# An Overview of Sema Detect

## Executive Summary

Technical debt is a major problem affecting software products. The time and effort involved in maintaining and repairing large software code bases can become so large that other projects can suffer. Does the entire product need refactoring? Is focusing on specific issues enough, and if so, which ones? How do you make the impact of technical debt clear to the rest of the team? How do you justify the time and cost to fix it? Sema Detect manages the effort to diagnose problems within your system.

## What is Sema Detect

Sema Detect proactively locates the origins of technical debt by identifying code quality issues at the architectural level. We use antipattern detection and visualization tools to create a clear picture of exactly where problems originate at the architectural level, and how they are connected to the code-level flaws.

Successful software maintenance is not achieved by solving a single problem. Most software maintenance activities are multifaceted and may involve many different parts of the system. These activities can be grouped into four categories as follows[1]:

- **Perfective**: changes that do not alter functionality but improve performance or structure of the code.
- **Adaptive**: changes that adjust for modifications in the operating environment, such as adapting to interoperating systems, or switching languages.
- **Corrective**: changes to address bugs or other issues that correct functionality.
- **Preventive**: changes that do not affect functionality or performance but improve future maintenance or usage.

Most software maintenance projects involve more than one of these categories.

Before software maintenance activities can begin, software quality issues must first be identified. This is not an easy proposition since most quality issues are not single points of failure but complex issues that can span across the entire codebase. Such issues are often referred to as code smells, or antipatterns.

Antipatterns are ineffective responses to recurring problems that risk becoming counterproductive to the system. Sema scientists have prioritized the antipatterns that were most important based on interviews and analysis with active programmers. **Table 1** features this list of antipatterns.

[1] See e.g. NIST at https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nbsspecialpublication500-106.pdf, p 14

**Table 1: Antipatterns**

| Antipattern | Definition |
|---|---|
| BlobClass | One large class monopolizes the behavior of a system and the other classes encapsulate data. |
| BlobOperation | A class that can perform compression and de-duplication, making it quicker and cheaper to store a great amount of data. |
| CyclicDependencies (mutually recursive) | The relation between two or more modules which either directly or indirectly depend on each other to function properly. |
| DataClass | A class containing only data and performs no processing on these data. It is often composed of highly cohesive fields and accessors. |
| DataClumps | Occurs when a group of variables are passed around together in various parts of the program. |
| Feature envy | Occurs when a method is focused on features of other classes than its own. |
| Functional decomposition | Occurs when a class is designed to perform a single function, usually produced by non-experienced object-oriented developers. |
| GodClass | A class that controls many other classes and have many dependencies and many responsibilities. |
| IntensiveCoupling | A method that calls many other methods from a few classes. |
| Lava flow | A section of code leftover from older development that ends up in production. |
| Lazy class | A class that is downsized during a refactoring effort, or when a class is added in anticipation of a future need that never occurs. |
| Long parameter list | Methods with numerous parameters that become difficult to maintain, especially if most of them share the same data type. |
| RefusedParentBequest | A class that overrides a method of a base class in such a way that the contract of the base class is not honored by the derived class. |
| SchizophrenicClass | A class that is complicated by conflicted statements, such as SELF/THIS, that refer to more than one object. |
| ShotgunSurgery | When a developer adds features to a codebase which span multiple implementors or implementations in a single change. |
| Spaghetti code | Code with a complex and tangled control structure. |
| TraditionBreaker | A class that does not use the protected members of its parent. |
| UnstableDependencies | A situation where an increasing number of packages are dependent upon each other in a non-reinforcing manner. |

To understand the damage these antipatterns capable of, let us look at one of them, **Lava Flow**, as an example. The Lava Flow antipattern, if left alone, can produce some of the following consequences:

- It can proliferate as the code is reused in other areas.
- It can grow, geometrically, as successive developers, who do not analyze the original code, continue to produce new, secondary code that exacerbates the problem.
- It becomes impossible to document the code or understand its architecture enough to make improvements.[2]

## The Sema Detect Approach

Sema Detect uses a group of software quality metrics to analyze the architecture and semantic nature of the code base. Antipatterns can be detected by looking for code that violates any of these metrics, defined in **Table 2**[3].

**Table 2: Software Quality Metrics**

| Metric | Definition |
|---|---|
| Attribute hiding factor (AH) | AH measures the invisibilities of attributes in classes. The invisibility of an attribute is the percentage of the total classes from which the attribute is not visible. |
| Attribute inheritance factor (AIF) | AIF is the fraction of class attributes that are inherited. |
| Coupling between object classes (CBO) | CBO measures the number of classes coupled to a given class. |
| Depth of inheritance tree (DIT) | DIT is defined as the maximum length from the class node to the root/parent of the class hierarchy tree and is measured by the number of ancestor classes. |
| Lack of cohesion of methods (LCOM) | LCOM in methods is defined as the number of pairs of methods in a class that do not have at least one field in common, minus the number of pairs of methods in the class that does share at least one field. When this value is negative, the metric value is set to **0**. |
| Method hiding factor (MH) | MH measures the invisibilities of methods in classes. The invisibility of a method is the percentage of the total classes from which the method is not visible. |
| Number of children (NOC) | NOC measures the number of immediate descendants of the class. |
| Number of lines of code (NLC) | NLC counts the code lines but excludes empty lines and comments. |

[2] http://www.drdobbs.com/architecture-and-design/antipatterns/184410581

[3] Multi-objective code-smells detection using good and bad design examples, Mansoor et. al. Software Qual J (2017) 25:529–552, p.534.

| Metric | Definition |
|---|---|
| Polymorphism factor (PF) | PF measures the degree of method overriding in the class inheritance tree. It equals the number of actual method overrides divided by the maximum number of possible method overrides. |
| Response for a class (RFC) | RFC is the number of different methods that can be executed when an object of that class receives a message. |
| Weighted methods per class (WMC) | WMC represents the sum of the complexities of its methods. |

Sema Detect treats antipattern detection as a bi-level optimization problem. What that means is that Sema Detect looks at antipatterns as both poor design choices and architectural defects, and that each of these characteristics can influence the other. Traditional code-smell detection searches the code base for only one of these characteristics. Our approach seeks out both of these characteristics by performing "smell tests" on a wide variety of antipatterns such as Data Classes, Data Clumps, and others listed in **Table 1**.

How Sema Detect performs code-smell detection is by using a set of rules. These rules are an application of the Software Quality Metrics (**Table 2**), including their upper and lower thresholds. Using these rules, code-smell detection has two objective functions to be optimized: maximizing the coverage of code-smell examples and minimizing the detection of good design practice examples[4].

Detecting these antipatterns is only the start of the process. Once detected, Sema Detect presents the code base visually, so developers and upper-level management can gain a better understanding of how architectural issues and poor design choices can create antipatterns.

[4] Multi-objective code-smells detection using good and bad design examples, Mansoor et. al. Software Qual J (2017) 25:529–552, p.534.

## Sema Detect Results

Sema Detect combines visualization and detailed data to help development leadership and engineers prioritize areas for refactoring that can have the greatest benefit for the existing codebase and new code before shipping new releases. This visualization is shown in *Figure 1*:

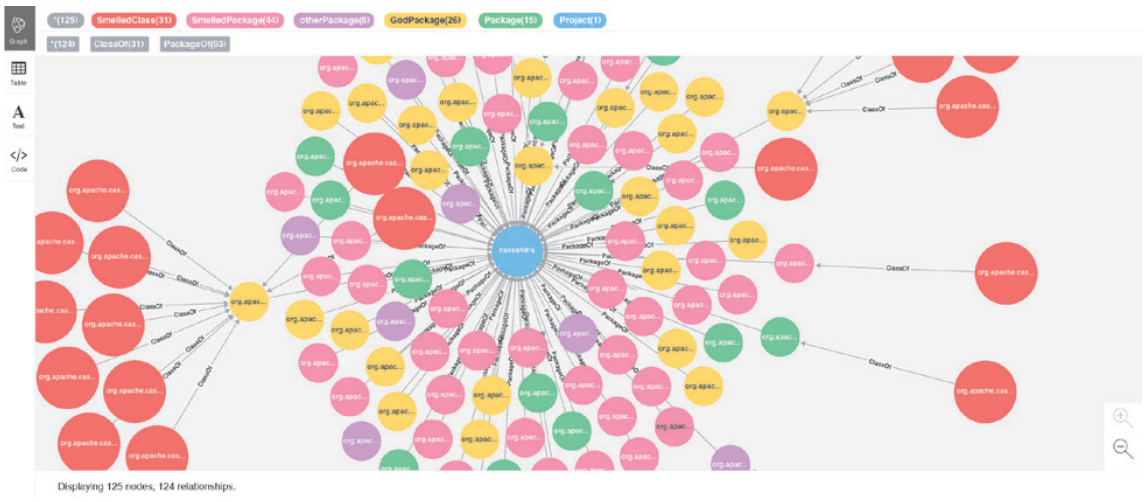


*Figure 1: Sema Detect Visualization*

Sema Detect allows developers to explore which classes need to be fixed, as well as all classes related to a specific quality issue. Developers can then identify problems that are rooted in architectural issues and see how these architectural issues are connected to each other, as shown in *Figure 2*.

This visualization of the entire system architecture helps not only developers but non-technical colleagues and high-level decision makers understand how antipatterns affect the entire system. The connections to problem areas can be identified and dependencies to these problem areas can be easily traced.

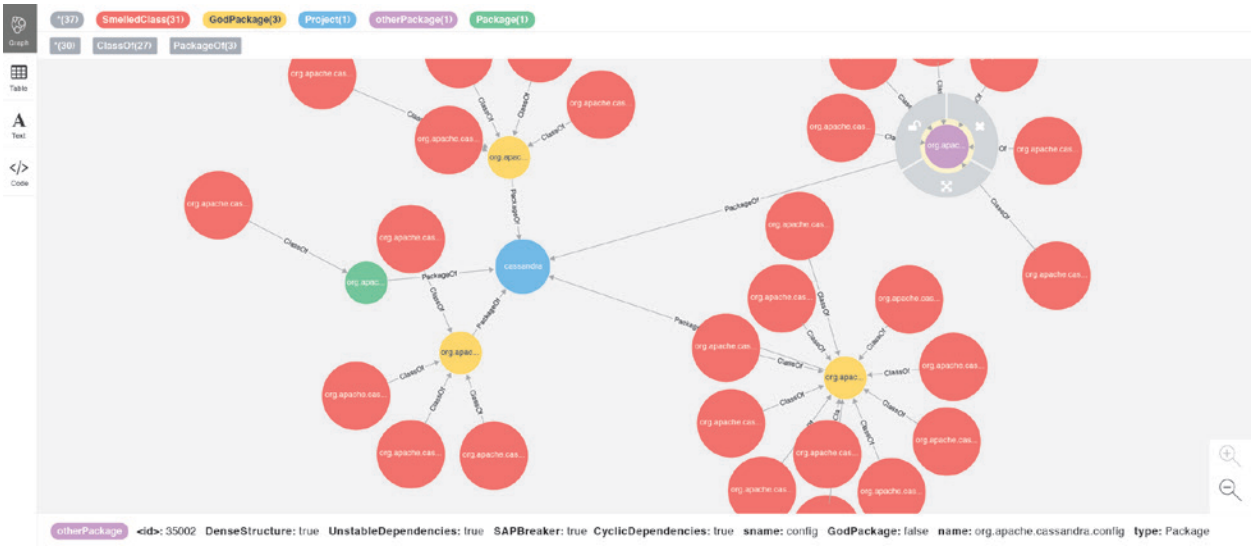


*Figure 2: Visual Exploration of Codebase to Reveal Details*

Another visualization that Sema Detect offers is shown in *Figure 3*. This hierarchical display offers a view of antipatterns in packages, classes, and methods. This view also allows the user the opportunity to “drill down” into the connections of each antipattern, providing another way to assess the extent of the issues. Each antipattern can be filtered for easier viewing.

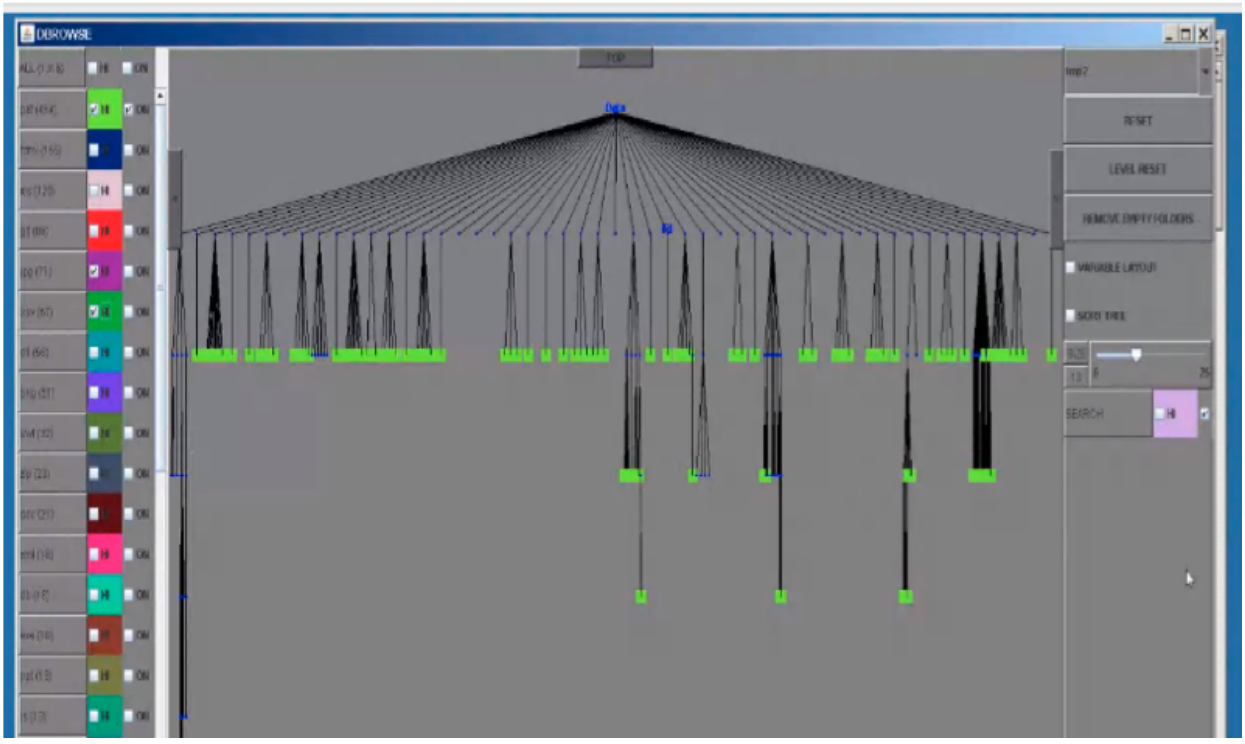


*Figure 3: Visual Hierarchy Display*

Sema Detect can also export the complete details on all packages, method and class levels to help prioritize and plan refactoring work. *Figure 4* is an example of the data

that Sema Detect exports for analysis.

| | pe | longTo | nseStructure | clicDependencies | pBreaker | stableDependencies | dPackage | bClass | dClass | taClass | hizophrenicClass | fusedParentBequest | ditionBreaker | storedHierarchy | ternalDuplication | klingDuplication | ternalDuplication | taClumps | ensiveCoupling | assageChains | otgunSurgery | atureEnvy | bOperation | RUES |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| bo.EROMigrationBO | Class | bo | F | F | F | F | F | T | F | F | F | F | F | F | F | F | F | F | T | F | F | F | T | 3 |
| bo.ExportsBO | Class | bo | F | F | F | F | F | T | F | F | T | F | F | F | F | F | F | F | T | F | T | F | T | 5 |
| bo.RegistrantListReportBO | Class | bo | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | T | F | F | T | T | 3 |
| bo.SessionOccurrencesBO | Class | bo | F | F | F | F | F | F | T | F | T | F | F | F | F | F | F | F | T | F | F | F | T | 4 |
| bo.SessionsBO | Class | bo | F | F | F | F | F | T | F | F | T | F | F | F | F | F | F | F | T | F | F | T | T | 5 |
| bo.TranscriptCertificateBO | Class | bo | F | F | F | F | F | T | F | F | F | F | F | F | F | F | F | F | T | F | F | F | T | 3 |
| bo.UserRegistrationsBO | Class | bo | F | F | F | F | F | T | F | F | F | F | F | F | F | F | F | F | T | F | F | T | T | 4 |
| bo.UsersBO | Class | bo | F | F | F | F | F | T | F | F | T | F | F | F | F | F | F | F | T | F | F | T | T | 5 |
| bo.UserTranscriptBO | Class | bo | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | T | F | T | T | F | 3 |
| controllers.ReportsController | Class | controllers | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | F | T | F | F | T | T | 3 |
| dao.SessionsDAO | Class | dao | F | F | F | F | F | T | F | F | F | F | F | F | F | F | F | F | T | F | F | F | T | 3 |
| dao.UserRegistrationsDAO | Class | dao | F | F | F | F | F | T | F | F | F | F | F | F | F | F | F | F | T | F | F | T | T | 4 |
| integration.blackboard.rest.Res | Class | integration.black | F | F | F | F | F | F | F | T | T | F | F | T | F | F | F | F | F | F | F | F | F | 3 |
| models.ProgramsSchedules_ | Class | models | F | F | F | F | F | F | F | T | T | F | F | T | F | F | F | F | F | F | F | F | F | 3 |
| models.Tenants_ | Class | models | F | F | F | F | F | F | F | T | T | F | F | T | F | F | F | F | F | F | F | F | F | 3 |
| workflow.bo.WorkFlowBO | Class | workflow.bo | F | F | F | F | F | F | F | F | T | F | F | F | F | F | F | F | T | F | F | F | T | 3 |

| NUMBER OF | Project | 1 | ANTIPATTERNS | | | |
|---|---|---|---|---|---|---|
| NUMBER OF | Package | 62 | 0 | 1 | 2 | 3+ |
| NUMBER OF | Class | 1070 | 316 | 534 | 205 | 16 |
| NUMBER OF | Method | 8132 | 7608 | 499 | 25 | 0 |

- Most classes had at least 1 antipattern detected (1070 – 316 = 754)
- Most methods had no issues (7608/8132)
- Most common method issues are: {FeatureEnvy, ShotgunSurgery}
- Most common class issues are: Intensive Coupling, DataClass, Schizo.)

*Figure 4: Sema Detect Export Data*

The following items are displayed in this data export:

- Frequency of antipatterns in packages, classes, methods
- Distribution of antipatterns within packages, classes, methods
- Summary analyses

## Conclusion

Sema Detect uses a better approach to discovering errors within your software. Sema Detect can locate the origins of technical debt by identifying the following code quality issues:

- code smells
- antipatterns
- semantic errors

Sema Detect is the smartest tool a developer can use to improve their own software.