Lean Software Development

- Lean Software Development is an Agile practice that is based on the principles of Lean Manufacturing

- Lean Software Development comes from the book "Lean Software Development: An Agile Toolkit" by Mary and Tom Poppendieck published in 2003

- Lean Software Development is based on 7 Principles and 22 Tools detailed in the book

- The fundamental principle of Lean Software Development is "Eliminate Waste", where waste is extra processes, defects, extra features, etc.

Lean Software Development

Chris Bubernak
Marc Schweikert

CSCI 5828
March 23, 2012

Agenda

- Lean History
- Who Uses Lean?
- 7 Principles
  - Eliminate Waste
  - Amplify Learning
  - Decide as Late as Possible
  - Deliver as Fast as Possible
  - Empower the Team
  - Build Integrity In
  - See the Whole
- References

Lean History (I)

- Lean is a translation of Lean manufacturing and IT practices into the software development domain
- Lean manufacturing itself is derived from the Toyota Production System (TPS)
- The term "Lean Software Development" comes from the book "Lean Software Development: An Agile Toolkit" written by Tom & Mary Poppendieck in 2003
- The book lays out 7 Principles and 22 Tools that encapsulate the Lean process

Lean History (II)

- The fundamental Lean principle is Eliminate Waste
- According to the father of the TPS, Taiichi Ohno, waste is defined as anything that does not produce value for the customer
- Designs and prototypes are not useful to the customer; they are only valuable when the new product is delivered

Who Uses Lean?

- Many software development companies employ agile development lifecycles and make use of Lean principles
  - Xerox
  - Rally Software
  - Patient Keeper
  - Helphire
  - Corbis

Eliminate Waste (I)

- Tool #1: Seeing Waste
  - Agile practices seek to eliminate waste, but the first step is to learn how to see waste
  - Good examples of waste are parts of the software process that are not analysis and coding - do they really add value to the customer?
  - Shigeo Shingo identified seven types of manufacturing waste
  - Mary and Tom Poppendieck translated these into the software domain in the following table

Eliminate Waste (II)

Manufacturing Waste	Software Development Waste
Inventory	Partially Done Work
Extra Processing	Extra Processes
Overproduction	Extra Features
Transportation	Task Switching
Waiting	Waiting
Motion	Motion
Defects	Defects

Eliminate Waste (III)

- Partially Done Work
  - Partially done software has a tendency to become obsolete
  - You have no idea if it will eventually work
  - You don't know if it will solve the business problem
  - Ties up resources in investments that have yet to produce results
  - Minimizing partially done software reduces risk and waste

Eliminate Waste (IV)

- Extra Processes
  - Paperwork consumes resources, slows down response time, hides quality problems, gets lost, becomes degraded and obsolete
  - Paperwork that nobody reads adds no value to the customer
  - Many development processes require paperwork for customer sign-off or to get approval for a change
  - Just because paperwork is required does not mean that it adds value
  - Good paperwork are documents that someone is waiting to be produced so they can do their job
  - A good alternative to writing requirements is writing customer tests (like Cucumber!)

Eliminate Waste (V)

- Extra Features
  - Developers sometimes like to add extra features to a system just in case they are needed
    - This seems harmless, but this is serious waste
  - All of the code has to be tracked, compiled, integrated, and tested
  - Every bit of code adds complexity and may become a failure point
  - The extra code may become obsolete before it is used
  - If the code isn't needed *now*, putting it in the system is waste

Eliminate Waste (VI)

- Task Switching
  - Assigning people to multiple projects is a source of waste
  - When a developer switches tasks, significant switching time is needed to gather his/her thoughts and get into the flow
  - The fastest way to complete two projects is to do them sequentially
  - Releasing too much work into a software development organization creates a lot of waste
  - Work progresses faster through a pipeline that is not filled to capacity

Eliminate Waste (VII)

- Waiting
  - A lot of time is wasted waiting for things to happen
    - Delays in starting a project
    - Delays in staffing
    - Delays due to excessive requirements documentation
    - Delays in reviews and approvals
    - Delays in testing
    - Delays in deployment
  - Delays keep the customer from seeing value as soon as possible

Eliminate Waste (VIII)

- Motion
  - How much time does it take to answer a question?
  - Are the right people staffed to answer technical questions?
  - Is the customer readily accessible to discuss features?
  - Development requires great concentration and chasing an answer to a question takes more time than you may think
  - Agile lifecycles generally promote a shared workspace so movement is minimized
  - Each handoff of design artifacts is an opportunity for waste
  - The biggest waste in document handoffs is that the document itself does not contain all of the information that the next person needs to know

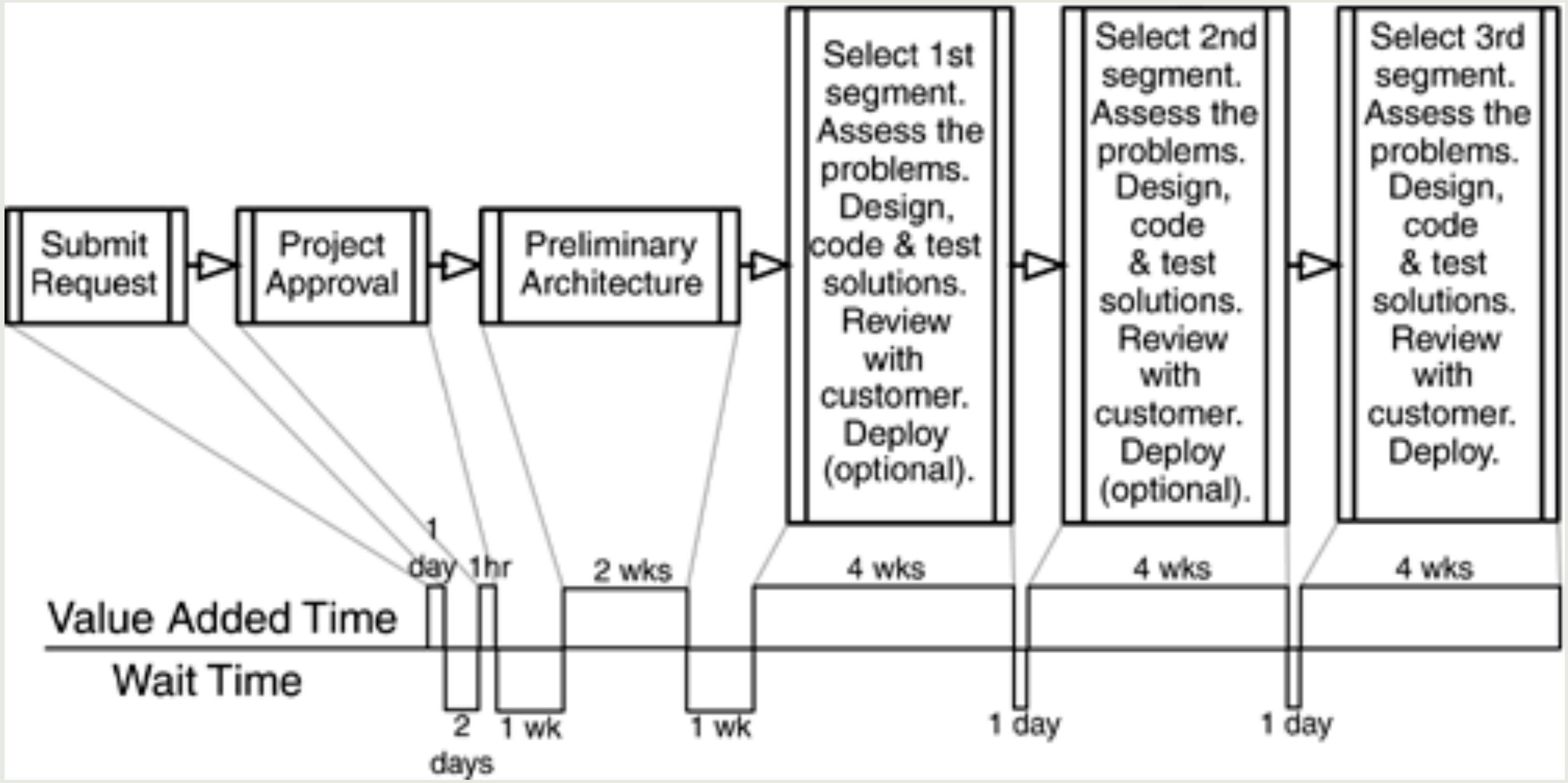
Eliminate Waste (IX)

- Defects
  - The amount of waste caused by a defect is determined by its impact and the amount of time it goes undetected
  - A critical defect detected in minutes is not a big waste
  - A minor defect not discovered for weeks is a much bigger waste
  - Strive to detect defects sooner through short iterations
- Management Activities
  - Management activities do not add value to a product, but they have a large impact on waste in the organization
  - If work is moved in a just-in-time manner then sophisticated project tracking systems are unnecessary

Eliminate Waste (X)

- Tool #2: Value Stream Mapping
  - Mapping your value stream is a good way to start discovering waste in your process
  - This involves drawing a chart of the average customer request, from arrival to completion
  - At the bottom, draw a timeline that shows how much time the request spends in value-adding, waiting, and non-value-adding activities
  - An agile value stream map follows on the next slide

Eliminate Waste (XI)


Submit Request
Project Approval
Preliminary Architecture
Select 1st segment. Assess the problems. Design, code & test solutions. Review with customer. Deploy (optional).
Select 2nd segment. Assess the problems. Design, code & test solutions. Review with customer. Deploy (optional).
Select 3rd segment. Assess the problems. Design, code & test solutions. Review with customer. Deploy.
1 day
1hr
2 wks
4 wks
4 wks
4 wks
Value Added Time
Wait Time
2 days
1 wk
1 wk
1 day
1 day

Amplify Learning (I)

- Tool #3: Feedback
  - Feedback adds complexity to a system, but it also adds considerable value
    - For example, a traffic light that senses cars is more useful than a traffic light on a strict timer
  - The Waterfall development model does not provide much feedback; it is generally thought of as a single-pass model
  - Traditional processes consider feedback loops threatening because incorporating feedback could modify the predetermined plan

Amplify Learning (II)

- Increasing feedback is the single most effective way to deal with troubled software projects
  - Instead of letting defects accumulate, run tests as soon as the code is written
  - Instead of adding more documentation or detailed planning, check out ideas by writing code
  - Instead of gathering more requirements, show the customer an assortment of potential users screens and get their input
  - Instead of studying which tool to use, bring the top three candidates in-house and test them
- Developers should know their immediate customer and have ways for that customer to provide feedback

Amplify Learning (III)

- Tool #4: Iterations
  - Iterations provide a dramatic increase in feedback
    - The operational environment is considered early
    - Design problems are exposed early
    - Change-tolerance is built into the system
  - Short iterations allow the system to respond to facts rather than forecasts
  - Iterations are a point of synchronization between the different teams and the customer
  - Iterations force decisions to be made because the system is deployed early and often

Amplify Learning (IV)

- Iteration Planning
  - The goal is to implement a coherent set of features each iteration
  - If a feature cannot be delivered in a single iteration, then it should be broken down into smaller features
  - If iterations are short and delivery is reliable, then the customer won't mind waiting until the next iteration for a feature

Amplify Learning (V)

- Team Commitment
  - The team must be small and have the necessary expertise
  - The team must have enough information about the features to decide how many features are in an iteration
  - The team must have the resources it needs
  - Team members have the freedom, support, and skill to figure out how to meet their commitments
  - The team must have a basic environment for good programming

Amplify Learning (VI)

- Convergence
  - There is reluctance to use iterations because there is a concern that the project will continue indefinitely
  - The iterative process limits customer requests to the beginning of each iteration and the team concentrates on delivering the features it committed to
- Negotiable Scope
  - A good strategy to achieve convergence is to work the top priority items first, leaving the low priority tasks to fall off the list
  - Let the customer only ask for their highest priority features, deliver quickly, and repeat will result in short lists of what's important

Amplify Learning (VII)

- Tool #5: Synchronization
  - Synch and Stabilize
    - Build the system every day after a small batch of work has been completed by each of the developers
    - At the end of the day, a build takes place followed by an automated set of tests
    - If the build works and the tests pass, then the developers have been synchronized
    - The more builds, the better because it provides rapid feedback
    - Full system tests should be run as frequently as possible

Amplify Learning (VIII)

- Spanning Application
  - A way to synchronize the work of several teams is by having a small advance team develop a simple spanning application through the system
  - If possible, the spanning application should go into production
  - When the spanning application is proven in production, you know you have a workable approach and multiple teams can use the same approach
  - Spanning applications can also be used to test various commercial components to select the best fit for your environment

Amplify Learning (IX)

- Matrix
  - A traditional approach to synchronizing multiple teams is to create an overall architecture and then have teams develop separate components
  - This approach is appropriate when the teams are not co-located
  - The problem with this, however, are the component interfaces
  - To combat this, the matrix approach starts by designing the interfaces and then moves to the components
  - This approach has the advantage that the highest risk areas are tackled at the beginning of the project when there is plenty of time and money and no code to change

Amplify Learning (X)

- Tool #6: Set-Based Development
  - Set-Based Versus Point-Based
    - The point-based approach starts with one choice and is refined until it works, which takes many iterations and might not converge
    - The set-based approach starts by defining everyone's constraints and then selects a choice that fits into those constraints
    - Set-based development bases communication on constraints rather than choices which requires significantly less data to convey far more information
    - Talking about constraints allows developers to defer making choices until the last possible moment

Amplify Learning (XI)

- Develop Multiple Options
  - When you have a difficult problem, develop a set of solutions, see how they actually work, and merge the best features of the solutions
  - It might seem wasteful to develop multiple solutions, but set-based development can lead to better solutions faster
- Communicate Constraints
  - Aggressive refactoring is the key to making sure that iterative development converges to a solution
  - Implementing a design that is available for refactoring is similar to a prototype in set-based development
- Let the Solution Emerge
  - When a group is tackling a difficult problem, resist the urge to jump to a solution; keep the constraints visible to explore the design space

Decide as Late as Possible (I)

- Tool #7: Options Thinking
  - People find it difficult to make irrevocable decisions when there is uncertainty present
  - Delaying Decisions
    - The underlying economic mechanism for controlling complexity in just-in-time systems is minimizing irreversible decisions
    - This contrasts with other methodologies that manage complexity by limiting decisions - e.g. "any color as long as it's black"
    - Delaying irreversible decisions leads to better decisions, limits risk, helps manage complexity, reduces waste, and makes customers happy
    - However, delaying decisions has a cost, but the overall system is more profitable and allows the correct decision to be made

Decide as Late as Possible (II)

- Options
  - The financial community has developed a mechanism - called options - which allows decisions to be delayed
  - An option is the right, but not obligation, to do something in the future
  - If things work out the way you expect you can exercise the option, and if they don't you can ignore the option
  - Financial options give the buyer an opportunity to capitalize on positive events in the future while limiting exposure to negative events
- Agile processes create options that allow decisions to be delayed until the customer needs are more clearly understood and the evolving technologies have had time to mature

Decide as Late as Possible (III)

- Tool #8: The Last Responsible Moment
  - The last responsible moment is the moment in which failing to make a decision eliminates an important alternative
  - If commitments are delayed beyond this moment then decisions are made by default, which is generally not a good approach to making decisions
  - This is not be be confused with procrastination; in fact, delaying decisions is hard work

Decide as Late as Possible (IV)

- Here are some tactics to delay decisions
  - Share partially complete design information
  - Organize for direct, worker-to-worker collaboration
  - Develop a sense of how to absorb changes
  - Develop a sense of what is critically important in the domain
  - Develop a sense of when decisions need to be made
  - Develop a quick response capability

Decide as Late as Possible (V)

- Tool #9: Making Decisions
  - Depth-first versus Breadth-first problem solving
    - Breadth-first problem solving could be thought of as a funnel and involves delaying decisions
    - Depth-first problem solving could be thought of as a tunnel and involves making early commitments
    - Depth-first is preferred when approaching new problems because it tends to quickly reduce the complexity of the problems
    - If there is an expert to make early decisions correctly and an assurance that there will not be changes that render the decisions obsolete, then depth-first is more effective. Otherwise, breadth-first will lead to better results

Decide as Late as Possible (VI)

- Intuitive Decision Making
  - Intuitive decision making relies on past experiences rather than rational thought to make decisions
  - Intuitive decision making paired with situational training is highly successful the vast majority of the time
  - Rational decision making involves decomposing a problem, removing the context, applying analytical techniques, and exposing the process and results for discussion
  - Rational decision making is useful in making incremental improvements, but suffers from "tunnel vision" - intentionally ignoring the instincts of experienced people

Deliver as Fast as Possible (I)

- Tool #10: Pull Systems
  - For rapid delivery to happen, it must be clear to every person what he/she should do to make the most effective contribution to the business
  - There are two ways to ensure that workers are productive: tell them what to do or set things up so they can figure it out for themselves
  - In fast-moving situations, it is more effective to let the workers task themselves

Deliver as Fast as Possible (II)

- Software Development Schedules
  - A project schedule is just as unreliable as a construction master schedule if it is used for fine-grained planning in an environment that experiences even a small amount of variability
  - An effective approach is to use a pull system that creates signaling and commitment mechanisms, so the team can figure out the most productive way to spend their time
- Software Pull Systems
  - The set of user stories are not assigned to developers; the developers choose the feature they want to work on
  - This allows the work to become self-directing

Deliver as Fast as Possible (III)

- Information Radiators
  - One of the features of a pull system is visual control, or management by sight
  - If work is to be self-directing, then everyone must be able to see what is going on, what needs to be done, what problems exist, and what progress is being made
  - Lists of problems, ideas for improvements, candidates for refactoring, business impact of the system to date, daily build status, and testing backlog are all candidates for entry onto a big, visible chart (information radiators)

Deliver as Fast as Possible (IV)

- Tool #11: Queueing Theory
  - Reduce Cycle Time
    - Queueing theory strives to make your wait as short as possible
    - The fundamental measurement of a queue is cycle time - the average time it takes something to get from the beginning of a process to the end
    - The time spent waiting in the queue is wasted time
    - There are two ways to reduce cycle time - look at the way work arrives or look at the way the work is processed
    - One way to control the rate of work arrival is to release small packages of work so that the queue is smaller

Deliver as Fast as Possible (V)

- Once variability has been removed from the arriving work, the next step is to remove the variability in the processing time
- Small work packages allow parallel processing of the small jobs by multiple teams so that if one team is stalled by a problem, the rest of the project can proceed without delay
- Software organizations, like highways, cannot function at full capacity
- Having slack in an organization gives the capacity to change, reinvent itself, and marshal resources for growth

Deliver as Fast as Possible (VI)

- Tool #12: Cost of Delay
  - Often, software teams are told that they must meet cost, feature, and date objections simultaneously. This sends two messages:
    - Support costs are not important because they were not mentioned
    - When something has to give, make your own trade-offs
  - Instead, give the team an economic model which will empower the members to figure out for themselves what is important for business
  - Trade-off decisions are easiest to make when they are expressed in the same units
  - Economic models justify the cost of reducing cycle time, eliminate bottlenecks, and purchasing tools

Empower the Team (I)

- Tool #13: Self-Determination
  - Create an environment in which capable workers can actively participate in running and improving their own work areas
    - Management's job is to coach and assist the team
    - Managers should not tell workers how to do their job, instead workers must show managers how to let them do their jobs
    - Feedback loops between managers and workers will let let managers better understand how the work should be done and drive improvement

Empower the Team (II)

- Conduct "Work-Outs"
  - Workers gather for 2-3 days to come up with proposals to help them better do their jobs
  - Before the Work-Out ends, the managers must make a yes/no decision on all proposals
  - All approved proposals are implemented immediately by those who proposed them
- Transferring processes from one environment to another is often a mistake
  - Instead, you need to understand the fundamental principles behind the practices
  - Then transform those principles into new practices for the new environment

Empower the Team (III)

- Tool #14: Motivation
  - Start with a clear and compelling purpose
    - All successful teams need someone to communicate a compelling purpose to the team
    - Team members who commit to a compelling purpose will bring passion to their work
  - Ensure the purpose is achievable
    - The fundamental rule of empowerment is to make sure the team has the capability of accomplishing its purpose
    - If a team commits to a business objective they should have all the resources necessary to complete it

Empower the Team (IV)

- Give the team access to customers
  - Talking to the actual customers gives the team a much better understanding of the purpose and what they are trying to accomplish
  - The project as a whole becomes more meaningful and they see how their work is going to impact real people
- Let the team makes its own commitments
  - At the beginning of every iteration teams should negotiate with customers to understand their priorities and select work for the iteration
  - The team should make decisions regarding how much work they will be able to accomplish
  - Team members understanding that when they commit to a set of features they are making a commitment to each other

Empower the Team (V)

- Management's role is to run interference
  - A highly motivated team does not need to be told what to do, but they may need to give their leaders some advice
  - Management may need to provide resources/protection
  - The team will maintain momentum if team members know someone is looking out for them
- Create sense of belonging
  - On a healthy team everyone understands what the goal is and be committed to it
  - Team members must understand that they succeed and fail as a group

Empower the Team (VI)

- "Safe" environment
  - One of the quickest ways to kill motivation is to employ a "zero defects mentality" which tolerates absolutely no mistakes
  - Mistakes will be made and team members need to have the confidence and support of team to understand that it is alright
- Desire to make progress
  - Teams will only succeed if members feel the need to make progress and have accomplished something
  - This reaffirms purpose and keeps everyone excited and motivated
  - The use of iterations helps force this by forcing the team to present their work to the customer on a regular basis

Empower the Team (VII)

- Tool #15: Leadership
  - Respected Leaders
    - Nearly every major product development is spearheaded by a *champion* who is excited and passionate about their work
    - They probably wrote the product concept, recruited the team, interpret the vision for the team, and set the pace for development
  - Master Developers
    - Exceptional developers exercise leadership through superior knowledge instead of bestowed authority
    - It is not essential to identify a master developer at the beginning of every project as more often than not they will emerge as a result of their technical prowess

Empower the Team (VIII)

- Where do master developers come from?
  - They grow into their role through extensive experience in the technology being employed or the domain being addressed by the system
  - They also possess exceptional abstraction and communication skills
- Project Management
  - Often the project manager does not have a technical background and is not responsible for developing a deep technical understanding of the product
  - They focus on identifying waste, coordinate iteration planning meetings, help team acquire resources, coordinate/ synchronize multiple teams, and provide a motivating environment (among many other things)

Empower the Team (IX)

- Tool #16: Expertise
  - Share Expertise
    - Promote mentorship and pair programming
    - Create communities of expertise
      - Identify the technical and domain specific competencies that are critical to an organization's success
      - Within these communities create forums for knowledge sharing; monthly meetings, newsletters, guest speakers, etc.
  - Enforce Standards
    - Naming standards, coding standards, language standards, check in/out standards, build standards
    - Develop standards and practice them
    - Lack of standards leads to sloppy work
    - Standards develop through experimentation and knowledge sharing

Build Integrity In (I)

- Tool #17: Perceived Integrity
  - Focus on keeping customer values at the forefront
    - Engineers have a tendency to get lost in the details and lose track of customer values
    - To prevent this, visions of perceived integrity should be refreshed regularly through customer feedback or access to groups of users who can judge the system's integrity
  - Model Driven Design
    - Construct domain models such that software implementation can flow directly from these models
    - These models must be understood and directly usable by the customers and developers
    - The models should express all the business rules, business processes, and domain related issues using ubiquitous language that can be understood by everyone

Build Integrity In (II)

- Recommended collection of Models
  - Conceptual Domain Model: Class model of the basic entities of the system
  - Glossary: Defines terms found in domain model and ensures consistent language for the team
  - Use Case Model: A dynamic view of the system that captures customers goals for interacting with the domain model and details their expected workflow
  - Qualifiers: Details the the qualifiers that might be applied to the basic functionality of the system such as number of users, acceptable response time, required availability, projections for growth, etc.

Build Integrity In (III)

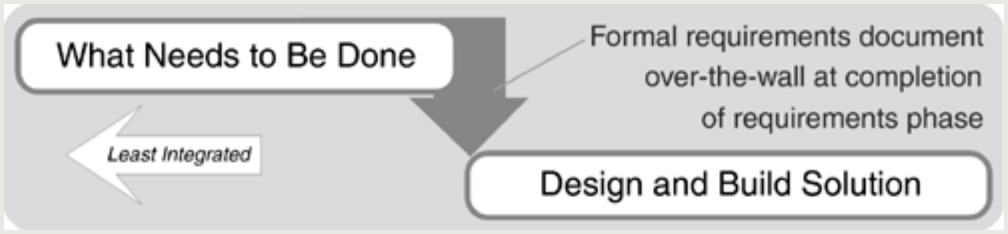
- Tool #18: Conceptual Integrity
  - Measured by how well a system's components work together as a smooth, cohesive whole
    - Components match/work well together
    - The architecture allows for a balance between flexibility, maintainability, efficiency, and responsiveness
  - Effective communication is key to achieving conceptual integrity is
  - Reuse existing solutions
    - If something has been proven to solve a problem in the past and you can make use of it, do it
    - This helps to remove degrees of freedom in the development process, reduces the complexity and need for communication

Build Integrity In (IV)

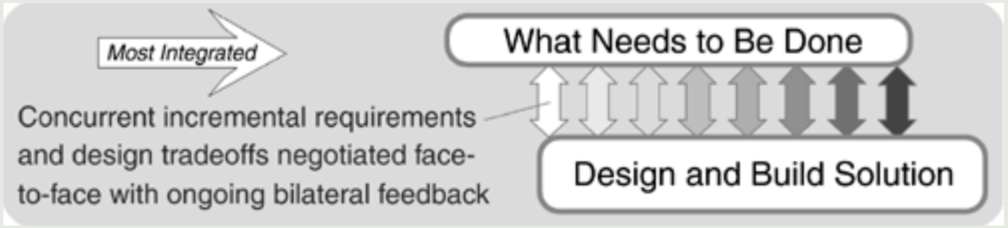
- Employ integrated problem solving practices
  - Understand and store the problem simultaneously
  - Release preliminary data early and don't wait for the complete answer
  - Information is released in small batches
  - Information flows in two directions
  - The preferred method of communication is face-to-face

Build Integrity In (V)

- Without integrated problem solving
  - Designers make decisions in isolation and send a large amount of information to those who must make decisions causing a "throw it at the wall approach"


What Needs to Be Done
Formal requirements document over-the-wall at completion of requirements phase
Least Integrated
Design and Build Solution

- With integrated problem solving
  - Frequent bilateral communication occurs which produces necessary feedback


Most Integrated
What Needs to Be Done
Concurrent incremental requirements and design tradeoffs negotiated face-to-face with ongoing bilateral feedback
Design and Build Solution

Build Integrity In (VI)

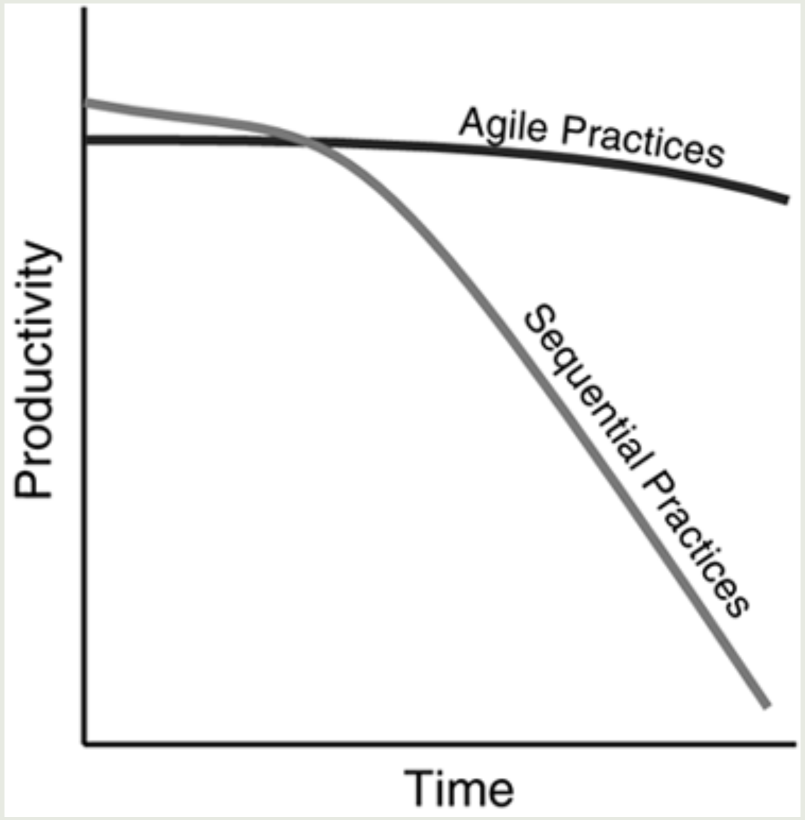
- Tool #19: Refactoring
    - Complex systems have effects that aren't fully understood at design time
    - Architecture must remain healthy as the system evolves
        - New features are always being requested by users
        - Features can always be added separately; often new features are related and redesigning architecture of system to support feature set is the best decision
        - Resist the urge of adding features in a brute force approach and instead adjust the architecture so that system integrity does not degrade

Build Integrity In (VII)

- Maintaining conceptual integrity
  - Simplicity
    - A simple design is nearly always the best design
    - Experienced developers know how to simplify complex designs
  - Clarity
    - Code must be easy to understand
    - Everything should be clearly named and naming conventions and practices should be upheld
  - Suitability for use
    - Every design must accomplish intended purpose
  - No repetition
    - Repetition should never exist, this is a sign of bad design
  - No extra features
    - When Code is no longer needed, get rid of it!

Build Integrity In (VIII)

- Does refactoring slow down productivity?
  - Conventional wisdom suggests that the overhead associated with refactoring would slow down the team - the opposite turns out to be true
  - By spending time making sure code does not get out of hand you make further development much easier and more efficient



Build Integrity In (IX)

- Tool #20: Testing
  - Testing proves that design intent is achieved and that the system does what customers want it to do
    - Whenever developers write code, tests should exist to ensure that the features being developed work as intended
  - When developers write code he/she should get immediate feedback about whether or not it works as intended from the existing test code
  - Tests should be automated as much as possible and run as part of the daily build
  - Tests that cannot be automated or take too long to be run daily have much less value than those that can be run frequently

Build Integrity In (X)

- Types of tests
  - Unit tests (single component), integration tests (multiple components) and system tests (the entire system)
  - Developer tests (tests that ensure code does what developer intended) and acceptance tests (also known as customer tests these tests ensure that the system functions how the customer expects)
- Roles of tests in development life cycle
  - Communicate how things are supposed to work
  - Provide feedback on whether or not the system actually works how it is supposed to
  - Provide the scaffolding for developers to make changes through the development process
  - After development tests provide a an accurate representation of how the system was actually built

See the Whole (I)

- Tool #21: Measurements
  - Measurements are important for tracking the progress of software development
    - Example: defect counts are important for gauging software readiness
  - People will increase their performance in areas that are measured
  - Measurements should motivate the team to collaborate and find better ways to do things
  - Information measurements, not performance measurements, should be used for tracking progress
    - If defects are attributed to individuals (performance measurement) we assume individuals are solely responsible for defects and can miss deeper underlying issues

See the Whole (II)

    - If defects are attributed to features (information measurement) underlying problems can be surfaced much easier
- Try to create measurements that will measure everything
    - Standardize
        - Abstract the development process into sequential phases
        - Standardize each phase
        - Measure conformance to the process
    - Specify
        - Create a detailed specification or plan
        - Measure performance against the plan
        - Find variation from the plan
    - Decompose
        - Break big tasks into little tasks
        - Measure each individual task

See the Whole (III)

- Tool #22: Contracts
  - A common misconception is that agile development cannot be used in the context contract work in which each firm is expected to look out for itself
  - To make it work, we need to change our idea of a contract being method of keeping two parties from taking advantage of each other into something that supports collaboration
  - In most cases this means being flexible with regards to scope

See the Whole (IV)

- Incorporating Lean Development into different types of contracts
  - Fixed-Price Contracts
    - This type of contract does not lend itself well to Lean Development unless the customer understands that when risk arises they must take responsibility for it and possibly incur further costs
  - Time-and-Materials Contracts
    - Use concurrent development with highest priority features implemented first and working, integrated code delivered at each iteration so that customer may easily manage cost by limiting scope
  - Multistage Contracts
    - Use a master contract and work orders to release each iteration, emphasize concurrent development of high priority features first and working, integrated code at each iteration

See the Whole (V)

- Target-Cost Contracts
  - Encourage the frontline workers of both parties to work together to come up with a solution to the problem that meets a target cost, giving them the freedom to limit scope as a primary mechanism to achieve target cost
- Target-Schedule Contracts
  - These should be avoided, however if necessary the team should be allowed to add resources or license components as necessary to ensure that they stay on schedule (which will incur further costs for the customer)
- Share-Benefit Contacts
  - Assume that the parties will modify what they are doing as time goes on to achieve mutual benefit

References

- Mary, Tom Poppendieck (2003), "Lean Software Development: An Agile Toolkit", Addison-Wesley Professional, ISBN 0321150783
- "Lean Manufacturing". Wikipedia. Wikimedia Foundation, 18 Mar. 2012 <http://en.wikipedia.org/wiki/Lean_manufacturing>
- *Lean Software Development*. Poppendieck.LLC. Web. 18 Mar. 2012 <http://www.poppendieck.com>