A Pattern Enforcing Compiler (PEC) for Java: Using the Compiler

Howard C. Lovatt, Anthony M. Sloane, and Dominic R. Verity

Department of Computing
Macquarie University
Sydney NSW 2109
Australia

howard.lovatt@iee.org

http://pec.dev.java.net

Abstract

A PEC is a Pattern Enforcing Compiler, which is like a conventional compiler only extended to include the extra checks needed to enforce design patterns. PECs are currently a research project and the PEC written is targeted at the Java programming language. This paper:

- Describes the PEC
- Describes how to use the PEC
- Demonstrates how the PEC combines static testing, dynamic testing (unit testing), and code generation synergistically into one utility
- Shows that the user of the PEC can write their own design patterns and have the compiler enforce them
- The PEC is believed to be unique in statically testing, dynamically testing, generating code and being user extendable.
- The PEC is stable enough for production code and is available for free download under the Lesser GNU General Public License (Lovatt 2004).

The PEC makes extensive use of reflection (runtime type identification); both when testing that a class conforms to pattern and also to allow the compiler to be user extendable.

Keywords: Design Patterns, Compilers, Static Checking, Dynamic Checking, Unit Testing, Extendable Compiler, Pattern Enforcing Compiler, PEC.

1 Introduction

Design Patterns for computer software were popularized by Gamma *et al.* 1995. Patterns are good ways of solving a problem as opposed to anti-patterns which are to be avoided. Many patterns are universal and are almost independent of the programming languages used. Some patterns are supported directly by the language, e.g. the Object Oriented pattern is supported by Java.

One of the desirable properties of using a pattern is that it documents intent in a high level abstract manner, for example the Singleton pattern is a pattern that states that there should be only one instance of the class. The Singleton concept is a concept readily understood by Object Oriented programmers independently of the programming language used.

The details of how you implement a given pattern do vary from language to language and even between programmers in the same language. This latter trend of different programmers using variations on a given pattern is undesirable and one of the problems the work reported in this paper addresses.

A common method of using a design pattern is to cut and past 'boiler plate'[1] code and then manually edit[2] to suit a particular application.

This approach has a number of disadvantages including:

- There are variations on a given pattern and it is unclear exactly what the programmer intended (Nobel and Biddle 2002).
- Mistakes when manually editing the 'boiler plate' or when maintaining the code means the code no longer conforms to the pattern
- The use of the pattern is not clearly documented to either the maintainer of an Application Programming Interface (API) or to the user of the API
- The exact implementation of the pattern may vary from one usage to the next in the same code because different programmers used different 'boiler plate'
- The 'boiler plate' can be impractically complicated

Copyright © 2005, Australian Computer Society, Inc. This paper appeared at The Second Asia-Pacific Conference on Conceptual Modelling (APCCM2005), University of Newcastle, Newcastle, Australia. Conferences in Research and Practice in Information Technology, Vol. 43. Sven Hartmann and Markus Stumptner, Eds. Reproduction for academic, not-for profit purposes permitted provided this text is included.

[1] Boiler plate code is example code that lacks details, e.g. method bodies, that the programmer uses as a starting point and 'fills in the blanks'.

[2] Alternatively use a wizard in an IDE

This paper describes the use of a Pattern Enforcing Compiler (PEC) for Java that eliminates the above problems and in addition gives the following advantages:

- The compiler is easy to use (typically you just add an implements clause to a class)
- The compiler doesn't require syntax extensions to Java and is therefore compatible with existing IDEs, pretty printers, etc.
- The resulting code from the PEC is backwardly compatible with old Java code and the PEC can be used to compile any Java file, not just files that use patterns
- A conventional compiler statically checks source code and generates binary code; the PEC has these functions and also dynamically checks (unit tests) code. The PEC is believed to be unique in statically testing, dynamically testing, and generating code all in one utility (see section 5 below).
- The PEC is extendable, i.e. you can write your own patterns and have the PEC enforce them. This is an important point since there are many patterns and many variations on a given pattern and it is useful to be able to state precisely what the pattern means (Nobel and Biddle 2002).
- PECs are currently a research project and the PEC written is targeted at the Java programming languages and is available for free download under the Lesser GNU Public License (Lovatt 2004). The available PEC is stable enough for commercial use.

This paper describes how you can write classes that conform to a pattern, how the pattern enforcing works, and how you can extend the compiler to enforce a new pattern. A brief description of the currently available patterns and how you run the compiler is given. Followed by a literature review and a summary. A more extensive description of the PEC is given in Lovatt 2004.

2 Enforcing a Pattern

2.1 Overview and Philosophy

The most important aspects of the system design are discussed in this section and with some reference to the literature; a more thorough literature review is given in section 5 below.

Patterns are enforced at the class level, i.e. the class has to be written by the programmer to conform to the pattern. This is to be contrasted with systems that allow the user of a class to apply a pattern to the instance of a class or to systems that are a mixture of these two approaches (class level and instance level).

The PEC does not require any new syntax for Java, instead it uses interfaces as markers to inform both the user of the class and the PEC that the class (is meant to) conform to the pattern specified by the marker. (The use of a marker interface allows a class to implement multiple patterns, since a class can implement multiple interfaces.)

This approach of specifying patterns at the class level and not requiring new syntax is to be contrasted with other approaches; e.g. Meijer and Schulte 2003, which requires new syntax and also specifies the patterns that are to be enforced when an instance of a class is to be made.

The approach of giving the class writer rather than the class user the responsibility of choosing the appropriate pattern has the advantage that classes and hence instances are not usually stand alone, they are part of a larger API. Therefore it is easier to use an API where the programmer of the API has already enforced the desired patterns. The observation that the writer of the API is the best judge of which pattern to use is supported by others; e.g. Steele 1998 argues the case for a small language (few keywords or symbols) that can be extended by adding APIs, i.e. the library writer not the language designer and hence not the user of the API decides what patterns are best.

The use of no new syntax extends to specifying the patterns themselves, not just using the patterns. This makes writing your own patterns much easier as no new language is needed. The advantage of writing specifications in the language that is to use the specifications is noted by Bokowski 1999, for example.

One of the problems with starting with 'boiler plate' code is that for an involved pattern is that the 'boiler plate' is impractically long and complicated. On the other hand you don't want to introduce special syntax as noted above. The PEC provides a solution, it allows a class to be marked as using a pattern and then the PEC can generate the necessary code. For example the Multiple Dispatch pattern, see section 3.7 below. Contrast this with the AspectJ system, Hannemann and Kiczales 2002, which does reduce the 'boiler plate' but requires new syntax.

The PEC does not issue warnings about possible problems; it either passes or fails a class when checking against a pattern. If a class fails then an error is given.

Some skill is required in designing the pattern to be enforced, too pedantic and it will find few applications, and too loose and it will add little to the formal correctness of the code. For example the supplied Static Factory pattern requires a class to have a method called `instance`*`XXX`* or *`xxx`*`Instance` that returns an instance of the class or a class derived from the class (where *`xxx`* is a sequence of zero or more characters or numbers). If the pattern where designed to require the factory method to be called exactly `instance` (no *`xxx`*) and was required to return exactly the type of the class, then this would be too restrictive for many applications. On the other hand if no restrictions on the method name where applied then the pattern couldn't easily inform the user which of the methods are the factory methods. An important feature of the PEC is that custom patterns can be added, so that new patterns can be written and so that the supplied patterns can be tailored.

When designing a pattern you need to be very familiar with Java; for example, how will the pattern interact with inner classes, serialization, multiple class loaders, or cloning? The level of protection offered by the pattern

should be documented, for example the supplied Singleton pattern does not protect against multiple class loaders and this is documented for the pattern.

2.2 Practice

There are three stages to writing a pattern enforced class and these are illustrated below using the Singleton pattern (the Singleton pattern ensures that there is only one instance of a class):

1. As with current practice, copy the 'boiler plate' code (https://pec.dev.java.net/nonav/compile/javadoc/pec/compile/singleton/Singleton.html) and modify to suit
2. Compile with the PEC (see section 4 below)

The main two differences between this approach and the current 'ad hoc' approach are that:

1. The PEC checks that you did not make a mistake when modifying the code
2. The 'boiler plate' code contains the following line:

```
public final class SingletonClass
implements Singleton {
```

The key is that the class, `SingletonClass`, implements the interface `Singleton`. This tells the PEC that the class is meant to be a Singleton and therefore the PEC checks the class for the Singleton pattern. This can be considered an extension of the type checking mechanism. A similar technique is already in Java, i.e. serialization and cloning via marker interfaces `Serializable` and `Cloneable` respectively (Sun 2003).

The complete 'boiler plate' for the Singleton pattern is:

```
import pec.compile.singleton.*;

public final class
SingletonClass implements
Singleton {

  private final static
  SingletonClass instance = new
  SingletonClass();

  private SingletonClass() {
      if ( instance != null ) {
          throw new
          IllegalStateException(
          "Attempt to create a
          second Singleton" );
      }
  }

  public static SingletonClass
  instance() {
      return instance;
  }

  // other methods

}
```

One obvious advantage of using an interface as a marker, e.g. `Singleton`, is that no new syntax is required; therefore existing IDEs, pretty printers, etc. can be used. A second advantage is that an interface is a type, so for example an Immutable container (a container that only contains objects that conform to the Immutable pattern) is possible because immutable objects implement the immutable interface. A third advantage is that the use of the pattern is documented to the:

- Programmer maintaining `SingletonClass` via the declaration "**`implements`** `Singleton`".
- User of the class `SingletonClass` via the Javadoc[3] for the class (interfaces appear as standard in Javadocs)

If the user of the class wishes to find out more about a Singleton they can click on the `Singleton` link in the Javadoc and they will be taken to https://pec.dev.java.net/nonav/compile/javadoc/pec/compile/singleton/Singleton.html which explains what a singleton is.

2.3 Under the Hood

As you might expect from a pattern enforcing compiler; the method of enforcing patterns follows a pattern! An interface that acts as a marker for checking extends `CompilerEnforced` and just like a normal interface can or cannot contain methods or static (class) fields. EG `Singleton` (minus Javadoc comments) is:

```
package pec.compile.singleton;

import pec.compile.*;
import pec.compile.creational.*;
import
pec.compile.staticfactory.*;

public interface Singleton
extends Creational,
StaticFactory, CompilerEnforced
{
  /* Empty - no instance methods
  or static fields required by a
  Singleton */
}
```

This declaration of the marker interface `Singleton` shows a number of interesting features:

- The name of the interface, `Singleton`, is the name of the pattern
- The interface *directly* (*not* by inheritance from a base interface) extends `CompilerEnforced`, this tells the PEC that `Singleton` is a pattern checking marker interface
- The interface is in a named package[4], see below for the significance of this

---

[3] Javadocs are a standard method of documenting APIs in Java. Stylized comments are extracted from the source files and turned into HTML by the Javadoc utility.

- The pattern is categorized under the heading Creational (it "**extends** `Creational`"). All the patterns are categorized, following Gamma *et al.* 1995, under the headings: Behavioral, Creational, or Structural. The purpose of the categorization is to act as an index, you can look at the Javadoc entry for `Behavioral`, for example, and it will list all the Behavioral patterns because their marker interfaces extend `Behavioral`. The use of interfaces to aid documentation is suggested by Wallace 2003[5]. (Also see section 3 below.)
- The pattern interface `Singleton` extends the pattern interface `StaticFactory`, this is because a Singleton class is a specialized type of Static Factory (one that only allows one instance to be created by the factory, see section 3.2 below). This extension of a pattern by simply extending a pattern's marker interface is very convenient; there is no need for the Singleton pattern to enforce the characteristics it has in common with the Static Factory pattern. This is because any class that implements `Singleton` will also implement `StaticFactory` (`Singleton` **extends** `StaticFactory`) and therefore the class implementing `Singleton` will also be checked for conformance to the Static Factory pattern.

In the same package as the interface there is a class called *`{interface name}`*`Utility`, where *`{interface name}`* is the name of the interface that acts as the marker. The 'utility' class needs to be in the same package as the interface and conform to the naming convention given so that the PEC can find the class. The PEC finds the class using reflection a.k.a. run-time type identification (RTTI), this allows the user of the PEC to add their own patterns; simply write an interface and 'utility' that follow the pattern described in this section and the PEC will enforce the new pattern if the new pattern is in its search path for classes.

This 'utility' class contains either the method:

```
public static void
compileTimeCheck( Class clazz )
throws CompilerEnforcedException
```

Or

```
public static void
compileTimeCheck( CtClass clazz,
JavaCompiler pec ) throws
CompilerEnforcedException
```

This second method signature is explained below; the Singleton pattern uses the first method signature and this is explained first. EG the class `SingletonUtility` (just showing declarations) is:

```
package pec.compile.singleton;

import pec.compile.noinstance.*;

public abstract class
SingletonUtility implements
NoInstance {
  public static void
  compileTimeCheck( Class clazz )
  throws SingletonException {
       …
  }
}
```

The above code snippet from `SingletonUtility` shows a number of interesting features:

- It is in the package "`pec.compile.singleton`"
- It uses a pattern itself, No Instance, which ensures that an instance of the class cannot be made
- The class is compiled with the PEC which enforces the No Instance pattern.
- It uses the first of the 'check' method signatures given above (see below for a description of the second signature)
- It throws SingletonException which extends CompilerEnforcedException
- `SingletonException` provides convenience constructors and/or static factories for making the exception and also gives an informative name, `SingletonException`. When the exception is created, i.e. the 'check' method has found an error, the exception is given a message that is used as the error message by the PEC. The exception is thrown by the 'check' method, and is caught by the PEC and the message reported. See section 4 belowfor an example of an error message.
- The method `compileTimeCheck` is called by the PEC for each class that implements `Singleton` with the argument clazz set to the corresponding `Class`[6] instance for the class to be checked for conformance to the pattern.

The reason for having two possible method signatures for the 'check' method is that there are three aspects to checking a pattern and two different APIs, `java.lang.reflect` (Sun 2003) and `javassist`[7]

[4] A package in Java is a set of class files that have the same `package` declaration at the start of the file, e.g. "**package** `pec.compile.singleton;`".

[5] In a future release the new Java 5 feature of annotations may be used for this purpose.

[6] The class `Class` in Java is part of the RTTI mechanism and by using the methods in package `java.lang.reflect` static type information about the class can be found and also instances of the class that `Class` represents can be made.

[7] `CtClass`, which is in this 'check' method's signature, is `javassist`'s equivalent of `Class`.

(Chiba 2004), are used to cover the three aspects. The two signatures given above correspond to the two APIs listed respectively. The required aspects for a PEC are:

1. The obvious aspect of pattern checking is static type checking, e.g. finding the signature of methods, this is possible using either API and therefore either method signature can be used
2. A less well recognized aspect of pattern checking is dynamic testing (unit testing). Dynamic testing is achieved using the `java.lang.reflect` API. Some researchers, e.g. Sefika *et al.* 1996, have recognized the importance of dynamic testing (see section 5 below). For the Singleton example part of the testing is to use `java.lang.reflect` to call `instance` twice and to check that it returns the same object on both calls.
3. Some patterns are impractically complex to rely on hand editing of 'boiler plate' code, e.g. multiple dispatch. For these the 'utility' method must generate extra code and the `javassist` API can do this. Section 3.7 below gives a feel for the extra code generated using `javassist` for the Multiple Dispatch pattern.

For a pattern that requires all three aspects listed above the pattern is split into two sub-patterns that are combined using inheritance of patterns as described above, for more information see Lovatt 2004.

It is believed that the PEC is unique in offering all three aspects listed above and in addition it provides an extendable compiler. A comparison with other systems is given in section 5 below.

3 Available Patterns

The PEC available is a research project and the number of patterns is expanding as the project progresses. As of writing (November 2004) there are seven available patterns. However, the Multiple Dispatch pattern is an early release and not as yet fully tested. Table 1 categorizes the available patterns and the rest of this section summarizes the available patterns. The purpose of the categorization is to aid documentation.

Behavioral	Creational	Structural
Immutable	No Instance	Immutable Value Conversions
Value	Singleton	
Multiple Dispatch (alpha release)	Static Factory	

Table 1: Pattern Categorization

An aspect of the research in this project is to find out which patterns can be usefully and practically enforced. For example it is not particularly useful to 'enforce' a Singleton pattern without checking that multiple calls to the constructor cause an exception or that the `instance` method always returns the same object (hence the need for unit testing). It is not practical to require the user to cut and past boiler-plate code for the multiple dispatch pattern, hence the need for code generation. In the description of patterns given below the Singleton pattern and the Multiple Dispatch pattern are described in greater detail than the other patterns to illustrate the static type checking, unit testing, and code generation. The other main feature of the PEC, ability to add your own patterns, is described in section 2.3 above.

As more patterns are added it may be necessary to expand the PECs capabilities. For example, it is not as yet clear that patterns like Not Null can be enforced with the current compiler.

In the summaries of each pattern references are given to three common 'catalogues' of patterns: Bloch 2001, Gamma *et al.* 1995, and Sun 2003. These references are not meant to imply that the PEC enforces a pattern that is identical to those described in these references. As noted in the Introduction above, there are many variations on a given pattern. The details of the patterns enforced are described in Lovatt 2004.

3.1 Static Factory

A Static Factory class (type) has a static (class) method that creates a new object or returns an existing object and the factory method is called instead of calling a constructor. Static factories are described by Bloch 2001 and Sun 2003 (e.g. `Boolean`). Static Factories are not to be confused with Factories described by Gamma *et al.* 1995, which are different because the Gamma *et al.* Factory method is an instance method. The main feature of Static Factory classes is that they contain static methods that act like constructors, see 2.1 above

3.2 Singleton

A Singleton class is a variation of a Static Factory class; it differs from a Static Factory because each call to its static `instance` method returns the same instance (object). Singletons are described by Bloch 2001, Gamma *et al.* 1995, and Sun 2003 (e.g. `Runtime`). The `Singleton` marker interface extends the `StaticFactory` marker interface and therefore any class that implements `Singleton` also implements `StaticFactory` and is therefore checked against both the Singleton and Static Factory patterns.

For a class to be declared a Singleton by the PEC it must be final, must have a single, private, no-argument constructor, can't be clonable, and must have a static, package or public, no-argument method called `instance` that return an object of the same type as the class. These tests are examples of static tests that the PEC performs using, in the case of `Singleton`, the `java.lang.reflect` API.

Further unit testing is also performed on a singleton, using the same API. Namely: that the `instance` method always returns the same object (it is called twice and the two objects returned have their memory addresses compared), it checks that if the constructor is called a second time that it throws an exception (private

constructors can be called using this API), and if the class is serializable it is checked to ensure that when the instance is de-serialized it is the same single instance.

3.3 No Instance

A No Instance class cannot have an instance of it made. In many ways it is similar to a Singleton and the two are often interchangeable. The use of No Instance classes is procedural instead of Object Oriented in nature. No Instance classes are described by Bloch 2001 and Sun 2003 (e.g. `System`). The main features of No Instances classes are that they contain only static methods and static fields and they have a private constructor.

3.4 Value

A Value class has equals and hash code methods that use the values of its fields instead of using the memory address of the instance. Value classes are described by Bloch 2001 and Sun 2003 (e.g. `Color`). The main features of Value classes are that they override both `equals` and `hashCode`. The Value pattern comes with a useful class, `ValueArrayList`, which interacts with `ImmutableArrayList` via the Immutable Value Conversions pattern (see below).

3.5 Immutable

An Immutable class is a variation of Value class; it differs by not changing its value once the instance is created. Immutable classes are described by Bloch 2001 and Sun 2003 (e.g. `String`). The main features of Immutable classes are that they have fields that are Immutable and that are initialized by the constructor. The Immutable pattern comes with a useful class, `ImmutableArrayList`, which interacts with `ValueArrayList` via the Immutable Value Conversions pattern (see below).

3.6 Immutable Value Conversions

An Immutable class can have a performance penalty because of excessive object creation and it can be advantageous to link an immutable class with a Value class that has a super-set of its interface, this pattern allows conversions between the Value and Immutable classes.

There is no direct equivalent of this pattern in the literature[8] but `String` and `StringBuffer` in Sun 2003 are similar. Immutable Value Conversions class's main feature is that they have `toValue` and `toImmutable` methods. An example of the interaction between a Value class and an Immutable class are the classes `ValueArrayList` and `ImmutableArrayList` (see above).

[8] The Value/Immutable/Immutable-Value-Conversion patterns are proposed as an extension to Java by one of the authors, Lovatt 2001.

3.7 Multiple Dispatch

In most Object Oriented languages the method called depends on the *runtime* type of the receiver (hidden this pointer) and the *declared* type of the arguments, i.e. single dispatch is used. In some circumstances this is insufficient; Gamma *et al.* 1995 suggest the Visitor pattern as a solution, however this is a difficult to follow pattern. Others, e.g. Nice (Bonniot 2004) and Relaxed MultiJava 2004, have suggested Multiple Dispatch as the solution. The Multiple Dispatch in the PEC doesn't require new syntax; all other solutions for Java are believed to require new syntax. Multiple Dispatch classes contain static (class) methods that are invoked like an instance method (i.e. *`{receiver}.{method}( {arguments} )`*). This pattern is an example of the PEC generating considerable code to minimize the amount of 'boiler plate' code that the programmer needs.

Below an idea of the amount of code generated by the PEC is given. It is not important to follow the working of the code, the purpose of showing the code is to give a feel for how much easier multiple dispatch is using the PEC than 'hand coding' the pattern.

To enable an appreciation of the code generated by the PEC a brief description of how you use Multiple Dispatch in the PEC is given. To use Multiple Dispatch with the PEC you identify multiple dispatch methods by listing them in a Multiple Dispatch interface, e.g.:

```
interface Shape implements
MultipleDispatch {
      Shape intersect( Shape
      shape );
}
```

In classes that implement `Shape`; static (class) methods are provided for specific implementations of `intersect`. Specific implementations are methods with type signatures that have more specific types than the base type, i.e. types derived from `Shape`. EG a class `Circle` might have a specific intersect method for `Circle`s and also for a `Circle` and a `Square` (where both `Circle` and `Square` implement `Shape`):

```
public static Shape intersect(
Circle c1, Circle c2 ) { … }

public static Shape
intersect(Square s, Circle c ) {
… }
```

Note how the static (class) methods in `Circle` specifically list the receiver (hidden `this` pointer) as their first argument.

The PEC generates something like the following code to implement the multiple dispatch of the method call to `intersect`. The code is simplified to make it easier to follow; e.g. simple and short class names are used in the example given below, in practice longer names that are guaranteed to be unique are required.

First a proxy class is needed that keeps track of the dispatching method and the available implementations of `intersect`.

```
public class ShapeProxy {

    public static
    ShapeDispatcher intersect;

    public static final
    Utilities.TreeList methods
    = new
    Utilities.TreeList();

}
```

For the dispatching an abstract base class is needed:

```
public abstract class
ShapeDispatcher {

    public abstract Shape
    dispatch( Shape s1, Shape
    s2 );

}
```

The class `Circle` needs to be modified to use multiple dispatch semantics. Firstly, `Circle` needs to register its `intersect` methods when it is loaded and ask for a new dispatcher to be written that can call its `intersect` methods. Thus a static (class) initializer is needed in `Circle`:

```
static {

    ShapeProxy.methods.add(
    new MethodNameArgsDepth(
    "intersect", new String[]
    { "Circle", "Circle" }, 2
    ) );

    ShapeProxy.methods.add(
    new
    MethodNameArgsDepth("inter
    sect", new String[] {
    "Circle", "Square" }, 2 )
    );

    ShapeProxy.intersect =
    (ShapeDispatcher)
    Dispatcher.instance(
    ClassPool.getDefault().get
    ( "ShapeDispatcher" ),
    ClassPool.getDefault().get
    Method( "Shape.intersect"
    ), ShapeProxy.methods );

}
```

(Classes `ClassPool`, `Utilities.TreeList`, `Dispatcher` and `MethodNameArgsDepth` are not written by the PEC and would not be in the boiler plate code if the pattern was 'hand written' but are supplied as part of the PEC and hence aren't described in this paper.)

The `intersect` method in `Circle` that overrides `intersect` in `Shape` is written by the PEC to call the dispatcher written by `Dispatcher.instance` via the proxy class, i.e.:

```
public Shape intersect( Shape
shape ) {

    return ShapeProxy.
    intersect.dispatch( this,
    shape );

}
```

The purpose of the above example was not to explain how the Multiple Dispatch pattern is implemented, but to show how generating code helps the programmer. If the PEC didn't generate the above code automatically, all the above code would have to be in the 'boiler plate' and maintained by the programmer making the programmer's job considerably harder.

4 Running the Compiler

As demonstrated above the PEC for Java requires no new syntax thus enabling existing IDEs, pretty printers, etc. to be used. A further advantage of not adding syntax is that an existing compiler can be used to generate a class file. This class file is easily manipulated, e.g.:

- It can be unit tested and/or inspected using the `java.lang.reflect` API (Sun 2003)
- It can be modified and/or inspected using the `javassist` API (Chiba 2000) and `javassist` can also create new class files

Therefore a conventional compiler can be used as a front end to generate a class file that is used as an intermediate representation of the program that is manipulated by the PEC. Final translation into machine binary is left, as is normal for Java, to a Java Virtual Machine (JVM). See Lovatt 2004 for details of running the compiler, particularly https://pec.dev.java.net/nonav/compile/javadoc/pec/compile/javacompiler/JavaCompiler.html.

A further advantage of using the class file as an intermediate language is that the PEC can at least in principle be used with any language that compiles to a Java class file. However this has not been tested and would rely on the language implementing all of the features of a given pattern. A further problem is that the error messages from the current patterns are Java centric. Therefore it is likely that another language would require a different set of patterns or at least modifications to the current patterns.

When the PEC compiles the faulty code it reports an error and no class file is written, for example if a Singleton class is made clonable[9] then the PEC reports this:

```
Singleton classes must not be
clonable
```

5 Literature Review

The PEC is believed to be unique in combining static testing, dynamic testing (unit testing), and code generation in one utility and is also unusual in having an

[9] By implementing `Cloneable` (Sun 2003)

extension mechanism. Although the whole is unique the individual elements pre-date the PEC in general computer science and in the context of patterns both static and dynamic testing are known. This section supports this view of uniqueness by reviewing the literature.

All of the papers below have influenced the PEC for the better; in particular features of these systems have been 'borrowed'! Only positive influences on the PEC work are included below because the body of literature is too large to include everything. The only negative influence on the PEC mentioned in this paper at all is Meijer and Schulte 2003 (see section 2.1 above).

5.1 Patterns in General

A good introduction to patterns is Coplien 1998, which gives a good description of software design patterns and their use. General 'catalogues' of patterns exist; e.g. Gamma *et al.* 1995 which did much to popularize patterns, details of how to implement patterns in Java are available; e.g. Bloch 2001, and the standard library (Sun 2003) contains many examples of pattern usage.

There is a problem in supporting patterns with a tool in that there are a lot of patterns and some of these are domain specific, this is noted by Chambers *et al.* 2000. In this paper the authors discuss the problem of supporting many patterns and note that it is undesirable to build language support for all the possibilities into a language. This is a similar argument to Steele 1998 and as noted in section 2.1 above this argument led to the design of an extendable system rather than adding new language syntax for the PEC.

Hedin 1996 notes the impracticality of supporting all patterns by syntax extensions but has a further useful observation that many domain specific languages are little more than an API of useful functions, some syntax to make the API easy to use, and a compiler that enforces the correct patterns required by the API. This last point, of pattern enforcing, is often not emphasized when discussing domain specific languages. A goal of the PEC is to enable development of APIs that require particular usage patterns and to enforce these. An example of such an API that requires a usage pattern is the interaction of `ValueArrayList` and `ImmutableArrayList` which interact via the Immutable Value Conversions pattern (see sections 3.4 to 3.6 above).

Some patterns have direct language support, e.g. the Object Oriented pattern is supported directly in many languages. Other patterns are supported by code generation systems that customize the 'boiler plate' code, e.g. Budinsky *et al.* 1996 and Mapelsden *et al.* 2002, the customized 'boiler plate' is then compiled and maintained as normal. Contrast this with the PEC that in the case of involved patterns like Multiple Dispatch (see section 3.7 above) generates the compiled code automatically.

The desirability of reducing the 'boiler plate' code is noted by Hannemann and Kiczales 2002. They use AspectJ to reduce the amount of 'boiler plate'. However AspectJ requires new syntax itself and their system does not incorporate static or dynamic testing unlike the PEC.

In specialized domains, particularly using Enterprise JavaBeans (EJBs) in a client-server application, high level development environments use patterns and generate code, e.g. Hammouda and Koskimies 2002. But these systems are very domain specific whereas the PEC is intended as a general purpose tool crossing all domains.

The PEC provides a means of documenting the use of the pattern via the Javadoc for the class. The need for documentation is recognized by Cornils and Hedin 2000 and they also note the undesirability of trying to support patterns with syntax extensions. The Cornils and Hedin system differs from PEC in that the documentation is provided via a separate file linked to the source code using a special editor. The PEC uses marker interfaces and the standard utility Javadoc to document the use of a pattern and hence does not require a special editor.

5.2 Pattern Checking Systems

One of the more complete systems in the literature is Pattern Lint, Sefika *et al.* 1996, which as its name suggests[10] processes source files and identifies patterns. The program doesn't require any new syntax for the patterns and performs both static and dynamic tests and in these regards is similar to PEC. Sefika *et al.* 1996 emphasize how unusual combining both static and dynamic testing is and gives a good list of references to systems that perform either static or dynamic testing. However Pattern Lint doesn't generate code or generate documentation and isn't a 'drop in' replacement for the compiler.

Pattern Lint uses heuristics to determine the patterns used and is also interactive to confirm its heuristics. This interactive nature is unusual in checking or compilation systems. Contrast this with the PEC that extends the type checking in a strong manner, via a declaration, to identify the pattern. This contrast between heuristics and declarations is similar to the contrast between type checking by inference (e.g. Haskell) and type checking by declaration (e.g. Java). Both checking systems have their advantages and one of the reasons for choosing by declaration for the PEC is that it is more consistent with the Java language (it feels like Java!).

A disadvantage of the current PEC compared to Pattern Lint is that anti-patterns, bad patterns, are not automatically detected. This negative pattern checking could be added to the PEC because much of the necessary infrastructure is present. However, anti-pattern detectors produce many false warnings and hence are not like the PEC that passes or fails a class (with an error and does not produce a class file), see Rutar *et al.* 2004.

A feature of Pattern Lint is that it uses heuristics to 'guess' the patterns used. This approach has much in common with reverse engineering systems that use similar approaches to aid understanding of programs either during program maintenance or for documentation. Murphy *et al.* 1995, for example, describes a system that

[10] Lint is a C language utility that performs extra static type checks beyond those of a *traditional* C compiler.

visualizes patterns in C source code. These post analysis systems differ from the PEC in that the PEC extends the type checking in a string manner and documents the use of the pattern via a declaration.

As already noted there are many static checking systems that for reasons of space are not reviewed in this paper. One system, CoffeeStrainer, however deserves special mention because it provides many more facilities than typical systems and because it has much in common with the PEC. Bokowski 1999 describes CoffeeStrainer which has in common with the PEC the use of marker interfaces to cause a class to be checked, the use of Java as the specifying language, no new syntax, user extendable, and it is a 'drop in' replacement compiler. The paper contains a good discussion of the pros and cons of using Java to specify a pattern compared with a domain-specific language. However it differs from the PEC in only providing static testing and not dynamic and also in that it does not automatically generate code for the pattern (you need to hand code the pattern it only enforces it). The PEC will generate most of the 'boiler plate' code associated with the pattern Multiple Dispatch for example (see section 3.7 above).

Compared to CoffeeStrainer the number of patterns that the PEC currently supports is much smaller but the patterns tend to be much higher level, e.g. Multiple Dispatch.

One area that the PEC *may* not be able to do, without significant extension, is detailed analysis of use of an instance of a class, e.g. to enforce a Not Null pattern. The CoffeeStrainer system can do the Not Null pattern and the techniques it uses to do this are currently under consideration for the PEC.

6 Conclusions

The PEC is believed to be unique in combining static testing, dynamic testing (unit testing), and code generation in one utility. These three aspects work synergistically together to allow software design patterns to be enforced.

As its name would suggest the PEC is a drop in replacement for a conventional Java compiler. Also: it is easy to use; classes are marked as conforming to a pattern by implementing a marker interface, e.g. a Singleton class implements `Singleton`.

The user can easily extend the PEC by writing their own patterns and include them in the PEC's classpath and the PEC will enforce them.

Pre-written patterns are supplied with the PEC that are useful in their own right (as well as demonstrating the power of the PEC). Some of these supplied patterns are novel, e.g. Value Immutable Conversion, and some demonstrate how the PEC allows what would otherwise be infeasible patterns to become practical, e.g. Multiple Dispatch.

Research is progressing to find out what patterns the PEC can enforce and what it can't. Complex patterns are possible (e.g. multiple dispatch) and cross class patterns are also possible (e.g. Immutable Value Conversion) as well as simple patterns like Singleton and a class can conform to multiple patterns (via implementing multiple pattern marking interfaces). It is not as yet clear whether the current compiler can enforce patterns at the instance level, e.g. Not Null.

You can download the PEC from Lovatt 2004 under the Lesser GNU General Public License. The PEC is a research project but is stable enough for commercial use.

7 Acknowledgements

The work reported in this paper was shaped by interactions within the Programming Language Reading Group at Macquarie University's Computer Science Department (Sloane 2004). Two anonymous referees, James Nobel from Victoria University of Wellington, NZ provided valuable feedback on the first draft of the paper.

8 Trademarks

Java, JVM, jar, and Javadoc are trademarks of Sun Microsystems, Inc. and Pattern Enforcing Compiler and PEC are trademarks of Howard Lovatt.

9 References

Bloch, J. (2001): *Effective Java™: Programming Language Guide.* Addison-Wesley Publishing Co., Inc., Boston, MA. ISBN 0201310058.

Bokowski, B. (1999): CoffeeStrainer: Statically-Checked Constraints on the Definition and Use of Types in Java. *Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE'99).* Toulouse, France. Pages: 355 – 374. ftp://ftp.inf.fu-berlin.de/pub/barat/esec-final.ps. Accessed 15 Aug. 2004.

Bonniot, D. (2004): The Nice programming language. http://nice.sourceforge.net/. Accessed 15 Aug. 2004.

Budinsky, F., Finnie, M., Patsy, Y., and Vlissides, J. (1996): Automatic Code Generation from Design Patterns. *IBM Systems Journal*, Vol. 35(2). Pages:151 - 171. http://www.research.ibm.com/designpatterns/pubs/codegen.pdf. Accessed 15 Aug. 2004.

Chambers, C., Harrison, W., and Vlissides, J. (2000): A Debate on Language and Tool Support for Design Patterns. *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* Boston, MA, USA. Pages: 277 – 289. http://doi.acm.org/10.1145/325694.325731. Accessed 15 Aug. 2004.

Chiba, S. (2004): Javassist Home Page. http://www.csg.is.titech.ac.jp/~chiba/javassist/. Accessed 15 Aug. 2004.

Coplien, J.O. (1998): Software Design Patterns: Common Questions and Answers. In Linda Rising, editor, *The Patterns Handbook: Techniques, Strategies, and*

Applications, Pages: 311 – 320. Cambridge University Press, New York, January 1998.

Cornils, A. and Hedin, G. (2000): Statically Checked Documentation with Design Patterns. *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 33).* Page: 419. http://www.cs.lth.se/Research/ProgEnv/Papers/TOOLS00.GH.pdf. Accessed 15 Aug. 2004.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995): *Design patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA. ISBN 0-201-63361-2

Hammouda, I. and Koskimies, K. (2002): Generating a Pattern-Based Application Development Environment for Enterprise JavaBeans. *Proceedings of the 26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment (COMPSAC).* Pages: 856 – 866. http://practise.cs.tut.fi/pub/papers/NWPER02_Imed.pdf . Accessed 15 Aug. 2004.

Hannemann, J. and Kiczales, G. (2002): Design Pattern Implementation in Java and AspectJ. *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications.* Seattle, Washington, USA. Nov. Pages: 161 – 173. http://www.cs.ubc.ca/~jan/papers/oopsla2002/oopsla02-patterns.pdf. Accessed 9 Nov. 2004.

Hedin, G. (1996): Enforcing programming conventions by attribute extension in an open compiler. In *Proceedings of the Nordic Workshop on Programming Environment Research (NWPER'96),* Aalborg, Denmark, http://www.cs.lth.se/Research/ProgEnv/Papers/LU-CS-TR:96-171.pdf. Accessed 15 Aug. 2004.

Lovatt, H.C. (2001): Bug ID: 4617197 RFE: Add Immutable types to Java. http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4617197. Accessed 15 Aug. 2004.

Lovatt, H.C. (2004): pec: Pattern Enforcing Compiler(TM) (PEC(TM)) Home Page. http://pec.dev.java.net/. Accessed 15 Aug. 2004.

Mapelsden, D., Hosking, J., and Grundy, J. (2002): Design Pattern Modeling and Instantiation using DPML. *Proceedings of Tools Pacific 2002,* Sydney, 18-21 Feb., CRPIT Press. https://www.se.auckland.ac.nz/courses/SOFTENG462/resources/jg/tools2002.pdf. Accessed 9 Nov. 2004.

Meijer, E. and Schulte, W. (2003): Unifying Tables Objects and Documents. http://research.microsoft.com/~emeijer/Papers/XS.pdf. Accessed 15 Aug. 2004.

Murphy, G.C., Notkin, D., and Sullivan, K. (1995): Software Reflexion Models: Bridging the Gap between Source and High-Level Models. *Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering.* Washington, D.C., United States. Pages: 18 – 28. http://www.cs.ubc.ca/spider/murphy/papers/rm/fse95.html. Accessed 15 Aug. 2004.

Noble, J. and Biddle, R. (2002): Patterns as Signs. *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP2002).* Pages: 368 – 391. Similar to http://www.mcs.vuw.ac.nz/comp/Publications/archive/CS-TR-01/CS-TR-01-16.pdf. Accessed 2 Nov. 2004.

Relaxed MultiJava (2004): The MultiJava Project. http://multijava.sourceforge.net/. Accessed 15 Aug. 2004.

Rutar, N., Almazan, C.B., and Foster, J.S. (2004): A Comparison of Bug Finding Tools for Java. To be published in the proceedings of *The 15th IEEE International Symposium on Software Reliability Engineering (ISSRE'04).* Saint-Malo, Bretagne, France., Nov., http://www.cs.umd.edu/~jfoster/papers/issre04.pdf. Accessed 9 Nov. 2004.

Sefika, M, Sane, A, and Campbell R.H. (1996): Monitoring Compliance of a Software System with its High-Level Design Models. *Proceedings of the 18th international conference on Software engineering.* Berlin, Germany. Pages: 387 – 396 http://choices.cs.uiuc.edu/Papers/SoftwareEngineering/icse-96.pdf. Accessed 15 Aug. 2004.

Sloane, A.M. (2004): Programming Languages Reading Group Home Page. http://www.comp.mq.edu.au/~asloane/plrg/reading/index.html. Accessed 15 Aug. 2004.

Steele, G.L. (1998): Growing a Language. *Object-Addendum to the 1998 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum to OOPSLA'98).* Vancouver, BC, Canada. http://research.sun.com/research/jtech/pubs/98-oopsla-growing.ps. Accessed 15 Aug. 2004.

Sun Microsystems Inc. (2003): *Java™2 Platform, Standard Edition, v 1.4.2 API Specification.* http://java.sun.com/j2se/1.4.2/docs/api/index.html. Accessed 15 Aug. 2004.

Wallace, B. (2003): PolyGlot, Inc. Design Markers. http://www.polyglotinc.com/DesignMarkers/. Accessed 15 Aug. 2004.