Jidoka in Software Development

Emanuele Danovaro
Center for Applied Software Engineering
Free University of Bolzano/Bozen, Italy
emanuele.danovaro@unibz.it

Andrea Janes
Center for Applied Software Engineering
Free University of Bolzano/Bozen, Italy
andrea.janes@unibz.it

Giancarlo Succi
Center for Applied Software Engineering
Free University of Bolzano/Bozen, Italy
giancarlo.succi@unibz.it

Abstract

Lean management is based on two concepts: the elimination of "Muda", the waste, from the production process, and "Jidoka", the introduction of quality inside the production process and product. In software production, the elimination of Muda received significant attention, while Jidoka has not yet been fully exploited. In this work we want to propose a holistic approach to insert Jidoka in software production. We depict the architecture of a tool to support Jidoka and describe the components that are part of it.

Categories and Subject Descriptors D.2.8 [**Software**]: Engineering – Metrics

General Terms Management, Measurement

Keywords Quality assurance; Jidoka

1. Introduction

Short after WWII Taiichi Ohno and Shigeo Shingeo revolutionized the Toyota Production System with the idea of lean production (Ohno 1988). Because of their visible and tangible success, their ideas were successfully exported from Japan to the Western world.

The Toyota Production System advises to eliminate from the production process all activities that do not produce value to the customer (i.e., Muda).

The philosophy to focus on customer satisfaction as a wayto increase flexibility came back in the '90 with the book "Lean Thinking" by Womack and Jones (1996). Lean Thinking brought the lean idea into new industries such as the pharmaceutical industry (Petrillo 2007) and software development (Poppendieck and Poppendieck 2006). Agile methods are a group software development methodologies that put the ideas of lean thinking into the practice of software development (Beck et al. 2001).

Ohno identifies two references for lean production: Just-in-time production and Jidoka. The elimination of Muda is a requisite for Just-in-time production, where the resources needed to complete a certain step are made available at the latest possible moment. Jidoka is often translated with "autonomation" or "automation with a human mind" and is usually illustrated making the example of a machine that can detect a problem with the produced output and interrupt production automatically rather than continue to run and produce bad output (Ohno 1988; Monden 1993). Some authors translate Jidoka with "quality-at-the-source" (Standard and Davis 1999) meaning that quality is inherent in the production system and is not checked after the process. In essence Jidoka is composed by two parts: a mechanism to detect problems, i.e., abnormalities or defects, and a mechanism to interrupt the production line or machine when a problem occurs (Monden 1993).

We think that the elimination of Muda has received a significant attention in software production, for instance in the analysis of the value stream, the focus on activities that provide value, the deferral of commitment of irreversible decisions to just the moment when it is needed (Poppendieck and Poppendieck 2006; Beck 1999). Jidoka has not received equal attention, in our view. An indicator that the concept of Muda is much more popular than Jidoka is that searching in Google for "Muda" results in about 21,000,000 hits, whereas searching for "Jidoka" results in 39,600 hits (on July 1st, 2008).

To our knowledge, all proposals to insert Jidoka in software production relate to automated testing and continuous integration. We agree that automated testing and continuous integration (and the use of tools supporting it such as

SIGPLAN'05 June 12–15, 2005, Location, State, Country.

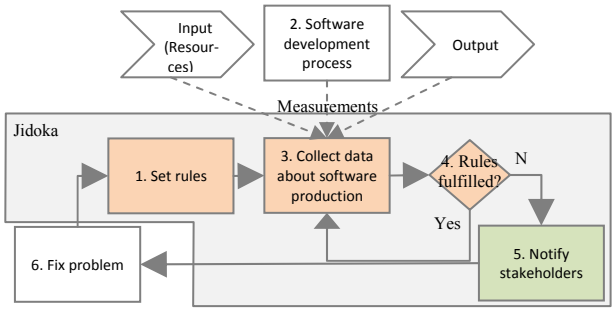

Figure 1. Data flows within a tool to support Jidoka in software production

jUnit[1] and CruiseControl[2] implement the idea of Jidoka and are extremely important, but this is only one aspect of quality in software development. Also other quality attributes about the code produced and the process can and should make use of Jidoka during software production.

In this article, we present the idea underneath a tool to promote Jidoka in software production.

The article is organized as follows: section 2 gives an architectural overview of a tool following Jidoka, section 3 describes our mechanism to identify software production problems, and section 4 describes our mechanism to interrupt software production. We conclude giving an outlook on future work.

2. General architecture

We depict a general architecture of a tool to support Jidoka in software engineering in figure 1. The dashed lines represent measurements, the solid lines execution flow. As said before, the idea of Jidoka consists of two mechanisms, one to detect problems (elements in orange), and one to interrupt the production when problems occur (elements in green). Both of these mechanisms need to be tailored specifically to the software domain.

This schema can be seen as an instance of the well-known Deming cycle "Plan, Do, Check, Act" (Deming 2000); in our architecture, the steps 1 and 2 correspond to "Plan" and "Do", the steps 3 and 4 to "Check" and the steps 5 and 6 to "Act". The idea to continuously monitor software artifacts and to alert the developer of possible mistakes or problems is not new. It has been proposed in the past e.g., within design tools like ArgoUML (Robbins and Redmiles 2000), in which a critiquing system constantly monitors the ongoing UML design and generates warnings

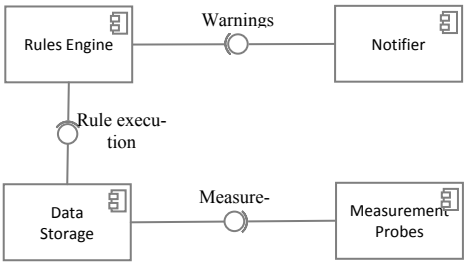

Figure 2. Architecture of the Jidoka tool.

of possible problems. Tools like Findbugs[3] or PMD[4] scan Java source code and looks for potential problems, which can be defined by rules written in Java or using XPath expressions. A last example is Holmes (Succi et al. 2001) which gives feedback on aspects of code quality specific for the domain of software product line development. The proposed approach introduces two novelties: first, we measure not only the produced artifacts but also the used resources as well as the performed activities to obtain a complete picture of software production, and second, we propose to use rules to build quality into the process, i.e., to enforce these rules without the need of a person to constantly check them.

In our view, the "push" paradigm is currently dominating the area of tools to ensure software quality: the user has to regularly run a tool to discover which rules are not fulfilled and to decide if a corrective action is required, the burden lies on the user. We want to propose a paradigm shift towards a "pull" paradigm as suggested by lean thinking (Ohno 1988) or proposed in other domains such as product line development (Succi et al. 2001), in which the user defines properties of critical situations and the system performs the regular check for him or her.

The architecture of our system is shown in figure 2. The Measurement Probes continuously extract data from the development process about the input (the used resources), the produced output, and the activities carried out without the need of manual intervention by developers. Such data are stored in the Data Storage. The Rules Engine contains a set of rules, retrieves the data from the data storage, analyses them, and generates warnings if a rule is violated. The Notifier contains the actions to perform if a rule is violated.

In the following we will discuss the components "Measurement Probes" (section 3), "Rule Engine" and "Notifier" (both in section 4). The data storage component, responsible of storing all data collected by the tool is omitted due to lack of space.

[1] JUnit, http://junit.sourceforge.net

[2] CruiseControl, http://cruisecontrol.sourceforge.net

[3] FindBugs, http://findbugs.sourceforge.net

[4] PMD, http://pmd.sourceforge.net

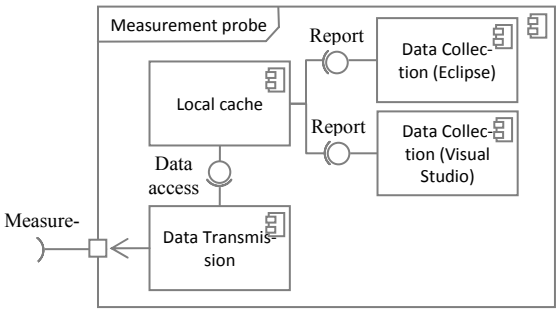

Figure 3. Architecture of measurement probes

3. Identifying problems in software production

Jidoka means not allowing defective parts to go ahead in the development process. Therefore, we need a clear criteria, or rule, to decide whether a software artifact is ready to proceed to the next production step. To evaluate these criteria, suitable data have to be collected.

In principle, only necessary data are collected to avoid wasting resources. Our architecture allows such an approach; measurement probes tailored to a specific case can be developed and connected to the data storage. However, taking such rigid approach requires to develop new, or to adapt existing measurement probes every time the rules are changed. To increase the flexibility of the approach, we have developed three generic probes that collect detailed data about **(a)** consumed resources (output), **(b)** ongoing activities (process) and **(c)** produced artifacts (input). With such approach if later an additional piece of information is needed, it can be obtained just by elaborating the available data.

The following subsections describe the measurement probes. The architecture of all probes is similar, as shown in figure 3: the collected data is stored in a local cache and submitted as soon as a network connection is available (see figure 2).

3.1 Collecting data about resource consumption

The main cost driver in software development projects is typically the effort (Jorgensen and Shepperd 2007). To collect data about resource consumption we focused on collection of time spent creating and modifying artifacts like code and documentation.

We developed a set of probes that have to be installed on the machine of every developer. We have two types of probes: probes to track the time spent editing documents with applications like Microsoft Office[5] and OpenOffice[6], and probes to track the time spent editing source code within IDEs like Eclipse[7], Microsoft Visual Studio[8], Sun NetBeans[9], and JetBrains IntelliJIdea[10]. For each entity (i.e. an OpenOffice document, a Java method) we collect the time spent creating and editing it, and the user who did the modification.

Not all effort is spent in front of the computer. Effort spent off-line has to be entered manually, for this purpose we provide a measurement probe that allows the user to enter activities manually, defining the time spent and effort category.

3.2 Collecting data about process output

The output of the software development process are artifacts like source code and documentation. Each artifact is characterized by properties which help to decide whether it fulfills the requirements or not. We can distinguish two ways to test the output of a software development process: statically and dynamically (Fenton and Pfleeger 1998). The dynamic testing, i.e., through the execution of the source code is performed by tools like CruiseControl or TeamCity[11]. This is a Jidoka approach: on a regular basis they download source code and run all unit tests (this is "the mechanism to detect problems") and notify errors to the developers (this is "the mechanism to stop the production line").

Our measurement component for the inspection of the output focuses on the second aspect - the static analysis. Our component regularly checks out the source code and calculates the Chidamber and Kemerer (Chidamber and Kemerer 1994) object oriented software metrics. The obtained measurements are transmitted to the Data storage component.

3.3 Identifying activities

According to our experience, the identification of activities is not a straightforward process. Depending on the context, the term "activity" refers to a different set of actions carried out during software production. In one case editing a specific artifact leads to the identification of an activity, e.g., the time spent editing a class that begins with the let-

[5] Microsoft Office, http://www.microsoft.com/office

[6] OpenOffice.org, http://www.openoffice.org

[7] Eclipse.org, http://www.eclipse.org

[8] Microsoft Visual Studio, http://msdn2.microsoft.com/vs2008/products

[9] Sun NetBeans, http://www.netbeans.org

[10] JetBrains IntelliJIDEA, http://www.jetbrains.com/idea

[11] TeamCity, http://www.jetbrains.com/teamcity

ters “test” could be attributed to the “testing” activity. In another case, observing a sequence of steps is needed to identify an activity, e.g., “test first”. To allow a contextualized definition of “activity”, we implemented the component responsible to identify activities with a rule based approach. Rules defined using a Prolog syntax are regularly run in order to identify the occurrence of an activity within a specific time span. This approach allows also the modification of rules “a posteriori”, i.e., rules can be changed according to the needs and rerun on the past data to have a coherent view of the entire data.

The output of this component is the sequence of identified activities within a specific time span, together with the respective timestamps and users.

4. Interrupting the software production line

Interrupting production is done to prevent further damageand waste of resources. Within software production an example hereof is the use of “check-in policies” in source control systems, which define conditions that a developer has to meet to check in modified code.

In our system, the criteria (i.e. the knowledge) to decide whether to stop software production or not are specified as a set of rules. The Rule engine fetches data from the Data Storage and checks the collected measurements against userspecified rules. The Notification component is triggered, if a rule is violated.

The quality of the rules has a high impact on the utility of the here proposed system. We suggest to align the rules put into this system with the business goals to create rules that are perceived by all stakeholders as purposeful. The rules defined in this tool have to lead to a decision whether to interrupt the production line (in our case the software production) or not. It’s alignment to the business goals allows the user to understand the considerations that led to the conclusion that something is wrong. Furthermore, such an alignment involving all stakeholders in setting the goals before actual implementation has been found to improve organizational performance (Gopal et al. 2002).

A systematic way to do this alignment is given by the GQM method, which “is based upon the assumption that for an organization to measure in a purposeful way it must first specify the goals for itself and its projects, then it must trace those goals to the data that are intended to define those goals operationally, and finally provide a framework for interpreting the data with respect to the stated goals” (Basili and Rombach 1987).

We envision two mechanisms to “stop the production line”: **(a)** an “alert lamp” as implemented in the Eclipse IDE, and for users of other development environments **(b)** an application that is shown in the tray area visualizing alerts. The first mechanism, within Eclipse, shows an alert lamp together with the description of the violated rule, next to the “incriminated” code element. This creates visibility since every developer working on that project sees that there is a problem. The second mechanism is an application shown in the tray area, which visualises the rules that have been violated.

5. Conclusion and future work

In this work, we present a novel scenario in software development. We demonstrate, how quality assurance can be “built into the process”, as proposed by lean thinking. We give an example of how to integrate the continuous monitoring of produced artifacts, ongoing development activities, and consumed resources with the production process together with the knowledge about unacceptable values of the monitored data.

Further research is needed to establish **(a)** which other types of knowledge can be integrated in this way – some examples are: the adherence to a defined workflow, advice about testability, warnings about the violation of user interface guidelines, and **(b)** which ways to interrupt software production lead to an easy adoption and maximize the outcome of this approach. The requirements of all stakeholders, management, developers, customers have to be considered.

We aim to inspire others to follow our line of research in trying to “build quality into” the software development process and in this way pay attention to the quality of artifacts, even at early stage of the development.

References

V. R. Basili and H. D. Rombach. Tailoring the software process to project goals and environments. In ICSE ’87: *Proceedings of the 9th international conference on Software Engineering*, pages 345–357, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

K. Beck. *Extreme Programming Explained: Embrace Change*. Addison Wesley Professional, 1999.

K. Beck, M. Beedle, van A. Bennekum, A. Cockburn,W. Cunningham, M. Fowler, J. Gren-ning, J. Highsmith, A. Hunt, R. Jeffries, J. Kern, B. Marick, R. C. Martin, S. Mellor, K. Schwaber, J. Sutherland, and D. Thomas. Manifesto for agile software development, 2001. http://www.agilemanifesto.org.

S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

W. E. Deming. *Out of the Crisis*. The MIT Press, August 2000.

N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*, Revised. Course Technology, 1998.

A. Gopal, M.S. Krishnan, T. Mukhopadhyay, and D. R. Goldenson. Measurement programs in software development: De-

terminants of success. *IEEE Trans. Softw. Eng.*, 28(9):863–875, 2002.

M. Jorgensen and M. Shepperd. A systematic review of software development cost estimation studies. *IEEE Trans. Softw. Eng.*, 33(1):33–53, 2007.

Y. Monden. *Toyota Production System*. Industrial Engineering Press, Norcross, GA, 2nd ed. edition, 1993.

T. Ohno. *Toyota Production System: Beyond Large-Scale Production*. Productivity Press, Cambridge, MA, USA, 1988.

E. W. Petrillo. Lean thinking for drug discovery - better productivity for pharma. *DDW Drug Discovery World*, 8(2):pp. 9–16, 2007.

M. Poppendieck and T. Poppendieck. *Implementing Lean Software Development*: From Concept to Cash. Addison-Wesley Professional, 2006.

J. E. Robbins and D. F. Redmiles. Cognitive support, uml adherence, and xmi interchange in argo/uml. *Journal of Information and Software Technology*, 42(2):79–89, 2000.

C. Standard and D. Davis. *Running today's factory: a proven strategy for lean manufacturing*. Hanser Gardner Publications, Cincinnati, 1999.

G. Succi, J. Yip, and W. Pedrycz. Holmes: an intelligent system to support software product line development. In ICSE '01: *Proceedings of the 23rd International Conference on Software Engineering*, pages 829–830, Washington, DC, USA, 2001. IEEE Computer Society.

J. P. Womack and D. T. Jones. *Lean thinking: Banish waste and create wealth in your corporation*. Simon & Schuster, New York, NY, USA, 1996.

t