Adaptability Performance Trade-off: a Controlled Experiment

Adam Brennan and Des Greer
Queen's University Belfast
Belfast BT7 1NN
+44 28 90974656
{adambrennan07|des.greer}@qub.ac.uk

ABSTRACT

It is well established that a very large proportion of the total cost of developing software systems is attributed to maintenance and evolution. Various design patterns have become known for increasing the adaptability of software to reduce this cost. In this study the focus was placed on performance with the research question 'How does the use of popular design patterns impact performance?' To answer this, an experiment was designed were a number of music players were developed using popular patterns, and the performance compared to a functionally identical player developed avoiding the patterns under investigation. The results show that in all cases the design patterns had an impact on performance, but except in the case of severe hardware constraints this impact is expected to be so minimal that any performance downside is outweighed by the advantages of using the pattern.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features – *patterns*. D.2.8 **[Software Engineering]**: *Metrics, Performance measures, Product Metrics*.

General Terms

Performance, Design, Experimentation.

Keywords

Design Patterns, Adaptability, Performance

1. INTRODUCTION

Research has shown that up to 90% of the total cost of developing software systems is attributed to maintenance and evolution, a situation often referred to as the legacy crisis [7]. Much of this cost is due to changing requirements and operating environment. Software adaptability is concerned with "producing applications that can readily adapt in the face of changing user needs, desires, and environment" [5] therefore reducing the cost of change. Various design patterns have become known for increasing the adaptability of software, the most popular having been made famous by the 'Gang of Four' in the publication 'Design Patterns: Elements of Reusable Object-Oriented Software' [1].

Design patterns offer adaptability predominantly through use of indirection and decoupling. In some cases this may complicate the design and in many cases impact performance. The potentially negative effect on performance will be experienced every time the software is used. If it is investigated, the results could be used to give a more informed perspective of whether the benefits gained by implementing design patterns are worth the performance trade-off. There do not appear to have been any empirical studies into the performance aspect of design patterns, with focus instead having been placed on adaptability offered by the use of patterns. Therefore this work will add to the existing research, focusing on the performance trade-off associated with this added maintainability.

2. RELATED WORKS

Previous work has been carried out analyzing the benefits of using patterns, that is, that they make software more adaptable and less costly to maintain.

Prechelt et al investigated this insinuated benefit in a study that resulted in the recommendation that 'unless there is a clear reason to prefer the simpler solution, it is probably wise to choose the flexibility provided by the design pattern solution' [6]. Masuda, Sakamoto and Ushijima carried out a quantitative evaluation of a system redesigned with design patterns [3]. The design patterns used were shown to improve the flexibility and extensibility of the software. It has been demonstrated that when modifying a piece of software that had been refactored to introduce patterns, 'the time spent even by the inexperienced maintainers on a refactorized version is much shorter than that of the experienced subjects on the original version' [4].

These investigations have concentrated on the maintainability aspects of using design patterns and not on the performance impact, which is the focus of this study.

3. STUDY DESIGN

The research question is 'How does the use of popular design patterns impact performance?'A simple music player application was developed without using any of the design patterns chosen, then with each of the patterns implemented so the performance of the area of implementation could be measured and compared to the alternative method.

The patterns chosen were selected for the increase in adaptability they provide (shown in previous studies), their popularity, and their suitability in the music player application. The patterns are; Mediator, Observer, State, and Strategy [1].

The version of the player developed not using the patterns under study is the 'Alternative Music Player' (AMP). Each of the players that use a pattern were named 'Music Player' with the pattern name indicated via a prefix (i.e. MMP for Mediator, OMP for Observer, Strategy MP for Strategy and State MP for State.)

The problem was investigated using a number of experiments, that is, a 'formal, rigorous and controlled investigation' [8] referring to published guidelines [2]. An experiment is by definition quantitative, and the variable under test was the use of a particular design pattern in the implementation of the music player. The measurements used were execution time, CPU utilization, and memory utilization. These 3 measurements are the most commonly used indicators of performance.

To gain a reliable representation of the 3 performance indicators each experiment was repeated 3 times and an average taken of the results. In each repetition 500 recordings were taken for each of the performance measures. This reduces the effect of anomalous recordings in the experiments, and provided details on a 25 second period of resource utilization (CPU and memory recordings were taken at 50ms intervals; 20 per second).

To maintain internal validity, care was taken when placing the timer points for execution time recording, ensuring that the same functionality was recorded in each version of the player. Number of background processes running was minimized whilst the experiments were being performed. External validity was preserved by ensuring that the design patterns were adhered to in design, and that the implementation met the design. This was achieved through use of reverse engineering tools to generate UML class diagrams from the implementation and comparing to the design.

The null hypothesis is 'There is no effect on performance in using the chosen patterns' and the alternative hypothesis 'With the exception of the observer pattern, the chosen design patterns have a negative effect on performance.'

4. RESULTS

An experiment was run for each of the patterns under investigation, comparing the player implementing the pattern with the AMP.

Table 1: Mediator Experiment Results

	Execution Time (ms)	CPU Utilization (%)	Memory Usage (kB)
MMP	2.31	2	45623.33
AMP	1.84	2	45954
Difference	0.46	0	-330.67

The arithmetic mean, mode and median values were calculated from these datasets of 500 recordings. The median was identified as the most useful value to analyze as it takes into account skewing in data sets. The nature of the measurements being taken means skewing and extreme irregular values are likely. This gives 3 median values, one for each repetition of the experiment. The mean of these 3 values is then taken to give an average for each of the performance indicators. The mean was chosen over the median for this second calculation as skewing in the data sets has already been accounted for when calculating the first average. It is these mean values that are included in the tables in this section.

All values were rounded to 2 decimal places. This gives precision to 10 microseconds in execution time, hundredths of a percent in CPU utilization, and 10.24 bytes in memory utilization.

The difference values in the tables in this section are calculated by taking the AMP value from the pattern player value. A positive difference means better performance from the AMP, and a negative difference means better performance from the pattern player.

4.1 Mediator Experiment

In the Mediator Media Player (MMP), the mediator pattern is used to decouple behavior behind button clicks which sets the enabled property of other buttons. This communication between buttons in the UI was centralized in the Mediator class. This is a realistic example of how the mediator pattern might be applied in a real application.

The alternative method used in the AMP is code directly behind the buttons which is generally seen as bad practice.

Referring to Table 1, Execution time was expected to be slower when mediator was implemented as the overhead is greatly increased. In the AMP, the first recording and second recording are only a few lines apart in the code behind the play button in the GUI class. The MMP however follows a much more complex chain of events. The additional message passing was expected to add to the time between trigger event and the appropriate action being taken, and this is shown to be true in all 3 repetitions of the experiment.

CPU utilization remained comparable as expected, with neither player requiring additional computation whilst the remaining idle.

The result of the memory utilization performance indicator was unexpected, showing that the MMP used less memory than the AMP in two of the three runs. These differences were quite varied, with runs 1 and 2 showing 1.1MB and 1.6MB more memory used by the AMP, and run 3 showing 1.7MB less memory used by the AMP. The standard deviation of the difference of 1842 is the highest of any of the players. This large variance suggests there may have been inaccuracy in the recording of the memory usage values.

4.2 Observer Experiment

The observer pattern is commonly used to separate the user interface layer from the application data or controller in Model View Controller architecture. As an example of this, the playlist component of the music player UI was updated using this pattern. The UI subscribed to the controller to be notified of changes to the playlist data, and updated the playlist GUI component when this occurred. An example of a change that would be more easily facilitated by the observer pattern is the addition of another view, for example many music players come with full and mini player interfaces. The execution time experiment measures the time taken from an event which triggers a UI update to the update taking place. The AMP performs this updating through the use of polling. The results for the Observer Media Player (OMP) are shown in Table 2.

As expected, the execution time average is approximately half of the polling interval value used. This is because it is mid-way between the minimum and maximum update possibilities. The OMP time was on average over 30 times faster at performing the same update. This is because of the regular updating of the AMP using polling, whereas the OMP performs updating on demand. The polling interval could be decreased to combat this. An interval of 3ms should bring the execution time in line with the OMP, but this would have a detrimental effect on the other performance indicators. Based on preliminary experiments to find an optimal polling time, an interval of 3ms would likely lead to CPU utilization of 30-40% which is drastically higher than the OMP.

Table 2: Observer Experiment Results

	Execution Time (ms)	CPU Utilization (%)	Memory Usage (kB)
OMP	1.58	0	44460
AMP	49.57	2	46729.33
Difference	-47.99	-2	-2269.33

The CPU utilization of the OMP was lower than the AMP, again due to the regular calling of the update method for polling in the AMP. The OMP only calls the update method when it receives user input that triggers the change. Since this did not happen during the test, the CPU utilization remained on average 0%. This was the modal value in all 3 repetitions of the experiment. The mean in all 3 repetitions was 0.08% showing that there were only small amounts of CPU usage by the OMP. Again this could be reduced by changing the polling time. Even with a 1 second interval the CPU utilization of the AMP remained at 1.96% (identified when performing the experiment to determine optimal polling time); still much higher than that offered by the OMP. This would also lead to a noticeable delay between user interaction and GUI response.

Memory usage was also around 2MB less for the OMP. This was expected due to the removal of the thread and resources required for polling.

4.3 State Experiment

A music player is an ideal application to demonstrate the state pattern. It is an example often used to explain state machines which can be easily translated to the state pattern.

All of the states that determine behavior were separated into classes that implement an abstract state interface. This interface defines the actions or transitions possible between states. The concrete states then fill in the action for each transition. The execution time experiment involved identification of state and execution of the appropriate behavior. Specifically, this will be the time taken from the play button is pressed, until the track starts playing. During this time the state is identified as loaded and the transition is made to the playing state.

The AMP takes the approach of conditional statements to determine behavior based on player state. This easily leads to a situation where changes are difficult to follow.

Table 3: State Experiment Results

	Execution Time (ms)	CPU Utilization (%)	Memory Usage (kB)
State MP	0.35	2	46774
AMP	0.26	2	46354.67
Difference	0.09	0	419.3333

Referring to Table 3, the State Music Player (State MP) was on average 0.09ms slower than the AMP when identifying the player state. This was expected because of the addition of late binding by the state pattern. The average CPU utilization of the players is identical at 2% usage.

The memory usage was expected to be slightly higher for the State MP due to the additional classes used when the state pattern has been implemented. The values in Table 3 include Run 3 which indicates that the AMP used approximately 1.7MB more memory than the State MP. There was however irregularly high memory usage during this run of the experiment by the State MP process whilst the AMP remained fairly constant. The other 2 runs also contradicted this run, indicating savings of 146kB and 358kB for the AMP. Therefore the memory recordings for run 3 are regarded as anomalous; meaning that on average the AMP used more memory than the State MP. Disregarding run 3, the average difference between the State MP and AMP was -252kB, meaning the State MP used less memory than the AMP.

4.4 Strategy Experiment

In a music player, similarity exists in the playing of different audio formats. Regardless of the format, the player can load, play, stop, skip forward, or skip back through tracks. This similar behavior is extracted into an interface or abstract class, which the family of algorithms (in this case each of the specific music format players) implements or extends. The context then refers to the interface or superclass, enabling dynamic run time binding. This allows algorithms to be interchanged and new algorithms to be added easily, for example the support for MP3 or AAC. The execution time experiment involves the identification of the concrete strategy to be used after input from the user. This will be taken as the time from identification of file type (after the play button press event occurs) to the time the music file is played by the specific format player.

In the AMP this is handled in a conditional statement based on file format. This would require an additional branch for each additional format to be supported, whereas with Strategy Media Player (Strategy MP) new formats can be dynamically added.

Table 4: Strategy Experiment Results

	Execution Time (ms)	CPU Utilization (%)	Memory Usage (kB)
Strategy MP	0.27	2	46417.33
AMP	0.22	2	46439.33
Difference	0.05	0	144.67

Table 4 shows the results of the experiment. The time taken to identify the music format player to be used and begin playing the music file was 0.05ms slower in the Strategy MP where dynamic binding is used than in the AMP. This was the expected result

As with the state pattern (which is closely related to strategy) there was no difference in the average CPU utilization times. This was expected as any differences are likely to be minute and therefore removed when calculating the median results from the recording data sets.

Memory utilization was expected to be higher for the player implemented using strategy which was shown in the results. This was expected due to the increase in number of classes required when using the strategy pattern over alternative methods. In the first repetition of the experiment was there a slight increase in memory utilization of the AMP over the Strategy MP. The overall average suggests that the Strategy MP uses more memory and therefore the first run of this experiment is regarded as anomalous.

5. CONCLUSIONS & FUTURE WORK

The alternative hypothesis stated in the method of investigation was 'With the exception of the observer pattern, the chosen design patterns have a negative effect on performance'. This can be said to be true for the state and strategy patterns, both of which resulted in a detrimental effect on execution time and memory usage. The mediator pattern which was expected to have a detrimental effect on performance did result in slower execution time recordings, but the memory utilization decreased. The standard deviation value for memory usage was unusually high, suggesting that the measurements taken were widely spread and therefore could be considered as irregular or unreliable. The experiment could be repeated a larger number of times to see if this is the case. The observer pattern did cause an increase in performance which goes against the results of the other patterns. This was however expected, and observer is a unique case in the patterns under study as it does not make use of indirection or dynamic binding. This also shows that not all patterns that offer increased adaptability do so at the price of performance.

With the exception of the Observer Experiment, the resource utilization results were comparable with the most varying performance indicator being execution time. In many cases the resource utilization measurements would not cause a major problem as it is simple to add more processing power or memory to alleviate the problem. A notable exception to this is mobile devices which are restricted by size and power consumption, so the higher use of memory by the players implementing the state and strategy pattern would be a concern.

The accuracy of the study could be improved by selecting methods of performance analysis that are not open to impact from confounding factors such as other processes. The methods selected for this study were chosen for their practicality and regard as indicators of performance. Due to the very small performance impact caused by the use of patterns it may be beneficial to perform a study using more controllable methods of measurement. The study could also be performed on more restrictive hardware, which should lead to a greater difference in performance of the players.

The choice of Java as development language may also be improved by using a language that is not be reliant upon the Java Virtual Machine (or similar) which adds another level of complexity to controlling the operating environment.

It could also be useful to monitor resource utilization of the players when in use. This study analyzed performance of the players when idle, as a repeatable method of interaction with the player could not be achieved with the hardware analysis method used. If a means of reliably repeating actions in the various players could be constructed, or an alternative method of resource usage reporting implemented, this could give an insight into any performance impact the use of patterns may have whilst in use.

A final suggestion for any additional research into this area would be to choose different applications for testing each pattern. This would improve the external and ecological validity of the study and would enable a greater number of patterns to be investigated.

6. REFERENCES

1. Gamma, E., Helm, R., Johnson, R., & Vlissides, J. M. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley.
2. Jedlitschka, A., & Pfahl, D. (2005). Reporting Guidelines for Controlled Experiments in Software Engineering. International Symposium on Empirical Software Engineering, pp. 95-104.
3. Masuda, G., Sakamoto, N., & Ushijima, K. (2000). Redesigning of an Existing Software using Design Patterns. *International Symposium on Principles of Software Evolution* p. 165.
4. Ng, T. H., Cheung, S. C., Chan, W. K., & Yu, W. T. (2006). Work Experience versus Refactoring to Design Patterns: A Controlled Experiment, Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of Software Engineering, pp. 12-22.
5. Norvig, P., & Cohn, D. (2010). *Adaptive Software*. Retrieved June 10, 2010, from http://norvig.com/adapaper-pcai.html
6. Prechelt, L., Unger, B., Tichy, W. F., Brossler, P., & Votta, L. G. (2001). A Controlled Experiment in Maintenance Comparing Design Patterns to Simpler Solutions. IEEE Transactions on Software Engineering *27* (12), pp. 1134-1144.
7. Seacord, R., Plakosh, D., & Lewis, G. (2003). *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices*. Addison-Wesley.
8. Wohlin, C., Höst, M., & Henningsson, K. (2006). Empirical Research Methods in Web and Software Engineerin, in Mendes, E. and Mosley, N., Web Engineering, Springer Berlin Heidelberg, pp. 409-430.