A New Generation of Class Blueprint

Nour J. Agouf*, Stéphane Ducasse, Anne Etien, Michele Lanza.

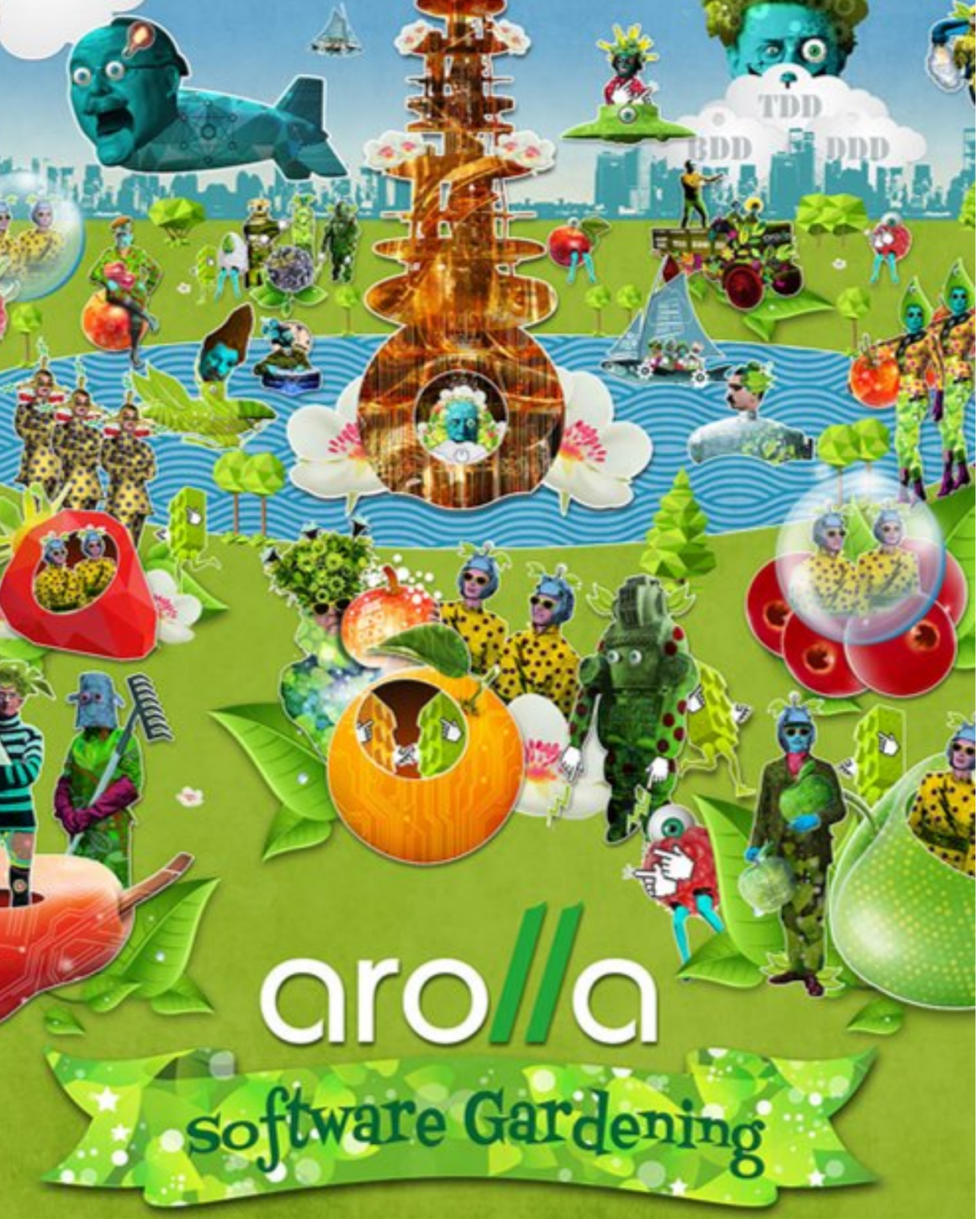
*__Arolla__, Inria, Univ. Lille, CNRS, Centrale Lille,

UMR 9189 - CRIStAL-France

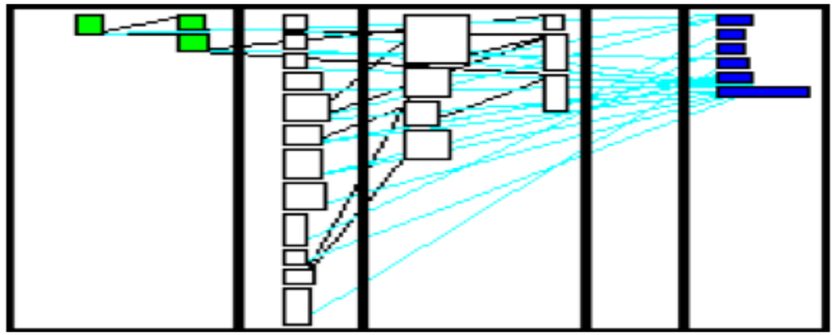
nour-jihene.agouf@arolla.fr

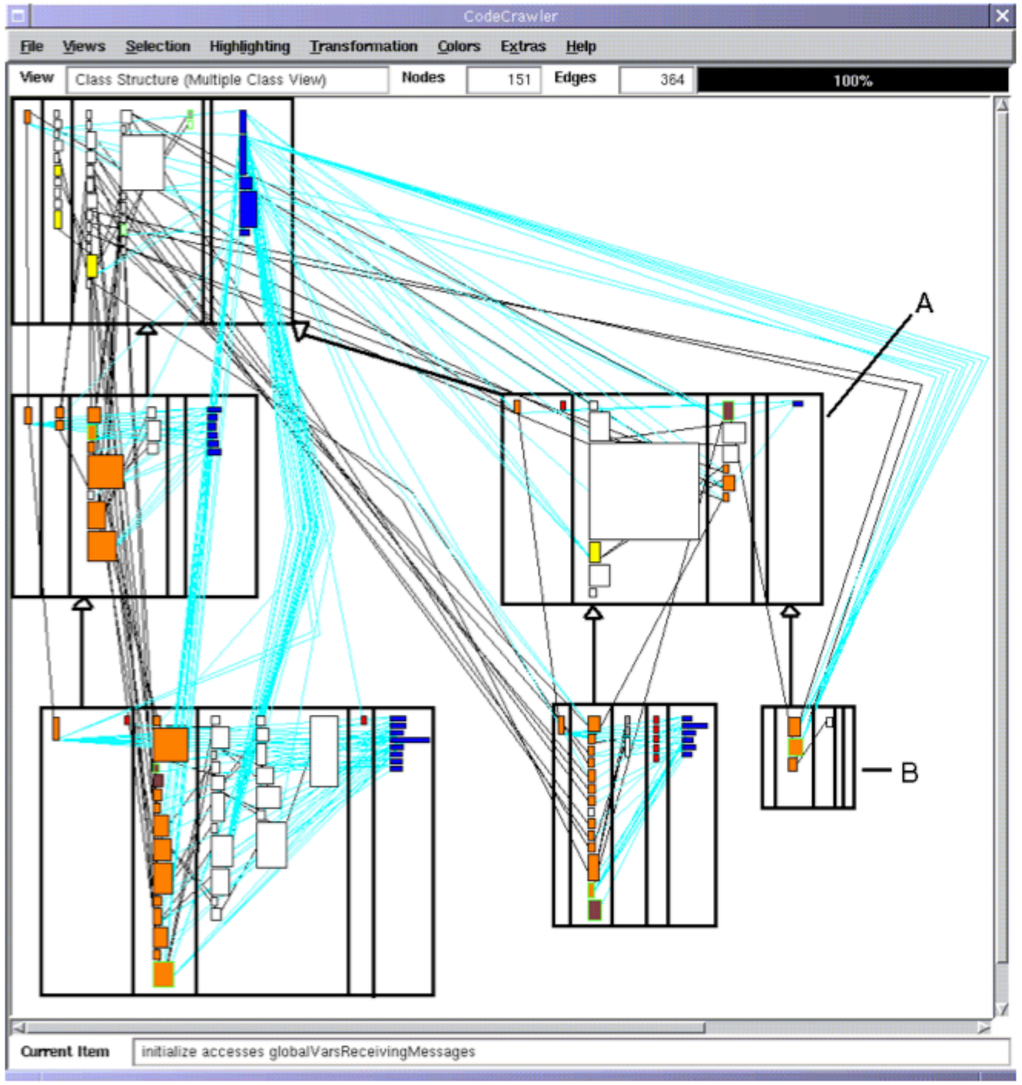
WE ARE CRAFTERS

AROLLA IS A CONSULTING COMPANY SPECIALIZED IN THE ADVANCED TECHNIQUES OF SOFTWARE DEVELOPMENT: CLEAN CODE, TDD, BDD, LEGACY REMEDIATION, etc.



The time spent on reading and understand source code is over 70% of the maintenance task





The Class Blueprint: Visually Supporting the Understanding of Classes

Stéphane Ducasse and Michele Lanza, *Member*, *IEEE*

Abstract—Understanding source code is an important task in the maintenance of software systems. *Legacy systems* are not only limited to procedural languages, but are also written in object-oriented languages. In such a context, understanding classes is a key activity as they are the cornerstone of the object-oriented paradigm and the primary abstraction from which applications are built. Such an understanding is however difficult to obtain because of reasons such as the presence of late binding and inheritance. A first level of class understanding consists of the understanding of its overall structure, the control flow among its methods, and the accesses on its attributes. We propose a novel visualization of classes called *class blueprint* that is based on a semantically enriched visualization of the internal structure of classes. This visualization allows a software engineer to build a first mental model of a class that he validates via opportunistic code-reading. Furthermore, we have identified *visual patterns* that represent recurrent situations and as such convey additional information to the viewer. The contributions of this article are the class blueprint, a novel visualization of the internal structure of classes, the identification of visual patterns, and the definition of a vocabulary based on these visual patterns. We have performed several case studies of which one is presented in depth, and validated the usefulness of the approach in a controlled experiment.

Index Terms—Object-oriented programming, software visualization, reverse engineering, visual patterns, smalltalk.

1 Introduction

It has been measured that, in the maintenance phase, software professionals spend at least half of their time analyzing software to understand it [1] and that code reading is a viable verification and testing strategy [2], [3]. Sommerville [4] and Davis [5] estimate that the maintenance of a software system accounts for 50 to 75 percent of its overall cost. These findings show that understanding source code is an important task in the maintenance of software systems.

Legacy systems are not only limited to procedural languages, but are also written in object-oriented languages. Contrary to what one may think, the object-oriented programming paradigm has exacerbated this problem, since in object-oriented systems the domain model of the application is distributed across the whole system and the behavior is distributed accross inheritance hierarchies with late-binding [6], [7], [8].

Reading object-oriented code is more difficult than reading procedural code [9]: In addition to the difficulties introduced by the technical aspects of object-oriented languages such as inheritance and polymorphism [6], the reading order of a class' source code is not relevant as it was in most of the procedural languages where the order of the procedures was important and the use of forward declarations required. This lack of reading order is emphasized in languages such as Smalltalk, a language based upon a powerful integrated development environment (IDE) in which the concept of source files is used only for external code storage, but seldom for code editing. Moreover, even for file-based languages like Java, IDEs such as Eclipse[1] are literally eclipsing the importance of source files and putting forward a *code browsing* practice as in Smalltalk.

Understanding classes is of key importance as they are the cornerstone of the object-oriented paradigm and the primary abstraction from which applications are built. Therefore, there is a definitive need to support the understanding of classes and their internal structure. In the past, work has been done to support the understanding of object-oriented applications [10], [11], [12]. Some other work focused on analyzing the impact of graphical notation to support program understanding based on control-flow [3]. Such approaches are powerful for supporting the identification of design patterns, but too generic and not fine-grained enough for the specific purpose of class understanding.

In this article, we present an approach to ease the understanding of classes by visualizing a semantically augmented call and access-graph of the methods and attributes of classes. Our approach only takes into account the internal static structure of a class and focuses on the way methods call each other and access attributes, and the way the classes use inheritance, i.e., we leave out the runtime behavior of a system.

We have coined the term *class blueprint*, a visualization of a semantically augmented call-graph and its specific semantics-based layout. The objective of our visualization is to help a programmer to develop a mental model of the classes he browses and to offer support for reconstructing the logical flow of method calls. Our approach targets the

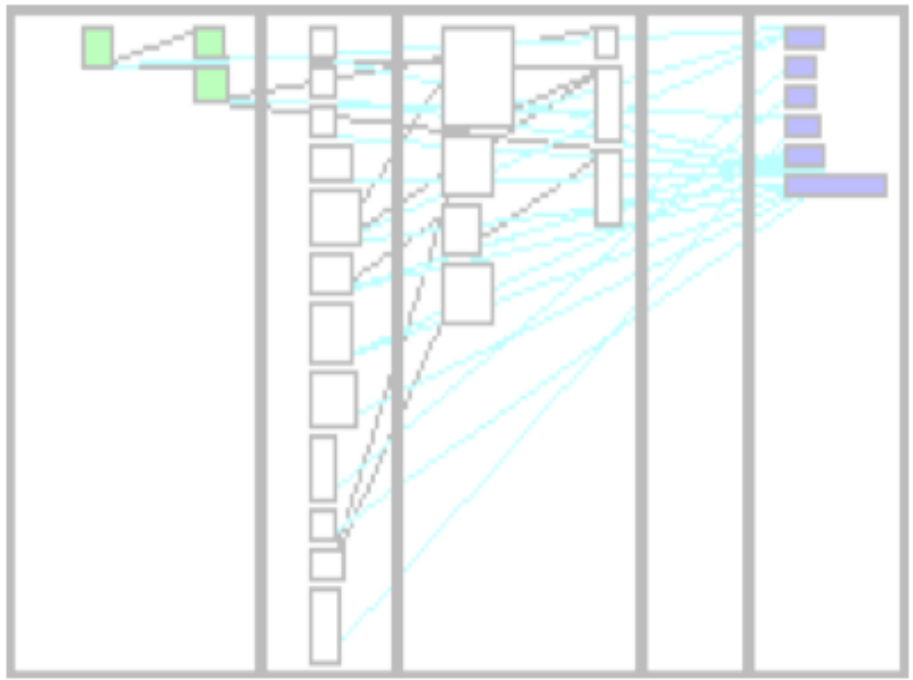
- *S. Ducasse is with the Software Composition Group, Institute of Applied Mathematics and Computer Science, University of Bern, Neunrueckstrasse 10, 3012 Bern, Switzerland. E-mail: ducasse@iam.unibe.ch.*
- *M. Lanza is with the Faculty of Informatics, University of Lugano, Via G. Buffi 13, 6900 Lugano, Switzerland. E-mail: michele.lanza@unisi.ch.*

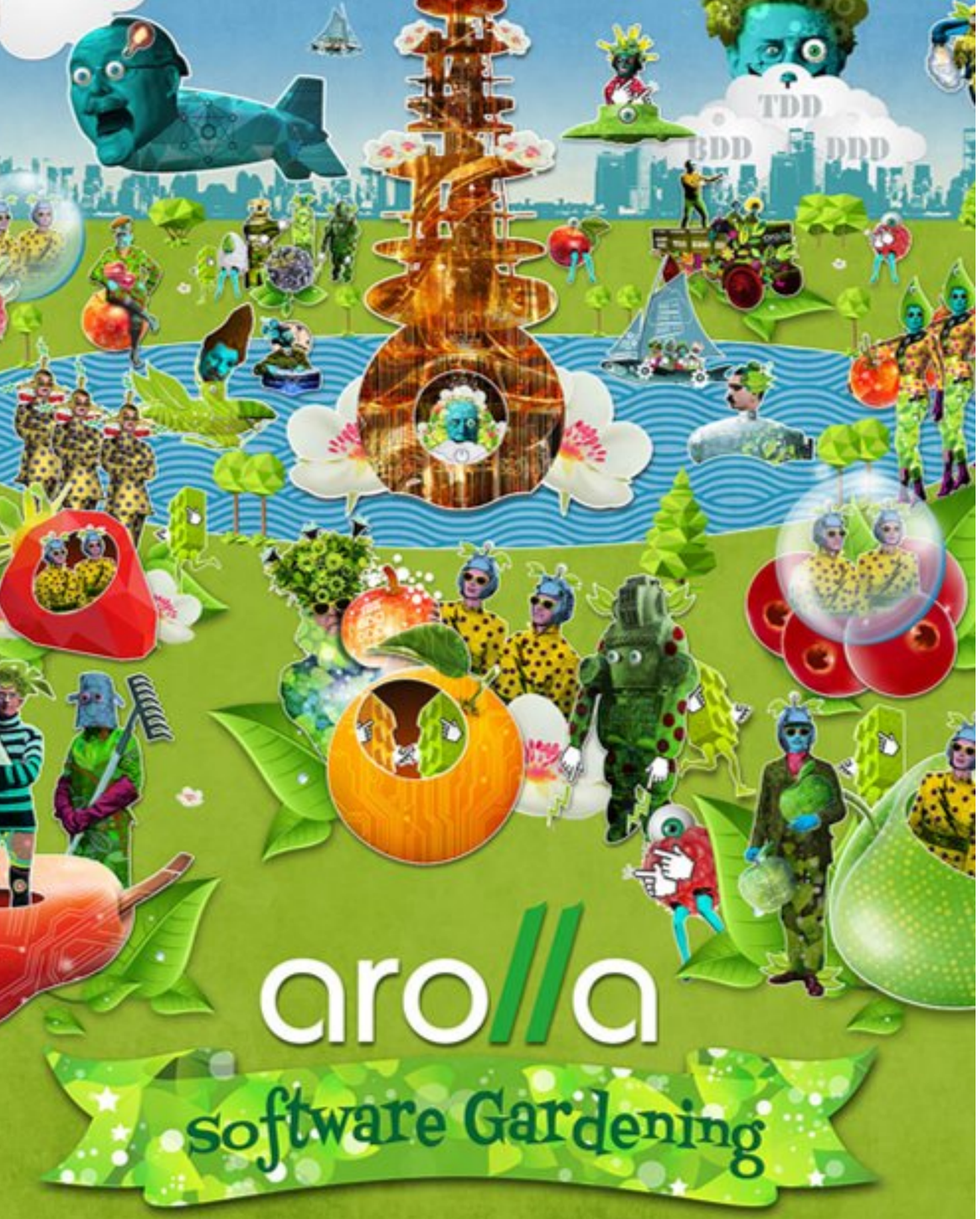
Manuscript received 1 June 2004; revised 8 Oct. 2004; accepted 22 Dec. 2004; published online 9 Feb. 2005.
Recommended for acceptance by J. Knight.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0103-0604.

1. See http://www.eclipse.org/ for more information.

The Class Blueprint

Is a representation of static data of classes in object-oriented programming. It gives an overview of a *taste* of the class, focusing on methods classification and displaying their call-flow.

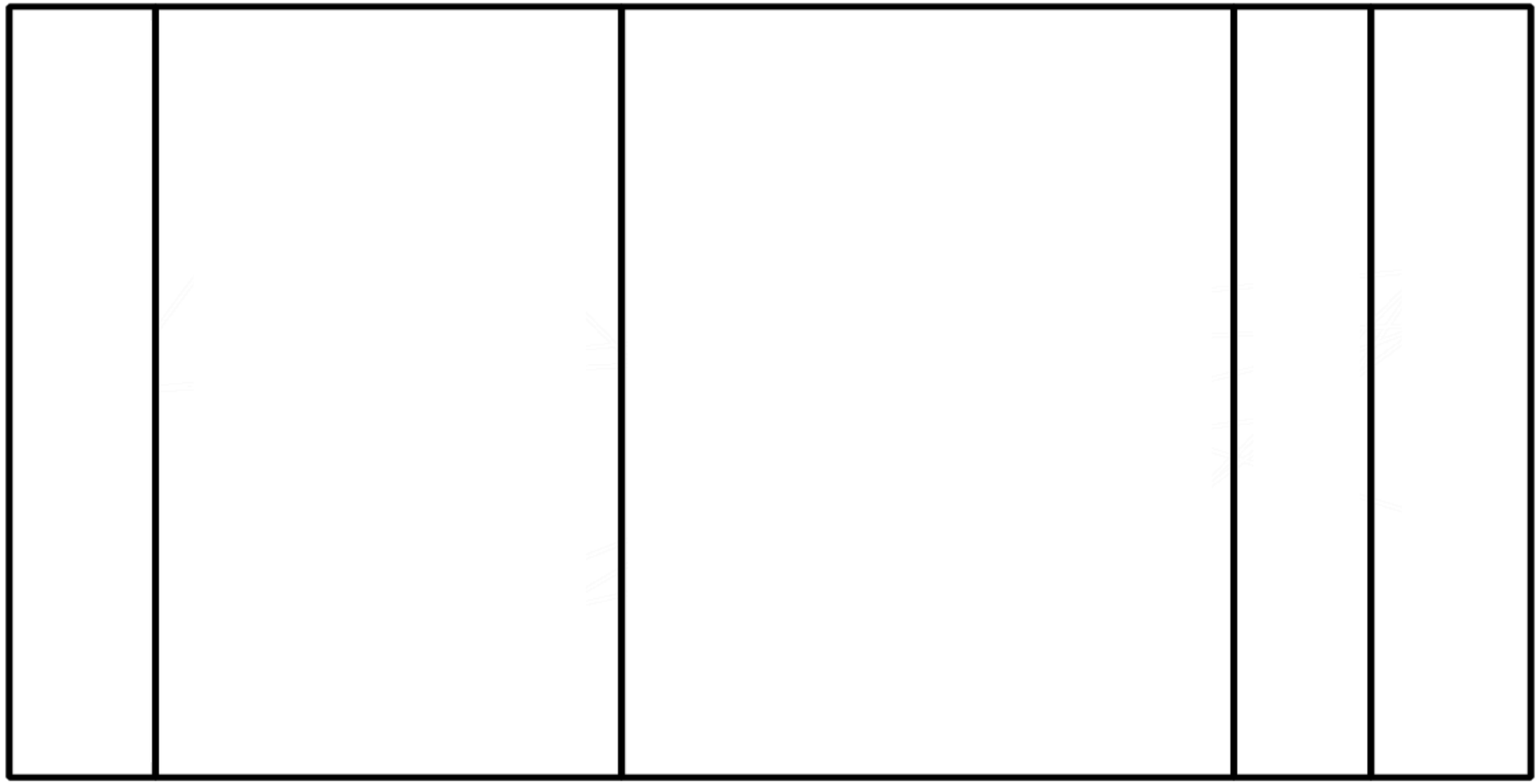




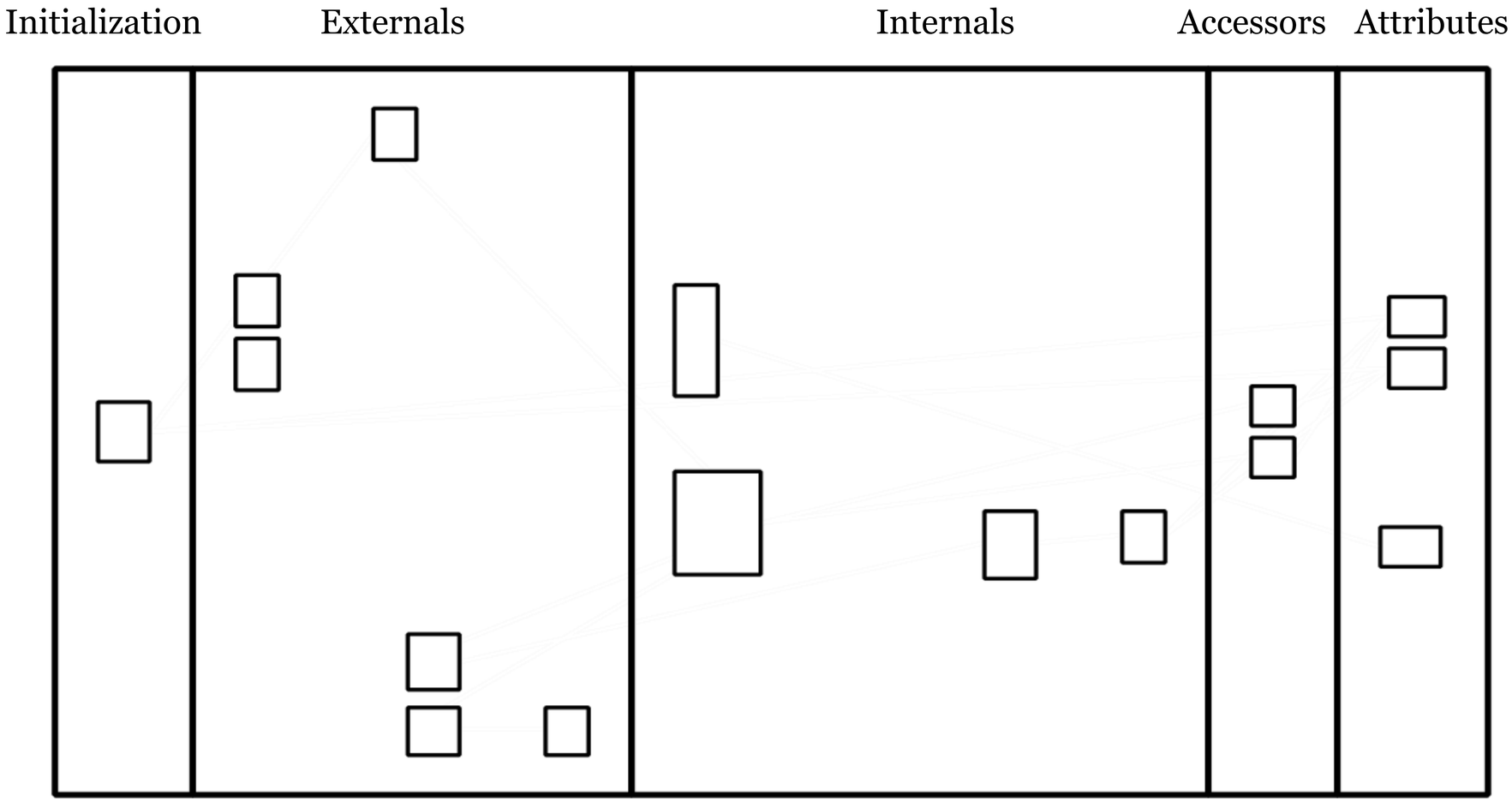
Class Blueprint V1

V1: Methods/attributes classification

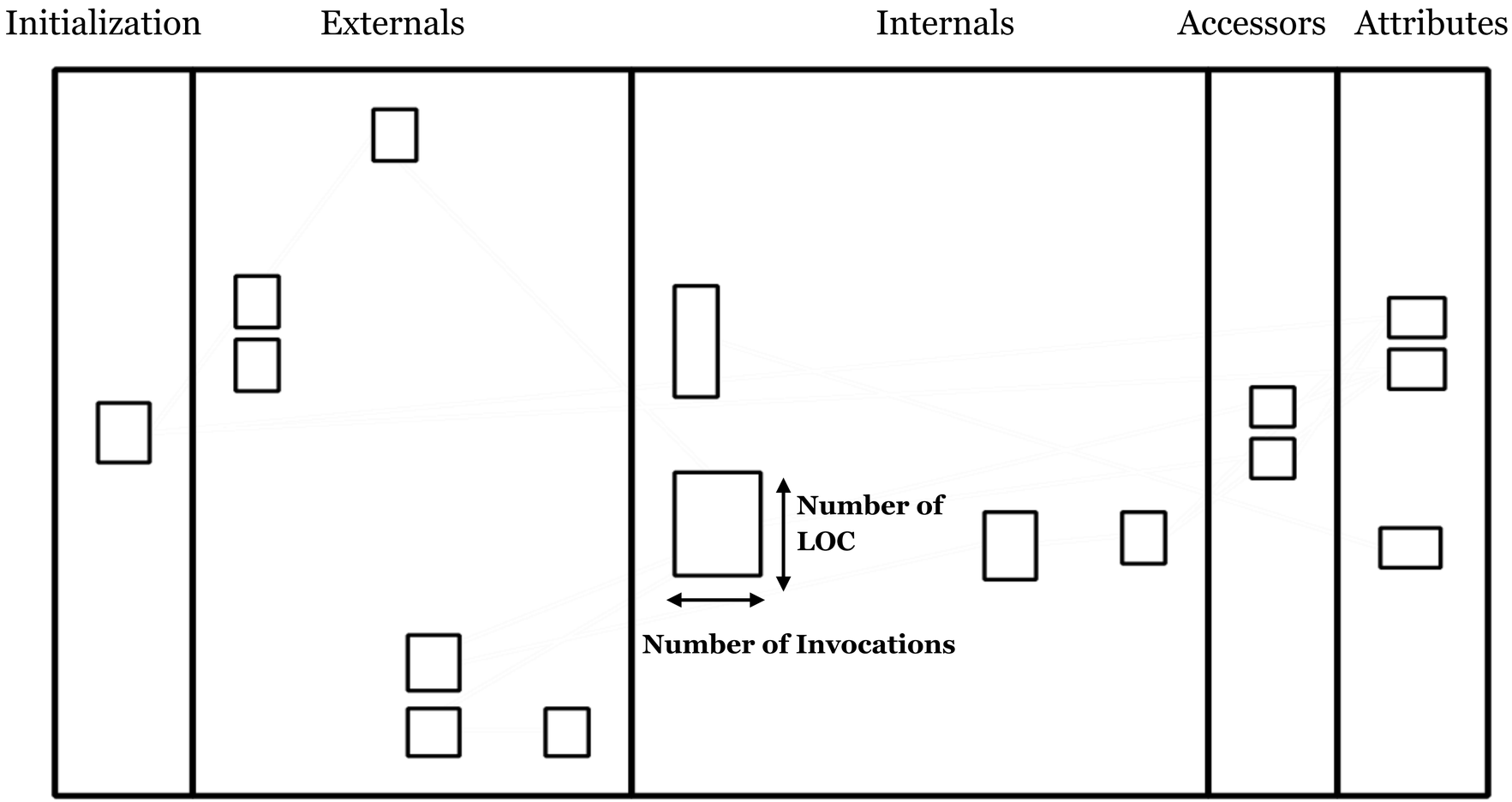
Initialization Externals Internals Accessors Attributes



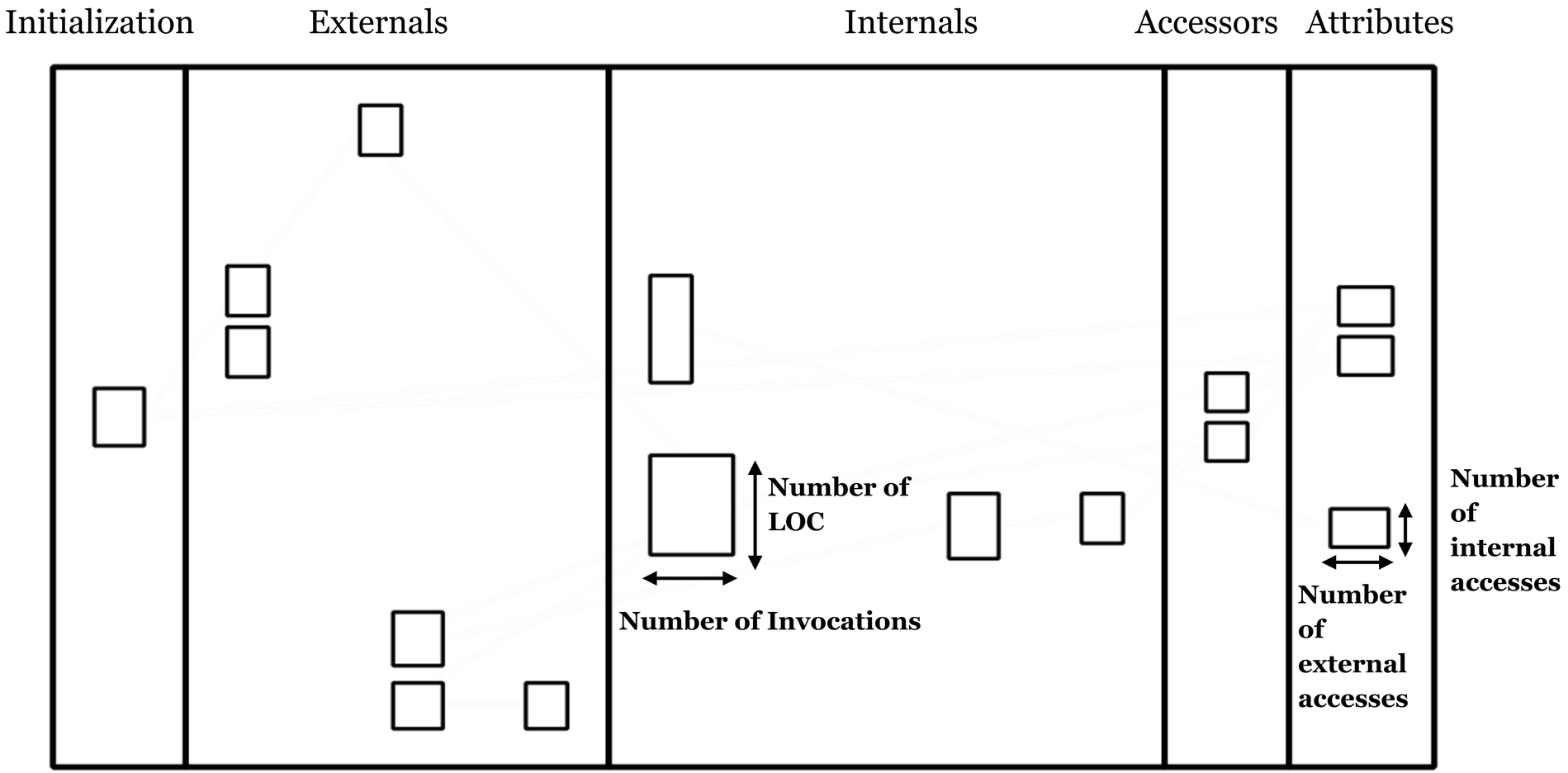
V1: Methods/attributes nodes



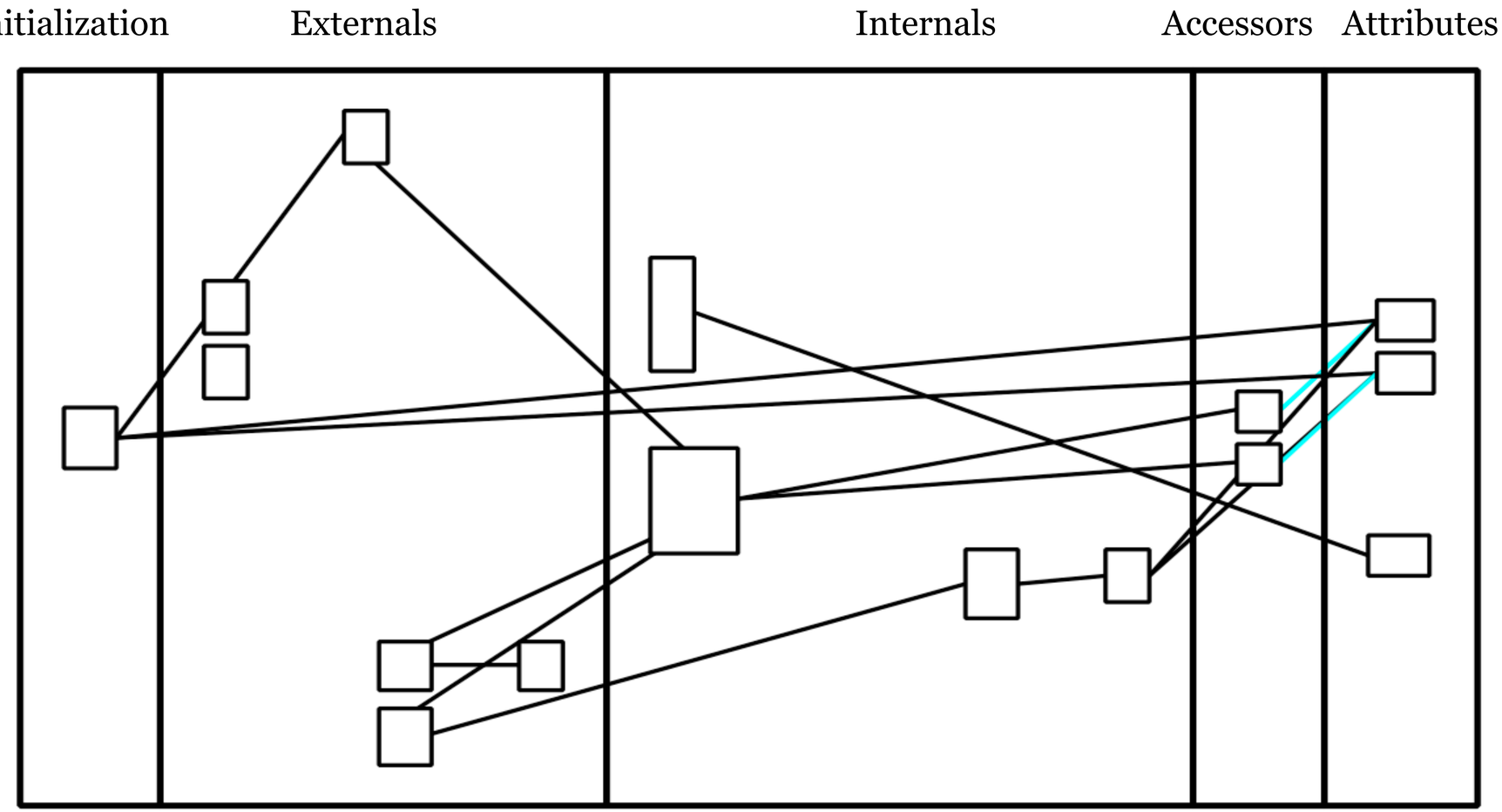
V1: Method node metrics



V1: Attribute node metrics



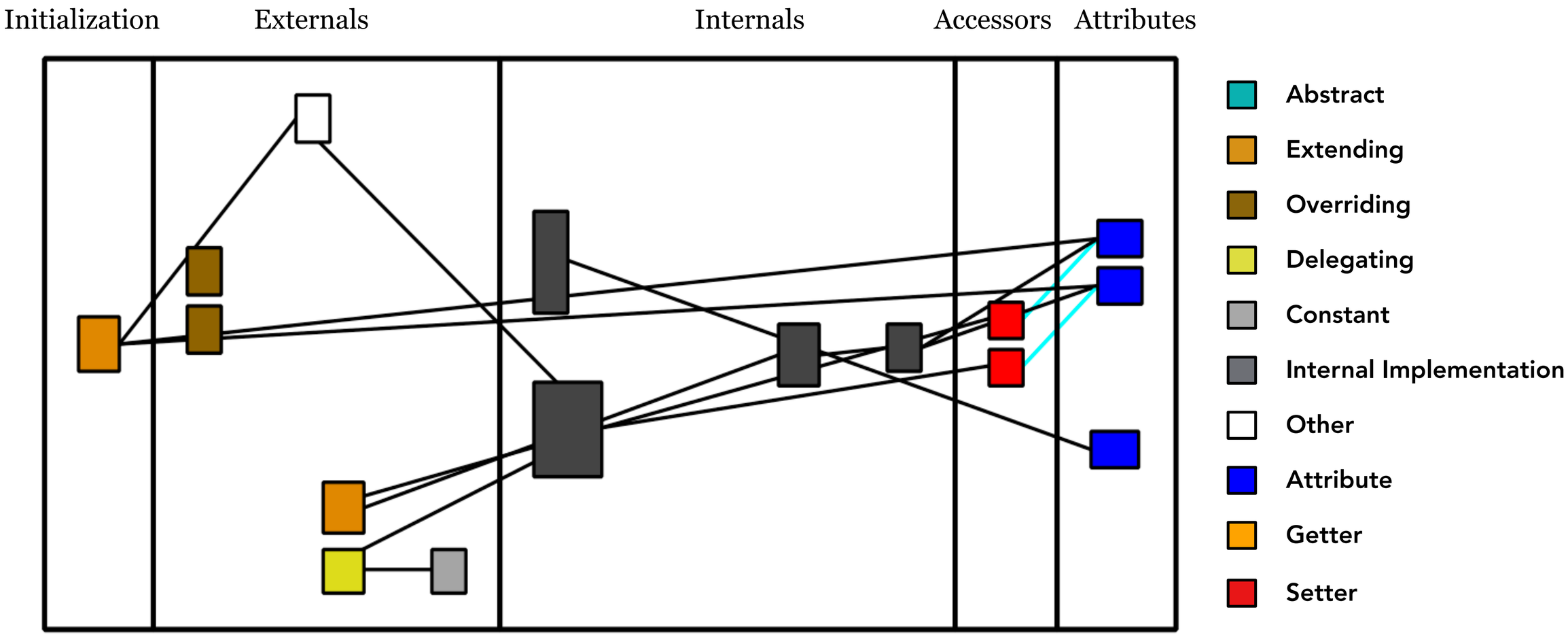
V1: Simple line connection



Black line: Connection between methods

Cyan line: Connection from accessors to attributes

V1: Node type = a Color



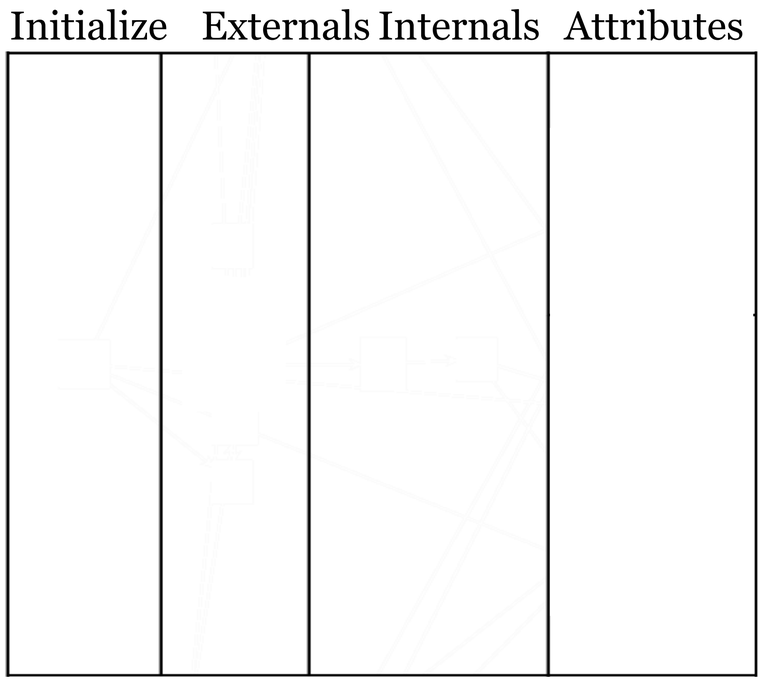
RMODPublicationsBblForReportDocBuilder class

V1: Limitations

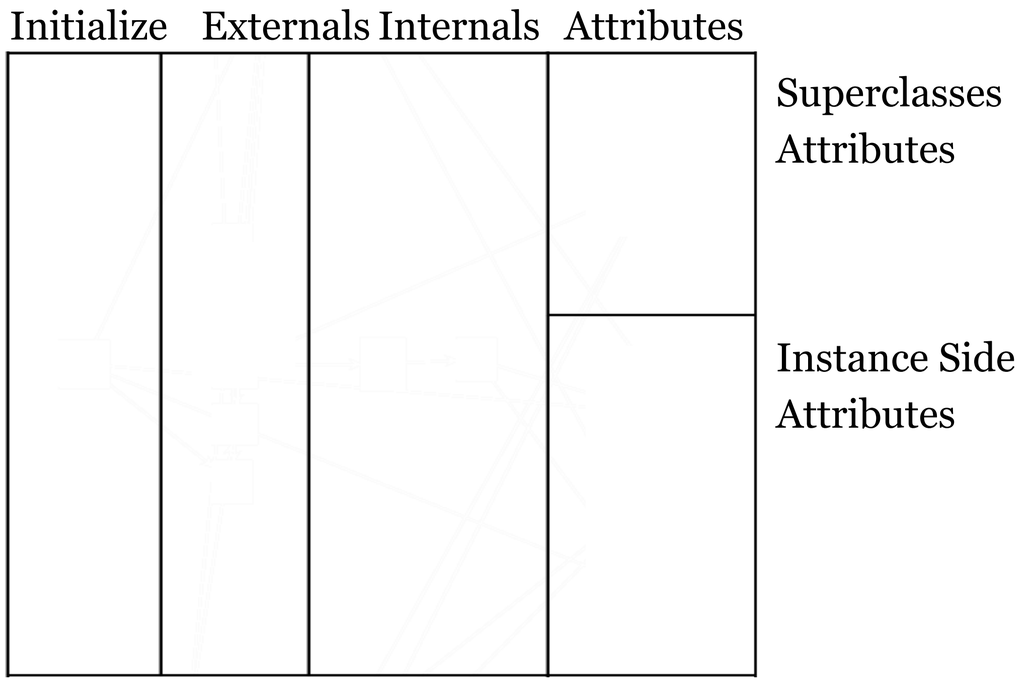
- Obsolete classification of methods
- Missing information about attribute accesses
- The interplay between instance side and class side is not well supported
- Does not heed dead code
- Unclear direction of links
- Does not show the occurrences of method names
- A method cyclomatic complexity is not revealed
- Does not show if a method is tested or not

Class Blueprint V2

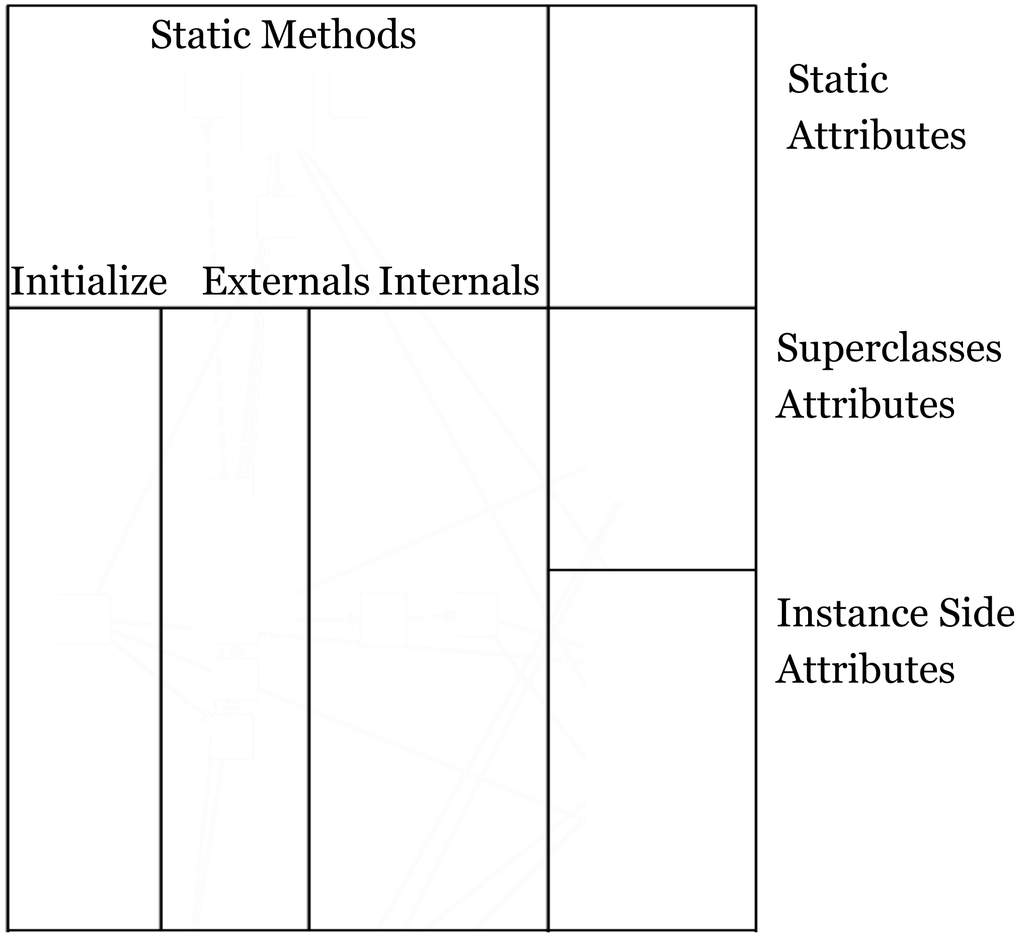
V2: Merging attributes & accessors layers



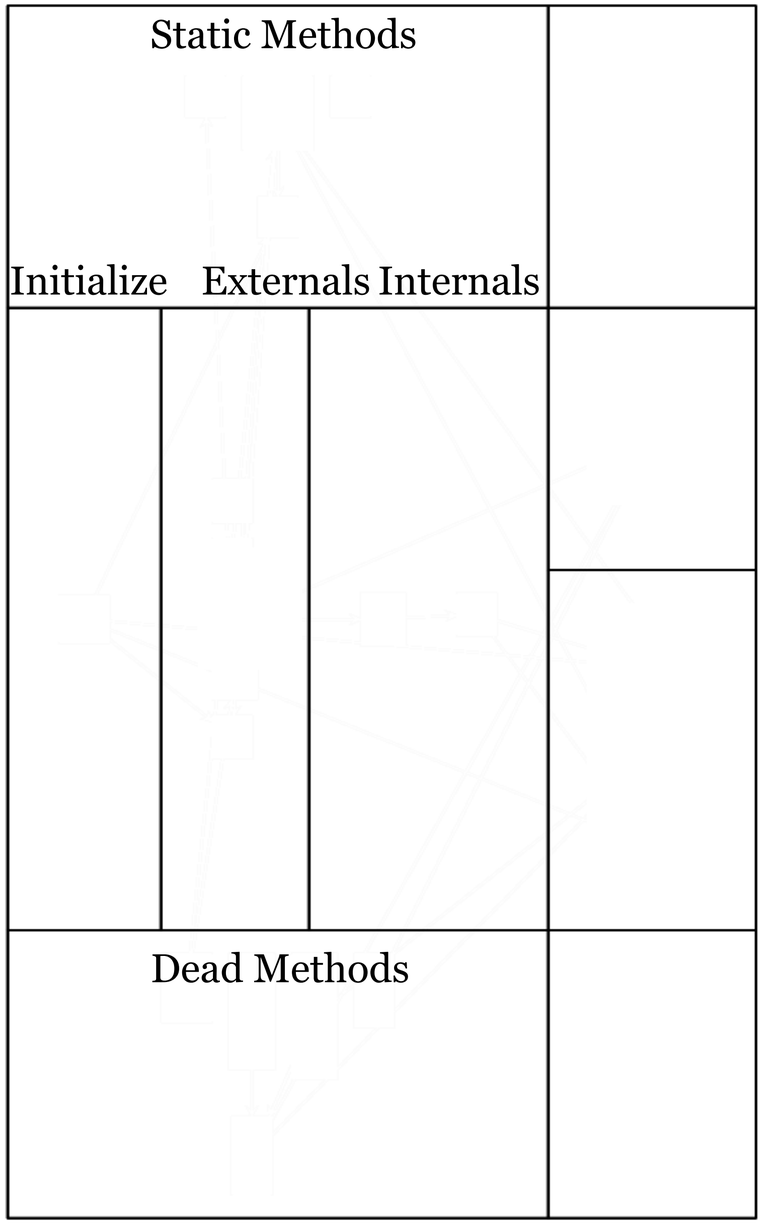
V2: Superclass attributes



V2: Static vs Instance



V2: Used vs Unused code

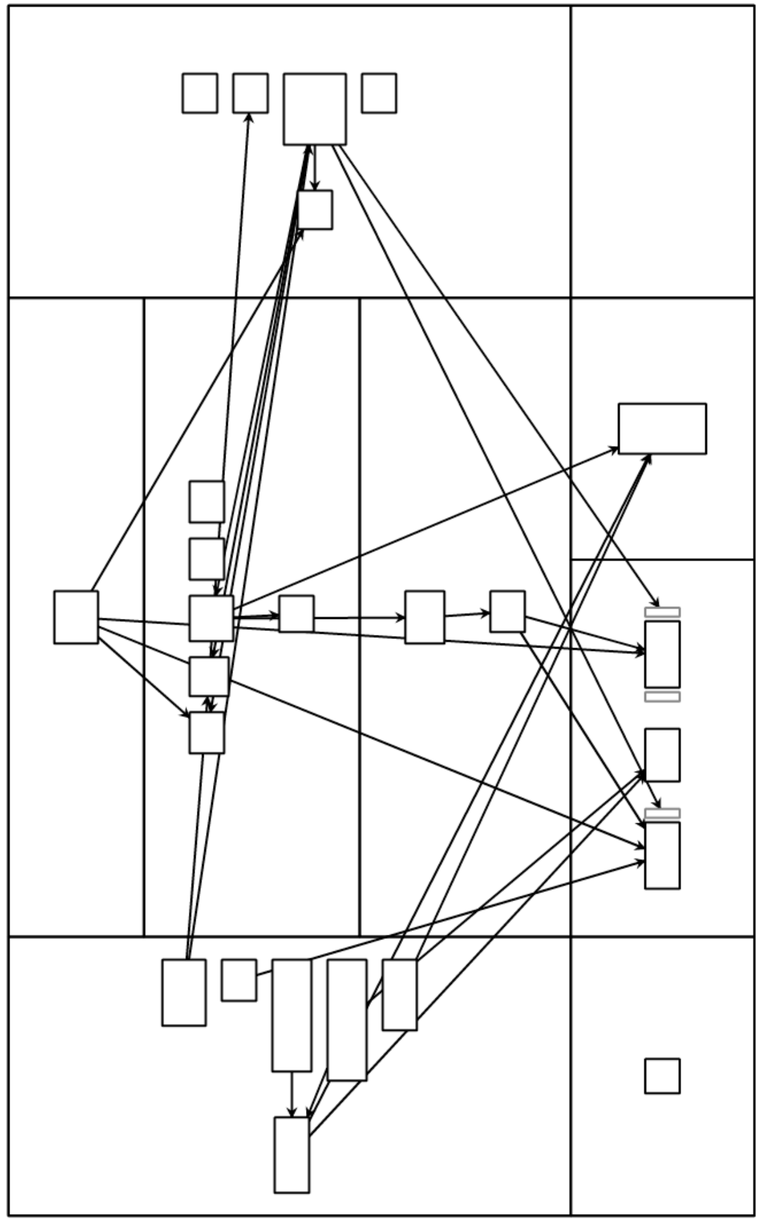


V2: Segment connection

Static

Instance

Dead

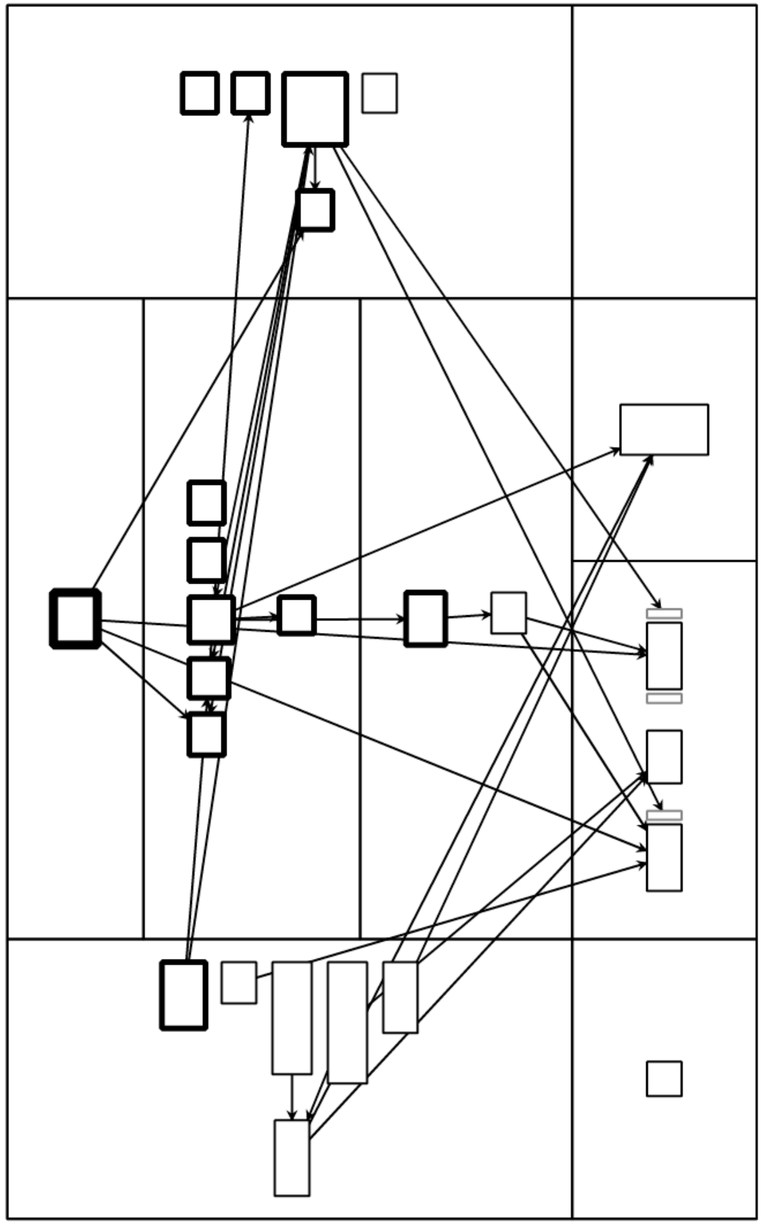


V2: Border width = Occurrences

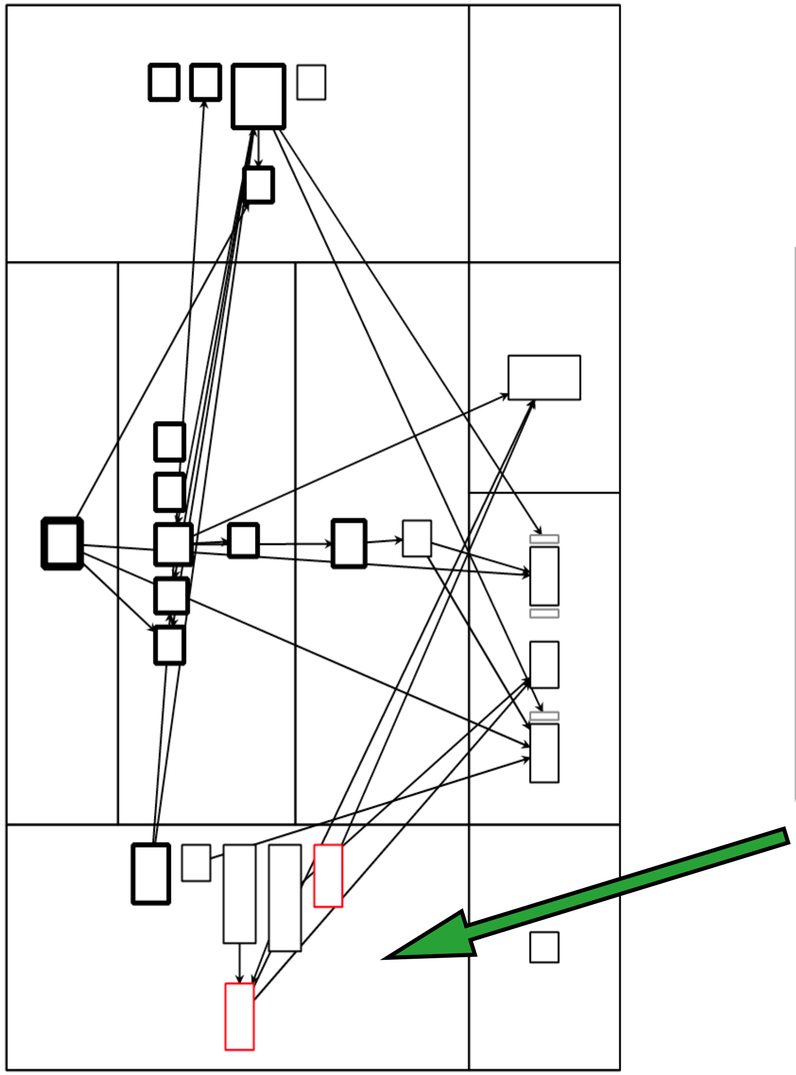
Monomorphic: One method by that name in the whole project.

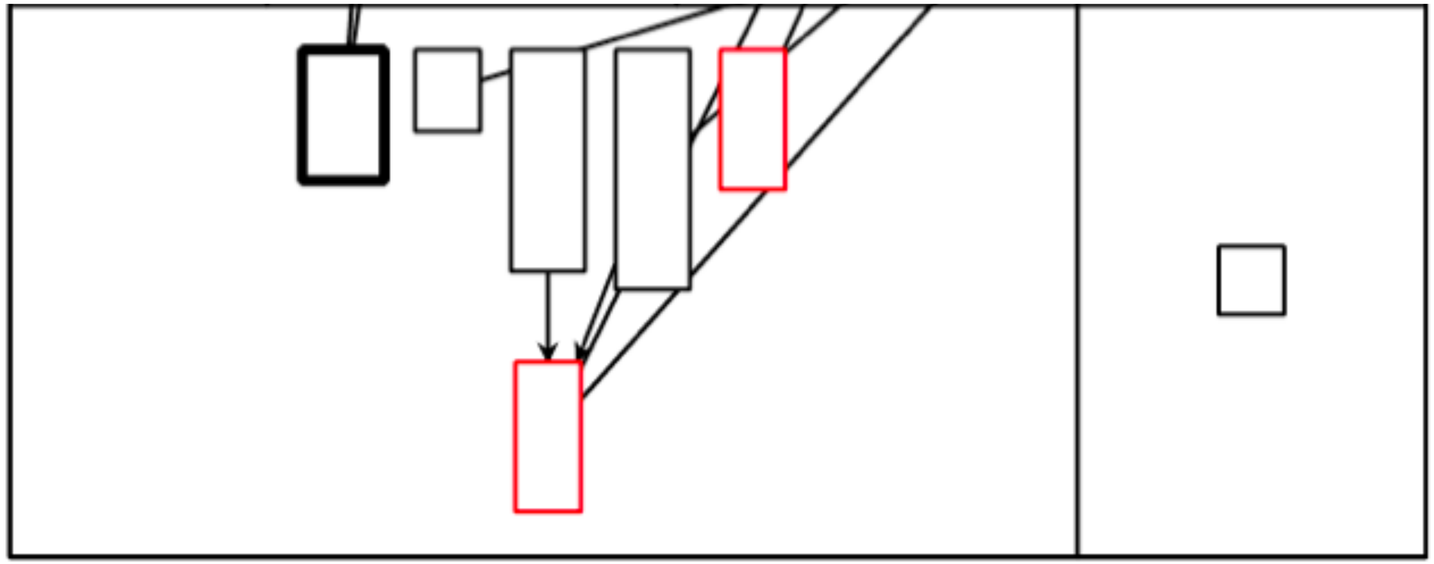
Polymorphic: Commonly named method

Megamorphic: Frequently named methods



V2: Border color = Cyclomatic complexity

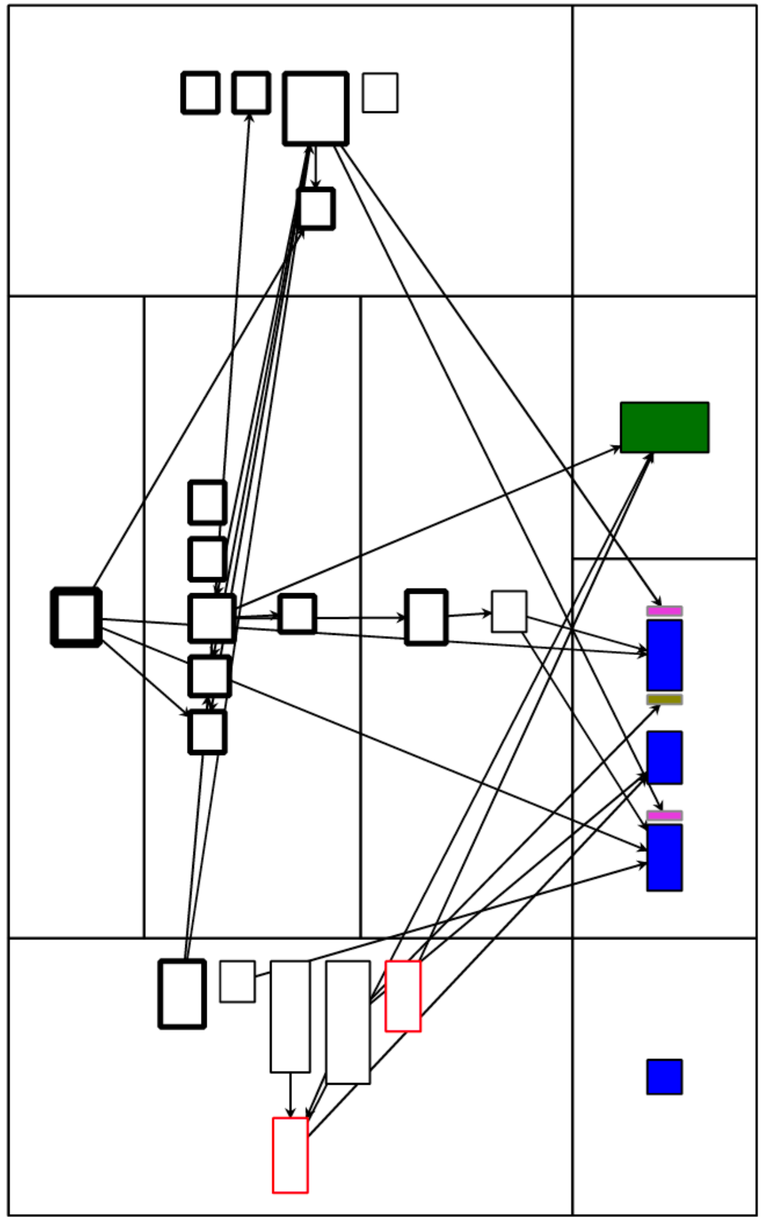




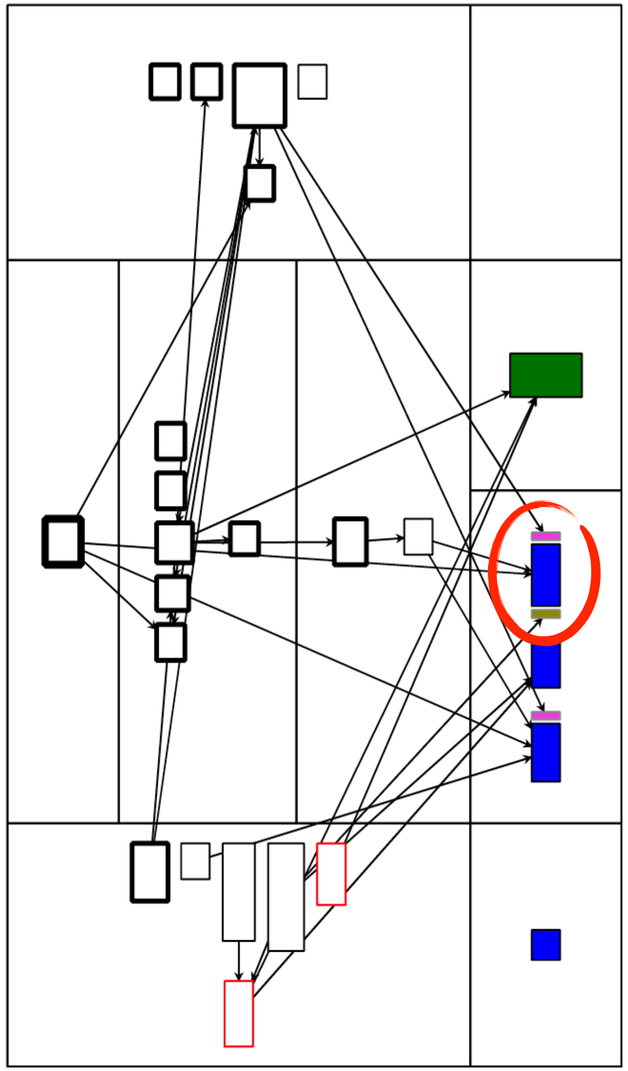
V2: Sub-hierarchy attribute access

Green: Accesses in the class and in the subsystem

Blue: Accesses in the class

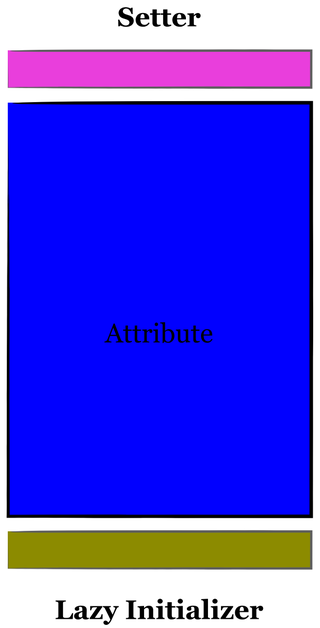


V2: Attribute protectors

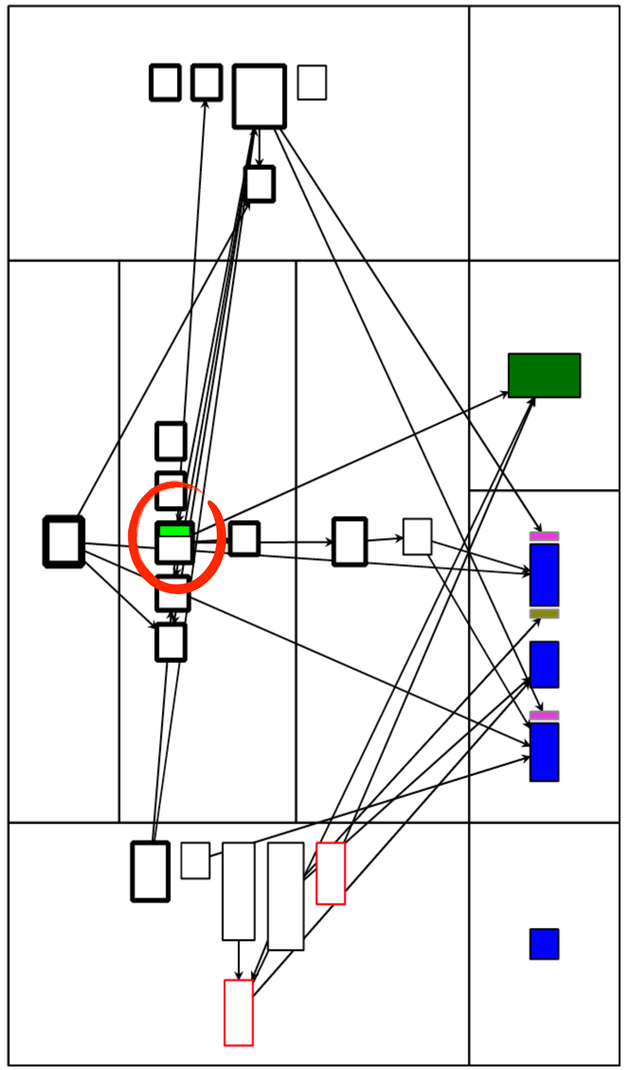


Accessor

Setter	Top ↑	
Getter	Bottom ↓	
Lazy Initializer	Bottom ↓	

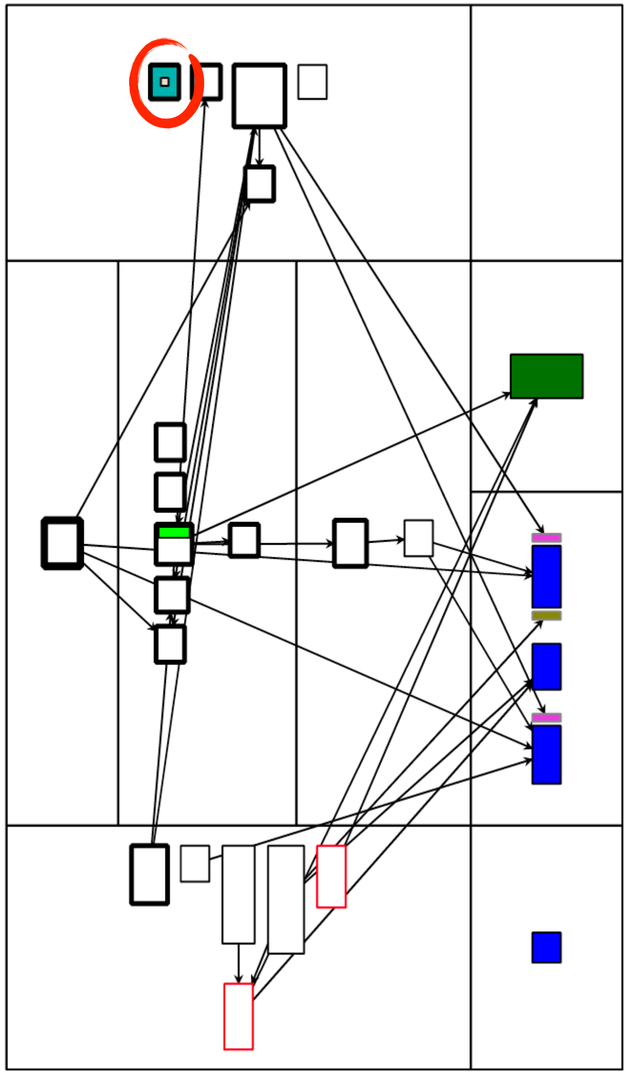


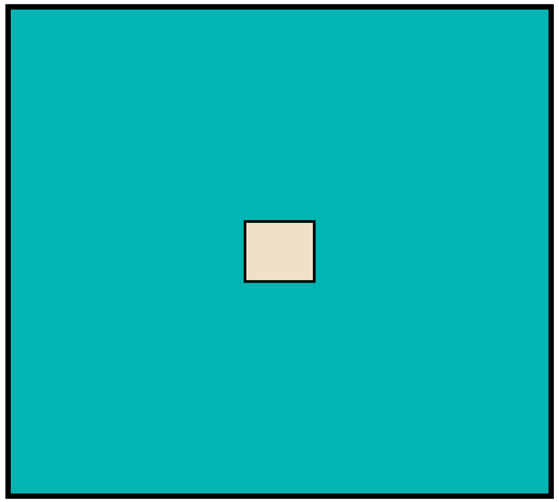
V2: Test annotation



Tested method

V2: Abstract reimplementation

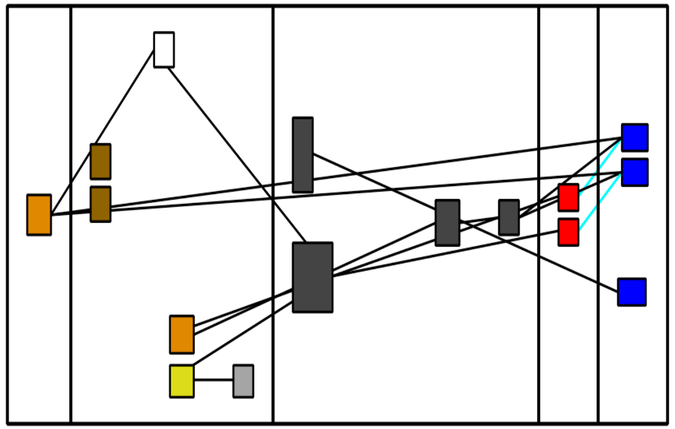




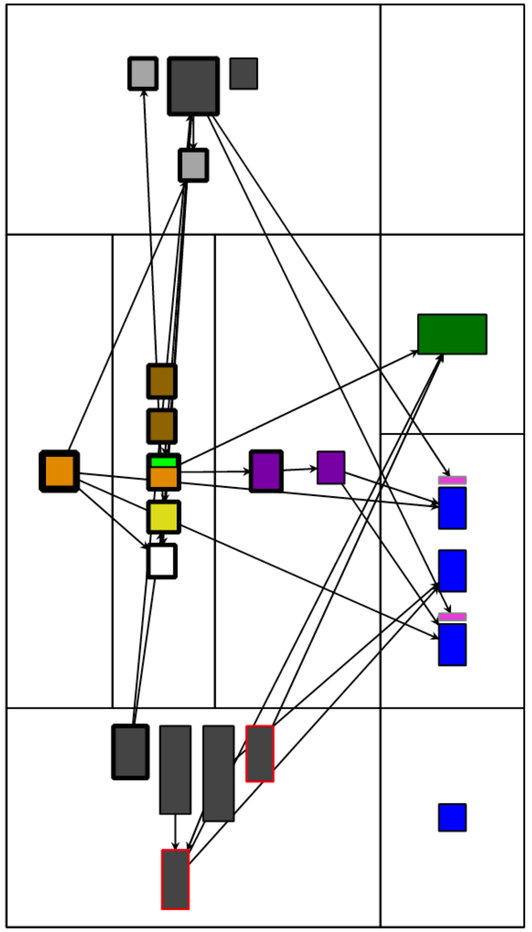
Abstract & Reimplemented

V2: Recap

- Merging attributes & accessors layers
- Superclass attributes
- Static vs Instance side
- Used vs Unused code
- Segment connection
- Method names occurrences
- Cyclomatic complexity
- Sub-hierarchy attribute access
- Detection of lazy initializers
- Detection of tested/untested methods
- Reimplemented abstract methods

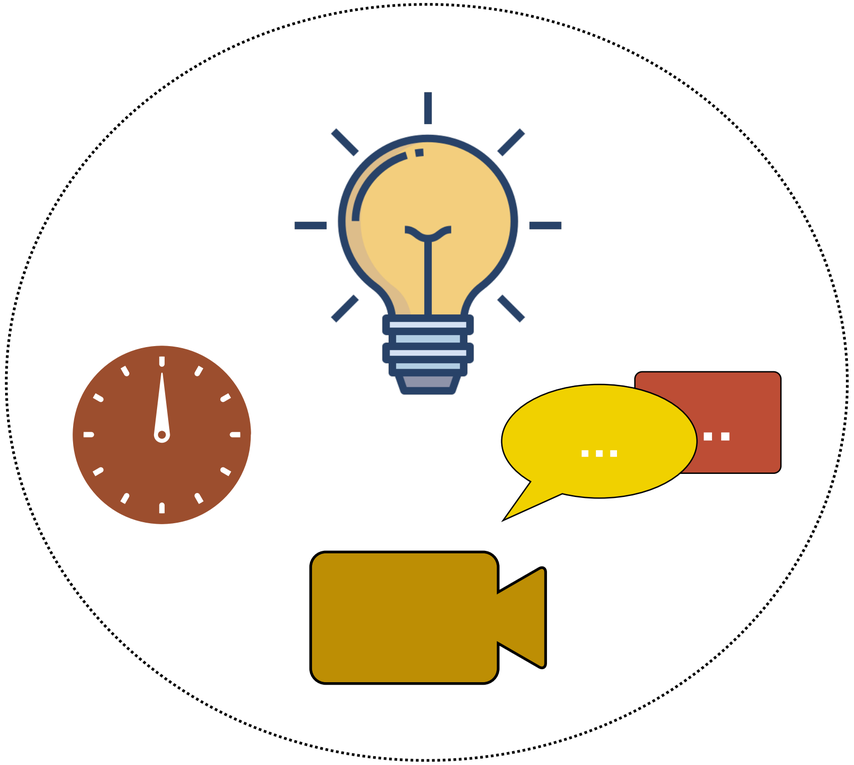


ClassBlueprint V1



ClassBlueprintV2

Evaluation.



Qualitative

Leon Zernitsky

Quantitative

Evaluation: Protocol

Invited people from the community (**26** participants)

Individual/ Group meetings

The meeting took from 10 to 25 minutes

Evaluation: What we asked for?

- ✓ Select a project they wish to analyze
- ✓ Use the visualization on the selected project
- ✓ Screen record the experiment
- ✓ Write a report summarizing their findings
- ✓ Fill the post-experiment survey

Evaluation: About the projects

Project	#Packages	#Classes	Median of methods	Doma
Avatar	2	18	6	Proxy
Sindarin	3	18	14	Debugging
MoTion	2	35	5	Pattern Machine
Clap	5	47	8	Parsing
Slang	2	73	29	Virtual machine
Polyphemus	3	79	9	Virtual machine
AST-Core	3	101	21	Domain-Specific-Language
Reflectivity	5	114	13	Domain-Specific-Language
Druid	1	170	12	Virtual Machine
Seeker	2	236	9	Debugging
MooseIDE	16	250	8	Analysis
Polymath	60	309	11	Computing
Refactoring	12	378	6	Refactorings
AIPharo	85	424	6	Artificial Intelligence
Roassal	39	445	12	Visualizations
Iceberg	11	488	10	Version Control
Fylgja	73	941	15	Migration
Microdown	29	268	11	Parsing

The projects chosen by the participants englobe several domains (19 projects)

Evaluation: About the participants

The participants have diverse profiles:

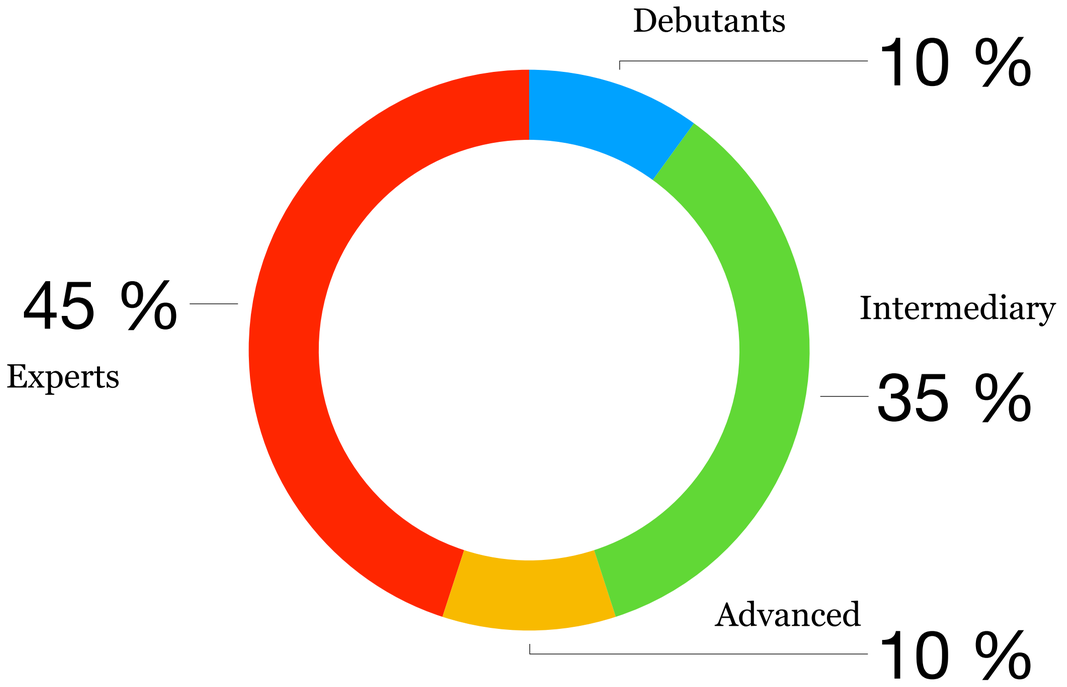
- Interns
- Developers
- PhD students
- Researchers

Participants level of knowledge about the project.

Evaluation: About the participants

The participants have diverse profiles:

- Interns
- Developers
- PhD students
- Researchers



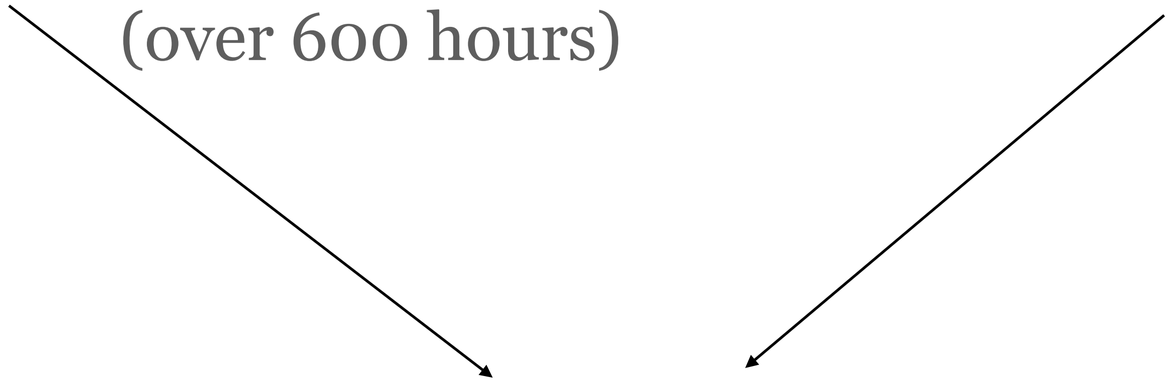
Participants level of knowledge about the project.

Evaluation: Qualitative

Screen records

(over 600 hours)

Findings reports



Analyse data about the human-visualization interaction

Qualitative evaluation: What did we find?

Qualitative evaluation: Flight over

- ✓ Empty Classes
- ✓ Big Classes
- ✓ Complex Classes
- ✓ Dying Classes
- ✓ Tested/Untested Classes

Qualitative evaluation: Plunge in

- ✓ Duplicated Code
- ✓ Complex Methods
- ✓ Dead Code
- ✓ Long Method Comments
- ✓ Tested/Untested Methods

Evaluation: Quantitative

1. The visualization helps in **understanding** the:

	Strongly agree	Agree	Undecided	Disagree	Strongly Disagree
Code/State of a class is reused	15%	46%	15%	23%	
Reused code from the superclass	3%	23%	53%	19%	
Class/instance side communication	19%	53%	19%	7%	
Design of the class	7%	38%	30%	23%	
Mono/poly/megamorphic methods	26%	46%	26%		

2. Does the visualization help in **detecting**:

Dead code	26%	46%	11%	3%	11%
Complex methods	46%	38%	11%	3%	
Tested/Untested methods	34%	34%	19%	7%	3%

3. The visualization is:

Scalable	3%	42%	30%	15%	7%
Easy to use	15%	76%		7%	

Quantitative

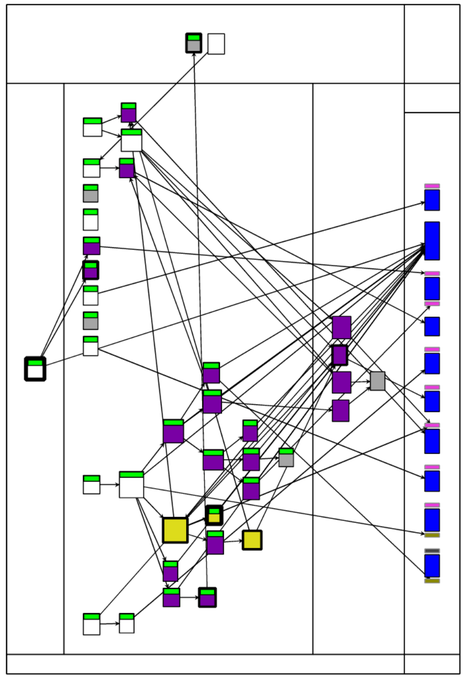
Anecdotal evidence

“

In the MicHTMLDoc class we could exclusively see the tested and untested methods.

”

- From the Microdown project



Anecdotal evidence

> “Dead methods correspond mostly to unused code that I forgot to remove.”
>
> - From the Seeker project

Anecdotal evidence

> “I found obsolete prototype code by taking a look at these long methods”
>
> - From the Seeker project

Anecdotal evidence

> “
>
> The visualization also helped me quickly identify dead code and eliminate it. As this is a new project (early stage of development) I didn’t remove all dead methods or classes, but in other kinds of projects I would do it.
>
> ”
>
> - From the Druid project

Anecdotal evidence

> “I couldn't used it in the large classes, those are the most interesting to analyze.”
>
> - From the Iceberg project

Conclusion

- An enhancement of the Class Blueprint visualization based on new requirements
- Qualitative & quantitative evaluations on 26 participants and 19 projects
- Participants reported some interesting findings about anomalies in their software

A New Generation of CLASS BLUEPRINT

Nour Jihene Agouf[1], Stéphane Ducasse[2], Anne Etien[2], Michele Lanza[3]
1: Arolla and Univ. Lille, CNRS, Inria, Centrale Lille
2: Univ. Lille, Inria, CNRS, Centrale Lille, UMR 9189 CRIStAL — 3: Software Institute, USI Lugano, Switzerland

***Abstract*—In object-oriented programming, classes are the primary abstraction mechanism used by and exposed to developers. Understanding classes is key for the development and evolution of object-oriented applications. The fundamental problem faced by developers is that while classes are intrinsically structured entities, in IDEs they are represented as a blob of text. The idea behind the original CLASS BLUEPRINT visualization was to represent the internal structure of classes in terms of fields, their accesses, and the method call flow. Additional information was depicted using colors. The thus created visualization proved to be an effective means to support program comprehension. However, a number of omissions rendered it only partially useful.**

We propose CLASS BLUEPRINT V2 (in short BLUEPRINTV2), which in addition to the information depicted by CLASS BLUEPRINT also supports dead code identification, methods under tests, and calling relationships between class and instance level methods. In addition, BLUEPRINTV2 enhances the understanding of fields by showing how fields of super/subclasses are accessed. We present the enhanced visualization and report on a first validation with 26 developers and 18 projects.

***Index Terms*—Visualization, program comprehension, code quality.**

I. INTRODUCTION

Understanding application logic is a time-consuming task during maintenance and software evolution. Researchers report that over half of the maintenance time is spent on reading and understanding source code [1], [2],where developers pore over source code, looking for clues that help them to construct a coherent mental model of a system [3], so as to make appropriate changes while ensuring its quality [4]–[6].

This is a difficult undertaking for any programming language, however maintaining and monitoring the quality of an object-oriented system is more complex than for procedural programs [7], [8], due to several reasons, such as inheritance and polymorphism [9]–[12]: Inheritance and polymorphism increase the flexibility of programs by allowing dynamic binding of messages. Inheritance allows the extension of an existing behavior through an inheritance hierarchy; polymorphism the performance of a task in multiple forms, with different objects responding to messages with the same name but different implementations. What can and should be considered a strength of object-oriented languages de facto hinders program comprehension [11], [12]. Dynamic binding of messages leads to more complex traceability of the call flow of a program since the type of the object receiving the message is determined at runtime. To follow the call flow of a program, developers proposed several approaches, including integrated development environments (IDE) and debuggers [13].

However, the usage of such tools is often too fine-grained, and thus time-consuming. Software visualization can provide a graphical view of a piece of software rather than a sequence of source code text. To this end, researchers proposed several visualization approaches, both in 2D [14] and 3D [15], [16]. Lanza and Ducasse [17] proposed the CLASS BLUEPRINT visualization to help developers get a "taste" of the class. CLASS BLUEPRINT presents the internal structure of classes in terms of fields, their accesses, and the method call-flow. Additional information was represented using colors. The authors classified classes based on their internal structure [18]. Regardless of its effectiveness, it did not display some up-to-date properties of object-oriented programming.

We present BLUEPRINTV2, an extension of the CLASS BLUEPRINT visualization based on updated requirements for program understanding. This approach discerns it from other (visualization) techniques that focus on views of sequential text, by offering a technical portrayal of the class per se. BLUEPRINTV2 supports the identification of dead code (single and branch), methods under tests, and call flow between instance and class (static) methods. It also enhances fields understanding by showing how fields of super/sub-classes are accessed, as well as lazy initialization in a compact form. In addition to hook understanding from a superclass point of view. After detailing the principles behind BLUEPRINTV2, we discuss its in-vivo validation with developers.

II. LIMITS OF CLASS BLUEPRINT

The CLASS BLUEPRINT visualization was created to help developers understand class structures [17], [18]. It decomposes classes into layers representing the invocation sequence going through external, internal, and accessor methods. This decomposition into layers organizes the method call-graph, and allows one to see which attributes are accessed by which methods, directly or through their accessors (see Figure 1).

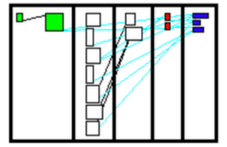


Fig. 1. A class blueprint from [18] with 5 layers: initialization, interface, internal implementation, accessor and attribute.