APRT – Another Pattern Recognition Tool

ROBINSON, Ashley and BATES, Christopher <http://orcid.org/0000-0002-1183-1809>

Published version

ROBINSON, Ashley and BATES, Christopher (2017). APRT – Another Pattern Recognition Tool. GSTF Journal on Computing, 5 (2).

APRT – Another Pattern Recognition Tool

Ashley Robinson
Department of Computing
Sheffield Hallam University, UK

Chris Bates
Department of Computing
Sheffield Hallam University, UK
c.d.bates@shu.ac.uk

Abstract— **Understanding and using Design Patterns improves software quality through better comprehension of programs for both experienced developers and relative novices. Often design patterns are learned through simplified "toy" programs and exemplars that foreground the structure of the pattern. In production code the objects and methods that comprise the pattern can be hidden within a complex mesh of intra-code relationships. This paper introduces APRT, an ANTLR-based tool that recovers the structure of both static and dynamic patterns from large codebases so that they can be studied in context.**

Keywords— ***object-oriented; software architecture; program constructs; design patterns***

I. INTRODUCTION

Design Patterns provide abstract, reusable object-oriented structures that provide generic solutions to common development problems. Most famously catalogued in [8], patterns have proven to be so powerful that they now cover areas as diverse as software design, HCI, data structures, software architectures and business processes. Even the pedagogy of Computer Science has proven to be fertile ground for the development of patterns [4].

Design patterns are now foundational knowledge for developers, part of the industry's *lingua franca*. Understanding them is as important as understanding sorting or searching algorithms. Yet teaching of patterns is targeted at having learners end in "the right place", a place in which they can implement a *Singleton* or an *Abstract Factory* or use a *Decorator* in a classroom exercise [10]. Production code is large, complex and messy and students benefit greatly from understanding how code structures that are based on design patterns fit into it. This work describes an approach to the recovery of design patterns from existing code to produce a deterministic view of the objects and relationships within the system.

Open source code repositories contain many millions of lines of code and, whilst the quality of code in these repositories varies greatly, much of it is high quality and contains examples of best-practice programming. For computer science educators, the extraction of patterns from such repositories could demonstrate the use of design patterns in context in real-world applications. This would give concrete demonstration that such patterns are solutions to problems commonly encountered by developers. Inexperienced coders who have a background in abstract problem solving through classroom exercises would be able to see how those abstract ideas become powerful and reusable when applied to complex problems.

APRT, *Another Pattern Recognition Tool* is used to recover patterns in Java code. Patterns are discovered by parsing source code in a low-level analysis to find possible occurrences of patterns that are compared to pre-defined pattern structures. By comparing in-code structures with templates representing individual patterns, APRT is able to reveal a range of structures and, through the addition of further definitions, can be easily extended to work on new patterns. Although the current implementation is Java-specific the use of ANTLR's parse trees with language-independent pattern definitions means that it can be re-targeted to work with other object-oriented languages.

Section 2 introduces design patterns and reviews other tools that attempt pattern recovery. Section 3 examines the parser generator ANTLR. Section 4 introduces APRT including analysis of its performance on a number of complex programs.

II. DESIGN PATTERNS

Software design patterns provide abstract proven and reusable object-oriented solutions to commonly occurring code design problems. Patterns have been shown to greatly increase the quality of object-oriented code, [1]. The quality improvements that arise from the use of patterns is so great that they are now foundational material in software engineering and computer science degree courses where they are as important as programming and databases.

As software systems have become larger and more complicated the difficulty of analysing and understanding their design and architecture has also grown. In modern environments, design patterns are fundamental abstractions that give a clearer overview of a system without the need for a detailed understanding of all of the source code [9].

Effective software design requires consideration of issues that may not become visible until later in the implementation. Using design patterns can help to prevent such subtle issues and improves code readability for both programmers and software architects.

A. *Analysing code*

The prevailing types of code analysis of are structural, behavioural, semantic and formal composition analysis.

Structural analysis involves inspecting inter-class relationships to identify the structural properties of classes, regardless of their behaviour. They focus on recovering structural patterns such as Adapter, Proxy or Decorator from static codebases. [13] shows that such tools can extract entities which, through reference to a database, reveal the properties of a pattern. [13] demonstrates successful recovery of Decorator, Factory, Observer, Template and Singleton patterns.

Behavioural approaches adopt dynamic analysis, machine learning and static program analysis techniques to extract patterns. These can be combined with structural analysis when searching for patterns that are structurally identical. For example the State and Structure patterns are structurally identical whilst Façade objects can be implemented as Singletons. Because patterns can be syntactically similar, behavioural analysis can produce large numbers of false positives [6].

Semantic approaches use naming conventions and annotations to get role information about classes and methods. Using semantics allows for the recovery of patterns such as Strategy and Bridge that have similar static and behavioural properties. Whilst different techniques can be used, [6] conclude that naming conventions are the most appropriate and feasible option.

III. ANTLR

ANTLR is a parser generator that uses an LL(*) parsing strategy. ANTLR takes as input a context-free grammar. The grammar can be augmented with syntactic predicates that allow for arbitrary look-ahead based on defined grammar fragments, and semantic predicates that represent Boolean values and allow the state and context of a predicate to direct the parse [11].

A. *Parsers*

The general purpose of a parser is to break the source of a program into elements that can be translated into a target language. Parsers take input in the form of a sequence of tokens and build a data structure such as an *abstract syntax tree* that represents the input, retaining all of the information from the target program.

Parsing is either bottom-up or top-down. The former is considered to be the more powerful technique but using it for anything but trivial cases is more complex.

a) *Bottom-up parsing*

A bottom-up parsing strategy starts from the leaf nodes of a tree and works upwards towards the root node. The goal here is to reduce the tree to the start symbol and report a successful parse. The most commonly used technique in bottom-up is shift-reduce parsing which allows for incremental parse tree generation without guessing or backtracking.

b) *Top-down parsing*

A top-down parser starts from the parse tree's root node and, following the rules of a formal grammar, works down towards the leaf nodes. The top-down strategy accommodates ambiguity by expanding all alternative right-hand-sides of grammar rules.

Many computer languages were designed to be LL(1), requiring only one token look-ahead during parsing as this simplifies parser construction. Because of the inherent ambiguity in languages, LL(1) parsing is often insufficiently powerful. Techniques such as the ANSI C lexer hack, described by [2] and which feeds data from the parser's symbol table back into the lexer to determine context, can help but some ambiguities are not solved so easily.

c) *LL(*) Parsing*

LL(*) Parsers are a class of recursive descent parsers, which are constructed from a set of mutually recursive procedures where each procedure implements one of the productions of the grammar. The LL(*) approach uses predictive parsing, meaning that it utilizes look-ahead rather than backtracking which allows the parser to run in linear time [11]. The LL(*) parser is not restricted to a fixed number of tokens of look-ahead, but can make decisions by token recognition using deterministic finite automata.

B. *Syntax Trees*

Syntax Trees are a commonly used data structure in compiler, used as an intermediate representation of the program throughout the stages of compilation.

a) *Abstract Syntax Tree*

An Abstract Syntax Tree, AST, is a tree representation of the abstract syntactic structure of a program, with each node denoting a construct of the language. Thesy are abstract because the tree does not represent every detail of the language syntax, so for instance parentheses are not present in the AST but are derivable from the tree structure. Figure 1 gives a simple exemplar. AST are specified in terms of Extended Backus-Naur Form, EBNF, and are commonly used in specifications and implementations to describe the abstract syntax trees of a language.

Figure 1. An abstract syntax tree

b) Concrete Syntax Tree

A Parse Tree is a common designation of a concrete syntax tree, CST, which both maintains all of the information from the input and, more concretely, reflects the input syntax in its structure. A CST, as shown in Figure 2, is a cluttered data structure and, therefore, is often converted to an AST prior to the semantic analysis stage of compilation.

C. Context-Free Grammars

A context free grammar is a formal notation for describing languages, consisting of a finite set of grammar rules. Production rules are a set of rewrite rules specifying symbol substitutions to transform nonterminal symbols into a set of either terminal or non-terminal symbols. When the rules are applied recursively they generate a terminal representation of the input [7]. A terminal symbol is a standalone language construct, whilst a non-terminal symbol denotes a syntactical phrase composed of one or more terminal symbol and can contain other valid phrase structures.

D. ANTLR

The language recognition process of ANTLR has two distinct stages: Lexical Analysis and Parsing. As used in APRT the process is conceptually similar to that of a compiler but rather than generate executable forms, the process is stopped once the parse tree has been built.

Lexical analysis in ANTLR involves first scanning an input stream of characters and then grouping those characters into words or symbols in a process called tokenizing. These Tokens contain at least two pieces of information, the token type and the raw value matched for that token by the lexer.

Syntax analysis is performed by the parser. The parser takes the token stream generated by the lexer to recognize the sentence structure and ensures that the token stream

Figure 2. A concrete syntax tree

adheres to the rules of the grammar. ANTLR uses an LL(*) parsing strategy that implements an LL(1) parser with depth-first look-ahead grafted on. The parser is top-down, recursive descent and mostly non-speculative [11].

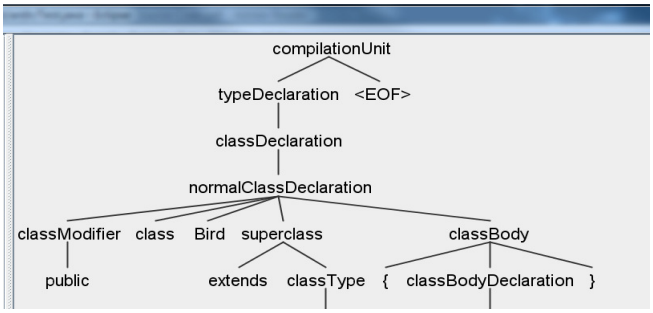

Figure 1. A concrete syntax tree

IV. ANOTHER PATTERN RECOGNITION TOOL

APRT is written in Java, utilizing ANTLR4 to generate parse trees from Java 8 source code. These trees are navigated using a listener-based tree-walker that receives event notifications based on the context of the current node. Nodes are generated from the rules of the grammar into a base listener that contains entry and exit rules for every node, allowing for manipulation of the output based upon the node's occurrence. Context-specific subclasses of the base listener are created to allow for code evaluation based on the current token.

The beauty of using ANTLR to define the whole of the language of Java and then navigating that is that the approach allows for compile-time evaluation of dynamic aspects of patterns, the process of which is essentially replicated during the language recognizer which lexically

analyzes the input stream into a token stream before parsing the token stream using the grammar as a symbol table, while maintaining static references to the code to allow for the structural aspects of patterns.

APRT detects patterns when there is a concrete definition of either structural or behavioural aspects. In the current scope of this project, language-provided patterns are excluded as their detection can be done through keyword analysis. Currently two patterns are considered for detection: Singleton and Strategy. Several styles of Singleton were found to be in common use and each of these can be detected successfully by APRT. The examples given here will show that structural patterns can be extracted by navigating interclass relationships.

A. *Design*

The general approach of APRT is to take the source code files in a directory and search for instances of design patterns. The individual files are first read into an *AntlrFileStream* object which behaves as a char array buffer. The lexer then draws input symbols from the char stream using the *match()* function. A *CommonTokenStream* is initialized based on the results from the lexer and tokenizes the file. A *Parser* subclass built from the grammar is initialized by the token stream and a *ClassDeclarationContext* is acquired from the parser. This is a grammar-defined construct that will be used by the *ParseTreeWalker* to denote the boundaries of the pattern recognition. A *ParseTreeListener* subclass that extends the base grammar *ParseTreeListener* is then defined. The listener is used because all of the code can be traversed due to the nested definitions of the language. The *ParseTreeWalker* then walks then parse tree, with the listener evaluating the classes extracted from the file with the pattern recognition rules.

B. *Detecting Creational Patterns*

Creational patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly. The Singleton pattern was chosen as a proof of concept. The definition of a Singleton is that an object has only one instance with a global point of access and can be initialized on its first use. There are many implementations of Singleton in Java, of which four are used here.

The intent of a Singleton is to ensure that a program only has one instance of a class and that there is a global point of access to it. The pattern is one of the easiest to detect as it does not require that the tool analyse interactions with other classes.

The *ClassicSingleton* has a static reference to a class of its type, a private constructor and a static getInstance() method that checks if the object is initialized. If not it creates a new instance of itself, before returning a reference to itself. This is commonly referred to as a lazy instantiation and is the most common implementation in Java code.

DoubleCheckedLockingSingleton has a private, static, volatile reference to an instance of its classtype, a private constructor and a static getInstance() method that checks whether the type has been initialized and synchronizes with the class declaration to ensure that it doesn't exist elsewhere. This has been the *de facto* standard since Java 5, after Bill Pugh's work on the idiom led to changes in the Java memory model and is generally regarded as the standard way to write Singletons in Java, [3].

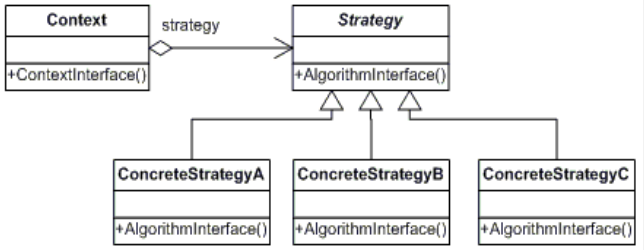

Figure 2. The strategy pattern

EnumSingleton has an EnumDeclaration (public enum <Classname>) with a single reference to 'INSTANCE;' This approach is functionally equivalent to the public field approach, except that it is more concise, provides the serialization machinery for free, and provides an ironclad guarantee against multiple instantiation, even in the face of sophisticated serialization or reflection attacks [5]. This is considered to be the most effective way to write Singletons in Java but it has not been widely adopted due to concerns over thread safety, concerns that are mostly unfounded as a Java *enum* is usually both stateless and thread safe.

The final Singleton code that APRT can detect is a *StaticSingleton* which is a static version of the 'basic' Singleton, which has shown to be reasonably uncommon but is detected here for completeness.

The pattern detection logic must allow for all implementations of a Singleton to be detected. The High level definition for a singleton is that:

- The class declaration is of the *Enum* type.
- An instance is declared within the class.
- A minimum of one method is declared.

The method declaration does not have to be static in an *EnumSingleton*, [5], as it is implied by the *enum* structure, but it can be declared that way and both must be detected by the tool. An alternative definition is:

- Has a private, empty constructor
- Has at least one static method declared

Since the detection is accounting for multiple variants of the pattern, an alternate definition must be declared to allow for use with older coding practices. This also allows the tool to differentiate which version of the pattern is detected, allowing for scope in a dynamic tool to detect the pattern usage and recommend upgrading the pattern to the more effective structure.

C. Detecting behaviour-driven patterns

The detection of behavioural patterns is based on the communication between objects. The patterns are used to define common communication patterns between objects and provide a more extensible, loosely coupled solution that is easily extensible. The Strategy pattern is shown as a proof of concept to validate the design. A Strategy pattern is defined as a family of algorithms that has various implementations depending on its client.

The strategy pattern has three participants: a strategy that declares an interface common to all supported methods; a concrete strategy that implements the method using the strategy interface; the context is then determined based on the client, and determines which Concrete Strategy implementation of the Algorithm will be performed.

Detection of the Strategy pattern is a more complex proposition than the detection of Singleton because it requires cross-class dependency checking. This is achieved by creating an internal file list containing all of the classes in the package, then searching a collection of parse trees for a match on the declaration required, to determine the token context.

```
public class Animal {
   private String name;
   private double speed;
   private String sound;
   public Flys flyingType;

   public String tryToFly(){
       return flyingType.fly();
   }
   public void setFlyingAbility(Flys newFlyType){
       flyingType = newFlyType;
   }
}

public interface Flys {
   String fly();
}

class ItFlys implements Flys {
   public String fly() {
       return "Flying High";
   }
}

class CantFly implements Flys {
   public String fly() {
       return "I can't fly";
   }
}

public class Bird extends Animal {
   public Bird(){
       super();
       setSound("Tweet");
       flyingType = new ItFlys();
   }
}

public class Eagle extends Bird {
   public Eagle() {
       super();
       setSound("Sqwark!");
   }
}
```

Table 1. Strategy Implementation

The strategy example shown in Table 1, describes an example of the strategy pattern that was used to test APRT. The Strategy is defined by the interface *Flys*, which declares the *flys* function. This interface is implemented by the Concrete strategies *ItFlys* and *CantFly*. For the purposes of demonstration, the base class *Animal* has a reference to the Strategy object. This implementation demonstrates that the Strategy pattern can be detected in the superclass of an object and in subclasses that define the context as in *Eagle*.

The process of detecting the Strategy pattern is more complex undertaking. Since the Client has an instance of a class defining the context at the time of object declaration, we must infer whether this composite component is a super-interface (a class that implements an interface). This is performed by extracting the *UNannType*, that is the unannotated type, of a *FieldDeclaration* from the class, then using this value to search the resources for classes that have a matching *UNannType*. When a match is found the resource is inspected to see if it, or one of its superclasses is a super-interface.

D. Testing APRT

Testing of APRT demonstrated successful detection of variants of Strategy and Singleton as representatives of wider classes of creational and behavioural patterns. Successful detection supports the hypothesis that most "Gang of Four" patterns can be detected by the tool through extension of the detection method.

Testing of APRT was performed first on a small set of simple classes that clearly demonstrated each pattern. An example of these proofs of concept is given in Table 2. Further testing used a set of Java implementations of design patterns sourced from a popular design pattern tutorial set,

[12]. This set contains 740 Java source files from which 367 files were selected as suitable for parsing.

In the first test case, using simple structures, APRT achieved 100% recognition of the variations of Singleton and also detected 100% of instances of the Strategy Pattern, including detection from subclass instances. This was to be expected as the tool was built to succeed in this environment. In the second set of tests, APRT successfully identified 26 instances of Singleton, 12 of which being implemented in the Enum Singleton style, as well as 23 instances of Strategy pattern. These pattern instances were subsequently verified by inspection of the source file. In terms of performance, evaluating 367 files took, on average, 55ms per file to run the tests for both patterns. This average speed was consistent across all tests.

Files	Singleton	Singleton Time (ms)	Strategy	Strategy Time (ms)
46	0	5228	0	1635
132	11	7688	9	7196
227	18	13300	22	11127
346	26	13543	23	22512

Total Found	Total Time (ms)	Average Per Detection	Average Per File (ms)	Average Per File, Per Pass (ms)
0	6863	0	149.195	74.5978
20	14884	744.2	112.757	56.3787
40	24427	610.675	107.607	53.8039
49	36055	735.816	104.205	52.1026

Table 2. Performance tests

V. CONCLUSIONS

Software design patterns provide standard and well-tested solutions to common problems. Novice programmers who are learning object-orientation benefit from using and understanding patterns but they are often presented as toy examples that fail to reflect the complexities of the relationships within production code. One way to bridge the gap between toy code and production code is to recover the implementation of design patterns from production code.

Another Pattern Recover Tool demonstrates a viable and concrete way of detecting patterns. The use of Context Free Grammars and the exploitation of compiler-style structures to build and evaluate the semantic properties of code is a concept that seems to have numerous useful applications in real world projects. The recovery of patterns and other structures would be extremely helpful in the training of both student coders and new team members in commercial development teams.

REFERENCES

[1] Ampatzoglou, A., Frantzeskou, G. & Stamelos, I., 2012. *A methodology to assess the impact of design patterns on software quality*. Information and Software Technology, 54(4), pp. 331-346.

[2] Atkey, R., 2012. *The Semantics of Parsing with Semantic Actions*. Dubrovnik, s.n., pp. 75-84.

[3] Bacon, D. et al., n.d. *The "Double-Checked Locking is Broken" Declaration*. [Online]
Available at:
www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html [Accessed 13 February 2016].

[4] Bergin, J. et al, 2012. *Pedagogical Patterns: Advice for Educators*. Joseph Bergin Software Tools, 2012

[5] Bloch, J., 2008. *Effective Java (2nd Edition)*. Addison-Wesley.

[6] Dong, J., Zhao, Y. & Sun, Y., 2009. *A Matrix-Based Approach to Recovering Design Patterns*. IEEE Transactions on Systems, Man and Cybernetics, 29(6), pp. 1271-1282.

[7] Gallier, J., 2010. *Formal Languages And Automata Models of Computation*, Computability Basics of Recursive Function Theory. Philadelphia: s.n.

[8] Gamma, E., Richard, H., Johnson, R. & Vlissedes, J., 1994. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley.

[9] Haotain, Z. & Shu, L., 2013. *Java Source Code Static Check Eclipse Plug-in Based on Common Design Patterns*. Proceedings of Fourth World Congress on Software Engineering, IEEE, pp. 165-170.

[10] Kolfschoten, G., et al, *Cognitive learning efficiency through the use of design patterns in teaching*, Computers & Education, Volume 54, Issue 3, April 2010, Pages 652-660.

[11] Parr, T. & Fisher, K., 2011. *LL(*): The Foundation of the ANTLR Parser Generator*. ACM SIGPLAN Notices, Association for Computing Machinery, pp. 425-436.

[12] Seppälä, I., *Java Design Patterns.* [Online]. Available at: http://java-design-patterns.com/. [Accessed 14 March 2016].

[13] Vokac, M., 2006. *An efficient tool for recovering Design Patterns from C++ Code*. Journal of Object Technology, 5(1), pp. 139-157.