A Language-Agnostic Framework for Dependency Graph Construction

Francesco Refolli

Outline

Assessing Software Quality

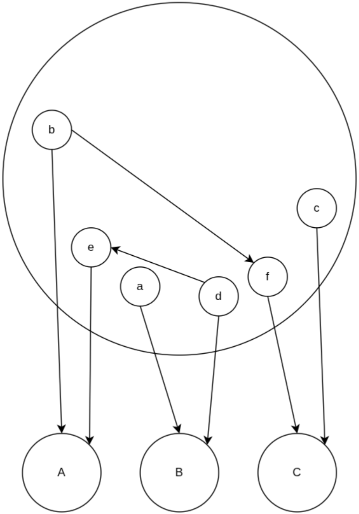
Architectural Smells

Definition

An **architectural smell** is a sign of poor design choices on the architectural level.

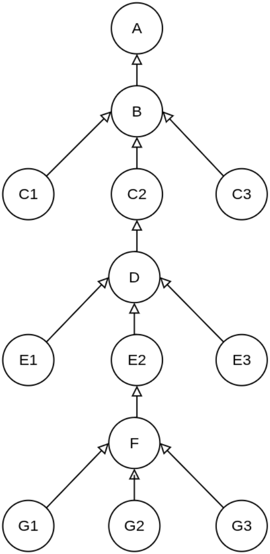
- Similar to the more known Code Smells.
- They can be detected by analyzing code shape, components and the dependency graph.
- The presence of many smells in a software project increases the estimate of **Technical Debt**, which indicates the future cost of maintenance, development and evolution.
- By Lehman's laws of software evolution we can expect it to grow over time, as bad practises effects stratify

God Component



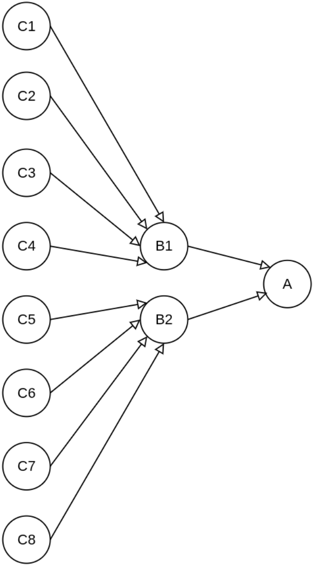
- One big component with a lot of heterogeneous sub-components inside
- Probably takes care of a lot of different concerns
- Probably has a lot of outwards dependencies and a few inwards dependencies
- Probably is a consistent percentage of the code
- Typical of Legacy Systems (especially **monolithic** architectures)

Deep Hierarchy



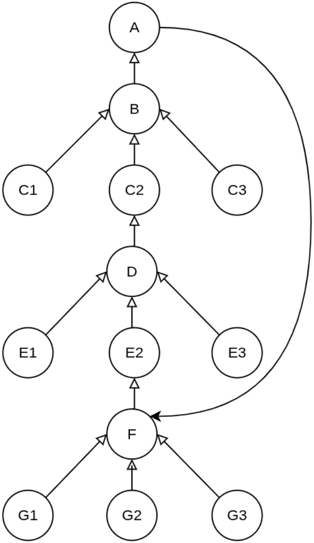
- A long inheritance chain, each level adding a bit of behaviour
- Suggests high implementation sparsity
- Suggests abuse/loss of generalization
- Typical of Object-Oriented Programming

Wide Hierarchy



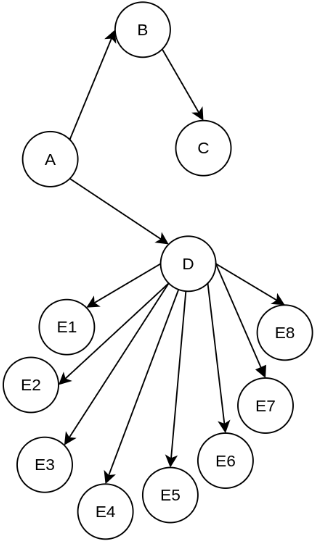
- A hierarchy chain with at each level many children
- Suggests high implementation sparsity
- Suggests abuse/loss of generalization
- Typical of Object-Oriented Programming

Cyclic Hierarchy



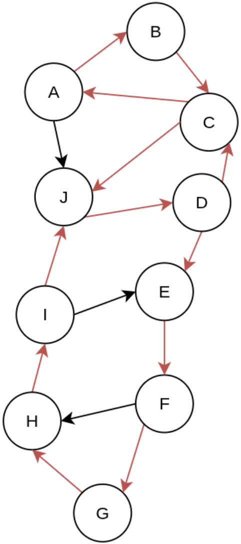
- Like the previous, but with a twist
- A parent component is actively depending on a child component
- Suggests a short-circuit of abstractions
- Typical of Object-Oriented Programming

Unstable Dependency



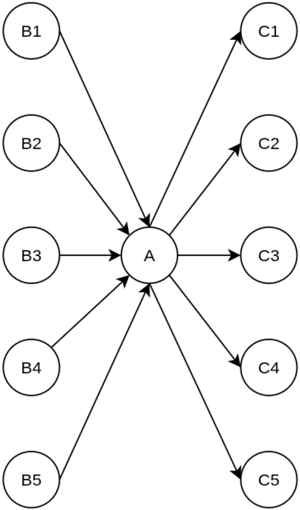
- A stable component depending on a unstable component
- (In)stability is usually defined by the ratio $\frac{Ce}{Ca+Ce}$
- Such a component is difficult to maintain stable because its dependencies have a lot of reasons to change for

Cyclic Dependency



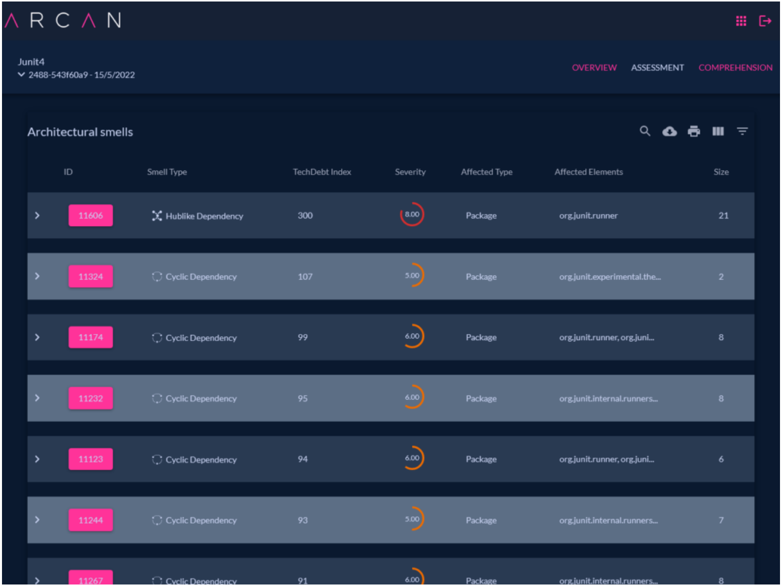
- Components involved in a cycle are difficult to change
- Typical of legacy systems

Hub-Like Dependency

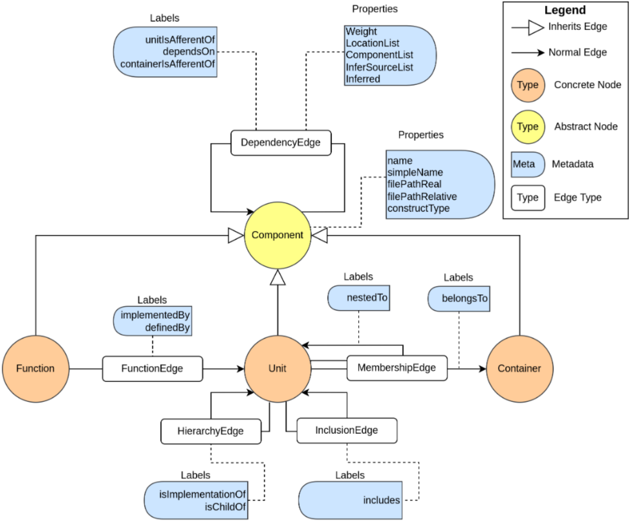


- A component with a lot of inwards and outwards dependencies
- To stable to allow for changes
- Too unstable to depend on
- Typical of many legacy library components

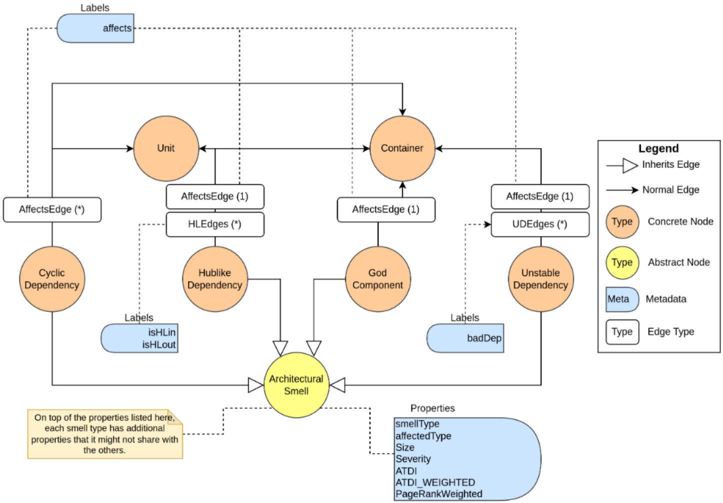
Arcan


ARCAN
Junit4
2488-543f60a9 - 15/5/2022
OVERVIEW
ASSESSMENT
COMPREHENSION
Architectural smells
ID
Smell Type
TechDebt Index
Severity
Affected Type
Affected Elements
Size
11606
Hublike Dependency
300
8.00
Package
org.junit.runner
21
11324
Cyclic Dependency
107
5.00
Package
org.junit.experimental.the...
2
11174
Cyclic Dependency
99
6.00
Package
org.junit.runner, org.juni...
8
11232
Cyclic Dependency
95
6.00
Package
org.junit.internal.runners...
8
11123
Cyclic Dependency
94
6.00
Package
org.junit.runner, org.juni...
6
11244
Cyclic Dependency
93
5.00
Package
org.junit.internal.runners...
7
11267
Cyclic Dependency
91
6.00
Package
org.junit.internal.runners...
8

The Dependency Graph



The Architectural Smell Graph



The "Tower of Babel" Problem

Language Segmentation

Computing language usage shares is not easy ...

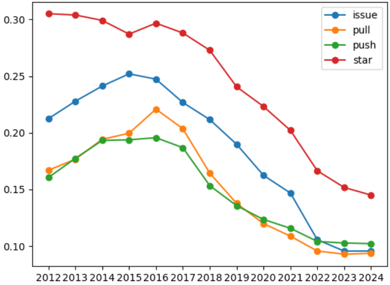


Figure: Stats of JavaScript

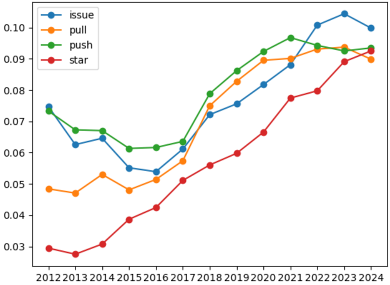
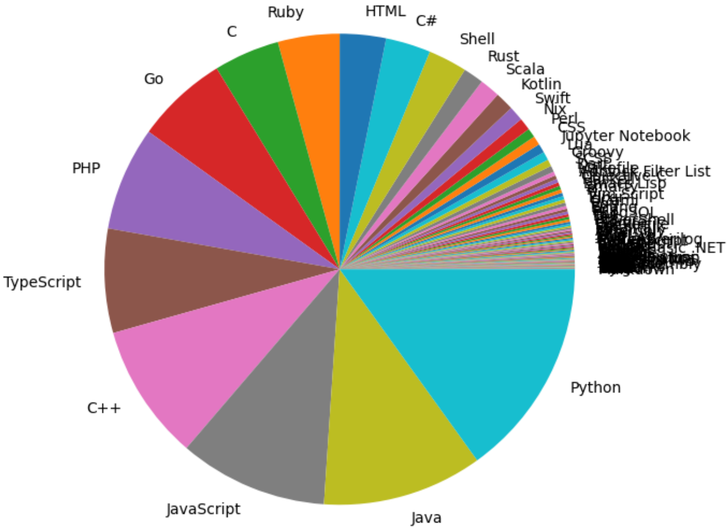


Figure: Stats of C++

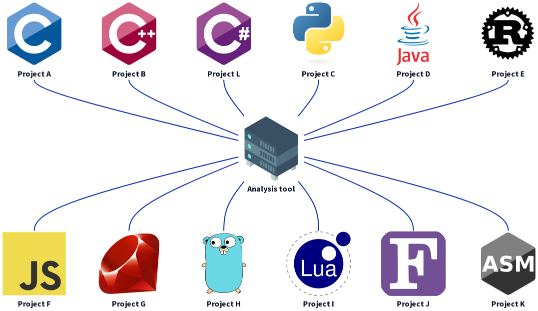
Language Segmentation

... but we can do some estimates



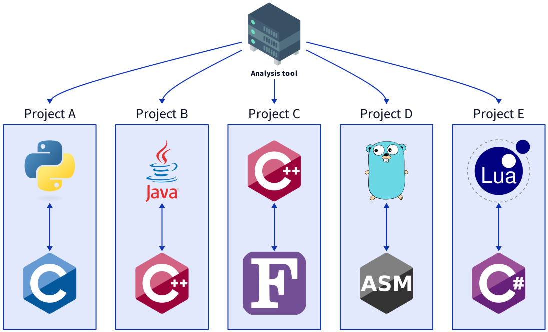
A matter of scale

- A software analysis tool need to work with many programming languages.
- Each deployed instance need every language interface to be useful
- Each language interface is a reason to change and increases maintenance costs



A matter of perspective

- A project can use more than one language, which is usually not supported by tools
- Each language interface usually need a different implementation for the same logic
- Sometimes it is easier to develop a distinct version of each language



A Bit of History

Language Frontends

- Common approach: use language frontends (e.g., `Clang`, `Eclipse JDT/CDT`) to parse code and extract data.
- APIs are heterogeneous and lack standardization.
- The *Language Server Protocol (LSP)* aims to unify such interfaces, but still has limits in query expressiveness and flexibility.
- Frontends are designed for IDEs/compilers, requiring extra adaptation for analysis tools.
- Typically heavy-weight — include many elaborations not needed by analysis tools.

Manual/Automatic Transpilation

- Some works avoid using multiple language frontends by translating source code into a single target language or DSL.
- Translation can be done manually or via transpilers.
- Works well for languages designed for transpilation (e.g., `TypeScript`, `Haxe`).
- For other languages, translation may lose semantic details (e.g., classes, namespaces, packages).
- To preserve semantics, the target language must be as expressive as the source.
- This approach remains inefficient:
  - Automatic translation between languages is still an open research problem.
  - Manual translation is time-consuming and error-prone.
  - Translated code must still be parsed and analyzed afterward.

AST/CST Translation

- Another approach: translate a **language-dependent AST** into a **language-independent AST**.
- Enables analysis tools to apply algorithms on a common, unified representation.
- Can also be applied to **Concrete Syntax Trees (CSTs)**, which capture full program syntax.
- Typically relies on language frontends (manual or generated) to perform near 1:1 translation into an *extended AST (eAST)*.
- Sometimes produces an intermediate serialized form (e.g., `XML`).

Meta Model Abstraction

- A **meta model abstraction** offers a language-agnostic alternative:
  - Represents high-level program entities — classes, structs, functions — instead of raw syntax trees.
  - Simplifies cross-language analysis by focusing on shared structural concepts.
  - Reduces dependency on language frontends and heavy parsing.
- Enables more flexible and extensible analysis frameworks built around **semantic equivalence** rather than syntax.
- Often requires custom frontends to build the meta-model from source code.

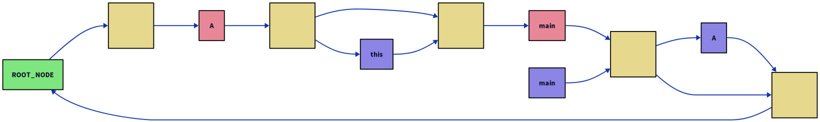
Advanced Data Structures

- Beyond traditional frontends, new data structures have been designed for specific analysis tasks (e.g., code navigation, reference resolution).
- Evaluated the use of **Stack Graphs** — composable graphs representing identifiers and their relationships — for language-independent reference resolution.
- Showed promising accuracy but faced:
    - Practical and scalability limitations.
    - Partial language-independence: graph construction still required language-specific CST queries.
- Despite limitations, the study highlighted:
    - The potential of **language-independent analysis**.
    - Its value as a guiding principle for future software analysis strategies.

ADS / Stack Graphs

- Composable graph
- Represents identifiers and scopes of source code
- Enables language agnostic reference resolution

```
class A {
  public static void main() {
  }
}
```



- A "TSSG" Grammar is composed of pairs
- Each pair matches a section of the CST with some procedural code
- Definition of nodes, edges and labels

```
(field_access object: (_)@object field: (_)@field)@this {
  node @this.scope
  node @this.pop_start
  node @this.pop_end
  edge @field.push_end -> @object.push_start
  let @this.push_start = @field.push_start
  let @this.push_end = @object.push_end
  edge @this.push_end -> @this.scope
  edge @this.pop_start -> @this.push_start
  attr (@field.push_start)
    is_reference, refkind = "accessField"
}
```

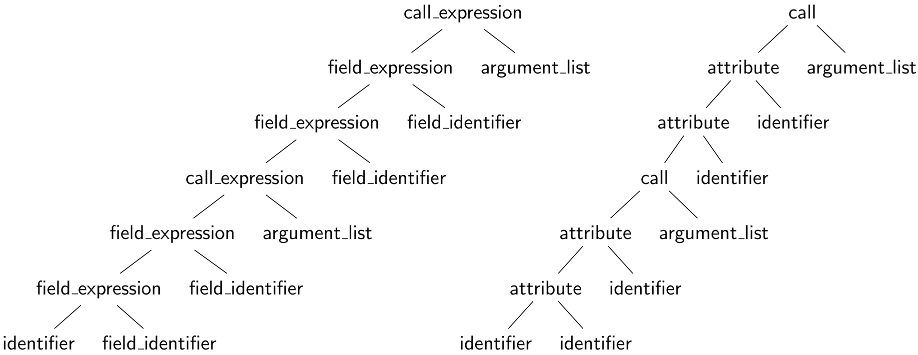
A New Solution

Very different ...

```python
import numpy

def dct(X: numpy.ndarray) -> numpy.ndarray:
  if len(X.shape) == 1:
    M = X.shape[0]
    return dct(X.reshape((1, M))).reshape((M,))

  N, M = X.shape
  Y: numpy.ndarray = numpy.zeros((N, M), dtype=numpy.float64)

  pi_over_M = numpy.pi / M
  sqrt_of_2_over_M = numpy.sqrt(2.0 / M)
  one_over_sqrt_of_two = numpy.sqrt(0.5)
  j_minus_half_from_1_to_M = (numpy.array(range(M)) + 0.5)

  for i in range(0, N):
    for k in range(0, M):
      k_pi_over_M = k * pi_over_M
      Y[i, k] = sum(
          numpy.cos(
            k_pi_over_M *
            j_minus_half_from_1_to_M) *
          X[i]) * sqrt_of_2_over_M
    Y[i, 0] = Y[i, 0] * one_over_sqrt_of_two
  return Y

def dct2(X: numpy.ndarray) -> numpy.ndarray:
  return numpy.transpose(dct(numpy.transpose(dct(X))))
```

```cpp
#include <dcct/slow_actuator.hh>

Eigen::MatrixXd dcct::SlowActuator::dct(Eigen::MatrixXd& X) {
  uint32_t N = X.rows(), M = X.cols();
  Eigen::MatrixXd Y(N, M);

  double_t pi_over_M = std::numbers::pi / (double_t) M;
  double_t sqrt_of_two_over_M = std::sqrt(2.0 / (double_t) M);
  double_t one_over_sqrt_of_two = std::sqrt(0.5);

  for (uint32_t i = 0; i < N; ++i) {
    for (uint32_t k = 0; k < M; ++k) {
      double_t sum = 0;
      double_t _k_pi_over_M = pi_over_M * k;
      for (uint32_t j = 0; j < M; j++) {
        sum += std::cos(_k_pi_over_M * (j + 0.5)) * X.coeff(i, j);
      }
      Y.coeffRef(i, k) = sum * sqrt_of_two_over_M;
    }
    Y.coeffRef(i, 0) *= one_over_sqrt_of_two;
  }
  return Y;
}

Eigen::MatrixXd dcct::SlowActuator::dct2(Eigen::MatrixXd& X) {
  Eigen::MatrixXd Y = dct(X).transpose();
  X = dct(Y).transpose();
  return X;
}
```

... Or very similar?



Tree Sitter

- Tree-sitter is a parser generator tool and an incremental parsing library.
- It can build a concrete syntax tree for a source file and efficiently update the syntax tree as the source file is edited.

- General enough to parse any programming language
- Fast enough to parse on every keystroke in a text editor
- Robust enough to provide useful results even in the presence of syntax errors
- Dependency-free so that the runtime library (which is written in pure C11) can be embedded in any application

Why Tree Sitter?

Tree Sitter is not only easy to operate, but it is also easy to implement parsers for new languages.

```
typedef [0, 0] - [0, 32]
  name: identifier [0, 8] - [0, 12]
  type: integer_type [0, 15] - [0, 31]
    size: integer [0, 23] - [0, 24]
    signed: boolean [0, 26] - [0, 30]
function [1, 0] - [1, 32]
  name: identifier [1, 3] - [1, 9]
  parameters: parameter_list [1, 9] - [1, 31]
    parameter [1, 10] - [1, 20]
      name: identifier [1, 10] - [1, 13]
      type: pointer_type [1, 15] - [1, 20]
        type: identifier [1, 16] - [1, 20]
    parameter [1, 22] - [1, 30]
      name: identifier [1, 22] - [1, 23]
      type: pointer_type [1, 25] - [1, 30]
        type: identifier [1, 26] - [1, 30]
```

Figure: Example of Lart CST

```
typedef char = integer<8, true>;
fn printf(fmt: &char, s: &char);
```

Figure: Example of Lart code

```
function: $ => seq(
  'fn',
  field('name', $.identifier),
  field('parameters', $.parameter_list),
  optional(seq('->', field('type', $._type))),
  choice(field('body', $.block), ';')
),

include: $ => seq(
  'include',
  choice(
    seq('<', field('globalpath', $.path_literal), '>'),
    seq('"', field('localpath', $.path_literal), '"')
  ), ';'
),
```

Figure: Slice of Lart grammar as TS config

The Approach

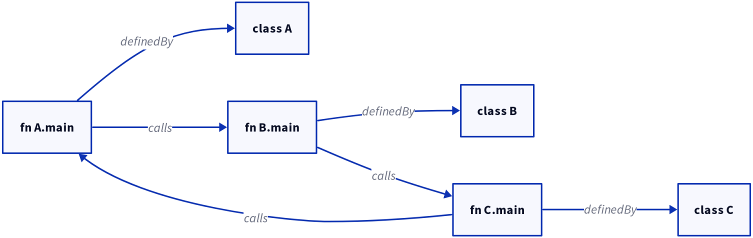
Why it works

Shenanigans

An Example / The Code

```java
class A {
  public static void main() {
    B.main();
  }
}

class B {
  public static void main() {
    C.main();
  }
}

class C {
  public static void main() {
    A.main();
  }
}
```

An Example / The Dependency Graph



Evaluation and Comparison

The Testing Framework

Tests are described with YAML manifests including constraints on nodes and edges that must be detected in the dependency graph written by the tool

```yaml
filepaths:
  - main.java
nodes:
  - name: ClassName
    kind: class
  - name: ClassName.MethodName
    kind: function
edges:
  - source: ClassName.MethodName
    sink: ClassName
    kind: definedBy
```

- node types: class, function, enum ...
- edge types: definedBy, includes, usesType, calls ...

The Dependency Detection Benchmark

Dependency type	Example code
Import	
Class import	import ModuleB.ModuleB2.Class3;
Declaration	
Instance variable	private Class3 class3;
Class variable	private static Class3 class3;
Local variable	public void method() Class3 class3;
Parameter	public void method(Class3 class3)
Return type	public Class3 method()
Exception	public void method() throws Class4throw new Class4 ("...");
Type cast	Object o = (Class3) new Object();
Call	
Instance method	variable = class3.method();
Instance method-inherited	variable = class3.methodSuper();
Class method	variable = class3.classMethod();
Constructor	new Class3();
Inner class	method variable = class3.InnerClass.method();
Interface method	interface1.interfaceMethod();
Library class method	libraryClass1.libraryMethod();
Access	
Instance variable	variable = class3.variable;
Instance variable-inherited	variable = class3.variableSuper;
Class variable	variable = Class3.classVariable;
Constant variable	variable = class3.constantVariable;
Enumeration	System.out.println(Enumeration.VAL1);
Object reference	method(class3);
Inheritance	
Extends class	public class Class1 extends Class3
Extends abstract	class Idem but in this case Class3 should be abstract.
Implements interface	public class Class1 implements Interface1
Annotation	
Class annotation	@Class3

The Dependency Detection Benchmark

Dependency type	Example code
Call	
Instance method	variable = class2.class3.method();
Instance method-inherited	variable = class2.methodSuper();
Class method	variable = class2.class3.classMethod();
Access	
Instance variable	variable = class2.class3.variable;
Instance variable-inherited	variable = class2.variableSuper();
Class variable	variable = class2.class3.classVariable;
Object reference-Reference var	variable = class2.method(class2.class3.class4);
Object reference-Return value	Object o = (Object) class2.getClass4();
Inheritance	
Extends-implements variations	public class Class1 extends Class2
	public class Class2 extends SuperClass

1

[1] The accuracy of dependency analysis in static architecture compliance checking, Pruijt et Al, Softw. Pract. Exp. 2017

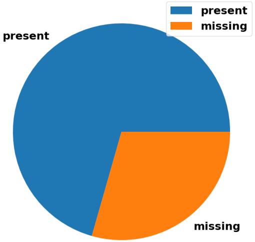


Figure: Adherence of Arcan

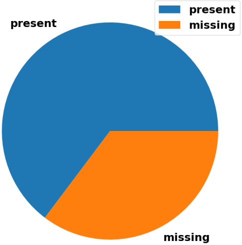


Figure: Adherence of the Stack-Graph based solution

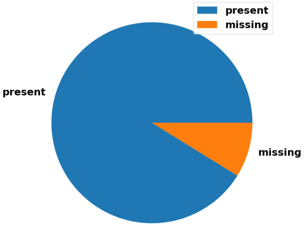
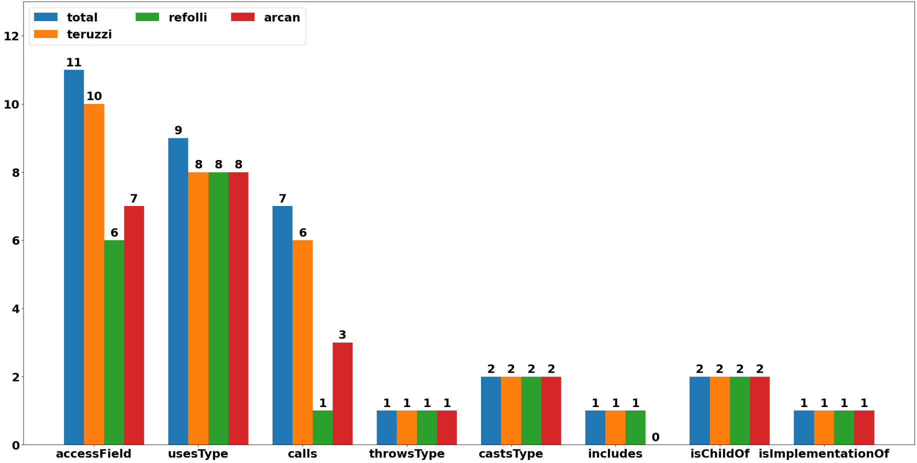


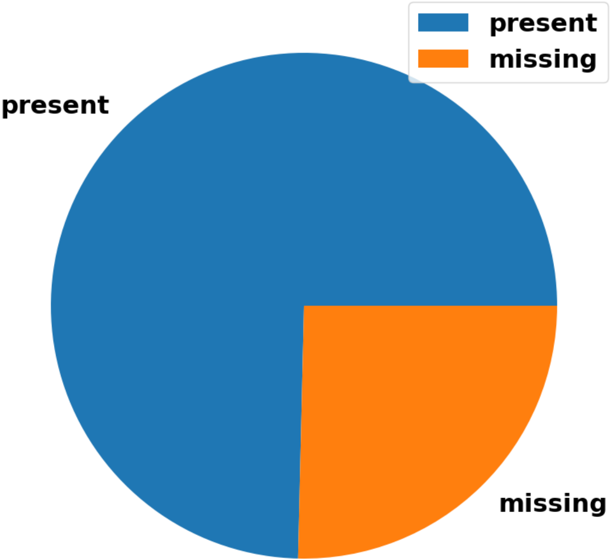
Figure: Adherence of the new proposed solution

DDB / JAVA

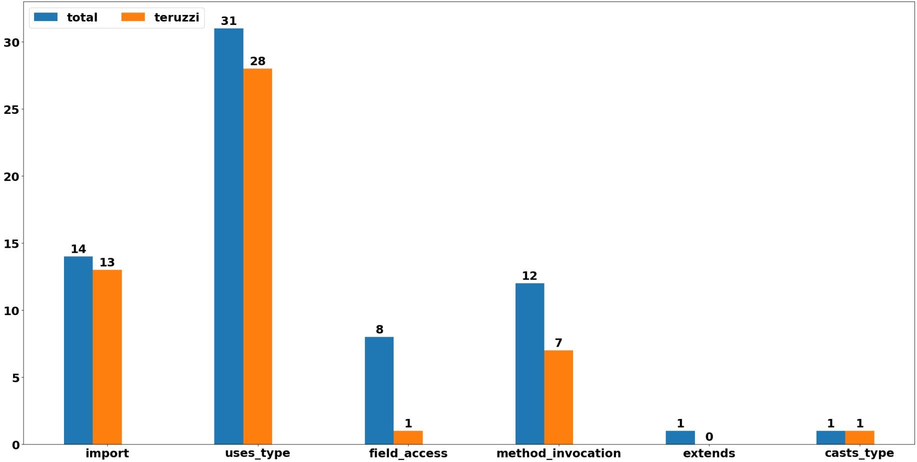


total
teruzzi
refolli
arcan
12
10
8
6
4
2
0
11
10
6
7
9
8
8
8
7
6
1
3
1
1
1
1
2
2
2
2
1
1
1
0
2
2
2
2
1
1
1
1
accessField
usesType
calls
throwsType
castsType
includes
isChildOf
isImplementationOf

DDB / RUST



DDB / RUST



total
teruzzi
0
5
10
15
20
25
30
14
13
31
28
8
1
12
7
1
0
1
1
import
uses_type
field_access
method_invocation
extends
casts_type

Comparison with the previous iteration

- The Stack-Graph based solution had a decent effectiveness when dealing with small scale projects, but efficiency is another story.
- In some cases it didn't even complete the scan within a reasonable time.
- For these reasons, the comparison is drawn mainly with Arcan in the next slides.

Project	**Tool**	**Min**	**Max**	**Mean execution time**
JUnit4	Stack-Graph	65,82	69,07	65,88
JUnit4	Arcan	13,850	17,079	14,611
JUnit5	Stack-Graph	132,87	134,75	134,44
JUnit5	Arcan	42,542	47,613	44,186
ANTLR	Stack-Graph	171,65	175,49	172,64
ANTLR	Arcan	19,222	20,140	19,691
Fastjson	Stack-Graph	N/A	N/A	N/A
Fastjson	Arcan	66,932	71,094	69,071

Comparison with Arcan / Accuracy

Comparison with Arcan / Similarity

Comparison with Arcan / Efficiency

Conclusions

Open Problems

Future Works

Bibliography I

[1] F. Refolli, D. Sas, and F. A. Fontana, "Lessons learned from implementing a language-agnostic dependency graph parser," in *Proceedings of the 20th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE*, INSTICC, SciTePress, 2025, pp. 484–491, ISBN: 978-989-758-742-9. DOI: 10.5220/0013277600003928.

[2] K. Weiss and C. Banse, "A language-independent analysis platform for source code," *CoRR*, 2022. DOI: 10.48550/ARXIV.2203.08424. arXiv: 2203.08424. [Online]. Available: https://doi.org/10.48550/arXiv.2203.08424.

[3] V. J. Marin and C. R. Rivero, "Towards a framework for generating program dependence graphs from source code," in *Proceedings of the 4th ACM SIGSOFT International Workshop on Software Analytics*, ser. SWAN 2018, Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, 30–36, ISBN: 9781450360562. DOI: 10.1145/3278142.3278144.

Bibliography II

[4] M. L. Collard, M. J. Decker, and J. I. Maletic, "Srcml: An infrastructure for the exploration, analysis, and manipulation of source code: A tool demonstration," in *2013 IEEE International conference on software maintenance*, IEEE, 2013, pp. 516–519.

[5] S. Ducasse, N. Anquetil, M. U. Bhatti, A. Cavalcante Hora, J. Laval, and T. Girba, "MSE and FAMIX 3.0: an Interexchange Format and Source Code Model Family," Research Report, Nov. 2011. [Online]. Available: https://inria.hal.science/hal-00646884.

[6] G. Antoniol, M. Di Penta, G. Masone, and U. Villano, "Xogastan: Xml-oriented gcc ast analysis and transformations," in *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*, 2003, pp. 173–182. DOI: 10.1109/SCAM.2003.1238043.

[7] S. Tichelaar, S. Ducasse, and S. Demeyer, "Famix and xmi," in *Proceedings Seventh Working Conference on Reverse Engineering*, IEEE, 2000, pp. 296–298. DOI: 10.1109/WCRE.2000.891485.