Static Analysis

Software Reengineering
(COM3523 / COM6523)

The University of Sheffield

Source Code

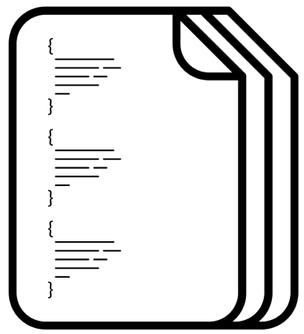
The **definitive record** of software structure and behaviour.

The primary component to be changed when the system is reengineered.

Source Code

The **definitive record** of software structure and behaviour.

The primary component to be changed when the system is reengineered.



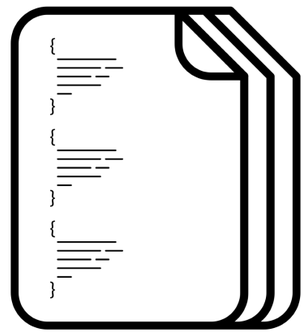
Source Code

The **definitive record** of software structure and behaviour.

The primary component to be changed when the system is reengineered.

Difficult to understand because it is:

- Big - hundreds of thousands or millions of lines of code.
- Complex - highly interconnected.
- Poorly designed - having deteriorated over decades.



Source Code

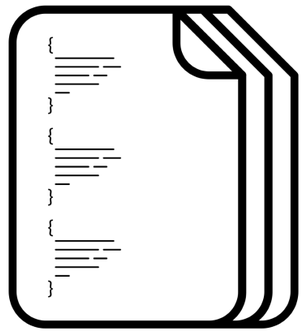
The **definitive record** of software structure and behaviour.

The primary component to be changed when the system is reengineered.

Difficult to understand because it is:

- Big - hundreds of thousands or millions of lines of code.
- Complex - highly interconnected.
- Poorly designed - having deteriorated over decades.

Loaded with latent information about what the system *should* be doing.



Source Code

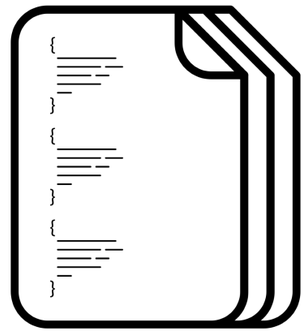
The **definitive record** of software structure and behaviour.

The primary component to be changed when the system is reengineered.

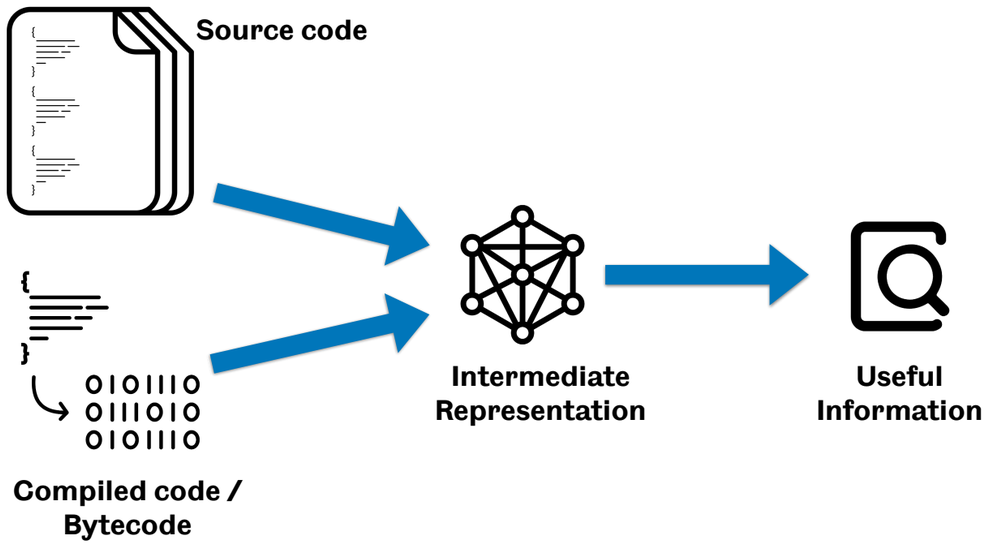
Difficult to understand because it is:

- Big - hundreds of thousands or millions of lines of code.
- Complex - highly interconnected.
- Poorly designed - having deteriorated over decades.

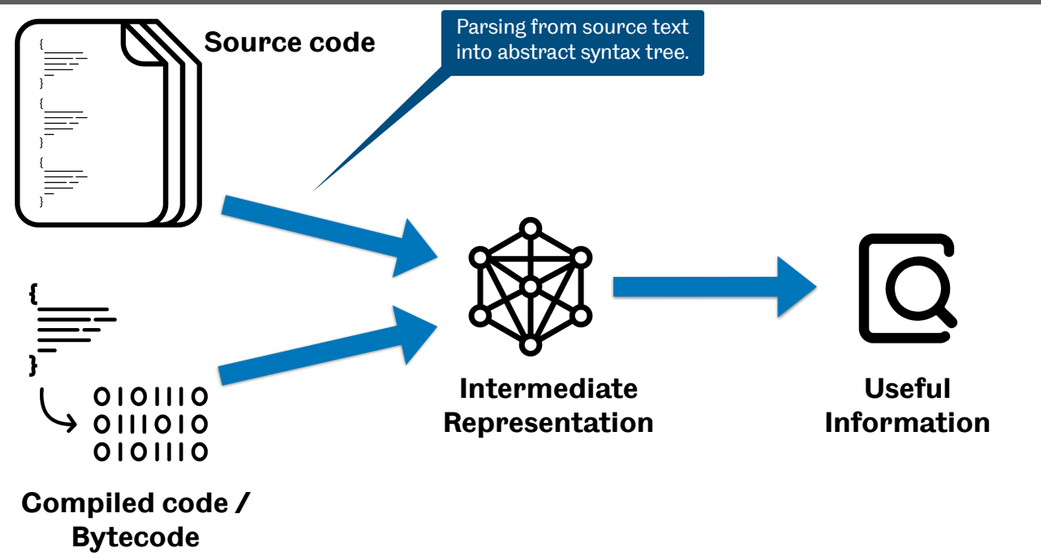
Loaded with latent information about what the system *should* be doing.



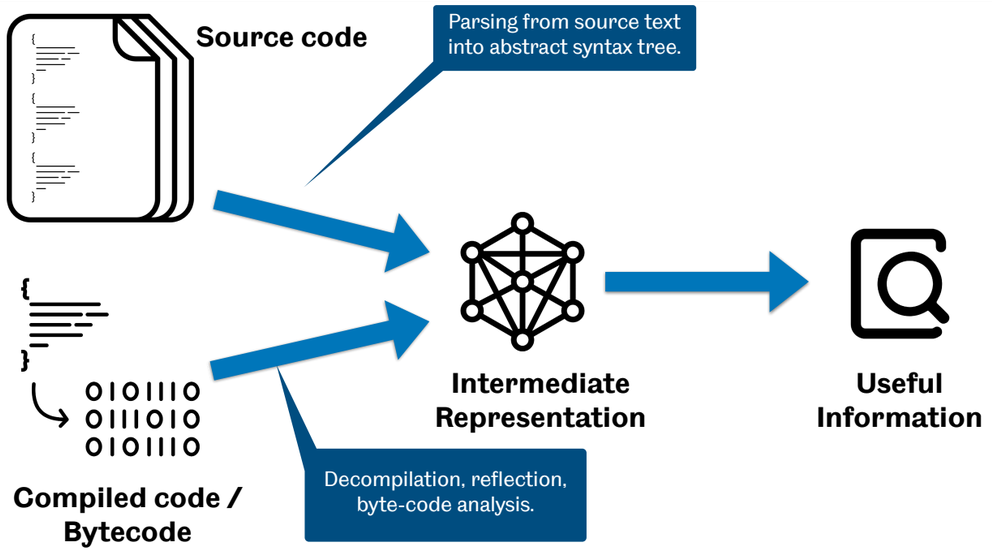
Source code analysis process



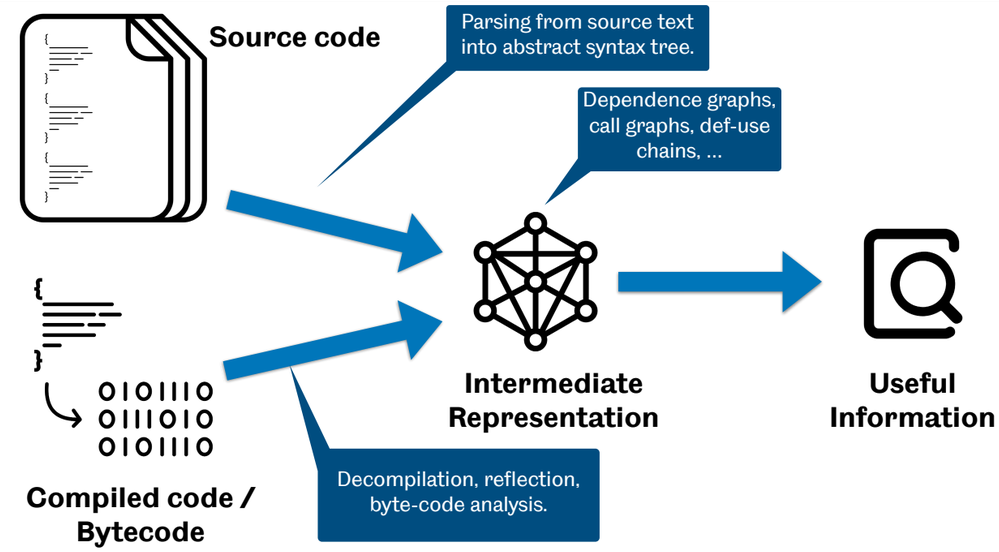
Source code analysis process



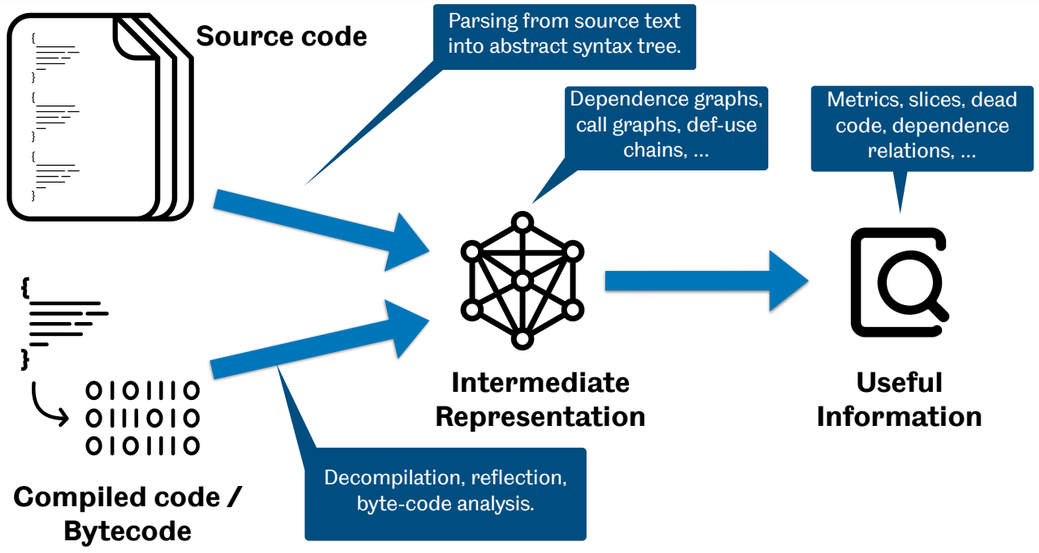
Source code analysis process



Source code analysis process



Source code analysis process



Reverse Engineering Class Diagrams via Reflection

Reflection

The ability of a program to inspect and modify its own structure and behaviour.

- Terrible idea to use as a primary programming mechanism.
- But - very useful for debugging, inspection, hot-swapping, and reverse-engineering.

Lots of languages have this ability - especially dynamically-typed ones.

- **Java**, C# (and other .NET languages), Go, Julia, Lisp, Perl, Python, R, Ruby, **Smalltalk**, ...

Built-in to languages, easy to use.

Useful if you want structure, and don't need to know anything about instructions within a method.

Class
getAnnotations():Annotation[]
getConstructors(): Constructor<?>[]
getDeclaredFields(): Field[]
getDeclaredMethods(): Method[]
getInterfaces(): Class<?>[]
getName(): String
getPackage(): Package
getSuper(): Class<?>
…

Some reflection methods in java.lang.Class

Reflection

The ability of a program to inspect and modify its own structure and behaviour.

- Terrible idea to use as a primary programming mechanism.
- But - very useful for debugging, inspection, hot-swapping, and reverse-engineering.

Lots of languages have this ability - especially dynamically-typed ones.

- **Java**, C# (and other .NET languages), Go, Julia, Lisp, Perl, Python, R, Ruby, **Smalltalk**, ...

Built-in to languages, easy to use.

Useful if you want structure, and don't need to know anything about instructions within a method.

Class
getAnnotations():Annotation[]
getConstructors(): Constructor<?>[]
getDeclaredFields(): Field[]
getDeclaredMethods(): Method[]
getInterfaces(): Class<?>[]
getName(): String
getPackage(): Package
getSuper(): Class<?>
…

Some reflection methods in java.lang.Class

Software Reengineering (COM3523 / COM6523) The University of Sheffield *Static Analysis* 5

Class Diagram

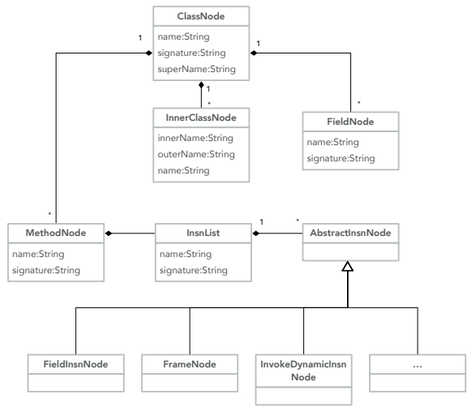
Classes are boxes.

- Class names at the top, field and method names below.

Edges represent associations:

- Inheritance (large, hollow arrow).
- Composition (a class is an attribute within another class).

All of this can be obtained via reflection.



Class Diagram Pseudocode

To create a class diagram via reflection:

- Iterate through classes in the system and for each class X:
- Create a "class" node corresponding to X.
- Load X via reflection.
- For each type of relationship from X to some other class Y:
  - Create a "class" node for Y if it doesn't exist already.
  - Create an edge X → Y (using the appropriate edge notation for the relationship type).

Reverse Engineering Class Diagrams via Decompilation / Bytecode Analysis

Call graphs

```
...
Loader.load("data.csv")
...
```

Loader

load(String)

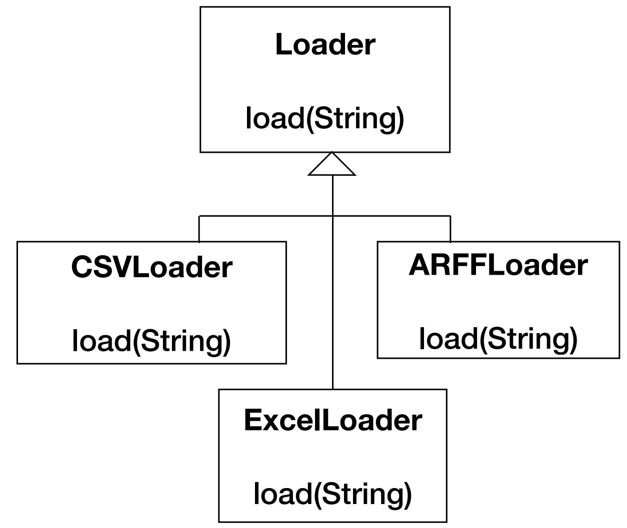
Call graphs

Call graphs

```
public static void main(String[]
args){
    Loader l = new CSVLoader();
    loadFile(l);
}

public void arffLoad(){
    Loader l = new ARFFLoader();
    loadFile(l);
}

protected void loadFile(Loader l){
    l.load("data.csv");
}
```

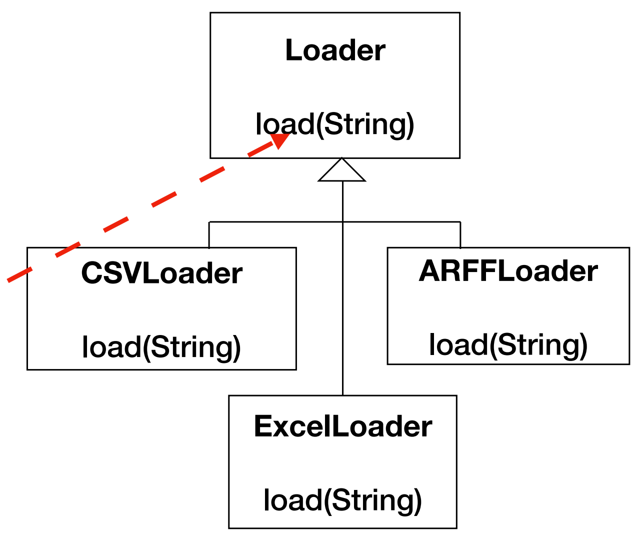


Call graphs

```
public static void main(String[]
args){
    Loader l = new CSVLoader();
    loadFile(l);
}

public void arffLoad(){
    Loader l = new ARFFLoader();
    loadFile(l);
}

protected void loadFile(Loader l){
    l.load("data.csv");
}
```

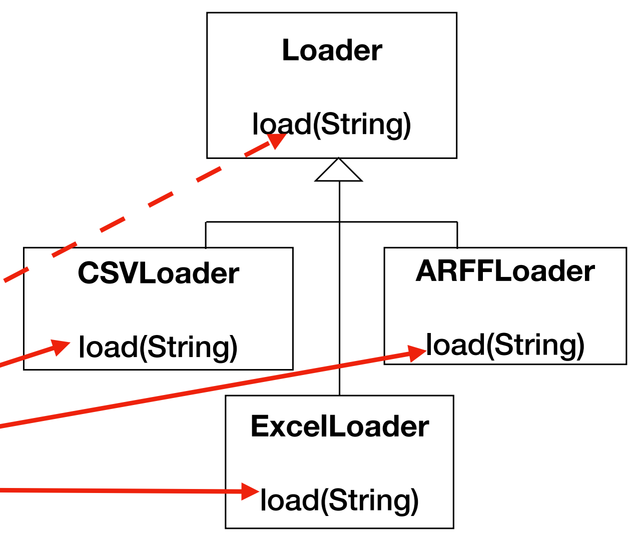


Call graphs

```
public static void main(String[]
args){
    Loader l = new CSVLoader();
    loadFile(l);
}

public void arffLoad(){
    Loader l = new ARFFLoader();
    loadFile(l);
}

protected void loadFile(Loader l){
    l.load("data.csv");
}
```



Points-To Analysis

Identify the possible destination(s) of a reference.

Lots of possible algorithms.

- Tend to trade-off efficiency against accuracy.

Class Hierarchy Analysis (CHA)

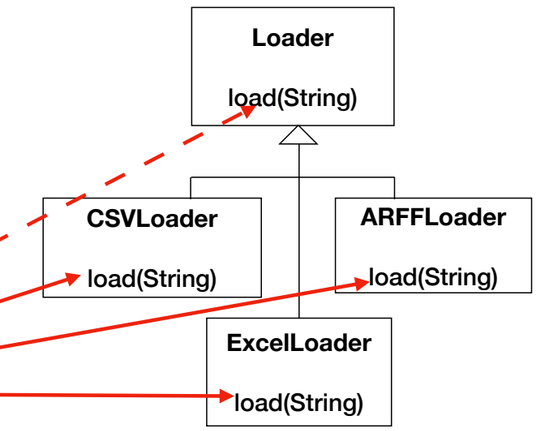
- For any class that is the target of a call, identify any sub-classes with overriding methods.
- Make these methods potential targets.

Call graphs

```
public static void main(String[]
args){
   Loader l = new CSVLoader();
   loadFile(l);
}

public void arffLoad(){
   Loader l = new ARFFLoader();
   loadFile(l);
}

protected void loadFile(Loader l){
   l.load("data.csv");
}
```



The "Fan-In" and "Fan-out" metrics

Call graph can be used to analyse inter-dependencies within a system.

Can be used to quantify this interconnectedness via **metrics**.

Fan-in:

- Number of incoming calls to a method or a class.
- Provides an idea of how "critical" or "useful" a class or method is.

Fan-out:

- Equivalent of fan-in with outgoing edges.

Can be computed from the call graph:

- For a method - number of incoming call edges!
- For a class - sum of number of incoming edges for all methods, where source of the edge lies in a different class.

Static analysis is conservative

Returns *everything* by default.

- Every single class or method in a system.
- Every single *potential* call (even calls that are infeasible in practice).

How useful is a class diagram with >700 classes?

Key strategies:

- For visual outputs (e.g. class diagrams) - **focus** on specific packages / classes.
- For non-visual outputs (e.g. call graphs) - summarise data into key metrics.

Key take-aways

Two useful technologies for source code analysis: Reflection and Bytecode analysis.

Reflection is useful for structural analysis - e.g. class diagrams.

Byte code analysis is more useful for detailed analysis - e.g. call graphs.

Overarching challenge: Information overload - static analysis is conservative.