A Cognitive Analysis of Static and Dynamic Typing

Patrick Dubroy
pat@{my last name}.com

PSYC 5105 Final Paper
Carleton University

December 15, 2006

Abstract

This paper uses the Cognitive Dimensions of Notations framework to analyze the usability aspects of static and dynamic type systems in programming languages. It presents seven problems, in the form of patterns, that are often encountered during programming. Solutions are presented for both static and dynamic type systems, and analyzed in terms of cognitive dimensions.

Introduction

In recent years there has been an explosion in the popularity of so-called "dynamic languages", such as Python, Perl, and Ruby. Although there is no strict definition of exactly what constitutes a dynamic language, it generally refers to high-level languages that are interpreted (rather than compiled), use dynamic typing, and allow the type system to be extended at runtime. Typically, these dynamic languages also feature automatic memory management (also known garbage collection) and include powerful standard libraries.

There are several possible reasons for the current trend in dynamic languages. Of course, most obvious is the novelty factor—Python, Perl, and Ruby are all relatively new languages. Most of their features exist in two older lan-

guages, Smalltalk and Common Lisp, but those languages are not nearly as popular.

However, there are also some legitimate reasons behind the popularity of dynamic languages. Java is generally considered to be a more productive language to work in that C++, and many people have reported even greater productivity gains when working in a dynamic language. This paper is based on the hypothesis that dynamic typing may be one reason that a programmer could be more productive in a dynamic language.

The notion of "productivity" in a given language or development environment is notoriously difficult to measure [5]. Due to the nature of software development, it is extremely difficult to produce empirical measures of productivity in a given language or development environment. There is no widely agreed-upon way to measure basic properties such as the size of a program or development effort. Furthermore, there can be significant differences between the abilities of two programmers, or even of a single programmer on different days.

Instead of attempting empirical measures, this paper takes an analytical approach. If a programming language is simply easier to use, then a programmer could be more efficient and thus more productive. This paper explores the usability aspects of type systems, to see how that could affect productivity.

It begins by defining exactly what is meant by "dynamic typing", and compares it to the alternative approach of static typing. In order to provide a basis for example, the type systems of Python and Java are briefly described. I then introduce the Cognitive Dimensions framework, which is the evaluation technique that was used in the comparison.

The analysis is based on a set of seven programming problems which are presented as *patterns*. This representation was chosen for several reasons. First,

it helped keep the analysis at a high level of abstraction, instead of getting bogged down in specific examples. Second, it encourages a perspective that programming is a human-centred activity; that a type system is a means to solve programming problems, rather than an end in itself.

Each pattern presents the problem, followed by some concrete examples, and then a solution for both static and dynamic type systems. The usability aspects of the solutions are analyzed in terms of the Cognitive Dimensions framework. Finally, I identify several common themes that emerge, and present suggestions for further work.

Type Systems

One of the most visible (and religiously debated) characteristics of dynamic languages is the *type system*. The type system of a programming language defines how values and expressions are divided into types, and how those types can be manipulated [1]. The primary functions of type systems are to prevent program errors and to provide a means of abstraction.

The terms *static typing* and *dynamic typing* refer to when and how type-checking occurs. In a static type system, type-correctness is verified by the compiler, without considering the runtime values of the expressions. In a dynamic type system, some or all of the type-checking at performed runtime.

It should be noted that there is no widely-agreed upon nomenclature for categorizing type systems. For the purposes of this paper, the above definition of static and dynamic typing is sufficient. Furthermore, since we are trying to understand why a programmer might choose a dynamic language like Python over a language like Java or C++, we will be concerned with the more traditional forms of static typing as implemented in those languages. Some languages, such as ML and Haskell, have static type systems that use type inference instead of

explicit type declarations. Those type systems are not discussed in this paper, so wherever the term *static typing* is used, it should be taken to mean *explicit* static typing.

In an explicit static type system, variables are declared to be of a certain type, and the compiler will raise an error if the programmer tries to store an object of an incompatible type into that variable. For example, the following code declares a variable that will contain an integer:

```
Integer age;
```

It would be legal to store an integer into that variable, as in:

```
age = 3;
```

The compiler would raise an error if you tried to store an incompatible type into that variable. For example:

```
age = "old";
```

would be illegal, because "age", which is of the String type, is not compatible with the Integer type.

In dynamic typing, types are not checked ahead of time by the compiler, but at runtime when the variables are used as part of an expression. For example, the following code would be valid in a dynamically-typed language:

```
var age;
...
age = 3;
age = "old";
```

In this case, when the variable is declared, it is not declared to be of any specific type. This means that the variable can hold any type. Some people will incorrectly refer to a dynamically-typed language as being untyped, but that is a misnomer. The language will still check types, as in the following example:

```
age = "old";
age = age - 2;
```

If the String type does not support the subtraction operation, a dynamically-typed language will raise a type error when this code is executed. In an untyped language, no error would be raised—the code would simply executed as is, with an undefined result.

Although languages are often described to be *either* dynamically-typed or statically-typed, it is really more of a continuum than a dichotomy. The Python type system is fully dynamic; it supports no static type checking. C, on the other hand, is fully static—it performs no dynamic type checking. But other languages fall somewhere in the middle. C++ supports limited dynamic typing via the operator, and Java has relatively sophisticated support for runtime typing.

As the patterns in this paper will demonstrate, a strict, fully-static type system is simply too restrictive in practice. Most languages that use static type systems also support some form of dynamic or weak typing.

Python

Python is one of the most popular dynamic languages in use today. Python is a dynamically-typed language with first-class functions and support for object-oriented principles. In Python, variables are not explicitly declared by the programmer. Instead, they are created when they are first used. This feature is common of many dynamic languages.

Java

Java is a very popular language for development of business systems. Java is generally not considered to be a dynamic language, although it does support many of the same features, such as runtime inspection and modification of the type system. Although it is primarily a statically-typed language, Java does support rather sophisticated runtime type operations. For example, in Java, it is common to write code like this:

```
public boolean equals(Object obj2) {
    if (obj2 instanceof Project) {
        return ((Project)obj2).name == this.name;
    }
    return false;
}
```

The instanceof operator performs a runtime type check that the parameter obj2 is in fact an instance of the Project class. This check is performed in order to avoid a runtime exception being thrown on the next line, when obj2 is cast to Project.

One of the more interesting aspects of the Java type system is that primitive types (such as int, float, and char) are not true objects: they cannot respond to methods, and they are not subclasses of the Object class. In practice, this causes many usability problems with the type system. However, this is a particular quirk of Java, and not a general property of statically-typed languages. Therefore, this paper will ignore all of those issues in the analysis.

Cognitive Dimensions of Notations Framework

How do we begin to analyze the usability aspects of type systems? Typical usability analysis of programming focuses on the lowest levels of the user inter-

action [4]. This bottom-up approach can yield useful results, but it is difficult to see how these results can be applied to a high-level concept like typing.

The Cognitive Dimensions of Notations framework [3] provides a more top-down approach to the analysis of programming activities. Rather than concentrating on minute details of the user interaction, this framework provides a means to analyze the cognitive issues embodied by the language itself.

In a general sense, a programming language is nothing more than a specific notation for representing computation and data. The Cognitive Dimensions of Notations framework provides a vocabulary for discussing the usability aspects of such notational systems. This vocabulary is informed by cognitive psychology, but oriented towards the people actually building these systems, rather than cognitive psychology experts. It is intended to be a "broad-brush" analysis of the interaction between people and information artifacts. In short, the Cognitive Dimensions framework asks: does the notation adequately support the user's intended activities? If not, what can be done about it, and what trade-offs would be involved?

The core of the Cognitive Dimensions framework is the set of so-called dimensions which identify distinct aspects of usability. For example, the *visibility* of a notation refers to the amount of structure that can be seen at any one time. In principle, the dimensions are intended to be mutually orthogonal (hence the term *dimension*, in allusion to physical dimensions). In practice, many of the dimensions are closely related, and changing the notation to affect one of the dimensions usually involves trade-offs in other dimensions. For example, another one of the dimensions is *diffuseness*, which refers to the verbosity of the notation. While it may be desirable to have high visibility in the notation, that would often also require that the notation be more verbose.

Programming Activities

When analyzing the cognitive dimensions of any information system, the authors emphasize that usability is not inherent in the notation; rather, the analysis must be performed with respect to certain user tasks that are carried out.

Not only does the CDs framework provide dimensions, but it also identifies high-level activities that describe a user's interaction with the system. The two most basic activities are *searching* for specific information in the structure, and engaging in *exploratory understanding* of the structure. In a software development context, these activities would typically be undertaken by all developers, before making modifications to the system. The activities that actually change the structure are *incrementing* the existing structure; for example, by adding new code to a method; *transcribing* from one notation to another; and *modifying* the structure. Finally, *exploratory design* describes the act of incrementing and modifying the structure when the end goal is not yet known. In programming, this would typically describe the way code is written inside a method.

The activities defined in the cognitive dimensions framework are very broadly defined, as they are intended to apply to any possible notational system. It is useful to provide concrete examples of these activities in a programming language. A detailed task analysis of programming is beyond the scope of this paper, but the examples provided here are sufficient for high-level analysis.

Searching: When inspecting code that calls a certain method, the programmer might want to search for the implementation of that method. Some other examples of searching would be searching for the declaration of a variable (is it a local variable, or an instance variable?), or finding all the locations where a variable is read or stored into.

Most programming environments feature relatively advanced search features, which greatly outstrip any slight advantages of a particular notation. Therefore

this paper does not place much emphasis on the analysis of searching activities.

Exploratory understanding: When the user is engaged in exploratory understanding, he is trying to discover the structure of the system. In a system with full visibility, there would be no need for this activity, since the entire structure would be visible. In programming, this exploratory understanding typically consists of determining the class hierarchy, understanding the structure of the data in the running system, and understanding the algorithms and control flow within the system. A more concrete example of exploratory understanding is inspecting a method to discover its purpose what operations are performed, how does it modify the data structures in the system, etc. Exploratory understanding would typically involve many search activities.

In programming, exploratory understanding is one of the most important activities that the user will undertake. Before modifying a program, it is first necessary to understand the system and its structure. Typically, a programmer spends more time understanding the system than in any other programming activity.

Incrementation: Incrementation is the act of adding to the structure of the system, without modifying the existing structure. In a programming language, this could mean adding a new class to the system, or adding methods or variables to a class. In programming, pure incrementation would rarely be performed, since the ultimate goal is to change the behaviour of the system in some way. If the behaviour is changed, then the programmer has performed *modification*.

Modification: This is the end goal of all programming—to modify the behaviour of the system in some way. Concrete examples of modification would be writing code in a method, and adding a new method to a class that overrides an existing method.

Although the Cognitive Dimensions framework distinguishes between incre-

mentation and modification, in the practice of programming, there is little difference. When writing code, a programmer is generally performing both activities.

Transcription: An example of transcription would be transferring a class hierarchy from some secondary notation, such as UML, to code. Typically, when going from a design to code, there would be some amount of transcription, and some amount of modification.

Exploratory design: Exploratory design is the act of performing modification and incrementation without a clear end goal in mind. The structure is eventually discovered, rather than planned and executed.

In programming, exploratory design and transcription are high-level activities: they consist of modification, incrementation, and exploratory understanding, but they also capture a higher-level intent of the programmer.

Analysis

The analysis is grouped into a set of patterns. Each pattern presents a problem that might be encountered during programming. The problem is presented in a general way, followed by an example scenario. Solutions are described, both for a static type system, and for a dynamic type system. Finally, both solutions are analyzed in terms of the cognitive dimensions.

The focus here is on the most significant cognitive dimensions at play. In many of the cases, there may be other differences that are not mentioned, because they are relatively minor compared to the ones presented.

It should be noted that these patterns are not intended to provide a comprehensive list of all the differences between static and dynamic typing. Rather, they are intended as a broad general analysis, and as a leaping-off point for further in-depth analysis.

Detecting type errors

In programming, as with any human activity, errors are an inevitable part of the process. Many of these errors are *type errors*: problems caused by using a type incorrectly, or by using the wrong type.

Problem

In order to provide stable and correct software, a programmer must be able detect as many of type errors as possible.

Type errors account for a large portion of the mistakes made during programming. Detecting as many of these errors as possible will have a direct effect on the quality of the program.

Solution

The main purpose of a static type system is to prevent as many type errors as possible. With static typing, the compiler performs type checking before the program is executed. This will check all code paths, even those that may never be executed.

Compile time type checking is sufficient to catch most type errors. However, in most static type systems, anything that cannot be proven to be type-safe at compile time will be considered a type error.

Dynamic typing, on the other hand, performs type-checking when the program is actually being executed. At this phase, there is more type information available than at the compile phase, so dynamic type systems can detect type errors more accurately.

However, dynamic type checking can only detect errors in code that is executed. In order to perform a thorough type check, all code paths must be executed.

Analysis

Since dynamic type checking will only check code that is executed, it is a potentially more *error-prone* notation. It places an additional burden on the development process: the program must be thoroughly tested, with all possible code paths executed. On very large projects, this may be quite difficult. On the other hand, although type errors can account for a large number of programming mistakes, static type checking might give the programmer a false sense of security. Most experienced programmers will acknowledge that the fact that a piece of code compiles says very little about its correctness.

Localized Understanding

Problem

Before making a change in a section of code, it is necessary to gain a good understanding of the code, in order to fully grasp what modifications need to be made, and what their implications will be.

This problem will be encountered very frequently by almost any programmer working on a relatively large project and/or collaborating with other developers. In even a moderately complicated program, it is beyond the capacity of a single developer to maintain a full understanding of the system model in his head. Even when examining one's own code, it is often necessary to engage in some form of exploratory understanding before performing modifications.

Solution

In order to understand the code, a programmer must discover the purpose and behaviour of objects and methods. To understand the behaviour of a method, he needs to examine the code for that method, which is found in the class

definition. Similarly, to discover the purpose and behaviour of an object, he must understand the behaviour of its methods.

Therefore, in order to understand a piece of code, a programmer needs to know what classes the objects belong to. In a static type system, the class of an object can be determined by examining its declaration. This can usually be achieved by a simple search—local variables will be declared somewhere in the method body, instance variables are declared in the class definition, and parameters are declared at the top of the method declaration.

In a dynamically-typed system, it is more difficult to learn the type of an object, since types are not explicitly declared. In some cases, this can be overcome by using a naming convention. In Smalltalk, instance variables and parameters are often given a name that indicates that they are an instance of a specific class. For example, a parameter that is expected to be an instance of the String class would be named "aString". However, since this is not enforced in any way, the name could be misleading: for instance, if the parameter was intended to be an instance of a subclass of String.

In the static typing case, a similar problem may be encountered. The declaration only captures the *static* type of the object, but at runtime, it may be an instance of a subclass of that type. For example, the object might be declared to be an instance of File, but at runtime it is actually an instance of HttpResource, which is a subclass of File. If the code in question calls the File.open() method, then the method that will actually execute is HttpResource.open(); however, the programmer might mistakenly assume that File.open() is the one being called. In cases like this, determining the runtime type of the object would be achieved in the same way as in a dynamic type system.

With dynamic types, it is necessary to perform a deeper code inspection to learn the type of an object. If the object in question is a local or instance

variable, the programmer would need to examine all the locations where the variable is assigned into. For method parameters, he would need to examine all callers of the method, and recursively discover the types of the objects that are passed in. In practice, it is often easiest to examine the code in a debugger, which reveal the runtime types of the objects.

Analysis

With static typing there is much greater *visibility*: the types of objects are part of the structure of the system, and explicitly declaring the types makes this structure more visible. In addition, it provides much greater *role expressiveness*: the purpose of an object can more easily be inferred, because the programmer can easily see what its type is. It is similarly easy to find the implementation of the method. This greater role expressiveness is a great advantage in statically-typed system, because the *Localized Understanding* problem, and more generally the activity of Exploratory Understanding, is an integral part of the software development process.

However, this greater role-expressiveness can also be somewhat *error-prone*, when the runtime type of an object is different from its static type, as described above. As we will see later, some problems in static type systems require trading some of this role-expressiveness for additional flexibility.

Another relatively minor downside to the explicit type declarations in a static type system is that it makes the code more verbose. This verbosity, or *diffuseness* as it is referred to in Cognitive Dimensions, can make code more difficult to understand. Given that it usually only results in a few extra words added to a line, this diffuseness is an acceptable trade-off for the greater role-expressiveness that is gained.

Lists of Heterogeneous Items

Problem

There is a collection of objects stored in an array or list structure. When processing this structure, the objects need to be handled in ways specific to their runtime types.

For example, when a parallel computation is being performed, the results might be collected in a matrix. The results would all be integers, except if an error occurred during the processing phase, in which case the result would be an instance of the Error class.

Solution

With static typing, all objects an array or matrix data structure must be of the same static type. In Java, a 100 by 100 matrix of integers would be declared like this:

```
Integer result[][] = new Integer[100][100];
```

However, that will not actually work in this example, because *sometimes* there will be an instance of the Error class stored into the matrix, which would cause a type error. The solution to this problem is to make the static type of the matrix the common ancestor of all the types that might be stored into the structure. This would be useful if the types were similar—for example, Integer and BigInteger. However, if the only shared ancestor in the class hierarchy is the root node, as in this example, then the programmer is essentially forced to throw the type information away. The matrix would have to be declared as:

```
Object result[][] = new Object[100][100];
```

When processing the objects, it would be necessary to use dynamic type checks:

```
for (int x = 0; x < 100; x++) {
    for (int y = 0; y < 100; y++) {
        if (result[x][y] instanceof Error) {
            Error errorResult = (Error)result[x][y];
            <perform error processing>
        } else {
            Integer integerResult = (Integer)result[x][y];
            <perform integer processing>
        }
    }
}
```

In effect, this is dynamically-typed code. In fact, the solution in a dynamic type system would look almost the same, except without any type declaration or casts. For example, in Python, the code would be written like this:

```
result = [] #declare a new list (Python's equivalent of an array)
...
for x in range(100):
    for y in range(100):
        if isinstance(result, Error):
            <perform error processing>
        else:
            <perform integer processing>
```

Analysis

The solution presented in Java is entirely dynamically typed. Compared to the Python version, it is more *diffuse*, which makes the code harder to read and understand. It also does not have any of the role-expressiveness that is usually gained from static typing.

This solution also demonstrates a usability problem that is not adequately captured by the cognitive dimensions framework—the feeling of "not doing the

right thing". A static type system provides a set of rules and constraints that a programmer must work within. Because the solution presented here effectively requires us to circumvent the type system, it can cause a feeling of unease, that there must be a better way to do it. Martin Fowler and Kent Beck coined the term *code smell* to refer to code that doesn't look quite right, possibly because it could be done in a better way [2].

The statically-typed solution also has a problem with *consistency*. The result of another computation might produce a similar matrix of integers, but with no possibility of a failure occurring. In that case, the matrix would be typed "Integer[]", and although the two structures would in practice have almost the exact same form, they would have a significantly different representation.

Passing a function as a parameter

Problem

You want to pass a function into a method.

Methods are often parameterized over a number of values. That is, the behaviour and result(s) of the method depend on a number of parameters that are passed into the method. Sometimes it is useful to parameterize a method over one or more *behaviours*.

For example, a method that is called from several possible contexts may need to report an error to the user. If it is called from a graphical user interface, the error is reported in a message dialog; and if it is called from a command line interface, an error message is printed to the console. The simplest solution is for the method to have a parameter, which is a function with a single parameter representing the message. When called from the different contexts, each would pass in a function that could report the error in an appropriate way.

Solution

Assuming that the language supports first-class functions, it is possible to pass a function in as a parameter to a method. With static typing, the parameter must include the full method signature, in order to perform static type checks when the function is called. This means that the method declaration would need to look something like the following [1]:

```
void removeAppendix(Patient patient, void reportErrorFn(String message)) {
    ...
}
```

With dynamic typing, the code is much simpler—the parameter that will be a function is declared just like any other parameter. For example, in Python:

```
def removeAppendix(patient, reportErrorFn):
    ...
    if (alreadyRemoved):
        reportErrorFn("Your appendix has already been removed!")
```

Analysis

The statically-typed solution is more *diffuse* than the dynamically-typed one. When the function signature is more complicated, or if more than one function is being passed into the method, then the method declaration can get quite verbose and confusing.

On the other hand, the statically-typed solution is more *visible*: it is clear what parameters the function takes. It also has greater *role-expressiveness*, since the purpose of the reportErrorFn is more clear if we can see the parameters it

[1] Java, however, does not support first-class functions, so it is not possible to pass a function as a parameter to a method. The work-around is to declare an interface containing a single method, and to pass in an object that implements this interface. Since the lack of first-class functions is not a general property of static type systems, but a particular property of Java, it will not be discussed any further.

takes. In fact, with dynamic typing, it is not even obvious that the parameter is expected to be a function.

Inserting code around a function call

Problem

You want to insert some code before or after a function is called, for many different functions.

This is a more general version of the *Passing a function as a parameter* pattern. However, that pattern addressed the problem of passing a *specific* function as a parameter to another function. In this case, we need to support many different functions with different signatures.

A common example of this problem is when you want to keep a log of when certain methods are called. Such a log might be used for performance tuning or debugging purposes.

Solution

With dynamic typing, it is possible to write a method that will insert the desired code before or after dispatching the target method, assuming that the language supports first-class functions. For example, this can be done quite easily in Python:

```
def logAndDispatchMethod(method):
    logMethodCall(method);
    return method();
```

At each call site, instead of directly calling the method, the calling code can be modified to call the helper method:

```
result = serveCustomer()
```

would be changed to:

```
result = logAndDispatchMethod(serveCustomer)
```

This code will work no matter what the return type of the method is, because the value returned by logAndDispatchMethod will be whatever the return type of the original method was. However, with static typing, you would need a different version of logAndDispatchMethod for every possible method return type. For different numbers and types of arguments, the dynamic solution can be adapted fairly easily, but with static typing, the combination of different signatures and return types makes a general solution virtually impossible.

Analysis

The statically-typed solutions presents a problem of *abstraction gradient*. With static typing, certain kinds of abstraction are either very difficult, requiring advanced use of reflection, or are simply not possible.

In programming languages, the effects of abstraction (or a lack thereof) can be described in terms of other cognitive dimensions. When a desired abstraction is not possible, it makes the program more diffuse, because more code is required to accomplish the same task. This typically also makes the code more error-prone, because in writing more code, the programmer is more likely to make a mistake.

Mapping between different structures

Problem

There is data in some external structure (e.g. a dictionary, an XML file, a SQL database) that needs to be extracted, and mapped to a different structure in

your program.

For example, a program may need to serialize its state to an XML file. When reading the data back in, the file is parsed into a dictionary of key-value pairs. Each key is a string representing the name of an instance variable in a particular class, and each value is the object that should be stored into the instance variable.

Solution

With static typing, in order to set the value of an instance variable of an object, it is necessary to know the type of the instance variable. For example, in Java, this would be done using the reflection library:

```
Field destField;
int fieldValue;
Object destObj;
...
Class c = destObj.getClass();
try {
    destField = c.getField("width");
    destField.setInt(destObj, fieldValue);
}
```

In the Java reflection library, there is a separate "set" method for every kind of primitive type (int, char, boolean, etc.). This is undesirable because it requires code that explicitly handles all possible types.

With dynamic typing, all the different types can be handled by the same code. In Python, the same example is brilliantly simple:

```
destObj.width = fieldValue
```

In some ways, this is not a fair comparison, because Python is simply more flexible and less verbose than Java, for many reasons other than its type system.

However, the key thing to note here is that only a single case is required, no matter what the type of the field is. The Java example presented about would in fact have to be repeated for every possible type.

In fact, this is only necessary in Java when dealing with primitive types, because they are outside of the normal object hierarchy (i.e., they are not subclasses of the Object type). Because this statically-typed solution is so cumbersome, Java also implements a dynamically-typed solution for regular objects. There is a generic "set" method which takes an Object as the field value, and throws a runtime exception if the field being assigned into is not a compatible type.

Analysis

In the statically-typed solution, there is a serious problem with *viscosity*. In the Cognitive Dimensions framework, viscosity refers to how resistant to change the notation is. This example is difficult to change because the program is hard coded for a specific set of types. If the types change, the code must be changed. Even worse, if the data must be handled in a special way for example, if it needs to be pre-processed before being assigned into the field—then this special handling must be duplicated for each of the types. This is referred to as *knock-on viscosity*, because several separate operations are required to restore consistency to the system.

By duplicating the same code several times, the code is more diffuse, and also more error-prone. A programmer might easily change one part but forget to change the other, or might make a mistake when modifying one of them. There is also a hidden dependency, because someone modifying external data set might not realize that there is code that relies on the current structure.

This is also a problem of *abstraction*. Dynamic typing allows a higher level of abstraction—it allows the programmer to call the same method to fetch the

data and to set the instance variable, regardless of the type. This higher level of abstraction makes the code easier to write and understand.

Implementing a contract

Problem

Two different objects in the system collaborate in some way. The collaboration needs to be implemented in a flexible way, so that one or both of the objects could be replaced with a compatible object of a different class.

In object oriented terminology, there is said to be a *contract* between the collaborating objects. That is, each object expects that the other will respond to certain methods.

Solution

In static typing, the contract of an object is defined by its static type. If a variable is declared to be of type Animal, then it is known to fulfill the Animal contract. In a language that supports multiple inheritance, an object can implement many different contracts by inheriting from multiple classes. Java, on the other hand, only supports inheritance from a single parent; so it has another way to represent contracts: an object can declare that it implements a specific *interface*. An interface is defined much like a class, except that it cannot contain any implementations.

So, with static typing, contracts are specified by a class or an interface; which an object must explicitly inherit from. However, there may be cases where you want to provide a compatible interface without subclassing, because subclassing might have other effects on the system. For example, it could cause a significant amount of resources to be allocated, or another piece of code might be tracking changes to the class hierarchy.

In a static type system, subclassing serves a dual purpose. First, it allows a class to inherit and the data members and behaviour of a superclass. Second, it is a way of declaring that the subclass implements the contract of the superclass.

Since contracts must be explicitly embodied in either a class or interface definition, it is fairly onerous for a programmer to distinguish between the contract of a class, and its specific implementation. In practice, Java programmers will only specify when absolutely necessary.

Dynamic typing allows a class to implement a contract simply by responding to the appropriate methods, so contracts and subclassing are two distinct concepts. This concept of compatible interfaces has been called *duck typing*, as in "if it looks like a duck, and quacks like a duck, then it's a duck."

Another interesting characteristic of dynamic typing is that it is possible for a class to implement only the necessary portion of a contract, and still successfully collaborate with other objects using that contract. For example, if a method is written expecting one of the parameters to be a File object, any object can masquerade as a File by implementing only the methods that are actually called, rather than *all* of the methods of the File class.

Analysis

In a static type system, because the contracts that a class will implement must be decided ahead of time—i.e., when the class is written—there is a problem with *premature commitment*. When the classes are part of a framework written by someone else, then it may not actually be possible to change the class definition. It would be necessary to use a work-around such as the Adapter pattern.

The statically-typed solution is also more *viscous* than the dynamic one. In order to make an object implement a contract, it is necessary to modify the class definition—even if the class already responds to all the appropriate methods.

However, there are also some advantage to the static solution. There is

greater visibility, because a programmer can tell at a glance what contracts a class implements. There is also greater role-expressiveness: by implementing a specific contract, it gives the programmer a better idea what kinds of roles the object can play in the system.

With dynamic types, there can be hidden dependencies. If a method is currently taking a File as a parameter, and you would like to pass another compatible object, you can't tell (without inspecting the code) what methods of File are being called, or in other words, what contracts are assumed between the File and the method it is passed into. In the static solution, you can at least know exactly what methods you have to implement (even if those methods are not being called).

The dyamically-typed solution provides a better *abstraction gradient*. The concepts of subclassing and implementing a contract are two distinct concepts, but the abstraction provided by static typing does not allow them to be used independently.

The fact that the two concepts are intertwined in most static type systems hints at a new cognitive dimension. It is not quite a hidden-dependency issue, because it's not hidden; however, there is a dependency between the two that is not necessary. "Undesired consequences" might be a good way to describe the problem here—that changes to one portion of the notation cause undesired changes to the whole system.

Conclusion

There are some clear themes among the seven analyses. Not surprisingly, explicit static type systems provide greater visibility and role-expressiveness, because the type declarations are explicit. However, this also comes at a cost of greater diffuseness in the notation. Except in some special cases, this diffuseness is

minor, and is an acceptable trade-off.

From the first pattern, "Detecting Type Errors", we might naively conclude that dynamic type systems are more error prone. This conclusion rests on the assumption that the compiler will catch type errors that testing does not. However, this would imply that it is acceptable to program code that is type checked but never tested, a statement which few programmers would agree with.

In fact, that is the only pattern which concludes that dynamic typing is more error prone, while the last three produce cases where a static type system fares much worse. Those patterns represent the most difficult problems of the set, for which static typing presents serious problems with abstraction gradient.

In programming languages, abstraction gradient can be seen as higher-level cognitive dimension, because a lack of appropriate abstractions causes greater diffuseness and viscosity in the notation. The programmer must use more code to express a concept, increasing diffuseness; and more often than not, making it more difficult to make changes to the structure.

Interestingly, two of the patterns identified concepts that don't seem to be adequately captured by the cognitive dimensions framework. The first is *code smell*, which describes the uneasy feeling you get when you see code that just "doesn't quite look right". Code smell can occur when the notation is overly complex, or when it is simply not aesthetically pleasing. It's not quite clear how code smell could be framed as a cognitive dimension.

The second concept that was not captured by the framework is *undesired consequences*. This occurs when two distinct entities in the domain are affected by a single aspect of the notation.

Both of these concepts were caused by the restrictive nature of static type systems. Code smells can occur when the programmer has to circumvent the type system; and when the type system provides an abstraction that is too

coarse, changes can have undesired consequences.

The general conclusion that can be drawn from this analysis is that dynamic typing causes fewer cognitive roadblocks during typical programming activities. That is not surprising, because dynamic typing is less restrictive then static typing. However, there are still some advantages to static typing.

The typical comparison of static and dynamic typing supposes an either-or proposition. Static typing holds a particular appeal to formalists, who dream of a future where programs can be proven to be correct, like mathematical formulae. But even the relatively simple problems presented here demonstrate that most static type systems must still depend a great deal on some form of dynamic typing.

By using the cognitive dimensions framework as the basis of comparison, this paper provides more of a cross-cutting analysis. The specific strengths and weakness identified in each approach provide an opportunity to design new type systems that combine the best qualities of both static and dynamic typing.

Further Work

This paper has identified several ways where dynamic typing falls short from a usability perspective, the most significant of which is the in the *role expressiveness* dimension. Explicitly declared types, as featured in most static type systems, can provide very useful clues to the role and behaviour of objects, methods, and classes.

There is an opportunity for further work in this area to create development tools that can help make up for this reduced role expressiveness. For example, by communicating with the language runtime, a development environment could keep track of the runtime type of objects, and present the information to programmer in some form of secondary notation.

I believe there is a benefit to this kind of targeted refinement of type systems, especially since it is not the usual approach taken in type theory. This paper provides a starting point, and suggests that a deeper and more comprehensive analysis could be fruitful.

References

[1] Type system. `http://en.wikipedia.org/wiki/Type_system`. Cited 14 December 2006.

[2] Fowler, M. *Bad Smells in Code.* Addison-Wesley Longman Publishing Co., Inc., 1999.

[3] Green, T. R. G. Cognitive dimensions of notations. In *People and Computers V*, A. Sutcliffe and L. Macaulay, Eds. Cambridge University Press, 1989, pp. 443–460.

[4] Green, T. R. G., and Petre, M. Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing 7* (1996), 131–174.

[5] Prechelt, L. An empirical comparison of seven programming languages. *Computer 33*, 10 (2000), 23–29.