Term Paper: Examining Consequences of Design Patterns and their Performance Characteristics for select Gang of Four (GoF) patterns

Paul Barnes
Student Number: 16210279

Table of Contents

Introduction

It is well understood that design patterns provide tremendous value in the creation and refactoring of software solutions. This statement is re-enforced with the recognition that most class libraries delivered in the support of various Operating System (OS) platforms draw heavily on some very recognisable patterns to deliver core OS functionality. A very recognisable example of this would be the various collection classes in .Net which all derive from IEnumerable <T>, an implementation of the GoF iterator pattern. Many more patterns are used extensively throughout the .Net class libraries. Chain of Responsibility for exception handling, Adapter for Runtime Callable Wrappers (RCWs), Observer for event handling and Proxy for .Net Remoting are just a few examples of how pervasive design patterns are in this specific framework.

This application of patterns is replicated across many programming languages, class libraries and core language features. An additional example of the application of design patterns is demonstrated with Observable and Singleton mix-ins for Ruby.

It is highly conventional to consider knowledge of design patterns and their intents to be a great strength as a developer. However, as with all strengths, if they are over played, and in certain circumstances, they can switch from being a strength to being a weakness. In most structured courses on design patterns, the focus is spent on the pattern itself and when to use it. And whilst many courses will outline the consequences of each pattern, frequently this does not get the same levels of focus and attention, by design you could say. In this paper, we will examine some of the characteristics of several of the GoF patterns and we will measure some run-time characteristics of these patterns. We will also identify some general considerations that could be useful to bear in mind when using these patterns.

Design Patterns Perform Poorly

One main dimension that this paper will examine is the following statement;

Design patterns have poorer run-time performance characteristics than comparable solutions without them.

One of the main advantages of design patterns is their impact on the readability, extensibility and maintainability of the underlying code that they influence. These attributes are understandably very good characteristics for software solution, however you could imagine that a bi-product of these strengths could be run-time performance. Put another way, design patterns tend to be optimised for code factoring and can often use OO mechanisms, such as inheritance, to break out functionalities into abstracted and encapsulated components and classes. Even though many compilers will optimise the code for run time performance it could be argued that more method calls imply a slower execution, higher processor usage or more memory usage, even if only at slightly higher levels. We will examine if this holds true for some selected patterns.

Performance

Software performance is an ambiguous space in general. For some problem spaces performance means run-time execution times, in others it could mean memory usage, or latency for remote calls. Whilst these may vary in importance from solution area to solution area, they are usually quite measurable.

For the purposes of this paper we will be considering two measurements;

1. Execution time (measured using ruby-prof).
2. Memory usage for the solution (measured using ‘ps’ on Mac OSX)

As these factors are completely measurable, we can conduct an experiment for some of the GoF patterns to see how the application of the design pattern affects the performance of the solution with respect to these dimensions. For each pattern considered, we will look at one implementation using the pattern which we will term the ***"with pattern"*** solution and a corresponding implementation without the pattern, termed the ***"without pattern"*** solution. A hypothesis will be made with respect to one or more of these measurements before testing and the experiment will examine whether this hypothesis holds true.

Experiment Conditions

For each pattern, we will examine some code that should have two reasonable implementations for the same problem. The solutions will then be profiled using numerous test runs executions and the results will be compared. Typically a test run will have 1000 iterations, and each test case will contain 10,000 individual test runs. We will look at max, min and mean values for each appropriate measure. The development environment will be RubyMine but the run time environment will be the OS ruby environment. Each solution will have a file main.rb which will execute the tests;

The method single_test_pass will run an individual pass (typically 1000 iterations of the solution) and the following methods will be used to measure CPU and memory respectively;

```ruby
def profile_runs_CPU
  run_count=0
  output_file=File.open('CPUtime - with.csv', 'w')
  output_file.puts 'Run number:,Thread number:,CPU_Total_Time'
  10000.times do
    run_count+=1
    RubyProf.measure_mode = RubyProf::CPU_TIME
    RubyProf.start

    single_test_pass

    result=RubyProf.stop

    thread_count=0
    result.threads.each do |thread|
      output_file.puts "#{run_count},#{thread_count},#{thread.total_time}"
      thread_count+=1
    end
  end
  output_file.close
end
```

```ruby
def profile_runs_mem
  run_count=0
  output_file=File.open('MemUsage - with.csv', 'w')
  output_file.puts 'Run number:,Kb in use'

  10000.times do
    run_count+=1

    single_test_pass

    memory_usage = `ps -o rss= -p #{Process.pid}`.to_i # in kilobytes
    output_file.puts "#{run_count},#{memory_usage}"

  end
  output_file.close
end
```

Experiment Code Location

All the code referred to in this term paper is available from the following GitHub location;
https://github.com/paulba71/DesignPatternsTermPaperCode
Each pattern will have a sub folder containing the code for that example with the pattern implemented and without the pattern implemented.

Experiment 1 – Singleton (Creational Pattern)

Overview and Intent

Singleton is one of the very simplest of design patterns. It is intended to provide global access to a single object removing the need to create or pass objects each time the functionality encapsulated is needed. Singletons also support "lazy instantiation" when the memory needed will only be allocated when the first call to instance is made.

General Consequences

One consideration that must be evaluated when using Singletons is if there could be multiple consumers of the code. As there is one global instance of the class, it may be necessary to provide a queueing mechanism to ensure that multiple consumers are not trying to use the object simultaneously.

Also, true Singleton classes cannot be sub-classed as to do so would result in more than one instance present.

As software grows and scales Singleton classes can present a barrier and indeed issues as often the need to have one single, global instance becomes a limiting constraint.

Experiment Code

The experiment code here will be a solution that uses the state pattern, as often these patterns go hand in hand. The "without pattern" solution implements the state pattern but not the Singleton pattern making Singleton the area of change we will measure.

In this solution specifically, we will look at a simulation of model of a storage system that tries to store items in contiguous blocks, defragmenting as it goes. The algorithm used for the defrag operation depends on the usage of the device and is encoded in the state objects. The defrag frequency will also change with the state change. There will be 5 states of usage; with increasing write access times as shown in the following table;

Usage State	% usage	Defrag times	Defrag rhythm
Low	0-20	2 ms	Each single write
Medium Low	20-40	20 ms	2 writes
Medium	40-60	50 ms	3 writes
Medium High	60-80	100 ms	4 writes
High	80-100	200 ms	5 writes

These defrag times will be simulated with sleep commands.

As a storage device moves from state to state, it well become slower to write data, although all devices with similar states will perform in a similar manner. The use of the state pattern is not the important part of this experiment, rather the fact that the state objects are singletons. The strategy pattern could also be applied to this problem.

In the ***"with pattern"*** solution the states will be implemented as singletons, and in the ***"without pattern"*** solution they will not be. The ***"without pattern"*** solution could also be implemented without the new-ing the state at each change but for the purposes of the experiment we will look at the worst-case implementation. The differences are shown in the following code fragments;

```
if @usage_percentage== 20
  @behaviour=MediumLowUsageState.instance
  puts''
  puts 'device has switched state to medium low usage state'
end
```

```
if @usage_percentage== 20
  @behaviour=MediumLowUsageState.new
  puts''
  puts 'device has switched state to medium low usage state'
end
```

A single test case will fill a device from 0-100% 1000 times. The test run will consider 10,000 test cases.

Expected Performance Characteristics

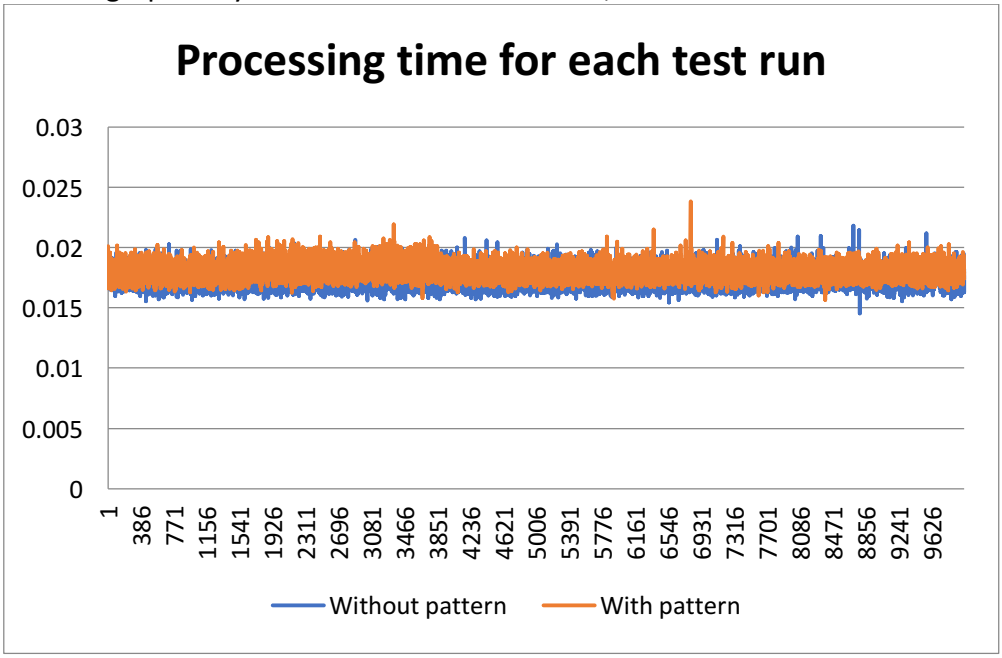
Initial hypothesis is that the use of Singleton will have performance gain in this scenario. As the code is being looped 1000 times, the use of Singleton will avoid the new-ing of state objects and the additional resources needed to do this.

Experiment Results

For processing time the results are as follows;

Solution Type	Total Time (seconds)	Average Test Time (seconds)	MAX Test Time (seconds)	MIN Test Time (seconds)
With Pattern	178.7644	0.0179	0.0238	0.0156
Without Pattern	173.2390	0.0173	0.0218	0.0146

Plotted graphically all test runs look as follows;

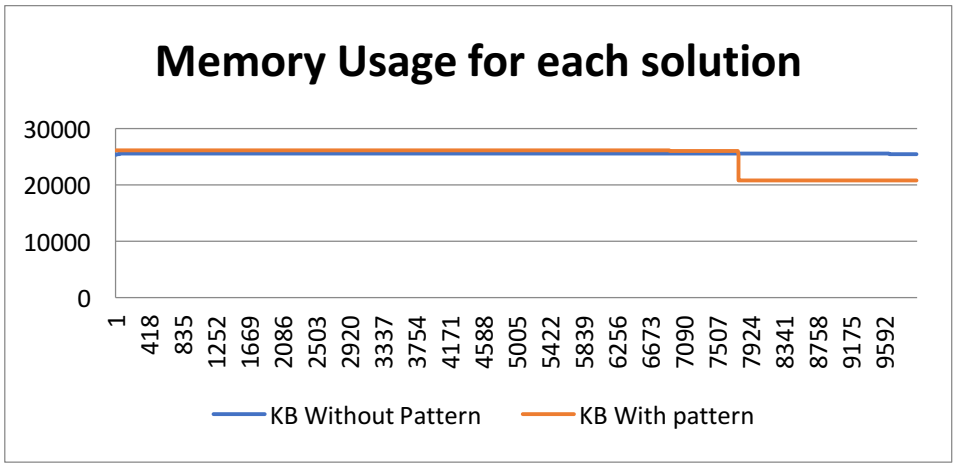


For this specific problem, the test runs are very, very close. The difference in total time for all runs is just 5.5 seconds for 10 million individual simulations (10,000 * 1,000). One could most of the time is spent in sleep() methods to simulate the additional load on the devices, but this is the case for both solutions.
In this case for CPU time, the initial hypothesis does not hold true.

When we look at memory usage, the results are as follows;

Solution Type	Average Mem Usage (KB)	MAX Mem Usage (KB)	MIN Mem Usage (KB)
With Pattern	24864.7112	26048	20752
Without Pattern	25498.87	25504	23724

Plotted graphically for all test runs the results look as follows;



As with processor usage, memory usage is almost the same for each solution. The solution with the pattern however did free up some memory at just under 7900 test runs to become more efficient.

Conclusion

For this problem, the hypothesis that the Singleton pattern will result in a better performance pattern does not hold true. In both solutions, the CPU usage and the memory usage are so close that the performance characteristics could be considered as the same.

Experiment 2 – Template Method (Behavioural Pattern)

Overview and Intent

The intent of the Template method is defining the structure of a solution as an algorithm and then delegate the specific implementation of individual parts to sub-classes. This allows for the separation of specific details from the general algorithm.

General Consequences

Template method can really improve both the readability and maintainability of code however it could be easy to either implement a method that should not be implemented, or to forget to implement a method that should be. Different languages can offer protection against these but care must be taken to extend appropriately. As flow passes from sub-class to parent and back again many times it can also prove tough to follow the logic and really understand where various flows really live.

Experiment Code

For this experiment the report example from the Template Method chapter in the Russ Olsen book will be used. In this example, there will be a very simple report writer that supports two modes; HTML format and plain text format. The ***"without pattern"*** implementation will use *if statements* to determine the actual output in an inline manner such as the following lines;

```
@text.each do |line|
  if format == :plain
    puts (line)
  else
    puts ("   <p>#{line}</p>")
  end
end
```

The implementation with the pattern creates a skeletal class Report and sub-classes HTMLReport and PlainTextReport which will contain the specific implementation for each sub report type. The structure is shows as follows;

```
def output_report
  output_start
  output_head
  output_body_start
  output_body
  output_body_end
  output_end
end
```

Some sub-class methods do nothing and others do some work depending on the report type.

Expected Performance Characteristics

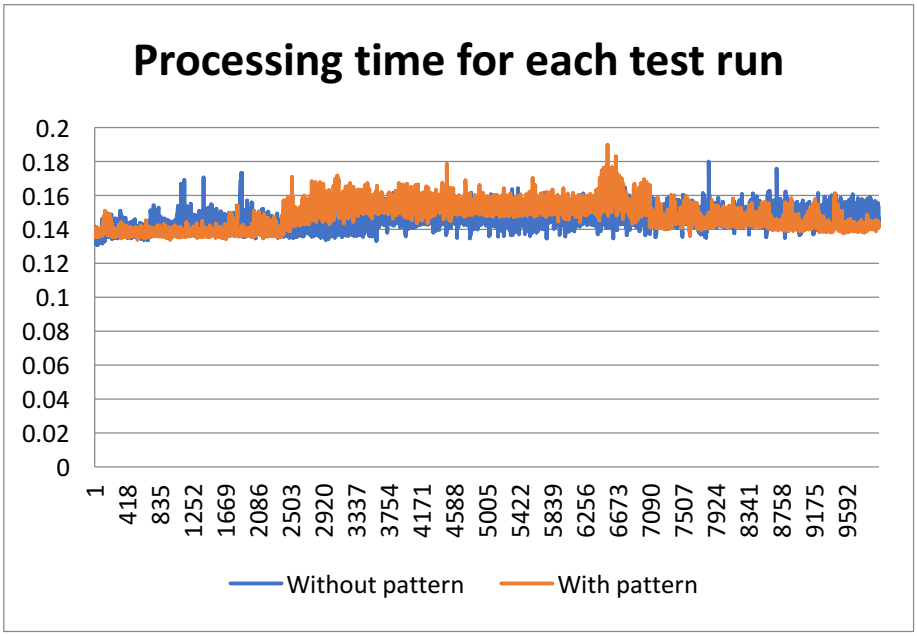
As Template Method makes use of classes and inheritance to encapsulate the functionality of the common and specific code, the hypothesis is that there will be additional runtime cost in using the pattern.

Experiment Results

For processing time the results are as follows;

Solution Type	Total Time (seconds)	Average Test Time (seconds)	MAX Test Time (seconds)	MIN Test Time (seconds)
With Pattern	1472.9423	0.1473	0.1896	0.1339
Without Pattern	1449.2855	0.1449	0.1800	0.1307

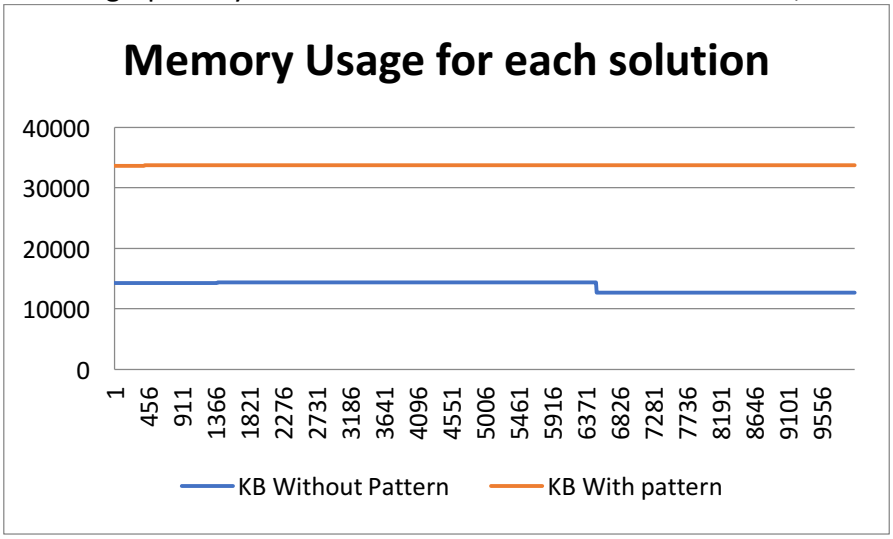
Plotted graphically all test runs look as follows;



When we look at memory usage, the results are as follows;

Solution Type	Average Mem Usage (KB)	MAX Mem Usage (KB)	MIN Mem Usage (KB)
With Pattern	33699.1688	33716	33640
Without Pattern	13754.3548	14336	12692

Plotted graphically for all test runs the results look as follows;



Conclusion

The CPU time is very close between the two solutions with only a 23 second "penalty" for the ***"with pattern"*** solution over the 10 million individual runs. However, the memory usage between the two solution is dramatically different with the ***"with pattern"*** solution taking 3 times as much memory as the ***"without pattern"*** solution. Whilst these numbers look a little suspicious, it would appear to corroborate the hypothesis that the design pattern has additional performance constraints.

Experiment 3 – State Pattern (Behavioural Pattern)

Overview and Intent

The state pattern allows an object to alter its behaviour when its internal state changes. The object will appear to change its class. As an object's context changes it can result in a state change also, and behave differently.

General Considerations

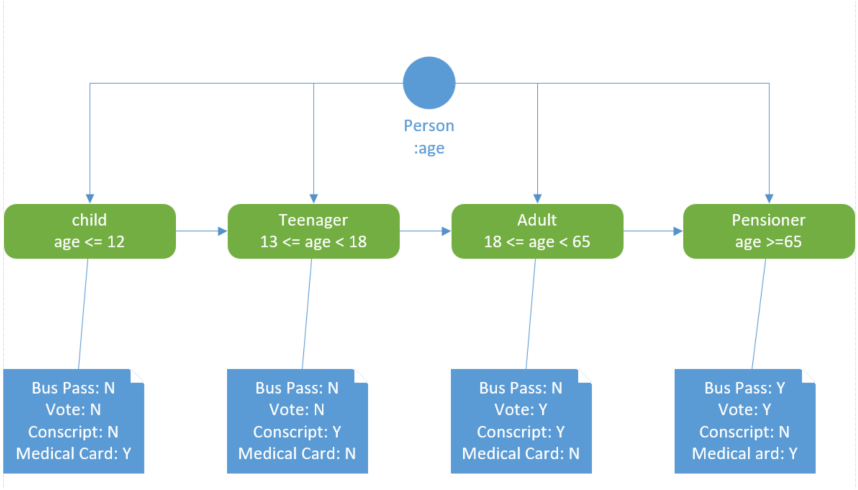
State objects are often Singletons so any potential downsides of using Singletons could also transfer to State objects also.

Depending on the solution, the State pattern can result in an increase in the number of objects as there will be additional state objects create to encapsulate each different state. If there are many different states, then there could be more efficient ways to solve the problem.

Experiment Code

For this design pattern the problem from the labs was used as the test case. The code provided at the start of the exercise will be used for the ***"without pattern"*** solution. Parts 2 and 3 from the lab were implemented to add the teenager *conscription* case and the *apply_for_medical_card* case also in a very similar way to the functionality provided.

The ***"with pattern"*** solution implements each stage of life as a state class; The can been seen in the following state diagram;



State objects are also implemented as Singletons, as is often the case, and this would be a reasonable implementation for this specific example.

Each test run will iterate a life cycle (age running from 0 to 100) 1000 times. Each test pass will measure 10,000 test runs as with all experiments.

Expected Performance Characteristics

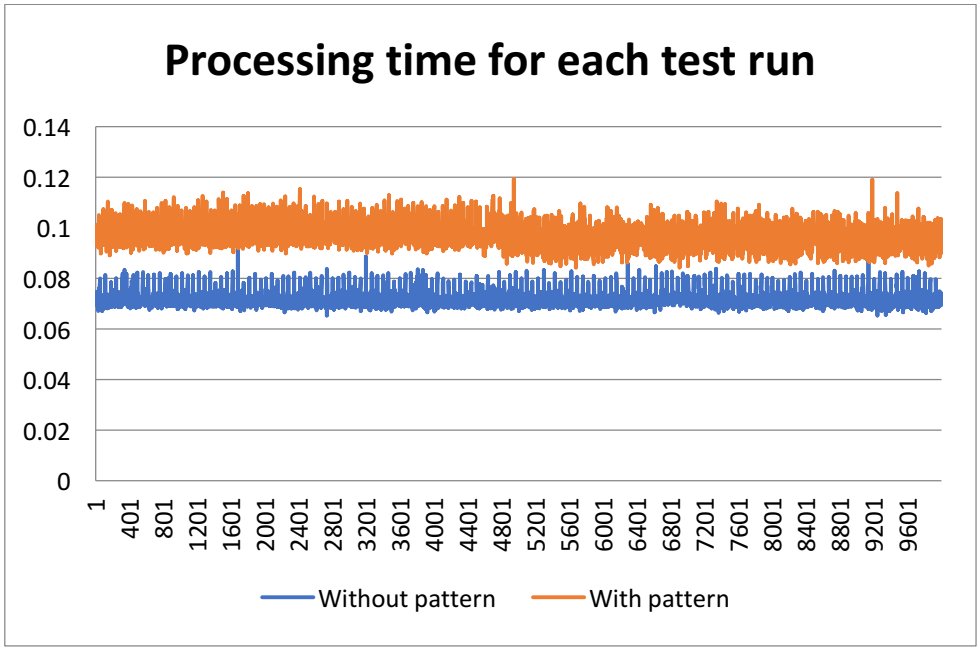
The hypothesis here is that as the state objects are also Singletons that CPU time as well as memory usage should be slightly better when using the ***"with pattern"*** solution. Memory usage should also be quite close between both solutions.

Experiment Results

For processing time the results are as follows;

Solution Type	Total Time (seconds)	Average Test Time (seconds)	MAX Test Time (seconds)	MIN Test Time (seconds)
With Pattern	963.0436	0.0963	0.1194	0.0844
Without Pattern	715.5176	0.0716	0.0912	0.0654

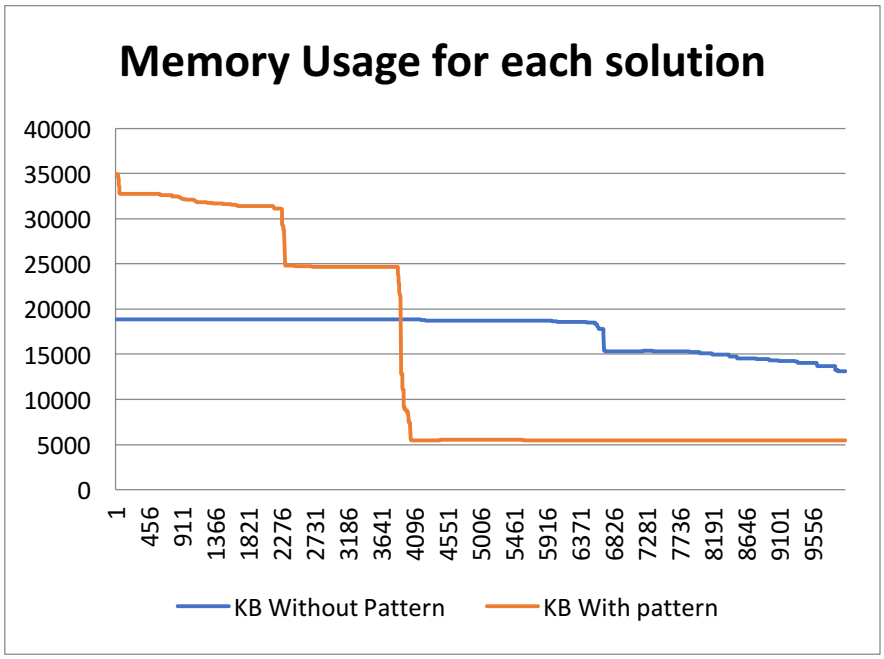
Plotted graphically all test runs look as follows;



When we look at memory usage; the solution with the design pattern implemented needs more memory to run. The results are as follows;

Solution Type	Average Mem Usage (KB)	MAX Mem Usage (KB)	MIN Mem Usage (KB)
With Pattern	14772	34954	5452
Without Pattern	17424	18840	13124

Plotted graphically for all test runs the results look as follows;



Conclusion

Results here do not corroborate the hypothesis that the CPU usage should be close with the "with pattern" solution being slightly better. The results show that the without solution is quite a bit more efficient when it comes to CPU usage. The memory usage pattern is also quite strange for the ***"with pattern"*** solution and out of character with the other experiments run. This possibly point to an implementation problem will applying the pattern. As this run is completely out of character with all other patterns we will run the experiment again and the results will be reported in the Appendix.

Experiment 4 – Decorator (Structural Pattern)

Overview and Intent

The Decorator pattern has the intent to attach additional responsibilities to an object dynamically. Objects do not know that they are being decorated. Decorators provide a flexible alternative to subclassing for extending functionality. (Source: lecture notes for the module)

General Considerations

Decorators are uni-directional, an object does not know that it is being decorated and in what manner that may be. The additional logic applied to the object could then corrupt the original intent of the object being decorated. This could be a valid consideration when decorating a class in a legacy system.

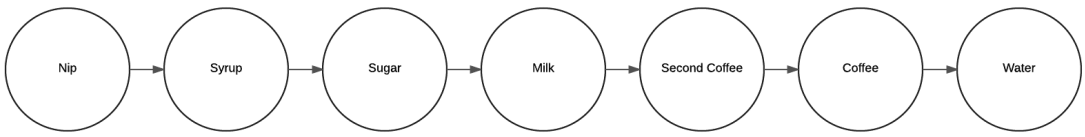
The extensive use of decorators can also give rise to an increase in the number of classes and complexity in the system. Whilst decorators are certainly more flexible than sub-classing, they may override, obscure logic below the decorator.

It is also entirely possible for decorators to contradict, or accelerate each other. A contrived example could be where a decorator applies a discounted price to an item. If multiple decorators are applied to an object then the effects will probably be cumulative, which may not be the correct behaviour.

Experiment Code

The code to compare here will be the code that was written in the lab of the module. It models a coffee production system where users can choose water (hot/cold), a coffee type, milk type, sugar type, syrup type and "nip" type (a general addition such as a type of whiskey). Each addition is optional, can be larger than 1 and has an associates unit price. The total price is the sum of all components. There will also be a means to heat the milk that other objects don't have. The drink will have a method, "declare", that will describe its constituent parts.

With the pattern the code will create a drink from water and then decorate the object with each different addition. The Object hierarchy would look as follows;



The Milk class supports a unique method to heat the milk. Other objects don't need to know this.

Without the pattern, there will be a single class "Drink" that will maintain a hash of each component added. The heat milk method is added to the class to support that functionality. This is visible to all aspects of the class. This solution is quite different to the ***"with pattern"*** solution as there is an additional hash that is used to provide some of the functionality that each decorator leverages as the classes are nested. It is however a reasonable solution to the problem when we are not decorating the objects.

Expected Performance Characteristics

As each component using the pattern is a class, there will be additional creation time needed to initialize these. However, as the solution chosen without the pattern implemented uses a hash to store the various attributes of the drink, it is likely that the insertion and lookup of the hash will outweigh any consequences of the class hierarchy of

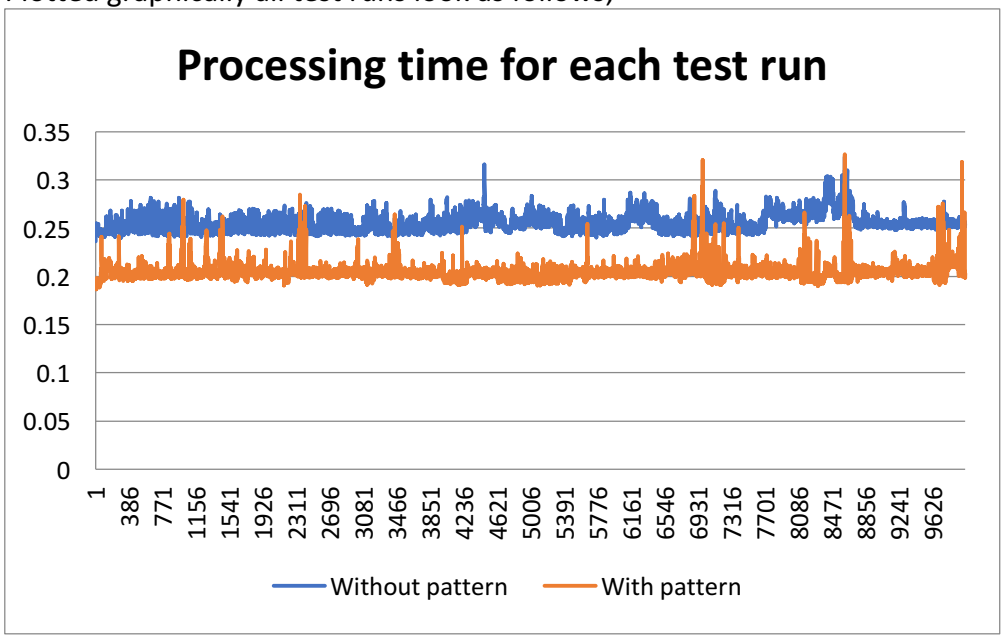
the solution utilising the pattern. Taking the hash into consideration the hypothesis is that for these given solutions, the solution with the pattern implemented will be more efficient both in terms of CPU time and in memory usage.

Experiment Results

For processing time the results are as follows;

Solution Type	Total Time (seconds)	Average Test Time (seconds)	MAX Test Time (seconds)	MIN Test Time (seconds)
With Pattern	2036.7238	0.2037	0.3264	0.1869
Without Pattern	2551.2541	0.2551	0.3239	0.2366

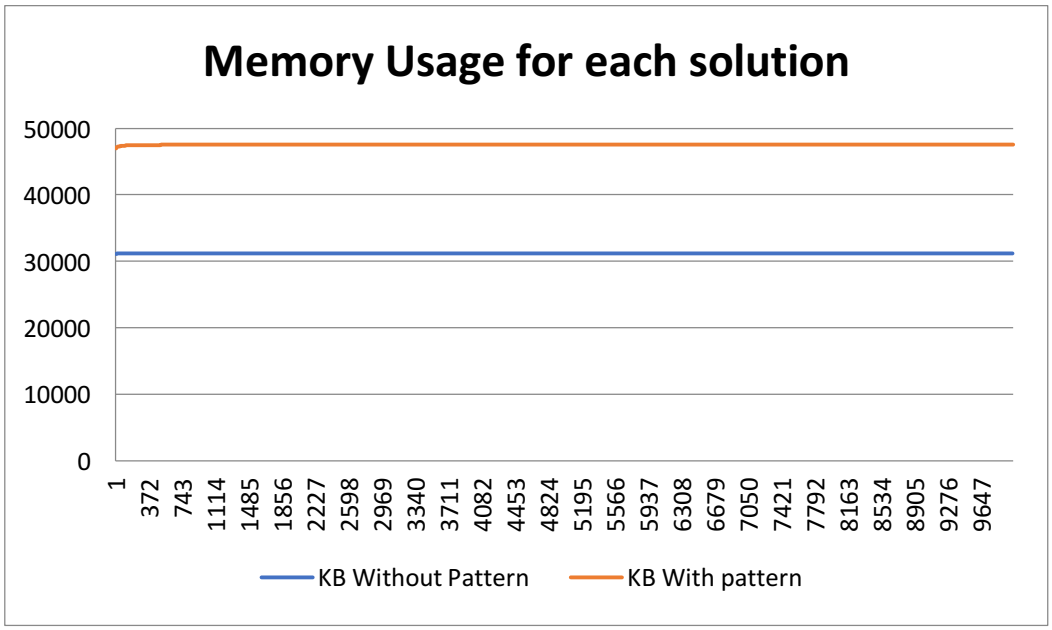
Plotted graphically all test runs look as follows;



When we look at memory usage; the solution with the design pattern implemented needs more memory to run. The results are as follows;

Solution Type	Average Mem Usage (KB)	MAX Mem Usage (KB)	MIN Mem Usage (KB)
With Pattern	47504.9804	47508	47004
Without Pattern	31135.9976	31136	21124

Plotted graphically for all test runs the results look as follows;



Conclusion

The hypothesis that the “with pattern” solution is more efficient from a CPU usage due to the “without pattern” solution needed to use a hash to store the individual components appears to hold true. However, the hypothesis that the memory usage would be more efficient also does not hold true. The memory usage for the “with pattern” solution is substantially higher than without. This would be an interesting problem to pursue in another paper.

Experiment 5 – Object Pool (Creational Pattern)

Overview and Intent

Object pooling can offer a significant performance boost; it is most effective in situations where the cost of initializing a class instance is high, the rate of instantiation of a class is high, and the number of instantiations in use at any one time is low. (Source http://sourcemaking.com)

General Consequences

One thing that is of concern when considering the object pool pattern is stale state. If a caller acquires and object and then changes its state, it must be reset when it is returned to the pool. Failing to do that would result in a highly-corrupted pool of objects.

Experiment Code

The test code for this pattern is a cut down version of a spell-checking system such as one in Microsoft Word. There will be 4 languages supported; English, French, German and Spanish and each language will have a small dictionary, stored in an array, and a series of words to check, some in the dictionary, some not. The check method will simply lookup the word from the dictionary and return true if it is known, and false if not.

The ***"without pattern"*** solution will new an instance of the requested language which will load the appropriate dictionary each time it checks a word. Whilst this may sound like an implausible design, it is indeed a design that exists across software systems in industry today. The creation code lives within the check method and looks as follows;

```
class Proofer

  def check? (language, text)
    case language
      when :english then @speller=EnglishSpellChecker.new
      when :french then @speller=FrenchSpellChecker.new
      when :spanish then @speller=SpanishSpellChecker.new
      when :german then @speller=GermanSpellChecker.new
    end
    @speller.check? text
  end

end
```

The "with pattern" solution will create a pool of 3 instances of each language speller with the dictionaries loaded at creation time. When a check call comes it, one of the three is chosen at random to process it. In a real-world scenario, there would be queues, and callers would acquire, and release the instances to manage capacity but for the purposes of this paper we are examining the case where there is full capacity. The creation code lives in the initialize method and looks as follows;

```
def initialize
  #Create a pool of 3 spellers for each language
  @english0=EnglishSpellChecker.new
  @english1=EnglishSpellChecker.new
  @english2=EnglishSpellChecker.new

  @french0=FrenchSpellChecker.new
  @french1=FrenchSpellChecker.new
  @french2=FrenchSpellChecker.new

  @german0=GermanSpellChecker.new
  @german1=GermanSpellChecker.new
  @german2=GermanSpellChecker.new

  @spanish0=SpanishSpellChecker.new
  @spanish1=SpanishSpellChecker.new
  @spanish2=SpanishSpellChecker.new
end
```

All the code is the same between the two solutions except the file "proofer.rb" which managed the creation of the instances according to the model used.

Each test run will make 1000 check calls. The test pass will measure 10,000 test runs.

Expected Performance Characteristics

Whilst it could be argued that for many design patterns primary focus is not on performance of run time execution; the object pool pattern is one where you might expect some performance gains. By using a pool of pre-created object as opposed to new-ing and deleting (or collecting) them as they are used should result in better performance in many large-scale scenarios.

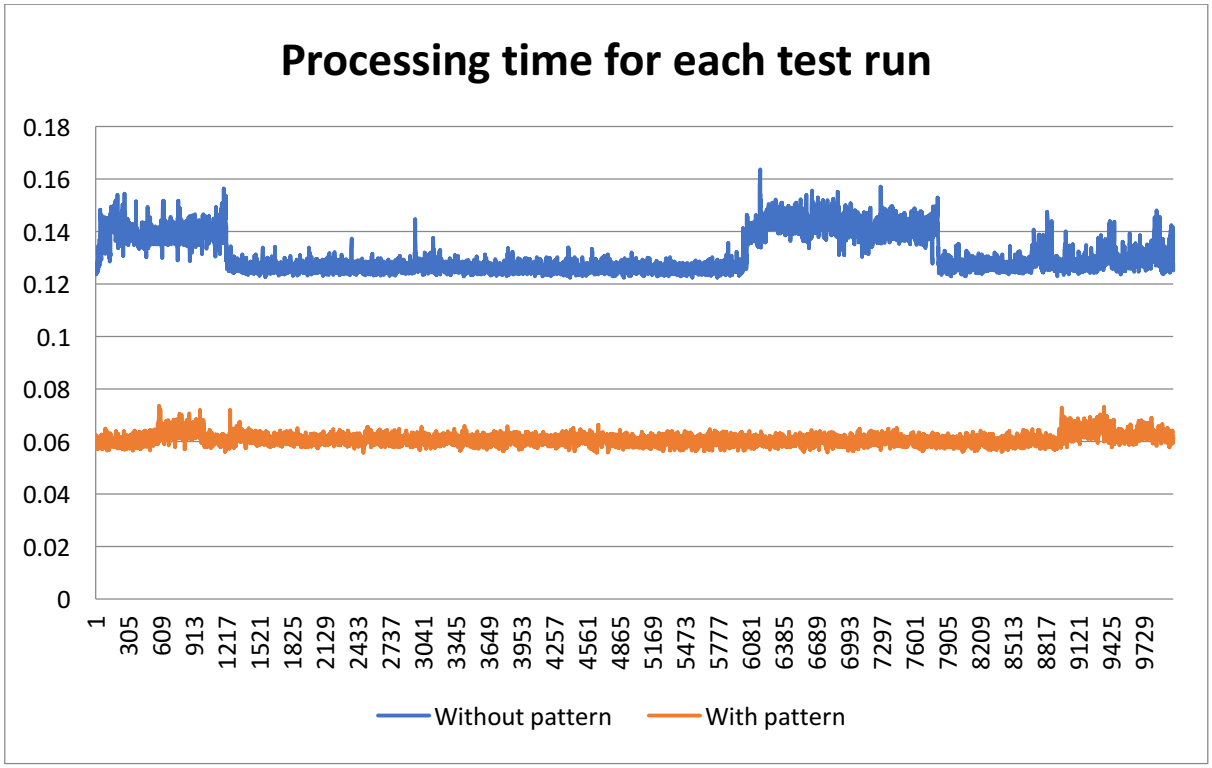
Memory usage could also be important here. As the solution without the pattern implemented is constantly new-ing the objects and allowing the garbage collector to destroy them the memory pattern should be less static than the solution with the pattern implemented.

Experiment Results

For processing time the results are as follows;

Solution Type	Total Time (seconds)	Average Test Time (seconds)	MAX Test Time (seconds)	MIN Test Time (seconds)
With Pattern	607.6155	0.0608	0.0737	0.0556
Without Pattern	1311.5144	0.1312	0.1637	0.1224

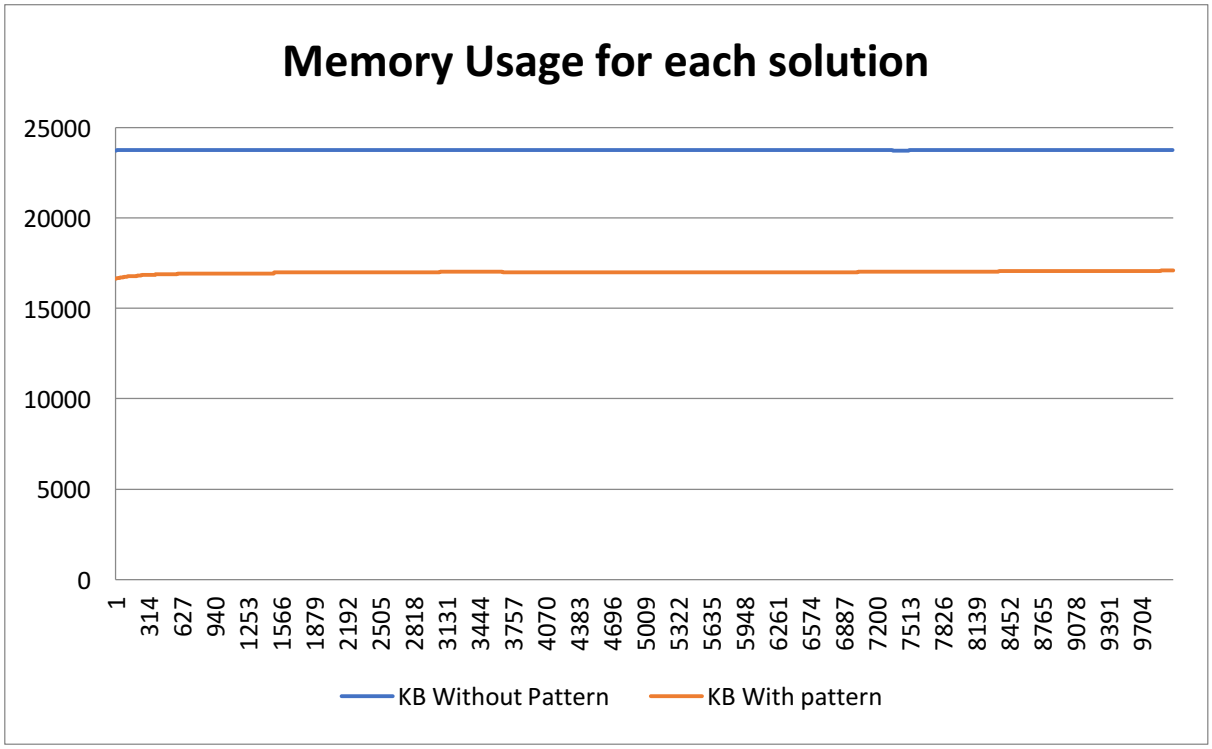
Plotted graphically all test runs look as follows;



When we look at memory usage, the results are as follows;

Solution Type	Average Mem Usage (KB)	MAX Mem Usage (KB)	MIN Mem Usage (KB)
With Pattern	16993.7096	17080	16636
Without Pattern	23734.8736	23736	23724

Plotted graphically for all test runs the results look as follows;



Conclusion

The solution with the pattern implemented runs consistently faster that the solution without. This appears to support the hypothesis that the use of an object pool increases the run-time performance of the code. Also in terms of memory usage we see the solution with the pattern implemented needed less memory.

One additional thing that is interesting in the memory test passes is that the amount of memory needed for the ***"without pattern"*** solution is that the memory usage is close to constant throughout the 10000 test runs. The max and min values vary by only 12 KB. However, the "with" solution sees memory grow over the duration of the test runs with a variance of 444KB between min and max values. This would be an interesting pattern to examine further outside of the paper.

In the case of both processing time and memory usage, the initial hypothesis that the Object Pool design pattern is more efficient is confirmed in the experiment.

General Conclusions

Whilst this may serve as an introductory examination of some common design patterns and their alternative solutions, it is not more than that. More scientific rigor would need to be applied and several solutions of each pattern could be considered to give a more rounded result. For several of the solutions the initial comment does not hold true;

Design patterns have poorer run-time performance characteristics than comparable solutions without them.

In several of the solutions the ***"with pattern"*** solution performs better when measuring CPU time and / or memory usage. The following table shows the summary results for the five patterns examined.

Pattern	Best CPU perf	Best Memory Perf
Singleton	Equal	Equal
Template Method	Equal	Without pattern
State	Without pattern	Inconclusive
Decorator	With pattern	Without pattern
Object Pool	With pattern	With pattern

As with all software solutions it is a skill to know the right approach to take to solve a problem. Performance, resource usage, maintainability, readability, extensibility and many other factors can lead a developer down a specific solution, and a knowledge of the right design pattern usage would really aid a developer in choosing the most appropriate solution for each problem.

Appendix

References, Citations and Sources

The following resources were used in the creation of this paper;
The main text for the module “Design Patterns in Ruby” by Russ Olsen was used to provide some of the example code for some of the patterns examined.

Several intents and descriptions have been taken from the following web site which was shared during the practical sessions of the module.
https://sourcemaking.com/design_patterns