Engineering Adaptive User Interfaces using Monitoring-Oriented Programming

Aaron John Buhagiar, Gordon J. Pace
Department of Computer Science
University of Malta
Malta
aaron.buhagiar.13@um.edu.mt, gordon.pace@um.edu.mt

Jean-Paul Ebejer
Centre for Molecular Medicine and Biobanking
University of Malta
Malta
jean.p.ebejer@um.edu.mt

***Abstract*—User interfaces which adapt based on usage patterns, for example based on frequency of use of certain features, have been proposed as a means of limiting the complexity of the user interface without specialising it unnecessarily to particular user profiles. However, from a software engineering perspective, adaptive user interfaces pose a challenge in code structuring, and separation of the different layers of user interface and application state and logic can introduce interdependencies which make software development and maintenance more challenging. In this paper we explore the use of monitoring-oriented programming to add adaptive features to user interfaces, an approach which has been touted as a means of separating certain layers of logic from the main system. We evaluate the approach both using standard software engineering measures and also through a user acceptance experiment — by having a number of developers use the proposed approach to add adaptation logic to an existing application.**

***Keywords*-Runtime monitoring, Monitoring-oriented programming, Software engineering, User interfaces.**

I. Introduction

The user interface is an integral part of the application as it is the means by which the user can communicate to the program, with discrepancies between its specification and implementation possibly resulting in inaccessible parts of the system, or unexpected behaviour. Not only is the user interface responsible for the user experience, but also to access to potentially critical parts of the underlying systems, thus requiring a high degree of reliability.

From the user experience perspective, in practice this is typically addressed through the application of certain fundamental principles in order to ensure that they are user friendly [1]. This becomes increasingly difficult as application logic increases in complexity, and it has been observed that the back end design of the system is equally important when discussing user interfaces [2]. The potentially cross-cutting nature of user interfaces has led to various approaches being proposed in the literature and adopted in practice, with the model-view-controller pattern being the most widely used [3]. However, as applications try to cater for different types of users, or for users using the application in different contexts, one direction which has been explored has been that of *adaptive user interfaces* [4] — the adaptation of a user interface based on usage patterns, for example, certain features might be relegated deeper in a menu if infrequently used, or a touch keypad interface might change slightly the layout of the buttons (increase in size, distance between buttons) depending on the type of errors the user typically makes when inputting values. Since the primary aim of using adaptation is to improve the user experience, ensuring that once adaptation is performed, it does not negatively affect the usability of the interface is a major concern. Apart from this, identifying what should be adapted, when it should be adapted and under what circumstances this should happen add to the challenges of performing adaptation.

From a software engineering perspective, adaptive user interfaces pose a challenge in code structuring, especially when a static user interface is changed into an adaptive one, in order to ensure that code remains properly structured and any changes remain local. Separation of the different layers of application state and logic — the core system, the user interface (actually a whole family of user interfaces) and the adaptation management unit — can typically introduce interdependencies which make software development and maintenance more challenging. Monitoring-oriented programming [5] has been touted as a software engineering approach in which monitoring logic is automatically injected into the system from specifications which are independently specified thus ensuring separation-of-concerns between the underlying system and the augmented logic. Although the approach has been primarily used as a means of achieving reliable software by adding dynamic checks on the system's behaviour, it has also been used as a means of adding functionality to a system through by having orthogonal aspects of the system programmed separately, and triggering them through the use of a runtime monitoring tool thus introducing less direct dependency between these aspects.

In this paper we explore the viability of developing adaptive user interfaces using a monitoring-oriented programming based design and evaluate whether one achieves separation-of-concerns. By comparing adding adaptive features to an application's UI directly through the source code, to adding the features using a specification written for a runtime monitoring tool, we assess the viability and usability of the approaches using both standard software complexity metrics and user evaluations through interviews. The results indicate

that the monitoring-oriented approach is more effective, even when performed by developers with no prior experience of runtime monitoring tools.

II. Monitoring-Driven Adaptive User Interfaces

Whilst there exist various types of adaptation, in this paper, we focus on adaptations that manipulate the user interface (UI) based on usage patterns — UI changes triggered on how the user is using the application, and intended to improve the user experience. Such changes range from ones such as hiding buttons whose short cut is predominantly used by the user, to more sophisticated ones which, for example, modify the spacing and sizes of buttons based on the number of key input errors which the user makes.

It has been shown [4], that if appropriately used, adaptive user interfaces may contribute to an increase in the application usability and a better user experience. However, such adaptivity should also be applied with caution, the effect on usability varies based on the user, the conditions the application is being used in and the application itself [6]. One of the main challenges of adaptive UIs is that adaptation varies from user to user, such that not all users prefer the same amount of adaptivity [4]. Apart from this, how the user interface is visually changed is also important to determining the effectiveness of the adaptive UI [7]. In a study by Gajos *et. al* different types of visual adaptation had drastically different results, with the best method improving the usability of the interface and the worst method lead to confusing the users.

These studies, in conjunction with the fact that no clear generally applicable guidelines exist (or perhaps can exist) to guide the design of adaptive UIs, indicate that the design and engineering of adaptability is largely an experimentally-driven endeavour, to see what helps and what hinders users. This implies that a methodology to support ease of adoption and modification in the design of adaptive features of the UI from the system designers' and developers' perspective is imperative. However, when adaptive features are hard coded, issues of extensibility, flexibility and reusability can arise [8]. This is particularly the case since the adaptive features heavily depend on the system's dynamic behaviour.

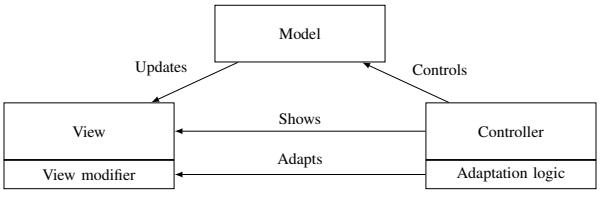

Figure 1: Architecture for adaptive user interface using directly instrumented directives

Much of modern UI design is based on the Model-View-Controller (MVC) approach or variants of it [3]. MVC separates what is shown in the UI (the view), the backend model of the data (the model) and the logic of how interaction with the UI influences this data (the controller). The addition of adaptive features of the UI have typically been handled using an extension of this approach, as shown in Figure 1. The controller is enriched with an adaptation subcomponent, which keeps track of how the user is interacting with the UI, and decides how the view is to be changed. This is sent to a subcomponent of the view, which handles modification of the UI. Using this approach has a number of drawbacks. Firstly, the adaptation and UI modification logic, which are closely interrelated, are integrated in two separate components of the architecture. Furthermore, the adaptation logic typically consists of two parts: (i) one part which keeps track of the history of user interaction with the UI; and (ii) another which finds patterns in this history in order to trigger modification directives. This standard structure of the adaptation logic subcomponent indicates that it would be better to delegate it to a separate component, thus separating concerns, and gives the opportunity to use standard techniques and tools to perform this in an efficient manner without local optimisations.

One approach which has been used in the literature to add features to a system which depend on runtime behaviour is monitoring-oriented programming [5]. This approach is based on the idea behind runtime monitoring — the use of monitors which observe events that occur within the application at runtime, usually bound to method calls, and react accordingly [9]. Much work in the area of runtime monitoring has focussed on addressing the issue of *separation-of-concerns* — how to build runtime monitoring tools which allow separation between the code of the application and the monitoring code itself. Internally, runtime monitoring tools use technologies such as aspect oriented programming [10] to add the new logic into the underlying system.

In order to specify the behaviour of the monitors, runtime monitoring tools typically use logics (such as LTL) or automata to specify system behaviour which should trigger the additional monitoring logic. Typically, upon the system's dynamic (runtime) behaviour matching a specification, the runtime monitoring system may choose to trigger a reaction which may affect the underlying system under scrutiny, depending on the intended use of the triggered code. Most commonly, monitoring systems are used to ensure system correctness, in the form of runtime verification (where the monitor is designed to match upon failure of the system, and the reaction would typically report or try to reduce the impact of the failure), or runtime enforcement (where the monitor attempts to realise that the system may fail, and the reaction is meant to redirect it in order to ensure that the failure does not happen). However, monitoring-oriented programming [5] has been put forward as an alternative use of such reactions, intended to trigger new or modified functionality to the system at certain points during its execution.

We propose an architecture based on monitoring-oriented programming to support a more structured approach to

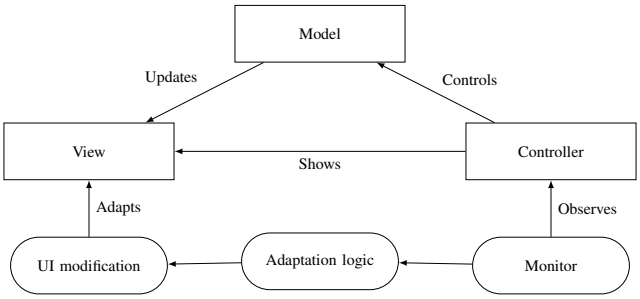

Figure 2: Architecture for adaptive user interface using using a monitoring-oriented approach.

programming adaptive UIs, as shown in Figure 2. The key to the proposed approach is that the adaptation layer (shown below the MVC in the diagram) is programmed independently of the rest of the system, and is integrated using a runtime monitoring tool. In fact, the three adaptation components correspond to the parts derived by the runtime monitoring tool, with the leftmost *Monitor* component being the event capturing part of such tools, the *Adaptation logic* corresponding to patterns specified using an appropriate logic and code to match which is automatically derived by the monitoring tool, and finally the *UI modification* component corresponding to the reaction code which is adding functionality to the UI view. The separation of concerns between the main UI system and the adaptation part of the logic gives advantages especially when adding a new adaptation layer, or when testing or modifying existing ones. It is worth also noting that given the power provided by most runtime monitoring tools, if required, one can also capture events pertaining to parts of the underlying system other than the controller without manual instrumentation of code in that part of the system.

The questions we set out to answer in the rest of the paper are (i) whether users find the use of monitoring techniques helpful or not when programming adaptability into a UI; and (ii) how a monitoring-oriented implementation of an adaptive UI compares to direct manual instrumentation in terms of standard software engineering metrics. The results are discussed in sections IV and V.

III. Instantiating the Framework in Larva

In order to evaluate how the adoption of such an architecture performs, we have used the runtime monitoring and verification tool Larva [11] to extend existing applications with fixed UIs in order to make adopt adaptive UIs.

Larva is a runtime monitoring and verification tool for Java systems. It captures events happening on the system under scrutiny (primarily events correspond to method invocations and exiting, although it also provides support for events related to exceptions). In order to identify relevant sequences of events, Larva uses an automaton-based specification language, *Dynamic Automata with Timers and Events* (DATEs), with a specification effectively being a set of communicating automata extended with timers and dynamic automaton replication[1]. Transitions in these automata are tagged by a triple: $e \mid c \mapsto a$ where (i) e is the event (which can be a system-side event such as a method call, or an event resulting from a Larva timer); (ii) c is a boolean condition which is checked whenever the event fires and enables the transition to be followed if and only if the condition is satisfied; and (iii) a is an action — a piece of code which is executed upon taking the transition (after the condition is evaluated to true). In the case of DATEs both the condition and the action are optional. If no condition its stated then it is taken to be true thus, the action is performed upon the event occurring. The transition is followed normally but no actions occurs if a is empty. Full details with the formal semantics of DATEs and their monitoring in Larva can be found in [11].

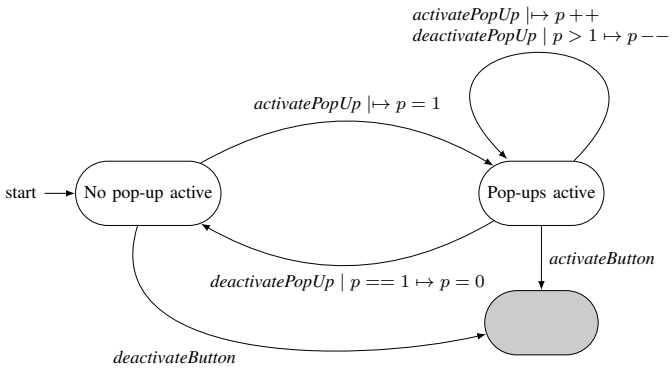

Figure 3: Verifying button activation and deactivation in Larva

Larva has been mostly used for runtime verification, allowing the identification of so-called *bad states* which will only be reached if the system runs afoul of its specification. In the context of UIs, consider the DATE shown in Figure 3, which verifies that deactivation of buttons on the main window can only happen if at least one popup window is open, while activation of such buttons can only occur if no popup windows are open. The DATE[2] keeps count of the number of popup windows opened in a variable p, and uses two states to encode whether at least one popup window is open. A third bad state (shaded in grey) is used to capture when unexpected activation or deactivation occurs.

In our context, such bad states will not be used, but it is straightforward to see how such an example can be modified to use monitoring-oriented programming such that, starting from a system which does not enable or disable buttons, does so through triggers from the monitor. Figure 4 shows how

[1]Note that Larva also supports various other temporal logics to be used as input by providing translators into DATEs.

[2]For convenience of presentation, multiple transitions with the same source and destination state are drawn using one arrow but with the different event, condition, action triples decorating the transition on separate lines.

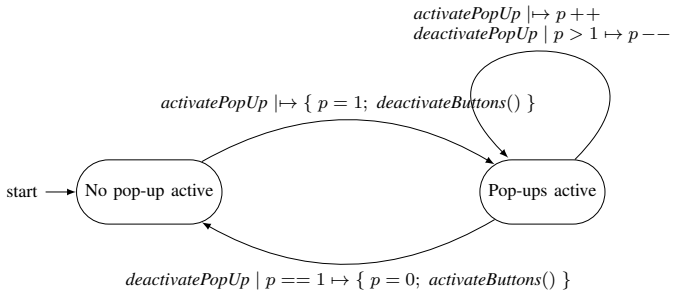

Figure 4: Verifying button activation and deactivation in Larva

Figure 5: Five button interface frequently used in medical infusion pumps

such an approach can be implemented using a Larva DATE.

To illustrate how DATEs can be used to instrument an adaptive UI using monitoring-oriented programming, consider a 5-key number entry UI as used in medical infusion pumps as shown in 5. This interface allows input of a five digit number shown in the display, and consists of (i) two horizontal direction buttons (◄ and ►) used to control the position cursor; (ii) two vertical direction buttons (▲ and ▼) used to set the digit at the current position of the cursor; and (iii) an OK button to accept the number currently displayed. Although on devices such as medical infusion pumps, these interfaces are physical ones, software emulating such input devices are increasingly being used to allow control from touchscreen device. Using such a context, we can explore the use of monitoring-oriented programming to adapt the UI view. Some simple adaptive features such as enlarging the commonly used buttons, adapting spacing between buttons based on frequency of errors and enabling or disabling wraparound of cursor can be easily added to an application.

Consider the DATE shown in Figure 6, which allows for automatic switching between cursor wraparound and strongly bounded display ends — the former allows the cursor to be moved from the leftmost to the rightmost position and vice-versa, while the latter ignores further left and right button presses when the cursor is at the left and right extremes of the display. The logic implemented observes the user and changes cursor movement mode accordingly: (i) if the system is in non-wraparound mode but repeatedly tries to use wraparound behaviour, then this behaviour is enabled; while (ii) if the system is in wraparound mode, but the user barely uses it over a one hour period, it is switched off accordingly. Note that $t@\delta$ is the event triggered when a timer t reaches time δ, while *t.reset()* resets timer t to zero. To avoid cluttered diagrams, $*$ denote any system event which does not match with any other outgoing transition.

Obviously, more sophisticated decision procedures to toggle between wraparound and non-wraparound behaviour can be implemented in a similar manner. Typically, complexity of these automata is contained by using multiple DATEs, some of which are used to compute statistics about user behaviour, and others which use these statistics to seek behavioural patterns and change between modes as required.

The use of DATEs can further be expanded to observe when mistakes are performed by the user whilst using a user interface. Take for example the same UI shown in Figure 5, mistakes occur when two opposite horizontal arrow buttons are pressed after in quick succession. For example if ◄ is pressed and is immediately followed by a ► then this is counted as a mistake, as the second button press was used to effectively undo the first. Upon logging 10 mistakes, the spacing between the buttons on the UI is increased so that it is less likely for the user to press the incorrect button. The UI adaptation can be implemented as two DATEs running concurrently as shown in Figure 7. The DATEs cater for the horizontal arrow buttons ◄ and ►, but the pattern can be similarly applied to ▲ and ▼.

If the adaptation logic were to be added directly to the MVC model of the UI implementation, the result is code combines both the UI and adaptation controls, which hinders maintenance and debugging. In order to avoid these interdependencies, the only solution is to include event generators in the controller code, and implement the adaptation logic as an event stream consumer — effectively replicating the structure provided by the monitoring-oriented programming approach in a manual manner.

IV. User Evaluation

The first question we set out to answer is whether users find the use of monitoring techniques helpful or not when programming adaptability into a UI. To investigate usability, an experiment was set up, in which participants were required to implement adaptive features into an application using both (i) a monitoring approach using Larva; and (ii) manually coding the adaptation in the main code. It is worth emphasising at this stage that we are evaluating the use of Larva to support adaptive UI development as part of an already existing Java system. The results are inevitably biased to the specification language used by Larva, and if we were to use another tool or formalism (e.g. using regular expressions or LTL) might, give different results.

Experimental setup: Before running the experiment, relevant information about the participants was gathered — their experience and knowledge of the technologies used (Java and Larva), programming experience and level of ICT education. They were given 30 minutes to familiarise themselves with the technologies involved through videos (i) detailing the

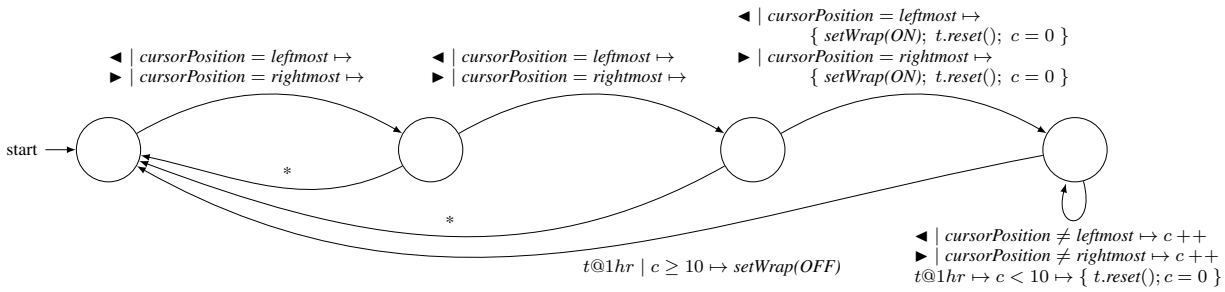

Figure 6: Switching wraparound mode on and off based on usage.

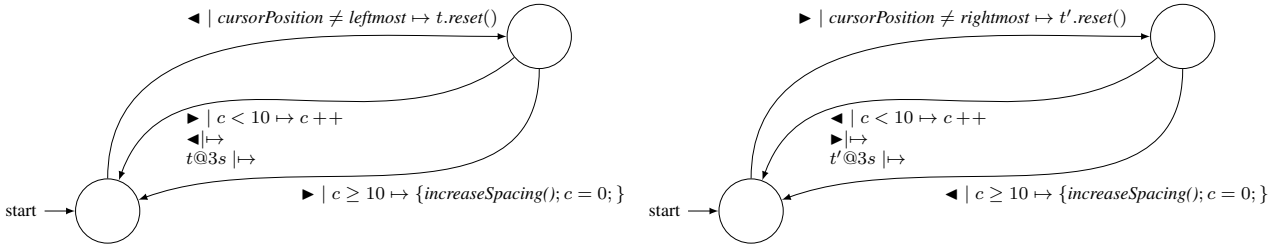

Figure 7: Adapting button spacing based on usage.

architecture and implementation of the application that they were to adapt, and the adaptive feature which they were to add; and (ii) a Larva tutorial. They were also provided with a printed short Larva tutorial and documentation for the code provided. Although all participants performed both tasks, half of them started with the manual approach, while the other half started with the monitor oriented task. All participants performed the tasks under supervision in order to enable reliable measurement of timing and ensuring a uniform setup of the development environment.

The Task: The application used in the experiment was an implementation of the five digit interface using a keypad. The adaptive feature which they were to implement was a hypothetical situation in which, if users repeatedly mispressed numeric keys (which would be displayed on a small screen), then increasing the distance between the buttons will help the user[3]. Concretely, the logic that the users were to implement was that if, in a 5 second period, the user presses backspace 15 or more times with less than 1 second between presses, then the distance between buttons on the UI is increased. The participants performed the task using Eclipse IDE with the Larva plug-in installed which gives basic syntax support, and allows for direct compilation of Larva scripts.

Measurements: During the experiment, the participants were timed[4] in order to see how long they took to complete each task. In this interview the participants were asked to comment about which approach they preferred and found easier to complete and the main problems encountered. They were also asked to comment about which approach they would have adopted if they were to apply more complex adaptations to larger systems. Apart from this, their code was analysed using standard software engineering metrics: (i) development time taken to complete each exercise; (ii) lines of code (Java or Larva) added to the source files and lines of code added by the participants; (iii) *Average Cyclomatic Complexity* (ACC) and *Weighted Methods per Class* (WMC) are also measured. WMC gives the maximum amount of possible execution paths the application may take while ACC measures the average complexity of the applications methods [12]. These metrics were measured using Google's CodePro AnalytiX plug-in for Eclipse IDE.

11 participants took part in the experiment, all having knowledge and experience of programming. All the participants were either IT students or professionals, with experience ranging from 2 to 10 years in IT. All the participants were experienced Java programmers, but only 3 had prior experience using Larva. In an ideal setup, the users were equally experienced in both technologies, but this was impossible due to the fact that Larva is not very widespread. Although the number of participants for such a study is not high, it is worth noting that participants were required to be programmers, and had to give a substantial investment in time for the experiment (not only implementing the two tasks, but also reading and viewing the support material, *etc.*). These requirements reduced our sample size significantly.

[3]Although the benefits of such an adaptive UI are arguable, it is not this feature that we are evaluating, but rather the difficulty of implementing it.

[4]This measure does not include the time it took the participants to familiarise themselves with the application or Larva. The timings measure how the long the participants took to design and code the adaptive feature.

Outcomes from interviews: The results from the post-experiment interviews are given in the table below. The majority (7 of 11) preferred using Java to perform the task, citing their knowledge of the language as the primary reason. However many added that with some experience in Larva they would have opted to choose it as their preferred approach.

	Larva	Java	Neither
Which task was found easier	4	7	0
Most problems encountered	7	2	2
Which is more understandable	8	3	0
Which to adopt in larger system	10	1	0

The most predominant problem the participants found when they were implementing adaptability was the fact that they were unfamiliar with the application's code and that this affected the first task they attempted. However, 7 of 11 participants noted that they had problems with the monitoring-oriented approach due to their inexperience in Larva.

In contrast, when questioned about understandability, 8 out of the 11 participants preferred the Larva approach, citing (i) the separation of concerns — the source code from the code of the application; and (ii) the use of automata to control adaptation. The other 3 cited prior knowledge of the language used (Java as opposed to Larva) as the reason for identifying the manual approach as the more understandable one.

When questioned as to which approach they would adopt if implementing more complex adaptive features into a larger application, 10 of the 11 participants opted for the monitoring approach. The reasons mentioned were: (i) the separation between the adaptation code and the application; (ii) that an automata-based language is less complex to describe such adaptations in; and (iii) the fact that the new features were implemented with little to no modification to the original code, which would also help since any tests on the original application can be left untouched.

Metrics applied to the code produced: The quantitative results from the analysis of the code produced by the experiment subjects are given in the table below. The time taken by each participant for the monitoring-oriented programming and the Java tasks are shown in Figure 8. The mean for the time taken for the Java exercise was slightly lower than for the Larva exercise (31.474 ± 3.239 vs 38.536 ± 5.456 mins). We attribute this to the fact that the participants all had working Java experience but only a few had used Larva before. Note that there is no statistical significant difference between the time measurements for the two different approaches (paired Wilcoxon test, $\alpha = 0.05, p = 0.413$).

Approach	Dev. time	Total LoC	Extra LoC	LoC (%)	WMC	ACC
Original code	n/a	242	n/a	100	31	1.47
Manually instrumented	31m28s	275	40	116	36	1.56
Monitoring oriented	38m32s	242	58	124	31	1.47

It is interesting to note that although no code was added to the original system files when using the monitoring-oriented approach, more lines of code were written due to the verbosity

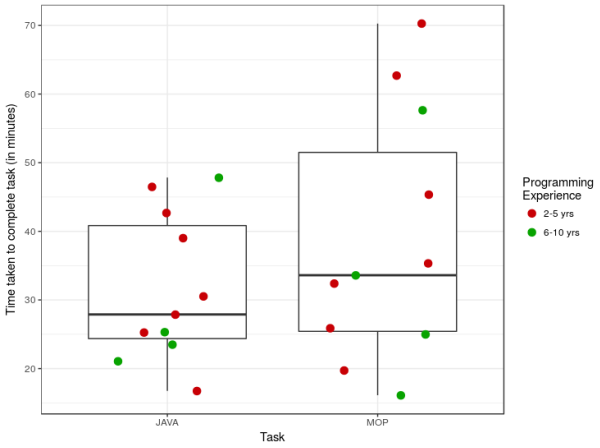

Figure 8: Time taken to implement the Java and Larva tasks by each of the 11 participants in the study.

of Larva scripts[5]. On average participants wrote 50% more lines of code in the Larva implementation when compared to the manual approach using Java.

Since WMC measures the number of possible execution paths through the application, the Larva approach resulted in lower complexity since the (source) logic is completely separate from that of the original system. The ACC for the monitoring-oriented approach remained unaltered since little (if any) code was added at the source level of the application. This, however was not the case with the manual approach where the average ACC rose by 0.08 with respect to all the methods in the application. It is important to note that the ACC and WMC were measured with respect to the original Java code and not the Larva script, thus the code written in Larva was not measured.

Manual code review: After the tasks were completed the code produced by the participants was reviewed. It was interesting to note that 2 of the participants who were instructed to implement the Larva approach first, adopted an architecture inspired by the monitoring design pattern imposed by Larva when implementing the Java version. Also, in virtually all cases, the logic implemented for the tasks was very similar, with the most common difference being that of handling timers.

Based on the interviews, most participants preferred using the manual approach as they noted that this was their first experience with Larva (and runtime monitoring tools in general). However, almost all (~91%) agreed that the monitoring approach was the way to go for larger systems. Given the small number of participants it is difficult to draw observations of statistical significance, however, it is encouraging to see how users, who are new to monitoring

[5]At the time of writing, no visual editor for DATEs exists, and DATEs have to be described in a text format which users usually spread over various lines to improve readability.

technology, found the approach attractive and how it took them similar time to implement an adaptive UI when compared to a more traditional approach.

V. Evaluating Complexity for a Larger Use-Case

We have also looked the impact of adopting a monitoring approach for a larger use-case. We have used SimpleNotepad[6], an open source word processor as the base application into which a set of adaptive features were added using both a manual and a monitoring approach. SimpleNotepad contains standard word processing functionality, including text formatting, saving, opening and printing of files.

The adaptive UI features introduced were: (i) adding compound formatting, through which the application tries to identify patterns of use of text formatting features (bold, italic, underline) usage and groups them together in a single new button; and (ii) adding auto-indent feature which notices indents done at the beginning of a new line and automatically indents new lines. The same metrics used in the user acceptance experiment were measured for this use-case, and can be found in the table below:

Approach	Total LoC	Extra LoC	LoC (%)	WMC	ACC
Original	2972	n/a	100	539	2.32
Manual	3292	885	130	643	2.37
MOP	3149	1185	140	604	2.28

Using Larva resulted in 30% more lines of code when compared to the manual approach. However, when using Larva, only 177 lines of code were added to the original application, mostly consisting of boilerplate methods for Larva to hook onto. With the manual approach, 320 lines of code had to be added to the original code and the rest of the logic (865 lines of code) written as a separate class.

In terms of code complexity, both approaches resulted in an increase in WMC, although the increase with the manual approach (from 539 to 643) was substantially higher than that of the monitoring approach (from 539 to 604). Given that the ACC is computed as the average complexity of the methods in the application, paradoxically, the monitoring-oriented approach resulted in a drop of ACC, due to the dummy methods added to the code. In the case of the manual approach the ACC increased with respect to the original code.

For the two cases, the code used in order to adapt the UI was almost identical between the two approaches. What differed was mainly the way they interacted with the original application and how relevant events were identified and logged. In the manual approach, counters and other logic were added to the methods in the original system to enable adaptation, while in the Larva monitoring approach, capturing the events and logic were within the monitor itself.

VI. Discussion and Related Work

From the evaluations, it was observed that a monitoring approach using Larva resulted in more lines of code than when using a manual approach. However, most of the extra code is contained within a single separate script file, whilst that in the manual approach has to be partially or completely embedded into the source code. Despite the increase in code size, none of the participants cited this as detrimental to the understandability or usability and when asked which option they would use to develop a larger adaptive UI almost all preferred the monitoring approach due to its modularity. This was also our experience with the large use case as during its development, the fact that more lines of code had to be used did not affect the experience whilst coding it.

However, the monitoring-oriented approach proved to be most problematic for most of the participants as they were unfamiliar and inexperienced with Larva. They did not however find Larva or the concept of monitoring-oriented programming difficult to understand and use – as can be noted from the difference in the average time taken to complete each task. Given that for more than half the participants, Larva was completely new and that all participants were adept in Java, the difference for the times between the two sets is not significant (as also showed by the statistical test).

Given that the additional code of the monitor oriented approach is in a separate file and internally works through the use of aspect oriented programming, the complexity of the application is lowered. This applies to both the ACC and the WMC and applies for both evaluations. Along with the decreased complexity, the monitoring-oriented code produces more understandable code as commented by the users. This, was also observed during the large use case as it was noted that designing the logger in terms of states and transitions is easier as opposed to designing it in terms of method calls.

Given that the participants noted that reasoning in terms of automata was helpful, the choice of Larva may be considered as fortuitous, but means that the users' views on their preference for this approach on larger projects cannot be automatically generalised for other monitoring tools.

Navarro *et al.* [13] have previously used monitoring tools on UIs, and used a variety of monitoring tools to evaluate the approach. However, in contrast with our approach, the work there focusses on the verification of UIs, rather than their development, and to the best of our knowledge, the use of monitoring-oriented programming has not been used to develop adaptive user interfaces before.

However, Zaman *et al.* [14] investigated the use of aspect oriented programming for this purpose. The architecture they propose is similar to ours, although our approach has the advantage that we can further abstract away from the events using the temporal logic supported by the monitoring tool (DATEs in the case of Larva). Furthermore, they provide no experimental evidence that the approach is, in fact, beneficial.

VII. Conclusions

From the results of the evaluations performed, it is evident that monitoring-oriented programming can be a viable

[6]Available on GitHub: https://github.com/statickidz/SimpleNotepad-Swing

approach to the development of adaptive user interfaces. Whilst many questions require more data and experimental setups to be able to answer effectively, the initial results indicate that this it is worth investigating further. The fact that all participants were of the opinion that for larger systems monitoring-oriented approaches would give better results in developing adaptive UIs since the approach aids modularity and code separation, further supports this statement. One of the drawbacks of using the monitoring approach with Larva was that of code verbosity, which are due to the language adopted by Larva. The other drawback, from a purely pragmatic perspective, is that most people are not familiar with monitoring technologies, although the speed with which some of the experimental subjects were able to implement UI adaptation using a technology with which they were not at all familiar was impressive.

This study did not touch upon overheads on system resources due to monitoring as opposed to direct coding. Other studies (e.g. see the results of the runtime verification tool competitions [15]) looking at overheads due to monitoring have shown overheads to be, of an acceptable level.

REFERENCES

[1] R. Molich and J. Nielsen, "Improving a human-computer dialogue," *Commun. ACM*, vol. 33, no. 3, pp. 338–348, 1990. [Online]. Available: http://doi.acm.org/10.1145/77481.77486

[2] N. F. Schneidewind, "The state of software maintenance," *IEEE Trans. Software Eng.*, vol. 13, no. 3, pp. 303–310, 1987. [Online]. Available: http://dx.doi.org/10.1109/TSE.1987.233161

[3] G. E. Krasner and S. T. Pope, "A cookbook for using the model-view controller user interface paradigm in smalltalk-80," *J. Object Oriented Program.*, vol. 1, no. 3, pp. 26–49, Aug. 1988. [Online]. Available: http://dl.acm.org/citation.cfm?id=50757.50759

[4] D. Benyon, "Accommodating individual differences through an adaptive user interface," Open University, Tech. Rep., 2010.

[5] F. Chen and G. Rosu, "Java-mop: A monitoring oriented programming environment for java," in *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, ser. Lecture Notes in Computer Science, N. Halbwachs and L. D. Zuck, Eds., vol. 3440. Springer, 2005, pp. 546–550. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-31980-1_36

[6] T. Lavie and J. Meyer, "Benefits and costs of adaptive user interfaces," *Int. J. Hum.-Comput. Stud.*, vol. 68, no. 8, pp. 508–524, 2010. [Online]. Available: http://dx.doi.org/10.1016/j.ijhcs.2010.01.004

[7] K. Z. Gajos, M. Czerwinski, D. S. Tan, and D. S. Weld, "Exploring the design space for adaptive graphical user interfaces," in *Proceedings of the working conference on Advanced visual interfaces, AVI 2006, Venezia, Italy, May 23-26, 2006*, A. Celentano, Ed. ACM Press, 2006, pp. 201–208. [Online]. Available: http://doi.acm.org/10.1145/1133265.1133306

[8] P. A. Akiki, A. K. Bandara, and Y. Yu, "Adaptive model-driven user interface development systems," *ACM Comput. Surv.*, vol. 47, no. 1, pp. 9:1–9:33, 2014. [Online]. Available: http://doi.acm.org/10.1145/2597999

[9] M. Leucker and C. Schallhart, "A brief account of runtime verification," *J. Log. Algebr. Program.*, vol. 78, no. 5, pp. 293–303, 2009. [Online]. Available: http://dx.doi.org/10.1016/j.jlap.2008.08.004

[10] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP*, 1997, pp. 220–242. [Online]. Available: http://dx.doi.org/10.1007/BFb0053381

[11] C. Colombo, G. J. Pace, and G. Schneider, "Dynamic event-based runtime monitoring of real-time and contextual properties," in *Formal Methods for Industrial Critical Systems, 13th International Workshop, FMICS 2008, L'Aquila, Italy, September 15-16, 2008, Revised Selected Papers*, ser. Lecture Notes in Computer Science, D. D. Cofer and A. Fantechi, Eds., vol. 5596. Springer, 2008, pp. 135–149. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03240-0_13

[12] S. P. Satyavan Nain, Anshu Parashar, "Empirical evaluation of software quality using object-oriented software metrics," *International Journal of Advanced Research in Computer Science*, vol. 3, no. 1, 2012, jan - feb.

[13] P. L. M. Navarro, D. S. Ruiz, and G. M. Pérez, "A lightweight framework for dynamic GUI data verification based on scripts," *Softw. Test., Verif. Reliab.*, vol. 26, no. 2, pp. 95–118, 2016. [Online]. Available: http://dx.doi.org/10.1002/stvr.1579

[14] S. A. T. M. Atif Zaman, Mudassar Ahmad, "Adaptive graphical user interface for web applications using aspect oriented component engineering," *International Journal of Computers and Technology*, vol. 10, no. 2, pp. 1384–1391, August 2013.

[15] G. Reger, S. Hallé, and Y. Falcone, "Third international competition on runtime verification - CRV 2016," in *Runtime Verification - 16th International Conference, RV 2016, Madrid, Spain, September 23-30, 2016, Proceedings*, 2016, pp. 21–37. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-46982-9_3