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

## Introduction

It is well understood that design patterns provide a lot of value in the creation and refactoring of software solutions. This statement is re-enforced with the recognition that most class libraries delivered in 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. This pattern is replicated across many class libraries and core language features as demonstrated with Observable and Singleton mixins for Ruby.

It is highly conventional to consider knowledge of design patters 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 side effects of several of the GoF patterns and we will measure some run-time characteristics of these patterns.

## General Downsides to Overuse of Design Patterns

Before looking at specific patterns, it could be useful to look at some of the general downsides to a pattern based mind set.

TODO – fill this in

## Performance

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 are clearly very good things however you could imagine that a bi-product of these strengths could be run-time performance. This could be considered from three measurements;

1. Execution time (measured using ruby-prof)
2. Processor utilisation for the solution (measured using top)
3. Memory usage for the solution (measured using top)

As these factors are completely measureable, 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, a hypothesis will be made with respect to one or more of these measurements and the experiment will examine whether this hypothesis holds true.

## Considerations of this paper

In this paper, we will also look at some of the GoF patterns from two dimensions;

1. Potential consequences and downsides to applying the pattern in certain context or indeed in general
2. An experiment to compare the performance measures for that pattern

## Experiment Conditions

For each pattern, we will write some very simple code that can be represented with the pattern (“present”) or without the pattern (“absent”). A new VM will be created for each pattern with only the operating system (Ubuntu) and RubyMine installed above that OS may run. The solutions will then be profiled using 1000 run executions and the results will be compared. We will look at max, min and mean values for each measure. For each individual pattern examined there will be a hypothesis with respect to the measures made before the runs and this will be analysed following the batch of executions.

## Experiment Code Location

All the code used 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 particular example with the pattern implemented and without the pattern implemented.

## Experiment 1 – Singleton

### Overview

Singleton is one of the very simplest of design patterns.

### General Consequences

### Experiment Code

The test for this pattern will be as follows; The Kangaroo game scenario that was submitted as part of the final lab report following the course has the grid implemented as a singleton. This will be the experiment for the Singleton pattern. A loop will be added to run the code 10 times per test run. In state “Present” the code will have the Singleton Pattern implemented, in state “Absent” the code will new the grid object each run.

### Expected Performance Characteristics

Initial hypothesis is that the use of Singleton will have performance gain in this scenario. As the code is being looped 10 times, the use of Singleton will avoid the new-ing of grid objects and the additional memory needed to store these (pre garbage collection).

### Experiment Results

## Experiment 2 – Template Method

### Overview

The intent of the Template method is

TODO

### General Consequences

### 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 non-pattern implementation will use *if statements* to determine the actual output in an inline manner such as the following lines;



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.

To get meaningful results, we will examine the test run after 1000 iterations in each case.

### 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

On a single run the following characteristics were measured for each case;

|  |  |  |
| --- | --- | --- |
| Measure | With Pattern | Without Pattern |
| Run time | 0.122999 seconds | 0.111681 seconds |
| Individual Methods | 26 | 14 |
| Total Method Calls | 83053 | 76013 |

For this first test run the results corroborate the hypothesis in that the implementation with the pattern takes longer to run and has a higher number of methods and consequently a higher number of method calls.

## Experiment 3 – Strategy

### Overview

### General Consequences

### Expected Performance Characteristics

### Experiment Code

### Experiment Results

## Experiment 4 – Decorator

### Overview

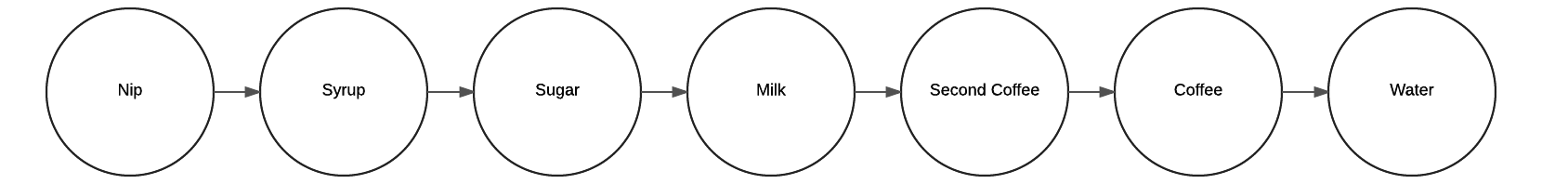
### General Consequences

### Expected Performance Characteristics

### 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.

### Expected Performance Characteristics

As each component using the pattern is a class, there will be additional creation time needed to initialize these.

### Experiment Results

# Appendix

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.