DOI: 10.2507/29th.daaam.proceedings.130

PERFORMANCE IMPACT OF USING ABSTRACTIONS IN OBJECT-ORIENTED PROGRAMMING

Vedran Juricic & Matea Radosevic

This Publication has to be referred as: Juricic, V[edran] & Radosevic, M[atea] (2018). Performance Impact of Using Abstractions in Object-Oriented Programming, Proceedings of the 29th DAAAM International Symposium, pp.0901-0907, B. Katalinic (Ed.), Published by DAAAM International, ISBN 978-3-902734-20-4, ISSN 1726-9679, Vienna, Austria
DOI: 10.2507/29th.daaam.proceedings.130

Abstract

The abstraction is a common mechanism that developers use when developing applications. It allows their code to be more robust and easier to maintain because it provides a loose coupling between application's components. This paper analyses an impact of using abstractions on application execution and its performance. Tests were devised so that one can measure an execution time when only concrete classes are used, when interfaces or implementation are used and when abstract classes or inheritance are used. This paper also presents an analysis of a special case of using interfaces, in which the objects are not directly instantiated, but instead their creation is controlled by an Inversion of Control container.

Keywords: OOP performance; abstraction performance; dependency inversion; inversion of control

1. Introduction

Object-oriented programming is a programming paradigm based on the concept of objects. Objects can have a state defined by its attributes, which are actually internally stored values, like fields. Objects can also have a set of behaviours that represent activities associated with the object and are usually represented as methods [1]. There are four requirements of a language to be considered into this paradigm. Those requirements are known as pillars of object-oriented programming and the requirements are abstraction, encapsulation, inheritance and polymorphism [2].

Abstraction refers to the ability to define and then use complex structures or operations in ways that allow many details to be ignored [3]. It is probably the most important mechanism and the hardest one to master. It greatly affects a resulting programming code, its drawbacks and advantages, and it requires a great deal of experience in order to well interpret real world problems and objects and then to design a quality programming code [14].

In order to achieve that, experienced developers study and apply the existing design patterns [4, 5]. Design patterns are guidelines or partial solutions to common occurring problems in software design and are documented so they are not tied to a specific problem. One of the most common patterns in modern web application development is the Model-View-Controller (MVC) pattern, which defines the responsibility of those three components and the interaction between them, but it can also be modified and used as a base for rapid web application development [15]. Design patterns are divided into three categories, based on a problem they solve: creational that deal with class instantiation problem, structural that deal with class and object composition, and behavioral patterns that solve problems with objects communication [4].

Those patterns heavily rely on abstraction and inheritance, and great majority of them are based on abstract classes and interfaces. Developers also aim to write source code that complies with five principles of agile software development, often referred as SOLID principles [6]. Those principles are: Single-Responsibility Principle (SRP), The Open-Closed Principle (OCP), The Liskov Substitution Principle (LSP), The Dependency-Inversion Principle (DIP) and The Interface-Segregation Principle (ISP). The Dependency-Inversion Principle is based on two claims:

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend upon details. Details should depend upon abstractions.

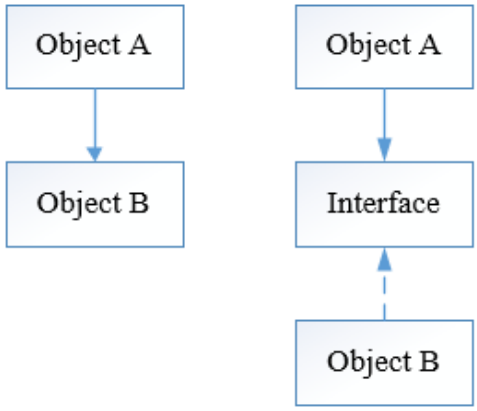

Fig. 1. Dependency-Inversion Principle

As we can see, this principle also relies on abstractions. Fig. 1 shows Object A, a high-level object that depends on a low-level object (Object B) to achieve its task. Left side of Fig. 1 shows a direct usage of Object B, while right side shows a state after applying the Dependency-Inversion Principle, where a direction of Object B arrow is inverted and so both objects depend on the Interface (abstraction). The main benefits of this principle are the reusability and the separation of objects, meaning that low-level objects can be changed without affecting high-level objects.

Another important principle is the Inversion of Control (IoC) principle or the Dependency Injection [7]. It is based on creating a third object (called Assembler), that provides objects an appropriate implementation of required interfaces. In Fig.2, Object A does not create Object B, but asks the Assembler to create valid implementation of the required Interface. Assembler is usually a part of an entire framework that provides the Dependency Injection, called the Inversion of Control Container. Containers allow us to describe Dependency Injection constructs or rules at development time and then automatically inject dependent objects at run-time, based on those descriptions [16].

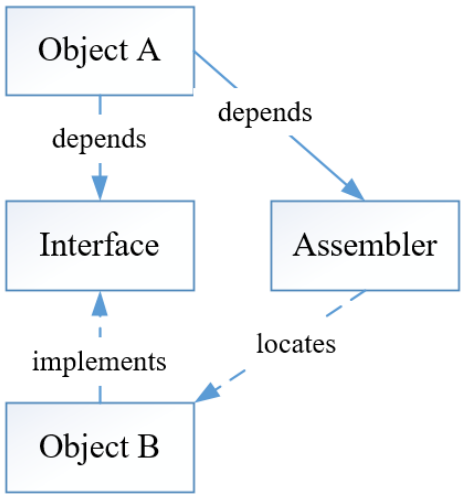

Fig. 2. Dependency Injection

This paper analyses the effects of using abstractions in an application development on the application performance, by comparing data obtained in a case without abstraction to a case where abstractions are used. Also, it analyses changes in the application performance when using some of the most popular Inversion of Control containers.

2. Method and analysis

We developed a simple C# application that enabled us to test all the above-mentioned cases. The class that is tested is called a Logger, which contains one method that outputs a text to the Console (standard output stream). In each test, the method was executed 100 million times. Since invoking a Logger method takes only 10^{-9} s, it is necessary to have such a great number of method executions, in order to obtain longer test execution times and to provide a quality analysis. Each test was then repeated 100 times to eliminate the influence of random processor activities.

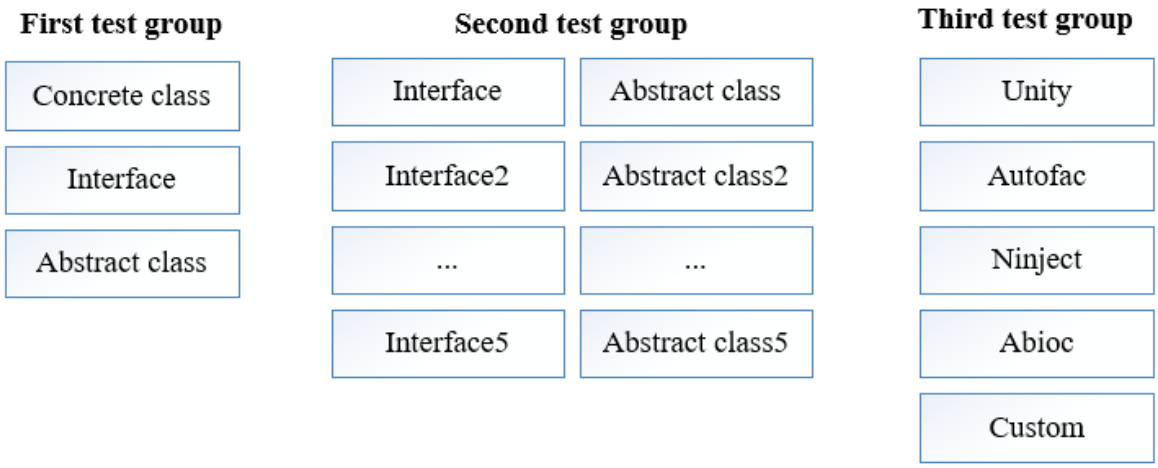

Fig. 3. Tests overview

Fig. 3 shows all tests that were performed within this paper. Tests were first made on the application that does not use any abstractions (concrete class test). Then the tests were performed on the code that was modified so that methods return and accept an interface (interface test). Finally, the interface was replaced by an abstract class (abstract class test).

Second group of tests analysed a performance impact of using interface and abstract class inheritance. Tests were made after introducing a new interface Interface2 that was inherited by the original interface. After that, we introduced a new interface Interface3 which was inherited by Interface2, etc. Last interface we tested was Interface5. Later then we repeated this hierarchy with abstract classes.

Third group of tests analysed the effect of introducing Inversion of Control containers. An IoC container was used to resolve interface implementation at run time and was queried each time a test needed a Logger implementation. We could not find relevant information about IoC containers popularity or usage, so we have included first three containers based on the number of downloads from NuGet site [8]. These are: Unity [10], Autofac [11] and Ninject [12]. We have also included Abioc [13] container that has shown a very good performance in a study called IOCBattle, the results of which are publicly available on GitHub [9]. We have made and tested our own simple IoCContainer (Custom) that is based on a dictionary for storing objects or delegates which are used to create them. A list of tested containers is shown in Table 1.

IoC container	Reason for inclusion
Unity	9 million downloads
Autofac	8.8 million downloads
Ninject	6.7 million downloads
Abioc	The fastest
Custom	The simplest

Table 1. Tested IoC containers

All tests suits were performed two times: a new object is created with each method call (transient, per request) and all method calls are performed on the same object (singleton). Tests were run on an Intel Core I7-5820K CPU with 16 GB RAM.

3. Results

More than 30 tests were performed for the purpose of this study, not counting test repetitions. Total running time of all tests is more than 24 hours. This chapter describes and interprets the results of each test group from Fig. 3.

3.1. Concrete class, Abstract class and Interface test

First test that was performed analysed differences in performance between concrete class, interface and abstract class tests. The results are shown in Fig. 4 that shows the average test durations in milliseconds for transient and singleton cases.

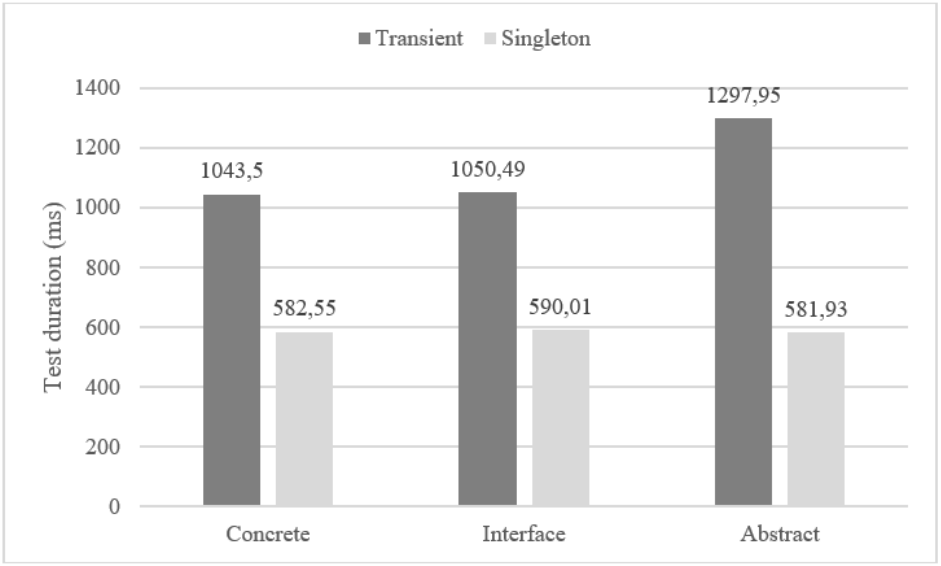

Fig. 4. Performance of concrete class, interface and abstract class

It can be seen that in the singleton case exists only a small difference between values for concrete class, interface and abstract class. The biggest difference is between interface and abstract class, and equals 8ms. We have made the two-tailed t-test using which we wanted to show if there is a statistically significant difference between those values. Results are shown in the first column of Table 2. As we can see, all P-values are above the significance level (0.01) and so we can accept H_0 (no statistically significant difference). The reason is obvious; all method calls use the same object that is instantiated at the beginning of the test.

In the transient case, i.e. when the object is created for each method call, the values for concrete class and interface are about the same. The middle column of Table 2. shows the t-test results. P-value is above the critical value thus we can conclude that the difference between values is not statistically significant.

Singleton mode	**Transient mode**	
H_0: The difference between concrete class, interface and abstract class mode is not statistically significant	H_0: The difference between concrete class and interface is not statistically significant	H_0: The difference between concrete class and abstract class is not statistically significant
concrete class and interface t = -2.42, df = 189 p = 0.016, p > 0.01 concrete and abstract class t = 0.18, df = 198 p = 0.857, p > 0.01 interface and abstract class t = 2.59, df = 188 p = 0.011, p > 0.01	t = -1.72, df = 192 p = 0.087, p > 0.01	t = -55.18, df = 195 p = 0, p < 0.01
We accept H_0 hypothesis	We accept H_0 hypothesis	We do not accept H_0 hypothesis

Table 2. t-test results

As we can see from the third column in Table 2, we cannot accept H_0 hypothesis based on the calculated P-value. We accept the alternative hypothesis that the difference between values is statistically significant. The test with abstract classes in average lasted 24.4% longer (from 1043.5 ms to 1297.95 ms). We assumed that the reason for this lies in a fact that an instantiation of a concrete class which inherits an abstract class requires two constructor calls, one for each class.

3.2. Abstract class and interface inheritance

In order to test the previous assumption, we have created a hierarchy of abstract classes and interfaces. Tests were made after introducing each level of hierarchy, as it can be seen in Fig. 3 (second test group). The results of the analysis are shown in Fig. 5. Vertical axis shows test duration in milliseconds, while horizontal axis shows the level of inheritance. For example, abstract class 1 means that a concrete class inherits only one abstract class. Abstract class 2 means that a concrete class inherits an abstract class, which also inherits another abstract class, etc. The same logic applies for interfaces.

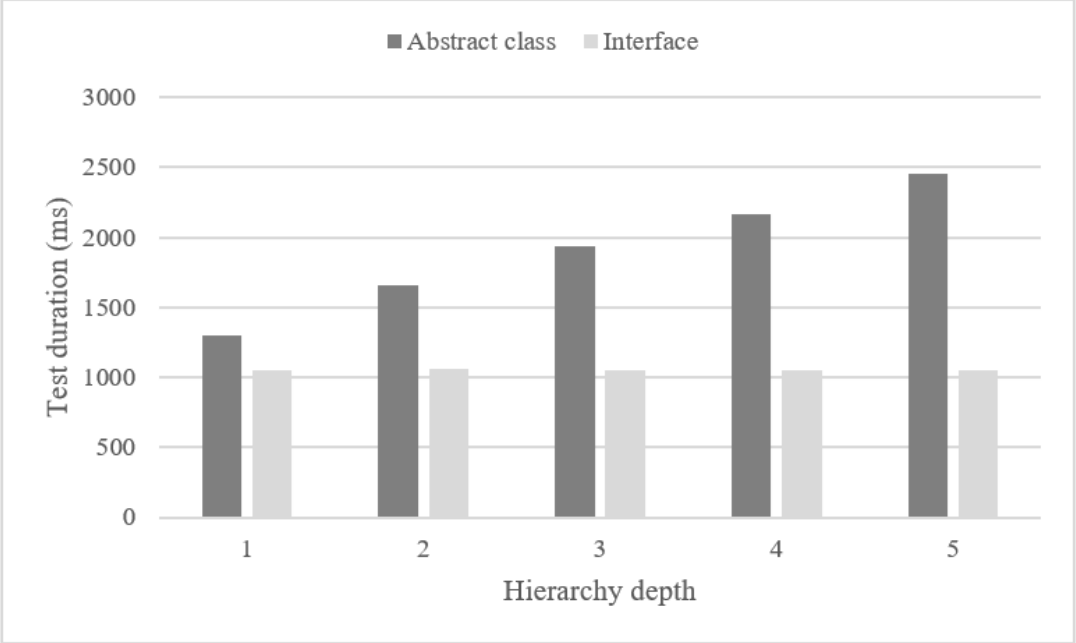

Fig. 5. Performance of abstract class and interface inheritance

As we can see from Fig. 5, interface inheritance does not have a significant impact on the application performance. Test duration for interfaces is almost unchanged. An average difference between levels is only 4 ms and it can be ignored. On the other hand, the abstract class test shows a significant difference between test durations that are proportional to the hierarchy depth. On average, the test is 287.96 ms longer each time a new level in hierarchy is introduced, and so, a difference between the fifth and the first level is 1151.83 ms.

We assumed a linear dependency between the test duration and the hierarchy depth, which can be expressed by the following equation: t = a x h + b, where *t* is the test duration, *h* is the hierarchy depth and *a* and *b* are coefficients. Using the Least Square Method, we calculated that the values of the coefficients *a* and *b* are 281.7 and 1057.3 respectively. Equation allows us to approximate the test duration for an arbitrary hierarchy depth. For example, the expected test duration with 10 abstract classes is 3874.3 ms.

3.3. Performance of IoC containers

We have tested the performance of five IoC containers, as shown in Table 1. The results of our analysis are shown in Fig. 6, where the vertical axis shows test durations in seconds, while the horizontal axis shows IoC containers. We have also repeated tests in transient and in singleton mode.

It is interesting to point out that there are no significant differences between test durations in the transient and the singleton mode, except for the Ninject container, where the singleton mode is 48% faster. It is unlikely that containers are optimized in a way that provides equal execution times when object is instantiated multiple (10^8) times and when it is instantiated only one time, in the singleton mode. We can conclude that tested IoC containers (except Ninject) are not optimized when running in the singleton mode.

As we can see from Fig. 6, Abioc is the fastest IoC container, with approximate test duration time of 4.4 s. Autofac and Unity are in the middle, with test duration time of 43.7 s and 58.4 s respectively. Ninject is the slowest IoC container, with transient time of 294.7 s. Our results are in a complete accordance with results obtained in IOCBattle study [9]. Although our custom implementation shows a good performance (test duration of 11.7 s), it does not have a full IoC functionality and is only able to instantiate an object for each request or to return the previously instantiated object in singleton mode.

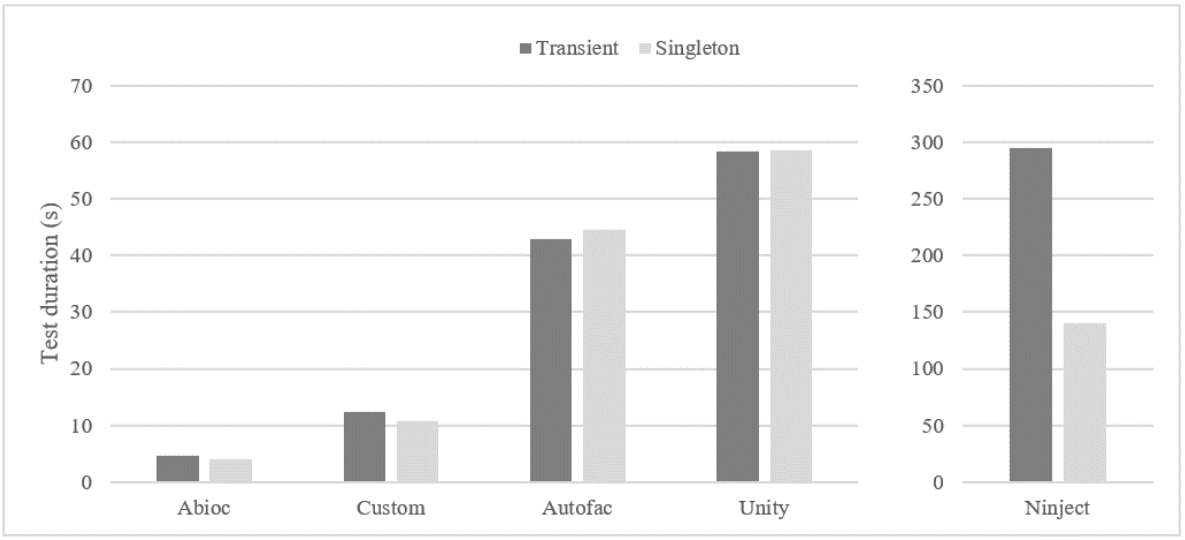

Fig. 6. Performance of tested IoC containers

4. Limitations

It is necessary to mention three limitations of our study and its results. Firstly, data used in analysis was obtained by performing tests on only one technology. The application we used for testing was written in C# programming language and was running under Microsoft .Net Framework. We would achieve more significant results if tests were repeated for other programming languages and technologies.

Secondly, results that are presented are based on the performance of only one problem; a logging component that was used throughout the application. Although the application suits its purpose, it would be interesting and useful to perform tests on a real or a more complex problem. We will address both limitations in our further work.

It must also be emphasized that the results in our study are based on tests with 100 million class instantiations. The majority of applications are not even close to that number and they are not affected by the changes described in this paper. Impact on the performance is expected to become noticeable in applications with an intense or an extreme load.

5. Conclusion

The general guideline when developing in object-oriented programming languages is to uncouple major application components or classes by using abstractions. The aim of this paper was to analyse possible differences in performance of such uncoupling, i.e. the difference between concrete classes, abstract classes and interfaces. The performance was tested by using a custom-made application that provided a mean to separately measure duration of all the above-mentioned cases. Obtained data was then compared and statistically analysed.

We have shown that there does exist a difference in the application performance when using interfaces and abstract classes. In case of abstract or concrete class inheritance, performance depends on the hierarchy depth, where higher inheritance level means a slower application execution.

We have also tested performances of four most popular IoC containers. They obviously differ in their functionality, possibilities, ease of usage, but also in their performance. We have shown that one of the most popular IoC container, Ninject, is actually the slowest one.

The results obtained in our study can be used as guidelines when designing application components and help with the decision whether to use abstractions, interfaces and abstract classes, or not. Our further work includes the analysis of performance of different Dependency Injection mechanisms, like constructor injection, method injection, etc., but we are also planning to test a difference between application component that applies design patterns (e.g. Abstract factory, Facade, Command pattern) and the one that does not apply design patterns.

6. References

[1] Lewis, J., Loftus, W., & Tahiliani, M. P. (2009). Java software solutions: foundations of program design. Pearson/Addison-Wesley.

[2] Wegner, P. (1990). Concepts and paradigms of object-oriented programming. ACM Sigplan Oops Messenger, 1(1), 7-87.

[3] Robert, W. S. (2018). Concepts of programming languages. Pearson.

[4] Vlissides, J., Helm, R., Johnson, R., & Gamma, E. (1995). Design patterns: Elements of reusable object-oriented software. Reading: Addison-Wesley, 49(120), 11.

[5] Schuster, S., & Schulze, S. (2012, September). Object-oriented design in feature-oriented programming. In Proceedings of the 4th International Workshop on Feature-Oriented Software Development (pp. 25-28). ACM
[6] Martin, R. C., & Martin, M. (2006). Agile principles, patterns, and practices in C. Pearson Education.
[7] https://martinfowler.com/articles/injection.html (2012). Accessed: 2018-02-02
[8] https://www.nuget.org/ (2018). Accessed: 2018-02-03
[9] https://github.com/MartinF/IOCBattle/tree/master/IocBattle.Benchmark (2012). Accessed: 2018-02-05
[10] https://github.com/unitycontainer/unity (2018). Accessed: 2018-02-05
[11] https://autofac.org (2013). Accessed: 2018-02-05
[12] http://www.ninject.org (2012). Accessed: 2018-02-05
[13] https://github.com/JSkimming/abioc (2017). Accessed: 2018-02-05
[14] Konecki, M. (2014). Problems in programming education and means of their improvement. DAAAM international scientific book, 459-470.
[15] Pop, D. P., & Altar, A. (2013). Designing an MVC Model for Rapid Web Application Development. Annals of DAAAM & Proceedings, 24(1).
[16] Hudli, S. R., & Hudli, R. V. (2013). A Verification Strategy for Dependency Injection. Lecture Notes on Software Engineering, 1(1), 71.