Lean Software Development

Dr. Christoph Steindl
Senior IT Architect and Agile Method Exponent
Certified ScrumMaster
Christoph_Steindl@at.ibm.com
Agenda

- History of Lean Manufacturing and Lean Thinking
- 7 Principles and 22 Tools (abbreviated)
- Summary
History – Rigid or Lean Processes

- The 1990s were the decade of Process for IT. We prostrated ourselves before the CMM and ISO. It wasn't enough just to do things right, we also had to say in advance exactly what we intended to do and then do exactly that ("plan the work, and work the plan")*.  

- The list of companies most successful at climbing up the CMM ladder early in the 1990s reads like a Who’s Who of downsizing by the end of the 1990s.

- Process rigor was simply not the right recipe for an era in which everything was changing.

- Today the era of fat process is over. As Jim Highsmith has said, “Thin is in.”

* see foreword by Tom DeMarco to the book “Agile Software Development Ecosystems” by Jim Highsmith, Addison-Wesley, 2002.
History of Lean Thinking and Lean Software Development

- On the other hand, Toyota has started in the 1980s to revolutionize the automobile industry with their approach of "Lean Manufacturing"
  - to eliminate waste
  - to streamline the value chain (even across enterprises)
  - to produce on request (just in time), and
  - to focus on the people who add value.

- Lean Thinking* capitalizes on the intelligence of frontline workers, believing that they are the ones who should determine and continually improve the way they do their jobs.

- Mary and Tom Poppendieck have transferred principles and practices from the manufacturing environment to the software development environment**.

* see the book "Lean Thinking" by James P. Womack and Daniel T. Jones
** see the article "Lean Software Development" and the book "Lean Software Development: An Agile Toolkit" by Mary and Tom Poppendieck
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.

* Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, Dave Thomas, 2001.
LSD with 7 Principles and 22 Tools

- Eliminate Waste
  - Seeing Waste, Value Stream Mapping
- Amplify Learning
  - Feedback, Iterations, Synchronization, Set-Based Development
- Decide as Late as Possible
  - Options Thinking, The Last Responsible Moment, Making Decisions
- Deliver as Fast as Possible
  - Pull Systems, Queuing Theory, Cost of Delay
- Empower the Team
  - Self-Determination, Motivation, Leadership, Expertise
- Build Integrity In
  - Perceived Integrity, Conceptual Integrity, Refactoring, Testing
- See the Whole
  - Measurements, Contracts
Principle #1: Eliminate Waste

... does not mean to throw away all documentation, but to spend time only on what adds real customer value.

- Eliminating waste is the most fundamental lean principle, the one from which all the other principles follow.
- Thus, the first step to implementing lean development is learning to see waste.
- The second step is to uncover the biggest sources of waste and eliminate them.
Features & Function Usage

Eliminate Waste by Seeing Waste (Tool #1) (1/2)

In manufacturing (software development), there are seven kinds of waste:

1. Inventory (partially done work in software development)
   Inventory has a tendency to become obsolete. You might have no idea whether or not it will eventually work or be needed until the software is actually in production; you don't really know if it will solve the business problem. What if the system never makes it into production?

2. Extra processing (extra processes)
   Paperwork consumes resources, slows down response time, hides quality problems, gets lost, degrades and becomes obsolete. Just because paperwork is a required deliverable does not mean that it adds value. If you must produce paperwork, keep it short and keep it high level.

3. Over production (extra features)
   Every bit of code in the system has to be tracked, compiled, integrated, and tested every time the code is touched, and then it has to be maintained for the life of the system. Every bit of code increases complexity and is a potential failure point. There is a great possibility that extra code will become obsolete before it’s used.

4. Transportation (task switching)
   Assigning people to multiple projects is a source of waste. Every time software developers switch between tasks, a significant switching time is incurred as they get their thoughts gathered and get into the flow of the new task. Belonging to multiple teams usually causes more interruptions and thus more task switching. This task switching time is waste.
Eliminate Waste by Seeing Waste (Tool #1) (2/2)

In software development, there are seven kinds of waste (continued):

5. **Waiting**
   One of the biggest wastes in software development is usually 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, and delays in deployment are waste.

6. **Motion**
   When a developer has a question, how much motion does it take to find out an answer? People aren’t the only things that move – various artifacts move also. Requirements may move from analysts to designers, and then design documents move from designers to programmers, and then code moves from coders to tests, and so on. Each handoff of an artifact is fraught with opportunities for waste, great amounts of tacit knowledge remain with the creator of the document and never get handed off.

7. **Defects**
   The amount of waste caused by a defect is the product of the defect impact and the time it goes undetected. Find them as soon as they occur, test immediately, integrate often, and release to production as soon as possible.
Eliminate Waste by Value Stream Mapping (Tool #2)

Mapping your value stream is a good way to start discovering the waste in your software development process. It focuses attention on products and their value to customers rather than on organizations, assets, technologies, processes and career paths.

Create a value stream map with paper and pencil by walking around your organization:

- Pretend you are a customer request and imagine yourself going through each step of your process. Don't ask people what happens; walk around, look at the data, find out for yourself.
- Go to the place where a customer request comes into your organization. Draw a chart of the average customer request, from arrival to completion.
- Working with the people involved in each activity, sketch all the process steps necessary to fill the request, as well as the average amount of time that the request spends in each step.
- At the bottom of the map, draw a timeline that shows how much time the request spends in value-adding activities and how much time it spends in waiting states and non-value adding activities.

A value stream map provides a starting point for evaluating and improving your software development process: Pick the biggest opportunities to increase flow and value-added time.

Once you have a value stream map of your organization, the next step is to extend it to your customers.
Agile Value Stream Maps

Value stream maps often show that nondevelopment activities are the biggest bottlenecks in a software development value stream. Figure 1.4 is an example of a value stream map that Kent Beck posted on the discussion group Software-in-Process. It shows that the biggest delays in this particular organization come after development and testing are complete.

Figure 1.4 Kent Beck's value stream map.

“It took us about a half hour to come up with this (don’t spend longer than that or you’ll have too much detail).”

— from Kent Beck
Principle #1: Eliminate Waste – Do It Yourself

1. Make a list of the 10 or 15 most important activities in your organization. Put yourself in the shoes of a customer and rate each item from 1 to 5, with 1 meaning customers probably don’t care about the activity and 5 meaning customers value it highly. Think of the low-scoring activities as waste. Take the two lowest scoring items and develop a plan to cut the time on these activities in half.

2. At your next seven team meetings, take some time to discuss each of the seven wastes of software development, one at a time:
   - Partially done work
   - Extra processes
   - Extra features
   - Task switching
   - Waiting
   - Motion
   - Defects
   For each waste, ask the questions
   - Do you agree that this “waste” is really a waste? Why or why not?
   - Whether or not you agree that the item is a waste, estimate how much time it consumes in an average week.
   - What can or should be done to reduce that time?

3. Develop a value stream map for your organization. Start with an incoming request and map a timeline of its progress to providing customer value. Find out how much of the time is spent adding value and how much is spent waiting. Take the biggest cause of delay and develop a plan to cut it in half.
Principle #2: Amplify Learning

… does not mean to keep on changing your mind, but to increase feedback, when you have tough problems.

- When an organization has software development challenges, there is a tendency to impose a more disciplined process on the organization, with more rigorous sequential processing
  - where requirements are documented more completely,
  - where all agreements with the customer are written,
  - where changes are controlled more carefully,
  - where each requirement must be traced to code,
  - where additional deterministic controls on a dynamic environment lengthen the feedback loop.

- Just as control theory predicts, this generally makes a bad situation worse.
Amplify Learning with Feedback (Tool #3)

- When a problem develops…
  - the first thing to do is to make sure the feedback loops are all in place,
  - the next thing to do is to increase the frequency of the feedback loops in the problem areas.
- Increasing feedback is the single most effective way to deal with troubled software development projects and environments.
  - Instead of gathering more requirements from users, show them an assortment of potential user screens and get their input.
  - Instead of letting defects accumulate, run tests as soon as the code is written.
  - Instead of adding more documentation or detailed planning, try checking out ideas by writing code.
  - Instead of studying more carefully which tool to use, bring the top three candidates inhouse and test them.
Amplify Learning with Iterations (Tool #4)

- A universal starting point for all agile software development approaches is **iterations**: short feedback loops enhance control; iterations are points of synchronization (the team and the customer see what has been accomplished); iterations force decisions to be made.
Advantages of Rapid Cycle Times

1. Increases learning tremendously.
2. Eliminates buggy software because you die if you don't fix this.
3. Forces complete Scrum automation which allows real time, rollup reporting across all products and releases.
4. Forces Scrum of Scrums to meet daily.
5. Forces weekly MetaScrum meeting to plan and coordinate product releases across the company.
6. Fixes the install process because you die if you have to install 45 releases this year and install is not easy.
7. Improves the upgrade process because there is a constant flow of upgrades that are mandatory. Makes upgrades easy.
8. Forces quick standardization of software via new features rather than customization and one off. This is the core of Oracle's current eBusiness strategy that Larry Ellison is using against PeopleSoft and SAP.
9. Forces implementation of sustainable pace, a basic Agile principle. You die a death of attrition without it.
10. Allows waiting to build new functionality until there are 4-5 customers who pay for it. This is counterintuitive, and caused by the fact that you can tell the customer it will be ready by the time they sign the contract and put together their installation team (since everything is ready within 90 days). If it is too big to build in 90 days, you give them a 90 day go-live release with top priority functionality and monthly upgrades thereafter until they have what they want.

Jeff Sutherland on scrumdevelopment@yahoogroups.com 23.11.2004
Amplify Learning with Synchronization (Tool #5)

Whenever several individuals are working on the same thing, a need for synchronization occurs. The need for synchronization is fundamental to any complex development process.

Synchronize / integrate technically:
- Integrate daily within a team (i.e. check-in at least daily into local repository)
- Integrate weekly across multiple teams (i.e. check-in at least weekly into the central repository)

More frequent builds are better; they provide much more rapid feedback.

Builds and build tests should be automated.
Development in Cycles

**Project Cycle (the cooperative game)**
- Engage
- Charter
- Wrap up and harvest
- Deliver

**Delivery Cycle [1w - 3 mo]**
- Complete (celebrate & reflect)
- Deliver to real users
- Calibrate release plan
- Iterate

**Iteration [1w - 3 mo]**
- Estimate & Complete plan
- Calibrate release plan
- Develop
- Iterate

**Integration Cycle [30’ – 3d]**
- Develop
- Perform system test
- Integrate
- Across teams
- Within team

**Developm. Episode [mins - hours]**
- Design
- Check in
- Code & test
- Within team
- Across teams

**Maintenance Cycle**
- Report bug
- Accept bug fix
- Integrate

**Day**
- Stand-up meeting

**Week**
- Deliver to real users
- Accept bug fix
- Integrate

**Month**
- Deliver to real users
- Accept bug fix
- Integrate

**Year**
- Deliver to real users
- Accept bug fix
- Integrate

At least twice At least once
Amplify Learning with Set-Based Development (Tool #6)

In set-based development, communication is about constraints, not choices. This turns out to be a very powerful form of communication, requiring significantly less data to convey far more information. In addition, talking about constraints instead of choices defers making choices until they have to be made.

Develop multiple options, communicate constraints, and let solutions emerge.

When you have a difficult problem,
- Develop a set of alternative solutions to the problem
- See how well they actually work, and
- Then merge the best features of the solutions or choose one of the alternatives.

Set-based development can lead to better solutions faster.

- Initial situation with problem
- Team develops alternatives.
- Customer selects alternative.
- Possible solution with consequences … to business drivers and constraints …
- Possible solution with consequences … to business drivers and constraints …
- Possible solution with consequences … to business drivers and constraints …
Principle #2: Amplify Learning – Do It Yourself (1/2)

1. Take your most difficult problem and devise a way to increase feedback.
   - Increase the feedback of development teams to management by asking each team at the end of each iteration the following questions:
     - Was the team properly staffed for this iteration?
     - Were there any needed resources that were not forthcoming?
     - How can things be changed to make things go better or faster?
     - What is getting in the way?
   - Increase the feedback of customers to development teams by holding a customer focus group at the end of each iteration. Ask questions such as the following:
     - How well does this section solve the problem it was meant to solve?
     - How could it be improved?
     - How does this iteration affect your view of what you need?
     - What do you need to put this part of the system into production?
   - Increase the feedback of the product to the development team in the following ways:
     - Have developers write and run developer tests as they write the code.
     - Have analysts, customers, or testers write and run customer tests as the developers work on the code. Have developers help with the customer tests if that’s what it takes to get them automated.
     - Have developers observe usability tests of each features as it nears completion, so they can see how users react to their implementation.
   - Increase the feedback within the team in the following ways:
     - Make testers an integral part of the development team.
     - Involve operations people at the beginning of the project.
     - Establish the policy that the development team maintains the product.
Principle #2: Amplify Learning – Do It Yourself (2/2)

2. Start iterations with a negotiation session between customers and developers. Customers should indicate which features are the highest priority, and developers should select and commit to only those features from the top of the priority list which they can realistically expect to complete in the iteration time-box.

3. Post a progress chart for your current project in a common area so the team can see what needs to be done and everyone can see how the project is converging.

4. If you divide a system across multiple teams, make every effort to have a divisible architecture that allows teams to work on their own areas as independently as possible. Find ways for multiple teams to synchronize as often as possible by integrating their code and running automated tests.

5. If strata teams work for machine interfaces, consider them for user interfaces also. If you have several teams working on different components of a system, consider forming strata teams focused on user interfaces that cross components.

6. Find your toughest outstanding development problem and have the development team come up with three options on how to solve it. Instead of choosing one of the solutions, have the team explore all three options at the same time.
Principle #3: Decide as Late as Possible

… does not mean to procrastinate, but to keep your options open as long as practical, but no longer.

- Establishing requirements before development begins is commonly thought of as a way to protect against serious errors. The problem with sequential development is that it forces designers to take a depth-first rather than a breadth-first approach.

- The easiest way to make big mistakes is to drill down to detail too fast.
- When big mistakes may be made, it is best to survey the landscape and delay the detailed decisions.

- With concurrent development, you start programming the highest value features as soon as a high-level conceptual design is determined, even while detailed requirements are being investigated. That exploratory approach permits you to learn by trying a variety of options before you lock in on a direction that constrains implementation of less important features.

- Concurrent development requires developers with enough expertise in the domain to anticipate where the emerging design is likely to lead and close collaboration with the customers and analysts who are designing how the system will solve the business problem at hand.
Decide as Late as Possible with Options Thinking (Tool #7)

- We usually don't give our customers the option to change their minds. And yet, almost everyone resists making irrevocable decisions in the face of uncertainty. Options allow fact-based decisions based on learning rather than speculation.

- Premature design commitment restricts learning, exacerbates the impact of defects, limits the usefulness of the product, and increases the cost of change.

- Creating detailed plans early means carving some decisions into stone. Plans and predictions are not bad, but making irrevocable decisions based on speculation is to be avoided. Rather develop options, communicate them and decide together with the customer.

- But, options are not free and it takes expertise to know which options to keep open.
Decide as Late as Possible with The Last Responsible Moment (Tool #8)
Concurrent development makes it possible to delay commitment until the last responsible moment, that is, the moment at which failing to make a decision eliminates an important alternative.

Share partially complete work: requiring complete information before releasing work increases the length of the feedback loop and causes irreversible decisions to be made sooner than necessary. Good work is a discovery process, done through short, repeated exploratory cycles.

Develop a sense of how to absorb changes (use components with interfaces, use abstraction, avoid repetition, separate concerns, encapsulate variation, avoid extra features).
Develop a sense what is critically important in the domain.
Develop a sense when decisions must be made.
Decide as Late as Possible with Making Decisions (Tool #9)

- **Intuitive** decision making is the more mature approach than **rational** decision making, and it usually leads to better decisions as well.
- Rational decision making involves decomposing a problem, removing the context, applying analytical techniques, and exposing the process and results for discussion. It suffers from tunnel vision, intentionally ignoring the instincts of experienced people. Rational analysis is unlikely to detect high-stakes mistakes.
- It is much more important to develop people with the expertise to make wise decisions than it is to develop decision-making processes that purportedly think for people.

Let me give you an example from a different area:

- **Marines** plan, but they don't predict.
  - They focus on understanding the essence of the situation and the strengths and weaknesses of both sides.
  - They find simplifying assumptions, boundary conditions, and alternative approaches.
  - They settle on an approach with a 70% chance of success.
  - They search for what is being overlooked; they invite dissent.
- **Marines** manage by end state and intent. They tell people what needs to be accomplished and why, and leave the details to them.
- **They** learn to balance discipline and creativity, empowerment and hierarchy, plans and improvisation, rapid action and careful analysis.
Principle #3: Decide as Late as Possible – Do It Yourself

1. Think of examples in your life when you have used options to delay decisions. For example, have you ever paid extra to lock in a low interest rate as you negotiated a mortgage? How effective has this been for you? Fill in the following table to the right. We think you will find most examples fall into either the favourable or no-gain category, but few fall into the unfavourable category.

<table>
<thead>
<tr>
<th>Example of keeping options open</th>
<th>Very favourable result</th>
<th>No gain; lost the cost of the option</th>
<th>Very unfavourable result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mortgage Negotiation</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2. At a team or department meeting, ask people to list decisions that are about to be made. Group the list of decisions into two categories – tough to make and easy to make. Then discuss what information you would need to turn each tough decision into an easy decision. Pick three tough decisions and apply the delaying tactics under “Tool 8: Last Responsible Moment” to delay those decisions as long as possible.

3. Evaluate your personality – are you inclined toward breadth-first or depth-first problem solving? Find someone who has the opposite inclination, and pair with him or her as you decide how to approach your next development project.

4. Select a few critical processes and develop simple rules for them so that people understand intent and can make independent decisions.
Principle #4: Deliver as Fast as Possible

… does not mean to rush and do sloppy work, but to deliver value to customers as soon as they ask for it.

- Customers like rapid delivery, which often translates to increased business flexibility. Companies can deliver faster than customers can change their minds. Companies have fewer resources tied up in work-in-progress.

- The principle *Deliver as Fast as Possible* complements *Decide as Late as Possible*: The faster you can deliver, the longer you can delay decisions. E.g. if you can make a software change in a week, then you do not have to decide exactly what you are going to do until a week before the change is needed.

- Rapid delivery is an options-friendly approach to software development. It lets you keep your options open until you have reduced uncertainty and can make more informed, fact-based decisions.
Deliver as Fast as Possible with Pull Systems (Tool #10)

It must be clear to every person, at all times, what she or he should do to make the most effective contribution to the business.

You can either tell them what to do ("command & control") or set things up so they can figure it out for themselves ("self-organization"). In a fast-moving environment, only the second option works.

For effective self-organization, methods for local signaling and commitment must be developed to coordinate work. No schedule can make effective fine-grained work assignments in a complex environment with even modest variability, anyway.

Information radiators / feedback devices: One of the features of a pull system is visual control, or management by sight. Everyone must be able to see what is going on, what needs to be done, what problems exist, what progress is being made.

Visual Feedback – Pin Wall with Stories

TO DO

Story 1 3x
Authenticate User Based on initial scan of left eye

Story 2 2x
Present actions authorized to users granted along with notifications of pending approval requests

Story 3 1x
Restore original settings after 15 minutes of inactivity unless the absence is excused

Story 11
When User Selects and the user is paid in full, request present data and budget validation

Story 4
Refuse all requests for the total amount more than 5% compensated work with the clear goal of doing less of it

Story 13 1x
Enable Selective memory for cases involving poorly compensated work

Story 7
Match the color of the background to the mood of the user as determined by the login retinal scan

Story 5 1x
Report the total amount remaining in the 10 largest accounts every 5 minutes

Story 6
Increase interest payments to the 5 largest accounts by 5% of the current rate

Story 9 2x
Set service fees to 5% for all accounts less than $500,000 unless the balance is a multiple of 3

Story 7
Converse any money to a currency of the user's choice by debiting 2% fee for each conversion and credit to the largest account

Story 8
Chou
Joshua
Amit

Tests Passed

Story 5
Chou

Story 16 3x
If the user says pretty please, undo the last action but log the action for later use when system is in a bad mood

Story 12 1x
Converse any money to a currency of the user's choice by debiting 2% fee for each conversion and credit to the largest account

LSD, p. 75
Deliver as Fast as Possible with Queuing Theory (Tool #11)

- The fundamental measurement of a queue is **cycle time**. When you are in a queue, you always want cycle time to be as short as possible.

- **Steady rate of arrival**: When arrival of demand is spread out to match the capacity of the system, queues, and therefore cycle times, will be shortened. One way to control the rate of work arrival is to release small packages of work.

- **Steady rate of service**: The easiest way to remove variability in the processing time is to increase the number of servers that process work in a single queue. Small work packages will allow parallel processing of the small jobs by multiple teams so that if one is stalled by a problem, the rest of the project can proceed without delay.

- **Slack**: Short cycle times are not possible if resources are overloaded. Full utilization provides no value to the overall value stream; in fact, it usually does more harm than good.
How Queues Work

The way to increase throughput is to look for the current bottleneck that is slowing things down and fix it. It doesn’t do any good to increase the utilization of non-bottleneck areas. Don’t keep piling up work that can’t be used immediately.

- Measuring the amount of work waiting to be done (let's call this *work-in-queue*) is equivalent to measuring the cycle time of a system.
- As variability (in arrival time or processing time) increases, cycle time and work-in-queue will increase.
- As batch size increases, variability in arrival and processing time increases, and therefore cycle time and work-in-queue will increase.
- As utilization increases, cycle time will increase nonlinearly.
- As variability increases, the nonlinear increase in cycle time happens at ever-lower levels of utilization.
- Continuous flow requires a reduction in variability.
- Variability may be reduced by an even arrival of demand, small batches, an even rate of processing, and parallel processing.
- Decreasing variability early in the process has larger impact than decreasing variability late in the process.
The benefits of rapid development are usually larger than you might expect.

Create a simple economic model of a new product for the next few years, basically a profit and loss statement (P&L). Get from marketing good estimates about what delay will do to sales volumes and market share. The model shows what the difference in revenue and market share will do to profits. If the delay means “loss of early high pricing” or “long-term loss of market share”, the cost of delay can be very high.

Keep the model simple, make sure everyone understands and buys into the economic model. Get help from your accountants.

Economic models may help you justify the cost of reducing cycle time, eliminating bottlenecks, and purchasing tools that will allow you to deliver as fast as possible.

Use that economic model of your business to drive development decisions.

Give the team an economic model, and you have empowered the members to figure out for themselves what is important for the business. You have given everyone the same frame of reference, so they can all work from the same assumptions.
## Baseline Software Profit and Loss Statement

<table>
<thead>
<tr>
<th>Assumptions</th>
<th>Year -2</th>
<th>Year -1</th>
<th>Year 0</th>
<th>Year 1</th>
<th>Year 2</th>
<th>Year 3</th>
<th>Year 4</th>
<th>Year 5</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Revenue</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Average Selling Price</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decreases 10% / yr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Market Units</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10,000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20,000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40,000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60,000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40,000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20,000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Market Share</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>30%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>50%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Units Sold</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>20000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Revenue</td>
<td>$ 3,000,000</td>
<td>$ 7,200,000</td>
<td>$ 16,200,000</td>
<td>$ 21,870,000</td>
<td>$ 13,122,000</td>
<td>$ 5,904,900</td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Expense</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unit Mfg &amp; Distribution Cost</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decreases 5% / yr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unit Warranty &amp; Support Cost</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Decreases 10% / yr</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Unit Cost</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ 400</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ 370</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ 343</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ 317</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ 294</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Manufacturing/Support Cost</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ 1,200,000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ 2,960,000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ 6,850,000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ 9,518,250</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ 5,882,425</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ 2,728,542</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gross Margin $</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ 1,800,000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ 4,240,000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ 9,350,000</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ 12,351,750</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ 7,239,575</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ 3,176,358</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Gross Margin %</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>60%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>59%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>58%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>56%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>55%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Development</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ 2,000,000</td>
<td>$ 2,000,000</td>
<td>$ 1,500,000</td>
<td>$ 750,000</td>
<td>$ 750,000</td>
<td>$ 500,000</td>
<td>$ 500,000</td>
<td>$ 500,000</td>
<td></td>
</tr>
<tr>
<td>Marketing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15% of sales</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ 450,000</td>
<td>$ 1,080,000</td>
<td>$ 2,430,000</td>
<td>$ 3,280,500</td>
<td>$ 1,968,300</td>
<td>$ 885,735</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>G&amp;A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5% of sales</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ 150,000</td>
<td>$ 360,000</td>
<td>$ 810,000</td>
<td>$ 1,093,500</td>
<td>$ 656,100</td>
<td>$ 295,245</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Expense</td>
<td>$ 2,000,000</td>
<td>$ 2,000,000</td>
<td>$ 3,000,000</td>
<td>$ 5,150,000</td>
<td>$ 10,840,000</td>
<td>$ 14,392,250</td>
<td>$ 9,006,825</td>
<td>$ 4,409,522</td>
</tr>
<tr>
<td>Profit (Loss)</td>
<td>$ (300,000)</td>
<td>$ 2,050,000</td>
<td>$ 5,360,000</td>
<td>$ 7,477,750</td>
<td>$ 4,115,175</td>
<td>$ 1,495,378</td>
<td></td>
<td></td>
</tr>
<tr>
<td>% of Revenue</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>-10%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>28%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>33%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>34%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>31%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>25%</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cumulative Revenue</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$ 3,000,000</td>
<td>$ 10,200,000</td>
<td>$ 26,400,000</td>
<td>$ 49,270,000</td>
<td>$ 61,392,000</td>
<td>$ 67,296,900</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cumulative Expense</td>
<td>$ 2,000,000</td>
<td>$ 4,000,000</td>
<td>$ 12,450,000</td>
<td>$ 23,290,000</td>
<td>$ 37,682,250</td>
<td>$ 46,689,075</td>
<td>$ 51,098,597</td>
<td></td>
</tr>
<tr>
<td>Cumulative Profit</td>
<td>$ (2,000,000)</td>
<td>$ (4,000,000)</td>
<td>$ (4,300,000)</td>
<td>$ (2,250,000)</td>
<td>$ (3,110,000)</td>
<td>$ (10,587,750)</td>
<td>$ (14,702,925)</td>
<td>$ (16,198,303)</td>
</tr>
<tr>
<td>Cumulative Profit % of Revenue</td>
<td>-143%</td>
<td>-22%</td>
<td>12%</td>
<td>22%</td>
<td>24%</td>
<td>24%</td>
<td>24%</td>
<td>24%</td>
</tr>
</tbody>
</table>
Application Model

- If your software development organization is not involved in “new product development”, develop an economic model for the application from the customer point of view.
- A simple look at the customer’s economic model helps the team make application tradeoff decisions.
- First, identify your customer’s economic drivers related to the application (e.g. call handling time, staffing levels, system support, and customer satisfaction for call center software).
- Second, translate the drivers to economic terms.
- Then, understand the goals that your customer wants to achieve with the application.
- Finally, model each goal separately in its own column and compare each one to the base.

- If you don’t have detailed financial numbers, even some rough estimates are useful. It’s best not to make the numbers too precise.
## Assumptions and Monthly Economic Impact of Desirable System Features

<table>
<thead>
<tr>
<th>Assumptions</th>
<th>Assumptions</th>
<th>Assumptions</th>
<th>Assumptions</th>
<th>Assumptions</th>
<th>Assumptions</th>
</tr>
</thead>
<tbody>
<tr>
<td># calls per day</td>
<td>$60,000</td>
<td>Goal 1: New Operating System</td>
<td>$60,000</td>
<td>Goal 2: $10,000 in new business</td>
<td>$70,000</td>
</tr>
<tr>
<td>avg minutes per call</td>
<td>$30,916</td>
<td>Goal 3: 10% lower time per call</td>
<td>$60,000</td>
<td>Goal 4: 50% less training required</td>
<td>$60,000</td>
</tr>
<tr>
<td>total time (hours) per day</td>
<td>$30,916</td>
<td>One time savings for 50% reduction in training</td>
<td>$30,916</td>
<td>$30,916</td>
<td>$30,916</td>
</tr>
<tr>
<td>peak call rate</td>
<td>$15,000</td>
<td>$30,916</td>
<td>$15,000</td>
<td>$15,000</td>
<td>$15,000</td>
</tr>
<tr>
<td>staff utilization</td>
<td>$9,460</td>
<td>$9,460</td>
<td>$9,460</td>
<td>$9,460</td>
<td>$9,460</td>
</tr>
<tr>
<td>required staffing level</td>
<td>$18,424</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>average hourly pay</td>
<td>$4,624</td>
<td>$8,803</td>
<td>$8,505</td>
<td>$5,044</td>
<td></td>
</tr>
<tr>
<td>regular hours in month</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>total monthly pay</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>% overtime</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>$ overtime</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>total base pay</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>supervision</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>benefits</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>benefits</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>turnover</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>training</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>total salary and benefits</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>total salary and benefits</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>system admin</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>system admin</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>content maintenance</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>content maintenance</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>hardware</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>facilities</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>facilities</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>total monthly cost</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>total monthly cost</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>revenue</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>revenue</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>profit</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>profit</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>margin</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>margin</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>customer satisfaction</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>customer satisfaction</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>first call resolution</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>first call resolution</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>abandoned calls</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>abandoned calls</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>downtime hours</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
<tr>
<td>downtime hours</td>
<td>$2,455</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
<td>$8,000</td>
</tr>
</tbody>
</table>

Revenue: $60,000
Baseline: $60,000
Goal 1: New Operating System: $60,000
Goal 2: $10,000 in new business: $70,000
Goal 3: 10% lower time per call: $60,000
Goal 4: 50% less training required: $60,000

Cost:
Call Center Staffing: $30,916
Support Staffing: $15,000
Hardware & Facilities: $9,460
Total: $55,376

Profit:
$4,624
Monthly Benefit: $6,460
One Time Benefit: $2,940

Profit Margin:
7.7%
Principle #4: Deliver as Fast as Possible – Do It Yourself (1/2)

1. Create a single place where everyone who is interested in a project can come to see:
   - The goal of the current iteration, and
     - What has already been done
     - What is being done
     - What has yet to be done
   - The mission of the overall project, and
     - What has already been done to meet the project mission
     - What has yet to be done to meet the project mission

2. At the end of the next iteration, review your process with an eye for understanding how everyone knows what to do. Ask the team to focus on the way they decide how to spend their time. What would help them make faster and better decisions about what is important? Pick the best two ideas and implement them for the next iteration.
Principle #4: Deliver as Fast as Possible – Do It Yourself (2/2)

3. Find the three longest queues in your area and chart the cycle time for each job as it goes through each queue. It might look something like the following figure. Look for patterns: Is variability high or low? Is there an upward or downward trend?

4. Pick the queue that represents your biggest bottleneck and form a bottleneck task force. Help the task force use queuing theory to find ways to reduce the queue. Measure the results.

5. Ask finance to assign an accountant to every development team, and have the accountant work with the team to develop a simple economic model showing the cost of delay, the cost of reduced features, the cost of maintenance, and so on.
Principle #5: Empower the Team

… does not mean to abandon leadership, but to let the people who add value use their full potential.

- Improvement programs (like MBO, TQM, Zero Defects, Optimized Operations, Service Excellence, ISO9000, Total Improvement Program, Customers First) all too often did not change the reality of how work got done. They increased the intensity of factors leading to job dissatisfaction (policy, supervision, administration) instead of increasing factors that contribute to job satisfaction (achievement, recognition, responsibility).

- Lean thinking capitalizes on the intelligence of frontline workers, believing that they are the ones who should determine and continually improve the way they do their jobs.

- While software development cannot be successful without disciplined, motivated people, experimentation and feedback are more effective than trying to getting things right the first time.

- The critical factor in motivation is not measurement, but empowerment: moving decisions to the lowest possible level in an organization while developing the capacity of those people to make decisions wisely.

- In a lean organization, the people who add value are the center of organizational energy. Frontline workers have process design authority and decision-making responsibility; they are the focus of resources, information and training.
Empower the Team by Self-Determination (Tool #13)

It is notoriously difficult to implement successful improvement programs, and even more difficult to sustain them over time.

General Electrics has a successful improvement program, called “GE Work-Out*”:
- At a Work-Out, 50 or so workers gather for two of three days and come up with proposals that will help them do a better job, specific proposals for doing away with processes that get in the way and implementing practices that will deliver value faster.
- Managers are required to make a yes-no decision on every proposal, either on the spot or within a couple of days.
- Those who made proposals are expected to be responsible for implementing them.
- A combination of simple tools, immediate action, and participation of virtually everyone in the company combined to make Work-Out a uniquely successful improvement program.
- Workers tell managers how to let them do their jobs.
- A Work-Out is a process that teaches managers how to listen to workers, charter them to take action on their ideas, and follow up to be sure that approved proposals are implemented promptly.

The combination of simple tools, immediate action, and participation of virtually everyone in the company combined to make Work-Out a uniquely successful improvement program.

Project Retrospectives

"Here comes Edward Bear now, down the stairs behind Christopher Robin. Bump! Bump! Bump! on the back of his head. It is, as far as he knows, the only way of coming down stairs. He is sure that there must be a better way, if only he could stop bumping for a moment to think of it."

_Winnie -The Pooh_
A.A. Milne, 1926
Empower the Team with Motivation (Tool #14)

Vision statements from William McKnight (leader of 3M from 1930s through the 1950s):
- “Hire good people, and leave them alone.”
- “If you put fences around people, you get sheep. Give people the room they need.”
- “Encourage, don't nitpick. Let people run with an idea.”
- “Give it a try - and quick!”

Intrinsic motivation requires a feeling of belonging, a feeling of safety, a sense of competence, and sense of progress (building blocks of motivation):
- **Belonging**: Everyone is clear on what the goal is and is committed to its success; finally, the team must win or lose as a group.
- **Safety**: Don't kill motivation fast by "a zero defects mentality"
- **Competence**: It is very motivating to be part of a winning team, very demotivating to believe that failure is inevitable. An undisciplined work environment does not generate a sense of freedom; it creates a sense of doom. A sense of competence comes from knowledge and skill, positive feedback, high standards, and meeting a difficult challenge. A leader who delegates and trusts workers must nevertheless verify that they are on the right track and provide the necessary guidance to allow them to be successful.
- **Progress**: Need to feel to have accomplished something. This reaffirms the purpose and keep everyone fired up. Every iteration, the team gets to put its best efforts in front of customers and find out how it has done. When a team reaches a particularly important objective, it's time for a celebration.
Empower the Team with Leadership (Tool #15)

Respected Leaders:
- If you look behind a passionate team, you will find a passionate leader.
- They write the initial concept, set the pace of development and determine how decisions are made.
- They don’t have direct authority over the people working on the project.

Master developers (Aka systems engineers, chief programmers, architects):
- For most large systems, a single or small team of exceptional designers emerge to assume primary responsibility for the design. Exceptional designers exercise leadership through their superior knowledge rather than bestowed authority.
- They are extremely familiar with the application domain and the technology, skilled at communicating their technical vision.
- They understand both the customers and developers, the system’s constraints, interactions, unstated requirements, exception conditions, and likely direction of change.
- They look at the system from a fairly high level of abstraction, yet can drill down to the complexity and detail that both developers and customers must cope with.
- They become the focal point of communication, are part of the team, enmeshed in the details of the work. They provide the leadership necessary for the team to make good decisions, make rapid progress, and develop high-quality software.
Empower the Team with Expertise (Tool #16)

- Some knowledge can be codified and shared by documentation, but much knowledge is tacit knowledge that will only be shared through conversation.

- **Standards** are usually developed by the relevant community of expertise or, when necessary, by the program team.

- It is usually better for a program team to work with existing standards than to develop their own.

- Where standards seem to be lacking and sloppy work is evident, foster **communities of expertise** and ask the communities to develop standards.

- Developers appreciate reasonable standards, especially if they have a hand in developing them and keeping them current.
Principle #5: Empower the Team – Do It Yourself

1. At the end of each iteration, do a process check with the team. Ask two questions:
   - What is slowing you down or getting in the way of doing a good job?
   - What would help things move faster, better, cheaper?
2. Make a list of bad and good practices. Decide which items on the first list can be eliminated and which on the second list can be implemented. Then make it happen. Don’t do this just once – repeat it after each iteration.
3. Make sure that the development team starts each iteration by writing down the goal of the iteration. The goal should be one or two sentences that give the iteration a theme related to the business value it will deliver. Post the goal in a prominent spot and refer to it when the team is struggling with a tough decision.
4. Use pair programming or design reviews within the framework of software craftsmanship. Encourage pair programming for the expertise sharing it provides. If design reviews are held, assure that the agenda and tenor of the meeting focus on learning and sharing expertise rather than on ferreting out mistakes.
5. Ask each person on the development team to write down one specialty area in which the team is low on expertise. List everyone’s answers and look for a pattern. Have team members pick their top candidate and see which one gets the most votes. Then work with the team to come up with a plan to make that expertise more available to the team. You might use the following strategies:
   - Buy everyone who is interested a relevant book and meet once a week at lunch to discuss a chapter.
   - Find a guru in the specialty in question and have him or her pair with various team members, as availability permits, so they can strengthen their skill in the area.
   - Set up a three-person subcommittee to establish team conventions for the area in question. Be sure they evaluate any corporate or industry standards in preference to designing their own.
**Principle #6: Build Integrity In**

... does not mean big, upfront design, but don't try to tack on integrity after the fact, build it in.

- Kim Clark of the Harvard Business School examined in the late 1980s how some companies could consistently develop superior products. The key differentiator was that the products had integrity.

- **External (perceived) integrity** means that the totality of the product achieves a balance of function, usability, reliability, and economy that delights customers.

- **Internal (conceptual) integrity** means that the system’s central concepts work together as a smooth, cohesive whole.

- The measure of perceived integrity is roughly equivalent to market share, or perhaps a better term might be mindshare.

- Conceptual integrity is a prerequisite for perceived integrity. It emerges as the system evolves and matures.

- The way to build a system with high perceived and conceptual integrity is to have excellent information flows both from customer to development team and between the upstream and downstream processes of the development team.
Build Integrity In – Perceived Integrity (Tool #17)

- Companies that consistently achieve perceived integrity have a way of constantly keeping customer values in front of the technical people making detailed design decisions. Chief engineers have added to their engineering and leadership skills the ability to understand the target customer base and create a vision.
- Customers will know a good design when they see it, but they can't envision it beforehand. To make matters worse, as their circumstances change, so will customers' perception of system integrity.
- **Model-driven design**: Domain models must be both understood and directly usable by the customer (as an ubiquitous language) and by the developers actually writing the code. Joint modeling ensures that the results will be both a correct representation of the domain issues and at the same time be effectively implementable in software.
- Software directly reflecting the domain model will be more robust to changing business needs than software with significantly different internal structures chosen for purely technical reasons.
- The best way to maintain institutional knowledge about a system and keep it maintainable is to deliver a suite of automated tests along with the code, supplemented by a high-level overview model created at the end of the initial development effort. Design documentation rarely reflects the system as it was actually built, so it is widely ignored by maintenance programmers.

* Eric Evan: "Domain-Driven Design"
Build Integrity In – Conceptual Integrity (Tool #18)

- The components match and work well together; the architecture achieves an effective balance between flexibility, maintainability, efficiency, and responsiveness.
- The key to achieving conceptual product integrity is the effectiveness of the communication mechanisms developed among the engineering groups as all the decisions are made.

- Use of existing parts immediately removes many degrees of freedom and thus reduces the complexity and need for communication.

- Use integrated problem solving to assure excellent technical information flow:
  - Understanding the problem and solving the problem at the same time, not sequentially.
  - Preliminary information is released early; information flow is not delayed until complete information is available.
  - Information is transmitted frequently in small batches, not all at once in a large batch.
  - Information flows in two directions, not just one.
  - The preferred media for transmitting information is face-to-face communication as opposed to documents.

- Rich, bilateral communication deemphasizes control mechanisms in favor of face-to-face discussions, small batches, speed, and flow.
- Predicting the future tends to be a waste of time and resources. It is better to take a breadth-first approach and get the basics right. Then, let the details emerge and plan on regular refactoring to keep the architecture healthy.
How can you be sure that a good architecture will emerge?

- First, use existing parts and off-the-shelf software when possible
  - By fixing as many points of the system as feasible with existing software and standards, you reduce the communication required, clearing the path for better communication on the remainder of the system

- Second, use integrated problem solving:
  - Getting started on writing software before the design details are finalized.
  - Show partially complete software to customers and users to get their feedback.
  - Make sure that developers have access to customers to get questions answered as soon as they arise.
  - Run usability tests on each feature as soon as it is developed.
  - Develop and run customer tests throughout each iteration, not just at the end.

- Third, be sure there are experienced developers involved in all critical areas.

- Finally, complex systems require the leadership of a master developer.
Build Integrity In with Refactoring (Tool #19)

- Engineers start with something that works, learn from its weaknesses, and improve the design. “It takes five to six attempts to really get a product right.”

- Maintain the conceptual integrity by refactoring when a system begins to lose the following characteristics:
  - Simplicity
  - Clarity (well-understood naming conventions, using a common language, code clarity, simple notation, encapsulation, and sparse, focused comments)
  - Suitability for use
  - No repetition (when change has to be made in more than one place, the possibility for error grows exponentially, so duplication is one of the biggest enemies of flexibility; DRY principle - don't repeat yourself)

- Good design evolves over the life of a system, but this does not happen by accident; poor code does not get better by being ignored
  - Stop the line (i.e. stop adding new features) when you detect a smell.
  - Take the time to find and fix the root cause of the problem before proceeding with more development.
  - Some teams spend too much time perfecting unimportant details. Experienced designers say that the one mistake they made in developing a system was not refactoring aggressively enough.

- Just like advertising, refactoring doesn't cost, it pays.
Build Integrity In with Testing (Tool #20)

- As we move from programming one module at a time to programming entire capabilities and features, the distinction between unit, system, and integration tests has less meaning.
  - A better name for these tests might be developer tests, because their purpose is to assure that code does what the developer intended it to do
  - Tests to be sure that the system does what customers want have been called acceptance tests, placed traditionally at the end of development. A better name for these tests that make sure that a system does what customers intend is customer tests. Customer tests are run throughout the development, not just at the end
- If there doesn't seem to be enough time, the first thing to do is reallocate the effort used in requirements documentation to writing customer tests.
- Tests should be automated as much as possible and run as part of the daily build.
- Make sure the tests are correct and complete, put them under version control, automate them, consider them as part of the released product, and continue to use and improve them.
- Maintain a set of comprehensive tests throughout the lifecycle of the system. Then, the system can be safely repaired and refactored throughout its useful life.
- If tests are clear and well organized, they are an invaluable resource for understanding how the system works from a developer's and a customer's point of view.
Principle #6: Build Integrity In – Do It Yourself (1/2)

1. Pick one of your current systems and find out if it has a common language. Chat with the customers and write down a glossary of what they consider key terms that they use when talking about the system. Take this glossary to the development team and find out if they use the same words or if they have a technical translation for some domain terms. Next, ask the developers to identify in the code the names they use for each word in the combined glossary. Finally, see if there are any key classes in the systems that are not represented in the glossary. If you detect that there are two or three different vocabularies in use, explain to the development team why it is important for them to use the domain language, even among themselves.

2. Hold a team meeting and invite any of the following people who normally would not be there. People who will
   - Test the system
   - Deploy the system
   - Train the users
   - Be responsible for operating the system in production
   - Work at a help desk for the system
   - Maintain the system
   - Develop or maintain any system accessing the same data

Have the assembled group brainstorm any concerns they have about the system under development. Then, use prioritization to pick the three most important issues. Form a joint committee of interested parties to address the three issues. Meet again in two weeks to be sure the three issues have been resolved, and repeat the process.
Principle #6: Build Integrity In – Do It Yourself (2/2)

3. Put five sheets of flip chart paper on the wall in the team room. Label the top of each sheet:
   - Simplicity
   - Clarity
   - Suitability for Use
   - No Repetition
   - No Extra Features
Ask each developer to note on the appropriate piece of paper anything in the current system that does not seem to meet the standard. For instance, if they detect repetition, they would note the culprits on the No Repetition sheet. When refactoring has removed an offending item, it is crossed off the list. At the end of the iteration, let the team take a day or two to clean up the worst offenders on the charts.

4. Estimate the average cycle time of the following:
   - Time from writing feature until developer test is run.
   - Time from writing feature until it is integrated into system and automated developer test suite is run.
   - Time from writing feature until customer test is run.
   - Time from writing feature until usability test is run.
   - Time from writing feature until deployment
Next, write down a target cycle time for each item. Attack this list from top to bottom: Work with the team to come up with a plan to bring each cycle time down to its target number, and one by one, close the gap.
Principle #7: See the Whole

... does not mean to ignore the details, but to beware of the temptation to optimize parts at the expense of the whole.

- As a problem gets worse, managers apply even more aggressively the very policies that are causing the problem.

- An underlying problem produces symptoms that can't be ignored. However, the underlying problem is difficult to confront, so people address the symptoms instead of the root cause of the problem. Unfortunately, the quick fix allows the underlying problem to grow worse, unnoticed because its symptoms have been covered up.

- The five Whys: to address the root cause, keep asking why and don't stop when you find the first (reasonable) answer, since that may only be the symptom. You have at least one more why to ask before you arrive at the root of the problem.

- "I keep six faithful serving-men who serve me well and true: Their names are What and Where and When and How and Why and Who." – Kipling

- Sub-optimization:
  - The more complex a system, the more temptation there is to divide it into parts and manage the parts locally.
  - Local management tends to create local measurements of performance. These local measurements often create system-wide effects that decrease overall performance.
See the Whole with Measurements (Tool #21)

- Lance Armstrong won the Tour de France each year from 1999 to 2003, yet he won only a few of the daily stages. Just like the Tour de France, optimizing every task is often a very bad strategy.

- It is very difficult to measure everything that is important with knowledge work, especially when each effort is unique and uncertainty reigns.

- If you cannot measure everything that is important, partial measurements are very likely to turn into suboptimized measurements. If you can't measure everything that is necessary to optimize the overall business goal, then you are better off without the suboptimizing partial measurements.

- The way to be sure that everything is measured is by aggregation, not disaggregation. That is, move the measurement one level up, not one level down.

- Information measurements (obtained by aggregating data to hide individual performance), not performance measurements, should be used.

- Don't track defects by developer: Less than 20 percent of all quality defects are under the worker's control; the rest are rooted in the prevailing systems and procedures, which are under management control, not worker control.
  - We are not looking for the root causes of the problems if we trace defects to individuals.
  - The way to find the root cause of defects is to encourage the entire development organization to collaborate in seeking them out.
See the Whole – Contracts (Tool #22)

- Project managers have **four variables** that they can adjust when managing projects: **time, costs, quality**, and **scope**. From these four variables, fix time, cost and quality, but not scope. Prioritize features, but don't specify in the contract the fixed set of features to be delivered. Move from a fixed scope to a negotiable scope*: By delivering high priority features first, it is likely that you deliver most of the business value long before the customer's wish list is completed.
- As noted earlier, up to 45% of the features delivered might never be used (according to a study of the Standish Group).
- Barry Boehm and Philip Papaccio noted in 1988: the best way to develop low-cost, high-quality software is to write less code.
- Rigid control of scope tends to expand, not reduce, the scope. Save money overall by collaborating with the customer by using some form of optional scope contract.

Principle #7: Seeing the Whole – Do It Yourself

1. Make sure your defect measurement system is an informational measurement system rather than a performance measurement system.
   - Are defects traceable back to the developer who caused the defect? Why? If there is no good reason, then eliminate the person’s identity from the defect reporting system; don’t even collect the names.
   - If there is a reason why you need a developer’s identity (e.g., the developer must fix the code), then be sure that an individual developer is the only one who sees the reports related to his or her work. Aggregate all defect reports; do not publicly display or manage from defect measurements sorted by developer.

2. Whether you outsource or are a contractor, the first step to using agile methods under contract is to figure out a way to make scope optional. Ask your legal team to scour the available literature on methods to provide adequate protection to your company without using a fixed-scope specification.
Questions?
Backup Slides
## Comparison with 4 Values of XP

<table>
<thead>
<tr>
<th><strong>Communication</strong></th>
<th>Amplify Learning, Empower the Team</th>
</tr>
</thead>
<tbody>
<tr>
<td>Customers must communicate needs and desires; developers must communicate the ramifications of various options; and both must communicate with management to paint an accurate picture of the state of the project. Only with excellent communication can every faction in the development process know the intent of the product and thereby work together effectively for its successful delivery.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Simplicity</strong></th>
<th>Eliminate Waste, Deliver as Fast as Possible, Build Integrity In</th>
</tr>
</thead>
<tbody>
<tr>
<td>A product must only be as complex as the need it is intended to fill. Why use a freight truck to carry groceries home from the store? Development beyond what is necessary is a waste of time, resources, and hence money. Overly complex systems hide many subtle yet unnecessary problems.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Feedback</strong></th>
<th>Amplify Learning, Decide as Late as Possible</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feedback ‘guides’ the process along the way. Proper feedback stops problems at the earliest possible point during the development process. To highlight this, an incorrect or inadequate component stops being an issue at the moment when its shortcomings are identified.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Courage</strong></th>
<th>Amplify Learning, Empower the Team, See the Whole</th>
</tr>
</thead>
<tbody>
<tr>
<td>Developers, managers, and customers should not be afraid to try new ideas and approaches. Sometimes a chance must be taken to achieve the desired results. Communication and/or feedback will raise a flag if something isn't going to work.</td>
<td></td>
</tr>
</tbody>
</table>
Comparison with 12 Practices of XP

- **Test Driven Development (Testing)**
  Tests allow rapid change and collective code ownership by keeping the change from introducing mistakes. Developers write unit tests before and during programming. Customers write acceptance tests to define functionality. By running these tests you ensure that the system always works.

- **Small Releases**
  Building in small releases allows maximum benefit to the customer and gives the developer the best-possible feedback. Every feature added to the product enhances its usability.

- **Refactoring**
  As you learn what the design wants to be, you improve the design by refactoring. By constantly paying attention to design by refactoring, you remove duplication and discrepancies, thereby eliminating waste and reducing cost.

- **Simple Design**
  Do not overbuild! Complexity makes things difficult to understand, especially when it is not necessary. Simplifying the design is as important as coding itself.

- **Planning Game**
  Only plan enough to get started, and refine the plan as necessary. This applies to both release planning (when a version of the software will be released) and iteration planning (work done in a 2- or 3-week iteration).

- **Pair Programming**
  As the primary communication between and among programmers, all production code is written by two people collaborating at the keyboard. This allows for maximum speed and cross-training in support of shared code ownership and rapid progress. The total will be greater than the sum of the parts!

- **Onsite Customer**
  What happens when you have to wait for an answer or have to guess what the customer's intent was? With XP, questions are answered immediately and feedback is nearly instantaneous, because you have immediate access to the customer.

- **System Metaphor**
  In order for the customer and programmer to be in sync in their communication, it is important to have a common vernacular for discussing the product and its features. Ideally, the metaphor will reflect the real-world operations that will be performed by the product.

- **Collective code ownership**
  Sharing the ownership of all code means that all members of the team can make changes they deem necessary; no individual is a stumbling block in the development process. Any member of the team has the right to add to, refactor, or remove code from the system when necessary for the task on which he/she is working.

- **Continuous Integration**
  Keeping the system integrated at all times (1) means that you always have a good version to look at; and (2) lets you go fast without stepping on one another’s toes. This is done using version control software. Completed features are placed into the common code repository as soon as possible so that every developer has access to the ‘latest and greatest’.

- **Coding Conventions**
  By keeping the format consistent while coding, communication is improved because no developer ever looks at code with which he/she is unfamiliar.

- **Sustainable Pace (40 hour week)**
  How do you avoid project ‘management by crisis’? By working at a consistent, sustainable pace, accurate estimates can be made.
XP practices and LSD tools

- **Coding circle**
  - #1 Seeing Waste
  - #4 Iterations, #6 Set-Based Development
  - #17 Perceived Integrity
  - #18 Conceptual Integrity
- **Team circle**
  - #3 Feedback
  - #20 Testing
  - #19 Refactoring
  - #13 Self-Determination
  - #18 Conceptual Integrity
- **Process circle**
  - #11 Queuing Theory
  - #14 Motivation
- **Product circle**
  - #3 Feedback
  - #5 Synchronization
  - #9 Making Decisions
  - #16 Expertise
  - #4 Iterations
  - #13 Self-Determination, #16 Expertise
  - #14 Motivation

- **Planning Game**
  - Simple Design
  - Continuous Integration
  - Testing
  - Refactoring
  - Pair Programming
  - Coding Standards
  - Small Releases
  - 40 Hour Week
  - On-Site Customer
  - Collective Code Ownership
  - System Metaphor
  - #16 Expertise
  - #14 Motivation

- **#1 Seeing Waste**
- **#4 Iterations, #6 Set-Based Development**
- **#17 Perceived Integrity**
- **#18 Conceptual Integrity**
- **#3 Feedback**
- **#20 Testing**
- **#19 Refactoring**
- **#13 Self-Determination, #16 Expertise**
- **#11 Queuing Theory**
- **#14 Motivation**
- **#16 Expertise**
Comparison with Accelerators of Accelerated Solution Delivery (AMS Offering)

- **Solution Accelerators**
  - Scope prioritization
  - Timeboxing
  - Iterative design, development, and prototyping
  - Joint Application Design (JAD)
  - Reusable solutions and artifacts
  - Agile (but scalable) processes and documentation

- **Team accelerators**
  - Co-location of business customer and project teams
  - Small, dedicated project teams (4 to 8) with specialist support
  - Dedicated facilities and equipment

- **Organizational Accelerators**
  - Project organization structure supported by centers of competency
  - Common, resource pool that changes over time based on needs
  - Executive business and IT commitment
  - Responsive, flexible, governance that aligns IT capabilities with business needs
### How IBM ASD supports Standish Chaos Chronicles Top 10 project success factors

<table>
<thead>
<tr>
<th>Project Success Factor</th>
<th>Key points behind Success Factor</th>
<th>Key IBM ASD Model attributes that support Success Factor</th>
</tr>
</thead>
</table>
| User involvement       | This is the #1 factor to project success  
  Users know the requirements and will ultimately use the software | Dedicate them as full or partial team members  
  Ensure that involved user is knowledgeable & empowered  
  Collocate them with the project team  
  Environment that encourages user input & involvement |
| Executive Support      | They have something at stake & have an interest in project success | Partnership with client  
  PM Processes include sponsor and executive sponsor identification and participation |
| Experienced Project Managers | 97% of successful projects have an experienced PM at the helm | Experienced IBM PM's  
  Agile, scalable PM process is easy to use and train new PM's to use  
  Repetition on shorter projects  
  Project organization structure that provides adequate support & scalability |
| Clear Business Objectives | They set the framework for the entire project | Chartering process  
  Project organization with tight ties to governance |
| Minimizing Scope       | “Time is the enemy of all projects”  
  Prioritize functionality and Timebox | Scope prioritization & management  
  Scope as variable constraint vs time and budget  
  Fixed duration projects using Timeboxing  
  Decomposition of enterprise initiatives into small projects |
| Firm Basic Requirements | Acknowledges that SW requirements are in a constant state of flux  
  Changing requirements is as certain as death and taxes | JAR/JAD by experienced resource  
  Iterative development  
  CMM compliant processes |
| Standard Software Infrastructure | Keeps focus on business vs. technology | Recommends standard infrastructure  
  COC's/COE's drive standards through participation in projects |
| Formal Methodology     | Reproducible & reusable  
  Consistency maximized  
  Lessons learned  
  Increases success rate by 16% | Agile, scalable methodology is easy to learn, use, institutionalize  
  Project organization ensures consistency |
| Reliable Estimates     | Realistic, systematic approach | Standard process, performed by same experienced resources  
  Smaller projects  
  Dedicated resources |
| Skilled Staff          | Understand skills needed  
  Inside and outside company  
  Training program  
  Provide incentive | Proven team constructs, clear roles & responsibilities  
  Dedicated teams  
  Supported by specialists  
  Mix of mentor and train |
How LSD supports Standish Chaos Chronicles Top 10 project success factors

<table>
<thead>
<tr>
<th>Project Success Factor</th>
<th>Key points behind Success Factor</th>
<th>Key LSD tools that support Success Factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>User involvement</td>
<td>▪ This is the #1 factor to project success</td>
<td>▪ Help the customer see waste (#1)</td>
</tr>
<tr>
<td></td>
<td>▪ Users know the requirements and will ultimately use the software</td>
<td>▪ Use a lot of feedback (#3) in short iterations (#4)</td>
</tr>
<tr>
<td></td>
<td>▪ Make decisions (#9)</td>
<td></td>
</tr>
<tr>
<td>Executive Support</td>
<td>▪ They have something at stake &amp; have an interest in project success</td>
<td>▪ ???</td>
</tr>
<tr>
<td>Experienced Project Managers</td>
<td>▪ 97% of successful projects have an experienced PM at the helm</td>
<td>▪ ???</td>
</tr>
<tr>
<td>Clear Business Objectives</td>
<td>▪ They set the framework for the entire project</td>
<td>▪ ???</td>
</tr>
<tr>
<td>Minimizing Scope</td>
<td>▪ “Time is the enemy of all projects”</td>
<td>▪ Eliminate waste (#1, #2)</td>
</tr>
<tr>
<td>Firm Basic Requirements</td>
<td>▪ Acknowledges that SW requirements are in a constant state of flux</td>
<td>▪ ???</td>
</tr>
<tr>
<td></td>
<td>▪ Changing requirements is as certain as death and taxes</td>
<td></td>
</tr>
<tr>
<td>Standard Software Infrastructure</td>
<td>▪ Keeps focus on business vs. technology</td>
<td>▪ ???</td>
</tr>
<tr>
<td>Formal Methodology</td>
<td>▪ Reproducible &amp; reusable</td>
<td>▪ ???</td>
</tr>
<tr>
<td></td>
<td>▪ Consistency maximized</td>
<td></td>
</tr>
<tr>
<td></td>
<td>▪ Lessons learned</td>
<td></td>
</tr>
<tr>
<td></td>
<td>▪ Increases success rate by 16%</td>
<td></td>
</tr>
<tr>
<td>Reliable Estimates</td>
<td>▪ Realistic, systematic approach</td>
<td>▪ ???</td>
</tr>
<tr>
<td>Skilled Staff</td>
<td>▪ Understand skills needed</td>
<td>▪ Amplify Learning</td>
</tr>
<tr>
<td></td>
<td>▪ Inside and outside company</td>
<td>▪ Empower the Team</td>
</tr>
<tr>
<td></td>
<td>▪ Training program</td>
<td></td>
</tr>
<tr>
<td></td>
<td>▪ Provide incentive</td>
<td></td>
</tr>
</tbody>
</table>