Personal Software Process with Automatic Requirements Traceability to Support Startups

Waraporn Jirapanthong*

College of Creative Design and Entertainment Technology, Dhurakij Pundit University, Bangkok, Thailand

Abstract: This paper applies Personal Software Process (PSP) for software development activities, and uses PSP scripts to follow the activities in software development. In particular, we have adapted a development script in order to enable automatic traceability. The script is the cyclical process that is designed for developing a large program in a sequence of small incremental steps. Moreover, we have extended an XTraQue tool to enable an automatic traceability during using PSP. This enables the completeness of traceability during using PSP. The Part-of-Speech (POS) embedded XML-based templates of software artefacts for PSP-based development, that is, functional requirements (FR), use case, and class diagram are defined. We perform an explanatory case study in order to evaluate the effectiveness between manual and automatic traceability during the personal software process (PSP). In particular, the causal links between software artefacts created during software development are so-called traceability relations. The result evaluation are concerned with precision and recall measures on the creation of traceability relations.

Keywords: Personal Software Process, Software Improvement Process, Requirements Traceability, Incremental Development.

1. INTRODUCTION

According to Personal Software Process (PSP), the process drives a software developer improving their own performance by controlling and managing their work. It is a structured framework of forms, guidelines, and procedures for developing software. The process is driven by scripts through the process steps, namely design, code, compile, test, and postmortem steps. Additionally, traceability is included an activity during the process. However, there are still difficulties to use traceability records or relations in order to improve the software process.

A research question is whether it is more effective if PSP-based software development is supported by automatic traceability. We are concerned from the perspective of software developers, particularly when they work individually on the process. We have adopted several forms based on PSP and adapted some development scripts that support the personal software process, and also extended a traceability tool to enable it during PSP, which enables the completeness of traceability. Some of the software artefacts for PSP-based development, that is, use case and class diagram, are extended for the process. We have created a scenario for testing. A participant who takes the role of the software developer was asked to perform software development activities under PSP, and to follow the adapted development script. The case study was created to explore the experiences of our approach.

The remainder of the paper is organized as follows. Section 2 provides the background and related work to the research, Section 3 describes the research methodology, Section 4 presents the results evaluation and discussion, and a conclusion is given in Section 5.

2. BACKGROUND AND RELATED WORK

2.1. Personal Software Process

A Personal Software Process (PSP) is a self-improvement process that drives a software developer to control, manage, and improve their work (Humphrey, 2005). It is a structured framework of forms, guidelines, and procedures for developing software. The purpose of PSP is to assist a software developer to improve their software engineering skills. The baseline process, PSP0, provides a framework for writing the first program, and for gathering data on work. As shown in Figure **1**, the PSP0 process is driven by scripts which guide the work.

The scripts guide software developers through the process steps, the logs are recorded for process data, and the plan summary provide a summary record and reports. In the planning step, a software developer plans to do the work. The development steps include design, code, compile, and test. In the postmortem step, a software developer compares thier actual performance with the plan, and produces a summary report. There are three main process elements in PSP0, namely the planning, development, and postmortem phases.

*Address of correspondence to this author at the College of Creative Design and Entertainment Technology, Dhurakij Pundit University, Bangkok, Thailand; Tel: +66 29547300; Fax: +66 29548651; E-mail: waraporn.jir@dpu.ac.th

JEL: C88, L86, M13, M15.

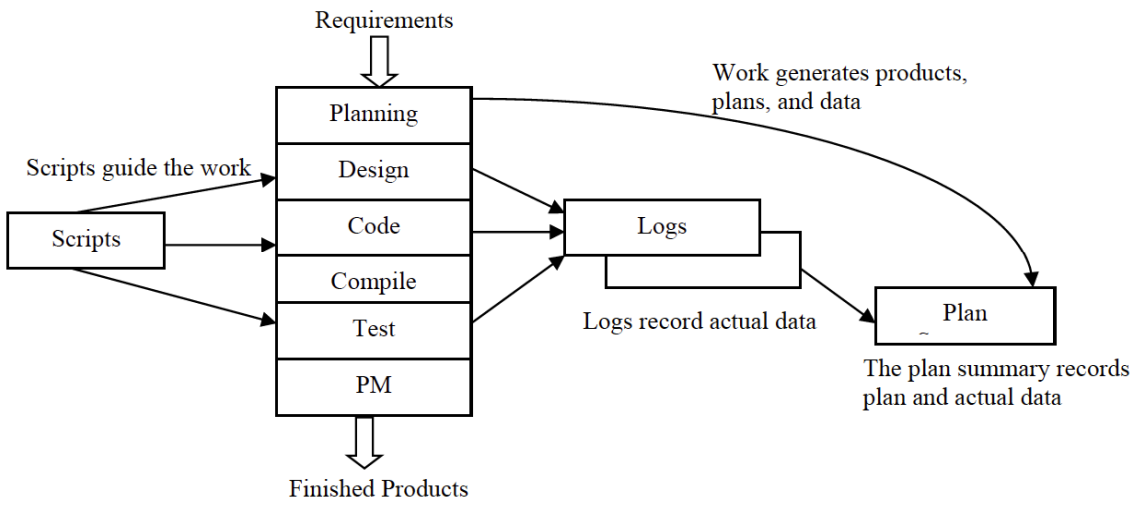

Figure 1: PSP0 Process Flow (Humphrey, 2005).

For the postmortem phase, three main activities are defined in the scripts, namely project review, defect recording, and time recording, as appears in the following activities during the postmortem phase:

- The activity called *defect recording* aims to:
  - review the project plan summary to verify that all of the defects found in each phase were recorded;
  - use recollection, and record any omitted defects.
- The activity called *defect data consistency* aims to:
  - check that the data on every defect in the defect recording log are accurate and complete;
  - verify that the number of defects injected and removed per phase are reasonable and correct;
  - use recollection, and correct any missing or incorrect defect data.
- The activity called time aims to:
  - review the completed Time Recording log for errors or omissions;
  - use recollection, and correct any missing or incomplete time data.

According to (Humphrey, 2005), the authors have proposed a template of defect recording log. The log document contains information, for example, a software developer's name, and program's name. The log document shows a list of defect log which consists of: i) *projectidentifier*, ii) *fixingdate*, iii) *uniquedefect-number*, iv) *defecttype*, v) *injected phase*, vi) *remove phase*, vii) *fix time*, viii) *fix reference*, and ix) *description.*

The *defecttype* is classified as a) *documentation* that refers a defect on comments or messages; b) *syntax* that refers a defect on spelling, and punctuation, typos, and instruction formats; c) *build,package* that refers a defect on change management, library, and instruction formats; d) *assignment* that refers a defect on declaration, duplicate names, scope, and limits; e) *interface* that refers a defect on procedure calls and references, I/O, and user formats; f) *checking* that refers error messages, and inadequate checks; g) *data* that refers a defect on structure, and content; h) *function* that refers a defect on logic, pointers, loops, recursion, computation, and function defects; i) *system* that refers a defect on configuration, timing, and memory; and j) *environment* that refers a defect on design, compile, test, or other support system problems.

The *injected* phase is one where a defect was injected. The *remove* phase is the one where a software developer found and fixed the defect. The *fixtime* is the time a software developer tool finds and fixes the defect. The *fix reference* notes the number of the defective fix which refers a mistake fixing one defect, and later finds and fixes the new defect. The *description* section is a note that describes the reason or location that the defect was fixed.

The authors proposed a template of time recording log. The log document contains information, for

example, a software developer's name, program's name. The log document shows a list of time log which consists of: i) *project identifier;* ii) *phase;* iii) *start date and time;* iv) *interruption time;* v) *stop date and time;* vi) *delta time;* and v) *comments.*

The *phase* is the one a software develop worked on, that is, planning, design, test. The *start data and time* is the date and time when a software developer worked on a process activity. The *interruptiontime* is duration time that was not spent on the process activity. The *stop date and time* is the date and time when a software developer stopped working on that process activity. The *deltatime* is the clock time that a software developer actually spent working on the process activity, less interruption time. The *comments* section is used to remind a software developer of any unusual circumstances regarding an activity.

2.2. Traceability

Software traceability has been recognized as an important activity in software system development (Ramesh and Jarke, 2001). In general, traceability relations can improve the quality of the software product being developed, and reduce the time and cost associated with the development. In particular, traceability relations can support the evolution of software systems, reuse of parts of the system by comparing components of the new and existing systems, validation that a system meets its requirements, understanding of the rationale for certain design and implementation decisions in the system, and analysis of the implications of changes in the system.

Support for traceability in software engineering environments and tools is not always adequate (Ingram and Riddle, 2012). Software developers may have a concern regarding the cost and benefits on traceability activity (Antoniol *et al.*, 2003), so that, despite its importance, traceability is rarely established. In order to alleviate this problem, other approaches have been proposed to support semi- or fully-automatic generation of traceability relations more recently (Egyed, 2005; Kim *et al.*, 2005; Marcus and Maletic, 2003; Jirapanthong and Zisman, 2009). Some of these approaches, such as the generated traceability relations, do not have well-defined semantic meanings that are necessary to support the benefits provided by traceability. Some approaches have defined semantic meanings to support the use of traceability relations. The traceability relations are generated between different types of software artefacts during the development of software systems.

3. RESEARCH METHODOLOGY

In order to learn and experience the situation of software development for startups, we chose to use a case study approach. Our case study design is considered because we want to cover contextual conditions as they are relevant to the phenomenon under examination.

3.1. Defining Research Questions

This research is based on the question: "Is it more effective if PSP-based software development is supported by automatic traceability?". The research proposes to discover the experiences of software developers following PSP using automatic software traceability activity is more effective than its manual counterpart.

3.2. Conducting Research

The research seeks to explain the presumed causal relationships between software artefacts under PSP-based development. We have used several forms to follow the activities in PSP (Humphrey, 2005) and a development script as shown in Table **1**, which is adapted from (Humphrey, 2005). It is a cyclical process that we applied for programming. The process is designed for developing a large program in a sequence of small incremental steps.

The paper extended an XTraQue tool (Jirapanthong and Zisman, 2009) to enable traceability during using PSP. This enables the completeness of traceability during PSP. According to the meta model in (Jirapanthong and Zisman, 2009), we have extended the templates of software artefacts for PSP-based development, that is, use case and class diagram. We also added a template of funcational requirements (FR) for specifying the user requirements.

In our templates, for example, a use case is composed of:

(1) Use_Case_ID – this attribute is identified as a use case;

(2) Title – the element Title is the title of use case;

(3) Description – the element Description is specified for a brief textual description;

Table 1: A Development Script

Purpose		To guide development of programs
Entry Criteria		Problem description or compoen
General		
Step	**Activities**	**Description**
1	Requirements and Planning	Obtain the requirements and produce the development plan. requirements document design concept size, quality, resource, and schedule plans Produce a master *Issue Tracking log*
2	High-level Design (HLD)	Produce the design and implementation strategy Functional Specifications State Specifications Operational Specifications Development Strategy Test Strategy and Plan
3	High-level Design Review (HLDR)	Review the high-level design Review the development and test strategy Fix and log all defects found Note outstanding issues on the *Issue Tracking log* Log all defects found
4	Development	Design the program and document the design in the PSP Design templates Review the design and fix and log all defects Implement the design Review the code and fix and log all defects Compile the program and fix and log all defects Test the program and fix and log all defects Complete the Time Recording log. Reassess and recycle as needed
5	Postmortem	Complete the Project Plan Summary form with the actual time, defect, and size data
Exit Criteria		A thoroughly tested program Completed Project Plan Summary with estimated and actual data Completed Estimating and Planning templates Completed Design templates Completed Design Review checklist and Code Review checklist Completed Test Report template Complete Issue Tracking log Completed PIP forms Completed Time and Defect Recording logs

(4) Level – the element describes the level of functionality that it describes within a system;

(5) Preconditions – the element describes the conditions that must be satisfied before its execution;

(6) Postconditions – the elements describes the conditions that must be satisfied after its execution;

(7) Primary_actors – the element specifies primary users of the use case;

(8) Secondary_actors – the element specifies secondary users of the use case;

(9) Flow_of_events – the element specifies a list of the events that triggers the use case and the specification of the normal events that occur within it. The element Flow_of_events consists of the sub-element Event, which specifies a particular event being preceded in the use case;

(10) Exceptional events – the element describes the events that do not always occur when the use case is executed;

Use_Case *UC1*
Title *Create an online order*
Description The web application is able to create an order online. The order information i.e. date, time, customer profile, product id., product name, amount of ordering are recorded into database server.
Level User Goal
Preconditions The user has already login as a customer.
Postconditions The order has recorded into the database server.
Primary_actor The user
Secondary_actors -
Flow_of_events
Event 1 The system shows a catalog of products.
Event 2 The user inputs a product id. and amount of ordered products.
Event 3 The system displays a message for next order.
Event 4 If the user sends a message to continue ordering,
Event 4a The system goes to Event 1, otherwise
Event 4b The system displays total amount of ordered products.
Event 4 The user enters the message and confirms ordering.
Event 5 The system records the order and displays an acknowledge on the screen.
Exceptional_events -
Superordinate_use_case –
Subordinate_use_case –

Figure 2: Template of Use Case.

(11) Superordinate use case – the element specifies a use case for which the use case is elaborated;

(12) Subordinate use cases – the element specifies a use case to which the use case is specified.

Figure **2** illustrates a use case *Create an online order*. The use case is identified with UseCaseID ("UC1"). The use case contains information, that is, Title, Description, Level, Preconditions, Postconditions, Primary_actor, Secondary_actors, Flow_of_events, Exceptional_events, Superordinate_use_case, and Subordinate_use_case that describe the context of the use case.

Additionally, the artefacts created under PSP-based development, that is, functional requirements, use case and class diagram, are then represented in terms of XML formats. We also added the textual parts of the artefacts with part-of-speech (POS) tags, which denote grammatical roles are annotated in terms of XML elements. As shown in Figure **3**, an example of some part of the XML-based POS-embedded use case is given. The words in the textual parts of use case are annotated with XML POS-tags denoting their grammatical roles.

4. TEST CASES

In order to take into consideration the PSP-based software developing with traceability, we have created a scenario in our testing: *changes to functional requirements*. In particular, a participant who takes the role of a software developer was asked to perform software development activities under PSP. In particular, the participant was asked to follow the development script, as shown in Table **1**.

We asked participants to perform the software development twice: (i) by applying the tool to enable the traceability; and (ii) by manually performing the traceability log. The scenario involved many types of documents, so the traceability relations were expected to be captured among various types of documents.

For the first development, the participant applied the tool to enable traceability. During the development

```
<Use_Case UseCaseID="UC1">
 <Title> Create an online order </Title>
 <Description> <AT0> The </AT0> <NN1> web </NN1> <NN1> application </NN1>
<VBZ> is </VBZ> <AJ0> able </AJ0> <TO0> to </TO0> <VVI> create </VVI> <AT0> an
</AT0> <NN1> order </NN1> <NN1> online </NN1> <SC> . </SC> <AT> The </AT>
<NN1> order </NN1> <NN0> information </NN0> <SC> i.e. </SC> <II> date </II> <SC> ,
</SC> <II> time <VVI> <SC> , </SC> <NN1> customer </NN1><NN1> profile </NN1>
<SC> , </SC> <NN1> product </NN1> <II> id <VVI> <SC> . </SC> <SC> , </SC> <NN1>
product </NN1> <NN1> name </NN1> <SC> , </SC> …
  </Description>
  <Level> User Goal </Level>
  <Preconditions> <AT> The </AT> <NN1> user </NN1> <VHZ> has </VHZ> <VHZ>
already </VHZ> <VVN> login </VVN> <IO> as </IO> <AT1> a </AT1> <NN1> customer
</NN1> <SC> . </SC>
  </Preconditions>
  <Postconditions> <AT> The </AT> <NN1> order </NN1> <VHZ> has </VHZ> <VVN>
recorded </VVN> <IO> into </IO> <AT> the </AT> <NN1> database </NN1> <NN1> server
</NN1> <SC> . </SC>
 </Postconditions>
  <Primary_actor> The user </Primary_actor>
  <Secondary_actors/>
  <Flow_of_events>
    <Event> <No> 1 </No> <AT> The </AT> <NN1> system </NN1> <VVI> shows
</VVI><AT1> a </AT1> <NN1> catalog </NN1> <IF> of </IF> <NN1> products </NN1>
<SC> . </SC> </Event>
    <Event> <No> 2 </No> <AT> The </AT> <NN1> user </NN1> <VVI> inputs
</VVI><AT1> a </AT1> NN1> product </NN1> <II> id <VVI> <SC> . </SC> …
    <Event> <No> 3 </No> <AT> The </AT> <NN1> system </NN1> <VVI> displays
</VVI> <AT> a </AT> <NN1> message </NN1> …
    </Event>      …
```

Figure 3: Template of XML-based Use Case.

script, we also asked the participant to perform the following tasks:

(i) Apply the tool to generate traceability relations after finishing *Requirements and Planning* activities;

(ii) Apply the tool to generate traceability relations after finishing *High-level Design Review (HLDR)* activities.

These tasks aimed to generate traceability relations between:

(a) functional requirements;

(b) functional requirements and class diagram;

(c) functional requirements and use cases.

For the second development, the participant manually captured the traceability relations. During the development script, the participant recorded the traceability relations in the log as their own development style. The results of manually traceability capturing were recorded in the log.

Our testing scenarios are based on a single software developer. In particular, there are changes on system or user requirements. As shown in Figure **4**, the stakeholders involved in this case are likely:

(a) software analysts (or imitators) that specify changes to be done in a design part of a software product;

(b) software analysts and software developers (or imitators) who identify the effects of these changes in the other related design software artefacts.

5. EVALUATION RESULTS AND ANALYSIS

For a test, we provides the participants with software requirements and some other software

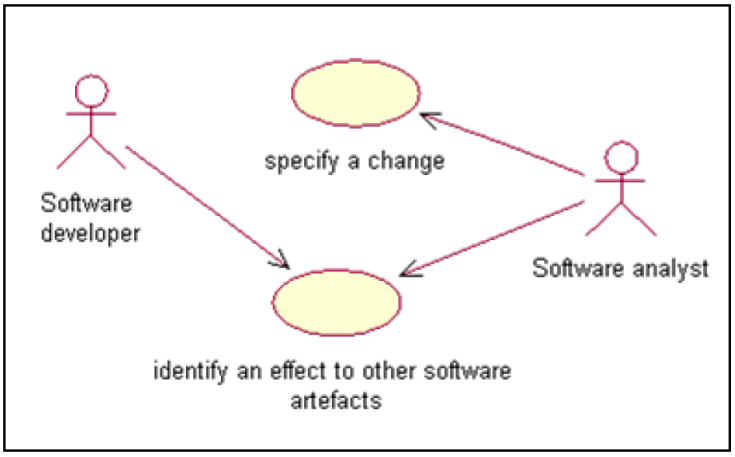

Figure 4: Testing Scenarios.

Table 2: Summary of Requirements and other Artefacts

	Number of artefacts
Number of *expected* requirements artefacts to be created or changed	2
Number of *actual* requirements artefacts created or changed	3
Number of *expected* design artefacts to be created or changed	4
Number of *actual* design artefacts to be created or changed	4
Total number of *expected* artefacts to be created or changed	6
Total number of *actual* artefacts that are created or changed	7

artefacts such as vision documents, and executive summary of the software project.

We manually counted the number of software artefacts created in the tests. As shown in Table **2**, the number of requirements artefacts created by applying the tool is 2 and the number of requirements artefacts are manually created is 3. The figures in the table show the difference in the numbers of software artefacts that are created in the same task, and having the same software requirements. Moreover, Table **3** gives a summary of the number of traceability relations identified in the tests. In the table, ST is the set of traceability relations automatically created by the tool; and UT is the set of traceability relations manually created by the software developer.

Table 3: Summary of Traceability Relations Created Manually and Automatically

	UT	ST
Number of requirements artefacts identified	16	17
Number of design artefacts identified	15	12
Total number of artefacts identified	31	29

Additionally, Table **4** shows a summary of the number of artefacts created or changed in the tests. In the table, ST is the set of artefacts expected to be created or changed, and UT is the set of artefacts that are created or changed.

Table 4: Summary of Artefacts Involved in the Tests

	Test 3
$UT_{group\ 1}$	31
$ST_{group\ 1}$	29
$\lvert ST_{group\ 1} \cap UT_{group\ 1} \rvert$	26

Table **5** shows the results for each test in terms of recall and precision rates, and provide positive evidence about our approach to apply the automatic traceability to PSP-based software development at a high level of recall and precision. In particular, the precision figure is 0.90, and the recall figure is 0.84.

Additionally, the time spent during the generation of traceability relations during PSP-based software development varies, depending on the size of the artefacts and the number of requirements and design

artefacts. We spent 6 hours to identify all traceability relations manually, and 2 hours by automatically.

Table 5: Precision and Recall Rates

	Test
Precision	0.90
Recall	0.84

6. CONCLUSION

A Personal Software Process (PSP) is a personal improvement process which drives a software developer to improve their own work. It provides a framework for writing the first program and for gathering data on work. Several forms of documents are involved. Recording traceability information becomes a challenge. Otherwise, the traceability activity for personal working is often ignored due to its difficulties. This research has shown that some degree of systematic process in creating traceability relations is facilitated by the tool.

The results of creation are measured by using precision and recall rates. The precision is measured as 90.0% and recall is measured as 84.0%. The results shown provide a positive outcome to the approach. The author has also discussed the experience with participants, who agreed that having a traceability log during PSP-based software development in an automatic way to allow them to work more effectively. Considering the developing time factor, the traceability relations assist them to verify and validate the requirements. Moreover, having completed and corrected traceability relations helped them to manage the tasks more easily.

However, we also found that data in software development grow continuously with the number of documents. It is important to specify the documents clearly and validly. The difficulty in documenting management leads the following issues, that is, missing semantics, failure to interpret semantics, missing of relevant documents, and failure to search documents.

ACKNOWLEDGEMENT

The author wishes to thank Chia-Lin Chang and Michael McAleer for helpful comments and suggestions.

REFERENCE

Antoniol, G., G. Canfora, G. Casazza, A. Lucia, E. Merlo. 2003. "Recovering Traceability Links between Code and Documentation." *IEEE Transactions on Software Engineering* 28(10):970-983. https://doi.org/10.1109/TSE.2002.1041053

Egyed, A. 2003. "A Scenario-Driven Approach to Trace Dependency Analysis." *IEEE Transactions on Software Engineering* 29(2): 116-132. https://doi.org/10.1109/TSE.2003.1178051

Humphrey, W.S. 2005. *PSP: A Self-Improvement Process for Software Engineers.* Addison Wesley. ISBN: 0-321-30549-3.

Ingram, C. and S. Riddle. 2012. "Cost-Benefits of Traceability." Pp. 23-43. in *Software and Systems Traceability*. edited by Huang, J., O. Gotel, and A. Zisman. Springer. ISBN: 978-1-4471-2239-5. https://doi.org/10.1007/978-1-4471-2239-5_2

Jirapanthong, W. and A. Zisman. 2009. "XTraQue: traceability for product line systems." *Software System Model* 8(1):117-144. https://doi.org/10.1007/s10270-007-0066-8

Kim, D., S. Chang, and H. La. 2005. "Traceability Map: Foundations to Automate for Product Line Engineering." *In Proceeding of the 3rd ACIS International Conference on Software Engineering Research, Management and Applications (SERA'05)*, Pp. 274-281.

Marcus, A., J. I. Maletic. 2003. "Recovering Documentation-to-Source-Code Traceability Links using Latent Semantic Indexing." *In Proceeding of the 25th International Conference on Software Engineering (ICSE)*, Oregon, USA, May 03-10, Pp. 125-135. https://doi.org/10.1109/icse.2003.1201194

Ramesh, B. and M. Jarke. 2001. "Towards Reference Models for Requirements Traceability." *IEEE Transactions on Software Engineering* 27(1):58-93. https://doi.org/10.1109/32.895989

Received on 16-02-2017 Accepted on 13-05-2017 Published on 09-06-2017

DOI: https://doi.org/10.6000/1929-7092.2017.06.38