Stop-the-Line Quality

Lessons from Lean
# The Toyoda’s

<table>
<thead>
<tr>
<th>Name</th>
<th>Years</th>
<th>Contributions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sakichi Toyoda</td>
<td>(1867-1930)</td>
<td>- Extraordinary inventor of automated looms</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Crucial Idea: <em>Stop-the-Line</em></td>
</tr>
<tr>
<td>Kiichiro Toyoda</td>
<td>(1894-1952)</td>
<td>- Bet the family fortune on automotive manufacturing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Crucial Idea: <em>Just-in-Time</em></td>
</tr>
<tr>
<td>Eiji Toyoda</td>
<td>(1913-Present)</td>
<td>- 50 years of Toyota leadership</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- Championed the development of <em>The Toyota Production System</em></td>
</tr>
</tbody>
</table>
The Toyota Production System

Taiichi Ohno

*The Toyota Production System*, 1988 (1978)

- Eliminate Waste
  - Just-in-Time Flow
- Expose Problems
  - Stop-the-Line Culture

Shigeo Shingo

*Study Of ‘Toyota’ Production System*, 1981

- Non-Stock Production
  - Single Digit Setup
- Zero Inspection
  - Mistake-Proof Every Step
**Lesson:**
Stop trying to maximize machine productivity.

**Lesson:**
Stop trying to maximize “resource” utilization.
Stop the Line Culture

1920’s:
- Idea: Unattended looms
- Invention: Looms that stopped the moment a thread broke.

Lessons:
- The greatest productivity comes from not tolerating defects.
- Create ways to detect defects the moment they occur.
“Only after American carmakers had exhausted every other explanation for Toyota’s success – an undervalued yen, a docile workforce, Japanese culture, superior automation – were they finally able to admit that Toyota’s real advantage was its ability to harness the intellect of ‘ordinary’ employees.”

Software is rather useless
  – all by itself
Software is embedded
  In hardware
  In a process
  In an activity

The product [or process] must be developed as a system.

The system is going to be around for a LONG time.

Saving money in development at the expense of production makes no sense.

60-80% of coding occurs after first release to production.

Developing a change-tolerant system is fundamental.
Lesson 1: Learn to See Waste

Put on Customer Glasses

MUDA
anything that does not add VALUE
Myth: Early Specification Reduces Waste

Features and Functions Used in a Typical System

- **Often or Always Used:** 20%
- **Sometimes Used:** 16%
- **Rarely Used:** 19%
- **Never Used:** 45%

Standish Group Study Reported at XP2002 by Jim Johnson, Chairman
Keep it Simple
Reduce Risk

The Biggest Risk is Work-in-Process

- The *Big Bang* is Obsolete

Sources of Risk

- Un-coded specifications
- Un-tested code
- Un-integrated code
- Code that has not been used in production

The Best Risk Mitigation is Low Work-in-Process

- Test early, integrate often, fail fast.
Lesson 2:
Don’t Tolerate Defects

There are Two Kinds of Inspection*

1. Inspection to Find Defects – WASTE
2. Inspection to Prevent Defects – Essential

The Role of QA

The job of QA is not to swat misquotes,
The job of QA is to put up screens.

A quality process builds quality into the code
✓ If you routinely find defects during verification
   – your process is defective.

* Shigeo Shingo
Where do defects come from?

**90% of all defects caused by the system***

1. They are not caused by individuals.
2. System problems are management problems.

*Dr. W. Edwards Deming*

**Change The System**

Mistake-Proof Every Step

- Detect defects the moment they occur
- Don’t track defects on a list
- Find them and fix them

Test FIRST
Case Study

Mobile Spectrometer to Analyze Grain

Techniques:

✓ Trouble log with different behaviors depending on development or field platform and severity of error.
✓ Dual-targeting: Bracket HW-dependent code and run only with target HW, mock-out otherwise.
✓ Isolate HW driver code, use scripts to test it with HW
  ✗ Became the HW acceptance tests
✓ Isolate and test domain-level code (eg communications)
✓ Special tests for unique domains (eg math algorithms)

Result:

✓ In 3 years, only 51 defects (18 critical, 23 moderate, 10 cosmetic), with a maximum of 2 open at once!
✓ Productivity 3X similar embedded software teams.
✓ HW engineers trusted SW and used it to debug HW.
Building Block Disciplines

Development

- ✔ Coding Standards
- ✔ Configuration Management
  - ✗ Tool
  - ✗ Team Practices
- ✔ One Click Build
- ✔ Continuous Integration
- ✔ Automated Testing
  - ✗ Unit Tests
  - ✗ Acceptance Tests
  - **STOP** if the tests don’t pass
- ✔ Nested Synchronization

Deployment

- ✔ Production-Hardy Code
- ✔ Automated Release Packages
- ✔ Automated Installation
- “Done” means the code is running live in production.
Nested Synchronization

✓ Every few minutes
  ✗ Build & run unit tests
  ✗ *STOP* if the tests don’t pass

✓ Every day
  ✗ Run acceptance tests
  ✗ *STOP* if the tests don’t pass

✓ Every week
  ✗ Run production tests
  ✗ *STOP* if the tests don’t pass

✓ Every iteration
  ✗ Deployment-ready code

✓ Every Release
  ✗ Deploy and run in production

Make it Flawless
## Types of Testing

<table>
<thead>
<tr>
<th>Support Programming</th>
<th>Automated: Every Day</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business Facing</td>
<td></td>
</tr>
<tr>
<td>Acceptance Tests</td>
<td>Manual: As Early as Practical</td>
</tr>
<tr>
<td>Business Intent</td>
<td>Automated: Every Build</td>
</tr>
<tr>
<td>Unit Tests</td>
<td>Manual: As Early as Practical</td>
</tr>
<tr>
<td>Technology Facing</td>
<td>Tool-Based: As Early as Possible</td>
</tr>
<tr>
<td>Property Testing</td>
<td></td>
</tr>
<tr>
<td>Usability Testing</td>
<td></td>
</tr>
<tr>
<td>Exploratory Testing</td>
<td></td>
</tr>
</tbody>
</table>

### Business Intent
- **Acceptance Tests**: Business Intent (Design of the Product)
- **Unit Tests**: Developer Intent (Design of the Code)

### Technology Facing
- **Property Testing**: Response, Security, Scaling, Resilience

*From Brian Marick*
Myth: Automated Testing takes too much time/costs too much money

Team struggling with legacy java code
10 defects / 1000 NCSS
Affected company’s reputation and threatened survival

Adopted Test Driven Development
Defects dropped to <3 / 1000 NCSS
✓ Including all untouched legacy code
✓ 80-90% improvement in quality

Productivity more than tripled

* Non-comment Source Statements

--- From Mike Cohn
Test Driven Development

Doesn’t cost, it pays!
Is a design technique

✓ Cleaner Design

✗ Acceptance Test Driven Design
  ❖ Matches the design to the structure of the domain

✗ Unit Test Driven Design
  ❖ Simpler, More Understandable Code

✓ Self-verifying

✓ Protects from unintended consequences
  ❖ For the life of the code
A Test Harness to Simulate Integration Testing

- Create a harness to simulate the remote system at each integration point in the system under test.
- Design a devious harness with nasty, malicious behavior that will beat up the system.
- Try to provoke all possible failure modes in any remote system at all seven OSI layers.
- A single harness can work for many networked applications, simulating similar bad behavior.

A harness for a Web Services call
- Refuse all connections
- Refuse all credentials
- Listen but time out
- Connect very slowly
- Send nothing but RESET’s
- Accept connection but don’t send data (or don’t acknowledge data)
- Accept a request and send response headers but no body
- Report data received but never empty the buffer
- Send 1 byte of data every 30 sec.
- Send megabytes of data when kilobytes are expected
- Send unexpected formats
- Etc.
Avoid Technical Debt

Anything that makes code difficult to change
(The usual excuse for batches & queues)

✓ Complexity
  The cost of complexity is exponential.

✓ Regression Deficit
  Every time you add new features
  the regression test grows longer!

✓ Unsynchronized Code Branches
  The longer two code branches remain
  apart, the more difficult merging will be.

Perfection is One-Piece-Flow:
Any useful feature set – at any time – in any order

Let it Flow
Case Study: Rally Software Development

“We found ourselves doing waterfall in time-boxed increments. During the first year we had a lot of technical debt.”

Testing:
- JUnit for unit tests
- HTTPUnit for testing the GUI
  - Not capable of testing page flows
  - Most GUI testing manual
  - All acceptance testing manual
- 6 weeks to develop, 2 weeks to test, and not all testing was done.

“...and it just kept going up.”

- Gradually moved page flow platform to Spring and AJAX
  - Tested Spring with FIT & Fitnesse
  - Tested AJAX by using JIFFIE to bind Java to IE. Wrote tests in Java to test AJAX through the browser.

- Hardening was reduced to 1 week.

- Responsibilities changed:
  - Testers: FIT tables & JIFFIE tests
  - Developers: FIT fixtures, JUnit tests, and GUI test harness

- Now release monthly, pre-hardened!

Ryan Martens, CTO
Lesson 3: Focus on Learning

Cycles of Discovery
Products emerge through iterations of learning.

Relentless Improvement
Engaged people design and improve their own processes.

Scientific Method

Feedback
Iterative Development

Iteration Planning

Daily

Iteration Execution

One Iteration Ahead

Every 2-4 Weeks

Deployment - Ready Software

Stories & Tests

Commitment

Feedback

Design

Goals

Road Map:

Prioritized list of desirable features

Deployment

April 07

Copyright©2007 Poppendieck.LLC
Relentless Improvement

Regular Work Team Meetings
- Every week or
- Every iteration

Data-Based Problem Analysis
- Don’t guess
- Find and analyze the data
- Experiment!

Kaizen

Kai
Change

Zen

Good

25 April 07 Copyright©2007 Poppendieck.LLC
Refactoring: Relentless Improvement of the Code Base

Just-in-time NOT Just-in-Case

✓ Start with what you know is needed now
✓ Add features only when you know you need them
✓ Refactor: Simplify the code based on what you know now

Maintain a Simple, Clean Design

✓ No features ahead of their time
✓ No features after their time
✓ No Repetition

Safety First!

✓ You can’t refactor without test harnesses.

Time to Refactor
Lesson 4
Change the Measurements

Fujitsu took over help desk of BMI (airline) in 2001

✓ Fujitsu analyzed all calls
  ✗ Found that 26% of calls were for printers
    ✗ Could not print boarding passes / luggage tags
  ✗ Quantified the cost of the calls
    ✗ Tracked the time to fix the problems
    ✗ Measured impact on business of the problem
  ✗ Convinced BMI management to get better printers
    ✗ Printer calls were down 80% in 18 months
    ✗ Total calls were down 40% in 18 months
    ✗ Major savings in flight operations

✓ What’s Wrong With This Picture?

From “Lean Consumption”
By James P. Womack & Daniel T. Jones
Harvard Business Review, March 2005
## Measure UP

### Decomposition
- You get what you measure
- You can’t measure everything
- Stuff falls between the cracks
- You add more measurements
- You get local sub-optimization

### Aggregation
- You get what you measure
- You can’t measure everything
- Stuff falls between the cracks
- You measure UP one level
- You get global optimization

### Example
- **✓** Measure Cost, Schedule, & Scope
  - ✗ Quality & Customer Satisfaction fall between the cracks
  - ✗ Measure these too!

### Example
- **✓** Measure Cost, Schedule, & Scope
  - ✗ Quality & Customer Satisfaction fall between the cracks
  - ✗ Measure Business Case Realization instead!
Three System Measurements

**Average Cycle Time**
- From Product Concept
- To First Release
- From Feature Request
- To Feature Deployment
- From Defect
- To Patch

**The Business Case**
- P&L or
- ROI or
- Goal of the Investment

**Customer Satisfaction**
- A measure of sustainability