Software Artefacts as Equipment: A New Conception to Software Development using Reusable Software Artefacts

Research-in-Progress

Subasinghage Maduka Nuwangi
Information Systems School, Queensland University of Technology, Brisbane, Australia
maduka.subasinghage@qut.edu.au

Darshana Sedera
Information Systems School, Queensland University of Technology, Brisbane, Australia
d.sedera@qut.edu.au

Abstract

Through the lens of Heidegger’s analysis of equipment, this study observes ‘software reuse’ – a popular phenomenon in the world. It presents an alternative conceptual view of the software artefact as ‘equipment’. This view provides a theoretical underpinning to this prominent practice to recognize software artefacts, as equipment. Employing the case study method, this study reports preliminary results of five software development projects to investigate the development and reusability of software artefacts. Two types of generalizability were identified: 1) horizontal generalizability, and 2) vertical generalizability. From the results it can be inferred that reusability of software artefacts may depend on the type of generalizability. The level of reusability of software artefacts may increase the level of maturity of software artefacts. Furthermore, the results indicated that the software artefacts were updated rapidly in the initial stages, compared to final stages of software development lifecycle.

Keywords: Software reuse, Software development, Software artefacts, Heidegger’s analysis of equipment
Introduction

Since the use of software has become a critical success factor for a variety of industries, including manufacturing (Staehr 2010), healthcare (Maass and Eriksson 2006) and education (Van Rooij 2009), the pressure for developing new software has increased. Moreover, new technological innovations create a rapidly changing environment in most industries (Lee and Xia 2005; Sarker and Sarker 2009; Vidgen and Wang 2009). To meet these challenges, software should be easily adaptable to industrial and technological changes (Conboy and Morgan 2011; Hoda et al. 2013; Paasivaara et al. 2009). The development of new software faces several issues such as the failure to meet deadlines, budget overruns, and poor software quality (Crnkovic 2001; Hildenbrand et al. 2008; Nuwangi et al. 2013b). Thus, the use of software artefacts across multiple projects, referred to as “software reuse”, has been widely utilized as a strategy for enhancing the software quality (Frakes and Kang 2005) and software development efficiency (Ravichandran and Rothenberger 2003).

Krueger (1992, p. 131) defines software reuse as “the process of creating software systems from existing software rather than building software systems from scratch”. The reuse of software artefacts ranges from ideas, algorithms and source codes to the re-use of any documentation created during the software development life cycle (Jalender et al. 2011). Since the software component1 is the most commonly reused software artefact, many people misinterpret software reuse as the reuse of software components alone (Jalender et al. 2011). Software reuse is considered to be an important strategy for enhancing software quality (Frakes and Kang 2005), productivity (Capiluppi et al. 2011; Standish 1984) and software development efficiency (Ravichandran and Rothenberger 2003). According to Ravichandran (2003), software reuse improves the scalability, adaptability, and maintainability of the resultant software.

The approach of reusing existing software artefacts for software development parallels Heidegger's analysis of equipment which was introduced in Being and Time (1927/1962). Through the lens of Heidegger's analysis of equipment, the present study identifies the important characteristics of software artefacts that a software development team should take into account when developing software that reuses existing software artefacts. Specifically, this study presents an alternative conception of software development using reusable artefacts, where the software artefacts are considered as equipment.

This study highlights that software artefacts are created and stored for reuse and are adopted by the software development team to develop new software, in a way that is similar to using equipment. The software artefacts should possess the right properties for the software development team to perform their tasks (Nuwangi et al. 2013a). Similar to equipment, some software artefacts such as software components may have issues such as temporary breakdowns and malfunctions. Software artefacts are not considered to be a software until the software development team assembles the artefacts in a meaningful way. When assembling software artefacts to develop new software for a client, the software development team must ensure that the software artefacts are assembled in a way that satisfies the client’s requirements. The inaccurate assembly of software artefacts can lead to software project failures. As equipment, a software artefact should be defined by its place in the software development practice, considering its relationship with other equipment (e.g. software platform), typical activities (e.g. software development) and purpose for which it is used (e.g. to develop sales management software). Reusable artefacts are generally created in such a way that they can be used by different applications. Therefore, software artefacts are different when embedded in different contexts. For example, a finance management component can be used for developing sales management software and hotel management software. The functionalities of the finance management component in the sales management software and hotel management software would be different.

Prior research (Basha and Moiz 2012; Ravichandran and Rothenberger 2003) has discussed software reuse; however, to the best of the author's knowledge, no study has discussed the important

---

1 Software components (also referred to as code-level components) are one type of software artefact, and are described as self-contained software packages that carry out certain business tasks. (Vitharana, P., and Jain, H. 2000. "Research Issues in Testing Business Components," Information & Management (37:6), pp. 297-309.)
characteristics of software artefacts which a software development team should consider when developing a new software for a client by using existing software artefacts. Thus, the present study addresses the question of: “What are the important characteristics of software artefacts, which a software development team should consider when developing software by reusing existing software artefacts?” This research-in-progress paper reports early observations on the important considerations when developing new software by utilizing reusable software artefacts. The analysis and recommendations presented here are preliminary; however, they provide insight into the landscape of software development using reusable software artefacts.

Theoretical Background: Heidegger’s analysis of equipment

Heidegger’s analysis of equipment has been used in the literature to discuss a variety of aspects including affordances (Turner 2005). IT artefacts (Riemer and Johnston 2014) and equipment use in sports (Breivik 2007). Heidegger’s philosophy focuses on the nature of being, meaning the ways that entities can be in the world. The human way of being (in Heidegger’s terms “Dasein”) is engagement in practices (Existenz). One way of being of humans is when human beings use things as equipment (Zeug). In-depth investigation of Heidegger’s concepts of being and time is outside the scope of this paper. Rather, this paper focuses on Heidegger’s analysis of equipment that is relevant to understand the important characteristics of software artefacts, which a software development team should consider when reusing existing software artefacts. Heidegger’s analysis of equipment provides a different perspective to the phenomenon being studied. Since the software artefacts can be considered as equipment used for the software development process, application of the Heidegger’s analysis of equipment concepts provide the ability to investigate the research question.

Human beings use equipment for different purposes (e.g. writing, working and transportation). According to Heidegger (1927/1962, p.97), “to the Being of any equipment there always belongs a totality of equipment, in which it can be this equipment that it is. Equipment is essentially something in-order-to”. Moreover, Heidegger (1927/1962, p. 98) discussed: “the kind of Being which equipment possesses – in which it manifests itself in its own right – we call ‘readiness-to-hand’”. When entities are ready-to-hand, the user does not notice the theoretical aspects of the entities. According to Breivik (2007), the ‘presence-at-hand’ (Vorhandenheit) condition is a derived condition, whereby an entity is isolated from its context and becomes an object to look at and describe in an isolated manner free of context.

Software artefacts such as software components and documents can be reused to develop software for different clients (Olsson et al. 2008). Thus, the same equipment structure that Heidegger depicts is visible in software development projects. Software artefacts refer to each other and to the task which need to be done. According to Heidegger, there is use of something ‘in-order-to’ do something. Similarly, software artefacts are used ‘in-order-to’ develop software. As per Heidegger, the equipment is laden with context-dependence, whereas the object is context-independent. When an entity is context independent, it becomes more general. Similarly, when the software artefacts are context independent, it becomes generalizable. According to Heidegger, equipment mature with reuse (Blattner 1995). Likewise, when a software development team reuse software artefacts, the software artefacts become matured. Table 1 summarizes the applicability of Heidegger’s analysis of equipment concepts in the context of software artefact reuse.

<table>
<thead>
<tr>
<th>Table 1: Heidegger’s Analysis of Equipment in the Context of Software Artefact Reuse</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Equipment Aspect</strong></td>
</tr>
<tr>
<td>The in-order-to of equipment</td>
</tr>
<tr>
<td>Readiness-to-hand</td>
</tr>
<tr>
<td>Presence-at-hand</td>
</tr>
<tr>
<td>Holistic nature of equipment</td>
</tr>
</tbody>
</table>
Generalizability | When the software artefacts are context independent, it becomes generalizable (i.e. the software artefacts are developed with the objective of providing the flexibility to easily adapt to develop new software requirements (Baum and Becker 2000)).
---|---
Reusability | Similar to equipment, software artefacts are reusable (i.e. the ability to use a previously developed software part, such as a data structure or logic functions, rather than developing a new one when building a software application or system (McClure 2001, p. XIII)).
Maturity | Software artefacts mature (i.e. the extent to which the software artefacts consist of the required features for the software development process) with reuse.

**Theoretical Propositions**

As per the Heidegger’s Analysis of Equipment, when an entity becomes isolated from its context and becomes something to look at and describe in an isolated manner that is free of context, the entity can be considered as an object (Riemer and Johnston 2014). An object becomes equipment when it is fully adopted and used within the practice. Thus, the equipment is laden with context-dependence (Hajri et al. 2014), whereas the object is context-independent. The same entity can become different equipment when embedded in different practice holisms (Heidegger 1927/1962). When the entity is context independent, it becomes more general. Similarly, software artefacts should be sufficiently general to allow the reuse in future projects (Sametinger 1997; Wu and Cao 2009). Many software reuse technologies merge similar artefacts into a single generalized artefact (Krueger 1992). According to Dusink and Latour (1996, p. 137), “a common rule of thumb in making components reusable is to ‘make components generic’”. Making generic components is difficult as: 1) software engineers tend to solve problems in a way which is very specific to the problem at hand, so the solutions are difficult to generalize to other problems (Dusink and Latour 1996; Salim et al. 2014), and 2) opportunities to reuse existing components are not easily identified (Dusink and Latour 1996). Some form of generalizability is essential for software artefacts reuse (Baum and Becker 2000). Matook and Indulska (2009) discussed that lack of generalizability minimizes the reusability. According to Sadaoui et al. (2004), “one method for enhancing the reusability of existing components is generalization”. Thus, the present study proposes:

**P1: The higher the level of generalizability of software artefacts, the higher the level of reusability**

Using the hammer as an example, Heidegger explains that if the hammer is too heavy for the task or is breaking, the builders can learn about metals and about setting metals into hammers so that they can make a better hammer (Blattner 1995). This highlights that the equipment mature with reuse. Preston (1998, p. 517) discusses the natural occurrence of tools with an example of using a stone as a tool: “In a sense it is the act of use that constitutes the stone as a tool; and if it is immediately discarded it might be considered to lapse back into a state of nature”. In contrast, if the person continues using a stone as a tool, the stone acquires special characteristics, which make it recognizable as a tool apart from the other stones. These discussions highlight that the equipment matures with ‘reuse’. Similarly, software reuse repositories have to be modified (Henninger 1997) according to the evolving and dynamic needs of software development organizations (Nuwangi et al. 2014). Sametinger (1997) proposed the “reuse spiral” whereby the systematic reuse of software components is incorporated into the software development lifecycle. The reuse spiral comprises of four stages: 1) the understanding, retrieval and reconfiguration of software components, 2) assessment and evaluation, 3) modification, adaptation and integration, and 4) evaluation and consolidation. The software components are adapted and modified in the third stage, and the fourth stage includes evaluating (Salim 2013) the reusability of the modified components for their potential contribution to the component repository. If there is a separate component development team, new knowledge should be transferred (Alarifi and Seder 2013, 2014; Nuwangi et al. 2012) to the component development team for the purpose of improving the existing components. According to Fayad et al. (1997, p. 86), “without intimate feedback from application developers, the software artefacts produced by a component team won’t solve real problems and will not be widely reused”. Therefore, it is necessary to provide pre-existing software artefacts to developers and update the software artefacts accordingly. When the software artefacts include high level of reusability, the software team members tend to reuse the software artefacts and provide feedback. Since the software artefacts are modified according to the feedback from the software development team, the software artefacts become matured. Thus, the present research proposes:
P2: The higher the level of reusability, the higher the level of maturity of software artefacts

Research Methodology

In order to evaluate the theoretical propositions in this study, a multiple case study approach was chosen. The qualitative case study approach is recognized to be appropriate for investigating complex environments (Klein and Myers, 1999) and contemporary events (Benbasat et al., 1987). The case study follows a deduction logic through which it is intended to evaluate the pre-identified propositions (Chatterjee et al. 2009; Chatterjee et al. 2013; Lee 1989). According to Chatterjee et al. (2009, p. 625), “a deductive approach involves starting with an already formulated theoretical proposition and using empirical evidence to assess the validity of the proposition”. Sarker et al. (2013) highlight the importance of selecting an appropriate sampling strategy in case study research. This research utilized the purposeful sampling strategy to reach the research objectives (Patton 2002). Three conditions formed the benchmarking criteria for the selection of the software development organization as the case study. First, the organization should create and reuse software artefacts. Second, the organization should be involved in multiple software development projects. Third, the organization must be sufficiently large, with a standard hierarchy of employment. Following the application of these criteria, one company was selected from the several companies contacted. Selected company was an ideal case: five software development projects within the company were targeted for the data collection purposes. Selection of software projects followed opportunistic/emergent sampling strategy. According to Patton (2002), opportunistic/emergent sampling follows new leads during fieldwork and take the advantage of unexpected flexibility.

Company specializes in developing software solutions for capital markets, with more than 30 capital market clients all over the world. Those software solutions include: 1) exchange solutions, 2) surveillance solutions, 3) Smart Order Routing (SOR) solutions, 4) post-trade solutions, and 5) market data solutions. Rather than developing software solutions for each client from scratch, the company reuses existing software products. Company has five types of products: 1) exchange products, 2) surveillance products, 3) SOR products, 4) post-trade products, and 5) market data products. Those products include the main and basic software artefacts of each type of solution. The solutions are developed according to the client’s requirements by reusing the software artefacts of the company’s products. Separate software development teams are involved in the company’s solution projects and product projects.

Five (5) software development projects within the company were selected as the cases in order to ensure control and replicability. Three solution projects (A, B, and C) and two product projects (D and E) were selected. According to Eisenhardt (1989, p. 545), “there is no ideal number of cases, a number between 4 and 10 cases usually works well”. Fifteen (15) semi-structured interviews (each of 20–30 minutes duration) were conducted with the employees from five software development projects. In order to avoid key informant bias, multiple informants from each project were interviewed (Kumar et al. 1993). The sampling technique was non-probable, purposive and employed the snowball technique, whereby the informants are selected as appropriate opinion leaders with well-developed views on the research topic (Minichiello et al. 1995). At the end of the interview, the informants were asked to suggest other employees who are knowledgeable about usable software artefacts. The informants were diverse as employees were added to the sample according to the recommendations from the previous informants (Ramaswami 1996). While conducting the interviews, additional notes were taken whenever necessary. The interview data was supplemented with documents such as test scenario and test case specifications. Those documents increased the validity and reliability of the collected data. All the interviews were audio-recorded and transcribed for subsequent data analysis purposes. The unit of analysis was the software development project. NVivo software was used for data analysis. The data analysis was conducted in two phases: 1) within-case analysis, and 2) cross-case analysis. Within-case analysis was performed to test whether or not the initial propositions were supported. This was followed by the cross-case analysis to investigate the similarities and differences between the cases. Table 2 includes the sample codes used for the data analysis. The sample codes were generated following the Heidegger’s Analysis of Equipment concepts.

<table>
<thead>
<tr>
<th>Term</th>
<th>Sample codes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>High</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Low</strong></td>
</tr>
<tr>
<td>Generalizability</td>
<td>Software artefacts are context independent</td>
</tr>
<tr>
<td></td>
<td>Software artefacts have generic characteristics</td>
</tr>
<tr>
<td></td>
<td>Software artefacts are context dependent</td>
</tr>
<tr>
<td></td>
<td>Software artefacts have specific characteristics</td>
</tr>
</tbody>
</table>
Table 2: Sample Codes

<table>
<thead>
<tr>
<th>Term</th>
<th>High</th>
<th>Low</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><strong>Sample codes</strong></td>
<td><strong>Sample codes</strong></td>
</tr>
<tr>
<td></td>
<td>Generic features</td>
<td>Specific features</td>
</tr>
<tr>
<td></td>
<td>Features can be easily customized</td>
<td>Features cannot be easily customized</td>
</tr>
<tr>
<td></td>
<td>Less changes</td>
<td>Require radical changes</td>
</tr>
<tr>
<td></td>
<td>Generalize the requirements</td>
<td>Do not generalize the requirements</td>
</tr>
<tr>
<td>Reusability</td>
<td>Reuse</td>
<td>Unable to reuse</td>
</tr>
<tr>
<td></td>
<td>Easy to reuse</td>
<td>Difficult to reuse</td>
</tr>
<tr>
<td></td>
<td>Use more</td>
<td>Reuse in some modules</td>
</tr>
<tr>
<td></td>
<td>Use product artifacts</td>
<td>Use less</td>
</tr>
<tr>
<td></td>
<td>Document the gap between client requirement</td>
<td>Do not use product artifacts</td>
</tr>
<tr>
<td></td>
<td>and the base product</td>
<td></td>
</tr>
<tr>
<td>Maturity</td>
<td>Do not implement new requirements in the</td>
<td>Implement new requirements in the product</td>
</tr>
<tr>
<td></td>
<td>product</td>
<td>Update the product</td>
</tr>
<tr>
<td></td>
<td>Do not update the product</td>
<td>Unstable</td>
</tr>
<tr>
<td></td>
<td>Stable</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Mature work</td>
<td></td>
</tr>
</tbody>
</table>

**Results**

P1: The higher the level of generalizability of software artefacts, the higher the level of reusability

The solution teams in the company reused the software artefacts developed by the product teams. According to R\textsuperscript{2}11 (Product D team), a product consists of several software components: “product actually it is like component structure. The full functionality is broken down to components”. In Solution C, the software quality assurance team reused existing test scenario and test case documents for the purpose of testing some functionality. For other functionalities, the quality assurance team developed new test scenario and test case documents. R\textsuperscript{0}9 (Solution C team) stated: “There were [test] scenario existing and test cases like derivative functionality. So, we can reuse test case document and test cases. But there was no reuse for other scenarios”. R\textsuperscript{0}8 (Solution C team) explained: “They implement some new test scenario document stuff. That case we didn’t reuse much. Some modules, we did reusing”. According to R\textsuperscript{0}4 (Solution B team), not all the software artefacts which are included in the product are reusable. Since the product has its own mechanisms for trading, the solution teams are unable to reuse some software artefacts. R\textsuperscript{0}4 stated: “Not all product features can they make use of in their exchange. Each exchange has their own trading way”. This indicates that when the software artefacts are generalized, those artefacts can be easily reused by the software teams. According to R\textsuperscript{0}4 (Solution B team), when a client requires some product artefacts to be customized according to the client’s special requirements, the team members have several discussions with the client. R\textsuperscript{0}4 stated: “Since we are a product based company this is the latest release of the product. When they wanted some of the [product] features customized for their [client’s] own particular specific requirements, we had many client meetings”. This highlights that when the product artefacts are generalizable, the solution teams will be able to reuse and easily update artefacts according to the client’s requirements. Software architecture for the solution teams is provided by the product teams. When a client’s requirement introduces a radical change to existing software architecture, the solution teams had to find alternative methods to provide the client’s requirement. As R\textsuperscript{0}1 (Solution A) mentioned, “if there is a new CR [change request] try to introduce a radical change to existing architecture, normally we are going to give alternative ways that can handle the new requirements”. This highlights the importance of generalizability of software architecture.

The product specifications were reused by the Solution B team. The solution team only documented the differences between the client requirements and the product specifications. R\textsuperscript{0}5 (Solution B team) explained: “we document the gap between the client requirement and the based product”. This indicates that, when product specifications are generalized, many projects can reuse the product specifications. According to R\textsuperscript{1}4 (Product E team), when the product teams receive the requirements from the solution teams, only the generalizable requirements are included in the product: “Depending on the situation we

\textsuperscript{2} R=Respondent
generalize the requirement as everybody can use it and we implement it in the product”. This indicates that the generalizability of the software artefact increases reusability. In some situations, the product had to be specialized according to the solution requirements. R15 (Product E team) stated: “Whenever there is the CR [change request] getting that to the solution team and if they think that the things should be fixed in the product level. Then we take them [requirements] and implement them in the product”. According to R10 (Product D team), “they [the solution team] internally decide whether this can be done in their layer or can that pass to the product layer”.

Through the cross-case analysis (see table 3), the study observed that when generalizability is high, it may increase the reusability of software artefacts as stated in proposition1. For example, since product D consisted of generalizable software artefacts, several solution teams were able to reuse product D artefacts. Cross-case analysis illustrated two types of generalizability, which this research refers to as ‘horizontal generalizability’ and ‘vertical generalizability’. Horizontal generalizability refers to situations, where the solution teams can reuse software artefacts of product ‘as is’ without modification. Although solution teams are not allowed to modify product artefacts, in required situations, the product team can update the product artefacts according to the solution team’s requests. This situation is named as ‘horizontal generalizability’ due to the fact that the product artefacts remain as the base. The solution teams are not allowed to update the base product. Vertical generalizability refers to situations, where solution teams are able to adopt and update the product artefacts according to solution requirements. This situation is named as ‘vertical generalizability’ due to the fact that the product artefacts can be updated by the solution teams. Vertical generalizability was observed in Solution B, where solution team utilized product specifications without modifications. Product D and E consisted of horizontal generalizability, where the products were updated by the product teams depending on the solution requirements. Vertical generalizability was observed in Solution B, where the product artefacts were adopted and updated by the solution teams. Solution A indicated vertical generalizability, where the software architecture was adopted and updated by the solution teams. From the results it can be inferred that reusability of software artefacts may depend on the type of generalizability.

<table>
<thead>
<tr>
<th>Project</th>
<th>Summary of the Cross-case Analysis – P1</th>
</tr>
</thead>
<tbody>
<tr>
<td>A [Solution]</td>
<td>• Software architecture should be generalizable. Solution team adopted the software architecture and updated it according to solution requirements [vertical generalizability]</td>
</tr>
<tr>
<td>B [Solution]</td>
<td>• CR documents for highlighting the differences between solution requirements and the product. Solution team did not update the product specifications [horizontal generalizability]</td>
</tr>
<tr>
<td>C [Solution]</td>
<td>• Reuse test scenario and test case documents when they include relevant features</td>
</tr>
<tr>
<td>D [product]</td>
<td>• Since product D was generalizable, it was reusable by many solutions.</td>
</tr>
<tr>
<td>E [product]</td>
<td>• When a product team received a requirement, the product team generalized the requirement and implemented it in the product</td>
</tr>
<tr>
<td></td>
<td>• When solution team thinks some functionality should be included in the product, solution team informed the product team. Product team updated the product by including new functionality [horizontal generalizability]</td>
</tr>
</tbody>
</table>

**P2: The higher the level of reusability, the higher the level of maturity of software artefacts**

When a solution team identified some software functionality which should be included in the product, the solution team informed the product team about the requirements. According to R10 (Product D team), “based on the requirement gathering with client they [the solution team] have log to the CR, the change request. So they [the solution team] internally decide whether this can be done in their layer or can that pass to the product layer”. R14 (Product E team) discussed the process of including a new requirement in the product. When a new requirement was received, the product team identified whether or not the new requirement could be generalized. When the requirement could be generalized, it was included in the product for future use. When the requirement could not be generalized, it was developed for a specific client. According to R14, “when requirement is good and forwarded, [...] we document the requirement and then it is discussed internally with the product team whether we implement it in the product or whether we implement this for the client and so then depending on the situation we generalize the
requirement as everybody can use it and we implement it in the product”. When the solution team suggested an update to the product, the product team checked the backward compatibility. In situations where the product update gave rise to compatibility issues, the product team identified alternative methods to meet the solution team’s requirements. R15 (Product E team) stated: “They [the solution team] know including the field may have an impact on the backward compatibility, what we [the product team] do it another way, use existing fees add some values, change value something like that”. According to R15, adding a new functionality to the product always create issues: “I can't compare it [the product] with project [the solution]. It [the product] is sort of stable. You can't say there are not any issues, but with each and every new thing that getting added always has issues”. R10 (Product D team) mentioned that the product includes mature functionalities (“product is actually mature work”). This indicates that the process of updating the product was a slow process compared to the solution.

Component Design Specification (CDS) was used by the company to describe the initial and basic functionalities of the products. When a solution team in the company requires some change or update to the product, the solution team creates a CR document. According to R11 (Product D team), in order to obtain a complete understanding of the product features, the team members should read both the CDS and CR documents: “CDS document has initial design not the changes. So, sometime it is difficult to track the new change, because we are not going to read all the CR documents. So from initial document we can’t gather full knowledge”. This indicates that the product is updated according to the solution team’s suggestions. According to R10 (Product D team), when the product is updated, the software artefacts related to the product (e.g. documents which describe the product features) are also updated accordingly: “Whenever we send this [product update] out the entire document: technical, operating and specification documents will be updated, because those are part of the delivery, not only the code of the system”. Moreover, R13 (Product E team) stated: “When a developer takes the responsibility of one component he has the responsibilities of changing the product document”. According to R03 (Solution A team), the solutions should be updated according to the updates of the product. Otherwise, mismatches can occur between the product and solution connections: “Assume they add new field to the product. Then, that field should be added to the solution level also”.

According to the cross-case analysis results (see table 4), increased level of reusability of software artefacts may increase the maturity of software artefacts as stated in proposition 2. When the software artefacts were reusable, several solution teams utilized the artefacts and provided feedback to product teams. Subsequently, software artefacts of the product were updated. Product D findings highlight that although solutions reused product artefacts, the process of updating the software artefacts of the product was a slow process. This was due to the reason that the product artefacts were matured. Findings of product D and E indicate that compared to final stages, the software artefacts of the product were updated rapidly in the initial stages as the reusability of the product artefacts were higher in the initial stages of solution projects.

<table>
<thead>
<tr>
<th>Project</th>
<th>Summary of the Findings</th>
</tr>
</thead>
<tbody>
<tr>
<td>A [solution]</td>
<td>• Although maturity of product may occur due to the requirements of a single solution team, the maturity of product affects all the connected solutions.</td>
</tr>
<tr>
<td>B [solution]</td>
<td>• When software artefacts are reusable, team members utilized those artefacts more. This lead to maturity of software artefacts.</td>
</tr>
<tr>
<td>C [solution]</td>
<td>• When software artefacts are lack of reusability, team members were unable to reuse software artefacts much. As a result, maturity of software artefacts was low.</td>
</tr>
</tbody>
</table>
| D [product] | • Functionalities which are required to be included in product were identified at the requirement gathering stage. This indicates that reusability and modifications of the software artifacts were higher in the initial stages of solution projects.  
  • The product includes mature software functionalities. Thus, the process of updating the software artefacts of the product was a slow process.  
  • When the product was updated, all the related documents were also updated accordingly |
| E [product] | • When a product team received an update request, the product team generalized the requirement and implemented the requirements in the product.  
  • Product team received many update requests from solution teams at the initial stages.  
  • When a software component is reused and updated, related documents were also updated. |
Discussion

This study was motivated by the need to understand the important characteristics of software artefacts, which a software development team should consider when developing software by reusing existing software artefacts. The application of Heidegger’s analysis of equipment in the context of software artefact reuse provided the ability to gain an understanding about these important considerations. Heidegger conceptualized that equipment is laden with context-dependence, whereas the object is context-independent. The same object can become different equipment when embedded in different practice holism. According to the results of this study, the generalizability of the software artefacts may increase the reusability. Two types of generalizability were identified: 1) horizontal generalizability - situations where the solution teams can reuse software artefacts of product ‘as is’ without modifications. In the required situations, only the product team members are allowed to update the product artefacts; and 2) vertical generalizability - situations where the solution teams are able to adopt and update the product artefacts according to the solution requirements. From the results it can be inferred that reusability of software artefacts may depend on the type of generalizability. Furthermore, the results highlighted that reuse of software artefacts in software development projects leads to the maturity of software artefacts. The software artefacts developed by the product team in the case study were reused by the solution teams to develop software according to client requirements. In some situations, the client’s requirements had to be included in the product. When a product team received an update request from the solution team, the requirements which could be generalized for other projects were included in the product. Since updating a product artefact can create issues, the product team had to check the backward compatibility of the new requirements. When a product team updated the software components of the product, it was necessary to update the product specifications as well. This highlights that as a result of reuse, software artefacts of the product such as software components and documents mature with the time. Since the product was completed and stable, the process of updating the software artefacts of the product was a slow process, compared to the solutions. Compared to final stages, the software artefacts of the product were updated rapidly in the initial stages of solution projects.

This research discussed the importance of the generalizability and maturity of reusable software artefacts. Since this research approached the software artefacts as equipment, the findings of the study highlight the importance of further exploring the concept of the reusability, generalizability and maturity of equipment. The preliminary study findings have the potential to influence practice. First, the findings help software development organizations to understand the important considerations in software development that utilizes reusable artefacts. Second, it highlights the importance of taking a holistic view of software artefacts. Third, the application of Heidegger’s analysis of equipment concepts in the field of software artefact reuse provides useful guidelines for software development, which could ultimately increase software development efficiency.

Several studies in the literature discuss Heidegger’s analysis of equipment in different contexts (Riemer and Johnston 2014; Turner 2005); however, to the best of the author’s knowledge, no studies have yet applied Heidegger’s analysis of equipment in the context of software development using reusable artefacts. Heidegger’s analysis of equipment offered a new perspective to the phenomenon being studied. Furthermore, application of the Heidegger’s analysis of equipment provided the ability to understand the important characteristics of software artefacts, which a software development team should consider when developing software by reusing existing software artefacts. The preliminary results of the present study are encouraging, and further work is underway to evaluate the identified propositions in more detail and establish the research objectives. Specifically, we are planning to investigate more on: (1) how the level of reusability changes according to the type of generalizability; and (2) how the level of reusability and maturity of software artefacts fluctuate in the different stages of software development lifecycle. Furthermore, we are planning to investigate other relevant theories such as software engineering theory, modular systems theory and enterprise systems fit theory in order to provide a broader perspective to the phenomenon being studied.
Software Artefacts as Equipment

References


