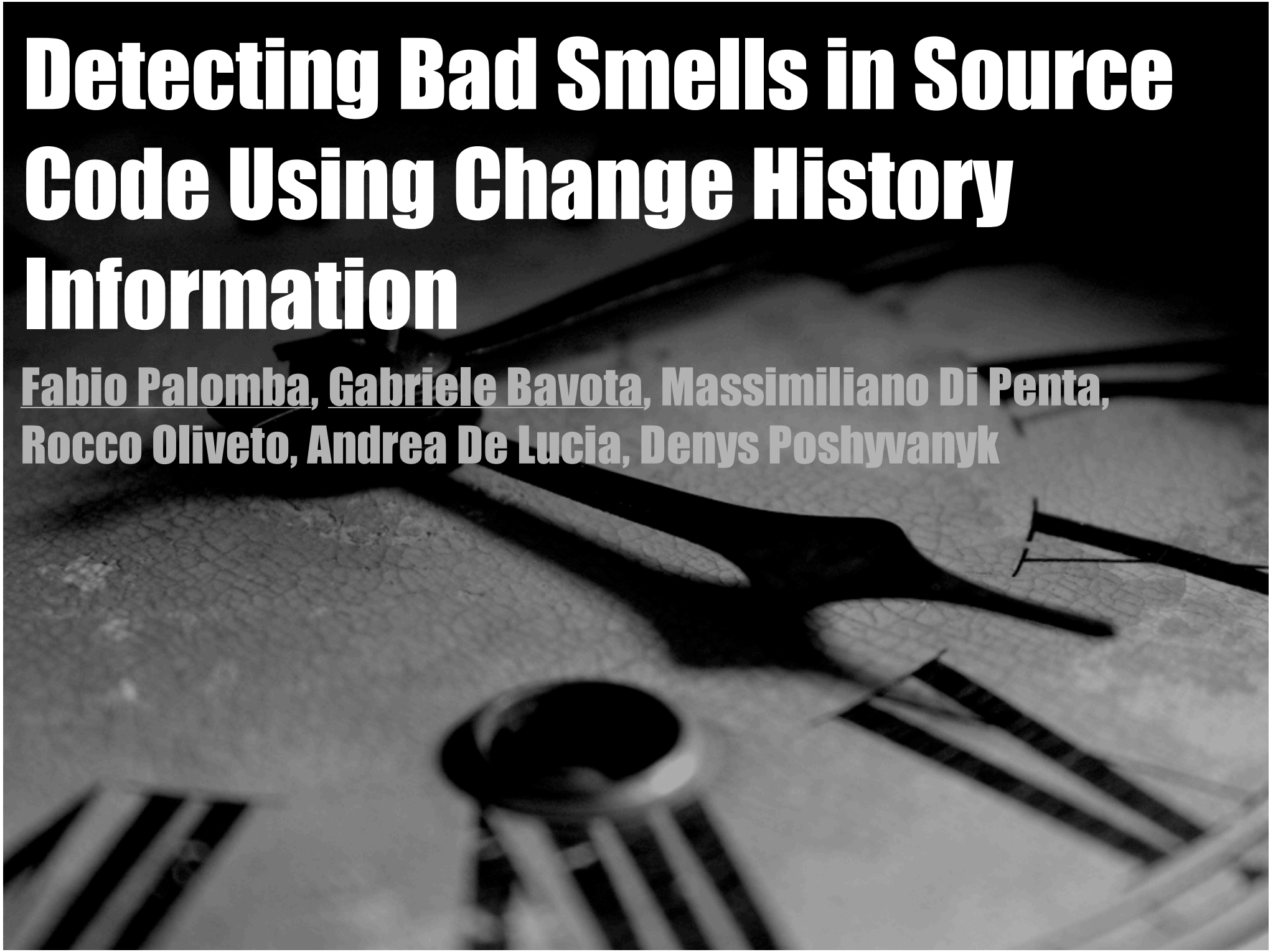
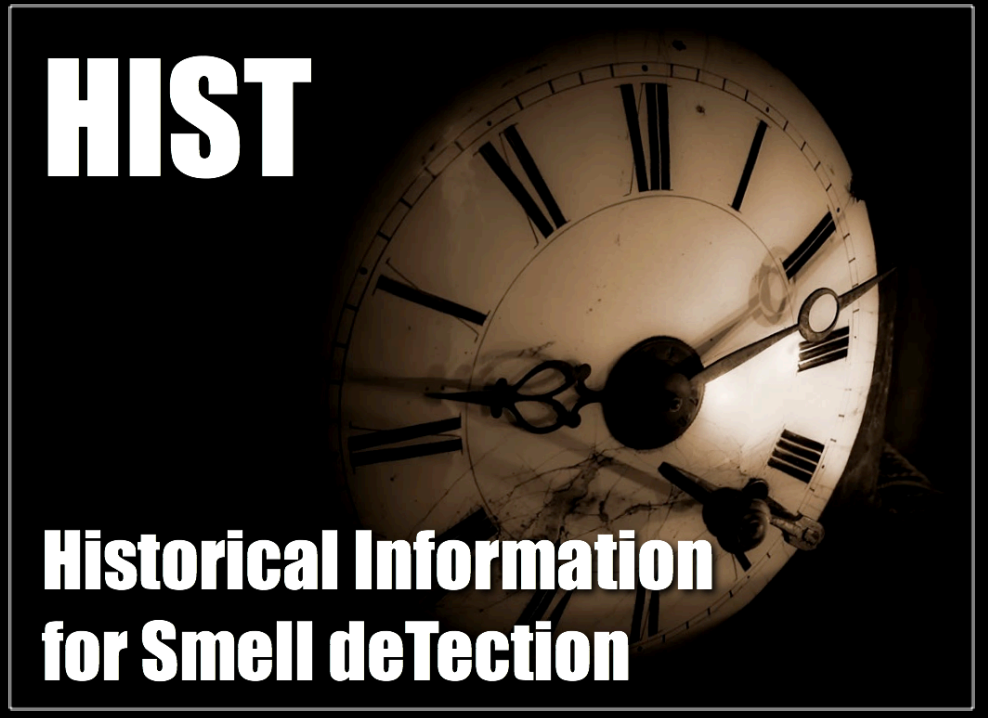
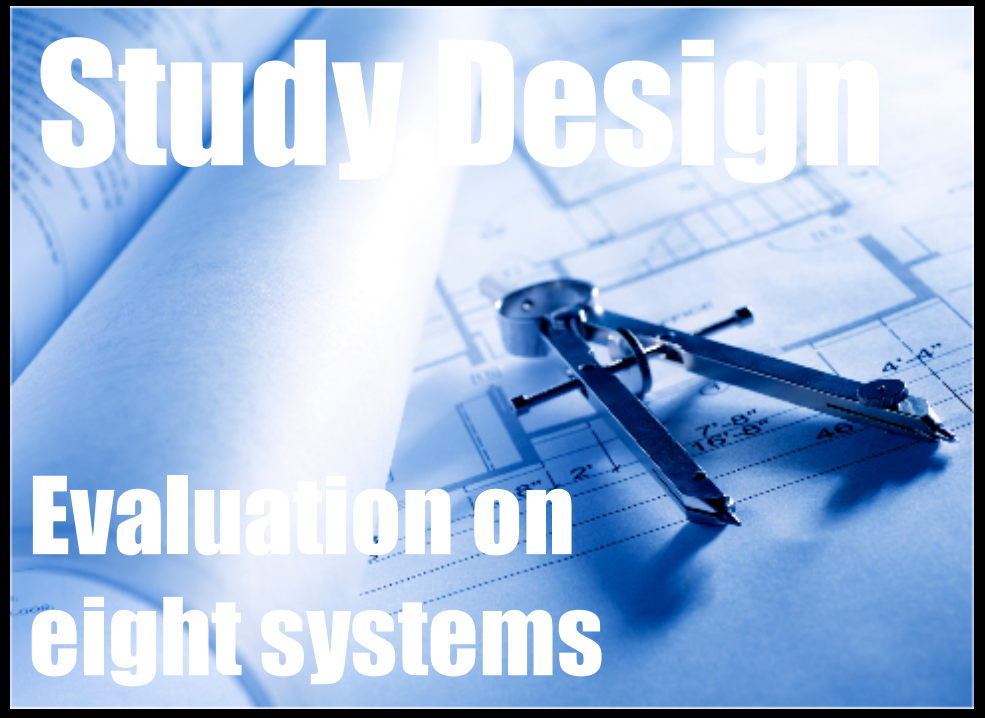
Detecting Bad Smells in Source Code Using Change History Information

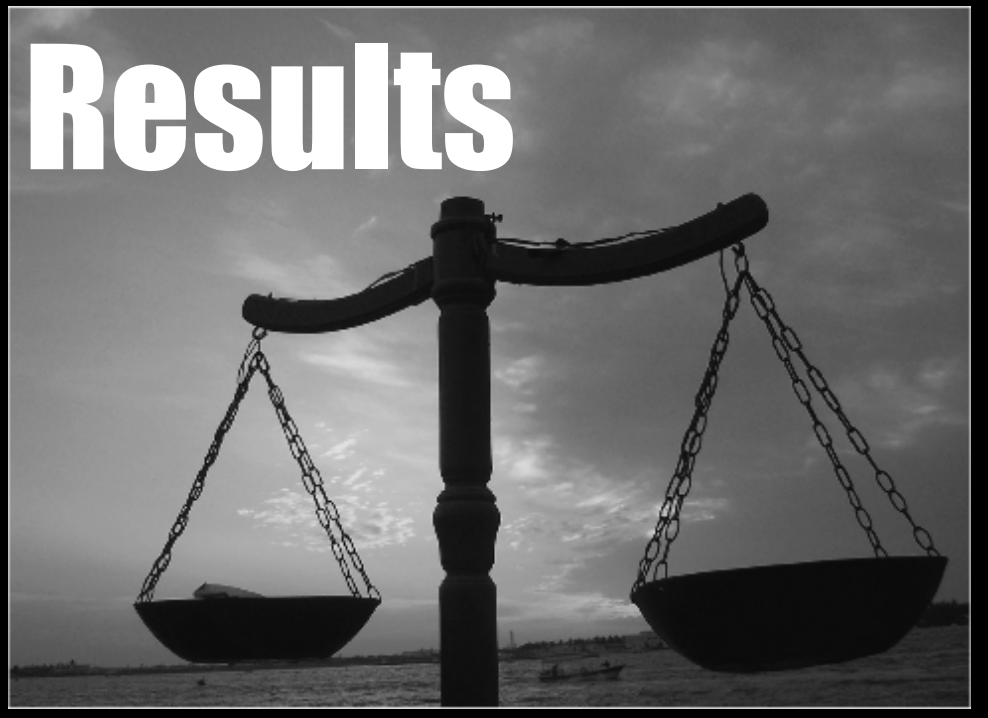
Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, Denys Poshyvanyk




Context
Code
Bad Smells

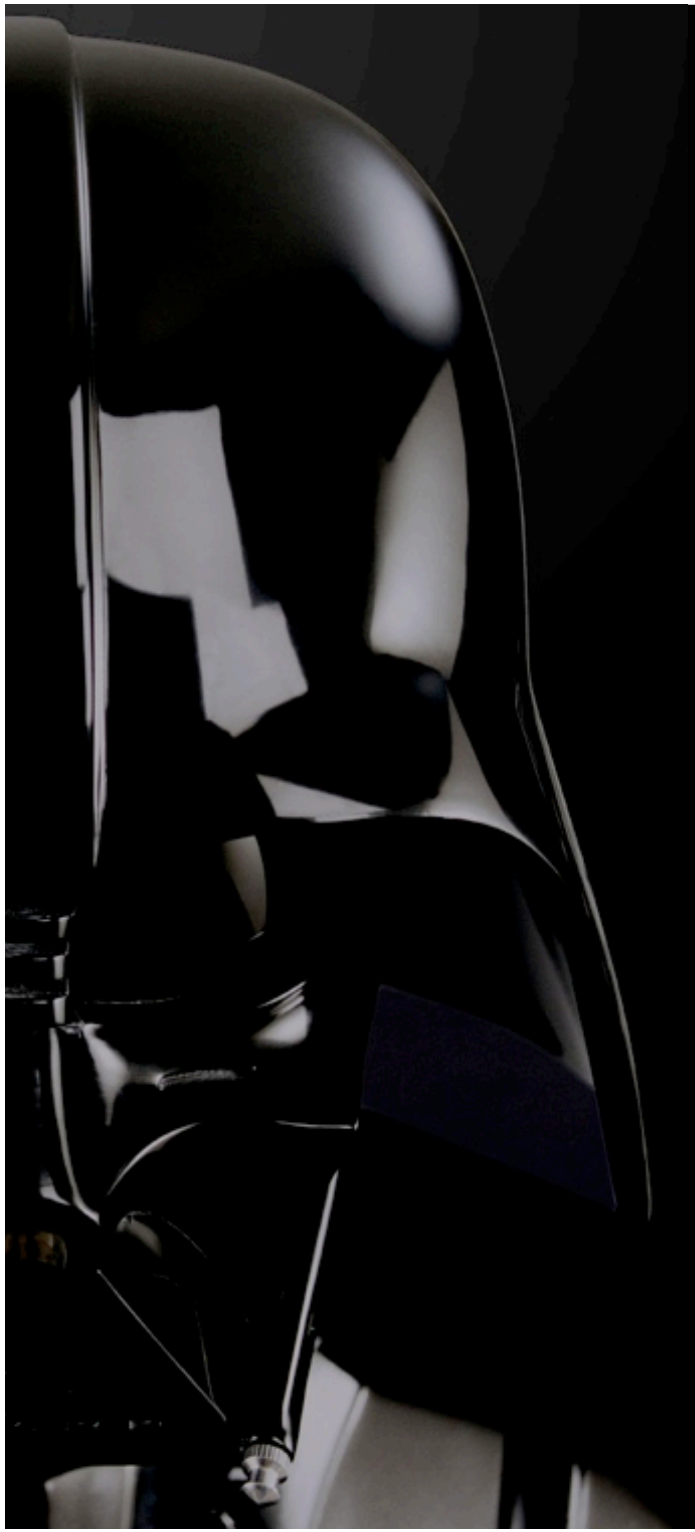

HIST
Historical Information
for Smell deTection


Study Design
Evaluation on
eight systems


Results


Context
Code
Bad Smells

Code Bad Smells



[Abbes et al. CSMR 2011]

2011 15th European Conference on Software Maintenance and Reengineering

An Empirical Study of the Impact of Two Antipatterns, Blob and Spaghetti Code, On Program Comprehension

Marwen Abbes[1,3], Foutse Khomh[2], Yann-Gaël Guéhéneuc[3], Giuliano Antoniol[3]
[1] Dépt. d'Informatique et de Recherche Opérationnelle, Université de Montréal, Montréal, Canada
[2] Dept. of Elec. and Comp. Engineering, Queen's University, Kingston, Ontario, Canada
[3] Ptidej Team, SOCCER Lab, DGIGL, École Polytechnique de Montréal, Canada
E-mails: marwen.abbes@umontreal.ca, foutse.khomh@queensu.ca
yann-gael.gueheneuc@polymtl.ca, antoniol@ieee.org

***Abstract*—Antipatterns are "poor" solutions to recurring design problems which are conjectured in the literature to make object-oriented systems harder to maintain. However, little quantitative evidence exists to support this conjecture. We performed an empirical study to investigate whether the occurrence of antipatterns does indeed affect the understandability of systems by developers during comprehension and maintenance tasks. We designed and conducted three experiments, with 24 subjects each, to collect data on the performance of developers on basic tasks related to program comprehension and assessed the impact of two antipatterns and of their combinations: Blob and Spaghetti Code. We measured the developers' performance with: (1) the NASA task load index for their effort; (2) the time that they spent performing their tasks; and, (3) their percentages of correct answers. Collected data show that the occurrence of one antipattern does not significantly decrease developers' performance while the combination of two antipatterns impedes significantly developers. We conclude that developers can cope with one antipattern but that combinations of antipatterns should be avoided possibly through detection and refactorings.**

***Keywords*-Antipatterns, Blob, Spaghetti Code, Program Comprehension, Program Maintenance, Empirical Software Engineering.**

I. Introduction

Context: In theory, antipatterns are "poor" solutions to recurring design problems; they stem from experienced software developers' expertise and describe common pitfalls in object-oriented programming, *e.g.*, Brown's 40 antipatterns [1]. Antipatterns are generally introduced in systems by developers not having sufficient knowledge and–or experience in solving a particular problem or having misapplied some design patterns. Coplien [2] described an antipattern as "something that looks like a good idea, but which back-fires badly when applied". In practice, antipatterns relate to and manifest themselves as code smells in the source code, symptoms of implementation and–or design problems [3].

An example of antipattern is the Blob, also called God Class. The Blob is a large and complex class that centralises the behavior of a portion of a system and only uses other classes as data holders, *i.e.*, data classes.

ral programming, and its association with data classes, which only provide fields and–or accessors to their fields. Another example of antipattern is the Spaghetti Code, which is characteristic of procedural thinking in object-oriented programming. Spaghetti Code classes have little structure, declare long methods with no parameters, and use global variables; their names and their methods names may suggest procedural programming. They do not exploit and may prevent the use of object-orientation mechanisms: polymorphism and inheritance.

Premise: Antipatterns are conjectured in the literature to decrease the quality of systems. Yet, despite the many studies on antipatterns summarised in Section II, few studies have empirically investigated the impact of antipatterns on program comprehension. Yet, program comprehension is central to an effective software maintenance and evolution [4]: a good understanding of the source code of a system is essential to allow its inspection, maintenance, reuse, and extension. Therefore, a better understanding of the factors affecting developers's comprehension of source code is an efficient and effective way to ease maintenance.

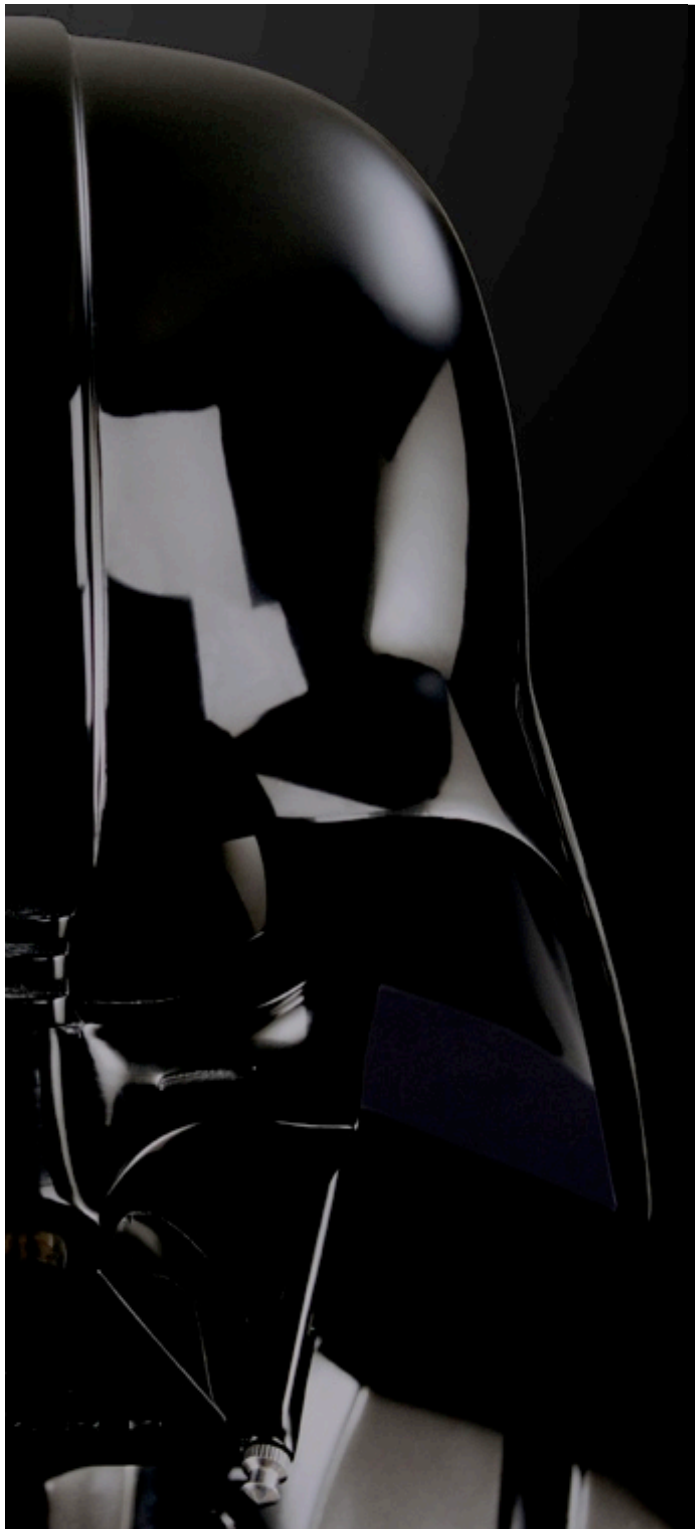
Goal: We want to gather quantitative evidence on the relations between antipatterns and program comprehension. In this paper, we focus on the system understandability, which is the degree to which the source code of a system can be easily understood by developers [5]. Gathering evidence on the relation between antipatterns and understandability is one more step [6] towards (dis)proving the conjecture in the literature about antipatterns and increasing our knowledge about the factors impacting program comprehension.

Study: We perform three experiments: we study whether systems with the antipattern Blob, first, and the Spaghetti Code, second, are more difficult to understand than systems without any antipattern. Third, we study whether systems with both Blob and Spaghetti Code are more difficult to understand than systems without any antipatterns. Each experiment is performed with 24 subjects

subjects are graduate students and professional developers with experience in software development and maintenance.

1534-5351/11 $26.00 © 2011 IEEE
DOI 10.1109/CSMR.2011.24

Bad Smells hinder code comprehensibility



[Khomh et al. EMSE 2012]

Empir Software Eng (2012) 17:243–275
DOI 10.1007/s10664-011-9171-y

An exploratory study of the impact of antipatterns on class change- and fault-proneness

Foutse Khomh · Massimiliano Di Penta · Yann-Gaël Guéhéneuc · Giuliano Antoniol

Published online: 6 August 2011

Editor: Jim Whitehead

Abstract Antipatterns are poor design choices that are conjectured to make object-oriented systems harder to maintain. We investigate the impact of antipatterns on classes in object-oriented systems by studying the relation between the presence of antipatterns and the change- and fault-proneness of the classes. We detect 13 antipatterns in 54 releases of ArgoUML, Eclipse, Mylyn, and Rhino, and analyse (1) to what extent classes participating in antipatterns have higher odds to change or to be subject to fault-fixing than other classes, (2) to what extent these odds (if higher) are due to the sizes of the classes or to the presence of antipatterns, and (3) what kinds of changes affect classes participating in antipatterns. We show that, in almost all releases of the four systems, classes participating in antipatterns are more change- and fault-prone than others. We also show that size alone cannot explain the higher odds of classes with antipatterns to underwent a (fault-fixing) change than other

We thank Marc Eaddy for making his data on faults freely available. This work has been partly funded by the NSERC Research Chairs in Software Change and Evolution and in Software Patterns and Patterns of Software.

F. Khomh (✉)
Department of Electrical and Computer Engineering,
Queen's University, Kingston, ON, Canada
e-mail: foutse.khomh@queensu.ca

M. D. Penta
Department of Engineering, University of Sannio, Benevento, Italy
e-mail: dipenta@unisannio.it

Y.-G. Guéhéneuc · G. Antoniol
SOCCER Lab. and Ptidej Team, Département de Génie Informatique et Génie Logiciel,
École Polytechnique de Montréal, Montréal, QC, Canada

Y.-G. Guéhéneuc

G. Antoniol

Bad Smells increase change and fault proneness

Identification of Move Method Refactoring Opportunities

Nikolaos Tsantalis, *Student Member, IEEE*, and Alexander Chatzigeorgiou, *Member, IEEE*

Abstract—Placement of attributes/methods within classes in an object-oriented system is usually guided by conceptual criteria and aided by appropriate metrics. Moving state and behavior between classes can help reduce coupling and increase cohesion, but it is nontrivial to identify where such refactorings should be applied. In this paper, we propose a methodology for the identification of Move Method refactoring opportunities that constitute a way for solving many common Feature Envy bad smells. An algorithm that employs the notion of distance between system entities (attributes/methods) and classes extracts a list of behavior-preserving refactorings based on the examination of a set of preconditions. In practice, a software system may exhibit such problems in many different places. Therefore, our approach measures the effect of all refactoring suggestions based on a novel Entity Placement metric that quantifies how well entities have been placed in system classes. The proposed methodology can be regarded as a semi-automatic approach since the designer will eventually decide whether a suggested refactoring should be applied or not based on conceptual or other design quality criteria. The evaluation of the proposed approach has been performed considering qualitative, metric, conceptual, and efficiency aspects of the suggested refactorings in a number of open-source projects.

Index Terms—Move Method refactoring, Feature Envy, object-oriented design, Jaccard distance, design quality.

1 Introduction

According to several principles and laws of object-oriented design [18], [25], designers should always strive for low coupling and high cohesion. A number of empirical studies have investigated the relation of coupling and cohesion metrics with external quality indicators. Basili et al. [3] and Briand et al. [7] have shown that coupling metrics can serve as predictors of fault-prone classes. Briand et al. [8] and Chaumun et al. [12] have shown high positive correlation between the impact of changes (ripple effects, changeability) and coupling metrics. Brito e Abreu and Melo [11] have shown that Coupling Factor [10] has very high positive correlation with defect density and rework. Binkley and Schach [4] have shown that modules with low coupling (as measured by Coupling Dependency Metric) require less maintenance effort and have fewer maintenance faults and fewer runtime failures. Chidamber et al. [14] have shown that high levels of coupling and lack of cohesion are associated with lower productivity, greater rework, and greater design effort. Consequently, low coupling and high cohesion can be regarded as indicators of good design quality in terms of maintenance.

Coupling or cohesion problems manifest themselves in many different ways, with *Feature Envy* bad smell being the most common symptom. Feature Envy is a sign of violating the principle of grouping behavior with related data and occurs when a method is "more interested in a class other than the one it actually is in" [17]. Feature Envy problems can be solved in three ways [17]: 1) by moving a method to the class that it envies (Move Method refactoring); 2) by extracting a method fragment and then moving it to the class that it envies (Extract + Move Method refactoring); and 3) by moving an attribute to the class that envies it (Move Field refactoring). The correct application of the appropriate refactorings in a given system improves its design quality without altering its external behavior. However, the identification of methods, method fragments, or attributes that have to be moved to target classes is not always trivial since existing metrics may highlight coupling/cohesion problems but do not suggest specific refactoring opportunities.

Our methodology considers only Move Method refactorings as solutions to the Feature Envy design problem. Moving attributes (fields) from one class to another has not been considered, since this strategy would lead to contradicting refactoring suggestions with respect to the strategy of moving methods. Moreover, fields have stronger conceptual binding to the classes in which they are initially placed since they are less likely than methods to change once assigned to a class.

In this paper, the notion of distance between an entity (attribute or method) and a class is employed to support the automated identification of Feature Envy bad smells. To this end, an algorithm has been developed that extracts Move Method refactoring suggestions. For each method of the system, the algorithm forms a set of candidate target classes where the method can possibly be moved by examining the entities that it accesses from the system classes (system classes refer to the application or program under consideration excluding imported libraries or frameworks). Then, it iterates over the candidate target classes according to the number of accessed entities and the distance of the method from each candidate class. Eventually, it selects as the final

• *The authors are with the Department of Applied Informatics, University of Macedonia, 54006 Thessaloniki, Greece.*
E-mail: nikos@java.uom.gr, achat@uom.gr.

Manuscript received 15 Apr. 2008; revised 5 Dec. 2008; accepted 15 Dec. 2008; published online 6 Jan. 2009.
Recommended for acceptance by H. Ossher.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2008-04-0150.
Digital Object Identifier no. 10.1109/TSE.2009.1.

DECOR: A Method for the Specification and Detection of Code and Design Smells

Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur

Abstract—Code and design smells are poor solutions to recurring implementation and design problems. They may hinder the evolution of a system by making it hard for software engineers to carry out changes. We propose three contributions to the research field related to code and design smells: 1) DECOR, a method that embodies and defines all the steps necessary for the specification and detection of code and design smells, 2) DETEX, a detection technique that instantiates this method, and 3) an empirical validation in terms of precision and recall of DETEX. The originality of DETEX stems from the ability for software engineers to specify smells at a high level of abstraction using a consistent vocabulary and domain-specific language for automatically generating detection algorithms. Using DETEX, we specify four well-known design smells: the antipatterns Blob, Functional Decomposition, Spaghetti Code, and Swiss Army Knife, and their 15 underlying code smells, and we automatically generate their detection algorithms. We apply and validate the detection algorithms in terms of precision and recall on XERCES v2.7.0, and discuss the precision of these algorithms on 11 open-source systems.

Index Terms—Antipatterns, design smells, code smells, specification, metamodeling, detection, Java.

1 Introduction

Software systems need to evolve continually to cope with ever-changing requirements and environments. However, opposite to design patterns [1], code and design smells —"poor" solutions to recurring implementation and design problems—may hinder their evolution by making it hard for software engineers to carry out changes.

Code and design smells include low-level or local problems such as code smells [2], which are usually symptoms of more global design smells such as anti-patterns [3]. Code smells are indicators or symptoms of the possible presence of design smells. Fowler [2] presented 22 code smells, structures in the source code that suggest the possibility of refactorings. Duplicated code, long methods, large classes, and long parameter lists are just a few symptoms of design smells and opportunities for refactorings.

One example of a design smell is the Spaghetti Code antipattern,[1] which is characteristic of procedural thinking in object-oriented programming. Spaghetti Code is revealed by classes without structure that declare long methods without parameters. The names of the classes and methods may suggest procedural programming. Spaghetti Code does not exploit object-oriented mechanisms, such as polymorphism and inheritance, and prevents their use.

We use the term "smells" to denote both code and design smells. This use does not exclude that, in a particular context, a smell can be the best way to actually design or implement a system. For example, parsers generated automatically by parser generators are often Spaghetti Code, i.e., very large classes with very long methods. Yet, although such classes "smell," software engineers must manually evaluate their possible negative impact according to the context.

The detection of smells can substantially reduce the cost of subsequent activities in the development and maintenance phases [4]. However, detection in large systems is a very time and resource-consuming and error-prone activity [5] because smells cut across classes and methods and their descriptions leave much room for interpretation.

Several approaches, as detailed in Section 2, have been proposed to specify and detect smells. However, they have three limitations. First, the authors do not explain the analysis leading to the specifications of smells and the underlying detection framework. Second, the translation of the specifications into detection algorithms is often black box, which prevents replication. Finally, the authors do not present the results of their detection on a representative set of smells and systems to allow comparison among approaches. So far, reported results concern proprietary systems and a reduced number of smells.

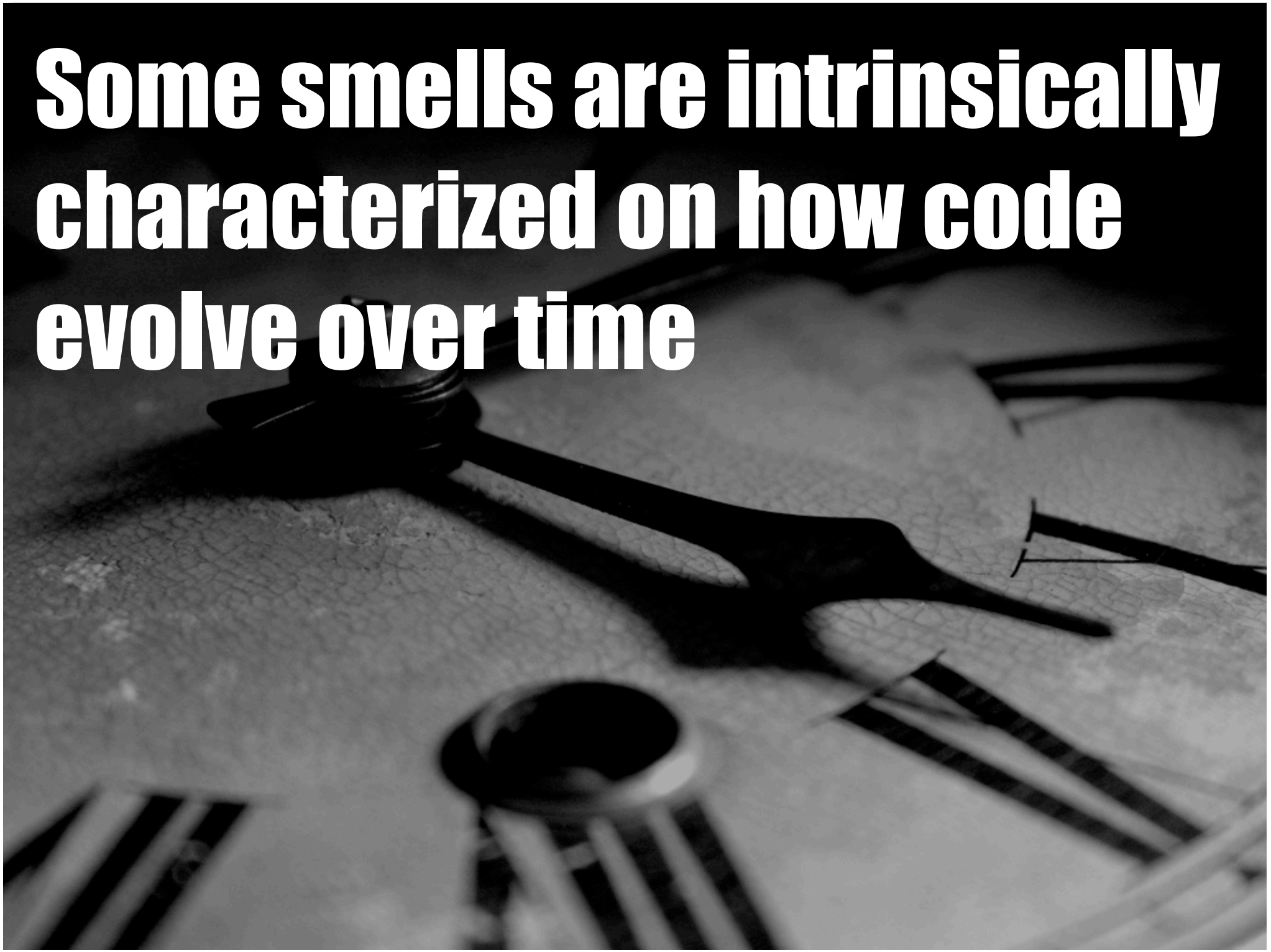
We present three contributions to overcome these limitations. First, we propose *DEtection & CORrection*[2] (DECOR), a method that describes all the steps necessary for the specification and detection of code and design

1. This smell, like those presented later on, is really in between implementation and design.

• *N. Moha is with the Triskell Team, IRISA—Université de Rennes 1, Room F233, INRIA Rennes-Bretagne Atlantique Campus de Beaulieu, 35042 Rennes cedex, France. E-mail: naouel@irisa.fr.*
• *Y.-G. Guéhéneuc is with the Département de Génie Informatique et Génie Logiciel, École Polytechnique de Montréal, C.P. 6079, succursale Centre-Ville Montréal, QC, H3C 3A7, Canada.*
E-mail: yann-gael.gueheneuc@polymtl.ca.
• *L. Duchien and A.-F. Le Meur are with INRIA, Lille-Nord Europe, Parc Scientifique de la Haute Borne 40, avenue Halley-BP. A, Park Plaza 59650 Villeneuve d'Ascq, France.*
E-mail: {Laurence.Duchien, Anne-Francoise.Le_Meur}@inria.fr.

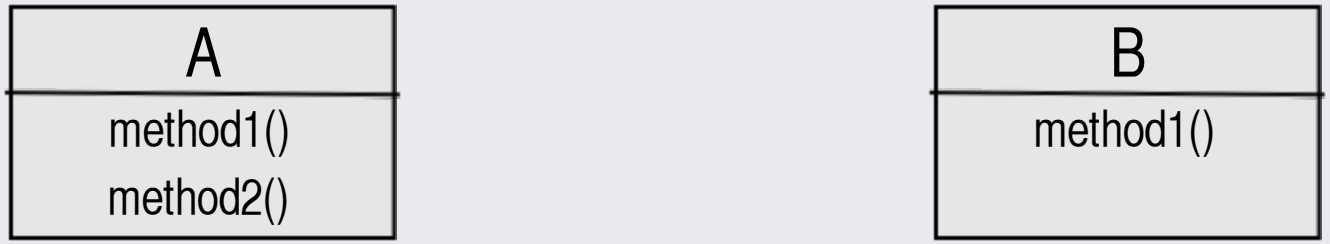
Manuscript received 17 Aug. 2008; revised 8 May 2009; accepted 19 May 2009; published online 31 July 2009.
Recommended for acceptance by M. Harman.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2008-08-0255.
Digital Object Identifier no. 10.1109/TSE.2009.50.

2. Correction is future work.


Some smells are intrinsically characterized on how code evolve over time

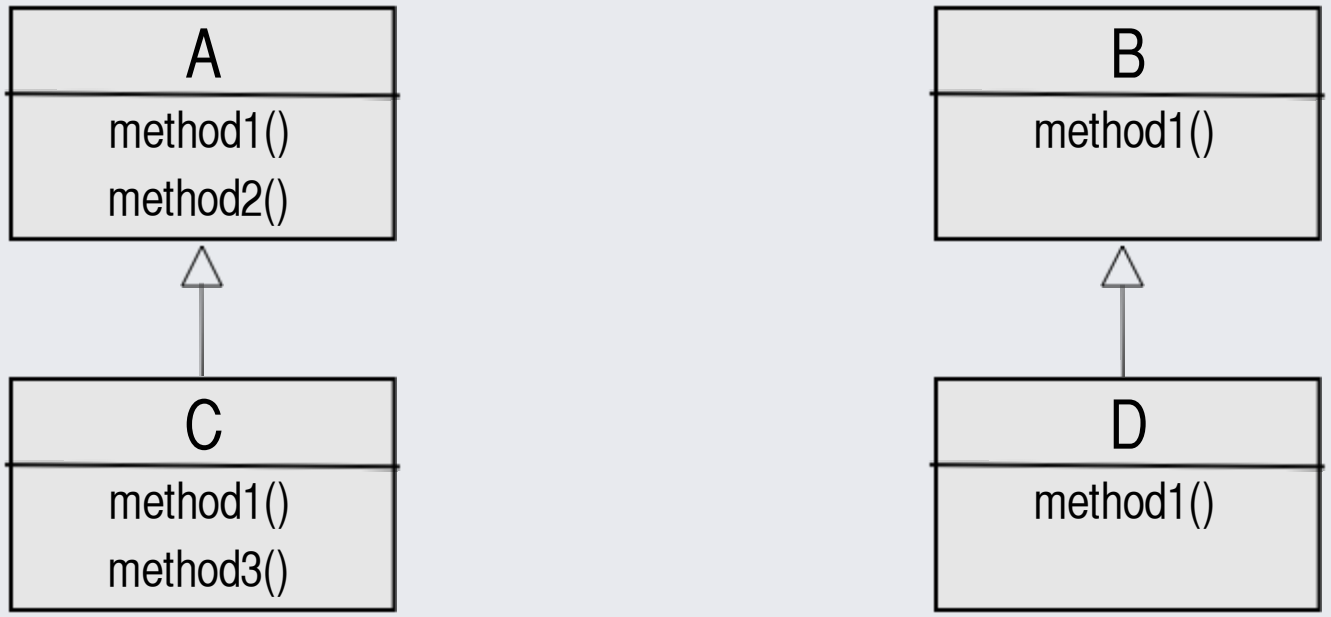
Parallel Inheritance

Every time you make a subclass of one class, you also have to make a subclass of another



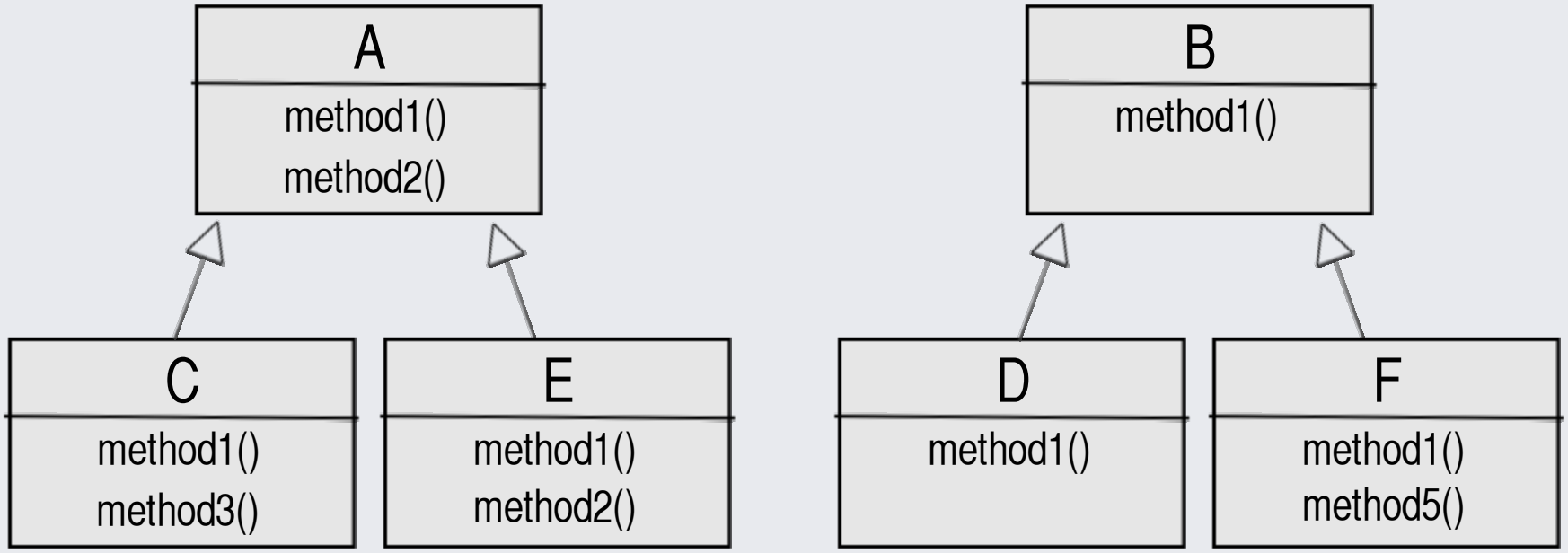
Parallel Inheritance

Every time you make a subclass of one class, you also have to make a subclass of another

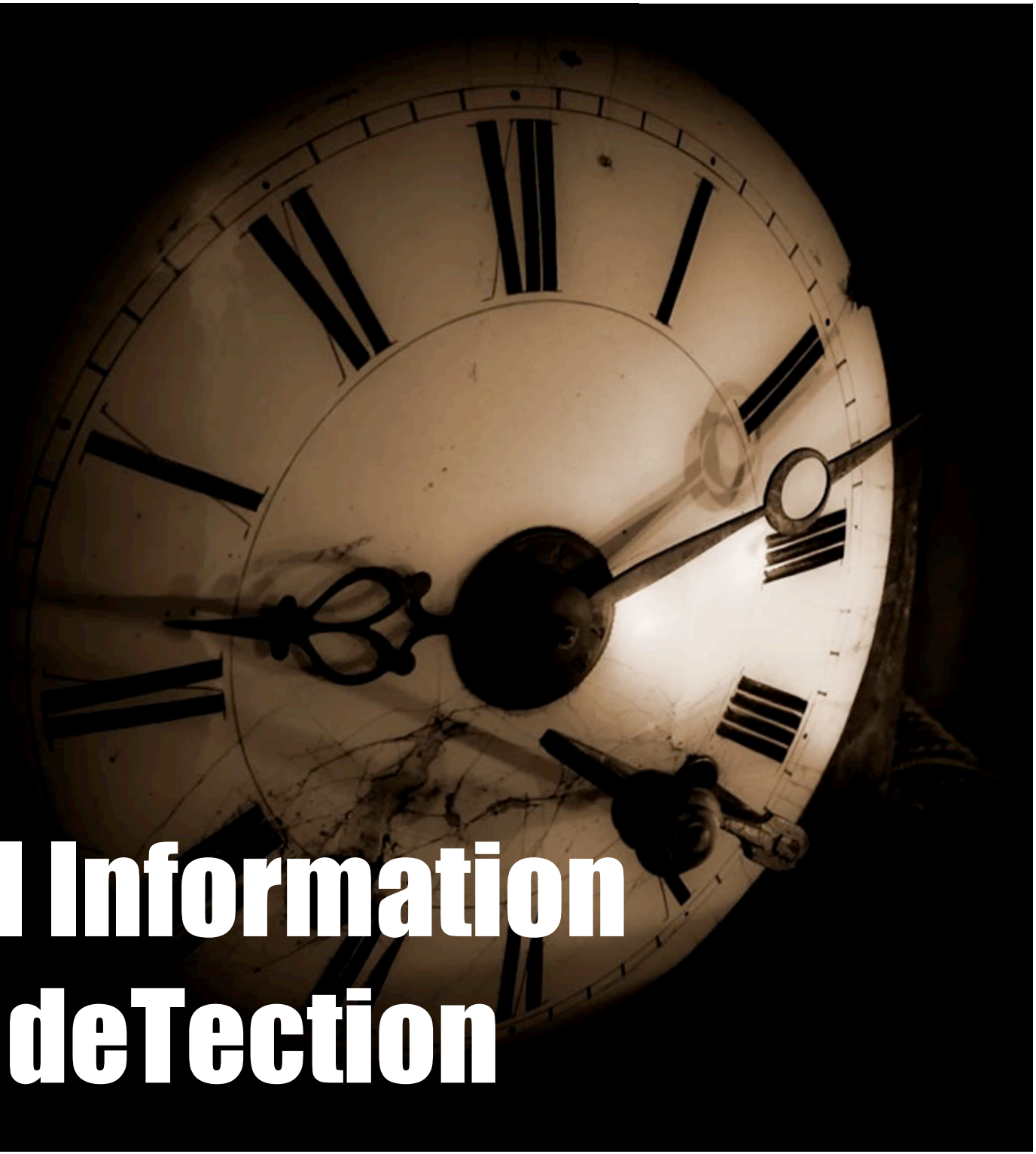


Parallel Inheritance

Every time you make a subclass of one class, you also have to make a subclass of another



HIST



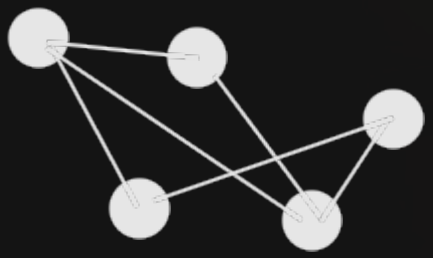
Historical Information for Smell deTection

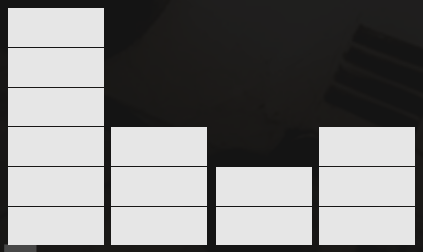
HIST

Change History Extractor

log download *code analyzer*

Code Smells Detector





Association rule discovery to capture co-changes between entities

Analysis of change frequency of some specific entities

Historical Information for Smell deTection

Code Smells Detector

divergent change

A class is changed in different ways for different reasons

Solution:
Extract Class Refactoring

Detection

Classes containing at least two sets of methods such that:

(i) all methods in the set change together as detected by the association rules

(ii) each method in the set does not change with methods in other sets

Code Smells Detector

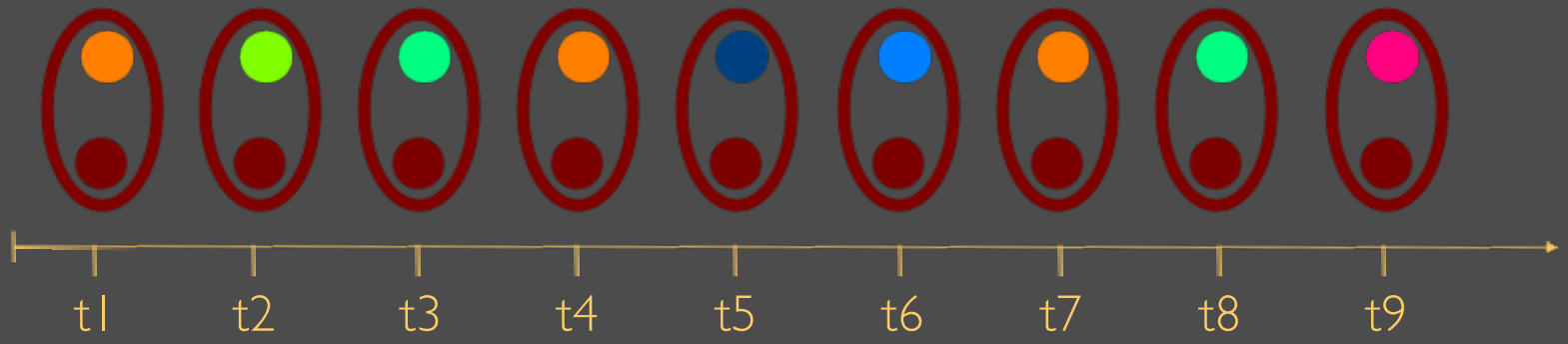
blob

A class implementing several responsibilities, having a large size, and dependencies with data classes

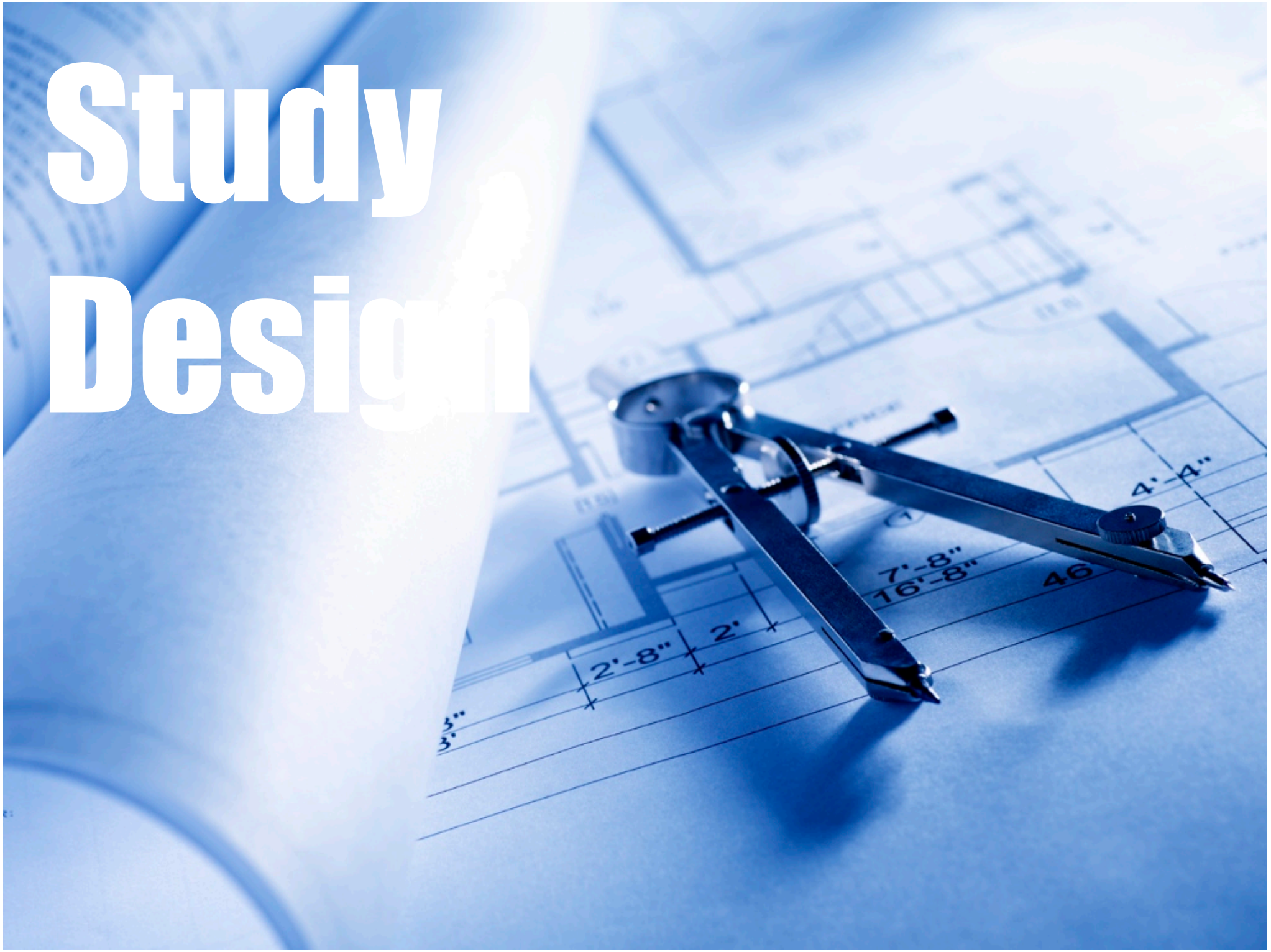
Solution:
Extract Class refactoring

Detection

Blobs are identified as classes frequently modified in commits involving at least another class.



Study Design



Apache Tomcat

Apache Ant

jEdit

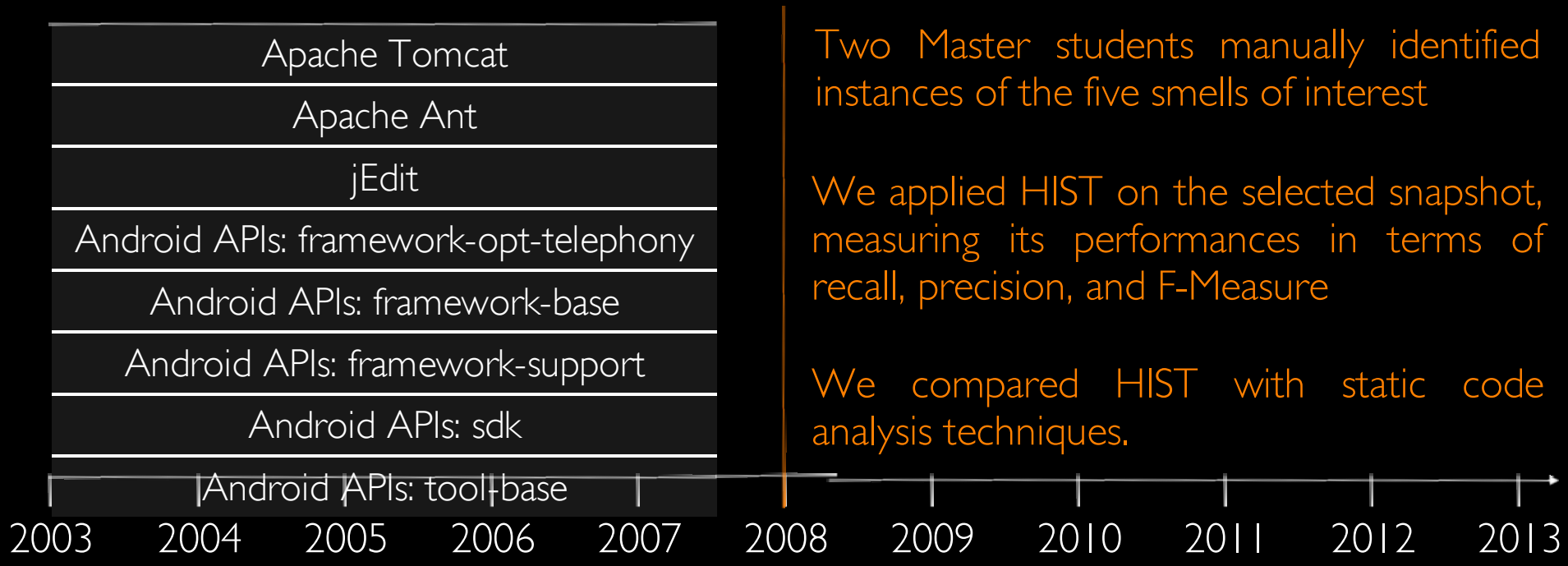
Android APIs: framework-opt-telephony

Android APIs: framework-base

Android APIs: framework-support

Android APIs: sdk

Android APIs: tool-base



Two Master students manually identified instances of the five smells of interest

We applied HIST on the selected snapshot, measuring its performances in terms of recall, precision, and F-Measure

We compared HIST with static code analysis techniques.

Blob: HIST ***vs*** DECOR [Moha et al., TSE 2010]

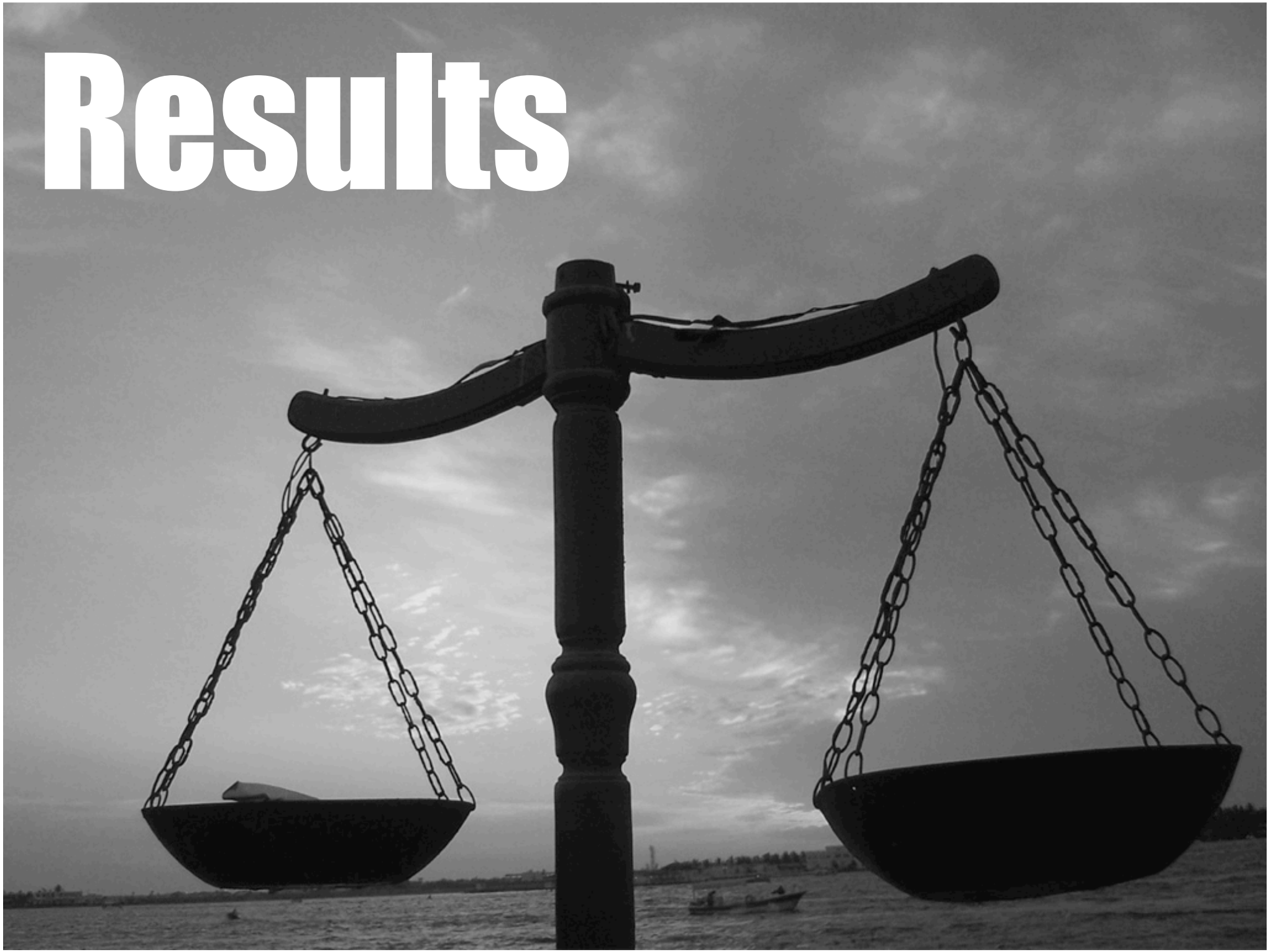
Feature Envy: HIST ***vs*** JDeodorant [Tsantalis et al., TSE 2009]

Divergent Change: HIST ***vs*** [Classes having a low cohesion as measured by the Connectivity metric]

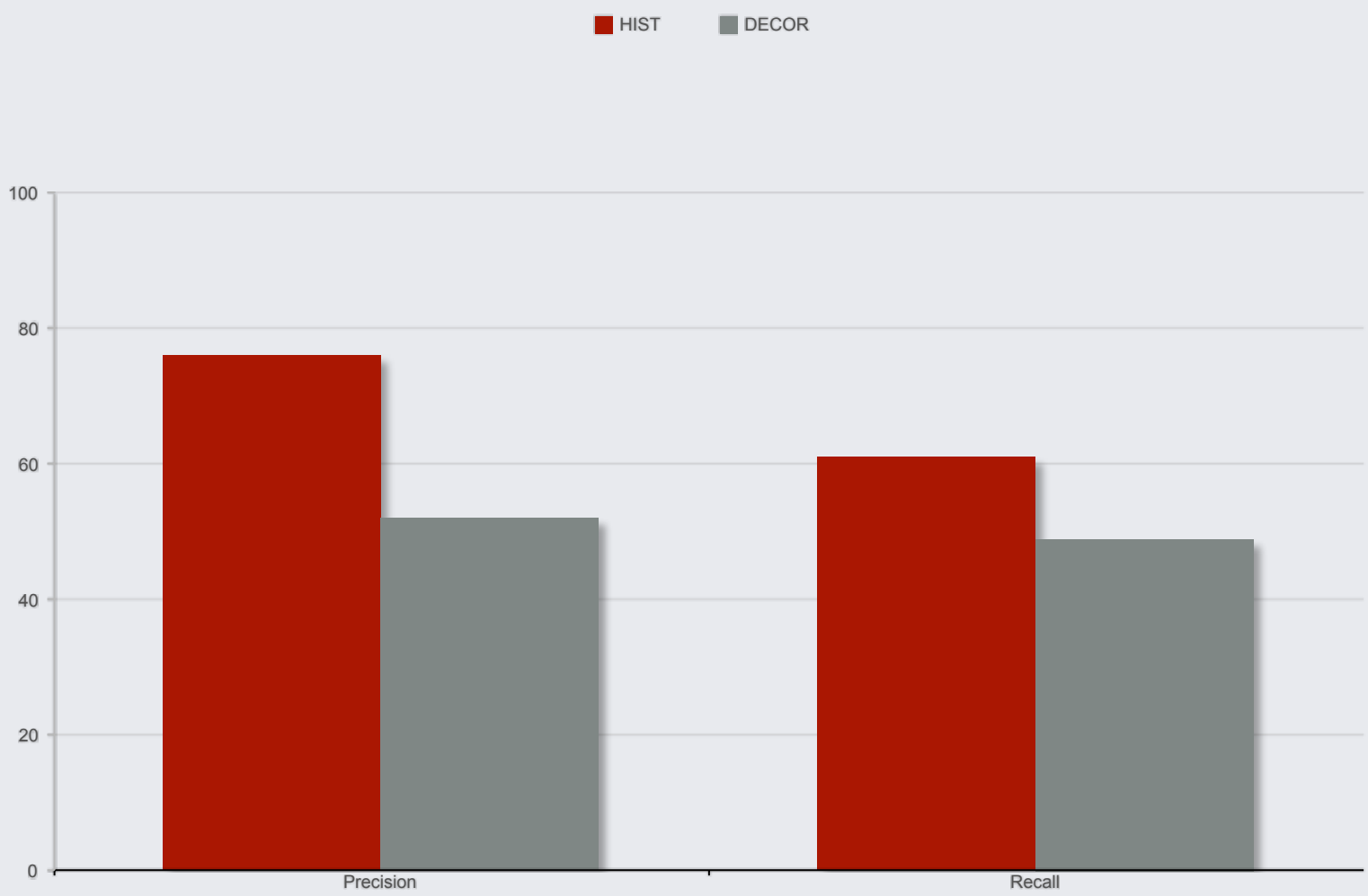
Shotgun Surgery: HIST ***vs*** [Classes having at least a method invoking at least 4 different classes]

Parallel Inheritance: HIST ***vs*** [Pairs of classes (i) both belonging to hierarchies and (ii) have the same prefix in the class name]

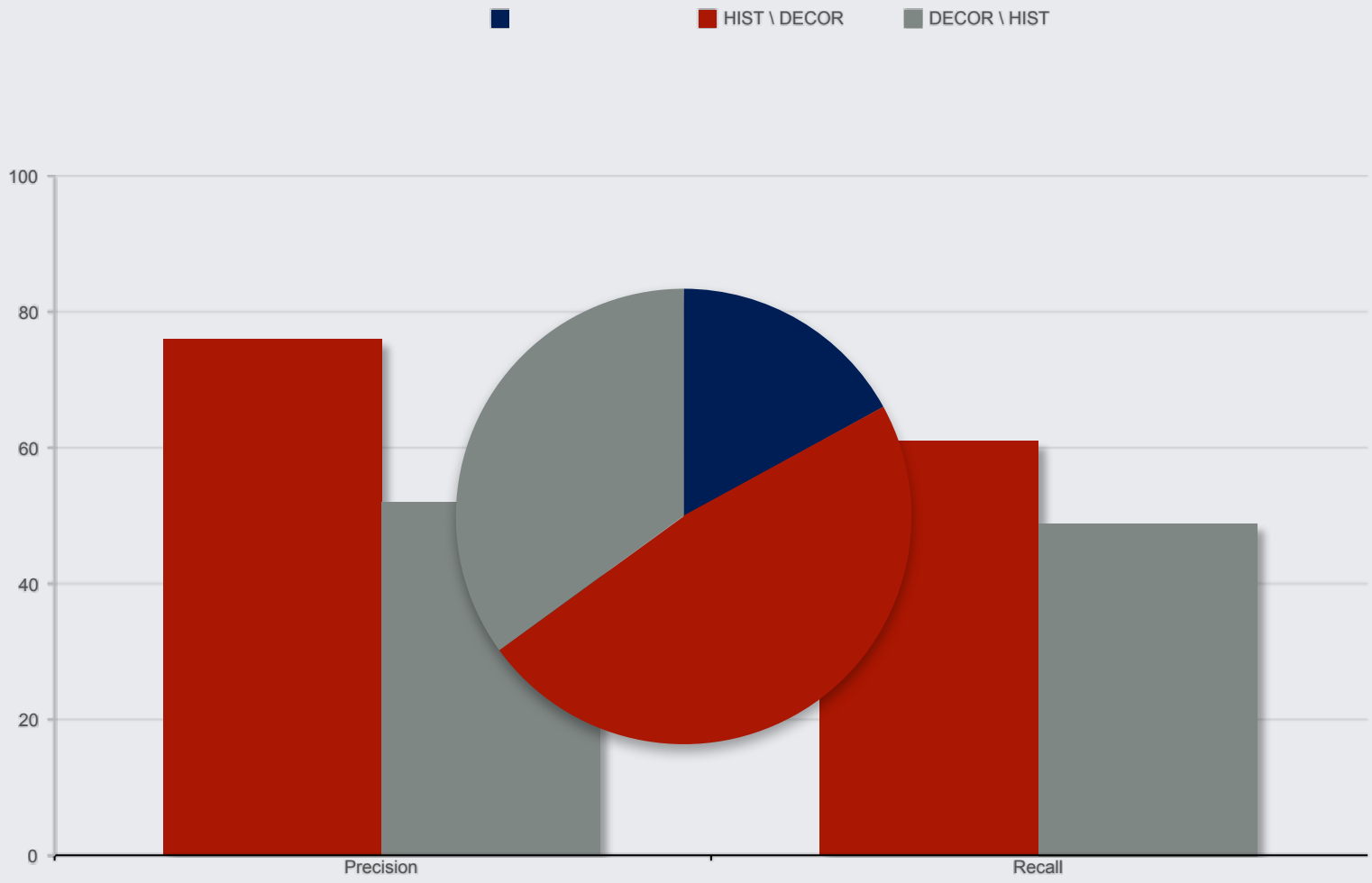
Results



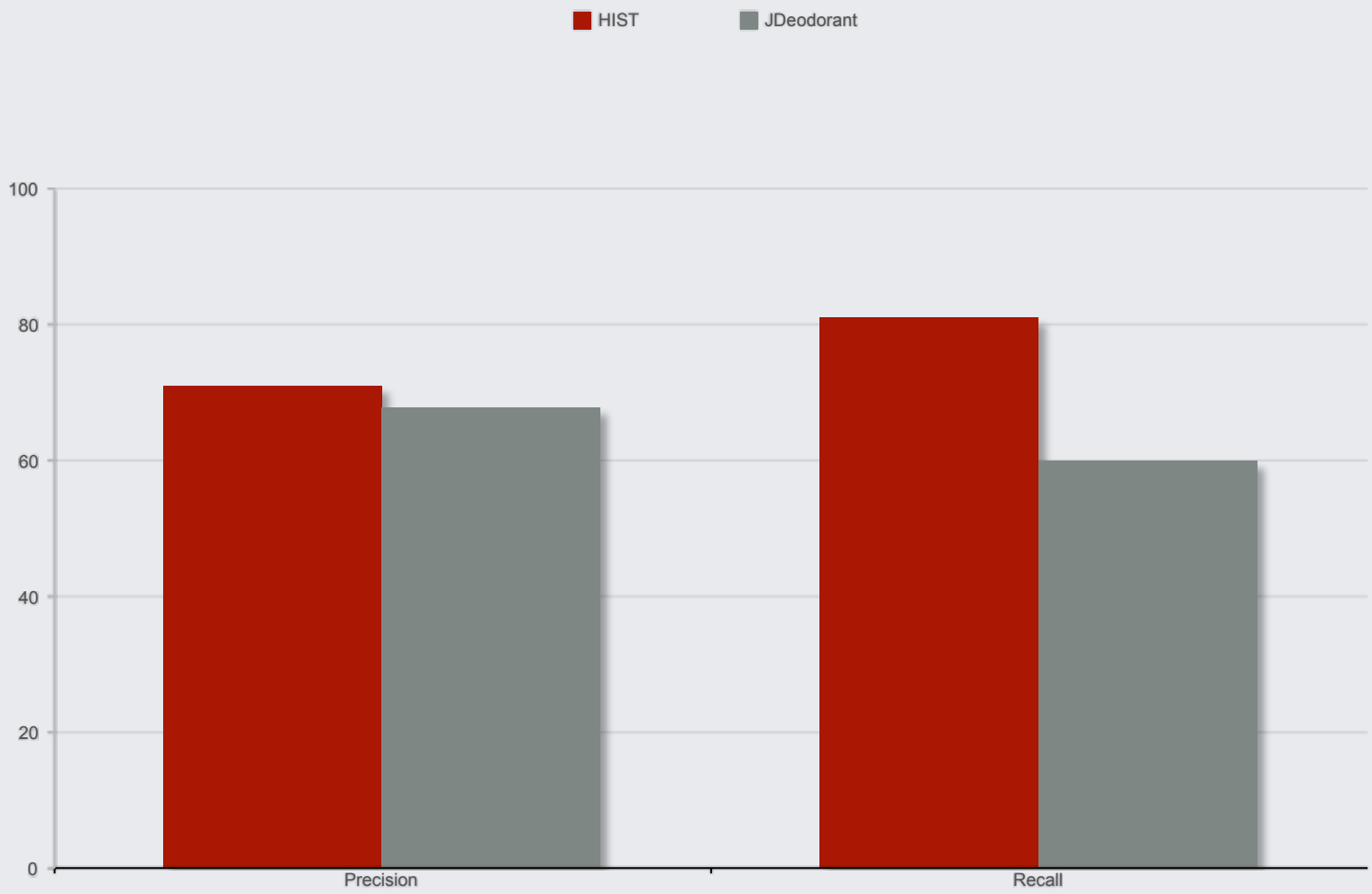
Blob



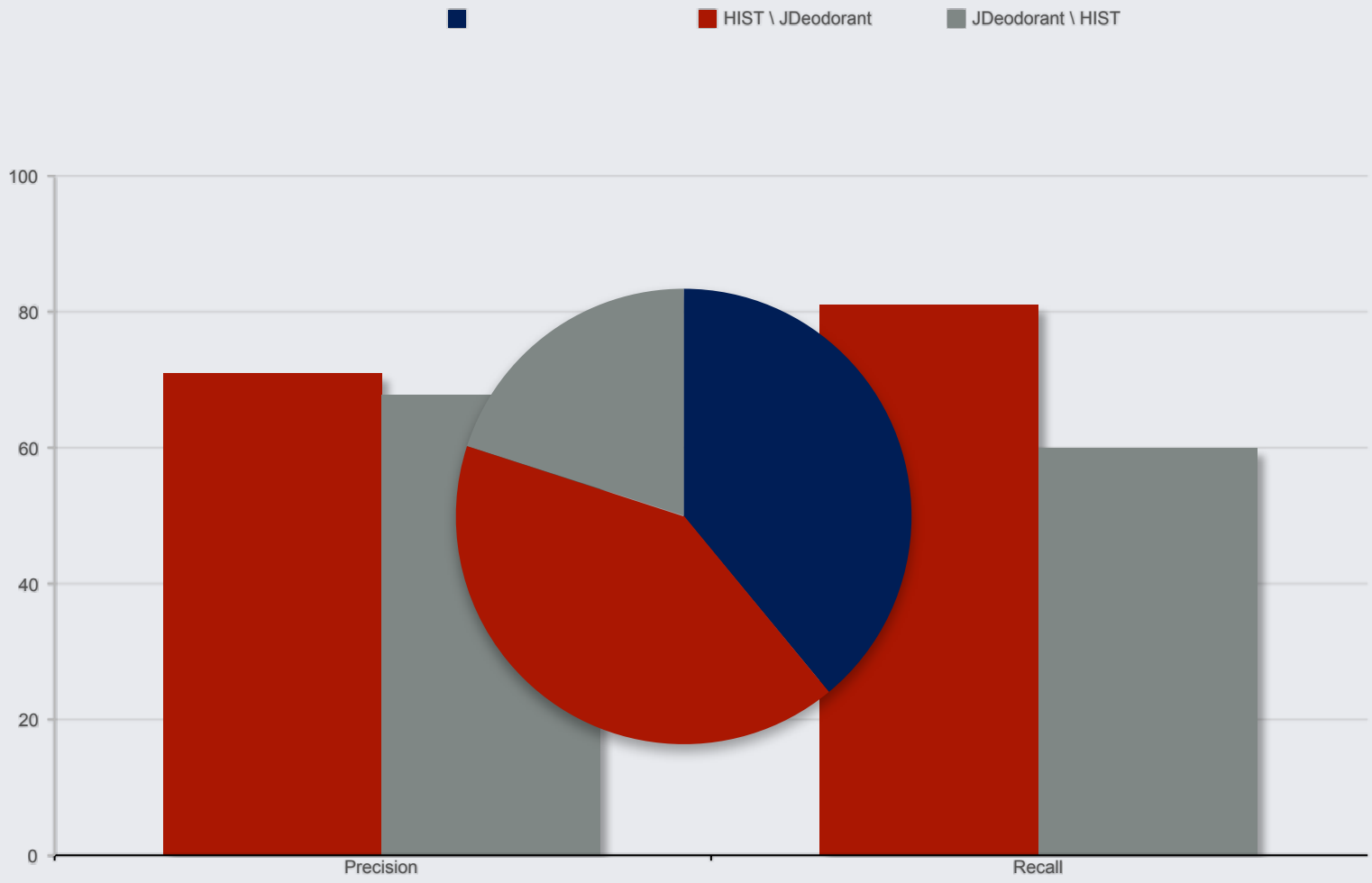
Blob



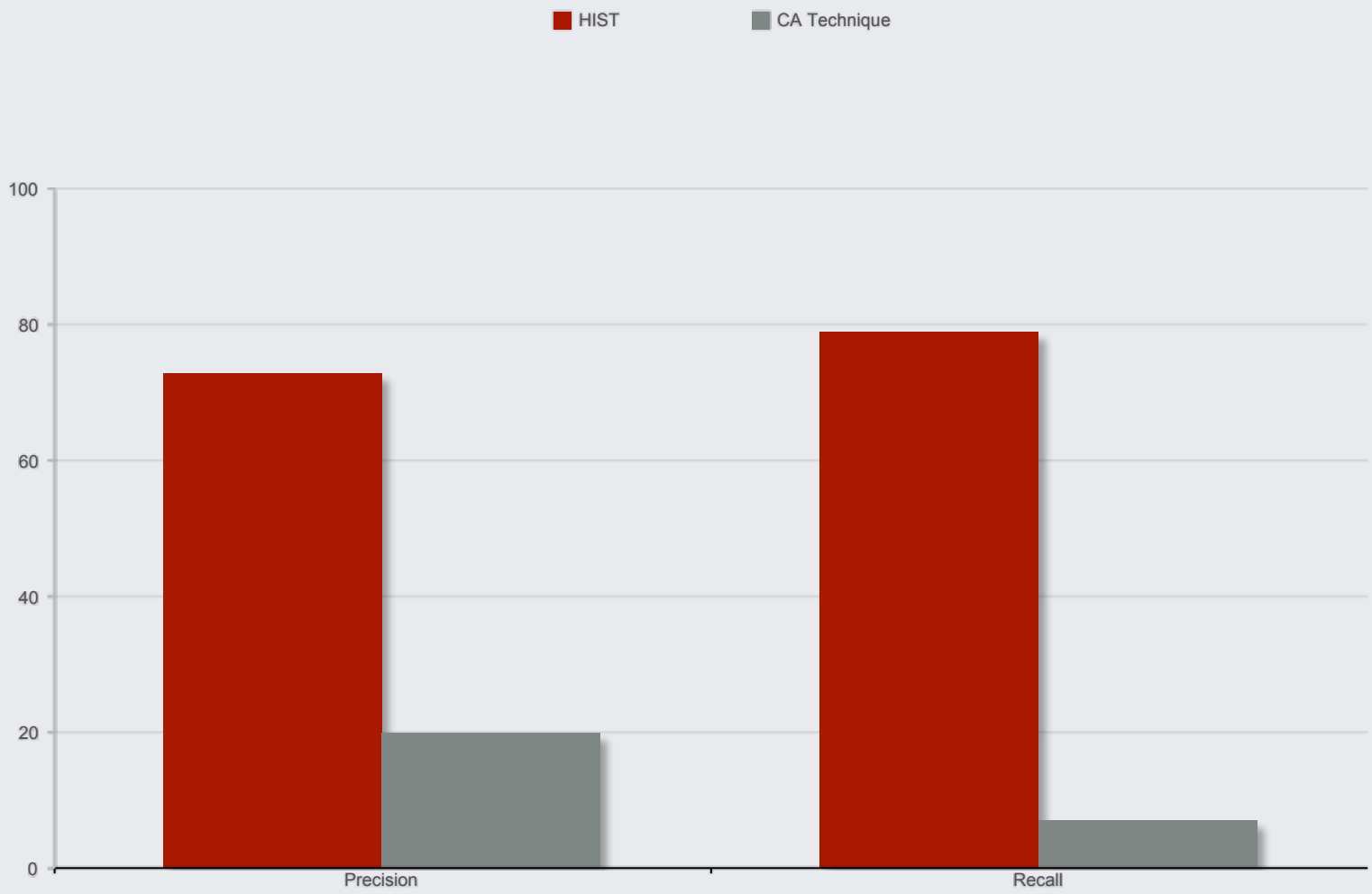
Feature Envy



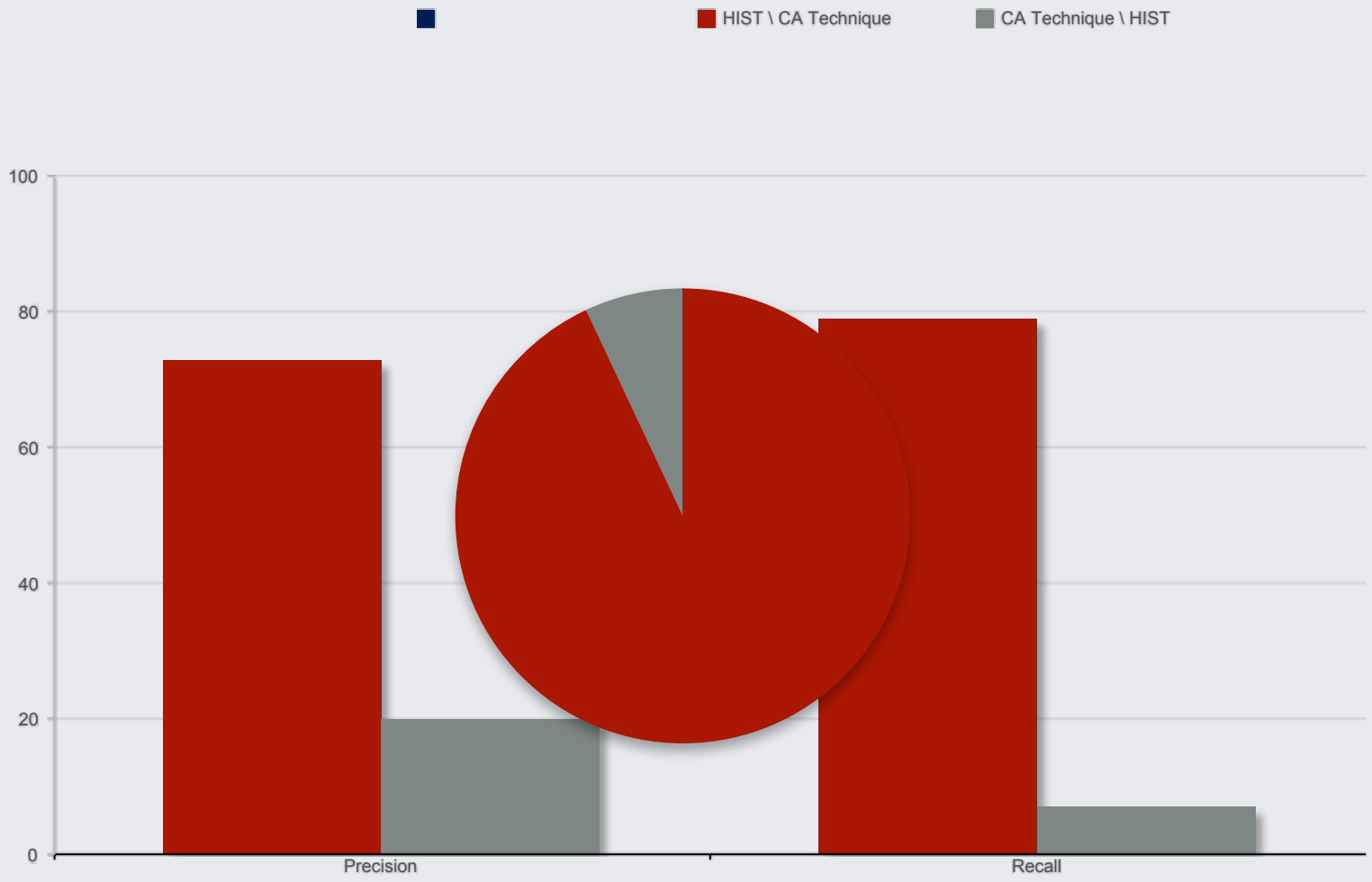
Feature Envy



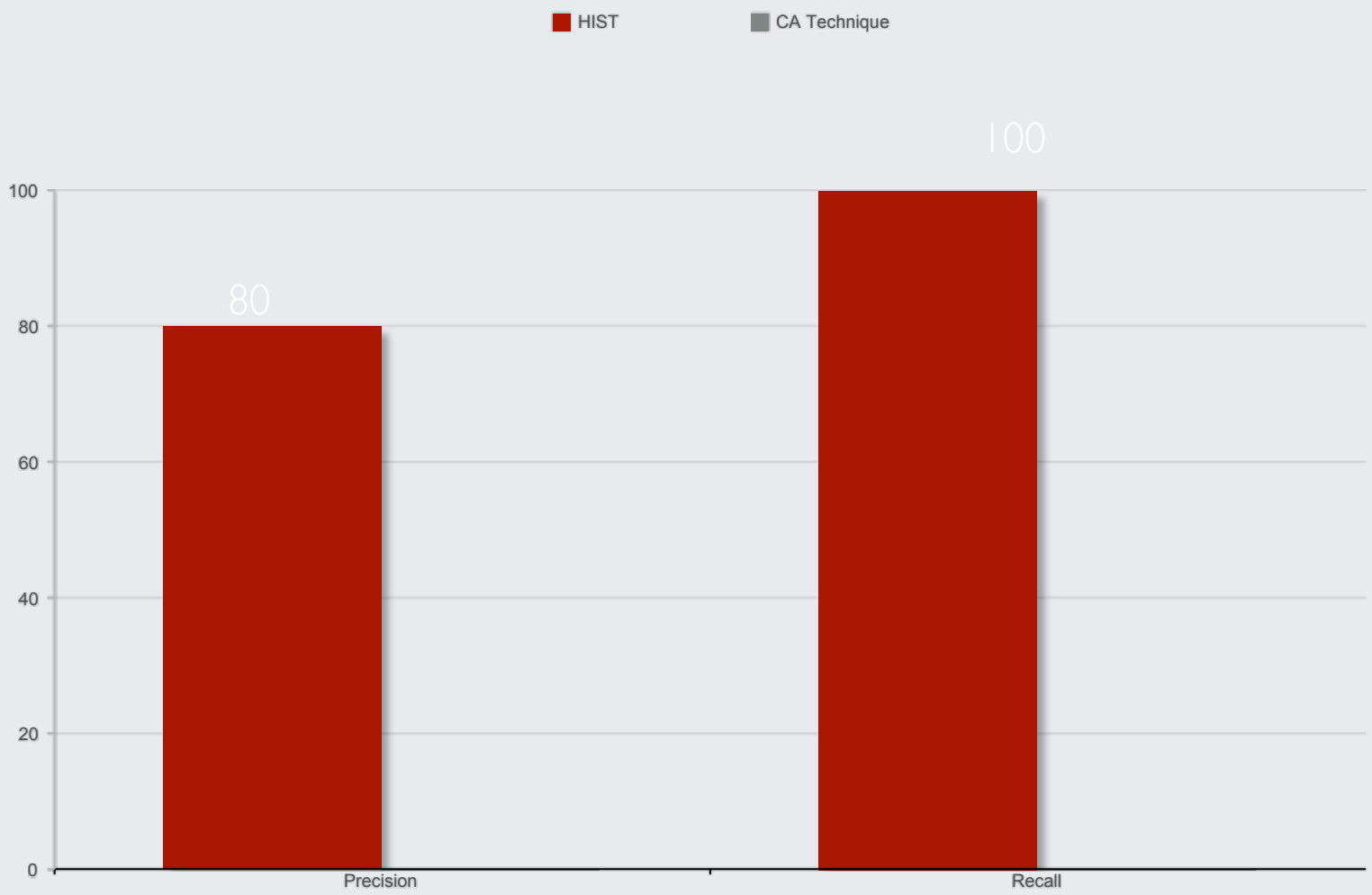
Divergent Change



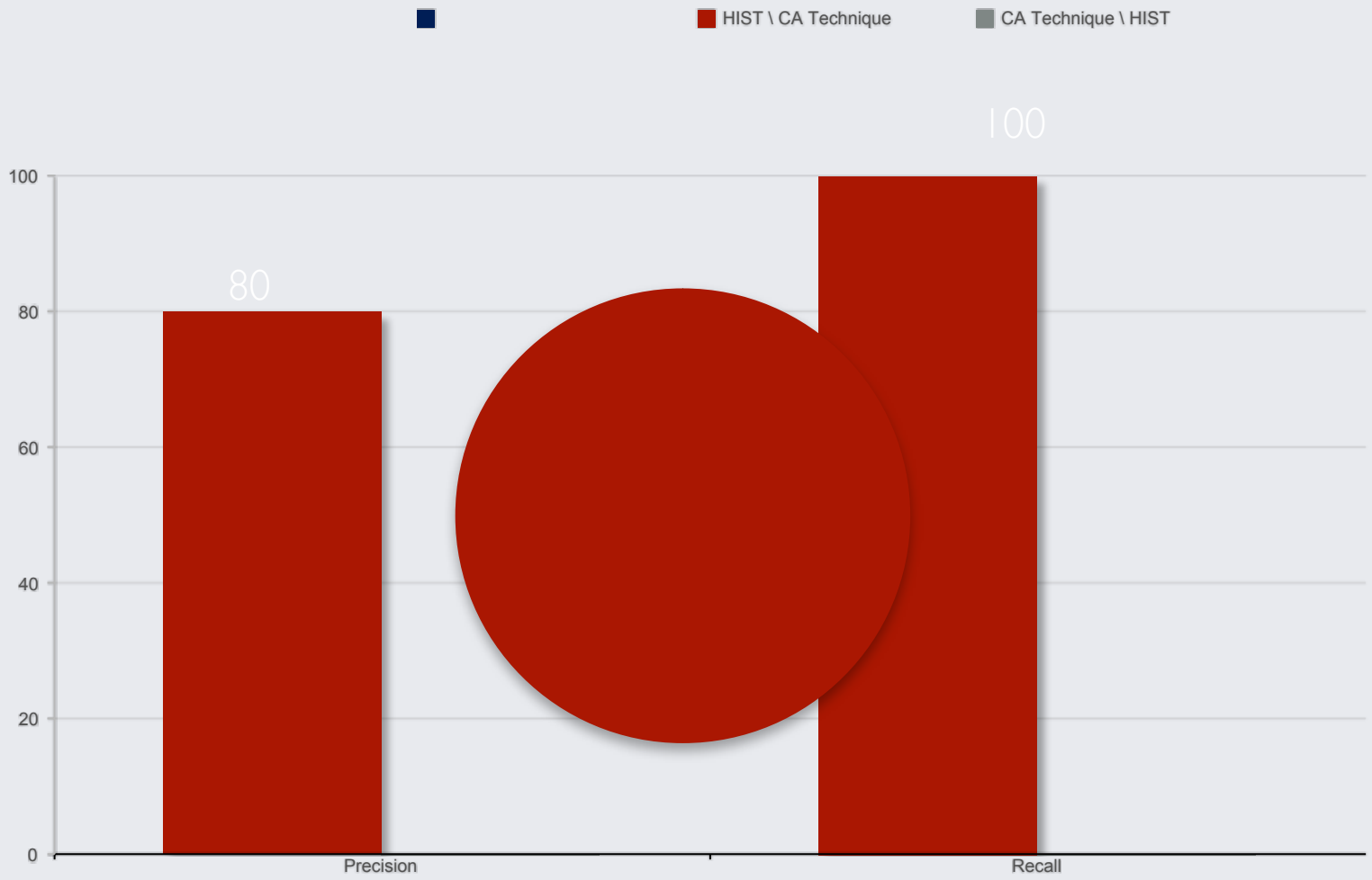
Divergent Change



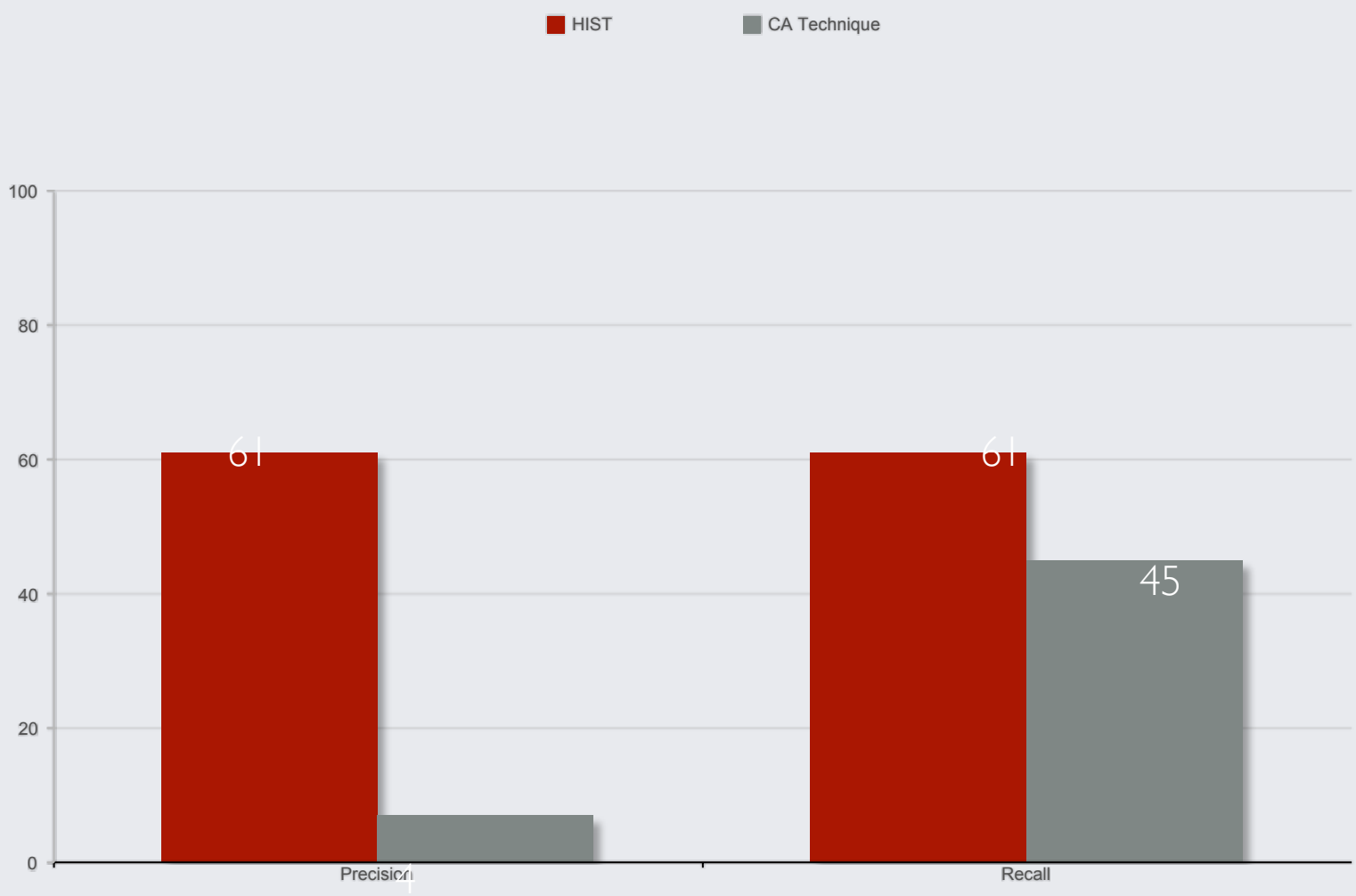
Shotgun Surgery



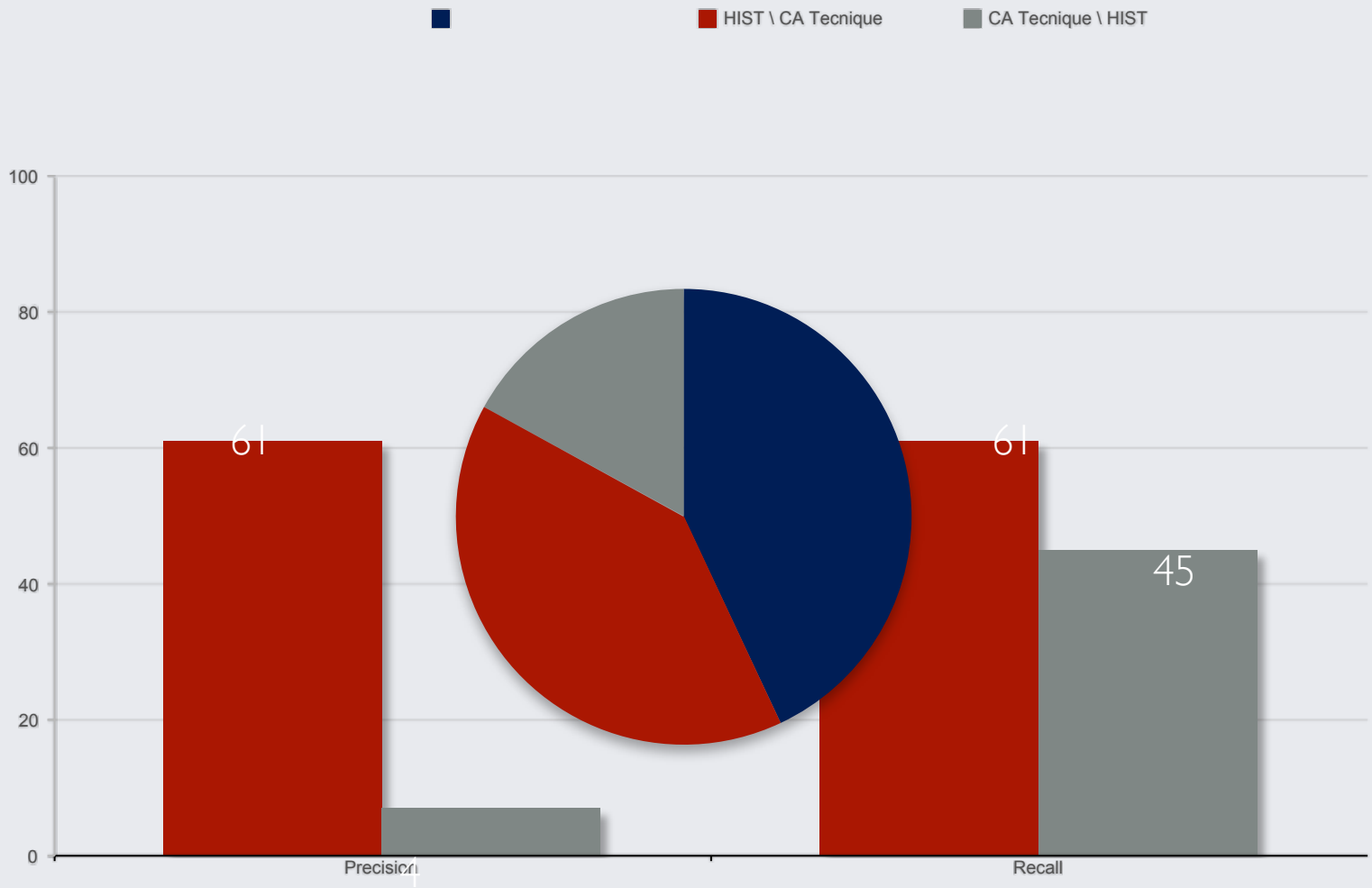
Shotgun Surgery



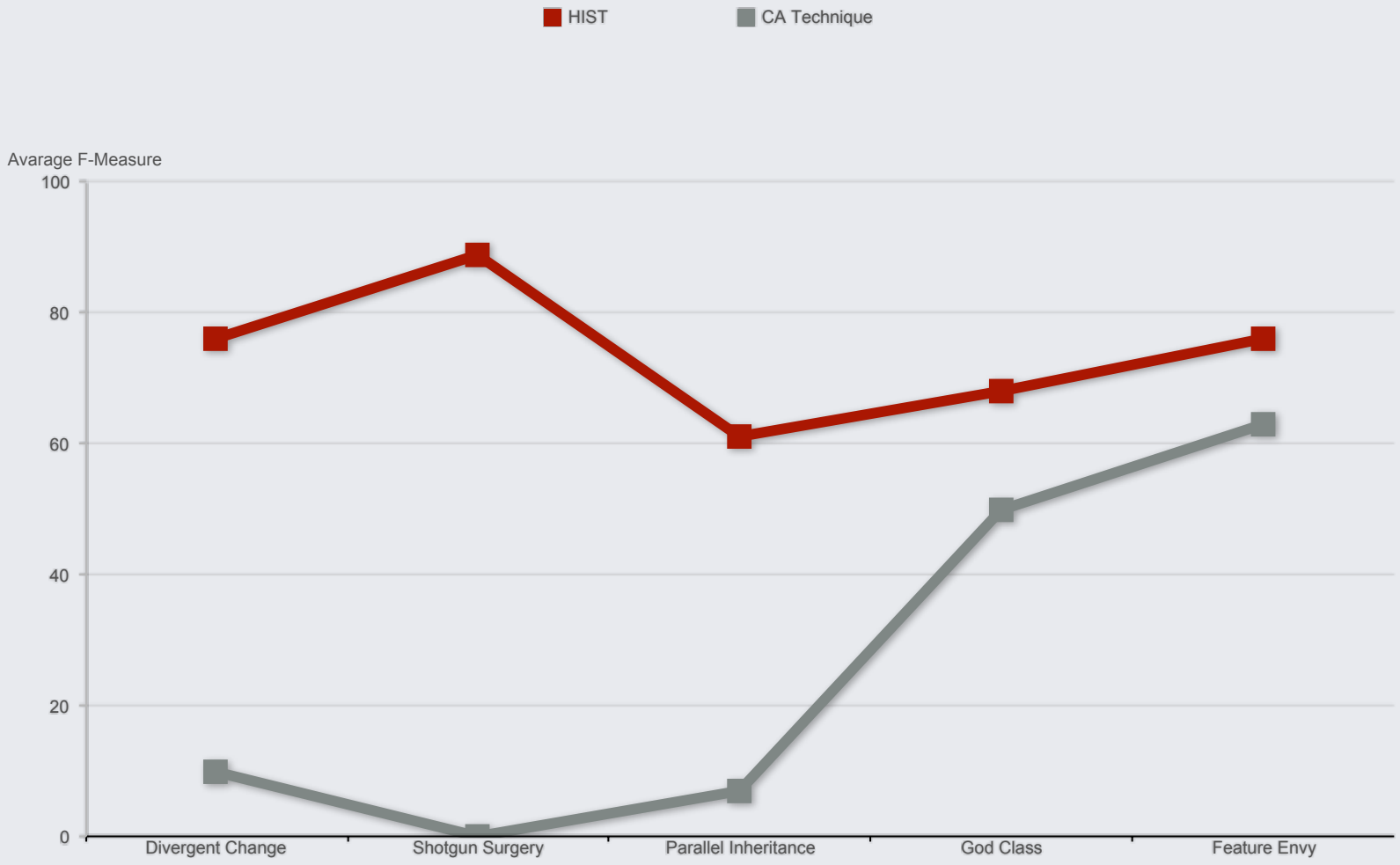
Parallel Inheritance



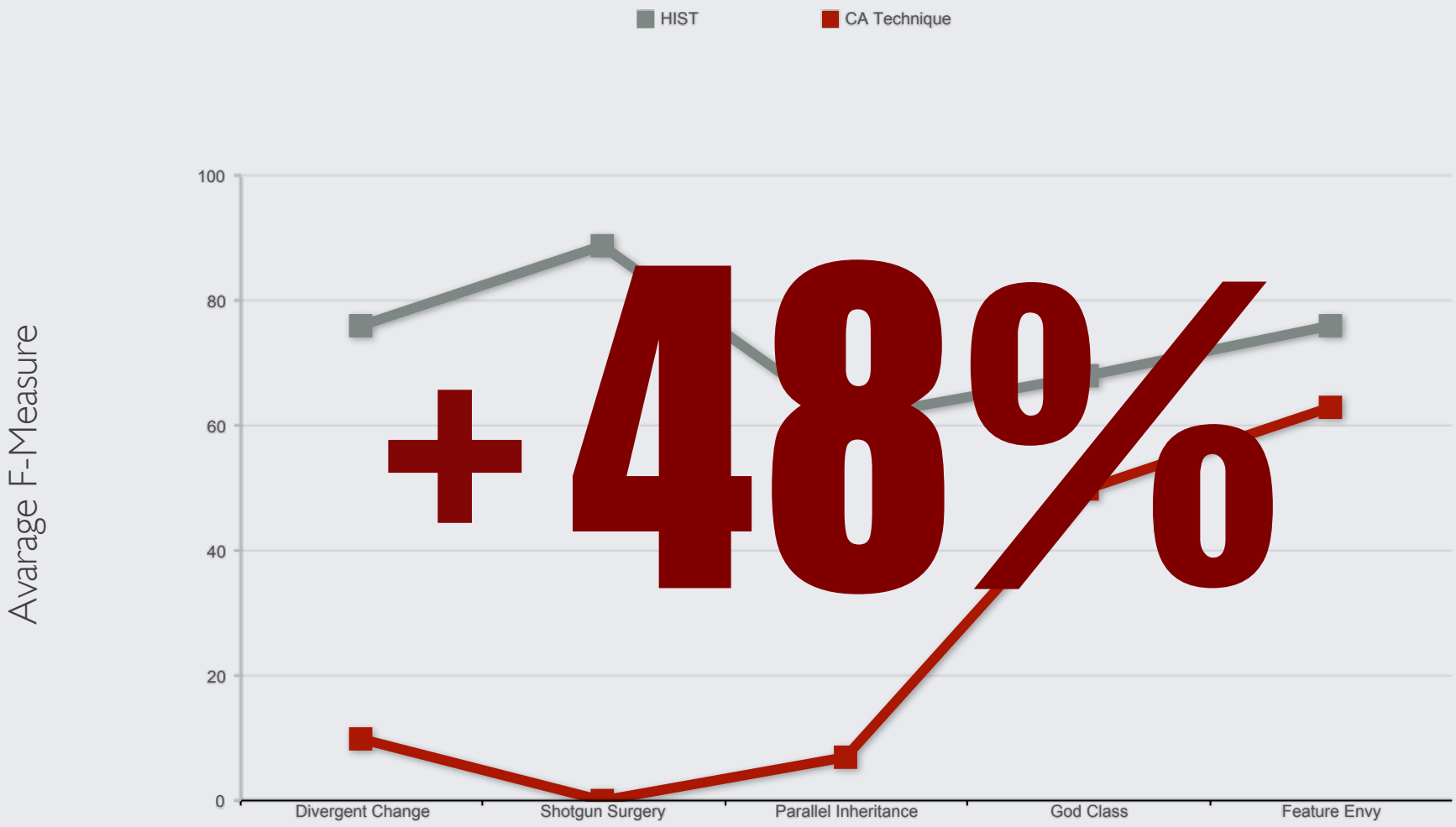
Parallel Inheritance



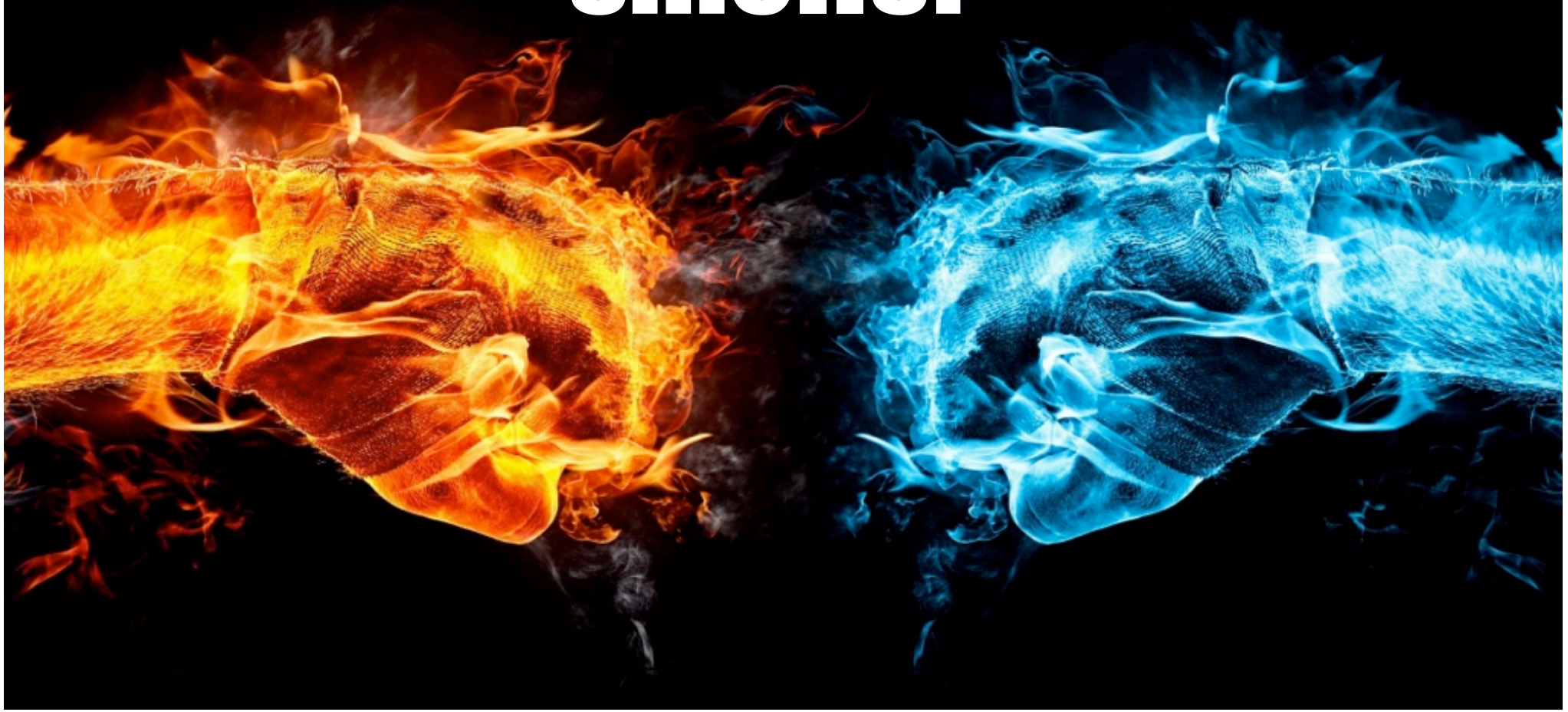
Results – Summary



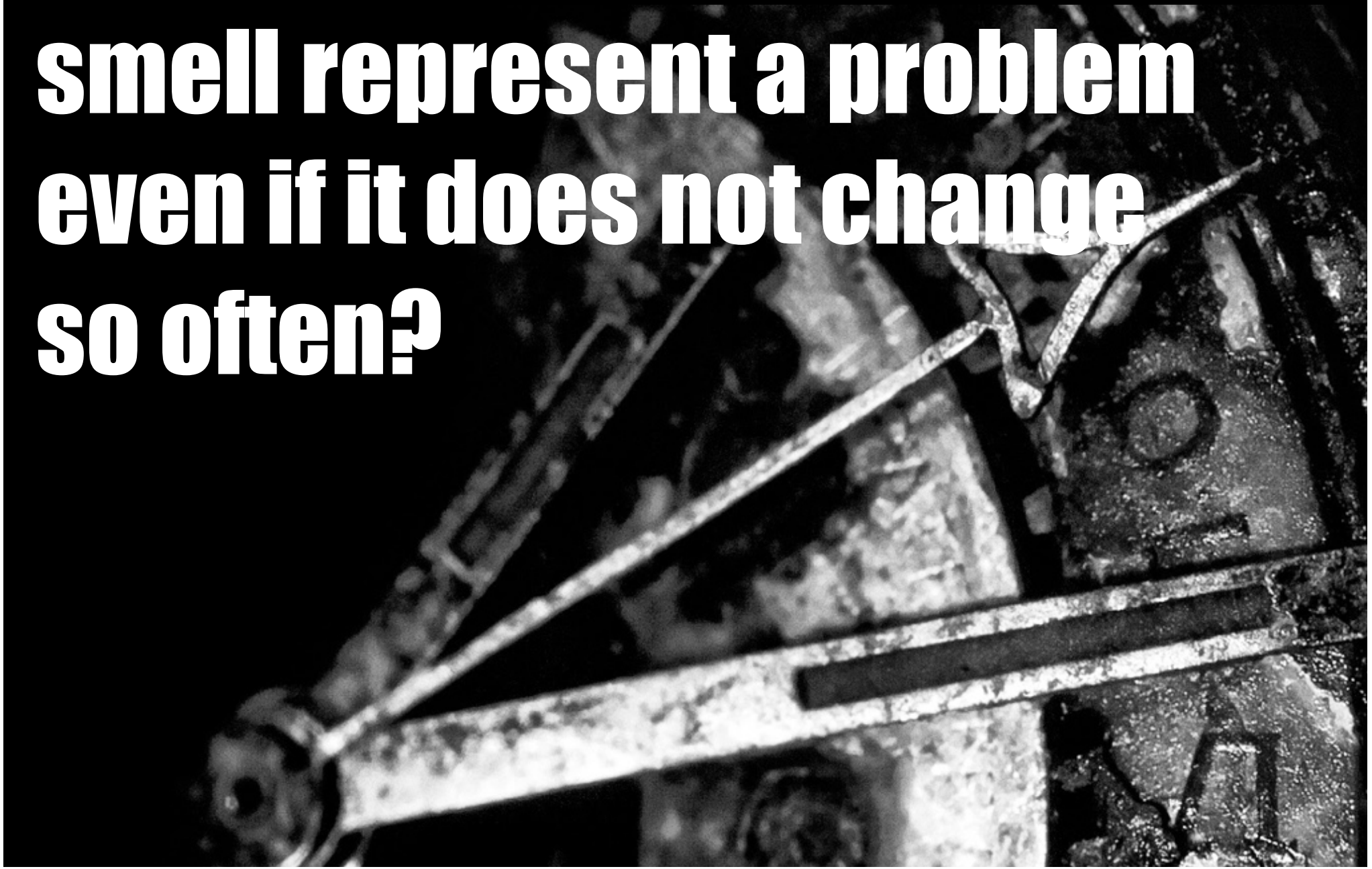
Results – Summary



Can we define a hybrid approach to detect smells?



Does a class affected by a smell represent a problem even if it does not change so often?



Some smells are intrinsically characterized on how code evolve over time

HIST

Historical Information for Smell deTection

Change History Extractor

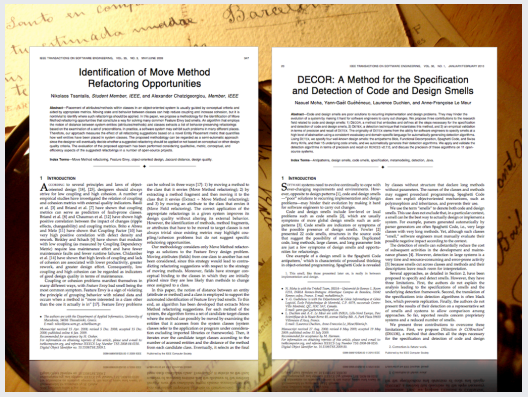
log download → code analyzer

Code Smells Detector

Association rule discovery to capture co-changes between entities

Analysis of change frequency of some specific entities

Study Design



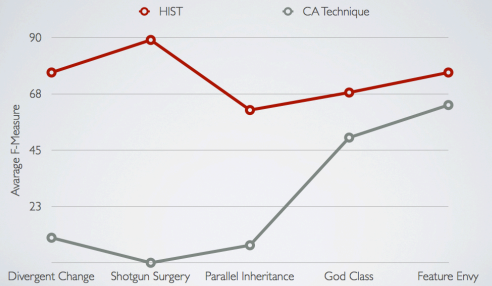
Identification of Move Method Refactoring Opportunities

Nikolaos Tsantalis, *Student Member, IEEE*, and Alexander Chatzigeorgiou, *Member, IEEE*

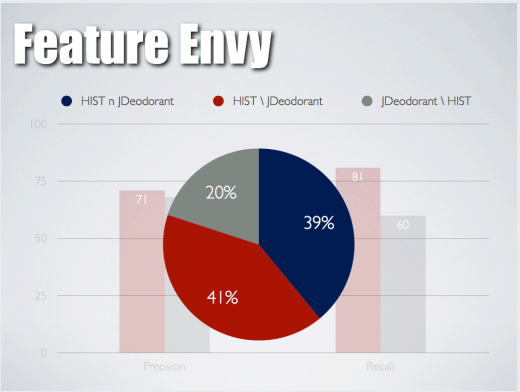
DECOR: A Method for the Specification and Detection of Code and Design Smells

Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur

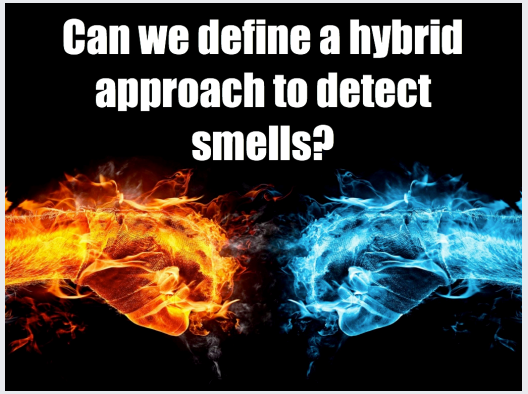
Results - Summary



Feature Envy



Can we define a hybrid approach to detect smells?



A class affected by a smell represents a problem if it does not change?