How Does the Implementation of the Proxy Pattern in Python Affect Flexibility as Measured by DAM, MOA, NOP and DCC?

Oscar Johansson
TDDDB84: Design Patterns
Linköping University
Linköping, Sweden
oscjo411@student.liu.se

Abstract— This paper compares two implementations that each contains two proxy patterns, one in C++ and one in Python. This is done to investigate how the implementation in Python affects flexibility, as measured by the software metrics DAM, MOA, NOP and DCC. The implementation in Python used duck typing instead of a common interface. The result indicates that the implementation in Python increase flexibility, one big factor to this is the use of duck typing. But that there are some threats to validity and the paper suggests that there is a need for further investigation, especially with a larger code base.

I. INTRODUCTION

In “Design Patterns: Elements of Reusable Object-Oriented Software” [1] the authors introduce 23 design patterns. A design pattern is a reusable solution to a common problem in object-oriented programming.

One of the patterns introduced is the proxy design pattern. The proxy design pattern acts as a wrapper and controls access to an object [1]. One area of use for this pattern is in the field of ubiquitous computing. In ubiquitous computing, where many computer devices should be able to share information and work together, flexible software with loose coupling is required [2, 3, 4]. This can be accomplished by using the proxy pattern [3].

Python is the most popular dynamic programming language [5, 6]. This language is flexible [7] and has some interesting mechanisms to let objects interact, one of which is duck typing [8].

J. Bansiya and C. Davis [9] define flexibility, using a formula, as the interaction between encapsulation, coupling, composition and polymorphism. An increase in encapsulation, composition or polymorphism is defined to increase flexibility, while coupling does the opposite. To measure the four properties the metrics DAM, MOA, NOP and DCC are used, these are chosen since they are defined to measure the properties [9].

This paper will conduct research on the implementation of the proxy design pattern in Python. Since it is already established that the proxy pattern increases flexibility [3], the focus in this paper is the implementation in Python. By using some of the language specific features it will be investigated if this could increase the flexibility of the proxy pattern further. To be able to measure and compare the effect of a Python implementation, a second programming language must be chosen. The second language of choice is C++, since proxy pattern was first defined using sample code in this language [1].

The proxy pattern has four defined application scenarios: protection proxy, smart reference, remote proxy and virtual proxy [1]. Python has automatic memory management; therefore, the smart reference scenario is built into the language. In the scope of flexibility, the protection proxy is of less interest. Due to this, only the two latter scenarios, namely virtual proxy and remote proxy, will be investigated in this paper.

The implementation will be part of a software that simulate how a web browser could communicate with a web site. When a user wants to visit a site the browser will download the entire site locally. It will then display the downloaded information to the user. The information consists of text and images. The images are heavy to load for the web browser, even when they are downloaded locally. So, instead of drawing every image the browser displays a placeholder that needs user interaction to draw the real image. A remote proxy should be used to store the website locally and fetch new information from the site. A virtual proxy should be used to implement the image placeholder. The user interactions that should be considered are: refreshing the website and drawing an image.

The system described above will be implemented in C++ and Python. The flexibility will be measured with the metrics stated, and the results will be compared. It can then be determined if the implementation in Python leads to an increase in flexibility, as measured by DAM, MOA, NOP and DCC.

II. BACKGROUND

This section will begin by introducing the proxy design pattern. It will present the software quality characteristics and metrics of importance to this paper. Lastly the necessary theory about Python will be presented, where the concept duck typing gets explained.
A. Proxy Design Pattern

The proxy design pattern [1] is a structural pattern used to control access to an object. If a proxy is used to control access to an object, then all communication with the object should be through the proxy [3]. The general structure of a proxy pattern can be seen in Fig. 1, here it can be seen that the client only have knowledge about an interface (Subject) and that Proxy delegates calls to RealSubject.

![UML Class diagram showing the structure of a proxy pattern](image)

Fig. 1. UML Class diagram showing the structure of a proxy pattern [18].

There are four typical scenarios where the use of a proxy pattern is applicable [1]:

1) Protection proxy
When there is a desire to restrict the access to an object depending on what access right the visiting object have.

2) Virtual proxy
When the creation of the object is regarded as expensive and should only be done when needed. In this scenario, the proxy will oversee the initialization of the object when another object request something through the proxy.

3) Remote proxy
When the object is stored in place that is regarded as expensive to fetch information from. In this scenario, the proxy will store a local copy of the object and will only fetch new information when needed.

4) Smart reference
When there is a need to handle the references to an object. This could include: counting references, free the memory of the object when no references exist and only allowed one other object at a time to access the object.

A proxy should have the same interface [10] as the object it is controlling access to. Depending on purpose the proxy can handle the requests differentially. For example, in the protection proxy the access could be denied if certain criteria are not met. Using the proxy pattern will decrease coupling between the object and its users [11].

B. Software Quality:

Software quality is a measurement of how well a software satisfies the needs of the user [12]. The software quality consists of different software quality characteristics, which differ from different quality models. The only characteristic used in this paper is flexibility.

1) Flexibility
Flexibility, as defined by J. Bansiya and C. Davis [9], is a characteristic measuring the ability for a software to adjust to change in existing design or incorporation of new design. The flexibility of a software, where higher is better, can be calculated with the formula in (1) [9].

\[
\text{Flexibility} = 0.25 \cdot \text{Encapsulation} + 0.25 \cdot \text{Polymorphism} + 0.5 \cdot \text{Composition} + 0.5 \cdot \text{Coupling}
\]  

2) Encapsulation/DAM
Encapsulation is a mechanism to wrap and restrict access to components underlying implementation. Instead of letting objects modify the implementation an interface is provided.

To measure the encapsulation the metric Data Access Metric (DAM) is used. The DAM for a class is the ratio between number of private attributes and the total number of attributes, the formula to calculate the ratio can be seen in (2) [9].

\[
\text{DAM} = \frac{\text{Number of private attributes}}{\text{Total number of attributes}}
\]

3) Coupling/DCC
Coupling is a term that describes the strength of relationships between different modules/classes [13]. Low coupling, weak relationship between modules, is desired.

To measure the coupling the metric Direct Class Coupling (DCC) is used. The DCC of a class A is the number of other classes that A are related to [9].

4) Composition/MAO
Closely related to aggregation, the composition property measures how much of a class that is composed by other objects [14].

To measure the composition of a class the metric Measure of Aggregation (MAO) is used. The metric counts how many user defined classes that are used inside an object. [9]

5) Polymorphism /NOP
The property measures the ability to use different objects with a shared interface without knowing the concrete class it is handling.

To measure polymorphism the metric Number of Polymorphic Methods (NOP) is used. The metric is the number of methods that can use polymorphic objects. [9]

C. Python

This section will cover the language specific features, in Python, that are of interest for this paper. Since the code will be compared with code written in C++ this section will also point out language differences that are of significance.

1) Private variables
Technically there are no private variables that can only be accessed within the object in Python. But there is a convention that if a variable name starts with an underscore it is regarded as private [15]. If a class variable starts with two underscores then
Python’s name mangling algorithm will be performed, making it difficult to access outside the class [16].

In this paper a class variable with one or more leading underscores are considered private and will be treated as the equivalent to C++’s private variables.

2) Duck typing

In duck typing the class of an object is not considered when performing calls to the object’s methods or members [8]. The code presented in Fig. 2 is valid in Python, because of duck typing. Duck typing is heavily used in dynamic programming [20]. Without duck typing it would be needed to check the type of the object duck before calling the function quack().

```python
def make_duck_quack(duck):
    duck.quack()```

Fig. 2. Example of duck typing.

Duck typing is considered to be a type of polymorphism [17].

III. IMPLEMENTATION

In this section the implementation to the scenario introduced in Introduction will be presented. First the general implementation details will be presented, using a UML class diagram. Then, when the general structure has been established, the implementation in code will be presented. The code in both C++ and Python will be following the general implementation structure, but taking advantage of the code specific features each language has to offer.

The code for the implementations, for both languages, is available to view at GitLab¹.

A. General

As described in the introduction, the scenario includes displaying information from a web server using a browser. The information from the web server is expensive to fetch and thus a local copy of the information is desired. The information consists of text and images. Compared to the text, the images are heavy to load. The browser wants to store the local information, but only load the images when the client want to see the information. A remote proxy should be used to store the local copy of the information. A virtual proxy should be used when representing the images.

In Fig 3, an UML class diagram is presented showing the structure for the general solution to the scenario. Client (user) communicates only with Browser, that will perform requested actions. Browser displays and stores Information that it fetches from WebSite. WebSite is an interface for WebServer and WebServerProxy. WebServerProxy is a remote proxy, fetching Information from WebServer when needed. Information is an interface for Text, Image and ImageProxy. ImageProxy is a virtual proxy, that only initializes an instance of Image when Browser calls the draw-function.

```cpp
std::vector<Information>* info;

void Browser::displayInfo()
{
    for (int i = 0; i < info.size(); i++)
    {
        std::cout << info.at(i)->draw() << "\n";
    }
}
```

Fig. 4. Code showing how the browser displays Information without knowledge of the concrete classes.

Each class that contained an instance of another class, needed to have knowledge about that class. For example, WebServerProxy needed to have knowledge about how WebServer represented the Information, in this case a vector, see Fig. 5.

¹ https://gitlab.ida.liu.se/oscjo411/TDDB84-paper-code
The use of Python could increase this ratio significantly. The ratio was calculated using (2).

Every variable declaration that consisted of another user declared class was counted.

Every function that could use polymorphic behavior was counted. Note that duck typing is a type of polymorphism, this will affect the metric. If a class in Python receives an object on which it does not know its concrete class and then performs operation that is specific for that class (for example calling its draw-function) it will count as polymorphic behavior.

Every class is measured with the metrics described above. The average value for each metric in each language is presented in Table 1. These values have been used to calculate the flexibility for each language, using (1). The flexibility for Python and C++ is also presented in Table 1.

Table 1. Result of software metric measurement

<table>
<thead>
<tr>
<th>Software Metric</th>
<th>C++</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAM</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td>DCC</td>
<td>1.125</td>
<td>0.833</td>
</tr>
<tr>
<td>MOA</td>
<td>1.125</td>
<td>0.833</td>
</tr>
<tr>
<td>NOP</td>
<td>0.125</td>
<td>0.500</td>
</tr>
<tr>
<td>Flexibility</td>
<td>0.594</td>
<td>0.708</td>
</tr>
</tbody>
</table>

V. ANALYSIS

The implementation scenario was created so that the use of a virtual proxy and a remote proxy was required. Using the written scenario, the UML class diagram (Fig. 3) was drawn to display how a general solution using the proxies could be structured. With the help of the diagram the implementations in both languages were written. Both implementations became rather short. The metrics used, counted and measured certain aspects of the code. The average of this measurement for each class in both languages were calculated. Since the implementation became short a small change in the code could affect these metrics significantly. The decision that the implementation in C++ should use interfaces (abstract classes) affected the calculation of the average of every metric, since it added two more classes. This shows a big problem with this measurement. A minor change in a design decision, that does not affect the use of a proxy pattern, could result in a big change in the flexibility. [19] states that without knowing the context, it is not possible to conclude the effect of a design pattern on a software quality. Perhaps the context in this paper, namely a small code base, affected the software quality considerably.

In the field of ubiquitous computing, as mentioned in the Introduction, flexible software with loose coupling is required [2, 3, 4]. It was stated that this could be accomplished by using the proxy pattern [3]. The use of Python could increase this further, but how this result would scale with the code size is still unclear. There is a need for further investigation on how the

```cpp
class WebServerProxy: public virtual WebSite {
public:
    WebServerProxy();
    void refresh();
    std::vector<Information*> getInfo();
private:
    std::vector<Information*> local;
    WebServer server;
};
```
proxy pattern in Python could affect flexibility when the code base is relatively large.

The most significant difference between the implementation in Python compared to the implementation in C++ was the use of duck typing. With duck typing there is no need to declare the type for a variable. This results in more polymorphic behavior, which contributes to more flexibility. With this argument, the use of Python with duck typing, should result in an increase of flexibility. [20] shows in an empirical study that programs written in dynamic languages make heavy use of duck typing, enforcing this claim further.

The choice of metrics was made regarding how flexibility was defined and on how these software characteristics were measured [9]. The choice of DAM, DCC, MOA and NOP still seems appropriate. In this paper, DAM turned out to be unnecessary since no public attributes was used. This shows that the proxy pattern does promotes encapsulation, but that was not the purpose of this paper. However, as concluded in [9] encapsulation does affect flexibility, making it necessary to measure.

VI. CONCLUSION

The research problem for this paper was if the implementation of the proxy pattern in Python would influence flexibility as measured by DAM, MOA, DCC and NOP. As can be seen in Table 1 the measured difference in flexibility is relatively small. The result indicates that the flexibility increases when implemented in Python as measured by DAM, MOA, DCC and NOP. The use of duck typing is important to increase the flexibility. One threat of validity for the result in this paper is the conclusion that a minor change in a design decision could affect the flexibility considerably.

VII. REFERENCES


[16] The Python Tutorial, Python 3.5.2 Documentation [https://docs.python.org/3/tutorial/classes.html, Retrieved 04-10-2016]

[17] Building Skills in Python, A Programmer’s Introduction to Python, Steven F. Lott, Creative Commons License, 2008


Improvements from review

I. INTRODUCTION

The reviewers gave me a grade of 4 and a grade of 5 on this section. One suggestion was to state the research problem clearly in the text, since it only was stated in the title. But during discussion at the seminar the group concluded that this was unnecessary since it would have made the text less fluent.

It was suggested that the proxy design pattern and the metrics needed to be explained further in the introduction. This was done.

II. INFORMATION QUALITY/EVIDENCE

In this category I received two grades of 4. One reviewer thought that the design pattern needed to be explained further, preferably by including a UML diagram. This was added. The concept of “Duck Typing” needed more explanation, which is something I strongly agreed with. That duck typing is a type of polymorphism needed to be explained since it is used in NOP when calculating the result.

III. ORGANIZATION

For this category I received two grades of 4. One reviewer requests a hypothesis, but at the seminar we concluded that the research paper problem is my hypothesis. Other than this no suggestions of improvements were made.

VI. LANGUAGE AND FORM

I received a grade of 4 and a grade of 5. No improvements were suggested. A colleague proof read the paper before final submit, checking for language errors.

V. ANALYSIS

For my draft, no analysis had been made. Because of this, no suggestions of improvement were made.