Adapting some Rules and Principles of TPS (Toyota Production System) to Software Development

S.R. Subramanya
School of Engineering, Technology, and Media
National University, San Diego, CA

ABSTRACT

The Toyota Production System (TPS) has been widely studied and copied by various companies to improve several of their processes. However, many of them have not been successful due to the fact that only the tools and tactics have been focused and the core underlying principles have been overlooked. There are examples of a few companies which have achieved cost savings and operational efficiencies by applying the TPS principles. TPS principles have also had significant influence on lean software development. In this paper, we present the adoption and adaptation of the spirit behind some of the principles from TPS to the software development process in a mobile phone software development scenario.

General Terms

Software Development, Lean Principles.

Keywords

Software Development, Toyota Production System, Lean Software Development.

1. INTRODUCTION

The Toyota Production System (TPS), which has enabled Toyota to outperform its competitors in quality, reliability, production, cost reduction, and growth, has been attributed to Taiichi Ohno for developing and systematizing the principles therein [1]. The TPS, its principles, and the underlying scientific method were never designed consciously or imposed upon. They grew naturally over time out of workings of the company. One of the striking features of the TPS is the continuous improvement of processes. The improvements are very many and each improvement is small and highly focused to a small operation, and these improvements are quickly tried out and incorporated. The TPS has been widely studied by industrialists, executives, researchers, and journalists (ex. [2], [3], [4], [5]). The book Lean Thinking [6] was popular and introduced some Toyota ideas to a much wider audience. Several companies have studied and tried to copy the TPS, but have not been successful. One of the reasons attributed to this is that they have focused on the tools and tactics of TPS without focusing on the basic set of operating principles. In the TPS, the people bring the system to life by working, communicating, resolving issues, and growing together [2]. There have also been a few companies which have adapted the basic principles of TPS to their advantages, by focusing on some of the core principles.

Any changes and improvements done using the TPS are based on sound scientific methods. They are made using rigorous problem solving processes. The changes and improvements are clearly specified, which is as if a set of hypothesis is established which is then tested. The experiment is carefully planned to carry out the test of hypothesis. Without the scientific methods, the changes and improvements would be more of random trials and errors.

There have been a few studies and experiments to adapt the TPS to software development, and the earliest among them are [7] and [8]. It is reported in [8] that the concepts of TPS such as elimination of waste, leveled production, and automatic detection of abnormal conditions were adapted which resulted in significant improvements in both of their processes and organizational climate. Several work also trace the concepts of lean software develop to lean manufacturing arising out of TPS.

An experiment was performed to adapt several of the rules and principles of TPS to the mobile phone (feature phones, not smart phones) software development process with the objective of improving the software development process and improving the quality of software. Since the results are proprietary, this paper presents the results of the study, observations, and recommendations. It also presents qualitative descriptions of outcomes, as appropriate.

In this paper, we consider four ‘rules’ in TPS – three of these are design rules, and one is an improvement rule. We believe that the spirit behind several of the core principles of TPS could be used to benefit any process. The software development process has several radically different characteristics compared to a manufacturing process. Despite this, the spirit of these TPS rules could be adapted and adopted to improve software development practices. We first describe the adaptation of the four TPS rules for improving software development. Subsequently, we describe several other best practices that could be used to improve the software development process.

In the next section we present some background of mobile (feature) phone software and development scenario, and motivation for this work. Sections 3 and 4 describe respectively, the adaptation of some of the TPS rules and TPS principles. This is followed by conclusions.

2. BACKGROUND

The mobile phones industry is fast-paced and highly competitive. Newer models with ever expanding and improving features need to be introduced by mobile phone vendors for survival in the marketplace. Software for the newer phone models, as well as software updates, must be deployed in quick succession, without compromising quality and with low software maintenance costs.

The mobile phone software is highly complex and would necessarily have to deal with bugs. The code base for a typical mobile (feature) phone consists of about 12,000 source files, with about 40% of them being header files and resource files. The software development for a mobile phone seldom starts from scratch. The broad process usually involves starting from a baseline code of a similar model and then adding new features/functionalities and/or customizing existing features/ functionalities. Innovative software development processes which improve the quality in a cost-effective way, reduce development times, and lower maintenance costs are highly desirable. Recently, several studies are being conducted in the mobile software reliability domain (a couple of examples being [9], [10]).

The study of this paper dealt with the software development scenario wherein (a) the UI (user interface) or feature of a given phone model of a particular operator had to be customized for the same or similar hardware but for a different operator, (b) the UI or feature had to be adapted for a similar but different phone model for the same operator, (c) new feature(s) had to be developed for a given model. In this scenario, the actual implementation of a new feature spanned 2–6 weeks, whereas the subsequent addressing of bugs, and modifications due to some changes in the requirements were done over a period of about 12–15 weeks. Thus the better part of developers' times were spent in fixing bugs as opposed to the actual development. It was our belief that by adapting TPS rules and principles to the software development process, the software development could be done in lesser time with higher quality and less cost.

The significant phases of any software development project are: (a) requirements acquisition, (b) requirements analysis, (c) specification, (d) design, (e) development, (f) testing, and (g) deployment and maintenance. Irrespective of the development model (waterfall model, spiral model, etc), these phases have to be executed.

2.1 Major differences between manufacturing and software development

Since TPS basically arose in the manufacturing domain, in order to adapt it to the software development, we need to look at their basic characteristics in order to understand and strategize the adaptation of TPS rules and principles. Some of the major characteristics are given Table 1 below.

Table 1. Some of the major characteristics of manufacturing and software development

Manufacturing	Software Development
Several automated processes.	Highly human-intensive.
Well-define engineering methodologies. For example, an engineering drawing for a component is unambiguous and would have the same interpretation.	Immature engineering methodologies (more of craft). For example, the requirements are often ambiguous and subject to different interpretations by different people.
Almost every quantity is measurable using standardized techniques and instruments.	Most quantities defy accurate and standardized measurements. There is lack of metrics and effective techniques for such measurements.
Long history.	Relatively short history.
Linear scaling in complexity.	Super-linear (quadratic) scaling in complexity.
Failures (wear/tear) of machines well studied (MTTF, MTBF).	Bugs crop up unexpectedly; Time to fix bugs is hard to determine.
Fixes for faults are highly localized.	Fixes for bugs tend to have side effects.
Time to fix a problem after detection is well defined/estimated.	Time to fix a bug after detection is not well defined.
Individual quality-tested components when put together, would (almost always) result in a working subsystem.	Individual tested modules (components) when put together, would (almost always) fail integration tests.

In order to adapt some of the efficiency-improving practices in manufacturing to software development, it is important to identify the components of software development which are repetitive and make them amenable for measurements and automation.

3. ADAPTING SOME TPS RULES

In this paper, we focus on a few of the TPS rules and a few of the TPS principles and present our analyses on how to adopt them in the software development process.

3.1 TPS rules

First, we will consider four rules in TPS – (1) how people work; (2) how people connect; (3) how the production line is constructed; (4) how to improve. These rules are concisely presented in Figure 1 below, followed by brief descriptions.

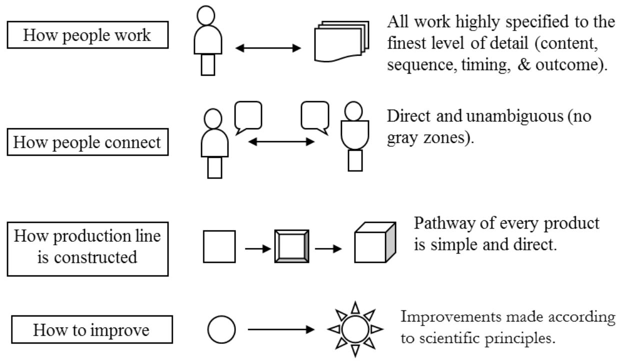
How people work
All work highly specified to the finest level of detail (content, sequence, timing, & outcome).
How people connect
Direct and unambiguous (no gray zones).
How production line is constructed
Pathway of every product is simple and direct.
How to improve
Improvements made according to scientific principles.

Fig 1: Four of the TPS rules

1. How people work: All work is decomposed into fine-grained activities, each of which is highly specified with respect to content, sequence, timing, and outcome. This ensures that the outcome of each activity is highly consistent irrespective of the person performing it. This principle is applied not just to the repetitive activities, but to the activities of all the people irrespective of their functional specialty or hierarchical role. The detection of any deviation from performing any of the activities which have been so clearly laid out in detail would become easier and corrective action could be taken quickly. If the deviations occur frequently, then the process is redesigned.

2. How people connect: Person-to-person communications are standardized, direct, and unambiguous, with no gray zones. This makes the requests for goods, services, and assistance very clear with regard to how to make requests, to whom to make requests, when to make requests. Also, it is clear as to who provides the services, to whom, and when.

3. How the production line is constructed: Pathway of every product and service is simple and direct. Generally, there are no forks and loops to convolute simple paths. This principle is also applicable to services (e.g. help requests) as well. One important thing to note in the flow is that the product does not go to the next available machine, but rather to a specific machine down the path. Similarly, the service will not come from the first available person, but rather from a specific person. Although the products (and services) flow in simple, well-defined paths, the production line is flexible – the paths accommodating many different types of products.

4. How to improve: The first step in improvement is to identify problems and to look for opportunities for improvements. People are explicitly taught how to improve, in addition to learning from personal experience. The improvements are made according to scientific principles at the lowest possible level in the organization under guidance of a teacher. An improvement effort is designed as an experiment with an explicit, clearly specified, and verifiable hypothesis ("If the following specific changes are made, we can expect to achieve this specific outcome").

3.2 Adaptation of TPS rules

In this section, we describe how to adapt the TPS rules to software development practices. In Tables 2a – 2d, we present the current (general) practices in software development in the realm of the four rules of TPS outlined above, and then describe the corresponding proposed practices which incorporate the spirit of the TPS rules.

Table 2a. Current and proposed practices related to "how people work"

How people work	
Current practice	**Proposed practice**
Many times customer requirements are written down in detail at the beginning of the project. However, most changes to specifications in the later stages seem to be less detailed and formal. Often times, it may be just a few emails and/or notes from phone or oral discussions. This leads to semantic gaps between the customer and the project leader, as well as between the project leader and team members.	Any changes in the requirements are clearly written down (documented), and then communicated to the team members. This reduces gaps in communications and understanding, and avoids re-work (which is waste) later in the project. The effort / time in clearly writing/documenting the changes would pay back (several times over) over the lifetime of he project.
Specifications from the project manager to team members are informal or absent.	Project manager decomposes the customer's requirements specifications into smaller chunks and writes a concise description for each of them. This can be done in collaboration with the team members. Breakdown the functionalities to as low level as possible, while clearly describing each functionality at every level
Specifications of work shared / work dependencies among team members are either informal or absent, due to lack of any standardized way.	Any significant interactions, work sharing, and dependencies among team members should be clearly laid out and documented.
Documentation of the progress of work by team members is absent.	At the beginning, each team member writes a 1-page description (text + figure) of the finer details of the functional and interface requirements of the component that one is responsible for. The textual descriptions should be put in the code as comments (just cut-n-paste), with some key-words (for quick searching). Any changes in the implementation are documented. Note: the documentations need to be concise and clear, not too verbose or elaborate.
Recommendation: A simple, standardized format should be developed for the various documentations. There could be a central 'scribe'. All descriptions from project manager and team members are rough but complete and communicated to the scribe. Scribe converts them to a concise, clearly specified document. This provides uniformity and single style, and saves reading time.	

Table 2b. Current and proposed practices related to "how people work"

How people connect	
Current practice	**Proposed practice**
The 'project leader – team members' communications is primarily via meetings, emails, and phone calls. There is virtually no written documentation.	Project leader – team members Meetings, Emails, Phone calls, Clear and concise (1-2 pages) written documentation of functionality, interface requirements expected from team members
Among team members: Emails, Phone calls, Discussions, Practically no written, descriptive communication	Among team members Emails, Phone calls, Discussions, Clear and concise (1-2 pages) written descriptive communication Note: The written communications are only for significant and non-trivial requirements/change communications
Recommendation: The written communications should not be too many, rather only for significant and non-trivial ones. Written communications have several advantages. While writing, several hidden aspects of design/change may emerge. It facilitates clear articulation of ideas and tasks to be performed. It overcomes several inherent ambiguities of oral/email communications. Over the life of the project, it saves tremendous time lost due to miscommunications and re-work. It also serves as a good documentation during the project life cycle, post-delivery, and also perhaps for other projects	

Table 2c. Current and proposed practices related to "product line construction"

Production line construction	
Current practice	**Proposed practice**
Work to be done is partitioned into modules. Modules assigned to team members.	Divided the modules further into finer grained components. Look for opportunities of independent development of components. This increases concurrent development; decreases sequentiality and dependencies. Independent components could be distributed among team members for development and then combined together. Thus, work could be more balanced among team members.
Team members work on their	Develop precise interface requirements between components and between modules. Testing of a

assigned modules. Team members perform unit tests on their modules.	component or module can proceed even when other components or modules on which it depends are not ready, by use of suitable 'stubs'.
Test engineers perform system / integration tests when a workable system is ready.	By making complete releases very frequently, albeit with smaller sets of features, the integration tests can be performed almost continuously. Thus, a working product (although with limited features) is always available.
Recommendation: Frequent builds have to go together with 'smoke tests'. The smoke test need not be exhaustive, but it should be capable of exposing major bugs. Frequent builds and integration tests have several advantages. Generally debugging would take longer if the integration occurred later. They facilitate quality problems to be kept under check. They enable location of the bugs more easily since not many changes would have occurred between successive builds where the bugs appear.	

Table 2d. Current and proposed practices related to "how to improve"

How to improve	
Current practice	**Proposed practice**
Occasional code improvements	Continuous code improvements – develop a methodology for code improvement on a regular or need basis. For example, code/algorithm improvements to improve the response times or battery power
Lack of rigorous principles	Develop scientific principles – based on Computer Science and Software Engineering. For algorithmic improvements, it is easy to analyze the improved algorithm and estimate the improvements. Thus the improvements can be formulated as hypothesis. Tools and metrics should be used for measuring improvements. Design suitable experiments under which hypotheses are tested
Manager – team-member relationship	Teacher / Tutor guidance is used to teach the skills required to look for (a) opportunities for improvements, (b) developing experiments / schemes for improvements, and (c) implementing the improvements (Doesn't teach any particular improvement to a component)
Inadequate grass roots participation in improvements	Foster a culture of participation in continuous improvements (competitions, incentives, rewards) at the lowest possible level in the organization. (Managers act as tutors and facilitators)
Recommendation: Develop appropriate development methodology incorporating sound scientific/engineering principles. Institute some form of incentives for product / process improvements.	

4. ADAPTING SOME TPS PRINCIPLES

4.1 Some principles of the TPS

Some of the major principles (elements) of the TPS which are relevant to software development process are given in this section.

Elimination of waste. This is one of the main themes of TPS. Many of the tools and techniques are focused on this principle. The broad definition of waste is "anything other than the minimum amount of equipment, space, and worker's time, which are not absolutely essential to add value to the product" (Fujio Cho, President, Toyota).

Kaizen. 'Kaizen' refers to 'continuous improvement'. The basis for this principle is that a large number of small improvements in processes are easier to implement and would have a significant cumulative effect than a few large-scale improvements.

Jidoka. This refers to the stopping the assembly line when a problem is encountered at any workstation, so that the problems do not propagate. The cause of the problem is detected and immediate and permanent solutions are put in place.

Mixed model production. This refers to the practice of building multiple models on the same assembly line simultaneously, rather than in large batches.

Pokayoke. This refers to the use of a variety of devices and techniques to prevent the occurrence of defects. (ex. using an attachment to gasoline tank cap to prevent it from being lost).

Heijunka. This refers to 'leveled production' – the distribution of work and exchange of knowledge. This ensures that all the employees engaged in the production of a product have about the same share of work and about the same level of knowledge about the work.

Work standardization. This refers to the development of specifications for the exact manner of performing a task and adhering to it. This ensures that workers execute their tasks in a well-defined manner and results in variations in different work methods.

Design of experiments for improvements. The proposed improvements to a process are designed as experiments with a explicit, clearly specified, and verifiable hypotheses ("If the following specific changes are made, we can expect to achieve this specific outcome").

4.2 Adapting some of the principles of the TPS to software development

A summary of the adaption of several TPS principles (elements) to the domain of software development is given in Table 3 below. The columns of the matrix give the stages in software development process, while the rows give the principles of TPS. The cells briefly outline how a particular TPS principle is adopted to a given software development stage.

Table 3. Some of the TPS principles adapted to software development stages

	Requirements Analysis / Specification	**Design**	**Development**	**Testing**	**Maintenance**
Elimination of waste	Get as clear requirements as possible using standard/known templates, as applicable. Develop precise specifications. Iterate quickly before moving to development.	Design reusable components.	Develop reusable code; Automate repetitive work; Eliminate re-work.	Design test cases judiciously. Focus on the 20% which cause 80% of the faults.	Develop and maintain clear error logs and update / maintenance documents.
Kaizen (continuous improvement)	Develop more precise requirements / specs learning from past mistakes	Use frequent design reviews and improvements	Frequent code refactoring.	Perform regression test even when a small feature is implemented.	Incorporate lessons learnt from previous upgrades into current/future cycles
Jidoka (automatic error detection and stopping propagation)	Eliminate / minimize ambiguous specifications as early as possible.	Design review: early prevention of errors getting into code.	Code review: early prevention of error propagation in later versions.	Reexamine requirements / specs and/or design upon serious errors.	Go back to redesign as applicable. Proactively examine similar models based on the same baseline.
Mixed model production	Maintain a document with differences between similar models based off of the same baseline (Delta document)	Design for 'device independence' / support for heterogeneity.	Use of 'code guards' for conditional compilation; Parameterize device specific characteristics (ex. LCD size).	Effectively use test schemes/cases of previous models for current models based off of the same baseline.	Design / schedule upgrades for models suitably (ex. batching models using same baseline).
Heijunka ('leveled production' – the distribution of work and exchange of knowledge)	Make use of team members from different groups (ex. Applications, Firmware, UI, etc.) of the same project to gather the related requirements.	Decompose modules into smaller components so that more people can concurrently work on bigger modules.	Concurrent development of several modules with well-defined interfaces by many teams.	Develop a distributed test plan for concurrent testing, taking into account the interactions among modules.	Decompose the upgrade / maintenance into independent tasks which can be carried out concurrently.
Work standardization	Use formal models (ex. UML).	Use standard design templates (STL) and design patterns as much as possible.	Develop and adopt standardized programming methods (Norms on function sizes, naming conventions, inter-module communications, documentation, etc).	Use standardized testing procedures.	Develop and use standardized upgrade / maintenance procedures.

5. CONCLUSIONS
The Toyota Production System (TPS) has been widely studied and copied by various companies to improve several of their processes. TPS principles have also had significant influence on lean software development. This paper presented the adoption and adaptation of the spirit behind some of the rules and principles in TPS to the software development process in a mobile phone software development scenario. The results were quite positive with better use of developer times, smaller turn-around times, and better quality.

6. REFERENCES
[1] T. Ohno 1998. Toyota Production System: Beyond Large Scale Production. Productivity Press

[2] J. Liker. 2004. The Toyota Way, McGraw-Hill.

[3] Hino, S. 2006. Inside the Mind of Toyota: Management Principles for Enduring Growth, Productivity Press.

[4] J. Liker and J. Morgan. 2006. The Toyota Product Development System, Productivity Press.

[5] S. Spear and H.K. Bowen. 1999. Decoding the DNA of the Toyota Production System. Harvard Business Review. (Sep. – Oct. 1999) 96–106.

[6] J. Womack and D. T. Jones. 1996. Lean Thinking, Free Press.

[7] T. Sekimura and T. Maruyama. 2006. Development of Enterprise Business Application Software by Introducing Toyota Production System. Fujitsu Sci. Tech. J. 42(3), 407–413.

[8] K. Furugaki, *et. al*. 2007. Innovation in Software Development Process by Introducing Toyota Production System. Fujitsu Sci. Tech. J. 43(1), 139–150.

[9] S. Malek, *et. al*. 2009. Improving the reliability of mobile software systems through continuous analysis and proactive reconfiguration. *International Conference on Software Engineering*, May 2009 (Companion Volume 978-1-4244-3495-4).

[10] S.R. Subramanya. 2011. Analysis of Some of the Root Causes of Bugs in a Mobile Phone Software Development Environment. *International Conference on Computer Applications in Industry and Engineering*, Honolulu, HI, Nov. 2011, 210–215.