Static Code Analysis With JavaParser

(A case study of the Visitors design pattern)
Lecture 11

Dr. Graham McDonald
graham.mcdonald@glasgow.ac.uk

Learning Outcome

- Learn how to build a system from reusable publicly available frameworks by studying its documentation.
- Understand the architecture of static analysis systems
- Understand the relationship between program code and its abstract syntax tree
- Demonstrate how static analysis can be used as a software engineering tool

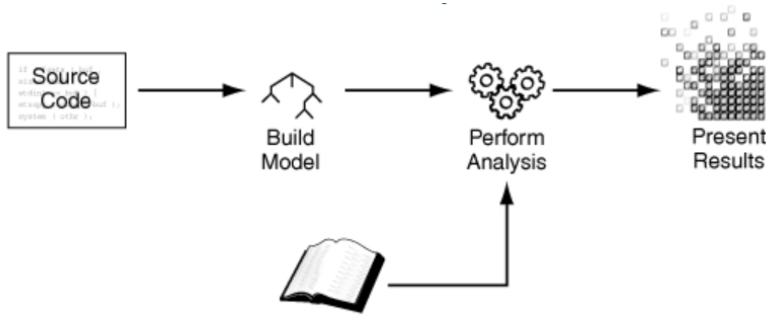
What is Static Analysis?

Don’t track everything (That’s normal interpretation)

Systematic examination of an **abstraction** of program **state space**

Ensure everything is checked in the same way

What is Static Analysis?



Specification:
Abstraction to look for in the source code

Building a model of the program

- Lexical analysis
- Parsing
- **Abstract syntax**
- Semantic Analysis
- Tracking control flow
- Tracking Dataflow
- Taint propagation

Abstract Syntax Tree (AST)

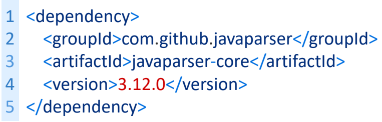
- Intermediate program representation which depicts source code as a tree
  - Defines a tree - preserves program hierarchy
  - Node types defined by class hierarchy
  - Nodes can be associated with properties
  - Generated by parser

Building a Java AST

- Popular APIs for generating AST from source code:
  - Eclipse JDT (https://www.eclipse.org/jdt/)
  - Java Parser (http://javaparser.org)

Getting Started with Java Parser

- Quickest way is to install JavaParser:
  - Create a simple Maven project in Eclipse
  - Add the following dependency to the POM.xml file



```
<dependency>
   <groupId>com.github.javaparser</groupId>
   <artifactId>javaparser-core</artifactId>
   <version>3.12.0</version>
</dependency>
```

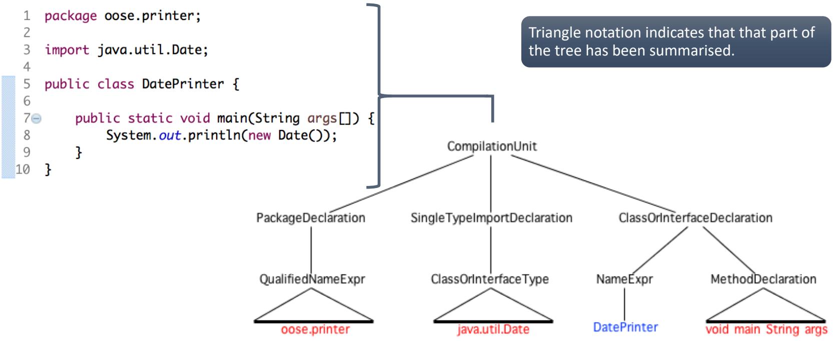
https://javaparser.org/getting-started.html

Java Parser API

- **JavaParser** class is what produces the AST from the code
- **CompilationUnit** is the root of the AST
- **Visitors** are classes which are useful to find specific parts of the AST

Java Doc (javaparser-core 3.12.0 API):
http://www.javadoc.io/doc/com.github.javaparser/javaparser-core/3.12.0

AST



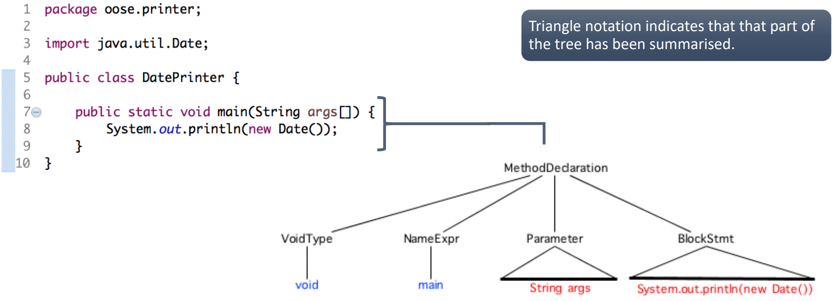
```
package oose.printer;

import java.util.Date;

public class DatePrinter {

    public static void main(String args[]) {
        System.out.println(new Date());
    }
}
```

Triangle notation indicates that that part of the tree has been summarised.

Compilation Unit

- The root is a **CompilationUnit** representing the whole file
- nodes directly connected to the root are all the top elements of a file (i.e package, import and class declarations)
- From a single class declarations multiple nodes, representing the fields or the methods of the class can be reached.

AST



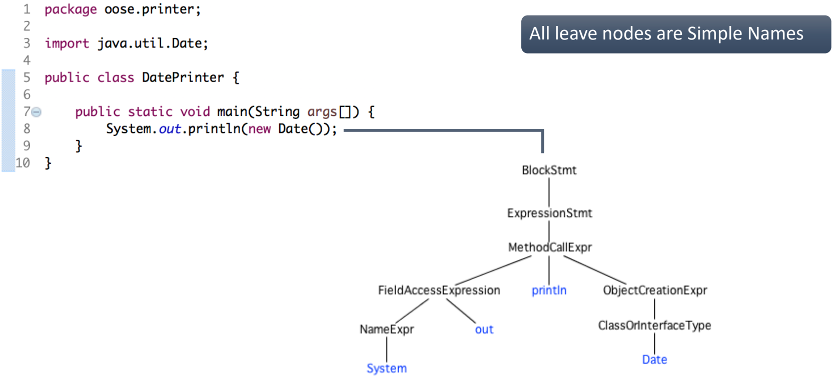
```
package oose.printer;

import java.util.Date;

public class DatePrinter {

    public static void main(String args[]) {
        System.out.println(new Date());
    }
}
```

Triangle notation indicates that that part of the tree has been summarised.

Method Declaration

Drilling down the MethodDeclaration, we can see the name, the return type and the parameter(s) for the method along with a BlockStmt, which again can be further elaborated.

AST



```
package oose.printer;

import java.util.Date;

public class DatePrinter {

    public static void main(String args[]) {
        System.out.println(new Date());
    }
}
```

All leave nodes are Simple Names

Block Statement

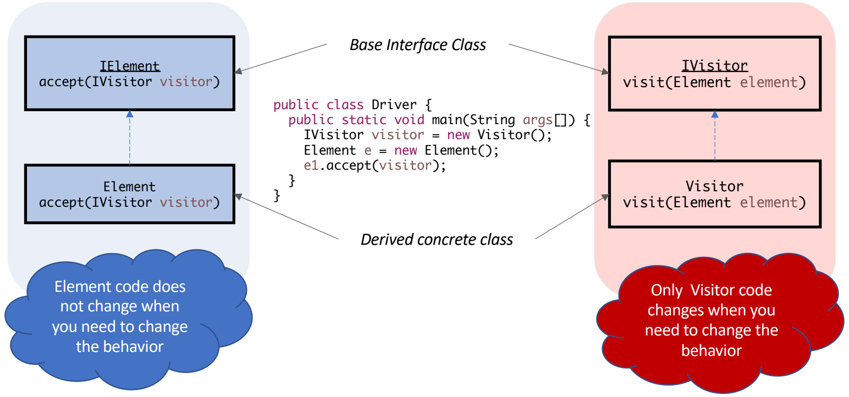
- A syntax tree gets complex quickly from small line of code.
- Each node is visitable

Visitable Nodes

1	AnnotationDeclaration	25	DoStmt	49	MarkerAnnotationExpr	73	TryStmt
2	AnnotationMemberDeclaration	26	DoubleLiteralExpr	50	MemberValuePair	74	TypeExpr
3	ArrayAccessExpr	27	EmptyMemberDeclarationclass	51	MethodCallExpr	75	TypeParameter
4	ArrayCreationExpr	28	EnclosedExpr	52	MethodDeclaration	76	UnaryExpr
5	ArrayCreationLevel	29	EnumConstantDeclaration	53	MethodReferenceExpr	77	UnionType
6	ArrayInitializerExpr	30	EnumDeclaration	54	Name	78	UnknownType
7	ArrayType	31	ExplicitConstructorInvocationStmt	55	NameExpr	79	VariableDeclarationExpr
8	AssertStmt	32	ExpressionStmt	56	NodeList	80	VariableDeclarator
9	AssignExpr	33	FieldAccessExpr	57	NormalAnnotationExpr	81	VoidType
10	BinaryExpr	34	FieldDeclaration	58	NullLiteralExpr	82	WhileStmt
11	BlockComment	35	ForeachStmt	59	ObjectCreationExpr	83	WildcardType
12	BlockStmt	36	ForStmt	60	PackageDeclaration		
13	BooleanLiteralExpr	37	IfStmt	61	Parameter		
14	BreakStmt	38	ImportDeclaration	62	PrimitiveType		
15	CastExpr	39	InitializerDeclaration	63	ReturnStmt		
16	CatchClause	40	InstanceOfExpr	64	SimpleName		
17	CharLiteralExpr	41	IntegerLiteralExpr	65	SingleMemberAnnotationExpr		
18	ClassExpr	42	IntersectionType	66	StringLiteralExpr		
19	ClassOrInterfaceDeclaration	43	JavadocComment	67	SuperExpr		
20	ClassOrInterfaceType	44	LabeledStmt	68	SwitchEntryStmt		
21	CompilationUnit	45	LambdaExpr	69	SwitchStmt		
22	ConditionalExpr	46	LineComment	70	SynchronizedStmt		
23	ConstructorDeclaration	47	LocalClassDeclarationStmt	71	ThisExpr		
24	ContinueStmt	48	LongLiteralExpr	72	ThrowStmt		

AST:

Applying the Visitors Design Pattern



```
public class Driver {
  public static void main(String args[]) {
    IVisitor visitor = new Visitor();
    Element e = new Element();
    e1.accept(visitor);
  }
}
```

Element code does not change when you need to change the behavior

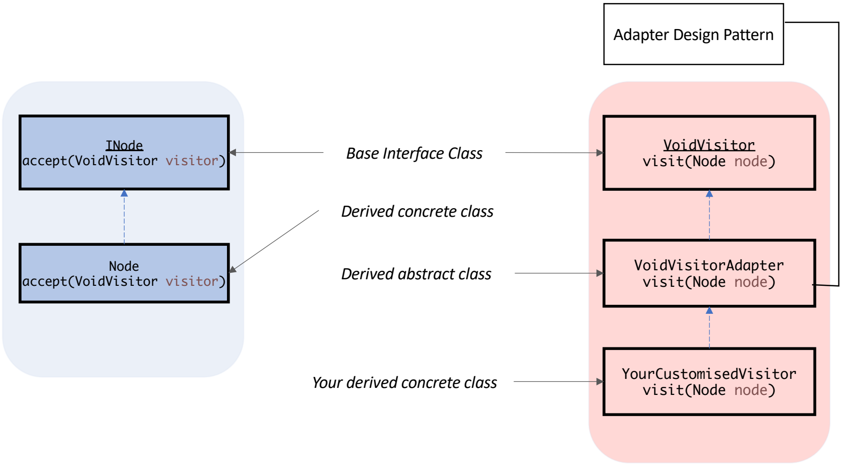
Only Visitor code changes when you need to change the behavior

This is the Visitors Design Pattern

Visitor Class

How JavaParser applies the Visitor Design Pattern

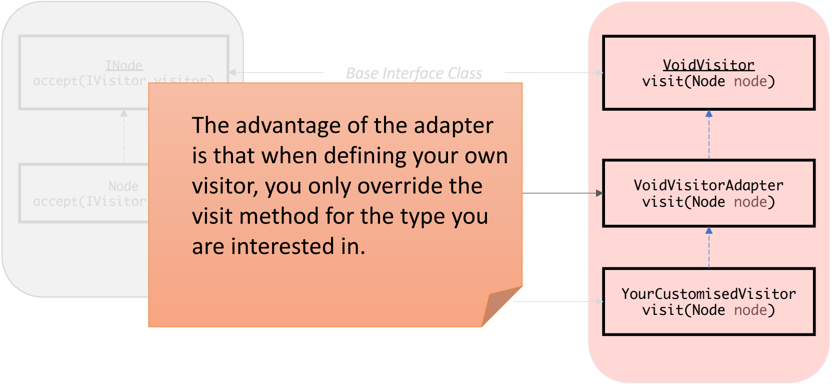
- Architecture used when the aim is **not** to modify the underlying AST



Visitor Class

How JavaParser applies the Visitor Design Pattern

- Architecture used when the aim is **not** to modify the underlying AST

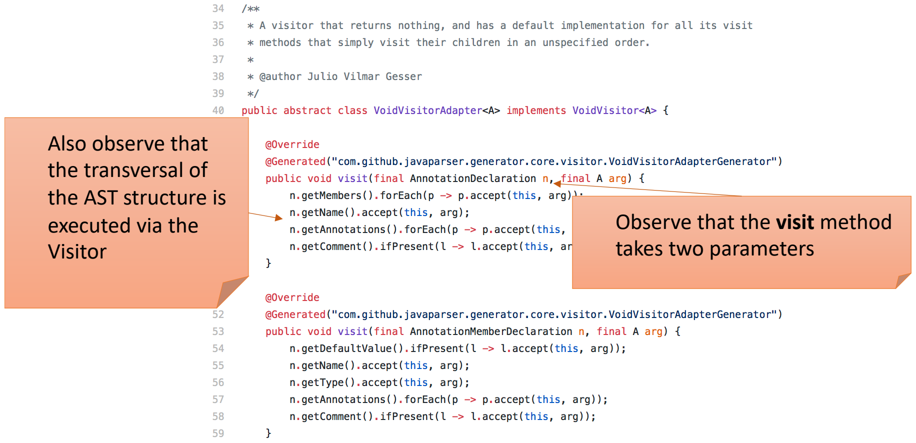


The advantage of the adapter is that when defining your own visitor, you only override the visit method for the type you are interested in.

Visitor Class

How JavaParser applies the Visitor Design Pattern

- Architecture used when the aim is **not** to modify the underlying AST



```
/**
 * A visitor that returns nothing, and has a default implementation for all its visit
 * methods that simply visit their children in an unspecified order.
 *
 * @author Julio Vilmar Gesser
 */
public abstract class VoidVisitorAdapter<A> implements VoidVisitor<A> {

    @Override
    @Generated("com.github.javaparser.generator.core.visitor.VoidVisitorAdapterGenerator")
    public void visit(final AnnotationDeclaration n, final A arg) {
        n.getMembers().forEach(p -> p.accept(this, arg));
        n.getName().accept(this, arg);
        n.getAnnotations().forEach(p -> p.accept(this,
        n.getComment().ifPresent(l -> l.accept(this, ar
    }

    @Override
    @Generated("com.github.javaparser.generator.core.visitor.VoidVisitorAdapterGenerator")
    public void visit(final AnnotationMemberDeclaration n, final A arg) {
        n.getDefaultValue().ifPresent(l -> l.accept(this, arg));
        n.getName().accept(this, arg);
        n.getType().accept(this, arg);
        n.getAnnotations().forEach(p -> p.accept(this, arg));
        n.getComment().ifPresent(l -> l.accept(this, arg));
    }
```

Also observe that the transversal of the AST structure is executed via the Visitor

Observe that the **visit** method takes two parameters

https://github.com/javaparser/javaparser/blob/master/javaparser-core/src/main/java/com/github/javaparser/ast/visitor/VoidVisitorAdapter.java

Exercise 1:

- Build a static code analyser that reports all comments in a source code showing its line number, text and whether it is an orphan comment or not

```
import java.time.LocalDateTime;

// Orphaned 1

//Attributed 1
public class TimePrinterWithComments {

   //Orphaned 2

   //Attributed 2
   public static void main(String args[]) {
      System.out.print(LocalDateTime.now());
   }
   //orphaned 3
}
```

Example of source code with orphaned and attributed comments

Solution

1. Implement a CommentReport class as the container for the properties of comments of interest

```
public class CommentReport {
  private String type;
  private String text;
  private int lineNumber;
  private boolean isOrphan;

  public CommentReport(String type, String text,
                              int lineNumber, boolean isOrphan) {
     this.type = type;
     this.text = text;
     this.lineNumber = lineNumber;
     this.isOrphan = isOrphan;
  }
  @Override
  public String toString() {
     return lineNumber + "|"+ type +"|"+ isOrphan
                        +"|"+ text.replaceAll("\\n", "").trim();
  }
}
```

Solution

2) Create a collection of CommentReport items, by mapping the fields of interest from the JavaParser Comment type.

```
public class CommentsAnalyser {

   private static final String FILE_PATH = ”…/TimePrinterWithComments.java";

   public static void main(String args []) {
      try {
         CompilationUnit cu = JavaParser.parse(new FileInputStream(FILE_PATH));
         List<Comment> comments = cu.getAllContainedComments();

         comments.forEach(com->{
              CommentReport cer = new CommentReport(com.getClass().getSimpleName(),
                                                   com.getContent(),
                                                   com.getRange().get().begin.line,
                                                  !com.getCommentedNode().isPresent());
              System.out.println(cer);
         });
   }
   catch (FileNotFoundException e) {
      e.printStackTrace();
   }
  }
}
```

Output

```
5|LineComment|true|Orphaned 1
6|LineComment|false|Attributed 1
9|LineComment|true|Orphaned 2
14|LineComment|true|orphaned 3
10|LineComment|false|Attributed 2
```

Exercise 2:

- Build a static code analyser with a Visitor that examines all the methods in a source code and prints their name and line length to the console.

```
public class Calculator {

public Calculator(){}

    public int add(int a, int b) {
        return a + b;
    }
    ...
    public static void main(String[] args) {
        Calculator myCalculator = new Calculator();
        System.out.println(myCalculator.add(5, 7));
        System.out.println(myCalculator.subtract(45, 11));
    }
}
```

Solution 1

Visitor without state:

1. Define a class that extends **VoidVisitorAdaptor.** This class also takes a type parameter, which is passes as argument into the visit method. But we make the parameter type void since the feature is not used for this solution

```
public class MethodAnalyser1 extends VoidVisitorAdapter <Void>{




}
```

Solution 1

Visitor without state:

2) Next, we defined the overriding implementation of visit. We're interested in method declarations so the first argument is of the type MethodDeclaration. The second argument is the VoidVisitorAdaptor's parameterised type.

```
public class MethodAnalyser1 extends VoidVisitorAdapter <Void>{

 @Override
   public void visit(MethodDeclaration md, Void arg) {
       super.visit(md, arg);

       String methodName = md.getName().asString();
       int begin = md.getRange().get().begin.line;
       int end =md.getRange().get().end.line;
       int length = end-begin;

       System.out.println(methodName+"|"+length);
   }

}
```

Solution 1

Visitor without state:

3. Instantiate the defined class and provide it with a ComputationUnit to operate on.

```
public class MethodMetricsDriver1 {
   private static final String FILE_PATH = "./…/….java";

   public static void main(String args []) {
        try {
           CompilationUnit cu = JavaParser.parse(new FileInputStream(FILE_PATH));

           VoidVisitor<?> methodVisitor = new MethodAnalyser1();
           methodVisitor.visit(cu, null);
        }
        catch (FileNotFoundException e) {
           e.printStackTrace();
        }
   }
}
```

Output

```
add|2
subtract|2
multiply|2
divide|7
modulo|7
main|4
```

Solution 2

Visitor with state:

1. Define a class that extends **VoidVisitorAdaptor.** This class also takes a type parameter, which is passes as argument into the visit method. **Given that we want to carry states between traversals, we make the parameter type a list of strings.**

```
public class MethodAnalyser2 extends VoidVisitorAdapter <List<String>>{




}
```

Solution 2

Visitor with state:

2) Next, we defined the overriding implementation of visit. We're interested in method declarations so the first argument is of the type MethodDeclaration. The second argument is the VoidVisitorAdaptor's parameterised type for states for each node visited.

```
public class MethodAnalyser2 extends VoidVisitorAdapter <List<String>>{

   @Override
   public void visit(MethodDeclaration md, List<String> collector) {
      super.visit(md, collector);

      String methodName = md.getName().asString();
      int begin = md.getRange().get().begin.line;
      int end =md.getRange().get().end.line;
      int length = end-begin;

      collector.add(methodName+"|"+length);
   }
}
```

Solution 2

Visitor with state:

3. Instantiate the defined class and provide it with a ComputationUnit to operate on.

```
public class MethodMetrics2 {
   private static final String FILE_PATH = "./…/….java";

   public static void main(String args []) {
      try {
         CompilationUnit cu = JavaParser.parse(new FileInputStream(FILE_PATH));

         VoidVisitor<List<String>> methodVisitor = new MethodAnalyser2();
         List<String> collector = new ArrayList<>();

         methodVisitor.visit(cu, collector);

         collector.forEach(m->{
            System.out.println(m);
         });
      }
      catch (FileNotFoundException e) {
         e.printStackTrace();
      }
   }
}
```

Output

```
add|2
subtract|2
multiply|2
divide|7
modulo|7
main|4
```

Conclusion

- Static analysis is an important software engineering tool
- One way to design a static analysis system is to use an Abstract Syntax Tree (AST) to discover interesting properties in program code.
- JavaParser is a publicly available framework for generating and analysing ASTs