Dynamic vs. Static Typing — A Pattern-Based Analysis

Pascal Costanza
University of Bonn, Institute of Computer Science III
Römerstr. 164, D-53117 Bonn, Germany
costanza@web.de, http://www.pascalcostanza.de

March 13, 2004

1 Introduction

This year's PostJava workshop (2004) has the motto "Back to Dynamicity", so this is a good opportunity to summarize some of the issues that revolve around the continuous dynamic vs. static typing debates. Of course, according to the workshop motto, this paper will take a pro-dynamicity point of view. This view is presented in the form of three examples in which a statically typed solution is forced to emulate dynamic type checking in order to do "The Right Thing"™. These examples are presented in the form of patterns. However, since they describe widely applied non-working solutions they are called anti-patterns here. See [1] for some essential pattern-related terminology and concepts.

Of course, this can only be a tessera to a broader investigation. This paper mainly focuses on shortcomings in Java (again, because of the workshop theme) which is only one example of a statically typed programming language. There are others that may have different and actually better solutions by default. But I hope this serves as a good starting point for a more objective analysis of an otherwise usually very heatedly discussed topic. In order to avoid misunderstandings, I have added the terminology that I use with regard to type systems in Appendix A.

2 The Anti-Patterns

2.1 Statically Checked Implementation of Interfaces

Thumbnail Implementing an interface may distract you from your real goal while developing code. Implementing the simplest workaround to avoid interference of your thinking process by the compiler can be worse than what dynamically typed languages provide by default.

Example Assume you want to implement very large character sequences that adhere to the interfaces provided in Java's core API. A natural choice is to implement the interface `java.lang.CharSequence` which consists of the following methods.

```
public interface CharSequence {
  public char charAt(int index);
  public int length();
  public CharSequence subSequence(int start, int end);
}
```

The idea is to store those character sequences in files, so that random-access memory restrictions do not hurt, and rely on the operating system's file caching facilities to ensure reasonable efficiency. For example, such an approach has been taken in the implementation of the Oberon System [14].

As a first step, each character sequence will be stored in its own file. We want to concentrate on implementing `charAt` and `length` correctly, and ignore `subSequence` and `toString` for the time being. Our first attempt is rejected by the compiler. (JDK 1.4.2 on Mac OS X 10.3.2.)

```
public class FileCharSequence implements CharSequence {
  public FileCharSequence() {...}
  public char charAt(int index) {...}
  public int length() {...}
}

costanza$ javac FileCharSequence.java
FileCharSequence.java:1: FileCharSequence is not abstract and does not override
abstract method subSequence(int,int) in java.lang.CharSequence
public class FileCharSequence implements CharSequence {
       ^
1 error
```

Since our goal is to get rid of this compiler error as soon as possible in order to continue our flow of thinking, we add the following stub implementation of the missing method.

```
public class FileCharSequence implements CharSequence {
  public FileCharSequence() {...}
  public char charAt(int index) {...}
  public int length() {...}

  public CharSequence subSequence(int start, int end) {
  }
}

costanza$ javac FileCharSequence.java
FileCharSequence.java:7: missing return statement
  }
  ^
1 error
```

We follow the compiler's advice and add the following code. (Note that we still just want to get rid of the compiler error as soon as possible.)

```
public class FileCharSequence implements CharSequence {
  public FileCharSequence() {...}
  public char charAt(int index) {...}
  public int length() {...}

  public CharSequence subSequence(int start, int end) {
    return null;
  }
}
```

Now, the compiler accepts the code without complaints. However, note that we have introduced a source for a bug that is potentially hard to find later on. Since we may shift our focus on other issues in the larger program that we are implementing, once we have successfully tested the basic functionality of `FileCharSequence`, we might forget about our little workaround and will only be bitten by it very late in the game, perhaps even only after deployment – a scenario that static type systems usually claim to prevent.

In order to ensure that the workaround implementation for `subSequence` will be noticed in simple unit tests, we should not return a default value, no matter how unlikely it is that it will survive subsequent execution paths, but rather issue an appropriate exception together with a message that describes the cause.

```
public class FileCharSequence implements CharSequence {
  public FileCharSequence() {...}
  public char charAt(int index) {...}
  public int length() {...}

  public CharSequence subSequence(int start, int end) {
    throw new UnsupportedOperationException
                ("FileCharSequence.subSequence not implemented yet.");
  }
}
```

Note that this behavior is exactly what dynamically typed languages provide by default: They issue the infamous "message not understood" error for methods that are not implemented. (`UnsupportedOperationException` is provided for such purposes by the package `java.lang`.)

Context An interface requires implementing two or more different protocols, and for a problem at hand one is interested only in a subset of those protocols, at least for the time being.

Problem (Forces) There are different reasons why one does not want to implement a complete interface at once. Apart from the fact that a certain set of protocols might just not be needed for a given program (which would indicate that the interface does not provide the right level of granularity), implementing the complete interface at once can just be too complex. It is important that one can focus on one problem at a time, and turn to a different problem later on. Furthermore, the sequence in which one wants to address problems typically cuts across the program – it is unusual that classes are implemented one at a time. In fact, this is the gist of iterative software development processes.

However, statically typed languages require programmers to deal with all methods of an interface at once, as soon as they want to compile their own code. In order to refocus on the problems that are of most importance to them, they have to provide stub implementations that might introduce potential sources for subtle bugs.

Solution Throw dynamically checked exceptions that indicate that the involved methods are currently not implemented. Be very descriptive: Mention the package, the class name, the method name and its signature, so that the message is always unambiguous, for example in the case of statically overloaded methods.

Implementation Dynamically typed languages do this by default.

Consequences Dynamically checked exceptions that indicate messages that are not (yet) understood ensure that they are correctly handled and displayed by unit tests, and that you can arrange the sequence in which problems are addressed according to your or your customer's needs. On the other hand, programmers do not get any hints from the compiler about what methods they might have forgotten to implement. They actually need to learn to use unit tests effectively.

Known Uses Modern integrated development environments (IDEs) for Java compensate for the requirement to fully implement interfaces by way of automatic or semi-automatic code generation. For example, NetBeans 3.5.1 provides a synchronization tool to automatically insert skeletons for methods required by interfaces. When opening the first version of class FileCharSequence in NetBeans, this tool adds the following code skeleton (as tested on the Java configuration mentioned above).

```
public CharSequence subSequence(int param, int param1) {
}
```

It seems to be the programmer's responsibility to fill such methods, leaving open the possibility of adding slightly incorrect code. The situation is even worse in the case of Eclipse. Version 2.1.2 provides

an entry "Override/Implement Methods" in its context menu that generates the following stub code (tested on the same configuration again).

```
/* (non-Javadoc)
 * @see java.lang.CharSequence#subSequence(int, int)
 */
public CharSequence subSequence(int arg0, int arg1) {
  // TODO Auto-generated method stub
  return null;
}
```

We have clearly identified the return of an arbitrary value as the wrong thing to do, even if it is `null`. We have not yet gathered empirical data whether this issue causes serious problems in real-world programs.

Related Patterns A collection of testing patterns and links to other resources related to unit testing are provided at [12].

2.2 Statically Checked Exceptions

Thumbnail Statically checked exceptions lead to leaking abstractions. This happens when a component interacts with another component that declares to throw unrelated exceptions that stem from a third component but which can only be handled on a higher level.

Example Here, we are using the term "component" in a very loose way – we mean any part of a larger program that can be distinguished from other parts in some way. We do not consider the notion of purely explicit dependencies, as described in [10].

Java provides the RMI library for accessing remote services. This library declares a few statically checked exceptions that clients need to handle either by catching them and reacting in some way, or by declaring them themselves which advises the runtime system to pass them on to the next level. In turn, that next level has these two options to handle the exception, and so forth up to the main method of the program.

Assume a component that performs some heavy computations based on statistical data. It should be possible to deploy this component by itself, in which case the data is read from a local file, or else it will fetch the data from a remote file via an RMI service. In the latter case, this statistical component needs to deal with the statically checked RMI exceptions. None of the two options provided by Java's exception handling mechanism are appropriate here: On the one hand, the statistical component cannot react properly to an RMI exception – only the user can properly check network connections in the end. On the other hand, declaring RMI exceptions is not appropriate to the abstraction provided by the statistical component.

One workaround would be to declare another type of "statistical component exception", and to rethrow an instance of that type in lieu of an RMI exception. However, this exception still only occurs in the interface of the statistical component because of the RMI exception. For the stand-alone use of the component, the exception does not make any sense. (The file with the statistical data could be missing, but such an error would occur at a different point in time than a network connection that might fail at arbitrary points in time.)

Context Three or more components interact with each other at different levels. The exceptional situations that might occur in different components are generally unrelated to each other, but some or all of the exceptions of one or more components must be passed on by one or more of the other components.

Problem (Forces) An exception that is thrown by component A needs to be handled by component B, but on the one hand B cannot react in appropriate ways, and on the other hand explicitly declaring the exception is unrelated to the domain that component B is concerned with.

Solution Use dynamically checked exceptions. They are passed on by any code without the need to even mention the exceptions. A component only needs to explicitly deal with exceptions that it is concerned with.

Implementation Dynamically typed languages that offer exception handling mechanisms do this by default.

Consequences Abstractions provided by components do not leak. On the other hand, programmers do not get any hints from the compiler about what exceptions they might have forgotten. They actually need to read and understand the documentation of the components they use.

Known Uses C# is generally a bad copy of Java, but improves on this issue by dropping statically checked exceptions. Dynamically typed languages typically do not mention the advantage that their respective exception handling mechanisms do not make abstractions leak because their users take this for granted.

Since JDK 1.4, Java provides a workaround for dealing with statically checked exceptions by way of wrapping an exception in another one. For example, a statically checked exception can be wrapped in an instance of `RuntimeException`, and then that instance can be rethrown. However, this requires clients to analyze caught exceptions in a lot more detail.

This "Statically Checked Exceptions" anti-pattern has already been discussed in [4].

Related Patterns See [6] for some techniques for dealing with exceptions in Java. See [9] for a rationale for advanced condition handling mechanisms in a dynamically typed language.

2.3 Checking Feature Availability

Thumbnail Checking if a resource provides a specific feature and actually using that feature should be an atomic step in the face of multiple access paths to that resource. Otherwise, that feature might get lost in between the check and the actual use.

Example Static typing promotes the notion that the availability of a particular feature should be checked before it is actually used. For example, fields and methods can be regarded as features of classes. Instead of the option to throw "message not understood" errors, particular features are considered to either always or never exist.

Assume a class Person that implements an interface IPerson and a wrapper class that implements the Employee role for a person. The instance check in the following code snippet is suggested by the static typing philosophy. (For example, see Section 15.20.2 in the Java Language Specification [5].)

```
IPerson dilbert = new Person("Dilbert");
Company dogbert = new Company("Dogbert Consulting");
...

dilbert = dogbert.hire(dilbert);
  // returns a wrapper of type Employee with new methods
  // old methods are forwarded to the original object

...
System.out.println("Name: " + dilbert.getName());
if (dilbert instanceof Employee) {
  System.out.println("Employer: " + ((Employee)dilbert).getEmployer().getName();
}
```

However, this fairly innocuous code can lead to a disastrous bug in the face of multithreading. Assume that Dilbert's employment status is changed in another thread by the following assignment.

```
dilbert = ((Employee)dilbert).fire(); // removes the Employee wrapper
```

Under unfortunate boundary conditions, this change may take place just between the instanceof check and the attempt to print Dilbert's employer. In a test using JDK 1.4.2 under Mac OS X 10.3.2, we have repeatedly executed the above print sequence, and have concurrently hired and fired Dilbert every five seconds. In one test run, the program terminated with the (unchecked) `ClassCastException` after only 2 hours 56 minutes 54 seconds. Such a race condition is the kind of bug that you definitely do not want to have in a production system!

Context You want to use a feature only if it is available.

Problem (Forces) Statically typed languages suggest to split the check for the availability of a feature and the actual use of that feature. However, the availability status of that feature might change in between. This can especially happen in multithreaded scenarios, but also in single-threaded cases when other code has the chance to change the constituent conditions for the check result, for example like this.

```
if (feature.isAvailable()) {
  observers.notify(new FeatureIsAvailableEvent(...));
  feature.use();
  ...
}
```

Solution Instead of splitting the check and the use, make sure that they form an atomic step in the control flow. If you want to avoid cluttering the program with synchronized blocks, just attempt to perform the cast without a preceding check, but catch the potentially thrown `ClassCastException` instead.

```
System.out.println("Name: " + dilbert.getName());
try {
  System.out.println("Employer: " + ((Employee)dilbert).getEmployer().getName();
} catch (ClassCastException e) {
  // do nothing !
}
```

Note that the throw of the `ClassCastException` also suppresses printing of the prefix "Employer: ".

Now, the output will always accurately reflect a person's employment status, even if her name has already been printed.

Implementation Dynamically typed languages throw "message not understood" errors by default. You only have to catch them instead of `ClassCastException`. Apart from that difference, the resulting behavior is the same.

Consequences Dynamically checked exceptions that indicate unavailable features ensure that they can be correctly dealt with in code that depends on the availability status of those features. Synchronization constructs can largely be avoided. On the other hand, programmers do not get any hints from the compiler about what features are mandatory or optional, especially when an unchecked exception is used to indicate the availability status. They actually need to read and understand the documentation of the classes they use.

Known Uses There are several variants of this anti-pattern. For example, the stream classes defined in the package `java.io` provide mark and reset methods to jump between positions in a stream. Whether a stream supports these two methods is indicated by the method `markSupported()` that has to be called before making use of mark and reset. This has been corrected in the package `java.nio`, introduced in JDK 1.4. Now, the reset method may throw an (unchecked!) `InvalidMarkException`.

This anti-pattern also arguably lies at the heart of the problems involving the double-checked locking idiom in Java [2], in which a singleton object is created after the respective field has been checked against `null`, without proper synchronization.

Related Patterns The Null Object pattern shows how to avoid checks against `null` before dereferencing variables [7].

3 Conclusions

Here are some preliminary conclusions, on different levels. First, the anti-patterns clearly seem to hint to examples in which dynamic typing is indeed The Right Thing™. However, in all those examples, the recommended solutions require programmers to better understand the issues involved, like how to use unit testing tools, API documentation and component contracts. It is strange that languages that are supposed to be addressed to average programmers seem to need to include non-working solutions by default in order to be better accessible. That this still seems to work in practice is truly amazing, and I would welcome some discussion on the issues involved.

On another level, this is the first attempt that I am aware of to use a pattern format as established in the patterns community in order to describe programming language features as such instead of only describing solutions implemented *using* certain language features. This is an idea I have had in mind for quite a long while. I think that the fact that it is used for anti-patterns here is largely irrelevant. To the contrary, I am convinced that this format can be used to describe working programming language constructs in a positive way. This could turn out useful for example for domain-oriented programming approaches [11].

Finally, I would like to present a slightly alternative view on the whole dynamic vs. static typing issue instead of the pro-dynamicity one presented in this paper. We are used to thinking in dichotomies, although in many cases there are overlaps and continuums when we take a closer look. In the context of the typing debate, there is clearly an overlap between different approaches that attempt to achieve program correctness. What static type checkers, unit testing tools and design by contract have in common is that they allow a programmer to state *assumptions* about the program being developed, and test whether these assumptions hold or not. Some of these are general (universally quantified) assumptions while others are particular (existentially quantified) ones. It is not even clear that these categories map unambiguously to static or dynamic approaches. Imagine a method for calculating prime numbers – is this property enforceable by a static type checker or not?

What we really need is a unified checking approach in which the programmer does not care whether her assumptions are checked statically or dynamically. Modern IDEs blur the lines between static analysis and dynamic checks anyway. Reflective programming languages (ad hoc reflection as in Lisp, or more principled as in metaobject protocols) are very well suited to provide a basis for such a unified notion of assumption checker. Intentional source-code views are pretty close [8].

Acknowledgement Thanks to Arno Haase and Dirk Theisen for some valuable feedback. Thanks to Erann Gat for encouraging me to write this paper.

References

[1] Brad Appleton. *Patterns and Software: Essential Concepts and Terminology.* http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html, 2000.

[2] David Bacon, Joshua Bloch, Jeff Bogda, Cliff Click, Paul Haahr, Doug Lea, Tom May, Jan-Willem Maessen, John Mitchell, Kelvin Nilsen, Bill Pugh, Emin Gun Sirer. *The "Double-Checked Locking is Broken" Declaration.* http://www.cs.umd.edu/∼pugh/java/memoryModel/DoubleCheckedLocking.html

[3] Luca Cardelli. *Type Systems.* In: Allen B. Tucker (ed.): *The Computer Science and Engineering Handbook*, CRC Press, 1997. (Note: Not the 2004 version!)

[4] Bruce Eckel. *Does Java need Checked Exceptions?.* http://www.mindview.net/Etc/Discussions/CheckedExceptions

[5] James Gosling, Bill Joy, Guy Steele, Gilad Bracha. *The Java Language Specification, Second Edition.* Addison-Wesley, 2000.

[6] Arno Haase. *Java Idioms: Exception Handling.* EuroPLoP 2002, Proceedings.

[7] Kevlin Henney. *Null Object.* EuroPLoP 2002, Proceedings.

[8] Kim Mens, Tom Mens, Michel Wermelinger. *Maintaining software through intentional source-code views.* Proceedings of the 14th international conference on Software engineering and knowledge engineering, ACM Press, 2002.

[9] Kent Pitman. *Condition Handling in the Lisp Language Family.* In: Romanovsky, Dony, Knudsen, Tripathi (eds.), *Advances in Exception Handling Techniques*, Springer LNCS 2022, 2001.

[10] Clemens Szyperski and Cuno Pfister. *Component-Oriented Programming: WCOP '96 Workshop Report.* ECOOP '96 Workshop Reader, dpunkt Verlag, Heidelberg.

[11] Dave Thomas and Brian M. Barry. *Model driven development: the case for domain oriented programming.* OOPSLA 2003 Companion, ACM Press. http://doi.acm.org/10.1145/949344.949346

[12] Testing Patterns. http://c2.com/cgi/wiki?TestingPatterns

[13] Charles Weir and James Noble. *The Hitch-Hiker's Guide to Google.* EuroPLoP 2003, Proceedings.

[14] Niklaus Wirth and Jürg Gutknecht. *Project Oberon - The Design of an Operating System and Compiler.* Addison-Wesley, 1992.

A Terminology

In order to avoid misunderstandings, here is the terminology that I have adopted in several years because I think it is the most neutral one with regard to different positions that are usually taken. I think it is important to use a terminology that is as neutral as possible in that it allows both pro-dynamic and pro-static views to be expressed equally well. Otherwise there can be no common ground for a fruitful exchange of knowledge between the various communities. The terminology given here is mainly influenced by [3], with some changes that make it more appropriate for dynamic languages.

- A *type system* maps either (compile-time) variables or (run-time) objects and values, or all of those, to sets of applicable operations.
- A *strong type system* ensures that for a given variable or object only the respective set of applicable operations are ever performed. A *weak type system* might let slip other operations through. For example, the allowance of integer additions on pointers in C is considered as a constituent for the classification as a weakly typed language. Note that this is not an either/or classification: There are languages that are strongly typed by default, but selectively allow for weakly typed operations, and the other way around. A language should be classified in the strong/weak typing category according to its default semantics, i.e. the one that comes "naturally" and takes less amount of work to achieve during programming.
- A *statically type-checked programming language* ensures strong typing by performing a static analysis on a program, without actually executing it, and rejecting programs that do not fulfill a certain set of criteria defined to achieve strong typing. A *dynamically type-checked programming language* ensures strong typing by performing checks at run time, while executing a program, just before operations are attempted to be performed on specific values or objects. It is important to keep strongly typed but dynamically checked languages separate from weakly typed languages: Dynamically checked languages let programs throw exceptions when an operation is not applicable, while a weakly typed language can easily lead to programs that produce core dumps by allowing them to write to, read from, or even jump to nonsensical memory locations. It is also important to note that static type checking does not necessarily mean "at compile time". For example, the Java Virtual Machine performs static type checks while loading class files, typically long after they have been compiled.

  Finally note that the terms "statically typed languages" and "dynamically typed languages" are in wide use, but are strictly nonsensical. A language is either strongly typed or weakly typed, but what can only be categorized as static or dynamic is not the type system, but how the type system is enforced. However, one can safely assume that "statically typed" usually means "statically type-checked", and accordingly "dynamically typed" usually means "dynamically type-checked". (Therefore it is ok to use these slightly more convenient terms.)
- An *explicit type system* requires to explicitly declare the type of variables while an *implicit type system* does not. Most languages in the Algol and C family use explicit type systems – for example, one has to state `int i` to state that `i` is of type `int` – while scripting languages and many functional programming languages employ implicit type systems – it is sufficient to declare `i` as a name without any type annotations, and the actual type is inferred from how the variable is used. For example, `i = 0` implicitly assumes that `i` is a number type. In explicitly typed languages, the use of a variable that does not conform to its declared type is usually regarded as a violation while in an implicitly typed language, usually the most general type is inferred from all known uses unless they are inconsistent.

It is important to note that these categories do not present either/or choices, but rather are continuums. Most dynamically typed languages perform at least some static checks, for example in order to avoid that typos unknowingly generate new variable bindings, and likewise most statically typed languages perform some dynamic checks, for example in order to avoid too rigid array bounds.

Even more important is the fact that all these categories are orthogonal to each other. Here are several examples.

- C has a weak, static and explicit type system.
- Java has a strong, static and explicit type system.
- Objective-C is a very interesting case: In its object-oriented layer, it allows for explicit typing, without requiring it, but generally checks types at run time. This means that explicitly declared types might actually not hold at run time. However by default, Objective-C compilers emit warnings at compile time when variables with explicitly declared types are inconsistently used. Objective-C programmers appreciate this combination of a strong type system that is explicit and dynamically checked at the same time.
- Haskell and ML have strong, static and implicit type systems.
- Lisp, Scheme and Smalltalk have strong, dynamic and implicit type sytems.

Note that implementations of the mentioned languages can sometimes divert from these categorizations. For example, there exist dynamically checked variants of Java (Eclipse, Dynamic Java) and statically checked implementations of Scheme, Common Lisp and Smalltalk (MrFlow, CMU Common Lisp, Steel Bank Common Lisp, Strongtalk). However, these implementations still ensure that the default semantics of the respective languages are not violated.

For an overview of all the languages mentioned above, see [13].