Journal of Software Engineering

ISSN 1819-4311

Academic Journals Inc.

www.academicjournals.com

Journal of Software Engineering 10 (1): 1-15, 2016
ISSN 1819-4311 / DOI: 10.3923/jse.2016.1.15

AJMU: An Aspect-Oriented Framework for Evaluating the Usability of WIMP Applications

Sandra Casas, Natalia Trejo and Roberto Farias
Research Group Pragmatic Software Engineering, Institute of Applied Technology, National University of Austral Patagonia, Lisandro de la Torre 860-9400, Río Gallegos, Argentina

Corresponding Author: Sandra Casas, Research Group Pragmatic Software Engineering, Institute of Applied Technology, National University of Austral Patagonia, Lisandro de la Torre 860-9400, Río Gallegos, Argentina Tel: +5492966211078 Fax: +5492966442377

ABSTRACT

Aspect-Oriented Programming (AOP) is recognized as one of the main techniques for separation of concerns. AOP has been used for automatic evaluation of the usability of WIMP applications, with the aim of monitoring events and GUI components and recording data in logs. A weakness of AOP approaches is low semantic value and a low level of abstraction of the results. One possible solution to overcome these limitations is to use an AO framework for assessing the usability of user tasks. However, little research has been devoted the development of AO frameworks from scratch. Although different design and programming patterns are available, insufficient experience has been reported regarding their application in the development of frameworks. This paper presents AJMU, an AO framework for the automatic evaluation of the usability of user tasks in desktop applications. AJMU was developed from scratch, using AO patterns. This paper also reports on experiments involving AJMU's instantiation with real applications.

Key words: Usability, framework, aspect-oriented programming, design patterns, Aspect J

INTRODUCTION

Aspect-Oriented Programming (AOP; Kiczales *et al*., 1997) has been used to implement the automatic evaluation of the usability of WIMP applications (Bateman *et al*., 2009; Holzinger *et al*., 2011; Humayoun *et al*., 2009; Shekh and Tyerman, 2010; Tao, 2008, 2012; Tarta and Moldovan, 2006). The predominant approach in this research has been to trace events and the execution of GUI components and store the data in a log, without any context of greater significance. Consequently, the evaluation of usability has little relevance for the evaluator and is not at an appropriate level of abstraction.

An AO framework that automatically assesses the different factors of usability (efficiency, effectiveness and satisfaction) for user tasks would be a more appropriate tool. A user task is a unit of analysis that is more relevant and more complex, requiring the implementation of diverse functions, relationships and interactions. Even though popular AO frameworks exist (Spring AOP and JBoss AOP), research on AO frameworks is still in a nascent stage. AOP has primarily been proposed to overcome the weaknesses of OO frameworks (Kulesza *et al*., 2006; Santos *et al*., 2007; Vaira and Caplinskas, 2011) and not so much for building frameworks from scratch.

When developing a framework rather than an ordinary application, some program constructs or "design elements," are very important. These design elements include abstract modules (classes

or aspects), design patterns, contracts and so on. The way in which AO features are applied is important, because this has a direct impact on how reusable aspects are and how easily they can be applied to other programs. Hence, the application of AO language features cannot be ad hoc and must follow deliberate design decisions. A number of studies have examined design and programming patterns and strategies to solve common problems in the development of modules and applications (Bynens *et al*., 2007; Griswold *et al*., 2006; Hanenberg and Schmidmeier, 2003a, b; Hanenberg and Unland, 2001; Laddad, 2002; Lagaisse and Joosen, 2006; Miles, 2004; Noble *et al*., 2007). However, there are few conclusive contributions concerning the use or application of AO patterns for framework development from scratch.

The present study arises from the following questions: How should an AO framework for automatic evaluation of the usability of user tasks in WIMP applications be developed? What strategies can be applied to achieve reusable and flexible design and implementation of the AO aspects in such a framework?

This study makes two contributions. First, the paper presents AJMU, a framework designed and implemented with aspects for evaluating the usability of desktop applications at the level of tasks. Second, the paper uses AJMU as a case study to analyze the application of AO patterns in the design and implementation of a framework that is built from scratch.

The aim of this paper is to report on the design and implementation of an AO framework for evaluating the usability of tasks in WIMP applications. At the same time, it seeks to provide empirical evidence that the strategies used achieve a reusable, flexible and feasible design for future evolution. The results comprise a general UML model of the modules and relevant code segments for the functions, relationships and interactions in the model and the applied design patterns are specifically described.

MATERIALS AND METHODS

The development process for AJMU was iterative and incremental (Pree, 1995). Beginning with the main, initial idea of placing the task at the center, as a concept and an entity, subsequent iterations consisted of activities such as requirement analysis, design, implementation and testing. The first iteration focused on the representation of a task and its connection to the domain, the second iteration on the logging service, the third iteration on the event metrics, the fourth iteration on the satisfaction metrics and profile data and the fifth iteration on settings. Most iterations required redesigning and reimplementing some of the modules developed in previous iterations. Throughout the process, black box tests were performed to validate the correct functioning of the framework (metric calculation, log registration and data capture, etc.). Throughout this process, the guidelines for AO patterns (Section 1) were followed to verify which of these should be implemented according to the requirements and constraints arising from the problem. AJMU was coded in Java and AspectJ (Kiczales *et al*., 2001) and the modeling was specified with UML diagrams.

Finally, various experiments were performed with real applications, in order to assess the pros and cons of adopting AJMU.

AJMU framework: A framework can be defined as an application design along with its implementation. A classic categorization (Adair, 1995) distinguishes three types of frameworks:

Table 1: Usability evaluation of tasks

Factors	Subfactor	Metrics
Efficiency		Time to perform the task
Effectiveness	Errors	
Effectiveness	Errors	Number of exceptions during task execution
Effectiveness	Errors	Number of error messages during task execution
Effectiveness	Errors	Number of alerts during task execution
Effectiveness	Errors	Number of informative messages during task execution
Effectiveness	Errors	Number of interrogative messages during task execution
Effectiveness	Errors	Number of dialogues during task execution
Effectiveness	Errors	Number of help accesses during task execution
Effectiveness	Completeness	Completed tasks
Effectiveness	Completeness	Incomplete tasks
Satisfaction	Satisfaction	Satisfaction with the complexity of the task
Satisfaction	Satisfaction	Satisfaction with the time required to perform the task
Satisfaction	Satisfaction	Satisfaction with use of the application to perform the task
Profile		Age, sex and training

application frameworks, domain frameworks and support frameworks. Domain frameworks such as AJMU capture the knowledge and expertise related to a particular domain problem.

The AJMU framework allows the evaluation of the usability of WIMP applications in terms of tasks. The components of the framework are a set of modules, aspects and one class. The relationships and interactions between these components make the AJMU framework highly reusable with minimal specialization (configuration) requirements.

Evaluation of the usability of a user task (Nielsen, 1992; Ivory and Hearst, 2001) must consider a number of factors. We have adopted the ISO 9241-11 (ISO 9241-11, 1998) attributes: efficiency, effectiveness and satisfaction. We chose metrics to analyze these attributes and the metrics also determine which data about user interactions should be collected. The factors and metrics supported by AJMU are presented in Table 1.

A user task can involve completing a form or performing a sequence of actions in which the user interacts with the application, i.e., tasks can have different levels of complexity. The strategy used in AJMU was to place the user task at the center of the design and implementation of the framework, as both a concept and an entity. The aspects connect the task with the domain and the required services for evaluation, so the user task can be easily identified and analyzed. Thus, while the task is running, aspects capture the events of interest (start, end, errors) and record the data required for calculating the metrics in relation to the task. Consequently, it is possible to obtain data that can be interpreted at a higher semantic level and thus achieve usability results at a higher level of abstraction.

The diagram in Fig. 1 shows the design of AJMU. In what follows, we analyze the most important aspects of the design and implementation of the modules. Details have been removed in order to highlight the code patterns. The hot-spots are identified in each case.

The Task class represents a user task and is the only class in the framework. The Task class includes several attributes representing states that accumulate metric values or subsequently allow their calculation. These states are initialized when the task is instantiated. For example, the init attribute initializes the system time and complete attributes with the value false. These two attributes will be updated when the task ends and will be used to calculate the time (the efficiency of the task) and whether the task was completed (the effectiveness of the task). The Task class also provides methods to properly update each attribute that counts errors or is related to satisfaction.

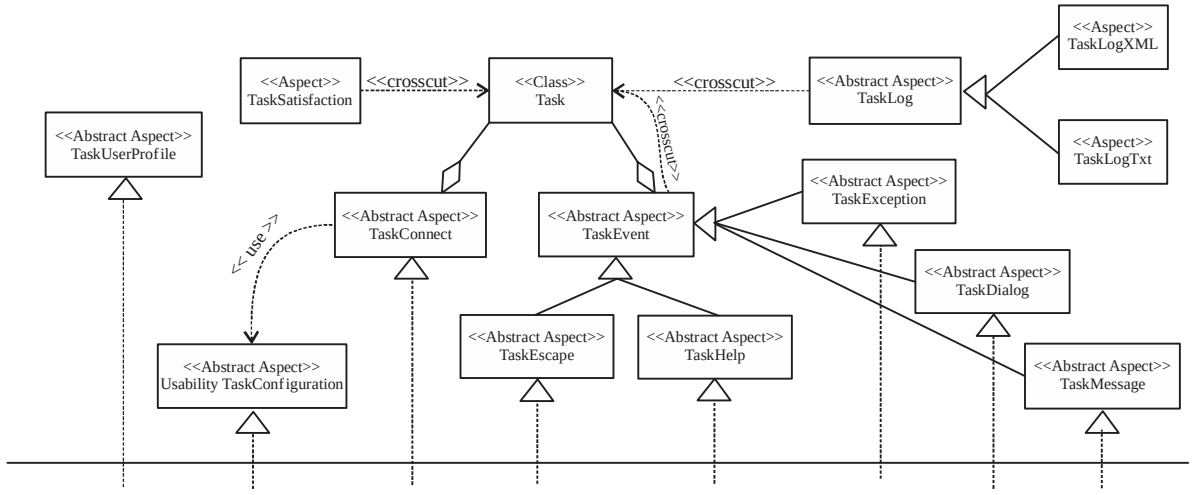

Fig. 1: Diagram of AJMU

```
class Task      {
   private String id,
   private Time init, end;
   private boolean complete;
   private int #exception, sat1, sat2, sat3;

   Task(...){
       // initialize states
    ...
   finalize()
       // update states
    ...
       }
}
```

The role of the TaskConnect abstract aspect is to connect the task with the application (domain) to be evaluated. Two abstract pointcuts, startTask and endTask, are needed to define the joinpoints (class and domain method) to determine the beginning and end of each task. These pointcuts are associated with advices that order the task instantiation and completion, respectively and must be defined in a concrete aspect by the developer. One sub-aspect of Task Connect should be created for each different user task that is to be evaluated. Instantiation of a task is possible because TaskConnect has a task field (object Task type) that is instantiated by this aspect when the startTask pointcut is activated and the associated advice is executed and this occurs before each task starts. Similarly, this aspect orders task completion when the finalize method is invoked by the advice associated with the endTask pointcut after the joinpoint execution.

The set IdTask method should also be redefined by the developer for each task. Its purpose is to provide specific values of the task to be assessed and it will later be used to identify the task in contexts in which more than one task is evaluated, so it requires configuration for each task.

```
abstract aspect TaskConnect {
   private String idTask;
   private Task t;
   abstract void setIdTask(); <- hot-spot
   abstract pointcut startTask; <- hot-spot
   abstract pointcut endTask; <- hot-spot
```

```
    before() : startTask();
    {   this.setIdTask();
            t = new Task(idTask);}
    after() : endTask()
    { t.finalize(); }
}
```

Data logging for a task seems to be a simple, uncomplicated function. However, this service has several characteristics that must be addressed: (a) the logging for a task is performed at different times during its execution and upon various events that may or may not happen; (b) if the log functionality is encapsulated in modules (aspects) that record the events, this will generate scattered and tangled code, which will be a drawback for subsequent framework evolution and (c) the log can be recorded in different formats (text file, XML and database). Therefore, to accommodate these characteristics, we designed an aspect hierarchy for the logging task. The abstract aspect TaskLog establishes a set of pointcuts and abstract methods. The pointcuts are concrete and therefore entirely subject to the execution of the task (not the domain). Abstract methods allow log formats to be defined. The TaskLogXML and TaskLogText aspects of TaskLog extend and implement the log in XML and TXT, respectively, by encoding events, initTask, endTask and log methods. The initTask and endTask methods record data about the initiation and end of the task execution, while the event and log methods record data during the task execution. The log method is invoked by other aspects to record contextual data about an event. Specifically, the number of events in a task, discriminated by type, are recorded in different accumulators, along with additional information that reports where or when these events have occurred.

```
abstract aspect TaskLog {
    abstract void events(Task t);
    abstract void initTask(Task t);
    abstract void endTask(Task t);
    abstract void log(Task t);

    pointcut logStart(Task t) : execution (Task.new(…)…)
    pointcut logEnd(Task t) : execution (Task.finalize(…)…)
    pointcut logEvent(Task t) : call (*.Task.setQ*(…)…)
    after(Task t) : logStart(t)
    { initTask(t);}
    after(Task t) : logEnd(t)
    { endTask(t);}
    after(Task t) : logEvent(t)
    { events(t);}
}
```

The evaluation of the effectiveness factor includes metrics to calculate the number of errors produced. Unlike other events, the succession of errors cannot be associated with unique, specific run times or locations (in the code), which would facilitate identifying and counting them. The errors occur “during” task execution and they manifest in different ways, which leads to different types of errors. Noting further that execution points may also be associated with other tasks, it is necessary to accurately identify the errors associated with the task that is being evaluated. Therefore, the aspects that collect information related to these metrics should consider three critical factors: they should (a) identify different types of errors, (b) capture the succession of errors at any point in the control flow of the task that is being assessed and (c) discriminate errors occurring at a point in the task’s control flow that do not correspond to the task that is being evaluated.

To meet these criteria, we designed a hierarchy of aspects, in which the first two levels are abstract. The TaskEvent aspect is at the top of the hierarchy; it simply defines a set of pointcuts that allow definition of the set of joinpoints in the execution control flow of the task for which errors must be intercepted. A reference to the task that is being assessed is returned from its instantiation (pointcut init) and the TaskEvent aspect stores this reference in the taskRef field. The TaskEvent aspect also defines two abstract elements: the complete pointcut and logEvent methods that must be implemented in the remaining levels of the hierarchy.

```
abstract aspect TaskEvent {
   abstract void logEvent(); <- hot-spot
   private Task taskRef=null;

   pointcut init(Task t) : initialization(Task.new(…));
   after(Task t) : init(t)
   { taskRef=t;}

   pointcut end(): execution (* Task.finalize());

   pointcut aspectFlow() : cflow(adviceexecution);
   pointcut initFlow() : cflow(init(…));
   pointcut endFlow() : cflow(end());
   pointcut isATask() : if ((taskRef!=null) and (!taskRef.isComplete()))
   abstract pointcut complete(…);
}
```

At the second level of the hierarchy, there is a set of aspects that adjust and/or complete the pointcuts to capture specific errors, based on the definitions of TaskEvent. The TaskException aspect counts exceptions, the TaskDialog aspect counts the number of dialog boxes that run and the TaskMessage aspect captures the invocations of dialog boxes and evaluates the message types as warning messages, informational messages, error messages, interrogative messages, or plain messages. The TaskHelp aspect counts the online help accesses and the TaskEscape aspect examines whether the task was abandoned before normal completion.

The design of these aspects uses common as well as diverse strategies. In all cases, the complete pointcut is implemented at this level; it reuses the definitions of inherited pointcuts and is completed with the necessary primitives and/or conditions to capture the specific event. In all cases, the advice associated with a pointcut updates the Task object (corresponding error counts) and orders a contextual data log update. The TaskException and TaskDialog aspects follow this scheme.

```
abstract aspect TaskException extends TaskEvent{
   pointcut complete(Throwable e) : !aspectFlow() & & !endTask() & & isATask()
      & & handler(Throwable+)…

   before(Throwable e) : complete (e)
   { taskRef.setQException(…);
      logEvent(taskRef);
   }
}
```

Through a complete pointcut expression, the TaskException aspect allows interception of exceptions that are generated during the execution of the task, using the previous definitions (aspectFlow, endTask and isATask) and incorporating the specific primitive (handler). The exception counter is updated (setQException) and the logEvent method updates the log with

contextual data. The logEvent method is not implemented at this level, since it depends on the log format that the developer chooses. A specific, concrete aspect, which must be created by the developer, is required and will adjust the logEvent method with the selected log format (XML, TXT or other). For this configuration, only one line of code is needed, as follows:

```
void logEvent (Task t) {
   TaskLogXML.aspectOf.log(t);
}
```

Through a complete pointcut expression, the TaskMessage aspect intercepts the GUI component to which a message corresponds (the component is inherited from JOptionPane), but this definition is not sufficient to discriminate between different types of messages. The identification of each message type is performed through the conditional expressions that analyze how these objects were configured when created. These conditional expressions are part of the advice and they are indispensable, since it is not possible to independently intercept each message type.

```
abstract TaskMessage extends TaskEvent {
   before(…) : complete (…)
   { if (type of message = informative)
      taskRef.setQInforMessage(…);
     if (type of message = …)
   …
   …
   logEvent(taskRef);
}
```

A special situation activates the TaskEscape aspect when a task is abandoned or voluntarily terminated by the user before it is completed. This is extremely important because it allows the logging of data corresponding to the effectiveness factor-since it allows the completeness of the task to be analyzed. This event can happen at any time when the user closes the window and the java.lang.System.exit method is invoked. But applications may also have other ways to cancel the execution of a task at defined points-for example, before saving, the user can use a specific button to cancel the operation (task). This last condition is specific to the application and might not even exist, so it must be set by the developer. The TaskEscape aspect is abstract, using the pointcut complete and reusing the above conditions and also capturing the output of the application or particular conditions by the pointcut not Complete. The associated advice for the pointcut complete updates the task and logs contextual data and then terminates the task.

```
abstract aspect TaskEscape extends TaskEvent{
      abstract pointcut notComplete(); <- hot-spot
      pointcut complete ():call(void java.lang.System.exit(…)) and and
      !endFlow() and and !aspectFlow() and and isATask() and and notComplete();
}
```

A similar situation occurs with the TaskHelp abstract aspect, whose purpose is to count hits for assistance during the task execution. The help may be an option on a menu or a button on a bar or both. This variability must be specifically determined in a concrete aspect and represents a new hot-spot, using the approach that has been described for the TaskEscape aspect.

When the finalization of a task is reached (the finalize method of the Task class is invoked by the TaskConnect aspect), the TaskSatisfaction aspect provides and activates a questionnaire to collect subjective information related to the user satisfaction dimension. This form is very simple, allowing the user to select a subjective rating on a scale of five different values for each of the three questions that are displayed (Sauro and Kindlund, 2005). The aspect updates states of the task with the satisfaction values chosen by the user.

```
aspect TaskSatisfaction {
      pointcut satisfaction(Task t): execution(void Task.finalize(…)) and and this(t);
      before(Task t): satisfaction (t) {

      // display questionnaire
      // get data entered by the user
      // update satisfaction attributes of the task

      return;}
}
```

The TaskUserProfile aspect is activated at the beginning of the execution of a session and allows a user's profile data to be obtained. The aspect presents a form that collects user information and logs it. As in the case of events and errors, the developer must configure the chosen log file format, so TaskUserProfile is an abstract aspect and the method for this configuration is logProfile.

```
abstract aspect TaskUserProfile implements ActionListener{
      // user profile attributes
abstract void logProfile(…) {…} <- hot-spot
pointcut initUserSession(): *.main(…);

before(): initUserSession() {
      // display questionnaire
      // get data for user profile
      logProfile();}
}
```

Finally, the UsabilityTaskConfiguration aspect configures the tasks that will be evaluated in the particular session or execution of the desktop application. This operation is the first thing the developer must reimplement. UsabilityTaskConfiguration is an abstract aspect that provides a pointcut that executes at the beginning of an application's execution (a session). The advice invokes three methods that will be redefined in a concrete aspect. The first method allows configuration of the application that will be evaluated, the second method allows configuration of the tasks and the third method creates the log for these data.

```
abstract aspect UsabilityTaskConfiguration {
      String appName, appVersion, appTest;
      // settings attributes
      abstract void setApplicationTest(); <- hot-spot
      abstract void addTask(…); <- hot-spot
      abstract void logApp(…); <- hot-spot

pointcut configuration(): call (*.main(…));

before(): configuration(){
      this.setApplicationTest();
      this.addTask();
      this.logApp(…);
}
```

Aspects, patterns and hot-spots: The AJMU framework is composed of thirteen aspects and one class (Task). In the design and implementation of twelve of the aspects, patterns with specific purposes were used. In the design of the TaskConnect aspect, the abstract pointcut pattern was used in the startTask and endTask pointcuts as a strategy for managing the changes that these pieces of code require, since they will be different for each domain and each task to be evaluated. The startTask and endTask pointcuts are hot-spots that should be defined by the developer in specific concrete aspects that extend from TaskConnect; this allows advice reuse for instantiating and ending the task. In the design of the TaskConnect aspect, the template advice pattern was also used, to encapsulate the variable part of the advice that instantiates each specific task. The setIdTask abstract method identifies each task among the different assessed tasks and domains. The setIdTask method must be redefined by the developer by extending the aspect, which provides flexibility and setting another hot-spot. The abstract pointcut and template advice patterns are used in combination because the pieces of code they propose (two pointcuts and a method) must be specialized in the same concrete aspect.

For the TaskLog aspect, the template advice pattern was used, as in this case the definition of the pointcuts is known precisely but the implementation of the advices will be different, depending on the different log formats. So through several methods, reuse of the abstract pointcuts and flexibility for the particular implementation of the log are possible.

For the TaskEvent aspect, in conjunction with its extended aspects, TaskException, TaskDialog, TaskMessage, TaskHelp and TaskEscape, the composite pointcut pattern was used. Given the complexity of the joinpoints, several conditions are split into simpler pointcuts of the TaskEvent aspect, allowing them to subsequently be properly combined in the complete pointcut. The more specific aspects, such as TaskException, reuse these pointcuts to define the interception of exceptions with other primitives, as happens with TaskDialog. Another common strategy applied across the hierarchy was use of the template advice pattern, through which the logEvent abstract method connects with the chosen log format in an aspect that should be implemented by the developer and is thus a hot-spot.

In the design of the TaskMessage aspect, the pointcut method pattern was also used when incorporating the conditions evaluated in the advice, to determine whether activation of the dialog box corresponds to a message and thus whether the aspect execution should continue.

In the design of the TaskEscape aspect, the abstract pointcut pattern was used for the notComplete pointcut (which is used in combination with the complete pointcut). However, this should be redefined by the developer with an aspect extension and this is therefore another hot-spot for defining the variation in what each application and/or task can submit. The complete pointcut at this level is concrete, but its implementation is partial. The situation for the TaskHelp aspect is analogous to that of the TaskEscape aspect and therefore it applies the same patterns.

Finally, for the UsabilityTaskConfiguration aspect, the template advice pattern was used to provide flexibility in specifying the application and tasks to evaluate. The setApplicationTest method extension allows the application to be configured. The addTask method extension allows the task to be configured. The logApp method connects this aspect with the chosen log file format (same situation as the logEvent method of the event-error hierarchy). These methods should be implemented by the developer in sub-aspects, with each set as a hot-spot.

Table 2 summarizes the aspects of AJMU, the patterns that were used, the purpose of their application and the hot-spots.

Table 2: Aspects and patterns used in AJMU

Aspects	Types	Patterns	Purpose	Hot-spot
Task connect	Abstract	Abstract pointcut	Manage the weak composition among aspects and base code, reuse advice	StartTask and endTask pointcuts
		Template advice	Flexible aspect design	setIdTask method
TaskLog	Abstract	Template advice	Flexible aspect design	NO
TaskLogText	Concrete			
TaskLogXML				
TaskMessage	Abstract	Pointcut method	Flexible aspect design	logEvent method
TaskEvent		Composite pointcut	Decompose and recompose complex conditions; Reuse parts of expressions of pointcuts	
TaskException		Template advice		
TaskDialog				
TaskHelp		Abstract pointcut	Partial implementation of pointcuts	notComplete pointcut
TaskEscape				
TaskSatisfaction	Concrete	None		
TaskUserProfile	Abstract	Template Advice	Flexible aspect design	logProfile method
UsabilityTask	Abstract	Template Advice	Flexible aspect design	addTask method
Configuration				setApplicationTest method logApp method

Table 3: Applications used in experiments

Applications	Descriptions	Classes	Methods	Fields	LOC
JMoney	Personal finance manager (http://sourceforge.net/projects/jmoney/files/JMoney/)	83	594	436.00	5.780
Freemind	Tool for creating mind maps (http://sourceforge.net/projects/freemind)	820	6974	2.816	108.378

LOC: Lines of code

Table 4: Set of tests

		Freemind		JMoney	
Sets	Test	Log = XML	Log = txt	Log = XML	Log = txt
Set 1	Task 1	T1	T4	T1	T4
	Task 2	T2	T5	T2	T5
	Task 3	T3	T6	T3	T6
Set 2	Task 1-Task 2	T7	T10	T7	T10
	Task 1-Task 3	T8	T 11	T8	T 11
	Task 2-Task 3	T9	T12	T9	T12
Set 3	Task 1-Task 2-Task 3	T13	T14	T13	T14

Experiments: We conducted various experiments to evaluate the development of usability testing with the AJMU framework. Two real desktop applications whose characteristics are presented in Table 3 were chosen for this purpose.

For each application, three tasks, each composed of three to seven actions, were defined. The appendix describes the tasks. These tasks were used to create three test sets that included all possible tests. In total, 28 tests were performed. Two developers coded the aspects; these were split per application. The developed test sets are presented in Table 4.

Table 5 shows the numbers of aspects and hot-spots that were specialized for each test. Set 1, which includes a test of only one task, required specialization of 8 aspects that included 14 hot-spots. Set 2, which includes tests of two tasks, required specialization of 15 aspects that included 25 hot-spots. Finally, Set 3, whose single test includes three tasks, required specialization of 22 aspects and 36 hot-spots.

RESULTS

The number of aspects needing specialization is directly proportional to the number of tasks to assess; indeed, it is possible to establish the relationship: number of

Table 5: Aspects and hot-spots required per test

Code element and applications	Set 1						Set 2						Set 3	
	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14
Aspects														
Freemind	8	8	8	8	8	8	15	15	15	15	15	15	22	22
JMoney	8	8	8	8	8	8	15	15	15	15	15	15	22	22
Hot-spots														
Freemind	14	14	14	14	14	14	25	25	25	25	25	25	36	36
JMoney	14	14	14	14	14	14	25	25	25	25	25	25	36	36

aspects = (number of tasks * 7)+1. Similarly, the number of hot-spots needing specialization is also directly proportional to the number of tasks to assess: Number of hot-spots = (number of tasks * 11)+3. These relationships can be critical, considering that in the same session the assessment of a significant number of tasks (e.g., 10 or more) may be required. Clearly, an undesirable and unmanageable proliferation of aspects will occur. In our experiments, the aspects and their hot-spots associated with each task were coded once, with different log formats. To use a new test configuration, only one aspect needed to be coded (UsabilityTaskConfiguration). Most specialized aspects require minimal code (one pointcut or method), which is simple enough. Given that the framework allows the evaluation of 13 metrics, this effort does not appear to be overly complex.

The number of aspects and hot-spots to be specialized can be reduced if the choice between different log formats is removed. But this results in a loss of flexibility. The number of aspects and hot-spots that must be specialized when there are many tasks could be a weakness in our proposal; thus, future work should propose alternatives for improvement. Modularity and flexibility are desirable features, but they have an associated cost that must be reduced.

DISCUSSION

As mentioned above, some studies have applied AOP to evaluate the usability of WIMP applications. The main limitation of these studies is that they do not distinguish between aspects and tasks. Aspects are simply monitors of events that record data in a log. A major intervention of developers is required in this case and the aspects do not provide much reusability and flexibility.

Tarta and Moldovan (2006) propose a design with aspects that supports automatic usability evaluations in desktop applications. They argue that through a hierarchy of aspects, pointcuts can be reused to share the same entry and exit points (joinpoints). The derived aspects define new pointcuts for error handling, completion of tasks, screenshots and data acquisition through questionnaires. Concrete aspects (advices) perform the data logging. These aspects code tracking and logging in the same advice, but they must be executed repeatedly since there are aspects for errors, calculation time and so on. Thus, developers must configure many aspects, the notion of a task gets lost and the whole structure is specific to each application in which it is used. The authors present simple diagrams and sample code in AspectJ.

Shekh and Tyerman (2010) develop a framework for evaluating usability with AOP. The authors record UI events such as mouse events. No details are provided regarding the characteristics of the framework or the design of the aspects, but the authors state that they have followed Tarta and Moldovan (2006) recommendations. The framework is developed in AspectJ. The authors present the results of controlled experiments performed in a laboratory.

Holzinger *et al*. (2011) propose implementing aspects to track some interface events (keyboard input, menu actions and drag and drop actions). They propose using Objective-C, adding aspects

to the object hierarchy through the technique of "method swizzling" and extending classes. The design focuses on an aspect that performs the logging centrally and three aspects are necessary to identify the targeted interface events.

Tao (2008) uses AOP to automatically capture user interface events in applications with Model-View-Controller (MVC) architectures. The author proposes a hierarchy of aspects that reuse a single method that constructs a report (time/date and event). The pointcuts and advices need to be redefined for each case and application, e.g., observer updating, notifiers and event handlers and dialog boxes. The author presents sample AspectJ code and a simple case study.

Tao (2012) presents a scheme that is very similar to previous work. The approach is based on AO techniques that run traces of GUI events and collect contextual information about WIMP applications. Tao proposes a hierarchy of aspects, which is basically a method containing the logic that enables the report. The concrete aspects define the events to intercept in the pointcuts and the advices invoke the method that executes the report. The author presents simple example code in AspectJ.

Bateman *et al*. (2009) present an approach called "Interactive Usability Instrumentation" (IUI). Usability evaluators specify which actions will be registered (logged) when a user interacts with the interface elements of the object of evaluation in the application, thus eliminating the need for additional support programming. This initial activity is based on AOP and allows direct interaction with the application to be instrumented to decide which (interface) elements of a system are related to particular tasks and usability issues.

Humayoun *et al*. (2009) present a tool called "UEMan" to manage and automate usability evaluation activities based on a User-Centered Design (UCD) approach during the software development process. The model and the tool incorporate the concept of a user task and facilitate the implementation of various types of experiments: Heuristic evaluation experiments, task-type experiments, experiments based on questionnaires and dynamic experiments that employ logging implemented with aspects. The tool uses very basic aspects that perform simple actions such as timing the duration of an activity (from an entry point to an exit point, defined by two pointcuts) and counting the number of mouse clicks and key strokes occurring in that time interval.

Our analysis of these approaches is that the concept/entity "task" is missing, both in the domain to be evaluated and in the frameworks. This is because the modules (aspects) that are designed and implemented to automate usability testing are restricted to the existing concepts (entities) in the domain, namely, methods/attributes and classes and this loses the conception of a task. Moreover, none of the approaches described above explicitly applies to AO patterns.

The strategy used in AJMU was to place the user task at the center of the design and implementation of the framework, as both a concept and an entity. We use AO patterns to aspects connect the task with the domain and the required services for evaluation, so the user task can be easily identified and analyzed.

The proliferation of aspects discovered in the experiments can be avoided if an XML configuration file is used to define the log format and other data required by AJMU instantiation, such as the set of tasks to evaluate, with their init and end points (joinpoints). Use of an XML file is an alternative solution (for managing the proliferation) that does not require changing the framework design and thus losing flexibility.

CONCLUSION

AJMU is a domain framework that supports usability evaluations of user tasks in WIMP applications. From the point of view of the usability expert, AJMU enables evaluation of the three typical usability dimensions: effectiveness, efficiency and satisfaction, with mechanisms for collecting and recording information and automatically calculating more than 10 metrics and contextual data associated with the tasks defined in the tests. AJMU facilitates analysis of the usability of a user task, identifying the most complex operations for the user, making comparisons between different users, comparing results between different tasks and identifying critical common factors in user interactions, among other possibilities. AJMU generates information relevant to the assessment of usability with a higher level of abstraction and semantic value than other frameworks.

From the point of view of the developer who uses AJMU for different applications, AJMU is a noninvasive framework for those applications on which execution tests are to be performed. AJMU is easily configurable and requires only the extension of aspects, which minimizes the effort required for the developer to learn how to configure and specialize the framework. The use of AO patterns is transparent at this level.

From the point of view of the construction of AJMU, the design and implementation of the modularization mainly employ aspects that are built using a set of AO patterns that individually and in combination enabled the construction of this framework from scratch. The AO patterns guided this construction and served as guidelines for improving the reuse of different pieces of code (pointcuts and advices), managing the variability presented by different domains and tasks and providing flexibility in AJMU. The use of AO patterns in the design of AJMU will also facilitate its evolution, since the incorporation of new log formats, events and metrics should follow a clear scheme and a modularized design.

APPENDIX

JMoney

Task 1: Create a new account:

- Click the right side panel and select the “new account” option
- Enter the account properties from the account properties panel
- Enter entries in the account entries panel
- Save using the “save” button on the toolbar or on the File/Save menu, using the shortcut “ctrl + s” or answering “yes” to the prompt to save changes before closing the application

Task 2: Add entries to an existing account:

- Click the “Entries” tab of the workspace where tickets and properties of the account are shown
- Click the “New” button and enter data for the fields Check, Data Value, Category Memo, Debit, Credit, Balance, etc.
- Save using the “save” button on the toolbar or on the File / Save menu, using the shortcut “ctrl + s” or answering “yes” to the prompt to save changes before closing the application

Task 3: Generate report balances for existing accounts:

- Click on the “Account Balance” button on the lateral panel
- Apply filters to the report, choosing from: All Entries/Cleared Entries/Date
- Click the “Generate” button on the right panel

Freemind

Task 1: Create a basic mental map:

- Create a new map by clicking on the “New” button on the toolbar or on the File menu
- Complete text of the map’s root node is created
- Build a three-level hierarchy with son and brother nodes (at least 11). For this, select the node for which a child or sibling node is

to be created and the new node can be inserted using the Context menu options or the Insert menu
- Using the Tools menu, sort the nodes by name
- Save the mind map on the desktop with a meaningful name

Task 2: Create an encryption mind map:

- Create a map encryption using "Create encrypted map" on the main menu
- Enter a password to encrypt the map
- After creating the map, add 11 nodes organized in a hierarchy of three levels that contains at least five encrypted nodes. The nodes can be inserted using the Context menu options or the Insert menu
- Save the mind map on the desktop with a meaningful name

Task 3: Open an existing map and edit it:

- Open the mental map provided for the task
- Remove two nodes
- Add at least three child nodes and three brother nodes
- Prioritize the nodes
- Add an image to the root node
- Change the text format for the root node
- Save the mind map on the desktop with a meaningful name

REFERENCES

Adair, D., 1995. Building object-oriented frameworks. AIXpert, February and May 1995.

Bateman, S., C. Gutwin, N. Osgood and G. McCalla, 2009. Interactive usability instrumentation. Proceedings of the 1st ACM SIGCHI Symposium on Engineering Interactive Computing Systems, July 15-17, 2009, Pittsburgh, PA., USA., pp: 45-54.

Bynens, M., B. Lagaisse, W. Joosen and E. Truyen, 2007. The elementary pointcut pattern. Proceedings of the 2nd Workshop on Best Practices in Applying Aspect-Oriented Software Development, March 12-16, 2007, Vancouver, BC.

Griswold, W.G., M. Shonle, K. Sullivan, Y. Song, N. Tewari, Y. Cai and H. Rajan, 2006. Modular software design with crosscutting interfaces. IEEE Software, 23: 51-60.

Hanenberg, S. and A. Schmidmeier, 2003a. AspectJ idioms for aspect-oriented software construction. Proceedings of the 8th European Conference on Pattern Languages of Programms, June 25-29, 2003, Irsee, Germany, pp: 617-644.

Hanenberg, S. and A. Schmidmeier, 2003b. Idioms for building software frameworks in AspectJ. Proceedings of the Workshop on Aspects, Components and Patterns for Infrastructure Software, March 17-21, 2003, Boston, Massachusetts, USA., pp: 55-60.

Hanenberg, S. and R. Unland, 2001. Using and reusing aspects in AspectJ. Proceedings of the Workshop on Advanced Separation of Concerns, October 14-18, 2001, Tampa Bay, USA.

Holzinger, A., M. Brugger and W. Slany, 2011. Applying aspect oriented programming in usability engineering processes-on the example of tracking usage information for remote usability testing. Proceedings of the International Conference on E-Business, {ICE-B} is part of {ICETE}-the International Joint Conference on E-Business and Telecommunications, July 18-21, 2011, Seville, Spain, pp: 53-56.

Humayoun, S.R., Y. Dubinsky and T. Catarci, 2009. UEMan: A tool to manage user evaluation in development environments. Proceedings of the 31st International Conference on Software Engineering, May 16-24, 2009, Vancouver, Canada, pp: 551-554.

ISO 9241-11, 1998. Ergonomic requirements for office work with visual display terminals (VDTs). Part 11: Guidance on Usability. International Organization for Standardization, Geneva, Switzerland.

Ivory, M.Y. and M.A. Hearst, 2001. The state of the art in automating usability evaluation of user interfaces. ACM. Comput. Surv., 33: 470-516.

Kiczales, G., E. Hilsdale, J. Hugunin, M. Kersten, J. Palm and W.G. Griswold, 2001. An overview of AspectJ. Proceedings of the 15th European Conference on Object-Oriented Programming, June 18-22, 2001, Budapest, Hungary, pp: 327-353.

Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier and J. Irwin, 1997. Aspect-oriented programming. Proceedings of the 11th European Conference on Object-Oriented Programming, June 9-13, 1997, Jyvaskyla, Finland, pp: 220-242.

Kulesza, U., V. Alves, A. Garcia, C. de Lucena and P. Borba, 2006. Improving Extensibility of Object-Oriented Frameworks with Aspect-Oriented Programming. In: Reuse of Off-the-Shelf Components, Morisio, M. (Eds.). LNCS., 4039, Springer-Verlag, Berlin, Heidelberg, ISBN: 978-3-540-34606-7, pp: 231-245.

Laddad, R., 2002. AspectJ in Action: Practical Aspect-Oriented Programming. Manning Publications, Greenwich, CT., USA.

Lagaisse, B. and W. Joosen, 2006. Decomposition into elementary pointcuts: A design principle for improved aspect reusability. Proceedings of the Workshop on Software Engineering Properties of Languages and Aspect Technologies, March 20-24, 2006, Germany.

Miles, R., 2004. AspectJ Cookbook. O'Reilly Media, New York, USA.

Nielsen, J., 1992. The usability engineering life cycle. Computer, 25: 12-22.

Noble, J., A. Schmiedmeier, D.J. Pearce and A.P. Black, 2007. Patterns of aspect-oriented design. Proceedings of the 12th European Conference on Pattern Languages of Programs, July 4-8, 2007, Irsee, Germany, pp: 769-796.

Pree, W., 1995. Hot-spot-driven framework development. Summer School on Reusable Architectures in Object-Oriented software Development, pp: 123-127.

Santos, A.L., A. Lopes and K. Koskimies, 2007. Framework specialization aspects. Proceedings of the 6th International Conference on Aspect-Oriented Software Development, March 12-16, 2007, Vancouver, British, Columbia, Canada, ACM, pp: 14-24.

Sauro, J. and E. Kindlund, 2005. A method to standardize usability metrics into a single score. Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, April 2-7, 2005, Portland, OR., USA., pp: 401-409.

Shekh, S. and S. Tyerman, 2010. An Aspect-Oriented Framework for Event Capture and Usability Evaluation. In: Evaluation of Novel Approaches to Software Engineering, Maciaszek, L.A., C. Gonzalez-Perez and S.J. Heidelberg (Eds.). Springer, New York, USA., pp: 107-119.

Tao, Y., 2008. Automated data collection for usability evaluation in early stages of application development. Proceedings of the 7th WSEAS International Conference on Applied Computer and Applied Computational Science, April 6-8, 2008, Hangzhou, China, pp: 135-140.

Tao, Y., 2012. Aspect-oriented instrumentation for capturing task-based event traces. Int. J. Control Syst. Instrum., 3: 32-35.

Tarta, A.M. and G.S. Moldovan, 2006. Automatic usability evaluation using AOP. Proceedings of the IEEE International Conference on Automation, Quality and Testing, Robotics, Volume 2, May 25-28, 2006, Romania, pp: 84-89.

Vaira, Z. and A. Caplinskas, 2011. Application of pure aspect-oriented design patterns in the development of AO frameworks: A case study. Informacijos Mokslai, 6: 146-155.