budget)

[1] http://www.standishgroup.com/

CHAOS Study



	1994	1996	1998	2000	2002	2004
Succeeded	16%	27%	26%	28%	34%	29%
Challenged	53%	33%	46%	49%	51%	53%
Failed	31%	40%	28%	23%	15%	18%

- **Success rate is improving at the glacial pace of 1.3% a year (average)[1]**
- **This is consistent with other (smaller) studies.**
  - For more details, see Craig Larman's book *Agile & Iterative Development, Chapter 6: Evidence*

[1] the CHAOS study is ongoing, having published additional reports in 2006 and 2008. See http://www.standishgroup.com/

Why Aren't the Successes Higher?

- **As with most things, no single reason, but…**
- **A large part can be traced to the spread of the Waterfall method**

- **Waterfall Method**
  - Has distinct phases
  - Phases are sequential
  - Handoffs to different teams
  - Has an appealing air of simplicity
  - Project managers like the easily tracked milestones

Waterfall History

- **1970: *Managing the Development of Large Software Systems* by Winston Royce**
  - Often cited as the paper that validates the Waterfall Method
  - It does describe the Waterfall Method
  - But concludes the Waterfall Method *is risky and invites failure.*
  - It then advocates iterative development.

- **1982: DOD-STD-2167**[1]
  - Required Waterfall in software procurement
- **1987: Internally, DoD recommends iterative development**
- **1994: MIL-STD-498**[1]
  - Says iterative development is preferred

- **Waterfall continues to be used, resulting in:**
  - Reduced productivity
  - Increased defect rates
  - Increased project failure

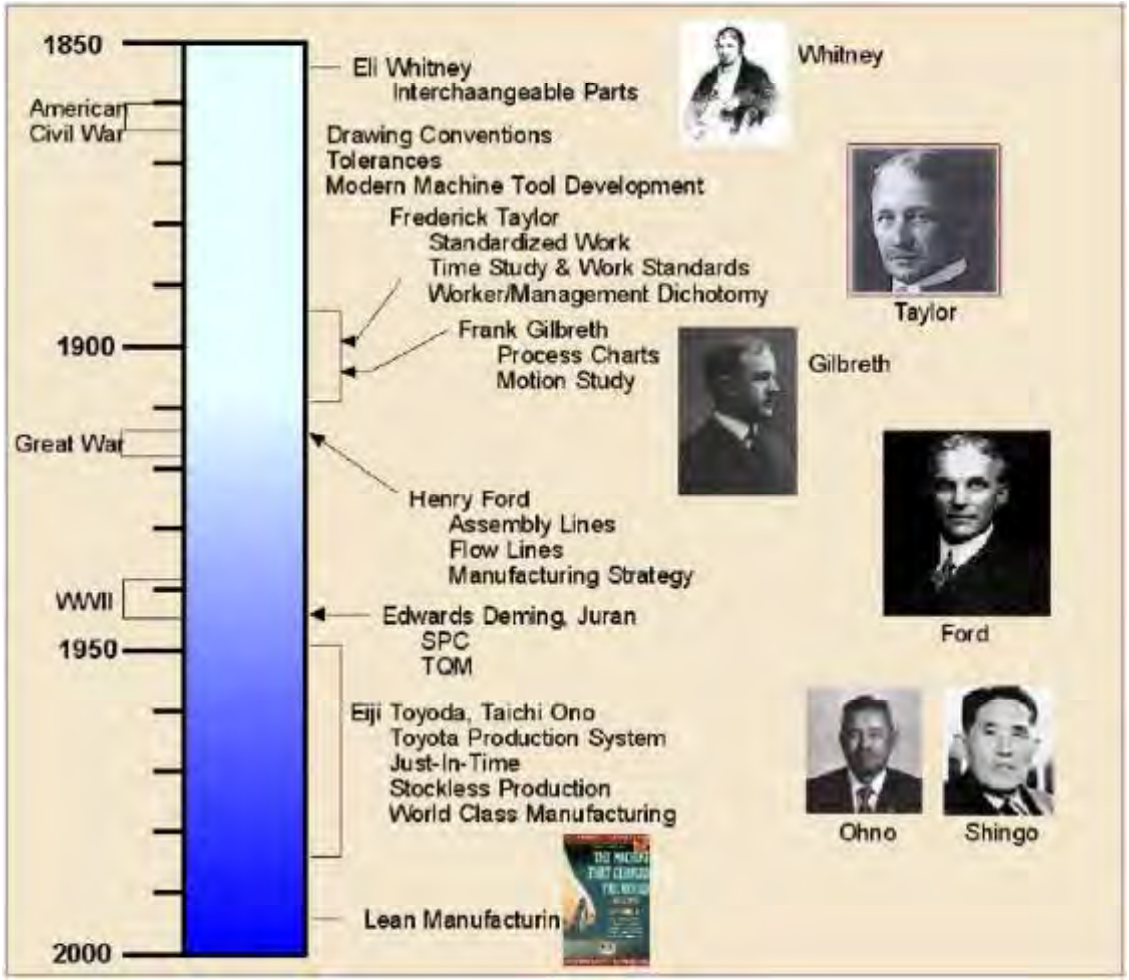
[1] DOD-STD-2167 and MIL-STD-498 are public.
See http://www.everyspec.com/DoD/DoD-STD/DOD-STD-2167A_8470/ and http://www.everyspec.com/MIL-STD/MIL-STD+(0300+-+0499)/MIL-STD-498_10233/

Lean & Agile SW Development

- **Agile (and predecessors):**
  - Response to failures of Waterfall
  - Called Lightweight methods in the 1990s,
  - Agile term coined in 2001
  - Focused specifically on Software Development
- **Lean**
  - Began in Toyota manufacturing around 1950 as *Just-In-Time*
  - "Lean" term coined in 1990
  - Originally focused on manufacturing
  - In the 1990s Lean principles were applied to other areas:
    - Product Development, Supply Chain, Office…
- **Lean Software Development**
  - Lean first applied to software development in 2003
  - Draws heavily on both Lean principles and the Agile experience

Whirlwind History of Lean

- **1800s**
  Craft Production
- **1900s**
  Mass Production
- **1950**
  Lean Production
- **1990**
  Office, Supply Chain,
  Engineering, Banking…
- **2003**
  Lean Software



Graphic used with permission from http://www.strategosinc.com/

Whirlwind History of Lean

- **Toyota Production System**
  - Taiichi Ohno described the Toyota Production System as a system for the absolute elimination of waste.
  - By the early 1990s Toyota was 60% more productive with 50% fewer defects than its non-Lean competitors.

- **Elimination of Waste**
  - A major part of any form of Lean
  - Waste is *anything* that does not add value in the eyes of the customer.

- **We'll say more on value and waste in just a bit…**

Lean Principles

- **Value**
  - understand what adds value for the customer
- **Value Stream**
  - understand how the organization generates customer value
- **Flow**
  - maximize speed and minimize cost by achieving continuous flow
- **Pull**
  - deliver value on a just-in-time basis based on actual customer demand
- **Perfection**
  - continuously improve the performance of your value streams

— Womack and Jones, *Lean Thinking*

Agile Software Development

Manifesto for Agile Software Development

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.

www.AgileManifesto.org

- **There is significant overlap between Lean and Agile!**

Lean & Agile

	Individuals and Interactions over processes and tools	**Working Software** over comprehensive documentation	**Customer Collaboration** over contract negotiation	**Responding to Change** over following a plan
Specify value in eyes of customer		✓	✓	✓
Identify value stream, eliminate waste	✓	✓	✓	✓
Make value flow at the pull of the customer	✓	✓	✓	✓
Pull value based on customer demand		✓	✓	
Continuously improve in pursuit of perfection			✓	✓

— Boeing IT: Lean Training for IT Systems

Lean & Agile

- **Agile Software Development**
  - Primary focus is
    - Close customer collaboration
    - Rapid delivery of working software as early as possible.
  - Formal methodologies: XP (eXtreme Programming), Scrum, Crystal, etc.

- **Lean Software Development**
  - Lean is not prescriptive, but analytical and open-ended
  - The focus is on delivering value to the customer at a quicker pace (primarily through the elimination of waste)
  - Lean has no formal methodologies
  - Has a toolkit of recommended practices

Value & Waste

- **Waste**
  - Waste is anything that does not add value in the eyes of the customer
  - There can be multiple customers, each valuing different things

- **Categories of Activities**
  - Value Added
    (MUST meet ALL 3 criteria to be VA)
    - The customer wants it (e.g., is willing to pay for it)
    - And, it changes the form, fit and/or function of the product or service (physical change)
    - And, it's done right the first time (no rework)
  - Non-Value Added but Necessary
    - Required by law, regulation or policy
    - Causes no value creation, but cannot eliminate based on current technology or thinking
  - Non-Value Added (Waste)
    - Consumption of resources without adding any value in eyes of customer
    - Pure waste
    - If an activity cannot be eliminated, it is NVAN

Types of Waste

- **A first step in implementing Lean software development is learning to identify waste in many forms**
- **The seven classic categories of waste in manufacturing have direct analogs in software development.**

Lean Manufacturing	Lean SW Development
Defects	Defects
Over Production	Extra Features
Transportation	Handoffs
Waiting	Delays
Inventory	Work in Progress
Motion	Task Switching
Over Processing	Unneeded Processes

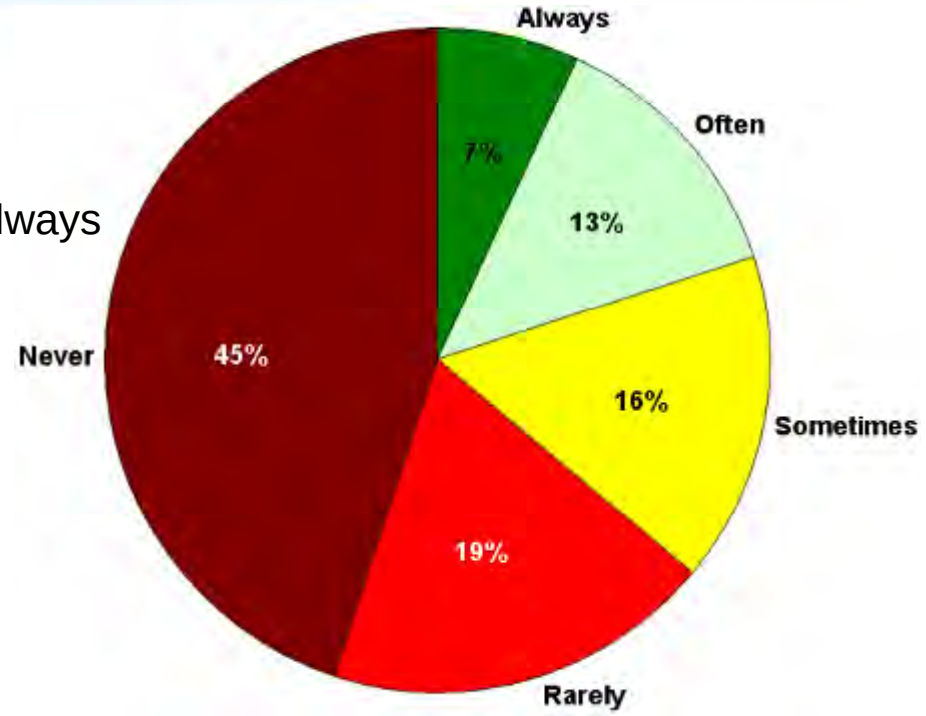
Let's go through these one at a time…

Defects → Defects

- **Defects cause expensive rework**
  - Lean focus: preventing defects
  - Defects are especially expensive when detected late
- **Lean response to a defect:**
  - Find the root cause
  - Ensure the defect cannot recur
- **In software this means:**
  - automated tests
- **When a defect does slip through:**
  - A new test is created to detect that defect
  - It cannot pass through undetected again

Over-Production → Extra Features

- **Every line of code costs money**
- **Standish CHAOS Study:**
  - 45% of features are never used
  - Only 20% of features were used often or always
- **A 2001 study[1]:**
  - 400 projects
  - Over a 15 year period
  - Less than 5% of the code was actually useful or used!
- **This is a huge waste**
  - Increasing drain on resources over time



If a feature does not address a clear customer need, then it should not be created

[1] "Improving Software Investments through Requirements Validation"
IEEE 26th Software Engineering Workshop

Transportation → Handoffs

- **Sound familiar? Classic Waterfall Process (worst case)**
  - Analysts create a document containing all of the product requirements and hand them off to the architects
  - Architects take the requirements and create the product design, which they hand off to the programmers
  - The programmers write the code to implement the design, and pass the results to the testers
  - The testers validate the resulting product against the requirements
- **Knowledge is lost in each handoff**
  - Architects won't understand the requirements as deeply as the analysts
  - Programmers will not understand the design as well as the architects
  - This incomplete understanding will lead to errors and omissions, which will require costly rework to correct

Try to avoid handoffs whenever possible

Waiting → Delays

- **In development, decisions are made almost constantly**
  - Most of the time a developer will know (or can deduce) the answers
  - But they can't know everything, and sometimes must ask questions
- **When immediate answers are obtained**
  - There is no delay
  - Development continues at full speed
- **Having to wait only has wasteful possibilities**
  - Suspend the current task; move on to something else
  - Try to find the answer; just guess the answer
    - And even when the developer tries to find the answer, if it's just too much trouble, they'll end up guessing just to save the hassle
- **Best organization: co-located, integrated teams**
  - Provides high bandwidth communications
  - Minimizes delays

Inventory → Work in Progress

- **Work in Progress (WIP) is anything started but not finished**
  - Requirements (features) that haven't been coded
  - Code that hasn't been tested, documented, and deployed
  - Bugs that haven't been fixed
- **Traditional approach**
  - Let WIP build up in queues to be completed later
  - End result: Incomplete features and deferred bugs accumulate
- **Lean approach**
  - Use single-piece flow to take a feature through to deployment as rapidly as possible
  - End result: more completed, bug free features ready sooner
- **A feature is not done until it is potentially deployable**
  - This means fully documented, tested, and error-free
  - "Potentially" because other constraints may not allow actual deployment as often as we would like

Motion → Task Switching

- **Task switching (and interruptions) kill productivity**
  - Focus on the task at hand
    - Mental ramp up takes time
    - After ramp-up, problem solving flows
  - Interruptions
    - Must restart mental ramp up
  - Task switching (a much longer interruption)
    - Must “relearn” where you were
    - Much longer mental ramp up

- **Lean advocates single-piece flow**
  - Productive because you can work completely through a feature or task without the waste of task switching

Over Processing → Unneeded Processes

- **Unneeded processes are pure waste**
  - They reduce productivity without adding any value
- **Includes things like:**
  - Procedures that accomplish no purpose
  - Documents that no one reads
  - Manual tasks that should be automated
  - Procedures that make simple tasks hard

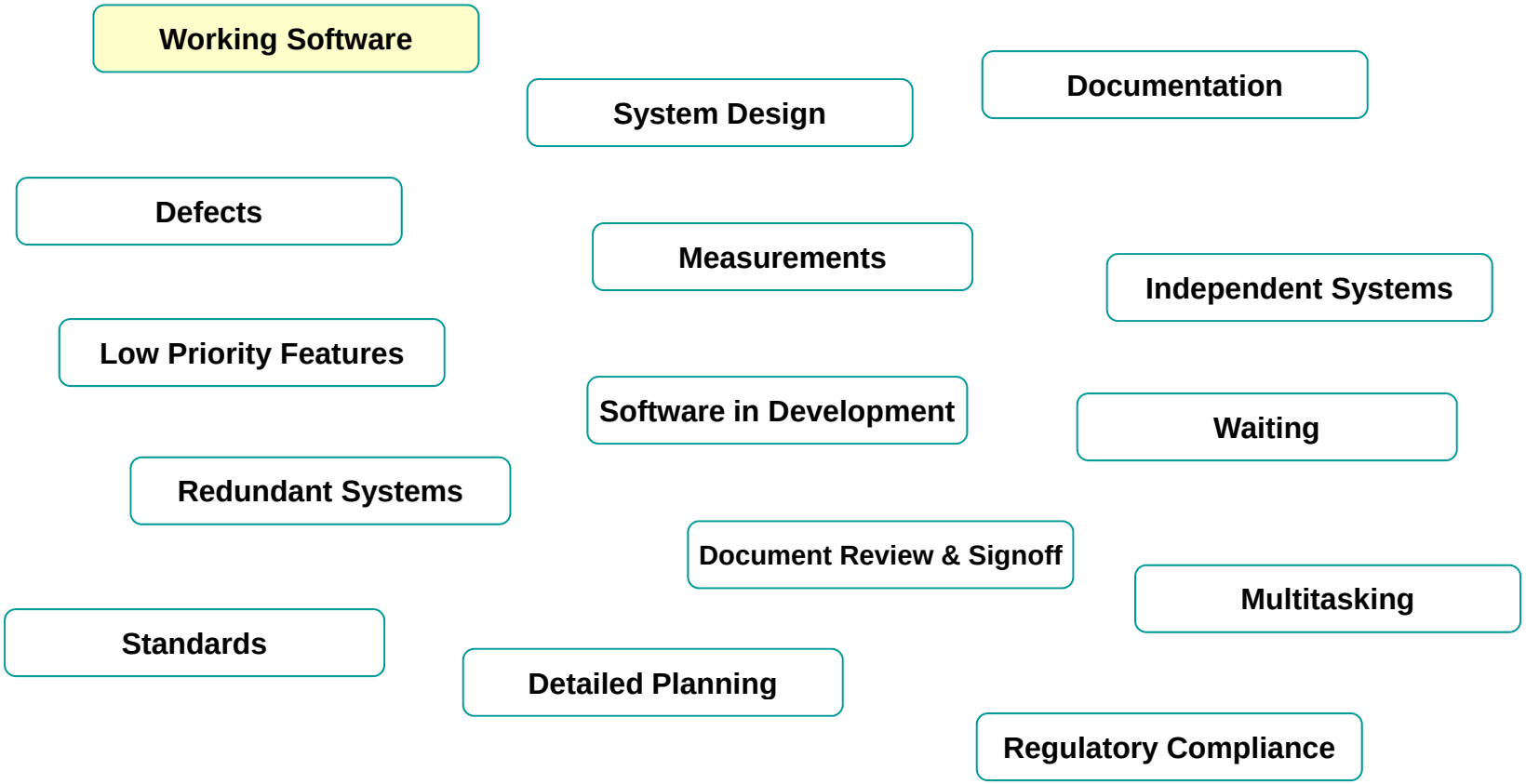
Value or Waste?

Which of the following deliverables and activities add value, and which represent waste?

- Working Software
- System Design
- Documentation
- Defects
- Measurements
- Independent Systems
- Low Priority Features
- Software in Development
- Waiting
- Redundant Systems
- Document Review & Signoff
- Multitasking
- Standards
- Detailed Planning
- Regulatory Compliance

— Boeing IT: Lean Training for IT Systems

Working Software: Value or Waste?



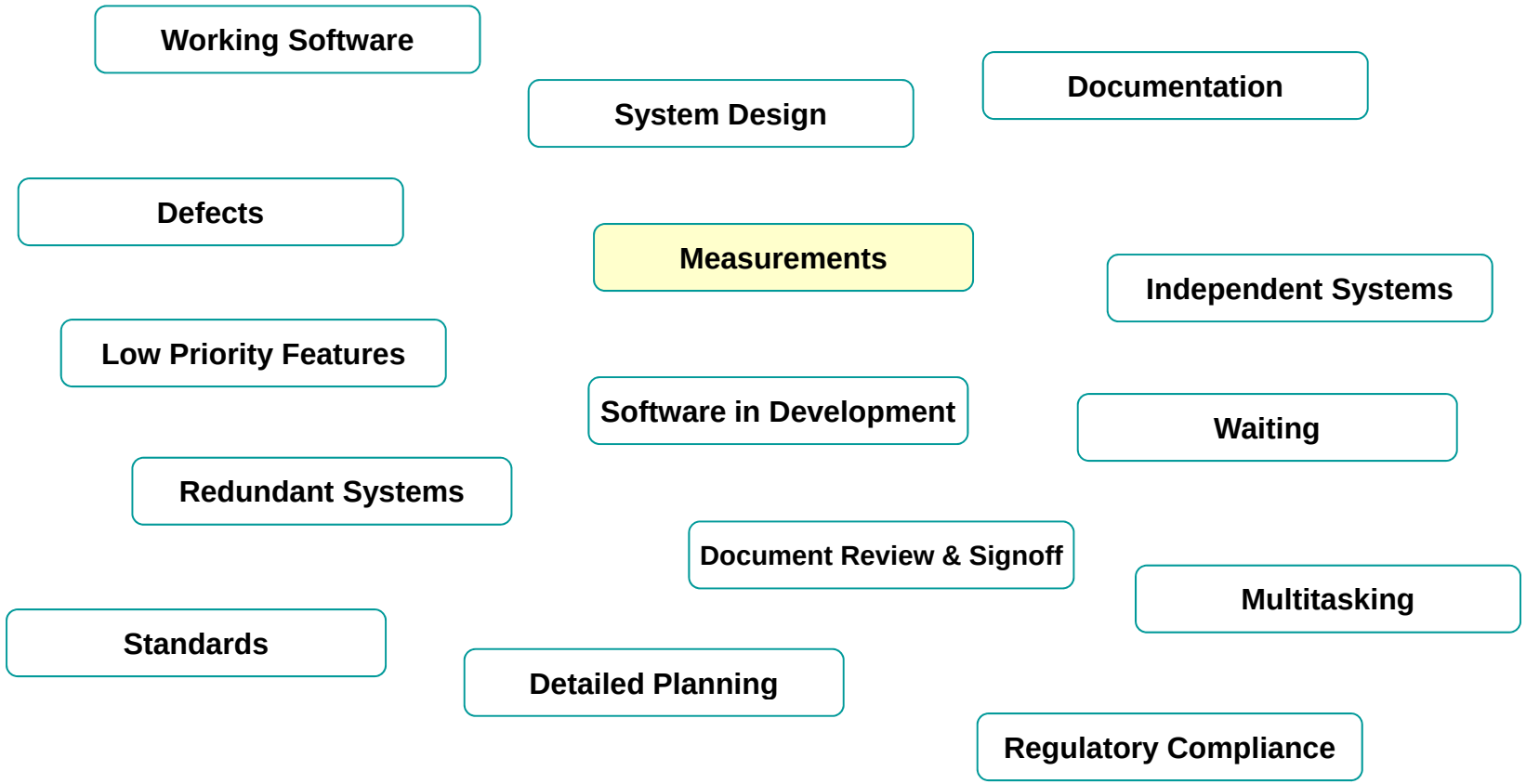
— Boeing IT: Lean Training for IT Systems

Working Software: Value or Waste?

Answer: It Depends

- Features that are actually used represent value**.**
- Features that are not used, are waste.

Measurements: Value or Waste?



— Boeing IT: Lean Training for IT Systems

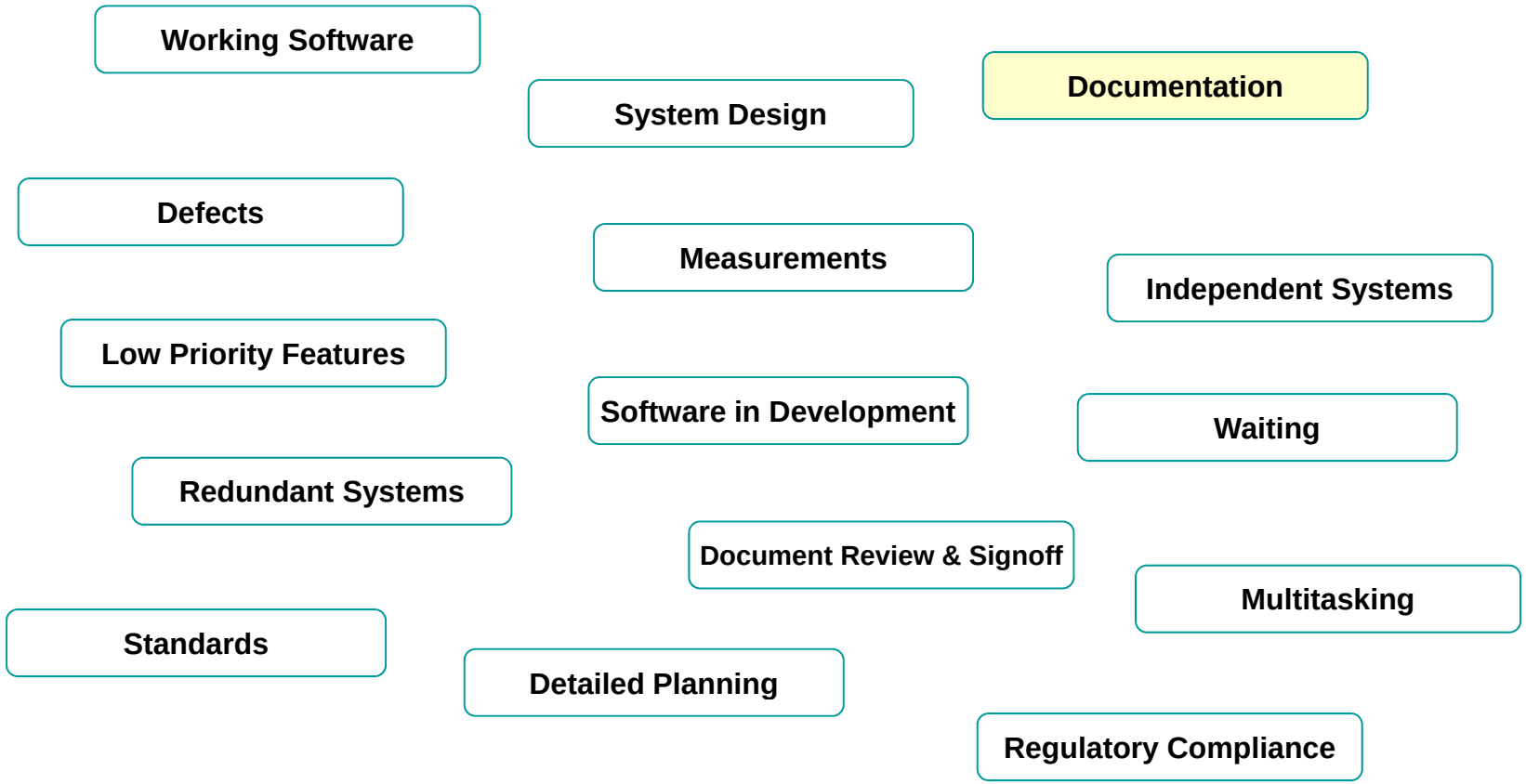
Measurements: Value or Waste?

Answer: It Depends

- **Informational measures**
  can be essential to:
  - problem-solving
  - process improvement
- **Motivational measures**
  - use performance measures as part of a rewards system
  - cause waste
- **The difference is not in the measure, but in how it is used**
  - makes it difficult to implement purely informational measures

- **Measurements that are collected but not used for decision-making are also wasteful.**

Documentation: Value or Waste?



— Boeing IT: Lean Training for IT Systems

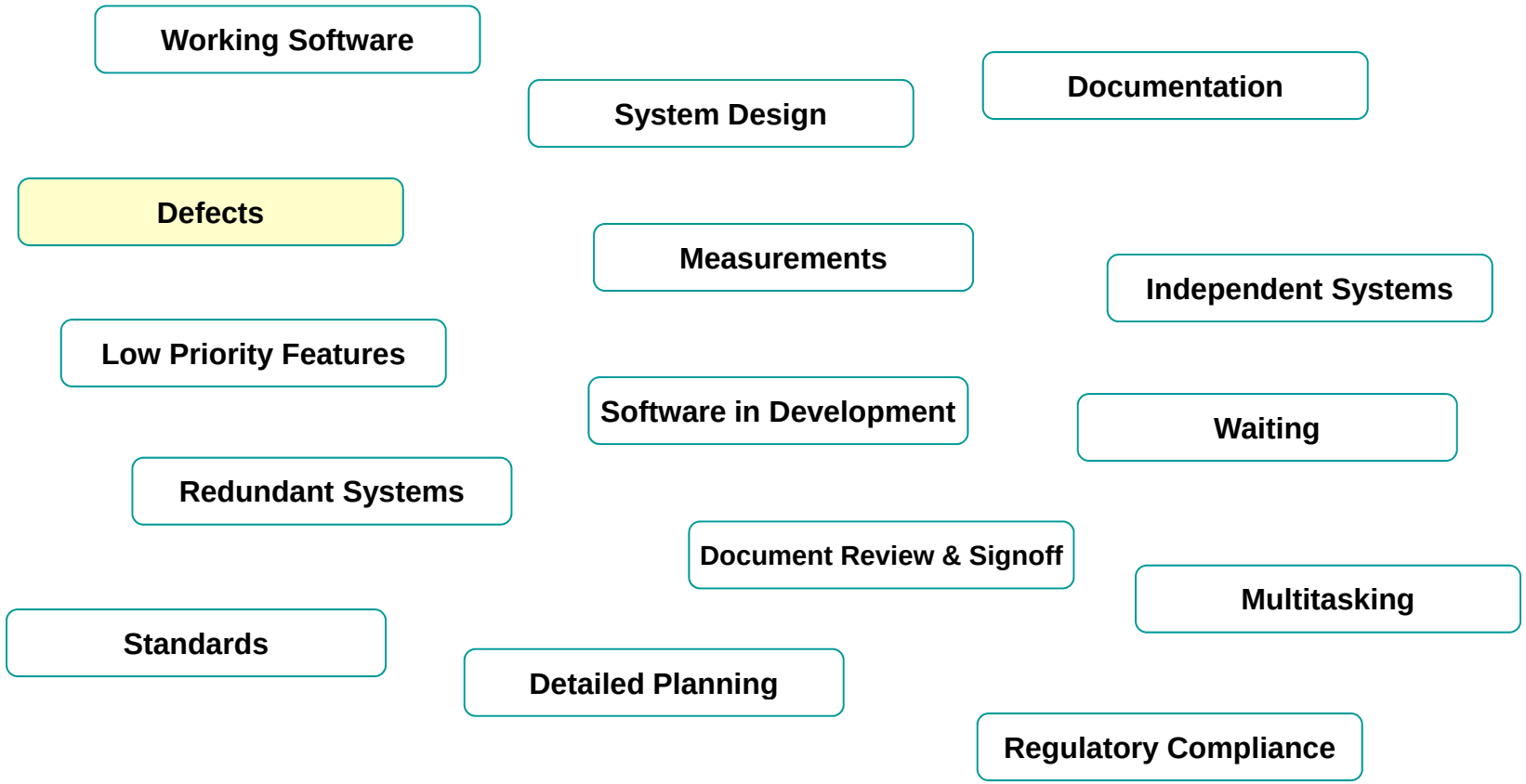
Documentation: Value or Waste?

Answer: It Depends

- **Documentation can be very wasteful.**
  - Too long or dense to read
  - Not kept up-to-date
    - Its production was waste
    - Misunderstandings based on it may cause defects.
  - Too vague, imprecise or poorly written to offer any value.
  - Can't find it when needed, then it is not adding value.

- **But, concise, well-written documentation can be valuable**
  - especially for audiences that are separated by distance or by time.

Defects: Value or Waste?



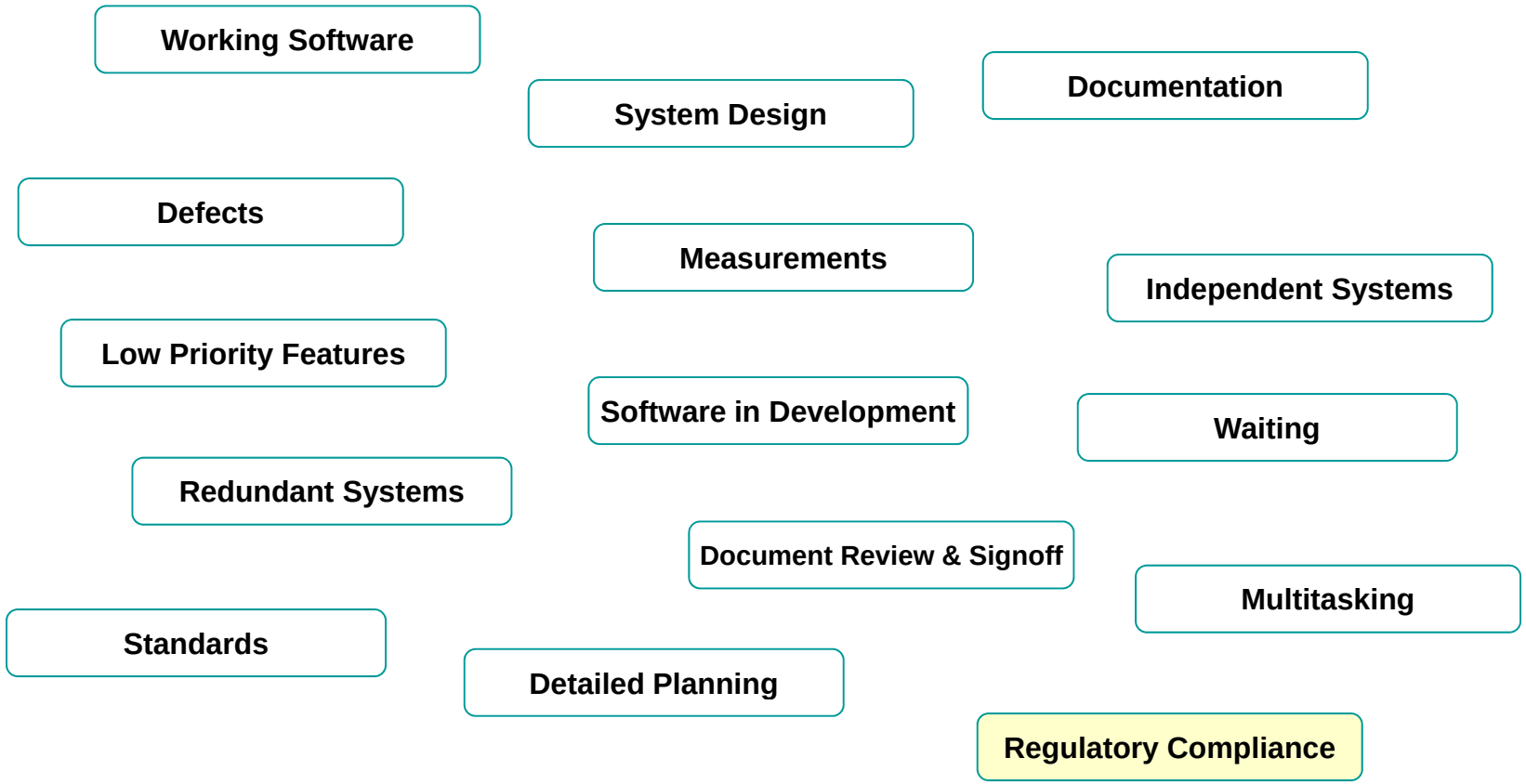
— Boeing IT: Lean Training for IT Systems

Defects: Value or Waste?

Answer: Waste

- **Defects cause expensive rework or outright scrap.**
- **Minimize defect waste**
  - Identify and correct them quickly
  - Automated tests prevent defects from reoccurring without detection

Regulatory Compliance: Value or Waste?



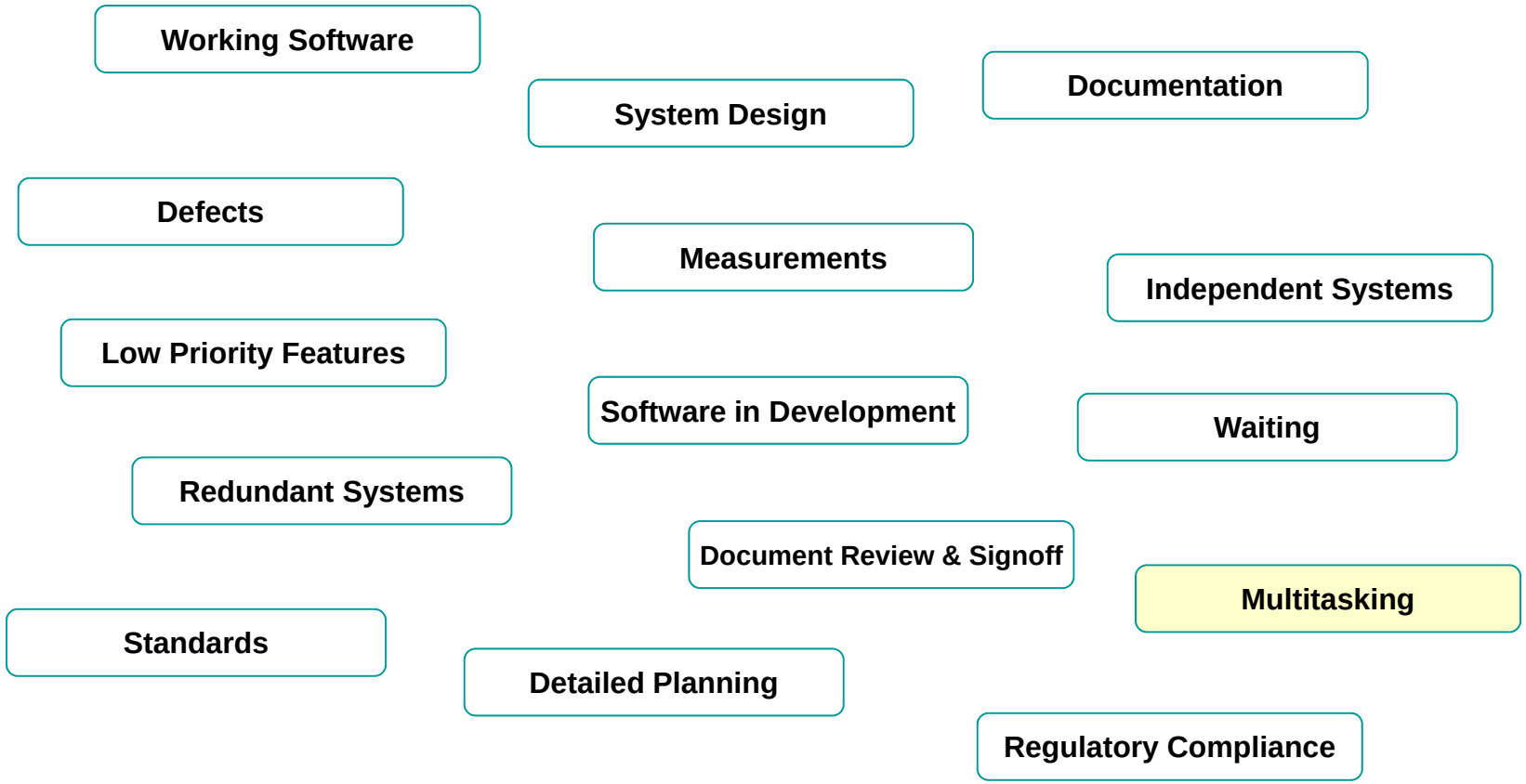
— Boeing IT: Lean Training for IT Systems

Regulatory Compliance: Value or Waste?

Answer: Non-Value Added but Necessary

- **Compliance allows you to stay in business**
  - Sarbanes-Oxley
  - ITAR/EAR
  - Compliance is a prerequisite for delivering value to our customers

Multi-tasking: Value or Waste?



— Boeing IT: Lean Training for IT Systems

Multi-tasking: Value or Waste?

Answer: Waste

- **Workers juggling multiple assignments**
  - Spend more time on task switching
  - Waste time reorienting themselves from one task to another
- **As workloads go up, work accomplished goes down**
- **Overloaded employees make more mistakes and are less productive.**

Lean Software Development Principles

- **Mary and Tom Poppendieck**
  - Derived Lean software development principles from *Lean Product Development*
  - According to the Poppendiecks:

    "*Principles are underlying truths that don't change over time or space, while practices are the application of principles to a particular situation. Practices can and should differ as you move from one environment to the next, and they also change as a situation evolves.*" [1]

- **Poppendieck Lean Software Development Principles[1]:**
  - Principle 1 - Eliminate Waste
  - Principle 2 - Build Quality In
  - Principle 3 - Create Knowledge
  - Principle 4 - Defer Commitment
  - Principle 5 - Deliver Fast
  - Principle 6 - Respect People
  - Principle 7 - Optimize the Whole

[1] from the book "*Implementing Lean Software Development: From Concept to Cash*" by Mary and Tom Poppendieck

Principle 1: Eliminate Waste

- **We've already covered this topic in excruciating detail, so let's move on…**

Principle 2: Build Quality In

- **Key insight from Lean manufacturing:**
  - You cannot inspect quality into a product at the end of a production line
  - This detects problems but does nothing to correct them
  - Each process step should be *mistake-proof* and *self-inspecting*
- **Traditional vs. Lean software development approach**
  - *Traditional approach:* Let defects slip through and get caught later by testing
  - *Lean approach:* Mistake-proof your code by writing tests as you code the features

Tests prevent subsequent changes to the code from introducing undetected defects

Principle 3: Create Knowledge

- **Retrospection**
  - There is a learning curve where we build on our experience and lessons learned
  - One technique for doing this is a retrospective where we review our processes at the end of each iteration
  - This way, poor performance or waste can be identified and mitigated prior to the next iteration
- **Repeating mistakes and relearning is waste**

You shouldn't have to relearn what you already know

Principle 4: Defer Commitment

- **The best decisions are made when the most information is available**
  - Not too soon…
    - Before all possible helpful information is available
  - Not too late…
    - Don't want to delay downstream work
  - Just right…
    - At last responsible moment

Wait until the last responsible moment to make an irreversible decision

(don't use this as an excuse to avoid planning)

Principle 5: Deliver Fast

- **Software is abstract**
  - When we can *see* it, its easier to *think* about it
  - We work best with concrete things
- **Software requirements are volatile**
  - Waterfall approach: wait until the end to get feedback
  - This is too late, prone to failure
- **Deliver fast means**
  - Developing features in small batches using short iterations
  - The customer can use these (now concrete) features
  - The customer can change and reprioritize the requirements based on real use

Delivering fast results in:

- A product that more closely meets the real customer needs
- Reduces the waste and rework created by requirements churn

Principle 6: Respect People

- **This lofty altruism is also the down-home truth:**

  "*Engaged, thinking people provide the most sustainable competitive advantage.*" [1]

- **Respect for people means**
  - Trusting them to know the best way to do their jobs
  - Engaging them to expose flaws in the current process
  - Encouraging them to find ways to improve their job and its surrounding processes
  - Recognizing them for their accomplishments and actively soliciting their advice

Don't waste your most valuable resource:
the minds of your team members!

[1] from the book "*Implementing Lean Software Development: From Concept to Cash*" by Mary and Tom Poppendieck

Principle 7: Optimize the Whole

- **A system is not a collection of collaborative parts - it is the product of these collaborative interactions**
  - Having all the best parts will not necessarily result in the best system
- **Any time you optimize a local process you are almost always doing so at the expense of the whole value stream**
  - This is sub-optimizing
- **If you don't have control over the entire value stream**
  - You may be forced to sub-optimize a piece of it

Always include as much of the value stream as possible when you optimize a process

Supporting Evidence

- **Solid data supporting Lean-Agile software development are increasingly available**
- **Report from the Cutter Consortium[1]**
  - 7,500 completed projects
  - 500 companies (in 18 countries)
  - 20 of the projects were Agile
- **Results: Agile projects (as compared with traditional)**
  - Delivered faster
    - Or delivered more functionality
  - Delivered fewer defects
  - Had significantly reduced costs
  - See the report for specific numbers

[1] *How Agile Projects Measure Up, and What This Means to You*, 2008, by Michael Mah, Cutter Consortium.

Part 2

The Core Practices

Core Lean Software Development Practices

Several core practices are common to both Lean software development and Agile methodologies

- **Practice 1: Automated Testing**
- **Practice 2: Continuous Integration**
- **Practice 3: Less Code**
- **Practice 4: Short Iterations**
- **Practice 5: Customer Participation**

Agile development is all about priorities, so we've listed these in the order we believe brings the highest return on investment

Practice 0: Prerequisites

- **Basic Software Development Best Practices**
  - These practices apply to all software development efforts, regardless of the methodologies in use. If you're not doing these, ***start now***.
- **Prerequisites for *any* viable software development process, but *particularly for Lean* software development.**
  - Implementing Lean or Agile processes without these practices in place is like trying to run before you learn to walk.
- **Many such practices exist, but two are applicable here:**
  - Source Code Management
  - Scripted Builds

Source Code Management and Scripted Builds

- **Source Code Management (SCM)**
  - Place **ALL** items required to build a product from scratch in an SCM repository (e.g. Subversion, ClearCase), including build scripts, test code, database/XML schemas and initialization, etc.

> ***Pragmatic Programmer Tip 23:***
>
> **"Always Use Source Code Control"** [1]

- **Scripted Builds**
  - Write a build script to create the product from scratch by retrieving all necessary items from SCM and performing all build actions.
  - Practice 2 (Continuous Integration) makes extensive use of build scripts.

> ***Pragmatic Programmer Tip 61:***
>
> **"Don't Use Manual Procedures"** [1]

[1] **"The Pragmatic Programmer"; A.Hunt, D.Thomas; Addison-Wesley; 2000**

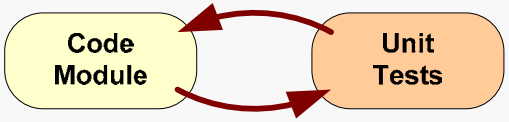
Practice 1: Automated Testing

Automated Testing is the use of test scripts and programmable test frameworks to execute tests *without user intervention*.

- **Automated Testing supports Lean development principles:**
  - Principle 1: Eliminate Waste
    - When defects are found, adding an automated test will prevent it from recurring
  - Principle 7: Build Quality In
    - Testing shouldn't find defects, it should prevent them from occurring in the first place

Types of Testing

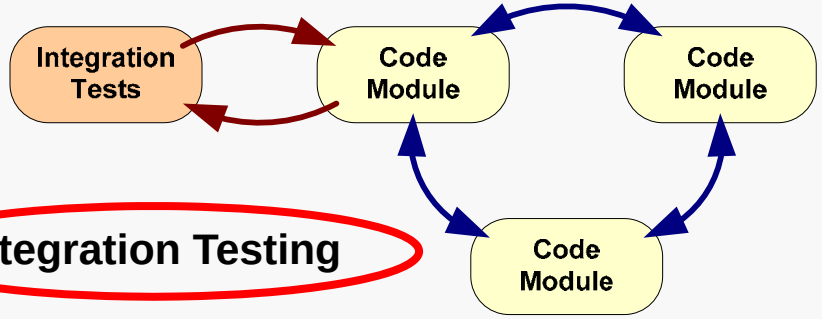
Unit Testing



Tests small code units individually

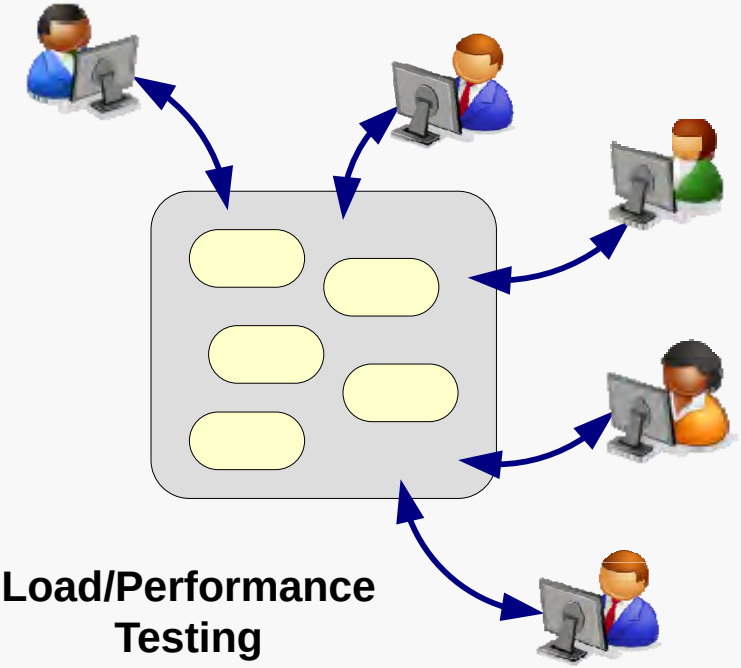
Integration Testing

Tests interaction between code units

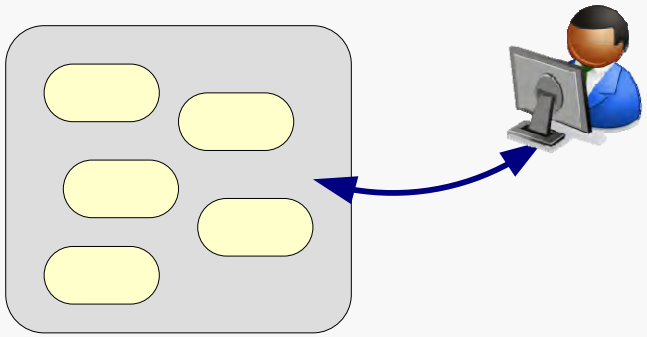


Load/Performance Testing

Tests ability to handle expected usage



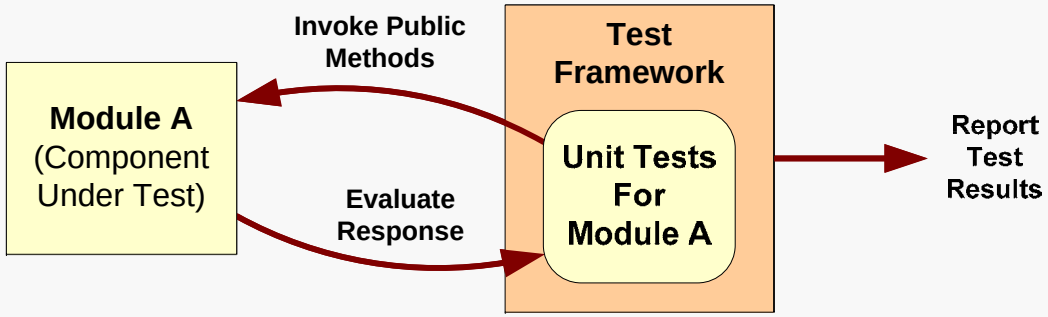
User/Acceptance Testing



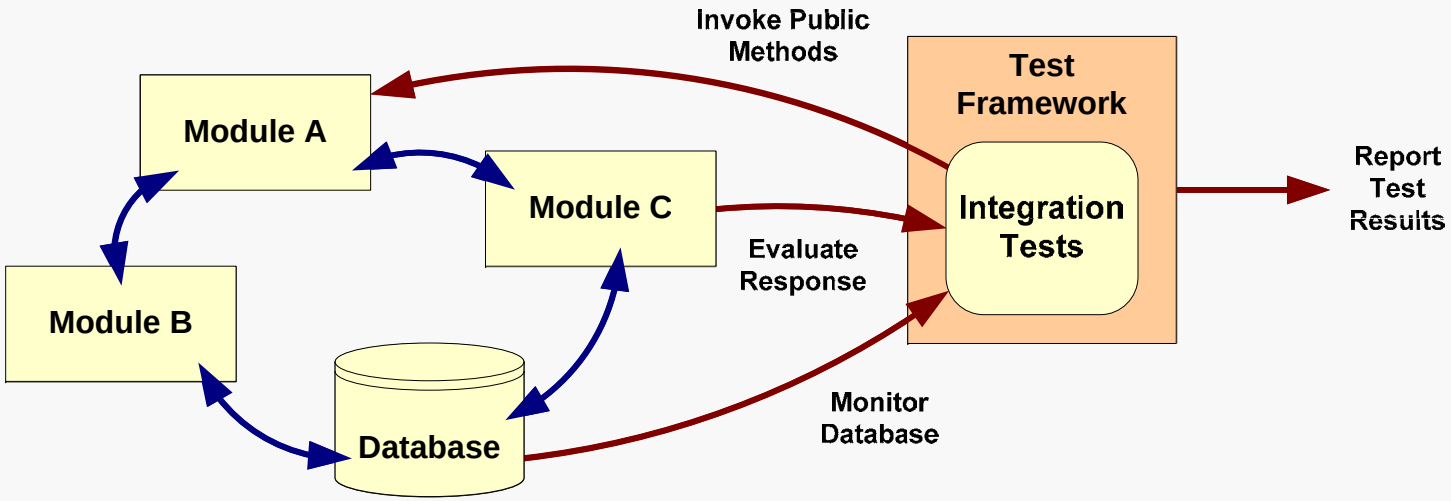
Tests functionality at the user level

Unit Testing vs. Integration Testing

Unit Testing



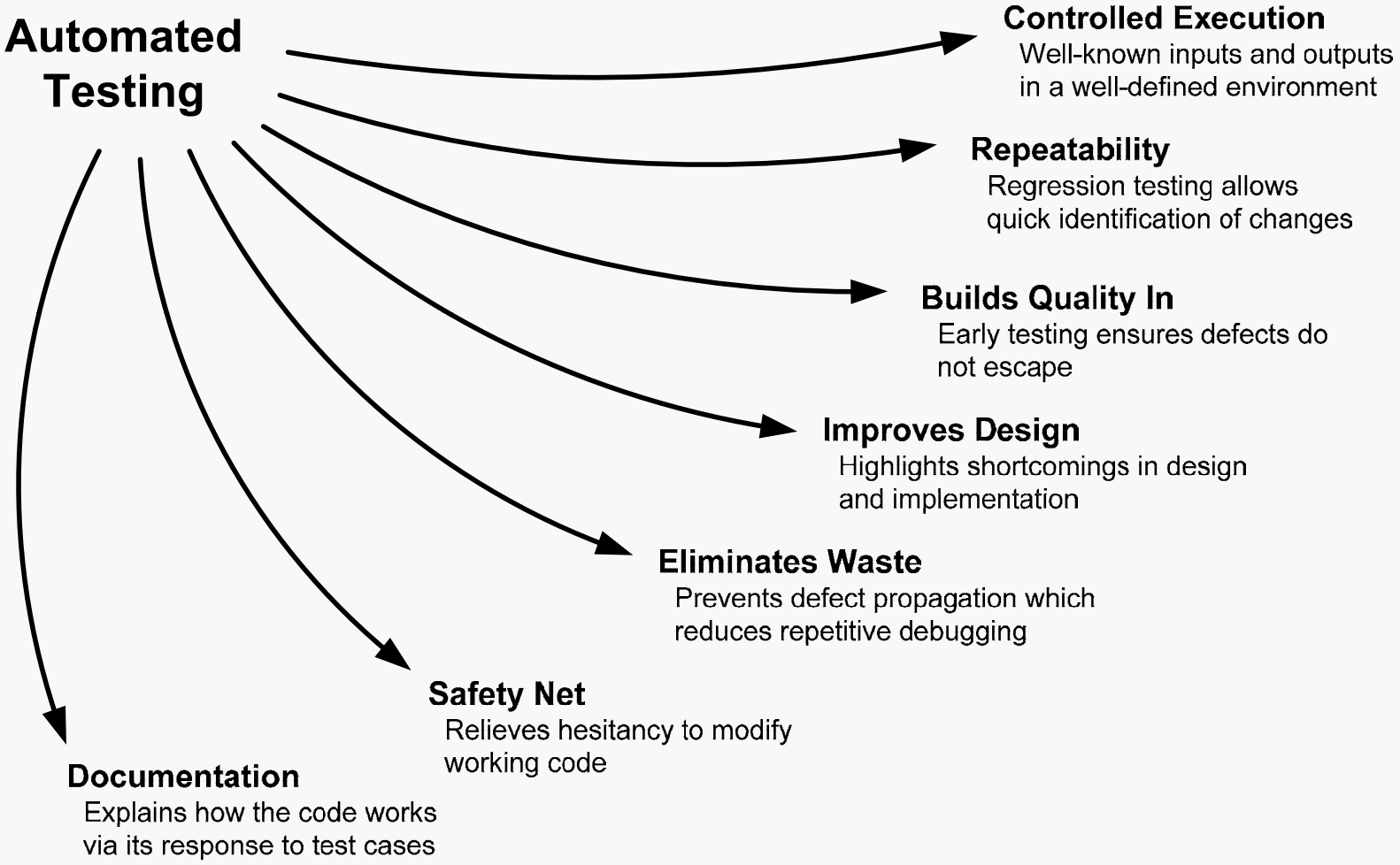
Integration Testing



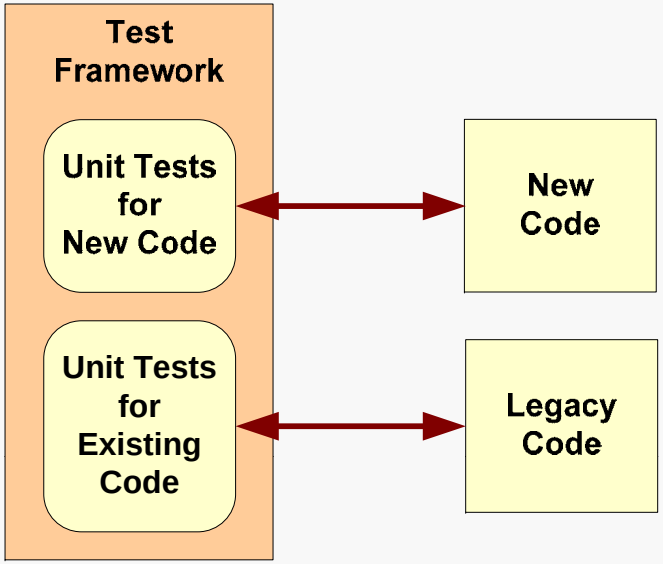
Automated Testing as a Methodology

- **Automated tests are a cornerstone of several test-oriented development techniques:**
  - **Test Driven Development (TDD)**
    - All application code is written along with a set of automated tests.
    - This may include any combination of unit tests, integration tests, or acceptance tests (unit tests are the most common).
  - **Test First Development (TFD)**
    - This is a variant of TDD where the tests are always written before the application code.
  - **Behavior Driven Development (BDD)**
    - An extension or evolution of TDD where tests are written from the point of view of the application's behavior.
    - Sometimes BDD includes writing the application's requirements as a set of behavior-style acceptance tests.

Benefits of Automated Testing



Implementing Automated Testing

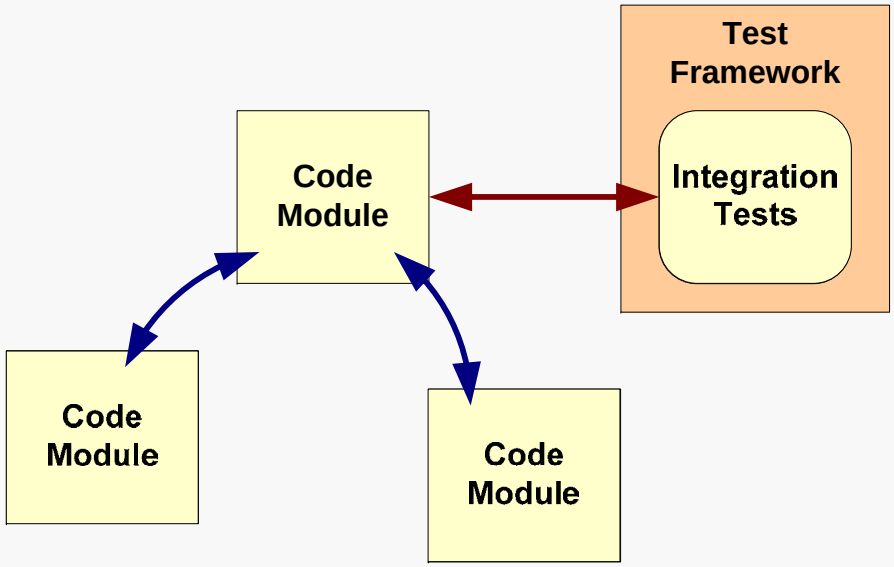


Unit Testing

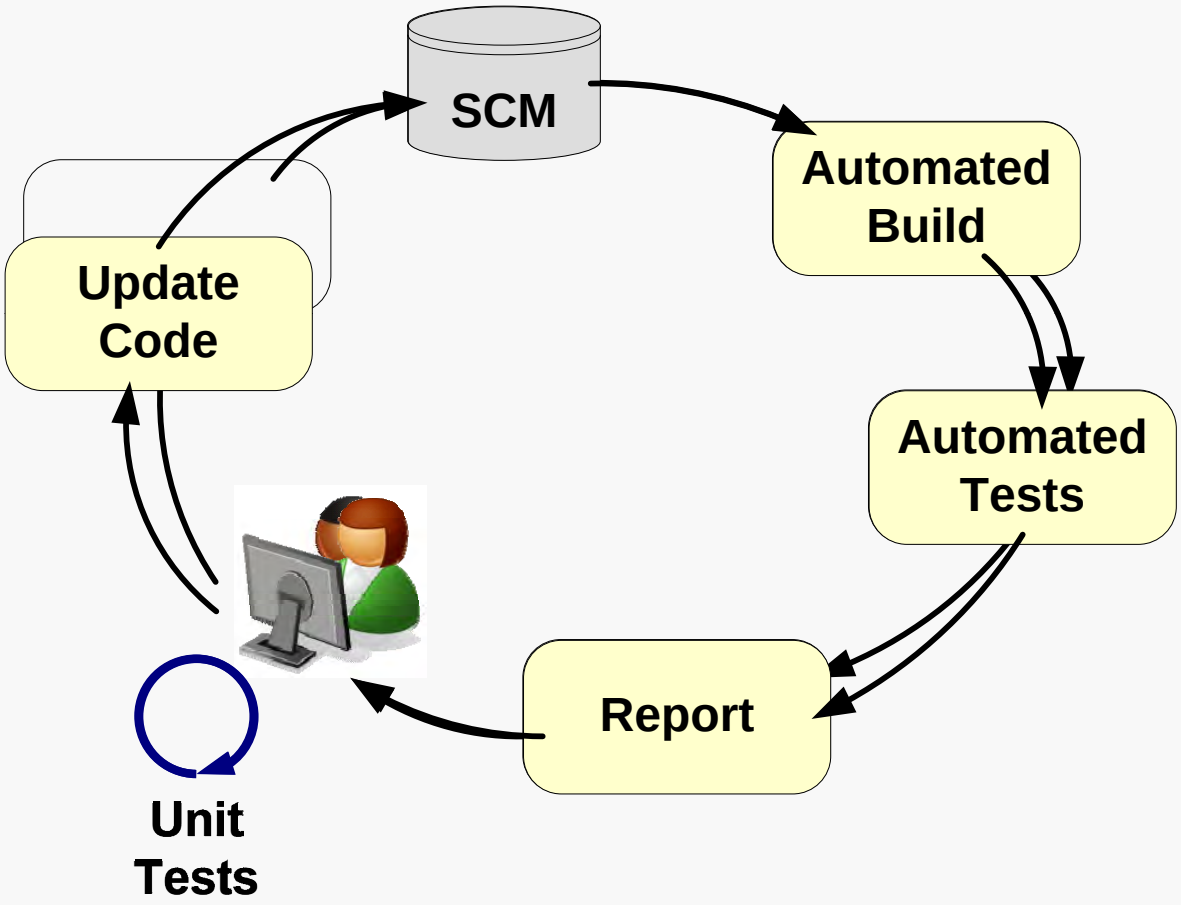
1. Identify appropriate unit test framework
2. Require unit tests for all new code
3. Retrofit unit tests to legacy code

Integration Testing

1. Use units test as basis for integration tests
2. Replace mock objects with real components
3. Script scenarios as test sequences

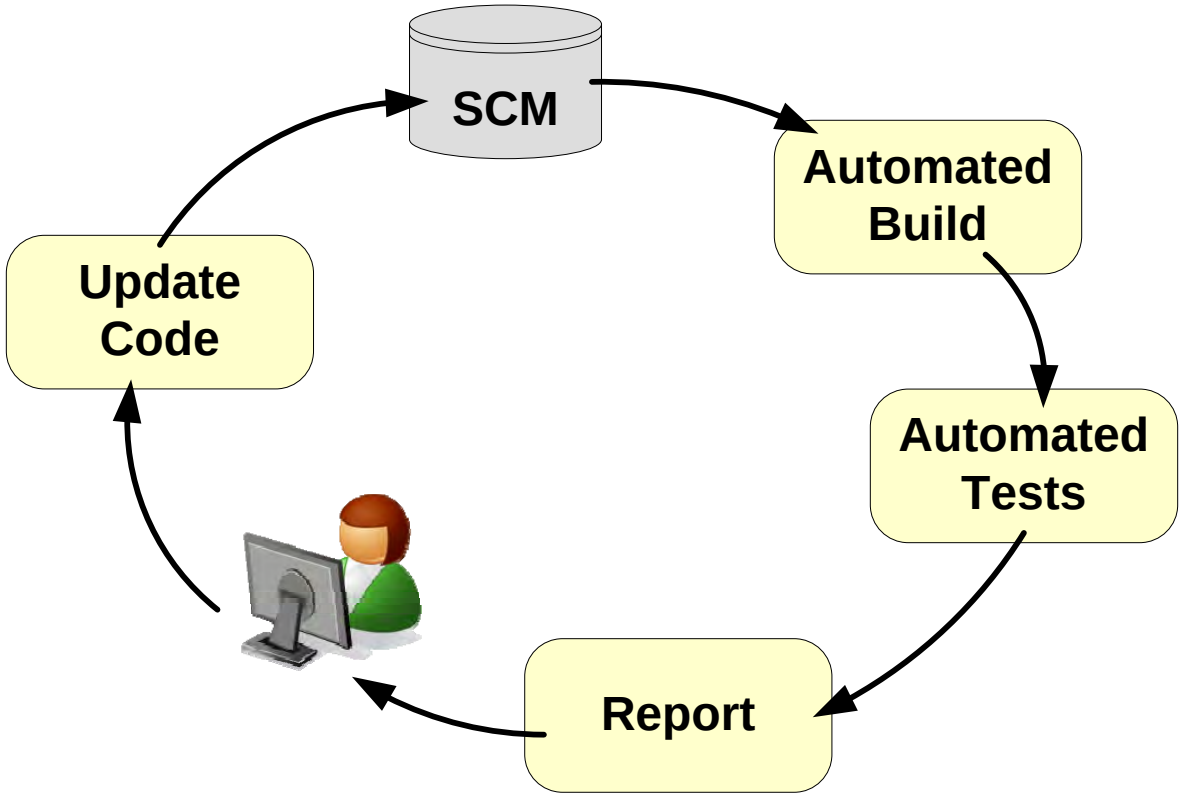


Executing Automated Tests

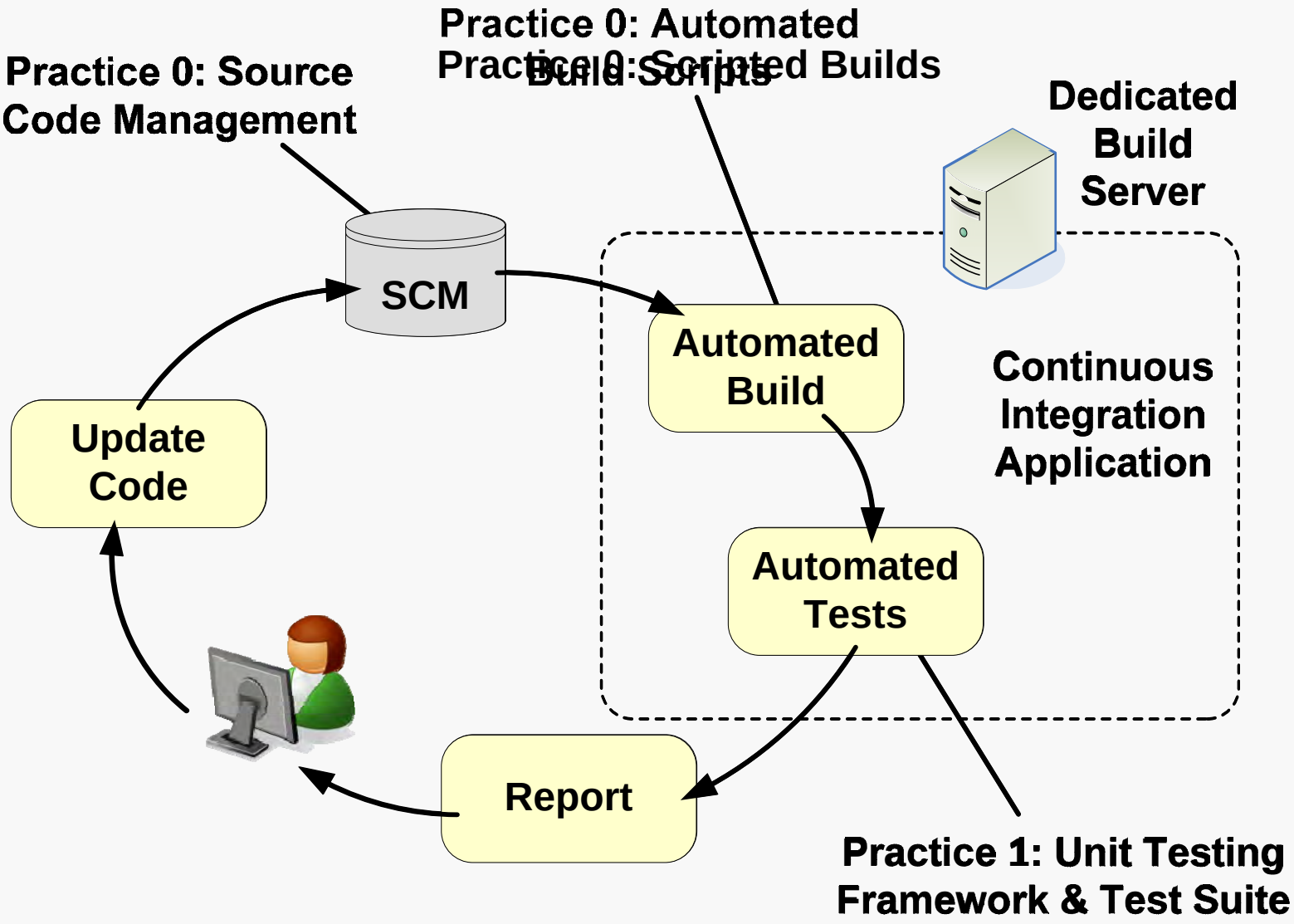


Practice 2: Continuous Integration

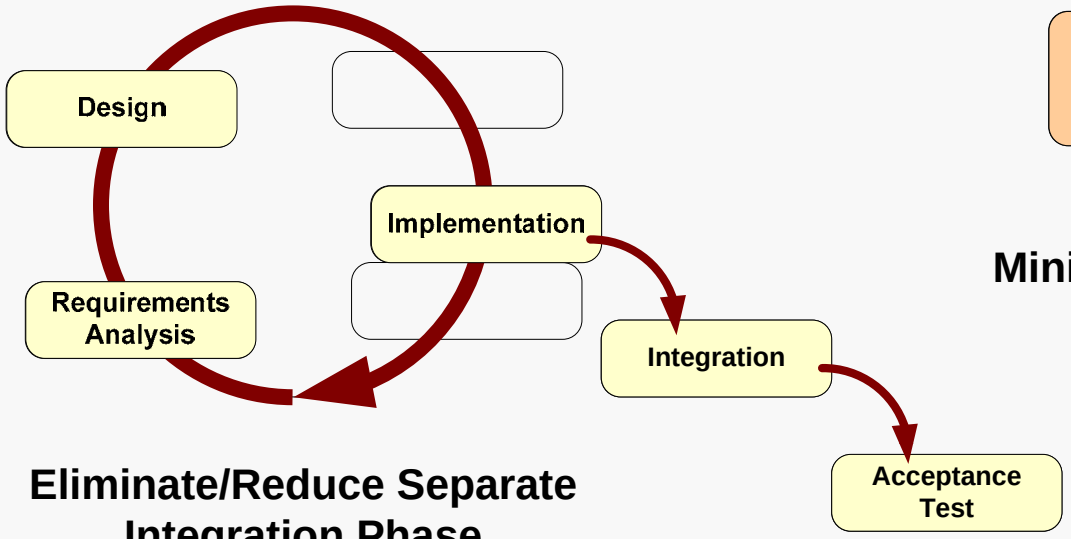
Continuous Integration is the *frequent* integration of *small* changes during implementation.



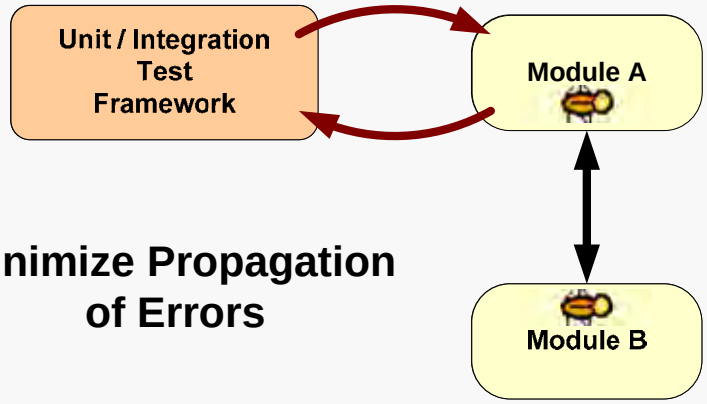
Continuous Integration Prerequisites



Benefits of Continuous Integration

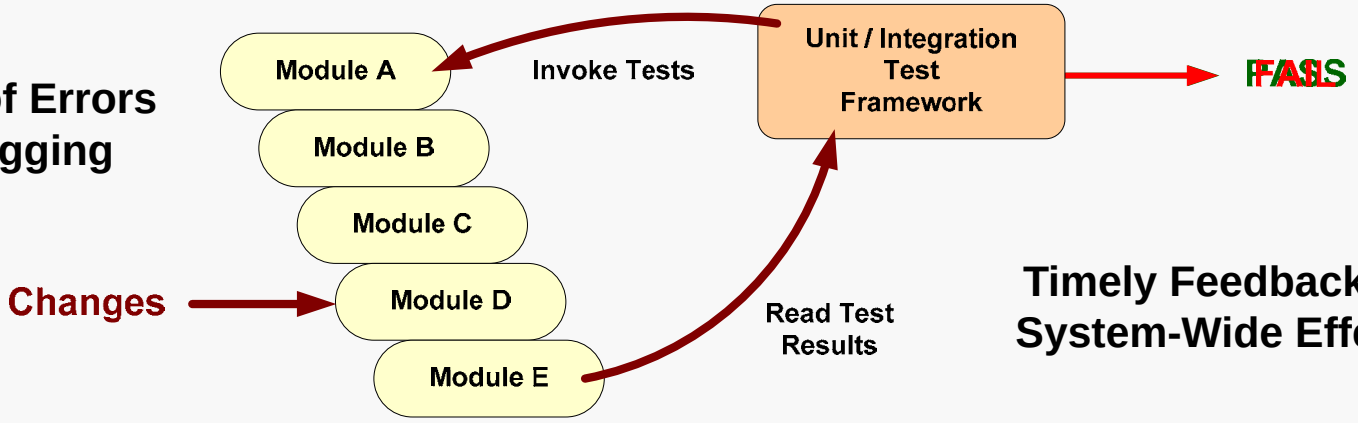


Eliminate/Reduce Separate Integration Phase



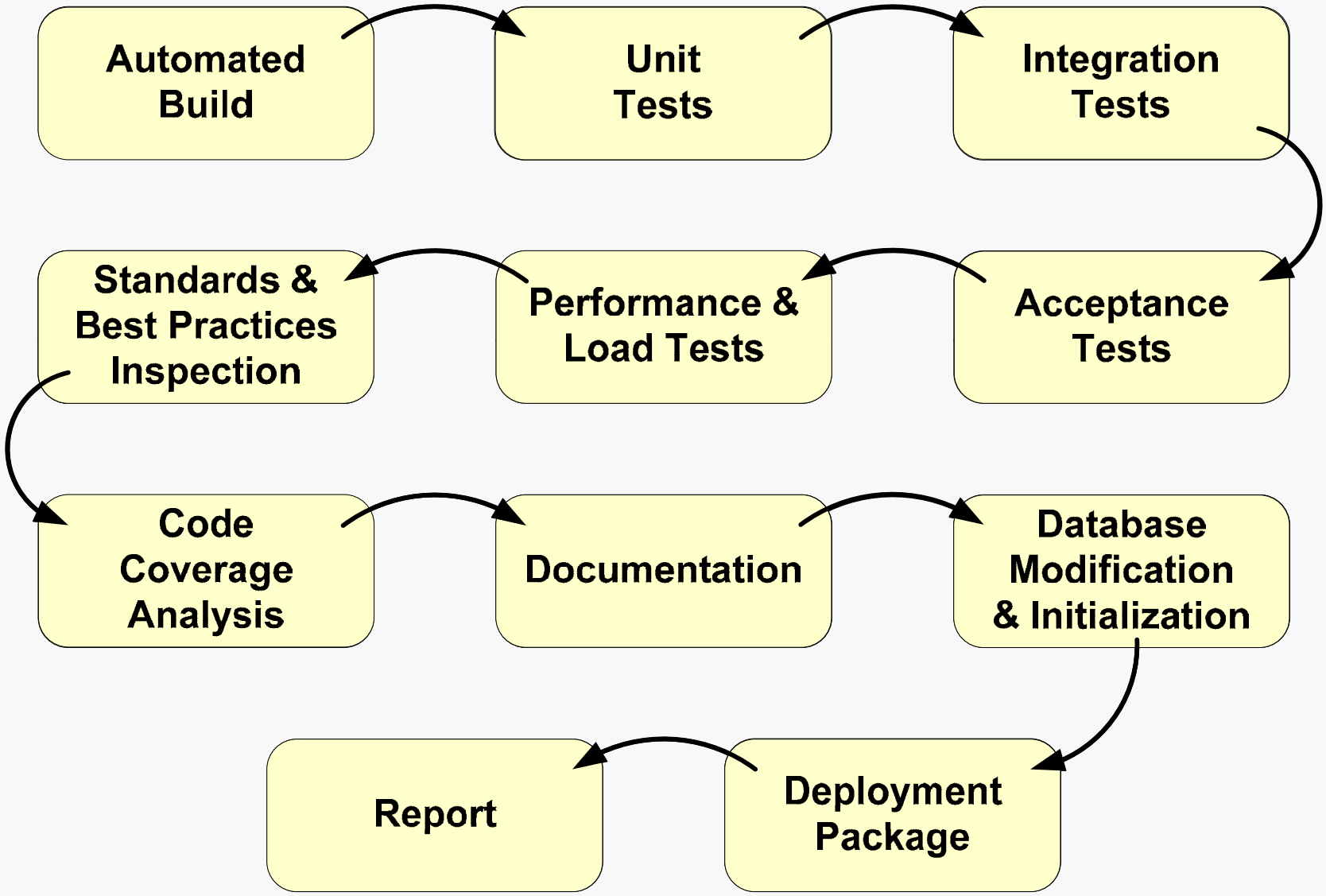
Minimize Propagation of Errors

Limits Scope of Errors During Debugging



Timely Feedback on System-Wide Effects

The Continuous Integration Process

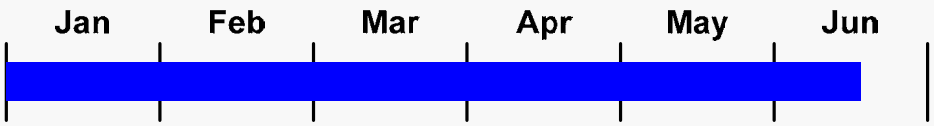


Practice 3: Less Code

Less Code ≠ Less Software

- **"Less Code" is a concept that drives development of *all required functionality* while *minimizing code base size*.**
- **The "Less Code" concept is realized in two ways:**
  - Eliminating unused and unnecessary code
  - Writing smarter, more efficient code
- **While "Less Code" is more of a concept than a practice, several specific techniques exist to eliminate unneeded code and make the code that is needed more efficient.**

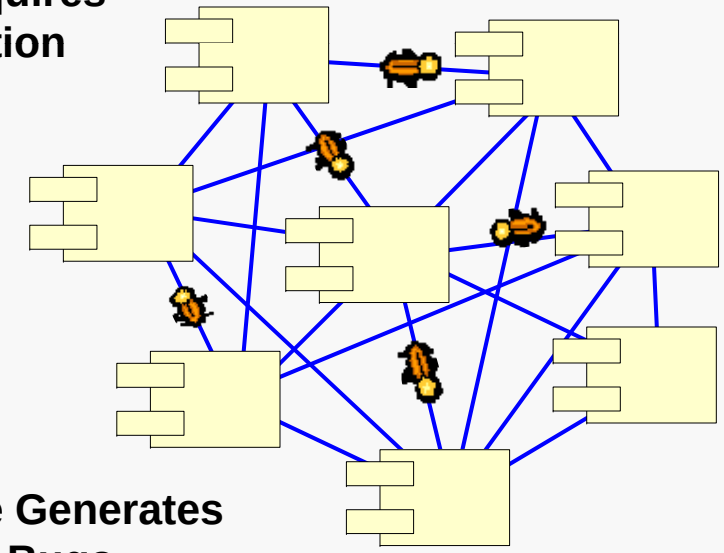
What's Wrong with a Big Code Base?



More Code Takes More Time

More Code Drives Higher Maintenance Costs

More Code Requires More Integration

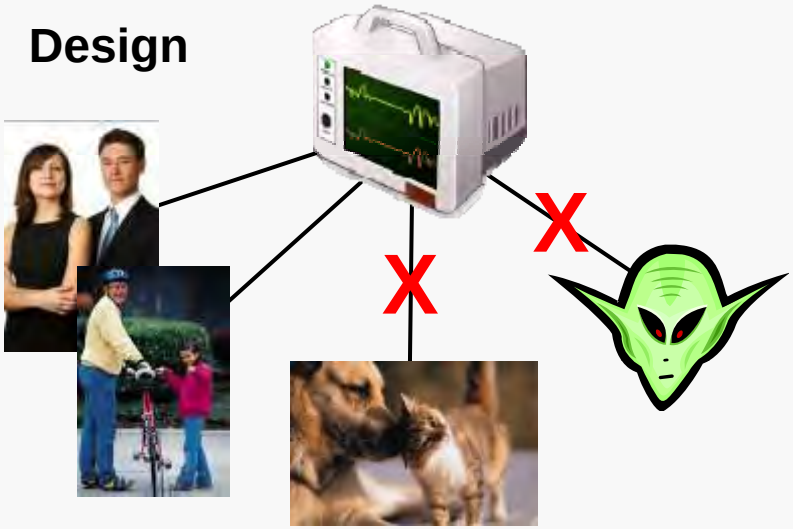


More Code Generates More Bugs

For Unused or Unneeded Code, This is Non-Valued Added Work, i.e. Waste

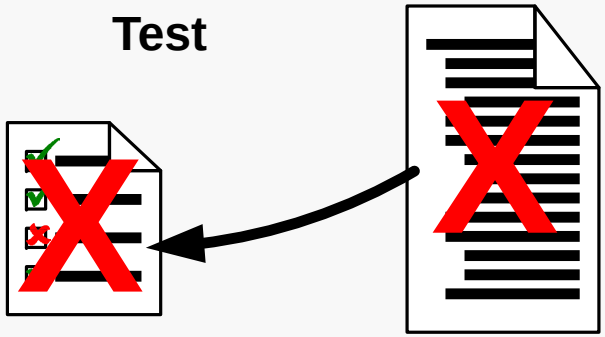
It's Not Just About Implementation

Design



Implementati

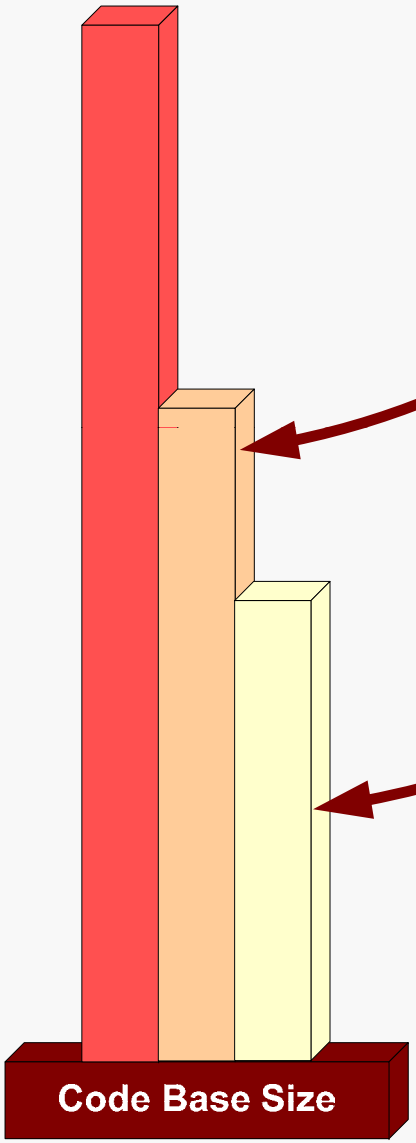
Test



Requirements

Code Base

Reducing Code Base Size



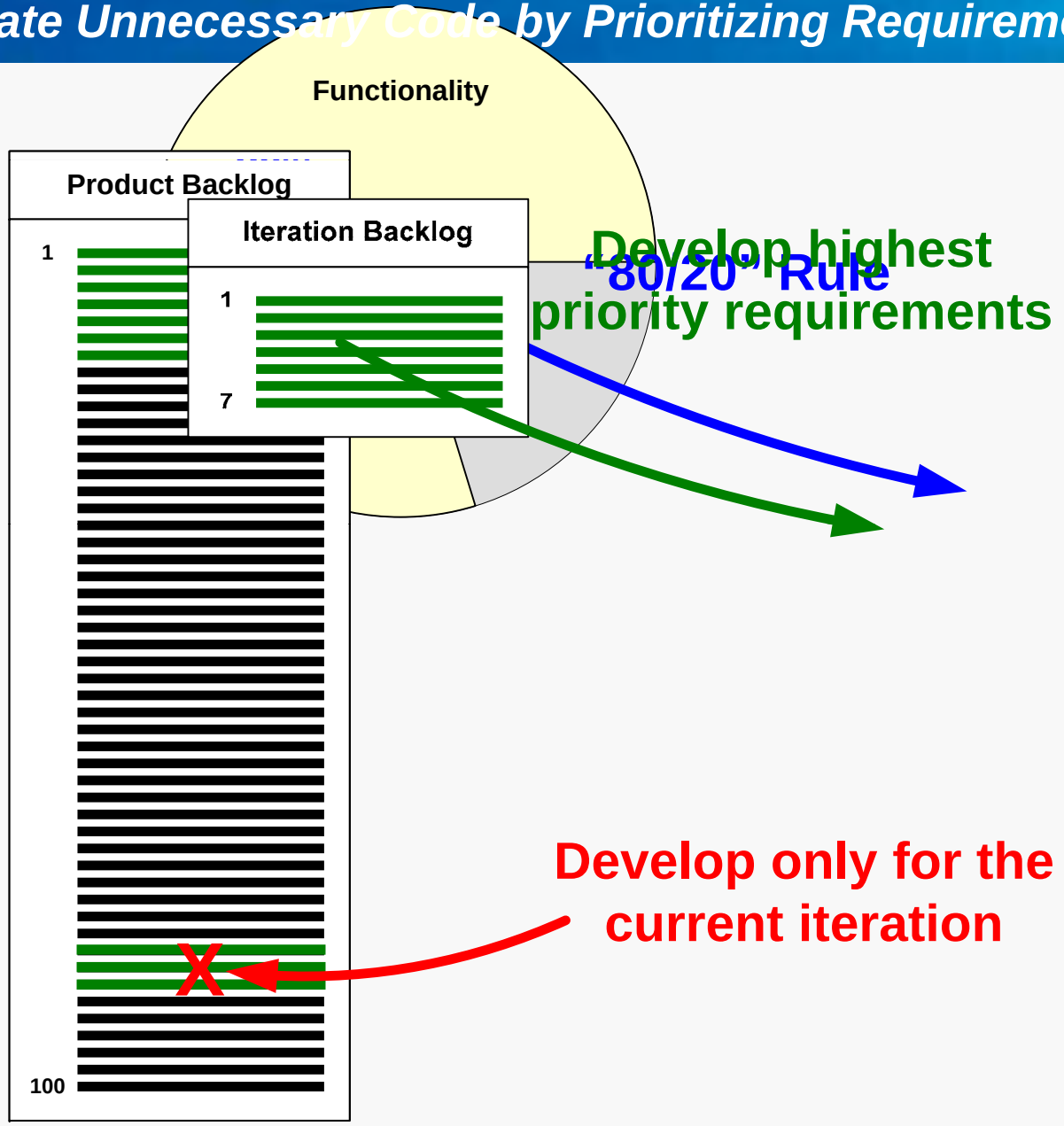
Eliminate Unnecessary Code

Diet

Employ Good Coding Practices

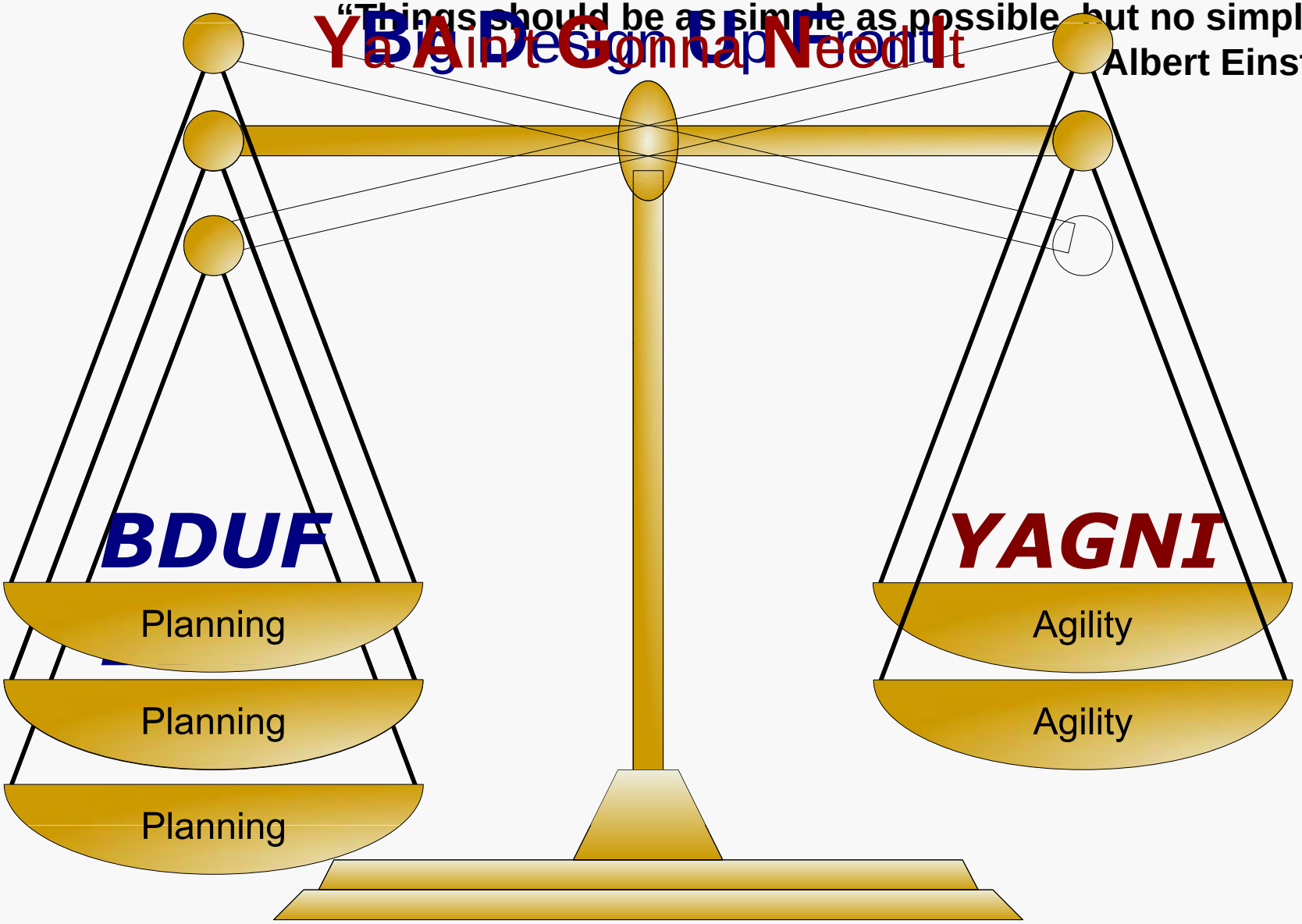
Exercise

Eliminate Unnecessary Code by Prioritizing Requirements

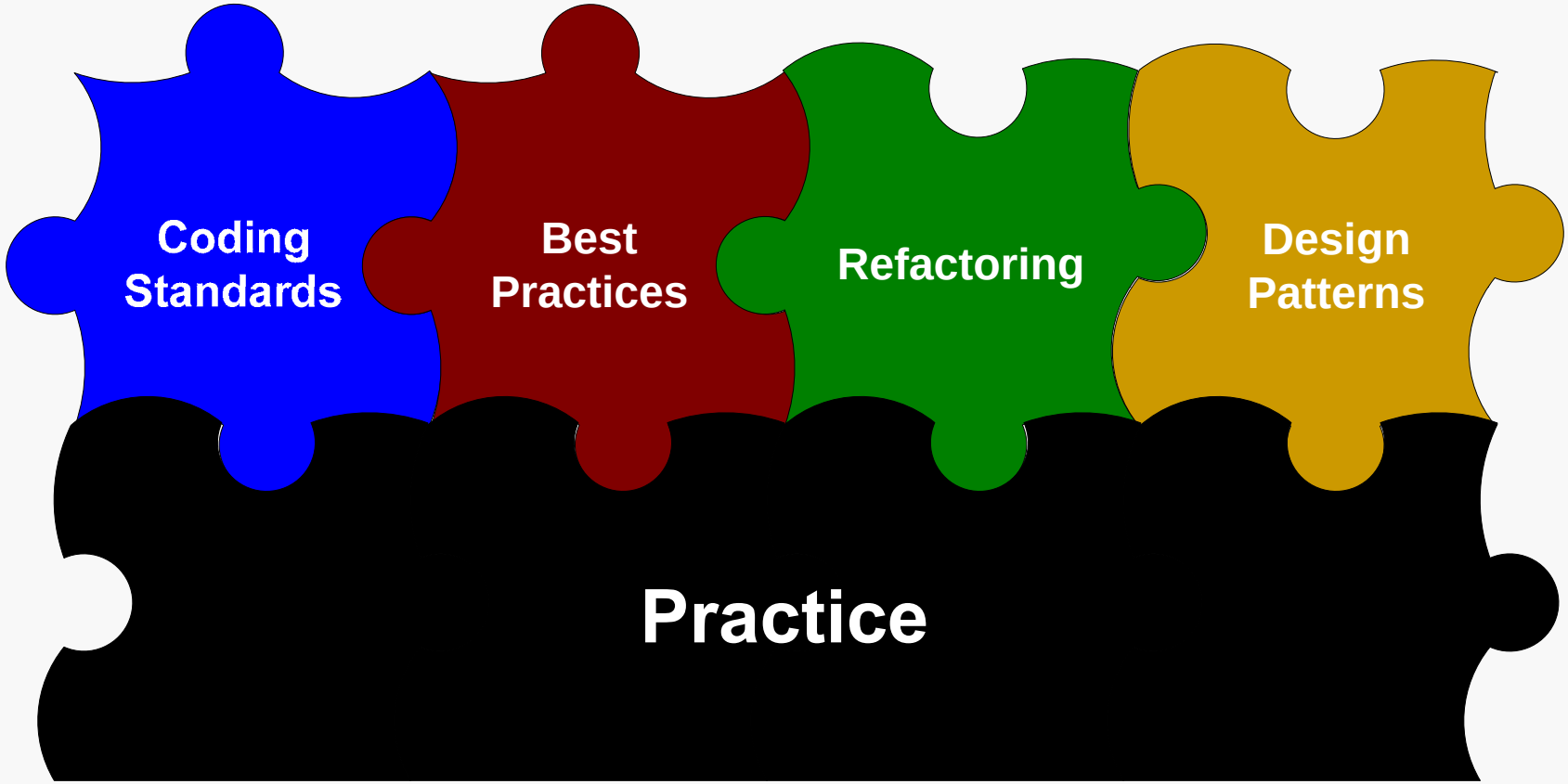


Balancing Planning and Agility

"Things should be as simple as possible, but no simpler."
Albert Einstein



Employ Good Coding Practices

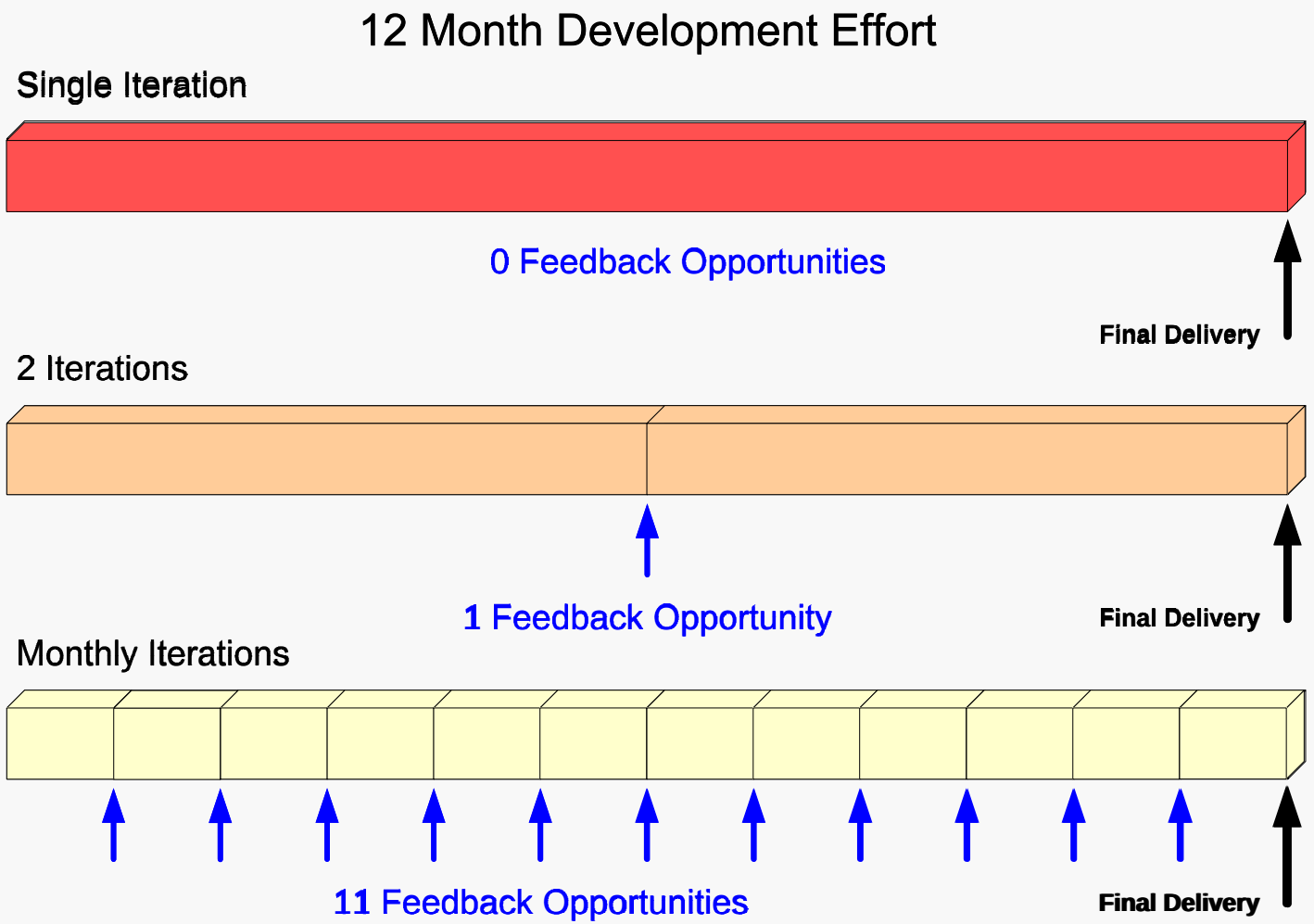


Practice 4: Short Iterations

"Short Iterations" is the use of *short development cycles* in iterative development.

- **Short iterations deliver functional software to the user at specific intervals, allowing the customer to evaluate the product and provide feedback.**
- **Short iterations support Lean development principles:**
  - Principle 1: Eliminate Waste
    - Short iterations reduce delays, one of "The 7 Wastes".
  - Principle 5: Deliver Fast
    - Short iterations put new functionality in the customers hands quickly and frequently.

Increasing Feedback Opportunities



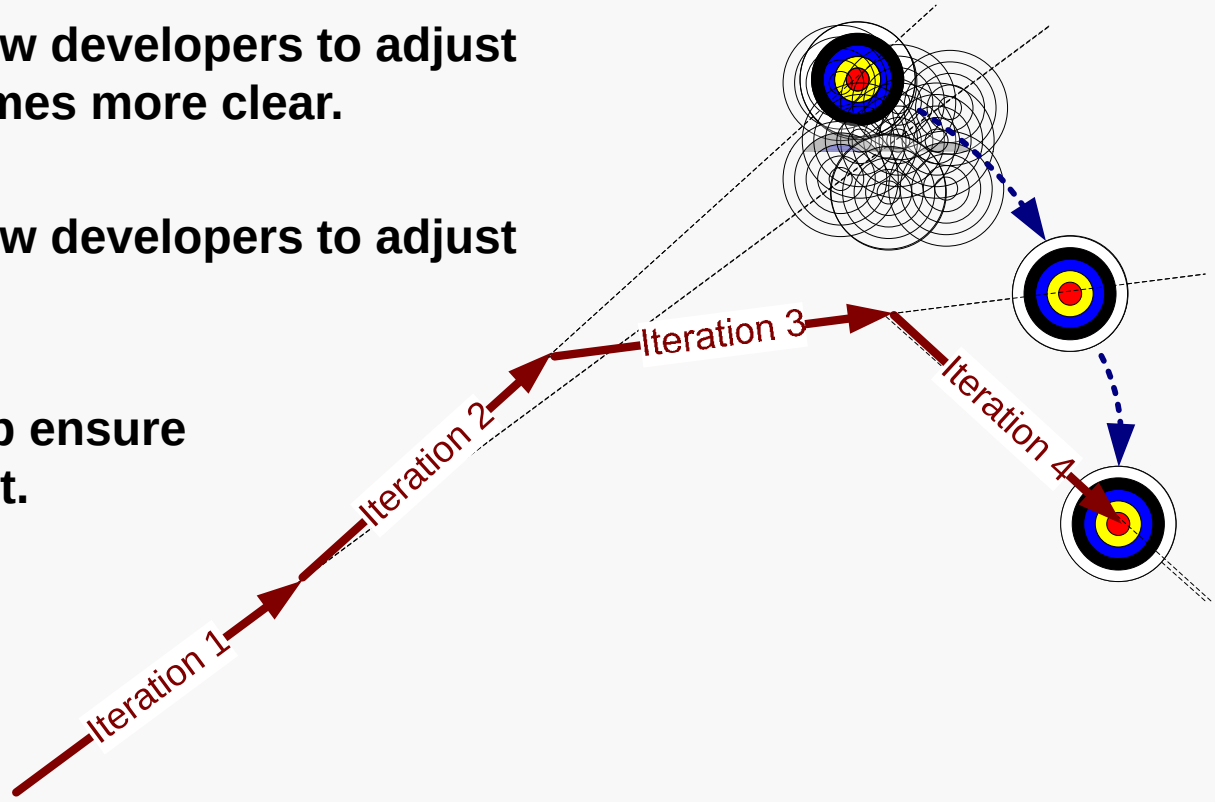
Course Corrections

Feedback opportunities become chances to correct the direction in which development is heading.

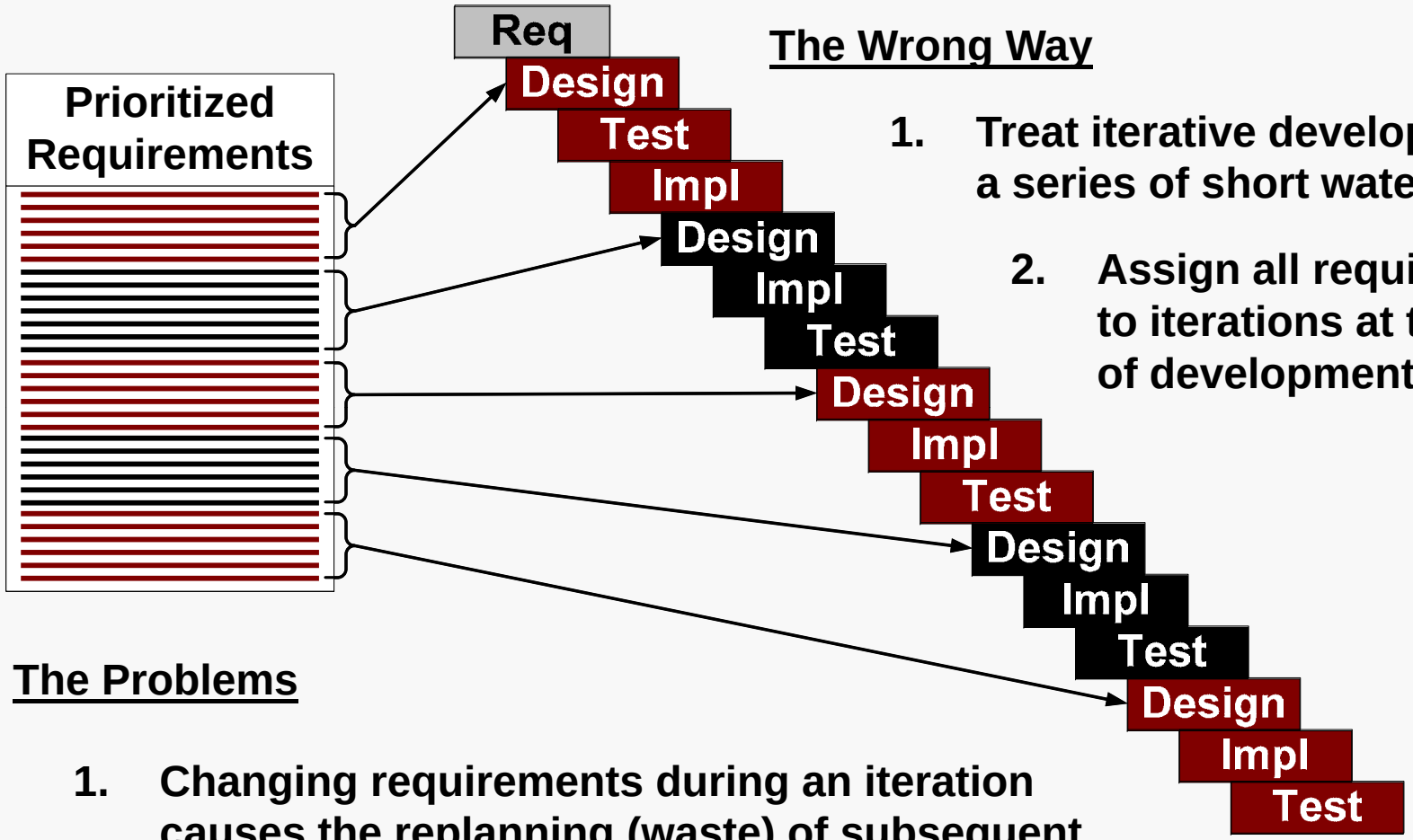
Course corrections allow developers to adjust focus as the goal becomes more clear.

Course corrections allow developers to adjust when the goal moves.

Course corrections help ensure the right product is built.



Iterative Development the Wrong Way



The Wrong Way

1. Treat iterative development as a series of short waterfalls.
2. Assign all requirements to iterations at the start of development.

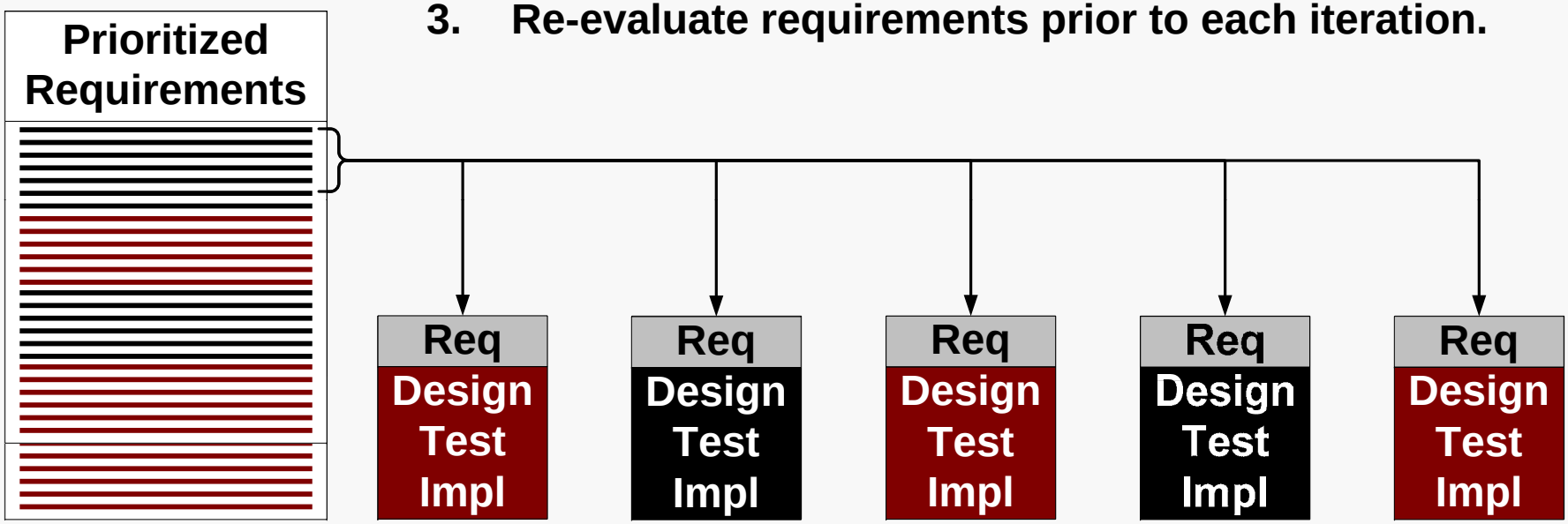
The Problems

1. Changing requirements during an iteration causes the replanning (waste) of subsequent iterations.
2. Failure to adjust to changes results in building the wrong product.

Iterative Development the Right Way

The Right Way

1. Prioritize requirements based on customer value.
2. Plan only the next iteration.
3. Re-evaluate requirements prior to each iteration.

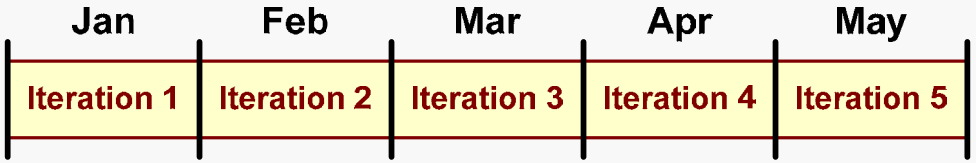


The Benefits

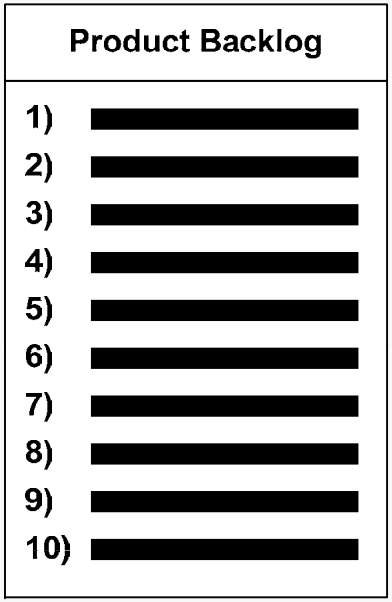
1. Each iteration implements requirements most important to the customer.
2. Requirements changes are accounted for in each iteration.
3. The RIGHT product is built.

Keys to Implementing Short Iterations

Set the Iteration Length and Stick to It



Work to Prioritized Requirements



Demo to the Customer

"Deliver" the Product

Practice 5: Customer Participation

Customer participation is about engaging the customer in *every* aspect of development.

- **Customer participation is a cornerstone of both Lean development and Agile methodologies:**
  - Lean Concept
    - Specify value in the eyes of the customer.
  - Agile Manifesto
    - Value customer collaboration over contract negotiation.

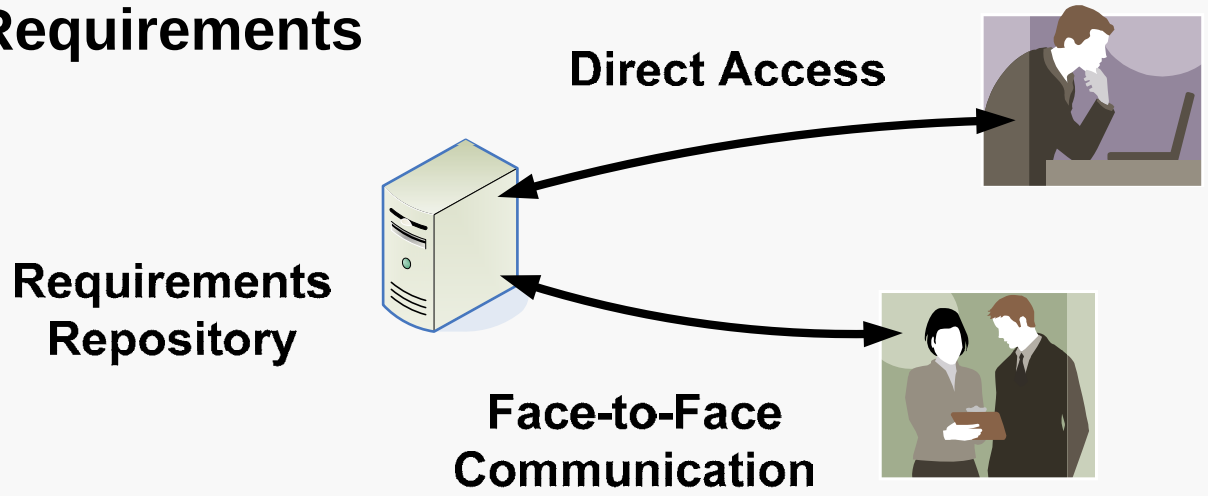
Customer Input is Essential to Success

Developers know technologies and software development, but that's not enough.

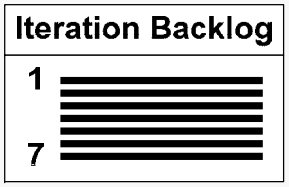
Customers know the business and the problems that need to be solved.

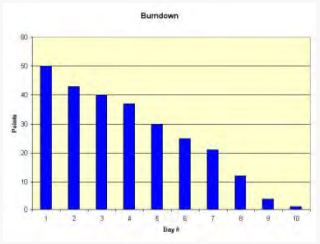
Enabling Customer Participation

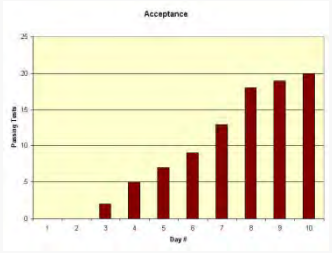
Provide Access to Requirements

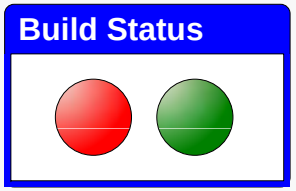


Provide Project Status



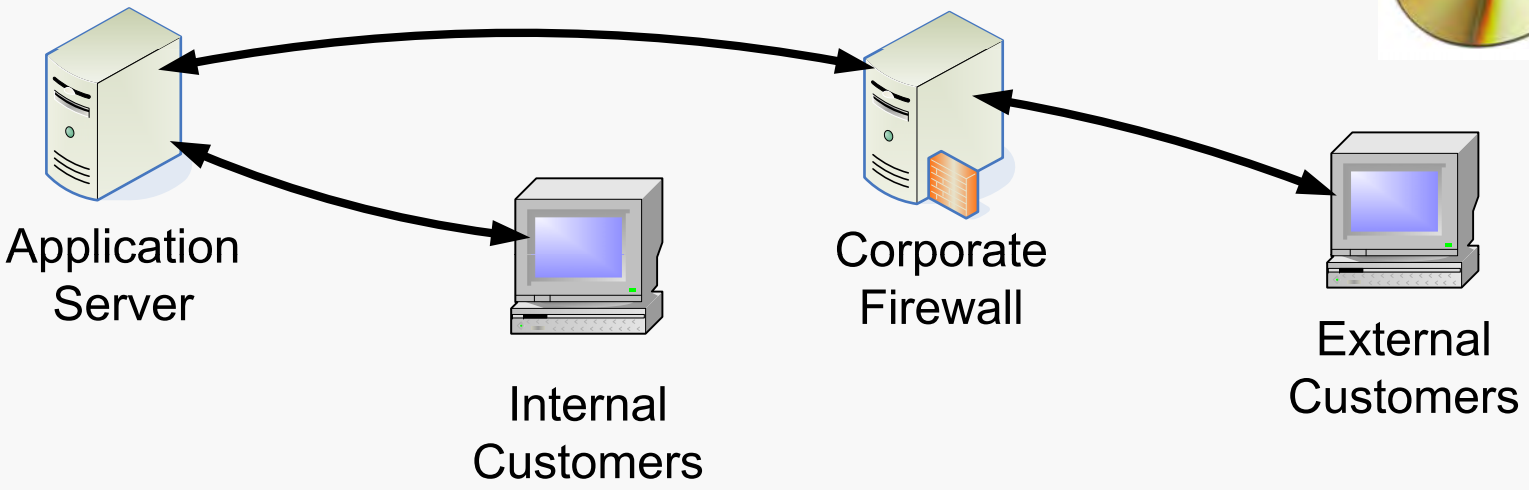






Enabling Customer Participation

Provide Access to the Product

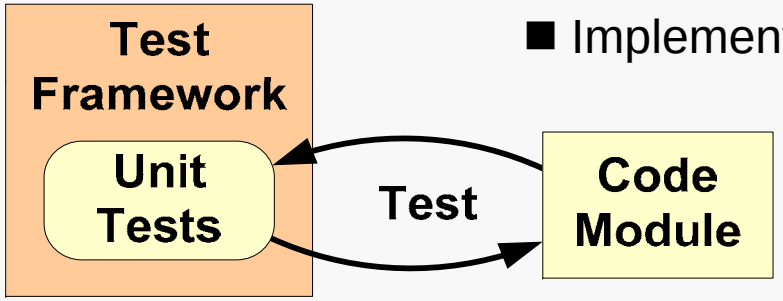


Provide a Feedback Loop

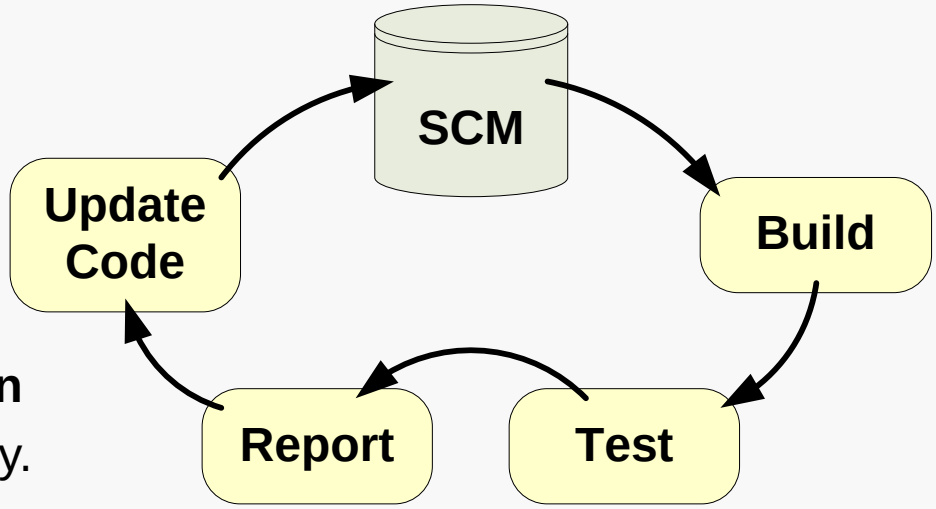
- **Formal**
  - Defect Tracking
  - Enhancement Requests
- **Informal**
  - Online Collaboration Tools, Wikis
  - Face-to-Face Meetings

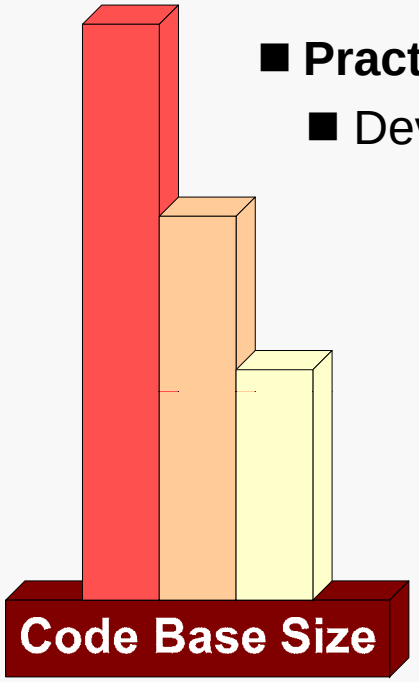
- **Practice 0: Source Code Management and Automated Builds**
  - Use best practices for Lean Software Development.

- **Practice 1: Automated Testing**
  - Implement reliable, repeatable test procedures.



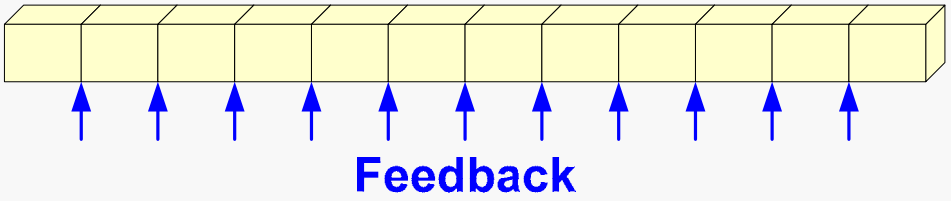
- **Practice 2: Continuous Integration**
  - Integrate small changes frequently.



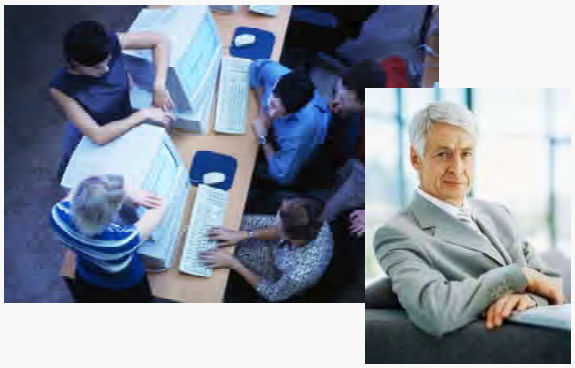


- **Practice 3: Less Code**
  - Developing unneeded code is wasteful.

- **Practice 4: Short Iterations**
  - Deliver fast and often to increase customer feedback.



- **Practice 5: Customer Participation**
  - Engaging the customer may be the most important practice of all.



Resources: Books

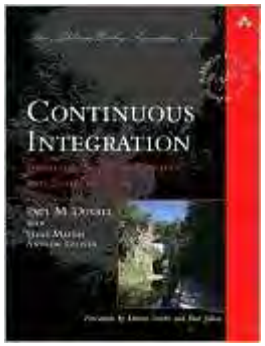
Implementing Lean Software Development: From Concept to Cash

by Mary Poppendieck; Tom Poppendieck

Agile and Iterative Development: A Manager's Guide

by Craig Larman

Resources: Books



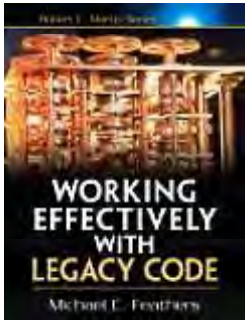
Continuous Integration:
Improving Software Quality and Reducing Risk

by Paul M. Duvall, Andrew Glover, Steve Matyas

xUnit Test Patterns:
Refactoring Test Code

by Gerard Mesaros

Resources: Books



Working Effectively with Legacy Code

by Michael Feathers

The Art of Lean Software Development:
A Practical and Incremental Approach

by Curt Hibbs, Steve Jewett and Mike Sullivan

Finis