Investigating the Effects of Framework Design Knowledge in Example-based Framework Learning

Daqing Hou
Electrical and Computer Engineering, Clarkson University, Potsdam, New York 13699
dhou@clarkson.edu

Abstract

Studying example applications is a common approach to learning software frameworks. However, to be truly effective in adapting example solutions with high confidence and accuracy, a developer needs to learn enough about the framework designs. The empirical study described in this paper investigates the effectiveness of a new approach to framework learning, where example-based learning is augmented with instruction on framework designs. Learning framework designs up-front from an instructor helps developers acquire the necessary design knowledge and avoid the time-consuming task of recovering such knowledge from code and other artifacts. The particular question of interest in this study is how characteristics of the framework designs influence project outcome. 11 student projects are analyzed using both qualitative and quantitative methods to characterize the overall reuse practice and to detect salient patterns that address the question. The contribution of this paper is a set of well-supported hypotheses that can be tested in future studies as well as their implications.

1. Introduction

The use of application frameworks may help reduce development time and cost, and improve software quality [14, 15, 16]. To an application developer, the value of a framework is in the solutions or features (both architectural and local) which it provides. Because framework designs tend to be abstract, feature-rich (for example, the 179 examples in the Swing tutorial would imply that at least 179 features of Swing are worth learning), and can be hard to discover or understand, the initial effort invested in learning a framework can be very high for novices [8, 12, 13].

Example applications have been identified as an effective learning aid for an application framework [12, 18, 21]. Example applications show what the framework is good for and point out features that the framework provides. Because examples are concrete, they are easier to learn than the abstract designs [12]. With examples available for manipulation, developers can focus more of their attention on analysis and modification rather than search and construction, which can be especially beneficial to novice learners. This benefit can be explained by the interaction design principle of 'recognition rather than recall' [20]. Examples have also been recommended as a necessary ingredient in effective teaching and learning, such as learning effective techniques for problem solving [2, 3], or learning how to write programs in a new programming language [17].

However, examples alone are not sufficient for the effective learning of application frameworks. First, although examples may contain concrete solutions that are useful to developers, they do not explicitly explain how the demonstrated solutions are provided by the framework. Second, the solutions contained in the examples may not exactly fit the needs of the new application. They may be incomplete and only partially useful, overly complicated, or completely irrelevant. Thus, developers must be able to identify and modify such example solutions in order to use them in their own applications. Third, when the number of example applications is large, it can be difficult to recognize which ones are relevant to the application at hand.

Knowing framework designs may yield both tactical and strategic benefits [7, 18]. Tactically, a developer will perform better in utilizing and adapting individual framework features with higher confidence and accuracy. More importantly, knowledge of framework designs will also enable developers to anticipate major architectural mismatch between the framework and the application to be built, when making strategic decisions like adopting a framework. Design knowledge helps developers focus on important design issues early and avoid the trap of driving application development by framework features [18]. Thus, to be truly effective in using a framework, a developer must learn enough about the framework designs. (We consider serious, long-term users of a framework. Short-term, casual users may be able to make use of a framework opportunistically without a serious investment in learning framework designs.)

One approach to learning framework designs is studying

design documentation and framework code. In a previous study, Shull et al. investigated the effectiveness of two reading techniques for framework learning (framework-design-based versus example-based). A major difficulty reported was that the design-based approach is too time-consuming for novice developers to use because "the technique [for learning framework designs] gave subjects no idea which piece of functionality provided the best starting place for implementation, or where in the massive framework hierarchy to begin looking for such functionality" [21]. They formulate the hypothesis that *a hierarchy-focused technique is not well-suited to use by beginners under a tight schedule.* But the question remains: How can novices learn the designs of a framework effectively?

The empirical study described in this paper investigates the effects of framework design knowledge on applications built by customizing examples and the implications on how to *teach* framework designs more effectively. The main research questions for this study can be phrased as follows:

> **Question 1.** *What are the characteristic effects of framework design knowledge on applications developed by customizing examples?*

> **Question 2.** *What lessons can be learned for effectively teaching frameworks?*

This study followed Eisenhardt's approach to building theories from case study research [4], which was also used in software engineering research, e.g., by Seaman et al. in [19] and by Shull et al. in [21]. Since relatively little is known about this topic, the goal of this study is to identify important variables and relationships, not to test established hypotheses. Our study was conducted in a classroom setting where novices learned to use the AWT/Swing framework [22] to build GUI applications within one semester. We analyzed both qualitative and quantitative data from the course to search for possible answers that address the research questions. Thus, the contribution of this paper is a set of hypotheses, along with supporting evidence, that can be further tested in the future.

The remainder of this paper is organized as follows. Section 2 surveys related work. Section 3 describes the setting of this study and Section 4 research methods. Section 5 presents the hypotheses derived along with the details of the quantitative and qualitative analyses performed on the collected data. Section 6 discusses the threats to validity. Finally, Section 7 concludes the paper by answering the preceding two research questions.

2. Related Work

In addition to the most closely related study conducted by Shull et al. [21], there is related work in the areas of empirical study of framework-based development, framework documentation, and tool support for framework usage.

Empirical study of framework-based software development. Based on a case study, Morisio et al. [15] report that both productivity and quality in framework-based development can be better than in traditional development, and that productivity increases massively when developers learn more about a framework over time. Mohagheghi et al. [14] report from an industrial setting that reused components have less bugs and are more stable than non-reused ones, and that bugs in reused components tend to receive high priority. Hou et al. [11] investigate the connection between the designs of a framework and questions asked about the framework, highlighting areas in the framework that are poorly designed or insufficiently documented, and recognizing inferior programmer practices.

Framework documentation. Johnson [12] points out that framework documentation serves three purposes: intent, detailed designs, and how-to instructions, and that patterns can be used to provide all three. But framework documentation does not have to cover all three purposes to be useful. For example, Froehlich et al.'s hooks [6] are a structured notation for describing how to use a framework, but not its design or intent.

Gangopadhyay and Mitra [7] recommend an approach to learning frameworks by concentrating on the framework architecture rather than individual components. In particular, they recommend the development of exemplars, executable models that visualize important framework collaborations.

Schneider and Repenning [18] also emphasize the importance of framework design knowledge and the danger of 'feature-driven development', and suggest that paradigmatic applications may be developed to facilitate the learning of framework designs.

Tool support for framework usage. Several researchers have built advanced, wizard-like tools to assist the use of frameworks. Fairbanks, Garlan, and Scherlis [5] propose design fragments and tool support for capturing, enabling, and enforcing the common patterns of using a framework. Antkiewicz and Czarnecki [1] propose to use domain-specific languages to model framework interfaces and automatically maintain the consistency between the model and application code. Hautamäki and Koskimies [8] investigate specifying and automating the application of specialization patterns.

3. Study Setting

To explore the role of framework designs in example-based framework learning, the author ran a study into framework usage as part of a software engineering course that he taught Fall 2007 at Clarkson University. There were 16 third and fourth year undergraduates (juniors and seniors) in the class. The main course objective is to learn component-based software construction in visual environments. JFC Swing [22] was used as the teach-

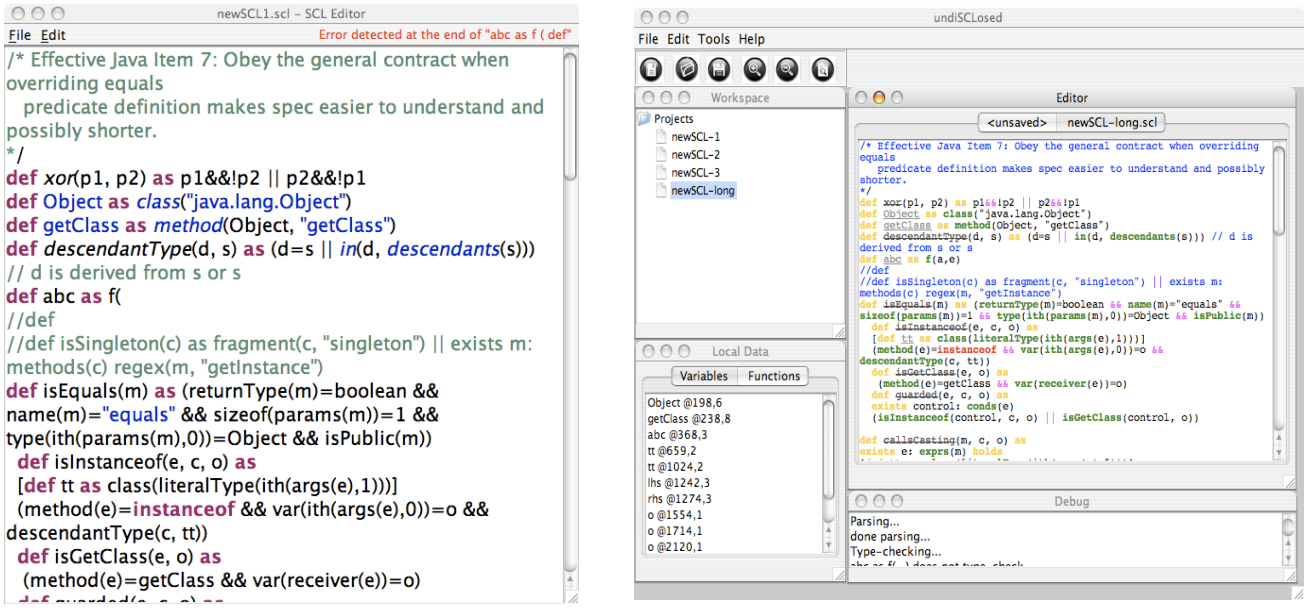

Figure 1: Two representative editors selected from the 11 submissions. The left one is created by modifying the given example application, and the right one has a completely new UI. (Figure better viewed online to see the visual effects.)

ing platform. Swing has been used in industry for almost a decade and the author is reasonably familiar with its design. Its source code is available, which can be a learning aid for some. Furthermore, written tutorials and numerous examples are available for the framework. For example, as of Java 1.6, the official Swing tutorial from Sun contains 179 examples.

We believe that it is too difficult for novices to learn the framework designs *on their own*, and, thus, some initial guidance is necessary to orient them through the complex framework. The first half of the semester was used to teach Swing design. Our students were also encouraged to leverage the rich set of examples provided by the framework, but our emphasis is on framework designs, not evaluating reading techniques as in [21], although the reading techniques were also taught. The lectures systematically guided the students through the main design elements of the framework (Table 1). This architectural focus was forced on us because the size of the framework makes it impossible to cover all of the details within the available time. Thus students were expected to generalize what they learned in class to similar scenarios in the framework. Selected examples were demonstrated in class to facilitate teaching, and programming exercises were given so that students could practice what they learned in lectures. Finally, a comprehensive midterm exam was used to measure how well the students had understood the designs taught.

In the second half of the semester, students carried out a course project while learning an introduction to the principles of interaction design [20]. The 16 undergraduates in our class were randomly divided into 11 one- and two-person teams. Teams were then examined to make certain that each team met certain minimum requirements in order to carry out a course project (e.g., at least one team member must have satisfactory Java and OO experience).

The application to be developed was a source code editor that would highlight language constructs with colors or other styles, mark the multiple occurrences of an identifier (marking from now on), and update the display while code is incrementally modified in the editor (parsing). These are common visual features in modern program editors like the Eclipse Java Editor, and teams were encouraged to study existing editors for inspiration. Figure 1 shows two submissions from our teams. The requirements for the project were specified by the author, and the teams performed the design and implementation. The Structural Constraint Language (SCL) [10] was chosen as the editor's target language because the author knows how to tailor SCL for use in this project. An SCL parser component was provided to the teams to obtain the constructs to be highlighted so that the teams can focus mainly on the user interface behavior.

An example application from the Swing tutorial [1] (Figure 2) was identified as the most suitable candidate with which the teams could start. The most useful solutions provided by the example are operations on a text pane, including the creation of the text pane as well as how to insert text into this widget and how to set visual styles to text regions. SCL program constructs could be highlighted by set-

1 TextComponentDemo.java, about 400 lines of code.

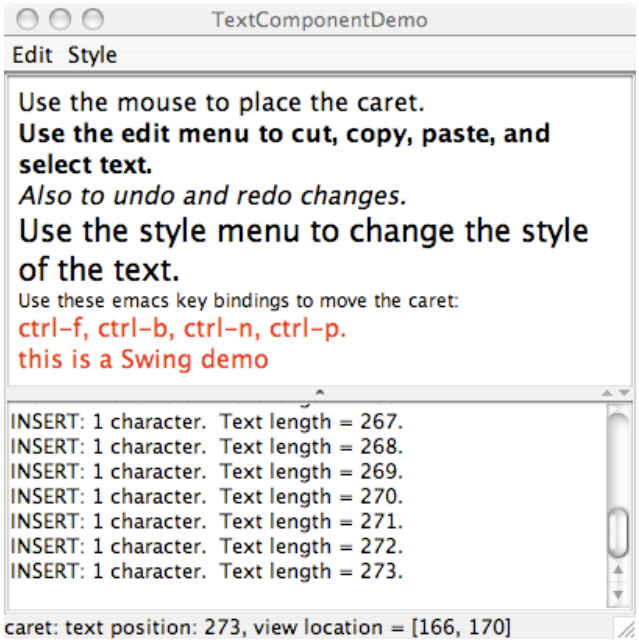

Figure 2: The user interface of TextComponentDemo. The text in the text pane is displayed with different styles and attributes. The text area displays the edits made to the text document. The status bar at the bottom shows the current position of the caret (with respect to both the text document and the text pane view).

ting the appropriate styles to positions returned from the SCL parser. Marking could be implemented by registering a caret listener on the text pane. When edits happen to the SCL program, parsing can be automatically invoked periodically to update the visual effects (e.g., once every character is changed or every 100 milliseconds). Such an automatic incremental parsing strategy can be enabled by registering a document listener on the text document associated with the text pane to detect changes.

4. Research Methods

A set of qualitative and quantitative data were gathered from the course to gain some insight into the research questions raised in the beginning. These include students' experience and background, list of Swing design topics taught, midterm scores, project requirements specification, project submissions, project reports, and project scores. Occasionally, anecdotes the author observed during the course and student interviews were also used as supporting evidence.

Since there has been little work on understanding this area of framework use, the focus of this study was on using the data to search for tentative but reasonable hypotheses and not on testing existing hypotheses. The empirical method of building theories from case study research was

Design topics	Details	Examples/ Exercises/ Midterm(30)
GUI composition	GUI containment, sizing and positioning, layout.	Y/Y/16
Common widgets	text fields, buttons, menus, dialogs, etc.	Y/Y/6
Event dispatching	Event queue, event versus app threads, event loop. Template design pattern.	Y/N/2
Event handling	Event listeners. Observer pattern.	Y/Y/3
Text pane	Document and JTextComponent, styles for document regions, document listener, caret listener.	Y/N/1
MVC	Model-View-Controller and tradeoffs.	Y/N/0
Painting	Painting in AWT and Swing; customizing painting behavior.	Y/N/0
PLAF	Internal design of AWT Pluggable Look And Feel. Abstract factory pattern.	N/N/2

Table 1: AWT/Swing design topics taught, their weights in the midterm exam (with full mark of 30), and whether examples were used in teaching or homework assignments were done.

first proposed in the social science literature [4] but it is also applied in the software engineering discipline [19, 21].

A critical step in this kind of research is for the researchers to become intimately familiar with the cases [4] in order to detect patterns in the data reliably. The most time-consuming task in our study was the detailed, critical evaluation of the code that the 11 teams submitted. This step is necessary because the goal of this study is to understand the effects of framework design knowledge on example-based applications. Initially, the author categorized the cases by design features. For example, to use the text pane, one must know the fact that a document is associated with the text pane. Thus the relationship between the text pane and the document was considered as a framework design feature. A problem with this approach is that it produced a long list of design features for each submission, making it hard to see patterns from the data. Eventually, the data were grouped according to the requirements features, resulting in the data in Table 2, which are further elaborated in Section 5.

Pattern detection from the data was an iterative process of comparing and contrasting data from different teams.

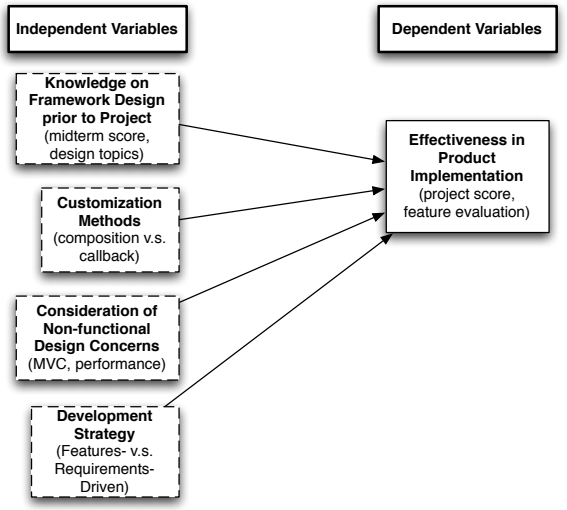

Figure 3: The variables (with measures in parentheses) and relationships (edges) studied.

Typically, a pattern was first identified from a subset of the cases and then verified against all the other cases. Statistical analyses as well as empirical evidence were then used to further confirm the findings. Naturally, some patterns were rejected. Eventually, a stable set of constructs and relationships were identified (Figure 3). Based on these, a set of hypotheses addressing our research questions were formulated, which are reported in the next section.

It is important to note that the hypotheses derived in the next section should be understood as applicable to only example-based application development. In a previous study, it was hypothesized that *example-based techniques are well-suited to use by beginning learners* [21]. Motivated by this and similar theoretical observations (e.g., [3, 17]), an example-based teaching and learning strategy was adopted in our course as well. The students were encouraged to start the project by studying an example identified from the Swing tutorial. In an interview with the author and the final project reports, all of the 11 teams confirmed that they had studied the example. Among the 10 design features identified from the example (not shown in this paper), 8 were reused in project implementation. Thus we conclude that a significant portion of the example has been reused in this study. However, code inspection revealed that our students did not copy-and-paste code from the example verbatim. In particular, clone detection with two tools [2] found only 'uninteresting' clones, for example, the boiler-plate code for starting the event loop. Other than that, only one team's code contains verbatim reuse of a method from the example. Instead, teams reused design features in the example with *modifications*, indicating that the subjects did possess some level of understanding of the design of these features.

5. Results

5.1. Framework design knowledge versus performance in project implementation

Since our goal was to teach the internal designs of the framework to help students succeed in the project, we would expect a strong positive correlation between teams' knowledge about framework design and their performance in project implementation. As shown in Figure 3, the design knowledge about the framework is mainly measured by the midterm score, which reflects a student's mastery of the design topics taught (Table 1), and the project implementation by the project score formulated on the basis of a detailed evaluation of each project, which is summarized in Table 2. While the midterm and project scores allow us to demonstrate the correlation quantitatively, the qualitative data on design topics and the detailed project evaluation provide concrete evidence that helps us explain the nature of the correlation constructively. Nonparametric statistical analyses were used because not all data conform to normal distributions. Our quantitative analyses followed the statistics procedures and the reporting format of [9].

Course projects were evaluated mainly according to how well they implemented the required features. Each feature was assigned one of four ranks [3], which is then multiplied with a weight to obtain a final mark for the feature. The evaluation combined black-box testing of the submitted programs and code inspection by us. The code inspection allowed us to obtain a detailed evaluation of project implementation. It also helped us detect correct parts of solutions that would otherwise be masked by bugs. Finally, 10 percent of the project mark was also allocated to common usability concerns. (For example, widgets must be sized and positioned properly when the top-level frame is resized; an appropriate set of menus must be defined.)

Table 2 depicts the details of the project evaluation, where teams are sorted in descending order of their overall project performance. In general, most teams did well in the first 3 features (UI, Load Save, and Set styles), which can be attributed to the detailed coverage on architectural topics like GUI composition and event handling. The overall trend in Table 2 is that the higher-ranked teams tended to work out larger numbers of features (the sum of required, optional, and extra features), with better quality. Since each of these features requires knowledge of different aspects

2 MOSS. http://theory.stanford.edu/ aiken/moss/: A System for Detecting Software Plagiarism; SimScan. http://blue-edge.bg/download.html: SimScan (Similarity Scanner) is a utility for finding duplicated or similar fragments of code in large Java source code bases. Last verified: Jan. 20, 2008

3 Feature not implemented (0), implemented with major problems (1), implemented with minor problems (2), and implemented fully correctly (3).

Teams[a]	Required Features[b] UI[c]	Load Save	Set styles[d]	Marking	Parsing[e]	Optional[b,f] Undo Redo	Textpane Actions	#Extra[g]	MVC	Issues	Rank[h]
T1	C	3	3,S2	3	3,A	3	3	1	Yes		T
T2*	R	3	3,S3	1	1,M		3	4	Yes		T
T3*	R	3	3,S1	1	1,M	2		3		bugs	T
T4	A	3	3,S1	3	1,M	1	3			UI	T
T5	S	3	3,S1	3	3,A				Yes		T
T6	C	3	2,S1	1	1,M			2	Yes	smells	M
T7*	C	3	3,S1	1	1,A					smells	M
T8*	S	3	2,S1							smells	L
T9	S	2	3,S1					1			L
T10*	S	2	1,S1								L
T11	S	1									L

[a] Teams with or without a star consist of 2 and 1 member, respectively.
[b] Degree of implementation quality: 1: with major problems, 2: with minor problems, 3: fully correct.
[c] What happens to example UI: A: reused As is, S: Simplified, C: Customized, R: Replaced. These actions are sorted in the ascending order of the amount of work involved. All teams are given 3/3 for this feature.
[d] S1: solution provided by example, S2 and S3: advanced solution learned elsewhere.
[e] How parsing is invoked when text changes: A: Automatically, M: Manually.
[f] Features present in example but optional to products.
[g] Number of extra features implemented beyond those in the example.
[h] Ranks of project: T: top, M: middle, L: low.

Table 2: Data analysis of 11 submitted projects.

of the framework, this would imply that the higher ranked teams had better knowledge of the framework. On the other hand, code inspection also revealed evidence that lower-ranked teams have not understood the framework design completely (see Issues column in Table 2). For example, in one team's code, after the document is changed, it is always explicitly passed back to the text pane. It seems that the team is not certain that once established, the subject and observer relation will persist unless it is changed explicitly. In several other cases, instead of reusing, teams created a new text style object whenever needed, unaware of the impact of unnecessary object creation on performance.

Figure 4 depicts the average midterm score a team received versus the project score. Spearman's Rank-Order Correlation Coefficient was used to measure the type and strength of the linear relationship between a team's midterm exam score and project implementation score (with correlation values close to 1 or -1 representing an exact linear relationship and values close to zero representing no linear relationship). A strong [4], positive correlation was found between a team's performance in learning framework designs and effectiveness in project implementation (with r_s= 0.707, which is significant at $\alpha = 0.05$ level, where r_{crit}= 0.535 for $N = 11$).

However, an ${r_s}^2$ value of 0.50 means that students' performance in learning framework designs accounted for only 50% of the observed variance in the project scores. This correlation value implies that although the performance in learning designs an important factor in predicting performance in project implementation, other factors might have contributed to the variation in project scores as well. In the following, we analyze three possible factors.

One is effort. Through informal interaction with the teams, the author knew that two specific teams did not devote to the project as much effort as the rest of the class, and consequently, achieved low marks in the project implementation. After these two teams were removed, an r_s of 0.82 was obtained , which is significant at $\alpha = 0.05$ level. A larger ${r_s}^2$ value of 0.67 highlights more of the impact of framework design learning on project implementation.

Another is the effectiveness of the midterm exam in detecting the variance among students' knowledge about the framework. While the design topics on GUI composition, common widgets, event handling, and text pane (Table 1) have clearly contributed to the general success of most teams in the 3 features of UI, Load Save, and Set styles (Table 2), these features account for only about half of the project score. Teams exhibited more variance in implementing other features (marking, parsing, and undo/redo). Al-

4 According to [9] (pp. 164), a correlation coefficient larger than 0.5 is considered strong.

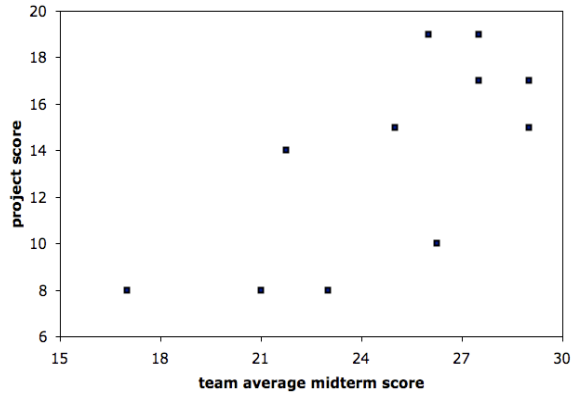

Figure 4: The average team midterm exam score (testing how well the team has learned the framework designs, max 30) and its correlation with project implementation score (max 20).

though the knowledge required to implement these features was taught and tested in the exam, maybe the test was not critical enough to reveal the difference.

Yet another is the difference between individuals' program comprehension ability. Some students may have outperformed others in understanding the details of the example, which contributes to the difference in implementation. Or students may vary in their ability in generalizing from what they have learned to new cases (e.g., the implementation of marking and parsing essentially relies on knowledge of two specific event listeners, which are special cases of the general topic of event handling). Alternatively, these features may be simply inherently hard for some students. We will elaborate on this issue further later.

Irrespective of any other factors that may also contribute to the difference in project implementation, an ${r_s}^2$ of 0.50 indicates that design knowledge about the used framework is a major factor, if not the most important one. Based on these analyses, we formulate our first hypothesis:

HYPOTHESIS 1: When learning to use a framework from examples, the more architecture knowledge about the framework the novice developers possess, the better they will perform in adapting the examples. However, the effectiveness of architectural knowledge on project implementation may vary among individuals due to difference in their ability of performing program comprehension and generalization.

In addition to project scores, the number of required features that a team worked out is another measure of their performance in project implementation (the Required Features column in Table 2). If the numbers of required features positively correlate with the final project scores, it would increase our confidence in the correlation reported (between framework designs learned and project performance in terms of project scores). A Spearman's Correlation Coefficient was calculated between project scores and number of required features and a strong, positive correlation was found between the two conditions ($r_s = 0.714$, which is significant at $\alpha = 0.05$ level, with r_{crit}= 0.535 for $N = 11$). This provides some confidence that our project scores indeed reflect team performance.

Table 2 shows that 10 teams were able to create a decent UI, load/save, and set styles for an SCL file. But only 3 teams correctly implemented occurrence marking (using CaretListener). Code inspection revealed that 4 other teams implemented it only partially correctly because they used a MouseListener. Only 3 teams were able to correctly implement automated parsing, which requires good knowledge about AWT/Swing's event dispatching. All other teams required a user to explicitly invoke the parsing function.

While searching for possible explanations for this difference in implementing these features, it was noticed that our students seemed to do better with framework features that can be accessed by function calls than features whose customization requires callbacks. In Table 2, the first 3 features from the left can be implemented mainly by function calls, and the next 3 require callbacks. The average ranks that the 11 teams achieved for these features were calculated. Using the average ranks, a one-tailed Mann-Whitney U test between the two groups of features (calls versus callbacks) was performed, and the result was significant (U1=0, equal to the one-tailed critical value 0 of Mann-Whitney U). This provides some supporting evidence that function call features indeed are implemented better than callback ones. Maybe this is due to the fact that our students are more familiar with function calls than callbacks. Another reason may be that features that involve callbacks participate in more complex interactions in the framework, and thus their usage simply requires deeper knowledge of the framework in general. However, we must note that our teams have been encouraged to implement as many features as possible, and thus it is less likely that the difference is caused by the lack of emphasis in requirements of the callback features.

HYPOTHESIS 2a: Framework features whose usage requires callbacks are more difficult to learn and modify than ones that require function calls.

HYPOTHESIS 2b: The more design knowledge about the framework it requires to use a framework feature, the more difficult it is to learn and modify.

One popular design consideration for GUI applications is the separation of Model, View, and Controller. Our teams were encouraged to structure their editors into the model-view-controller paradigm. However, through code inspection and project reports, only 4 teams were identified as having attempted using MVC in their code. One possible explanation is that for beginners, MVC, and non-functional concerns in general, come after functionality understanding. It is likely that students are already so overwhelmed

with understanding framework functionality that they don't have the time to work on non-functional concerns.

HYPOTHESIS 3a: In the early phase of learning a framework, with limited time available, novice learners are more likely to focus on functionality than other non-functional concerns like structure or performance.

Consequently, effectiveness in addressing non-functional concerns could be a good predictor of functional quality but not vice versa. Interestingly enough, two teams who did not use MVC were also top-ranked (T3 and T4 in Table 2).

To gather further supporting evidence, a one-tailed Mann-Whitney U test was performed to test whether teams who used MVC also achieved higher ranks in the feature of parsing (Table 2). The result is significant (U1=3, less than the critical value 4 at $\alpha = 0.05$ for $n1 = 4$ and $n2 = 7$). An eta square of 0.43 indicates that MVC accounts for 43% of the variance in the ranks of the parsing feature, which is fairly strong. In addition, in several other cases, we also observed that several top-performed teams were able to make use of framework features that would make a difference in efficiency, which other teams did not use. Based on these, we formulate the following hypothesis:

HYPOTHESIS 3b: Products which have considered non-functional design concerns like structure and performance are more likely to implement advanced requirements than ones that don't.

In trying to understand what caused the difference among teams, it was noticed that some teams had tried to add many nice, but not required, features to their products, while others seemed to be more conservative, first making sure that the requirements be satisfied. This can be seen from the data in Table 2, where T2, T3, and T6 seem to have adopted a feature-driven approach, implementing 4, 3, and 2 extra features, respectively. Moreover, T2 and T3 also implemented a new UI from scratch. (The right of Figure 1 shows one of them.) T6, although did not implement a new interface, chose to customize the example UI, which still required more work than simply using the UI as is or simplifying it. In contrast, the other 8 teams were more conservative and tried to reuse the example as much as possible. The three teams (T1, T4, and T5) that did the best in the two advanced features (occurrence marking and automated parsing) made only minimal changes to the example UI. T4 even kept the example UI without changing anything. Since much can be reused from the example, given the limited amount of time available for the project, the conservative reuse approach seems to work better than the feature-driven approach because it allowed teams to reuse the example UI, saving time to focus on required features rather than nice UI.

A one-tailed Mann-Whitney U test was performed using the data of the first 7 teams. The test result indicates that teams adopting a conservative reuse approach (T2, T3, T6) achieve significantly higher total ranks in the two advanced features than the other 4 teams (T1, T4, T5, T7). The effect size of this relationship is measured by an eta square of 0.75 obtained with the Rank Sums Test. Thus the adoption of a conservative reuse approach accounts for 75% of the variance in implementing the two advanced features.

HYPOTHESIS 4: When the example contains the required features, under a tight schedule, a conservative reuse approach to development can be more effective in fulfilling requirements than a feature-driven approach.

5.1.1. Potentially confounding factors: team size, academic programming experience, and effort Our 16 students were divided into 11 teams of 1 or 2 members. One concern is whether the 2-member teams performed better than the 1-member ones. In Table 2, the two types of team seem fairly equally distributed among the top, middle, and low ranks. The chi-square test for this yielded an ${X_{obt}}^2$ of 0.136, which is not significant at the $\alpha = 0.05$ level where ${X_{crit}}^2$ is 5.99 for $df = 2$. This provides evidence that in our setting, team size has no effect on project performance.

We were also concerned that the effectiveness of our teams might have more to do with the level of experience that the team members had with implementing similar projects than with the variables under study in our experiment. Since none of our students had either prior industrial programming experience or experience with AWT/Swing, we measured their academic programming experience in terms of the academic year they were in. We partitioned the students' project scores into two groups according to their 3rd and 4th year status. We then performed a two-tailed Mann-Whitney U test to determine whether the senior group ($M = 13.80$, $SD = 4.17$) and the junior group ($M = 13.55$, $SD = 4.16$) performed significantly differently in the project in terms of the received scores. The result is insignificant, with U_{obt}= 15, greater than U_{crit}= 3 at $\alpha = 0.05$ level for $n1 = 5$ and $n2 = 6$. This test provides some evidence that academic programming experience has an insignificant effect on project scores.

Yet another concern is that a team's performance in the project might have been influenced more by their effort than the amount of design knowledge that the students have learned about the framework. However, this appears unlikely in our case for two reasons. First, most of our students were motivated to work hard on the projects, and most teams reported that most of their effort was spent during the last several weeks of the semester. This implies that the variations among team effort should not be too large. Because of the small class size and the close interaction between the author and the students, we believe that this factor is within control. Second, design knowledge is critical to

beginning learners. If they had not learned enough from lectures, it would be extremely hard to make up by spending extra effort on their own. In other words, we believe that in the scenario of learning frameworks with limited time available, design knowledge dominates effort.

5.2. How can framework designs be better taught?

We observed that not all the example features were used by all teams (Table 2). Our teams tended to focus on features most useful to them, and would replace features that they were not confident with by alternative solutions that they were familiar with. For example, two teams used a KeyListener to define short keys instead of the built-in support in the framework (keymap). A similar case is with actions. In the original example, actions were used consistently to create menu items. Some teams completely removed actions and used ActionListener instead, which they were presumably more familiar with. Finally, all but two top teams removed the redo/undo feature from the example. Both teams who did include redo/undo modified it, and both made minor mistakes in their modifications.

Unlike the design topics we taught to students (Table 1), which for the most part were focused on architecture, these features are local to individual components of the framework. The large number of components make it impossible to cover all of them in detail. Perhaps in addition to teaching framework architecture, a set of component-level features could be selected according to the need of the project and taught in detail as well.

Lesson 1: In addition to the architectural design of the framework, further guidance is needed to facilitate the use of component-specific features, the coverage of which may be determined according to the need of the course project.

We also observed that our students did not perform equally well across the design topics taught. For example, all 11 teams were able to produce a well-behaved UI, either by customizing the example UI or creating a completely new one (Figure 1). But only a few teams did well in features that require knowledge of DocumentListener, CaretListener, and event dispatching. One difference in our way of teaching these design topics is whether a programming exercise was used to reinforce learning (see Table 1).

Lesson 2: Programming exercises can be used as an effective mechanism for reinforcing the learning of individual framework features. Whether designing exercises for a particular framework feature can be guided by such characteristics of the feature as whether it involves callback or whether its usage requires 'deep' knowledge of framework internals.

In light of Hypothesis 4, in retrospect, some of our students could have done better if they had been advised to focus on the example, at least initially in the project, before moving on to other features.

Lesson 3: When running a course project based on an example that contains most of the features needed by the project, students should be reminded not to move too far away from the example.

6. Threats to Validity

This section discusses threats to validity that can affect the results reported in Section 5, following a well-known template for case studies [23].

Regarding *construct validity*, threats can be due to the measurement performed. One tactic to enhance construct validity is triangulation: the use of multiple sources aimed at corroborating the same fact or phenomenon. Our study applied data triangulation by including two measures (project scores and number of features implemented) for the same aspect of interest (performance in project).

Threats to *internal validity* may confuse spurious relationships with correct causal relationships. We undertook qualitative and quantitative analyses to test the effects of potentially confounding factors (differences in team size, effort spent, and previous experience) which could be rival explanations to our findings. As pointed out in section 5.1.1, for our particular setting, the three confounding factors do not appear to have confounded our results.

Threats to *external validity* are related to the extent to which our findings can be generalized. First, AWT/Swing is clearly a typical framework and our observations on learning framework should not be specific to this particular framework. Second, our project is more closely related to real world applications than toys. Third, although our subjects are novices, the problems they encountered would also be problematic for expert developers without experience with the framework. Even if further investigation showed that the validity of our hypotheses held only for novices, they can still be significant since experts are always scarce.

Regarding *reliability validity*, all of the data referenced in this paper are available from the author, including project submissions (with author identity removed), midterm exams, student scores for both midterm and project, charts for statistical analysis. So others can replicate the analysis. The statistical conclusions described in Section 5 were supported by proper tests, Spearman's Rank-Order Correlation Coefficient for analyzing correlation, Mann-Whitney U and Chi-sqaure for testing independence.

7. Conclusion

While software is increasingly being developed using libraries and frameworks, many developers are still relying on a trial and error approach to reuse. Such practice is costly and the outcome is unpredictable, producing software that is buggy, hard to change, and fragile. It is not entirely clear either how a more disciplined, design-driven approach can be used in framework learning. This paper analyzes the data

gathered from a framework-based course project where students were first systematically taught the architectural designs of the framework. The contribution is a set of hypotheses, along with supporting evidence, that characterize the various effects of framework designs on project outcome.

The two questions raised in the introduction of this paper are addressed as follows. Hypotheses 1, 2a, 2b, 3a, 3b and 4 address Question 1. Overall, students who had learned the framework designs better exhibited stronger ability in correctly adapting example solutions. Framework designs that require callbacks and deeper behavioral customization were more difficult to learn than those that involve only function calls. Initially, novice learners appeared to focus on learning functional aspects of the framework than non-functional aspects. Finally, a conservative reuse strategy helps novice learners focus on gaining a comprehensive understanding of the example rather than being distracted by 'nice' features. Question 2 is addressed by the three lessons learned, which point out how framework designs can be taught more effectively, e.g., by balancing the allocation of teaching effort between framework architecture and component-specific features, and by designing programming exercises. A difficulty with teaching framework designs is the limited amount of time available. Teaching time could be more effectively used by reducing the time spent on those framework design topics not immediately useful to the course project, for example, in our case, the topics of PLAF, painting, or MVC in Table 1. Overall, it is encouraging to see that after only 20 hours of lectures, our students could produce programs that exhibit features of industrial products. We feel that similar practice is applicable to industry.

Acknowledgments

The author is grateful to the three anonymous reviewers, H. James Hoover, Chandan Rupakheti, and Patricia Jablonski for their useful comments on earlier drafts of this paper, Sumona Mondal for patiently answering my statistics questions, and students of EE 408, Fall 2007 for their hard work.

References

[1] M. Antkiewicz and K. Czarnecki. Framework-Specific Modeling Languages with Round-Trip Engineering. In *Proceedings of MoDELS '06*, 2006.

[2] D. S. Brandt. Constructivism: Teaching for Understanding of the Internet. *Commun. ACM*, 40(10):112–117, 1997.

[3] M. T. H. Chi, M. Bassok, M. Lewis, P. Reimann, and R. Glaser. Self-explanations: How Students Study and Use Examples in Learning to Solve Problems. *Cognitive Science,*, 13:145–182, 1989.

[4] K. M. Eisenhardt. Building Theories from Case Study Research. *Academy of Management Review*, 14(4):532–550, 1989.

[5] G. Fairbanks, D. Garlan, and W. Scherlis. Design Fragments Make Using Frameworks Easier. In *Proceedings of OOPSLA'06*, pages 75–88, 2006.

[6] G. Froehlich, J. Hoover, L. Liu, and P. Sorenson. Hooking into Object-Oriented Application Frameworks. In *Proceedings of ICSE'97*, Boston, MC, May 1997.

[7] D. Gangopadhyay and S. Mitra. Understanding Frameworks by Exploration of Examplars. In *Proceedings of 7th International Workshop on Computer Aided Software Engineering (CASE-95)*, pages 90–99, July 1995.

[8] J. Hautamäki and K. Koskimies. Finding and Documenting the Specialization Interface of an Application Framework. *Softw. Pract. Exper.*, 36(13):1443–1465, 2006.

[9] G. W. Heiman. *Basic Statistics for the Behavioral Sciences (Fifth Edition)*. Houghton Mifflin Company, 2006.

[10] D. Hou and H. J. Hoover. Using SCL to Specify and Check Design Intent in Source Code. *IEEE Trans. Software Eng.*, 32(6):404–423, June 2006.

[11] D. Hou, K. Wong, and H. J. Hoover. What Can Programmer Questions Tell Us About Frameworks? In *Proceedings of IWPC '05*, pages 87–96, 2005.

[12] R. E. Johnson. Documenting Frameworks with Patterns. In *Proceedings of OOPSLA 92*, Vancouver, Canada, 1992.

[13] R. E. Johnson. Frameworks = (Components + Patterns). *Commun. ACM*, 40(10):39–42, 1997.

[14] P. Mohagheghi, R. Conradi, O. M. Killi, and H. Schwarz. An Empirical Study of Software Reuse vs. Defect-Density and Stability. In *Proceedings of ICSE '04*, pages 282–292, 2004.

[15] M. Morisio, D. Romano, and I. Stamelos. Quality, Productivity, and Learning in Framework-Based Development: An Exploratory Case Study. *IEEE Trans. Softw. Eng.*, 28(9):876–888, 2002.

[16] S. Moser and O. Nierstrasz. The Effect of Object-Oriented Frameworks on Developer Productivity. *IEEE Computer*, 29(9):45–51, 1996.

[17] M. B. Rosson, J. M. Carrol, and R. K. E. Bellamy. Smalltalk Scaffolding: a Case Study of Minimalist Instruction. In *CHI '90: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 423–430. ACM, 1990.

[18] K. Schneider and A. Repenning. Deceived by Ease of Use: Using Paradigmatic Applications to Build Visual Design Environments. In *DIS '95: Proceedings of the 1st conference on Designing interactive systems*, pages 177–188. ACM, 1995.

[19] C. B. Seaman and V. R. Basili. An Empirical Study of Communication in Code Inspections. In *Proceedings of ICSE '97*, pages 96–106, Boston, MC, 1997.

[20] H. Sharp, Y. Rogers, and J. Preece. *Interaction Design: Beyond Human-Computer Interaction (Second Edition)*. Addison Wesley, March 2007.

[21] F. Shull, F. Lanubile, and V. R. Basili. Investigating Reading Techniques for Object-Oriented Framework Learning. *IEEE Trans. Software Eng.*, 26(11):1101–1118, 2000.

[22] K. Walrath, M. Campione, A. Huml, and S. Zakhour. *The JFC Swing Tutorial (Second Edition)*. Addison Wesley, February 2004.

[23] R. K. Yin. *Case Study Research: Design and Methods (Third Edition)*. SAGE Publications, London, 2002.