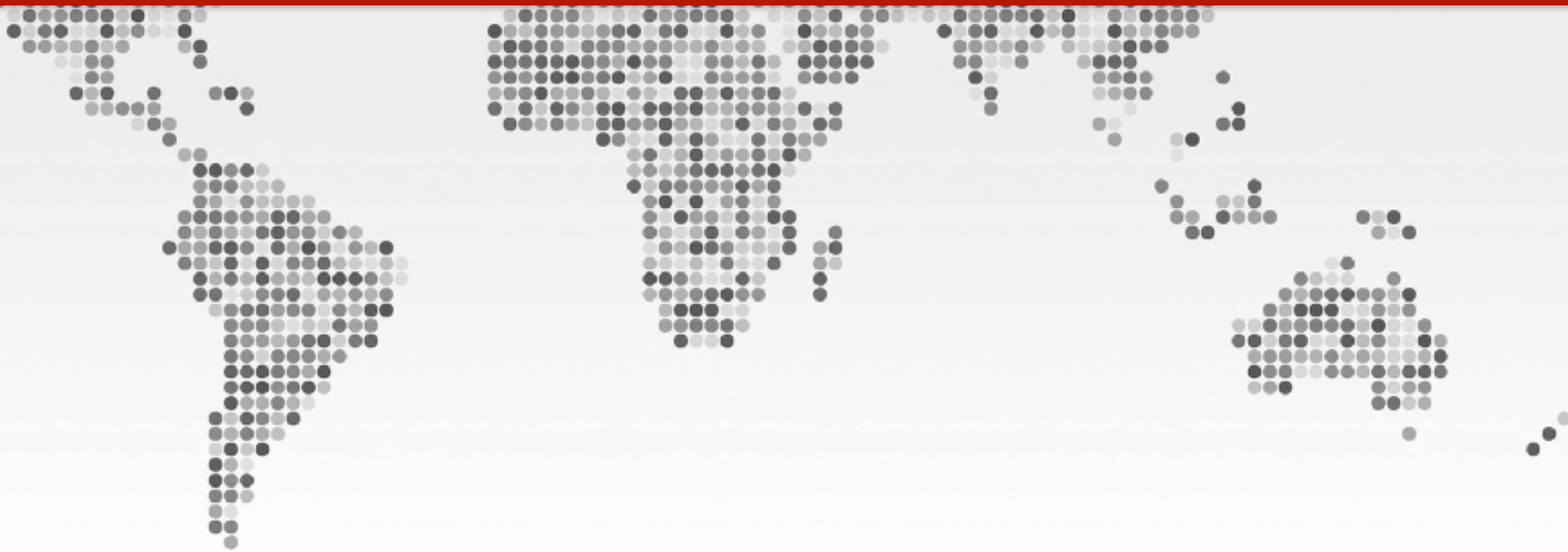


Code smells: detection & refactoring

Slides by Prof. Fabio Palomba





Fabiano Pecorelli
Associate Professor
Pegaso Digital University



fabiano.pecorelli@unipegaso.it



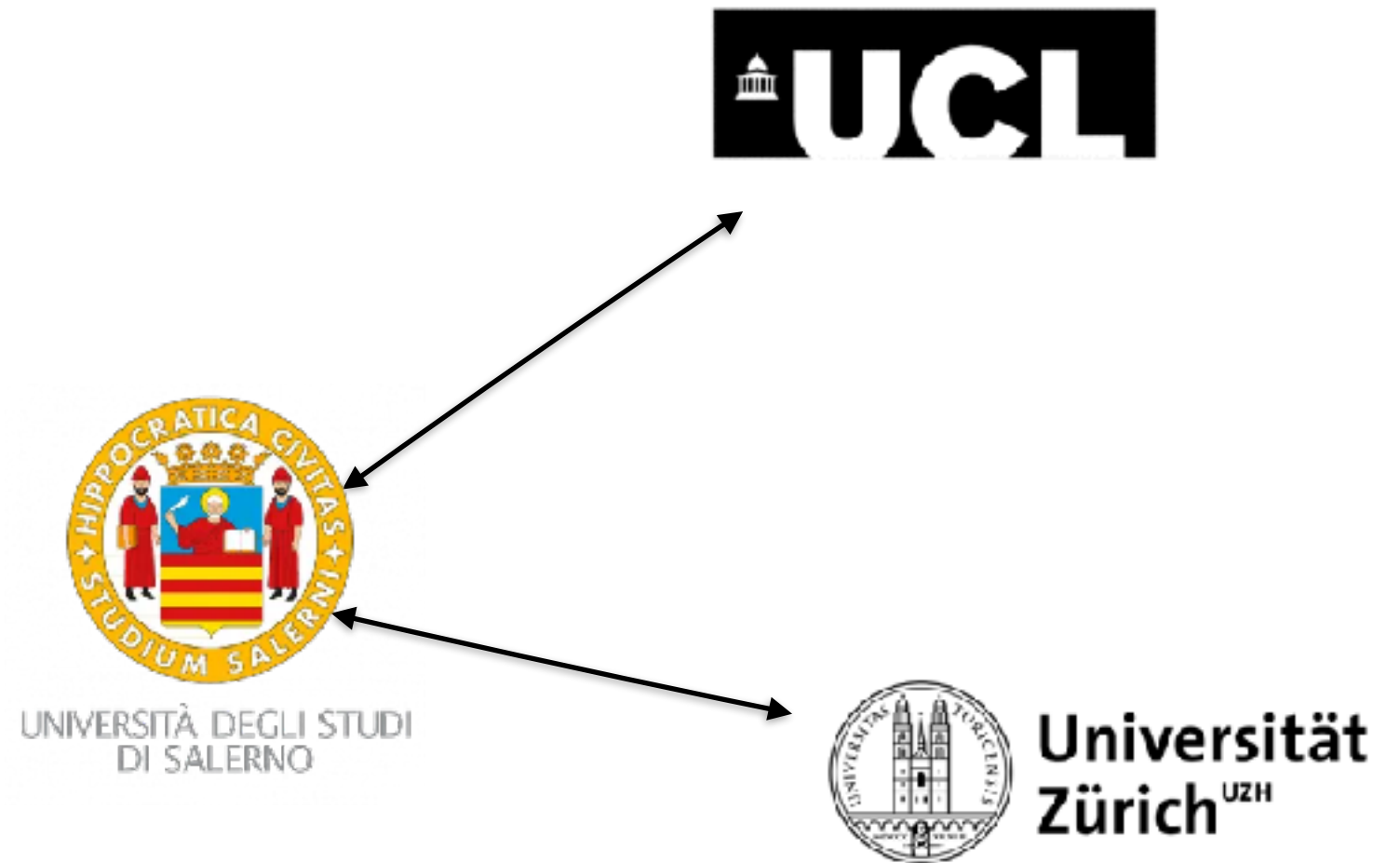
<https://fabiano-pecorelli.github.io>

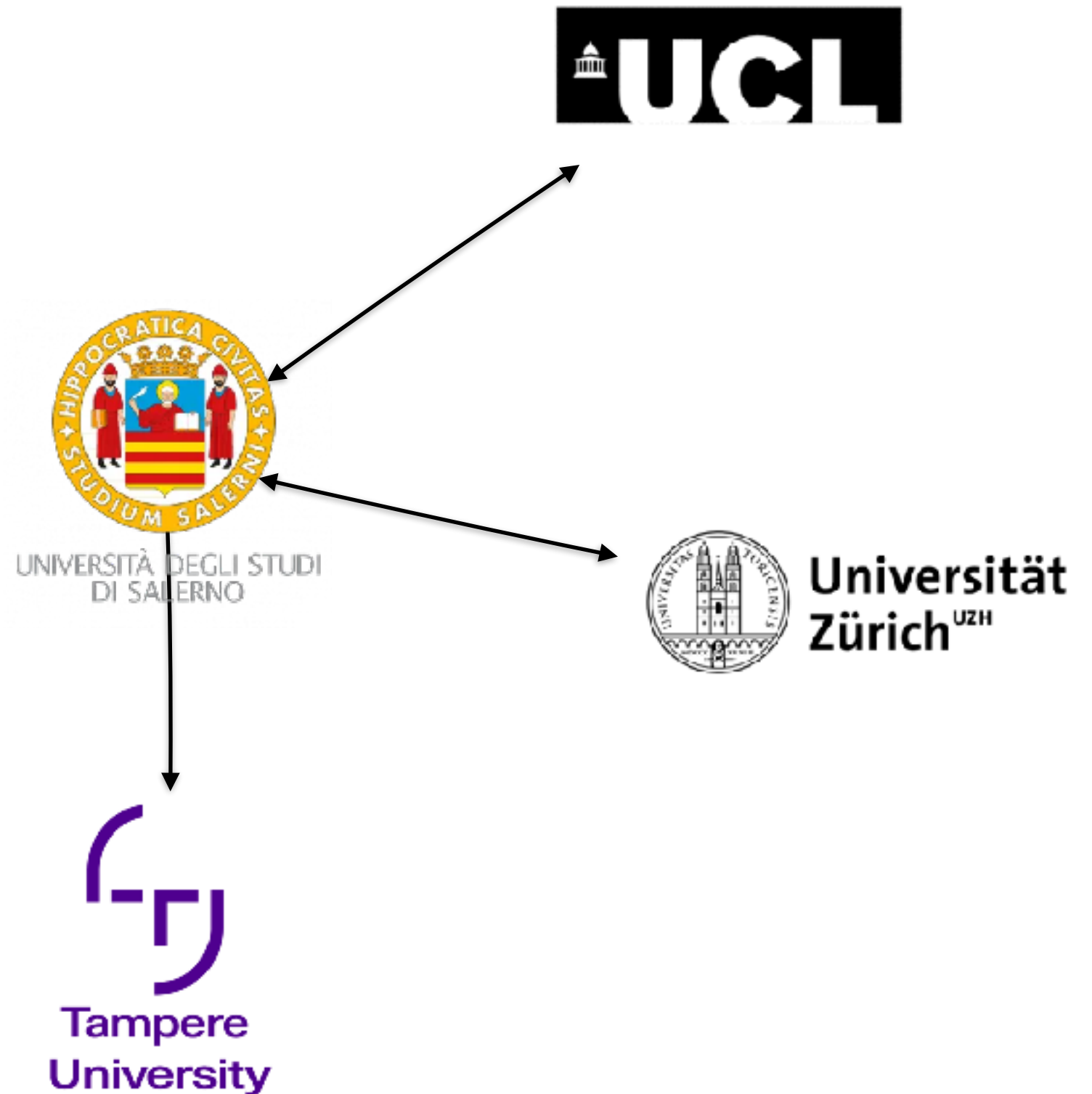


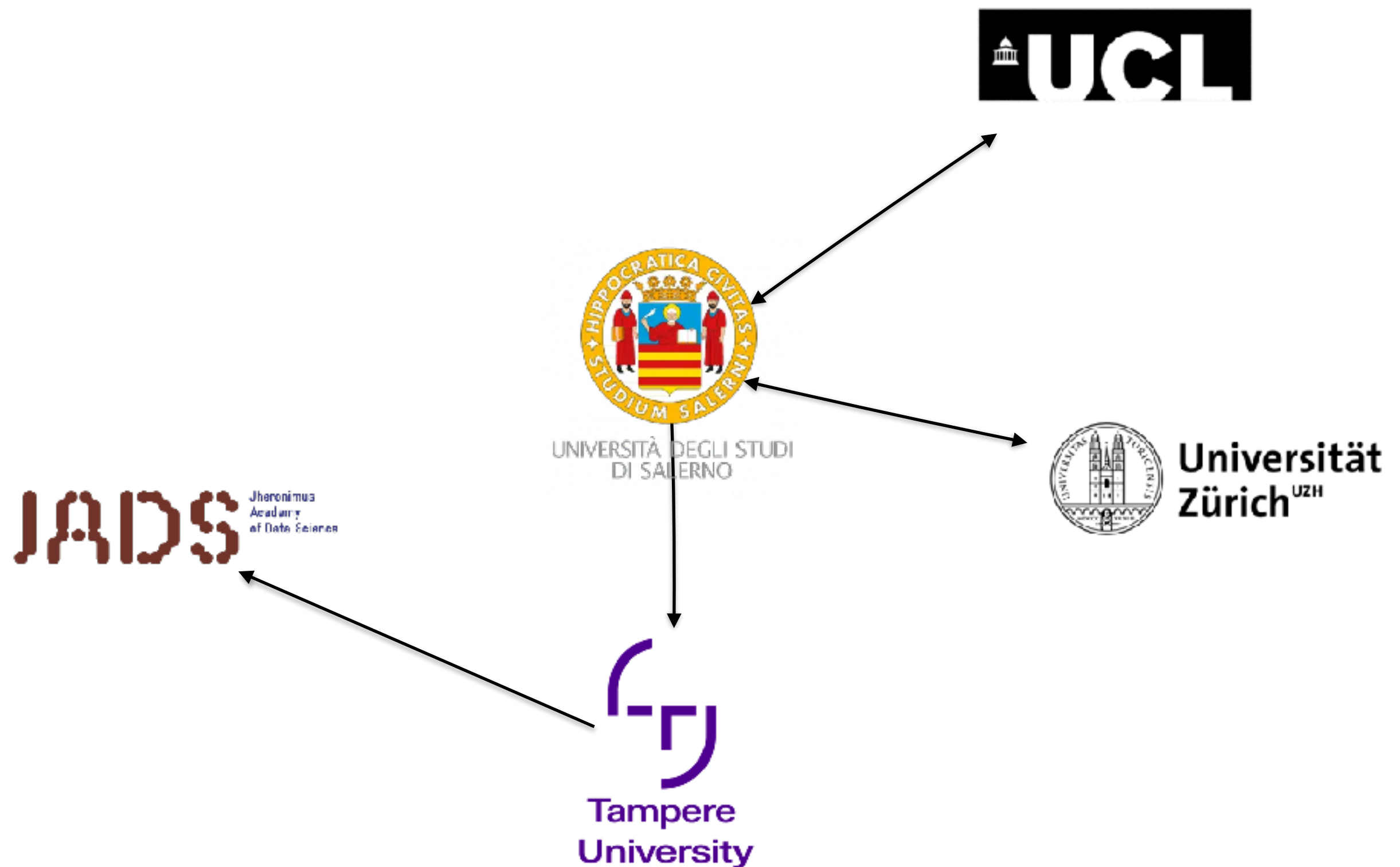
UNIVERSITÀ DEGLI STUDI
DI SALERNO

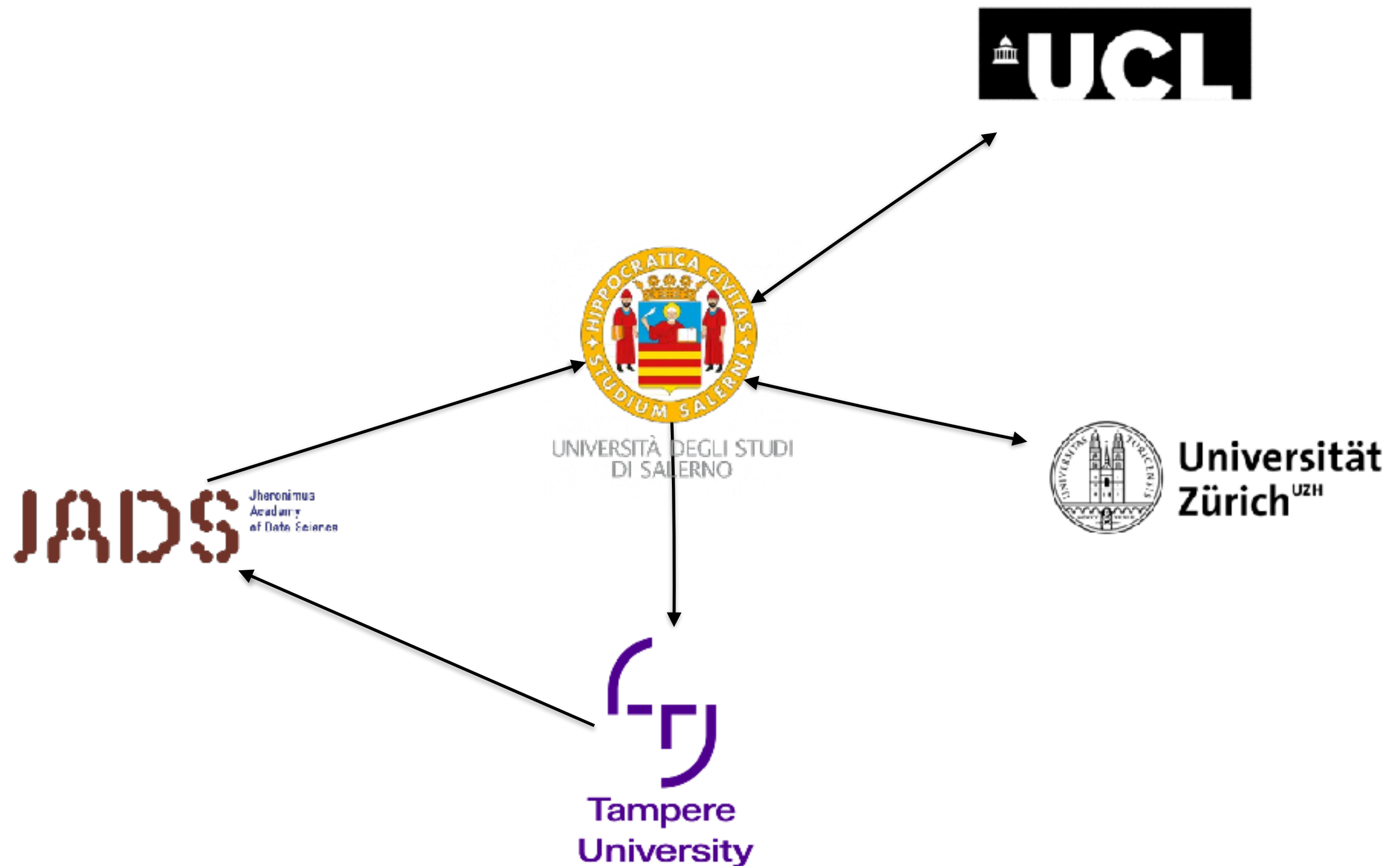


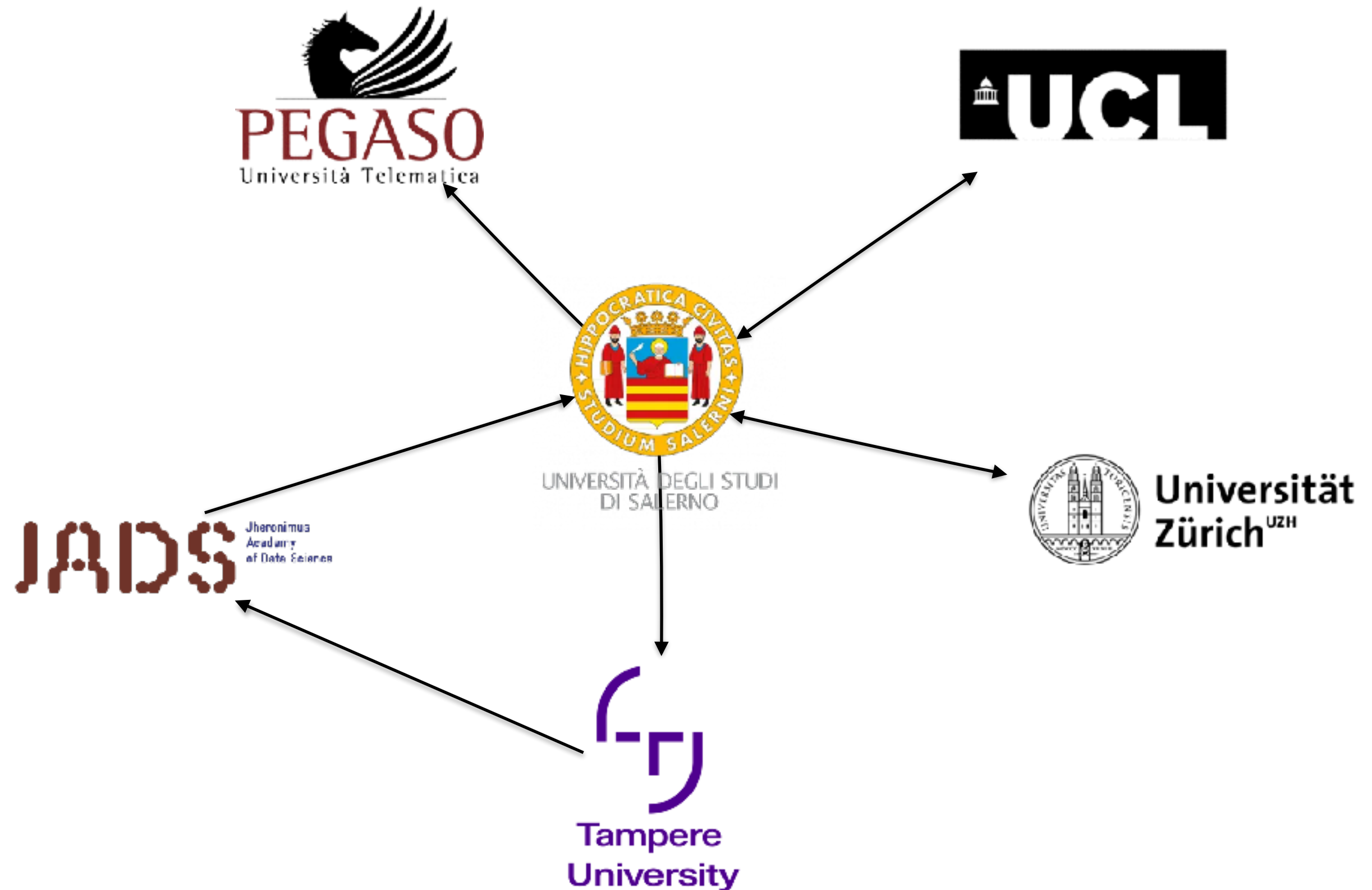
UNIVERSITÀ DEGLI STUDI
DI SALERNO











Introduction

During software maintenance and evolution, change is unavoidable (First Lehman's Law).
If a project does not change, **it will soon become useless**;

Introduction

During software maintenance and evolution, change is unavoidable (First Lehman's Law). If a project does not change, **it will soon become useless**;

Developers are, therefore, enforced to make changes that can (1) **enhance** the project, e.g., implementation of new features; (2) **adapt** it to new contexts, e.g., to new operating systems; (3) **fix** faults encountered by users or through software testing;

Introduction

During software maintenance and evolution, change is unavoidable (First Lehman's Law). If a project does not change, **it will soon become useless**;

Developers are, therefore, enforced to make changes that can (1) **enhance** the project, e.g., implementation of new features; (2) **adapt** it to new contexts, e.g., to new operating systems; (3) **fix** faults encountered by users or through software testing;

Unfortunately, due to time pressure, high workload, or other community-related constraints, **developers are not always able to produce high-quality software**.

Introduction

During software maintenance and evolution, change is unavoidable (First Lehman's Law). If a project does not change, **it will soon become useless**;

Developers are, therefore, enforced to make changes that can (1) **enhance** the project, e.g., implementation of new features; (2) **adapt** it to new contexts, e.g., to new operating systems; (3) **fix** faults encountered by users or through software testing;

Unfortunately, due to time pressure, high workload, or other community-related constraints, **developers are not always able to produce high-quality software**.

They are sometimes required to decide on whether to deliver a change (e.g., a new functionality) or postpone it. Often, **this is not a real decision: they must deliver in the shortest time possible**.

Introduction

During software maintenance and evolution, change is unavoidable (First Lehman's Law). If a project does not change, **it will soon become useless**;

Developers are, therefore, enforced to make changes that can (1) **enhance** the project, e.g., implementation of new features; (2) **adapt** it to new contexts, e.g., to new operating systems; (3) **fix** faults encountered by users or through software testing;

Unfortunately, due to time pressure, high workload, or other community-related constraints, **developers are not always able to produce high-quality software**.

They are sometimes required to decide on whether to deliver a change (e.g., a new functionality) or postpone it. Often, **this is not a real decision: they must deliver in the shortest time possible**.

As a consequence, **they could be enforced to set aside good implementation principles** and apply modifications that have the effect of drifting away the original design, leading to the introduction of the so-called **technical debt**.

Bad Code Smells

Introduction

During software maintenance and evolution, change is unavoidable (First Lehman's Law). If a project does not change, **it will soon become useless**;

Developers are, therefore, enforced to make changes that can (1) **enhance** the project, e.g., implementation of new features; (2) **adapt** it to new contexts, e.g., to new operating systems; (3) **fix** faults encountered by users or through software testing;

Unfortunately, due to time pressure, high workload, or other community-related constraints, **developers are not always able to produce high-quality software**.

They are sometimes required to decide on whether to deliver a change (e.g., a new functionality) or postpone it. Often, **this is not a real decision: they must deliver in the shortest time possible**.

As a consequence, **they could be enforced to set aside good implementation principles** and apply modifications that have the effect of drifting away the original design, leading to the introduction of the so-called **technical debt**.

Informally, technical debt represents *“not quite right code that we postpone to make it right”*. In other words, it is a metaphor that explains the **trade-offs between delivering the most appropriate but still immature product, in the shortest time possible**.

Bad Code Smells - Definition

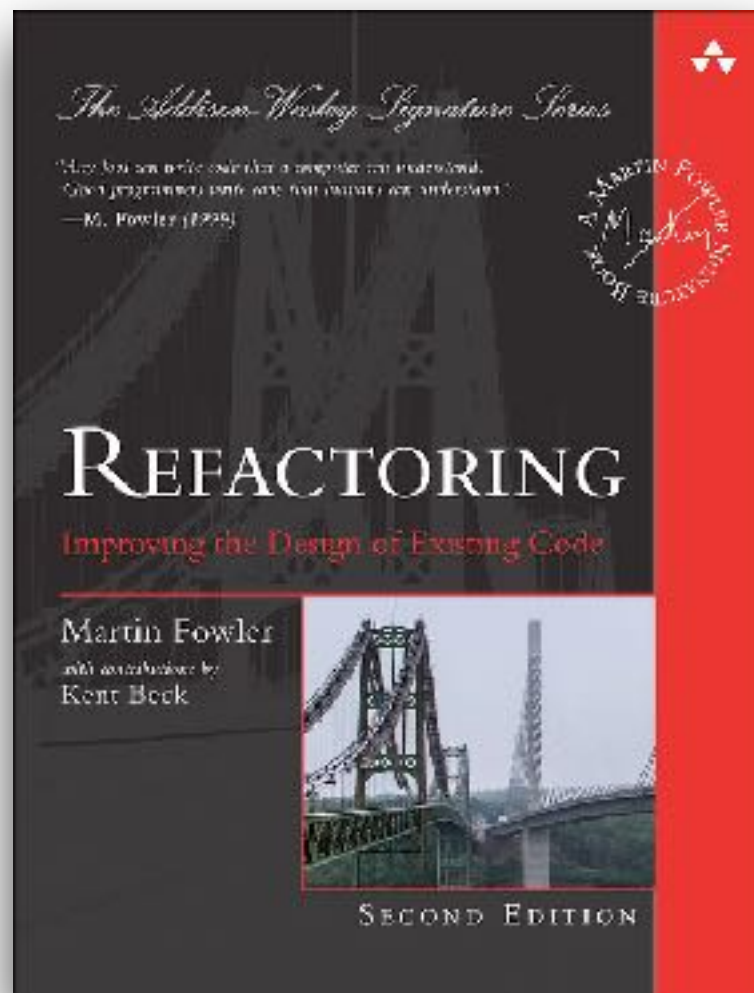
Bad code smells (a.k.a., code smells or simply smells) are one of the most relevant forms of technical debt. They refer to **wrong design or implementation choices** applied by developers when performing maintenance operations on a software system.

Bad Code Smells

Bad Code Smells - Definition

Bad code smells (a.k.a., code smells or simply smells) are one of the most relevant forms of technical debt. They refer to **wrong design or implementation choices** applied by developers when performing maintenance operations on a software system.

They have been originally introduced by Martin Fowler in his well-known book named: **“Refactoring: Improving the Design of Existing Code”** (preface of Kent Beck).

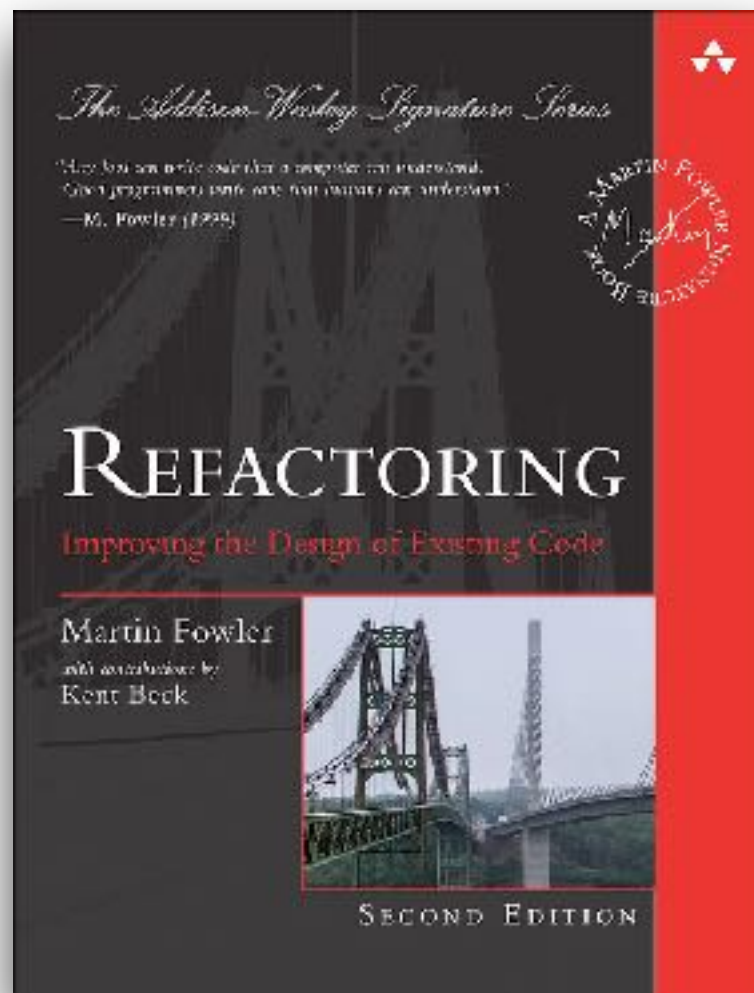


Bad Code Smells

Bad Code Smells - Definition

Bad code smells (a.k.a., code smells or simply smells) are one of the most relevant forms of technical debt. They refer to **wrong design or implementation choices** applied by developers when performing maintenance operations on a software system.

They have been originally introduced by Martin Fowler in his well-known book named: **“Refactoring: Improving the Design of Existing Code”** (preface of Kent Beck).



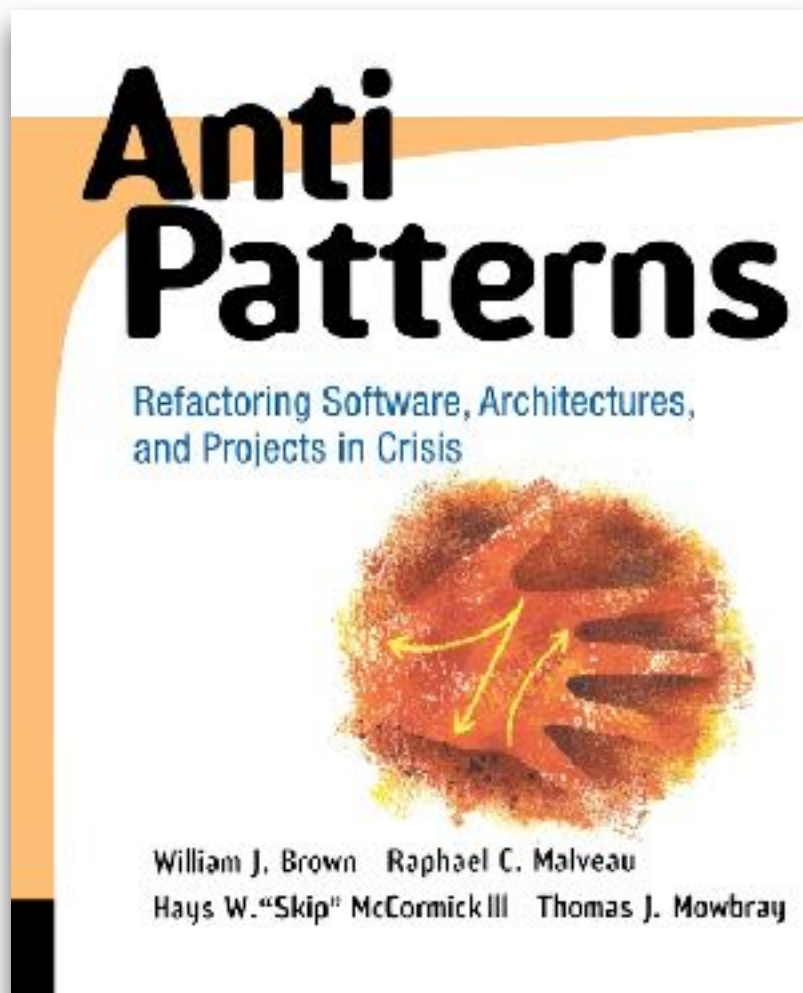
The book describes **more than 20 different types of code smells**, which have to do with violations to all Object-Oriented principles, i.e., cohesion, coupling, complexity, and so on.

Bad Code Smells

Bad Code Smells - Definition

Bad code smells (a.k.a., code smells or simply smells) are one of the most relevant forms of technical debt. They refer to **wrong design or implementation choices** applied by developers when performing maintenance operations on a software system.

They have been originally introduced by Martin Fowler in his well-known book named: **“Refactoring: Improving the Design of Existing Code”** (preface of Kent Beck).



The book describes **more than 20 different types of code smells**, which have to do with violations to all Object-Oriented principles, i.e., cohesion, coupling, complexity, and so on.

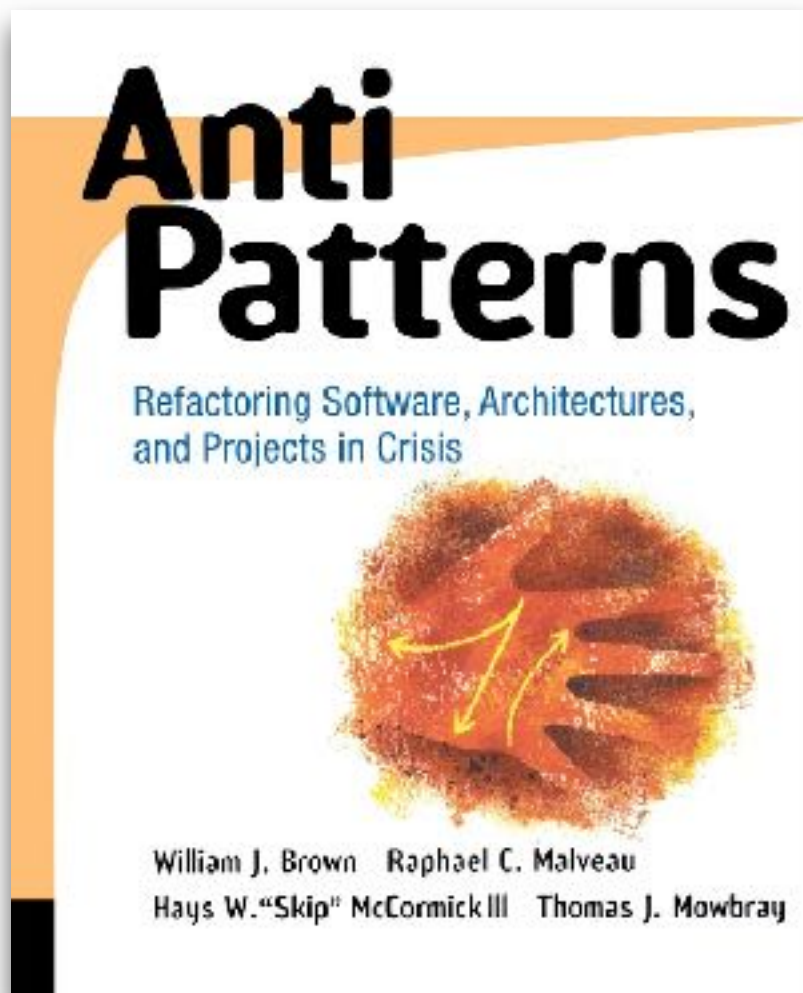
Brown et al., in the same years, defined the concept of **anti-patterns**, i.e., bad programming practices applied in the source code.

Bad Code Smells

Bad Code Smells - Definition

Bad code smells (a.k.a., code smells or simply smells) are one of the most relevant forms of technical debt. They refer to **wrong design or implementation choices** applied by developers when performing maintenance operations on a software system.

They have been originally introduced by Martin Fowler in his well-known book named: **“Refactoring: Improving the Design of Existing Code”** (preface of Kent Beck).



The book describes **more than 20 different types of code smells**, which have to do with violations to all Object-Oriented principles, i.e., cohesion, coupling, complexity, and so on.

Brown et al., in the same years, defined the concept of **anti-patterns**, i.e., bad programming practices applied in the source code.

Code smells are NOT anti-patterns. There is a relation, of course, but code smells are **potential** design problems that can eventually lead to the emergence of bad code.

Bad Code Smells - Refining the Definition

Bad code smells (a.k.a., code smells or simply smells) represent **symptoms of the presence of poor design and/or implementation choices** that may lead to additional unforeseen software development costs.



Bad Code Smells

Bad Code Smells - Refining the Definition

Bad code smells (a.k.a., code smells or simply smells) represent **symptoms of the presence of poor design and/or implementation choices** that may lead to additional unforeseen software development costs.

Bad Code Smells - Some Examples (More definitions when needed)

God Class (Blob). A large class implementing several responsibilities, having a large quantity of attributes, methods, and dependencies with data classes (i.e., classes only having getters and setters). This smell impacts on both cohesion and coupling.

Bad Code Smells

Bad Code Smells - Refining the Definition

Bad code smells (a.k.a., code smells or simply smells) represent **symptoms of the presence of poor design and/or implementation choices** that may lead to additional unforeseen software development costs.

Bad Code Smells - Some Examples (More definitions when needed)

God Class (Blob). A large class implementing several responsibilities, having a large quantity of attributes, methods, and dependencies with data classes (i.e., classes only having getters and setters). This smell impacts on both cohesion and coupling.

Swiss Army Knife. A class offering a large number of services, for instance implementing a large amount of interfaces. Typical examples are utility classes, which are built not taking into account cohesion of source code.

Bad Code Smells

Bad Code Smells - Refining the Definition

Bad code smells (a.k.a., code smells or simply smells) represent **symptoms of the presence of poor design and/or implementation choices** that may lead to additional unforeseen software development costs.

Bad Code Smells - Some Examples (More definitions when needed)

God Class (Blob). A large class implementing several responsibilities, having a large quantity of attributes, methods, and dependencies with data classes (i.e., classes only having getters and setters). This smell impacts on both cohesion and coupling.

Swiss Army Knife. A class offering a large number of services, for instance implementing a large amount of interfaces. Typical examples are utility classes, which are built not taking into account cohesion of source code.

Divergent Change. This code smell appears when one class is commonly changed in different ways for different reasons. It is clearly represented by low cohesion and typically low similarity among the methods implemented in the class.

Bad Code Smells

Bad Code Smells - Refining the Definition

Bad code smells (a.k.a., code smells or simply smells) represent **symptoms of the presence of poor design and/or implementation choices** that may lead to additional unforeseen software development costs.

Bad Code Smells - Some Examples (More definitions when needed)

God Class (Blob). A large class implementing several responsibilities, having a large quantity of attributes, methods, and dependencies with data classes (i.e., classes only having getters and setters). This smell impacts on both cohesion and coupling.

Swiss Army Knife. A class offering a large number of services, for instance implementing a large amount of interfaces. Typical examples are utility classes, which are built not taking into account cohesion of source code.

Divergent Change. This code smell appears when one class is commonly changed in different ways for different reasons. It is clearly represented by low cohesion and typically low similarity among the methods implemented in the class.

Duplicated Code. This smell occurs when the same code structure (block, method) is implemented in more than one place, hence making more complex maintenance activities because of change propagation.

Bad Code Smells - Diffuseness

A recent study has assessed the diffuseness of code smells, taking into account **a dataset composed of 395 open-source systems and 13 code smell types.**

Bad Code Smells

Bad Code Smells - Diffuseness

A recent study has assessed the diffuseness of code smells, taking into account **a dataset composed of 395 open-source systems and 13 code smell types.**

Some more definitions needed

Spaghetti Code. Classes having little structure, declaring at least one long methods with no parameters, and using instance attributes. Their names and their methods names may suggest a procedural programming style.

Bad Code Smells

Bad Code Smells - Diffuseness

A recent study has assessed the diffuseness of code smells, taking into account **a dataset composed of 395 open-source systems and 13 code smell types.**

Some more definitions needed

Spaghetti Code. Classes having little structure, declaring at least one long methods with no parameters, and using instance attributes. Their names and their methods names may suggest a procedural programming style.

Long Method. This is a kind of method-level Blob. Methods affected by this smell are large in size and implement more than one functionality.

Bad Code Smells

Bad Code Smells - Diffuseness

A recent study has assessed the diffuseness of code smells, taking into account **a dataset composed of 395 open-source systems and 13 code smell types.**

Some more definitions needed

Spaghetti Code. Classes having little structure, declaring at least one long methods with no parameters, and using instance attributes. Their names and their methods names may suggest a procedural programming style.

Long Method. This is a kind of method-level Blob. Methods affected by this smell are large in size and implement more than one functionality.

Message Chains. You see message chains when a client asks one object for another object, which the client then asks for yet another object, which the client then asks for yet another another object, and so on.

Bad Code Smells

Bad Code Smells - Diffuseness

A recent study has assessed the diffuseness of code smells, taking into account **a dataset composed of 395 open-source systems and 13 code smell types.**

Some more definitions needed

Spaghetti Code. Classes having little structure, declaring at least one long methods with no parameters, and using instance attributes. Their names and their methods names may suggest a procedural programming style.

Long Method. This is a kind of method-level Blob. Methods affected by this smell are large in size and implement more than one functionality.

Message Chains. You see message chains when a client asks one object for another object, which the client then asks for yet another object, which the client then asks for yet another another object, and so on.

Inappropriate Intimacy. As the name suggests, this smell appears when two classes are highly coupled and create a kind of inter-dependence between them. This possibly makes harder any evolution activity made on these classes.

Bad Code Smells

Bad Code Smells - Diffuseness

A recent study has assessed the diffuseness of code smells, taking into account **a dataset composed of 395 open-source systems and 13 code smell types.**

Some more definitions needed

Spaghetti Code. Classes having little structure, declaring at least one long methods with no parameters, and using instance attributes. Their names and their methods names may suggest a procedural programming style.

Long Method. This is a kind of method-level Blob. Methods affected by this smell are large in size and implement more than one functionality.

Message Chains. You see message chains when a client asks one object for another object, which the client then asks for yet another object, which the client then asks for yet another another object, and so on.

Inappropriate Intimacy. As the name suggests, this smell appears when two classes are highly coupled and create a kind of inter-dependence between them. This possibly makes harder any evolution activity made on these classes.

Feature Envy. This represents a method that is “more interested” to the functionalities implemented in a class different from the one it is actually in.

Bad Code Smells - Diffuseness

A recent study has assessed the diffuseness of code smells, taking into account **a dataset composed of 395 open-source systems and 13 code smell types**.

- Most of the considered code smells **are highly diffused** in all systems: almost 93% of projects exhibit at least one instance of code smell.

Bad Code Smells - Diffuseness

A recent study has assessed the diffuseness of code smells, taking into account **a dataset composed of 395 open-source systems and 13 code smell types.**

- Most of the considered code smells **are highly diffused** in all systems: almost 93% of projects exhibit at least one instance of code smell.
- **84% of the projects contain at least one Long Method** - this represents the most diffused code smell in practice.

Bad Code Smells - Diffuseness

A recent study has assessed the diffuseness of code smells, taking into account **a dataset composed of 395 open-source systems and 13 code smell types.**

- Most of the considered code smells **are highly diffused** in all systems: almost 93% of projects exhibit at least one instance of code smell.
- **84% of the projects contain at least one Long Method** - this represents the most diffused code smell in practice.
- **83% of the projects contain at least one Spaghetti Code** - more in general, all size-related code smells tend to be more diffused than others.

Bad Code Smells - Diffuseness

A recent study has assessed the diffuseness of code smells, taking into account **a dataset composed of 395 open-source systems and 13 code smell types.**

- Most of the considered code smells **are highly diffused** in all systems: almost 93% of projects exhibit at least one instance of code smell.
- **84% of the projects contain at least one Long Method** - this represents the most diffused code smell in practice.
- **83% of the projects contain at least one Spaghetti Code** - more in general, all size-related code smells tend to be more diffused than others.
- **Coupling-related code smells are, instead, poorly diffused:** on average, Inappropriate Intimacy, Feature Envy, and others affect only 8% of the considered projects.

Bad Code Smells - Diffuseness

A recent study has assessed the diffuseness of code smells, taking into account **a dataset composed of 395 open-source systems and 13 code smell types.**

- Most of the considered code smells **are highly diffused** in all systems: almost 93% of projects exhibit at least one instance of code smell.
- **84% of the projects contain at least one Long Method** - this represents the most diffused code smell in practice.
- **83% of the projects contain at least one Spaghetti Code** - more in general, all size-related code smells tend to be more diffused than others.
- **Coupling-related code smells are, instead, poorly diffused:** on average, Inappropriate Intimacy, Feature Envy, and others affect only 8% of the considered projects.
- **13% of the projects contain a Message Chains:** this is the coupling-related smell more diffused among those considered.

Bad Code Smells - Diffuseness

A recent study has assessed the diffuseness of code smells, taking into account **a dataset composed of 395 open-source systems and 13 code smell types.**

- Most of the considered code smells **are highly diffused** in all systems: almost 93% of projects exhibit at least one instance of code smell.
- **84% of the projects contain at least one Long Method** - this represents the most diffused code smell in practice.
- **83% of the projects contain at least one Spaghetti Code** - more in general, all size-related code smells tend to be more diffused than others.
- **Coupling-related code smells are, instead, poorly diffused:** on average, Inappropriate Intimacy, Feature Envy, and others affect only 8% of the considered projects.
- **13% of the projects contain a Message Chains:** this is the coupling-related smell more diffused among those considered.
- **The diffuseness of code smells seems to be independent from the granularity of the design problem:** method- and class-level code smells are roughly equally distributed over the dataset.

Bad Code Smells - Impact on Source Code Maintainability

Code smells affect both **change- and fault-proneness of source code**. But, of course, different smells affect this aspect differently.

Bad Code Smells - Impact on Source Code Maintainability

Code smells affect both **change- and fault-proneness of source code**. But, of course, different smells affect this aspect differently.

Change-proneness. External quality of source code that indicates the extent to which a class changes across the versions of a software project. This can be computed by looking at the number of commits involving a class over the change history.

Bad Code Smells - Impact on Source Code Maintainability

Code smells affect both **change- and fault-proneness of source code**. But, of course, different smells affect this aspect differently.

Change-proneness. External quality of source code that indicates the extent to which a class changes across the versions of a software project. This can be computed by looking at the number of commits involving a class over the change history.

Fault-proneness. External quality of source code that indicates the extent to which a class is subject to defects. This can be computed by looking at the number of issues opened and involving a class over the change history.

Bad Code Smells

Bad Code Smells - Impact on Source Code Maintainability

Code smells affect both **change- and fault-proneness of source code**. But, of course, different smells affect this aspect differently.

Change-proneness. External quality of source code that indicates the extent to which a class changes across the versions of a software project. This can be computed by looking at the number of commits involving a class over the change history.

Fault-proneness. External quality of source code that indicates the extent to which a class is subject to defects. This can be computed by looking at the number of issues opened and involving a class over the change history.

Classes affected by code smells have a statistically significant higher change- and fault-proneness than non-smelly classes.

Bad Code Smells - Impact on Source Code Maintainability

Code smells affect both **change- and fault-proneness of source code**. But, of course, different smells affect this aspect differently.

Change-proneness. External quality of source code that indicates the extent to which a class changes across the versions of a software project. This can be computed by looking at the number of commits involving a class over the change history.

Fault-proneness. External quality of source code that indicates the extent to which a class is subject to defects. This can be computed by looking at the number of issues opened and involving a class over the change history.

Classes affected by code smells have a statistically significant higher change- and fault-proneness than non-smelly classes.

- The higher the number of smells in a class, i.e., code smell co-occurrence, **the higher the impact on maintainability.**

Bad Code Smells - Impact on Source Code Maintainability

Code smells affect both **change- and fault-proneness of source code**. But, of course, different smells affect this aspect differently.

Change-proneness. External quality of source code that indicates the extent to which a class changes across the versions of a software project. This can be computed by looking at the number of commits involving a class over the change history.

Fault-proneness. External quality of source code that indicates the extent to which a class is subject to defects. This can be computed by looking at the number of issues opened and involving a class over the change history.

Classes affected by code smells have a statistically significant higher change- and fault-proneness than non-smelly classes.

- The higher the number of smells in a class, i.e., code smell co-occurrence, **the higher the impact on maintainability**.
- Most diffused smells **have a higher impact than the non-diffused ones**, indicating that some code smells are more harmful than others.

Bad Code Smells - Impact on Source Code Maintainability

Code smells affect both **change- and fault-proneness of source code**. But, of course, different smells affect this aspect differently.

Change-proneness. External quality of source code that indicates the extent to which a class changes across the versions of a software project. This can be computed by looking at the number of commits involving a class over the change history.

Fault-proneness. External quality of source code that indicates the extent to which a class is subject to defects. This can be computed by looking at the number of issues opened and involving a class over the change history.

Classes affected by code smells have a statistically significant higher change- and fault-proneness than non-smelly classes.

- The higher the number of smells in a class, i.e., code smell co-occurrence, **the higher the impact on maintainability**.
- Most diffused smells **have a higher impact than the non-diffused ones**, indicating that some code smells are more harmful than others.
- The refactoring of code smells reduces the change- and defect-proneness of classes.

Bad Code Smells - Impact on Source Code Maintainability

Code smells affect **maintenance costs**. Particularly, code smells affecting the complexity of source code (e.g., Blob) are those producing the highest loss.

Bad Code Smells

Bad Code Smells - Impact on Source Code Maintainability

Code smells affect **maintenance costs**. Particularly, code smells affecting the complexity of source code (e.g., Blob) are those producing the highest loss.

Some more definitions needed

Complex Class. Classes exhibiting complex methods (in terms of some metrics like McCabe cyclomatic complexity), typically having a large amount of lines of code.

Bad Code Smells

Bad Code Smells - Impact on Source Code Maintainability

Code smells affect **maintenance costs**. Particularly, code smells affecting the complexity of source code (e.g., Blob) are those producing the highest loss.

Some more definitions needed

Complex Class. Classes exhibiting complex methods (in terms of some metrics like McCabe cyclomatic complexity), typically having a large amount of lines of code.

Shotgun Surgery. This code smell appears when every time you make a kind of change, you have to make a lot of little changes to a lot of different classes.

Bad Code Smells

Bad Code Smells - Impact on Source Code Maintainability

Code smells affect **maintenance costs**. Particularly, code smells affecting the complexity of source code (e.g., Blob) are those producing the highest loss.

Some more definitions needed

Complex Class. Classes exhibiting complex methods (in terms of some metrics like McCabe cyclomatic complexity), typically having a large amount of lines of code.

Shotgun Surgery. This code smell appears when every time you make a kind of change, you have to make a lot of little changes to a lot of different classes.

Refused Bequest. Sub-classes inherit from a parent class but intentionally omits or doesn't use many of the inherited methods or properties.

Bad Code Smells - Impact on Source Code Maintainability

Code smells affect **maintenance costs**. Particularly, code smells affecting the complexity of source code (e.g., Blob) are those producing the highest loss.

Some more definitions needed

Complex Class. Classes exhibiting complex methods (in terms of some metrics like McCabe cyclomatic complexity), typically having a large amount of lines of code.

Shotgun Surgery. This code smell appears when every time you make a kind of change, you have to make a lot of little changes to a lot of different classes.

Refused Bequest. Sub-classes inherit from a parent class but intentionally omits or doesn't use many of the inherited methods or properties.

- Classes affected by Complex Class or Shotgun Surgery are those **affecting the most both maintenance costs and effort**.

Bad Code Smells - Impact on Source Code Maintainability

Code smells affect **maintenance costs**. Particularly, code smells affecting the complexity of source code (e.g., Blob) are those producing the highest loss.

Some more definitions needed

Complex Class. Classes exhibiting complex methods (in terms of some metrics like McCabe cyclomatic complexity), typically having a large amount of lines of code.

Shotgun Surgery. This code smell appears when every time you make a kind of change, you have to make a lot of little changes to a lot of different classes.

Refused Bequest. Sub-classes inherit from a parent class but intentionally omits or doesn't use many of the inherited methods or properties.

- Classes affected by Complex Class or Shotgun Surgery are those **affecting the most both maintenance costs and effort**.
- On the contrary, Refused Bequest does not significantly relate to maintenance effort and costs; likely, **this may depend on the intensity of the design problem**.

Bad Code Smells

Bad Code Smells - Is there a connection between actions on code smells and their perception?

The definition of code smells has been given in a **natural language form**. As such, it may sometimes be misunderstood and/or subjectively interpreted by developers. In any case, the way developers act on them **depends on how they are aware of smells**.

Bad Code Smells - Is there a connection between actions on code smells and their perception?

The definition of code smells has been given in a **natural language form**. As such, it may sometimes be misunderstood and/or subjectively interpreted by developers. In any case, the way developers act on them **depends on how they are aware of smells**.

- Some studies have assessed this aspect, either through semi-structured interviews with developers or controlled experiments.

Bad Code Smells - Is there a connection between actions on code smells and their perception?

The definition of code smells has been given in a **natural language form**. As such, it may sometimes be misunderstood and/or subjectively interpreted by developers. In any case, the way developers act on them **depends on how they are aware of smells**.

- Some studies have assessed this aspect, either through semi-structured interviews with developers or controlled experiments.
- Key findings reported that most of the code smells **are not perceived as problematic** by developers, e.g., the Class Data Should Be Private.

Bad Code Smells - Is there a connection between actions on code smells and their perception?

The definition of code smells has been given in a **natural language form**. As such, it may sometimes be misunderstood and/or subjectively interpreted by developers. In any case, the way developers act on them **depends on how they are aware of smells**.

- Some studies have assessed this aspect, either through semi-structured interviews with developers or controlled experiments.
- Key findings reported that most of the code smells **are not perceived as problematic** by developers, e.g., the Class Data Should Be Private.
- Developers **tend to recognize and correctly diagnose code smells related to size and complexity**, Blob or Complex Class.

Bad Code Smells - Is there a connection between actions on code smells and their perception?

The definition of code smells has been given in a **natural language form**. As such, it may sometimes be misunderstood and/or subjectively interpreted by developers. In any case, the way developers act on them **depends on how they are aware of smells**.

- Some studies have assessed this aspect, either through semi-structured interviews with developers or controlled experiments.
- Key findings reported that most of the code smells **are not perceived as problematic** by developers, e.g., the Class Data Should Be Private.
- Developers **tend to recognize and correctly diagnose code smells related to size and complexity**, Blob or Complex Class.
- In the cases of Refused Bequest and Feature Envy, the perception **strongly depends on the severity of the design problems**.

Bad Code Smells - Is there a connection between actions on code smells and their perception?

The definition of code smells has been given in a **natural language form**. As such, it may sometimes be misunderstood and/or subjectively interpreted by developers. In any case, the way developers act on them **depends on how they are aware of smells**.

- Some studies have assessed this aspect, either through semi-structured interviews with developers or controlled experiments.
- Key findings reported that most of the code smells **are not perceived as problematic** by developers, e.g., the Class Data Should Be Private.
- Developers **tend to recognize and correctly diagnose code smells related to size and complexity**, Blob or Complex Class.
- In the cases of Refused Bequest and Feature Envy, the perception **strongly depends on the severity of the design problems**.
- To sum up, the way developers act on code smell instances depends on various factors, **including their ability to diagnose them**.

Bad Code Smells - Is there a connection between actions on code smells and their perception?

The definition of code smells has been given in a **natural language form**. As such, it may sometimes be misunderstood and/or subjectively interpreted by developers. In any case, the way developers act on them **depends on how they are aware of smells**.

- Some studies have assessed this aspect, either through semi-structured interviews with developers or controlled experiments.
- Key findings reported that most of the code smells **are not perceived as problematic** by developers, e.g., the Class Data Should Be Private.
- Developers **tend to recognize and correctly diagnose code smells related to size and complexity**, Blob or Complex Class.
- In the cases of Refused Bequest and Feature Envy, the perception **strongly depends on the severity of the design problems**.
- To sum up, the way developers act on code smell instances depends on various factors, **including their ability to diagnose them**.
- As such, techniques to detect and refactor code smells **are necessary but not sufficient**, as they should integrate awareness methods.

Bad Code Smells - Relation with Refactoring

Refactoring is the process of **changing the internal structure of the source code without altering its external behavior.**

Bad Code Smells - Relation with Refactoring

Refactoring is the process of **changing the internal structure of the source code without altering its external behavior.**

Refactoring is the action required to remove a code smell. More refactoring operations can be applied for the removal of multiple code smells - so **there is a 1 to N relationship between code smells and refactoring.**

Refactoring - Decomposing Methods

Extract Method. In cases where a method presents a fragment of code that can be grouped together, turn it into a method with a name that explains its purpose.

Refactoring - Decomposing Methods

Extract Method. In cases where a method presents a fragment of code that can be grouped together, turn it into a method with a name that explains its purpose.

```
void printOwing() {  
    printBanner();  
  
    // Print details.  
    System.out.println("name: " + name);  
    System.out.println("amount: " + getOutstanding());  
}
```

<= The method prints a banner **AND** prints details.

Refactoring - Decomposing Methods

Extract Method. In cases where a method presents a fragment of code that can be grouped together, turn it into a method with a name that explains its purpose.

```
void printOwing() {  
    printBanner();  
  
    // Print details.  
    System.out.println("name: " + name);  
    System.out.println("amount: " + getOutstanding());  
}
```

<= The method prints a banner **AND** prints details.

This method may be seen as a code smell, i.e., Long Method, since it implements more than one function.

Refactoring - Decomposing Methods

Extract Method. In cases where a method presents a fragment of code that can be grouped together, turn it into a method with a name that explains its purpose.

```
void printOwing() {  
    printBanner();  
  
    // Print details.  
    System.out.println("name: " + name);  
    System.out.println("amount: " + getOutstanding());  
}
```

<= The method prints a banner **AND** prints details.

This method may be seen as a code smell, i.e., Long Method, since it implements more than one function.

```
void printOwing() {  
    printBanner();  
    printDetails(getOutstanding());  
}  
  
void printDetails(double outstanding) {  
    System.out.println("name: " + name);  
    System.out.println("amount: " + outstanding);  
}
```

An Extract Method refactoring allows you to extract the “exceeding” portion of code and create a new method.

Refactoring - Decomposing Methods

Extract Method. In cases where a method presents a fragment of code that can be grouped together, turn it into a method with a name that explains its purpose.

```
void printOwing() {  
    printBanner();  
  
    // Print details.  
    System.out.println("name: " + name);  
    System.out.println("amount: " + getOutstanding());  
}
```

<= The method prints a banner **AND** prints details.

This method may be seen as a code smell, i.e., Long Method, since it implements more than one function.

```
void printOwing() {  
    printBanner();  
    printDetails(getOutstanding());  
}  
  
void printDetails(double outstanding) {  
    System.out.println("name: " + name);  
    System.out.println("amount: " + outstanding);  
}
```

An Extract Method refactoring allows you to extract the “exceeding” portion of code and create a new method.

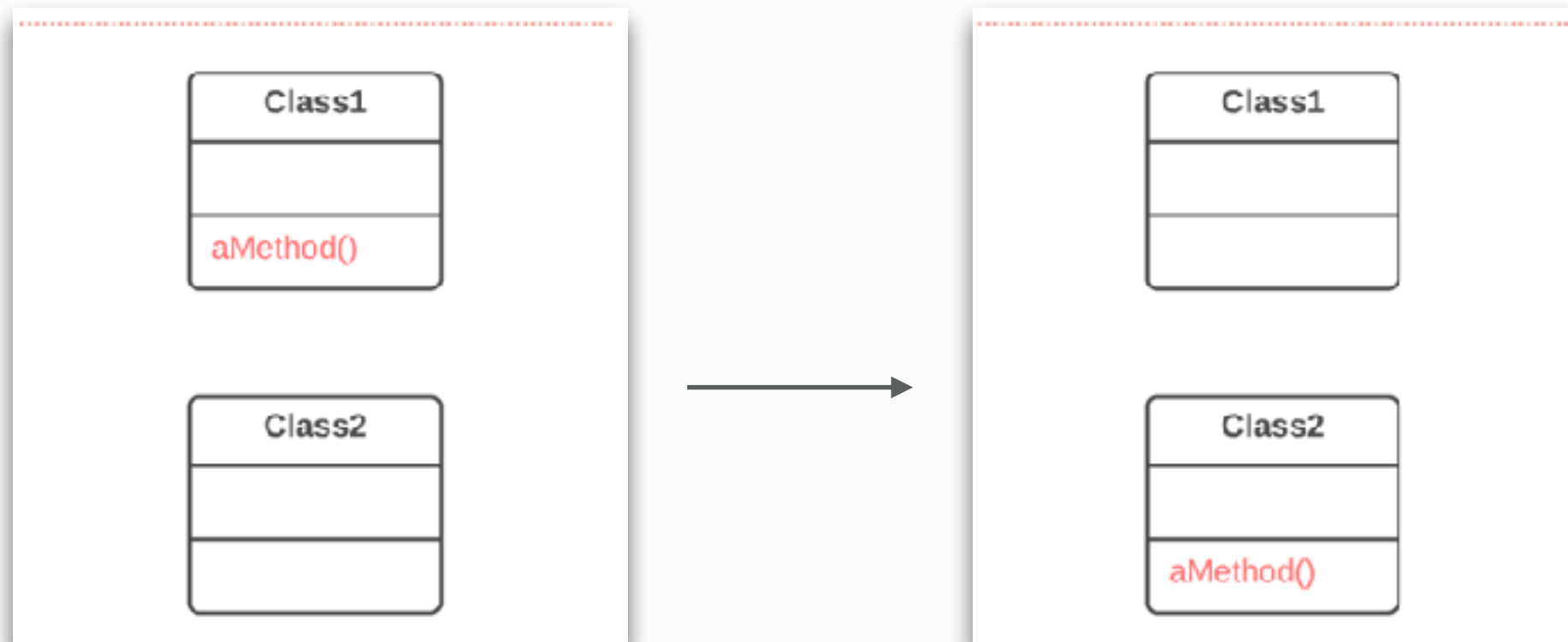
<= In the example, the details are printed in another method.

Refactoring - Moving Features Between Objects

Move Field/Method/Class. This refactoring family consists of moving a source code element toward another one in cases where it is more interested in the functionalities of the other element with respect to the one it is actually in.

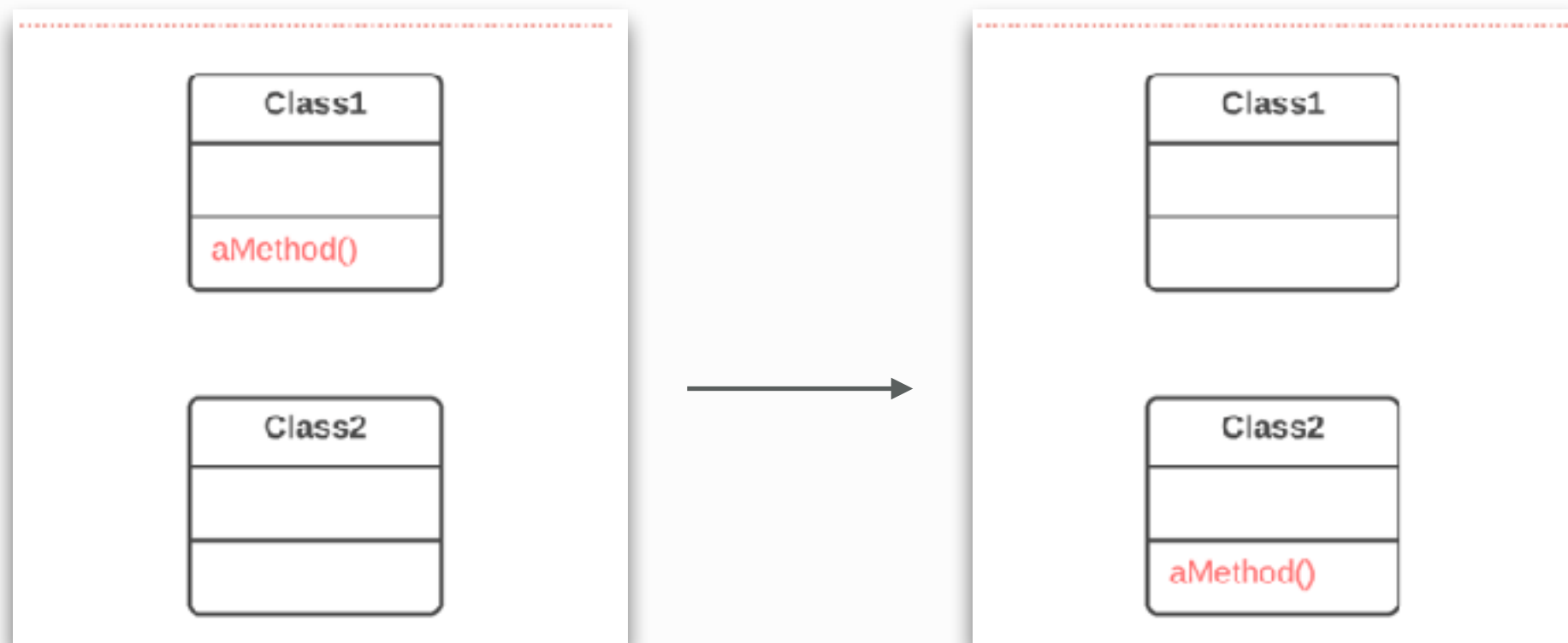
Refactoring - Moving Features Between Objects

Move Field/Method/Class. This refactoring family consists of moving a source code element toward another one in cases where it is more interested in the functionalities of the other element with respect to the one it is actually in.



Refactoring - Moving Features Between Objects

Move Field/Method/Class. This refactoring family consists of moving a source code element toward another one in cases where it is more interested in the functionalities of the other element with respect to the one it is actually in.



Some precautions

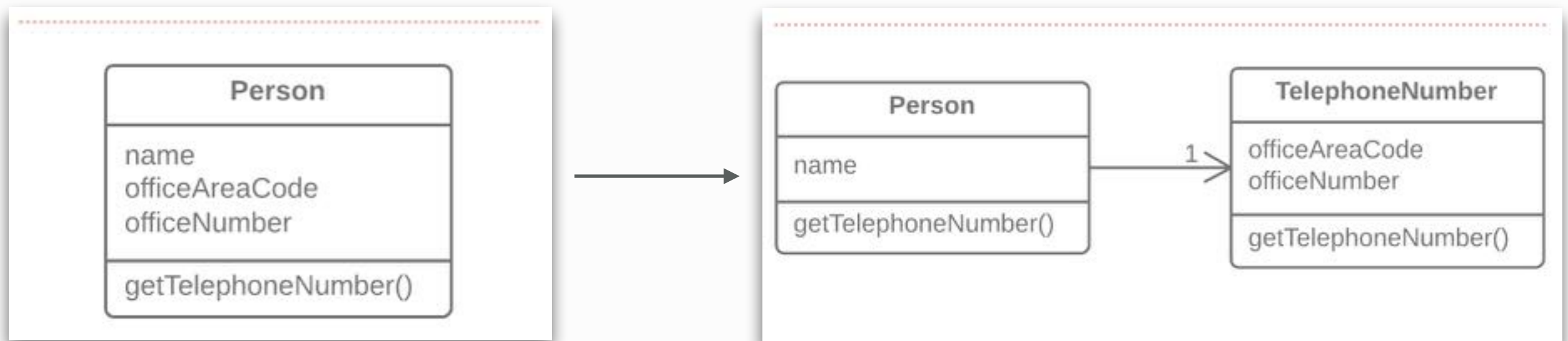
- Make sure that the method **is not** declared in superclasses and subclasses. If this is the case, you will either have to refrain from moving or else implement a kind of polymorphism in the recipient class in order to ensure varying functionality of a method split up among donor classes.

Refactoring - Moving Features Between Objects

Extract Class/Package. This refactoring family consists of moving methods/classes into a new class/package to increase cohesion and possibly improve the decomposition of subsystems in a software project.

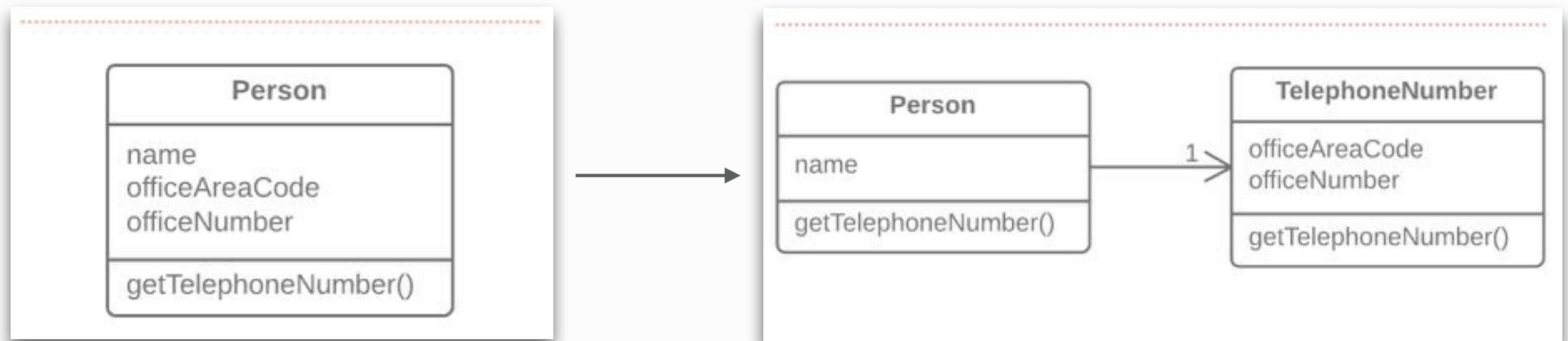
Refactoring - Moving Features Between Objects

Extract Class/Package. This refactoring family consists of moving methods/classes into a new class/package to increase cohesion and possibly improve the decomposition of subsystems in a software project.



Refactoring - Moving Features Between Objects

Extract Class/Package. This refactoring family consists of moving methods/classes into a new class/package to increase cohesion and possibly improve the decomposition of subsystems in a software project.

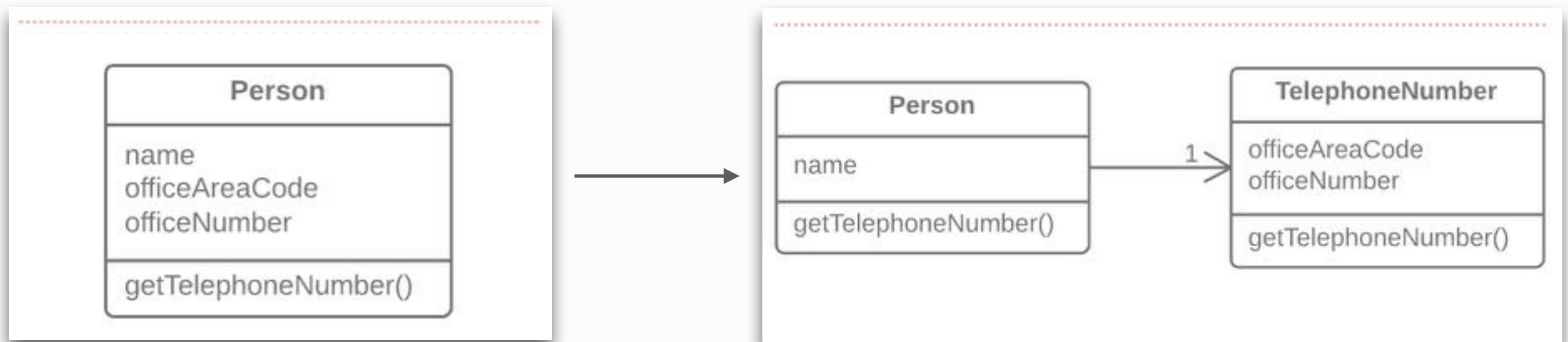


Some precautions

- Be careful with the selection of methods/classes that must be extracted.

Refactoring - Moving Features Between Objects

Extract Class/Package. This refactoring family consists of moving methods/classes into a new class/package to increase cohesion and possibly improve the decomposition of subsystems in a software project.

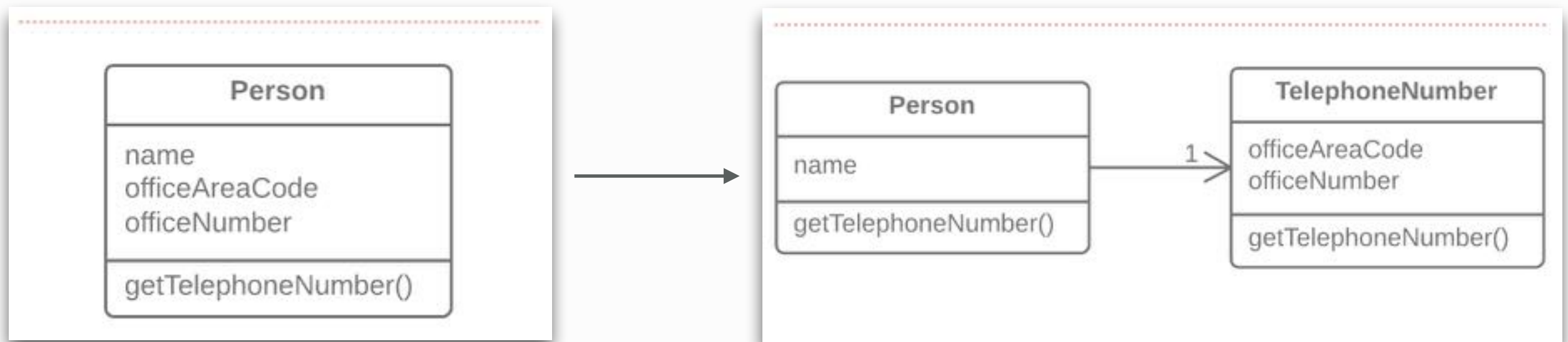


Some precautions

- Be careful with the selection of methods/classes that must be extracted.
- Create a relationship between the old class and the new one. This relationship should be unidirectional - this allows reusing the second class without any issues. However, if you think that a two-way relationship is necessary, this can always be set up.

Refactoring - Moving Features Between Objects

Extract Class/Package. This refactoring family consists of moving methods/classes into a new class/package to increase cohesion and possibly improve the decomposition of subsystems in a software project.



Some precautions

- Be careful with the selection of methods/classes that must be extracted.
- Create a relationship between the old class and the new one. This relationship should be unidirectional - this allows reusing the second class without any issues. However, if you think that a two-way relationship is necessary, this can always be set up.
- An old class with changed responsibilities may be renamed for increased clarity.

Bad Code Smells

Bad Code Smells - Relation with Refactoring

Refactoring is the process of **changing the internal structure of the source code without altering its external behavior.**

Refactoring is the action required to remove a code smell. More refactoring operations can be applied for the removal of multiple code smells - so **there is a 1 to N relationship between code smells and refactoring.**

God Class (Blob)

Swiss Army Knife

Divergent Change

Bad Code Smells

Bad Code Smells - Relation with Refactoring

Refactoring is the process of **changing the internal structure of the source code without altering its external behavior.**

Refactoring is the action required to remove a code smell. More refactoring operations can be applied for the removal of multiple code smells - so **there is a 1 to N relationship between code smells and refactoring.**

God Class (Blob)

Swiss Army Knife

Divergent Change



Extract Class Refactoring. Action through which a class is split in multiple classes in an effort of improving code cohesion and reducing coupling between objects.

Bad Code Smells

Bad Code Smells - Relation with Refactoring

Refactoring is the process of **changing the internal structure of the source code without altering its external behavior.**

Refactoring is the action required to remove a code smell. More refactoring operations can be applied for the removal of multiple code smells - so **there is a 1 to N relationship between code smells and refactoring.**

God Class (Blob)

Swiss Army Knife

Divergent Change



Extract Class Refactoring. Action through which a class is split in multiple classes in an effort of improving code cohesion and reducing coupling between objects.

Feature Envy

Bad Code Smells

Bad Code Smells - Relation with Refactoring

Refactoring is the process of **changing the internal structure of the source code without altering its external behavior.**

Refactoring is the action required to remove a code smell. More refactoring operations can be applied for the removal of multiple code smells - so **there is a 1 to N relationship between code smells and refactoring.**

God Class (Blob)

Swiss Army Knife

Divergent Change



Extract Class Refactoring. Action through which a class is split in multiple classes in an effort of improving code cohesion and reducing coupling between objects.

Feature Envy



Move Method Refactoring. Action through which a method is moved toward another class in cases where it is more interested in the functionalities of the other class.

Bad Code Smells - Automating the process

A typical refactoring process is composed of the steps below. In this lecture, we focus on the first two: **Where** and **How to refactor**.

Where to refactor

How to refactor?

Guarantee behavior preservation

Apply the refactoring

Assess its effects on quality

Consistently modify other artifacts

Mens and Tourwé
A Survey of Software Refactoring
TSE 2004

Bad Code Smells - Automating the process

A typical refactoring process is composed of the steps below. In this lecture, we focus on the first two: **Where and How to refactor**.

Where to refactor

How to refactor?

Guarantee behavior preservation

Apply the refactoring

Assess its effects on quality

Consistently modify other artifacts

- Identification of portions of code in need of refactoring.

Mens and Tourwé
A Survey of Software Refactoring
TSE 2004

Bad Code Smells - Automating the process

A typical refactoring process is composed of the steps below. In this lecture, we focus on the first two: **Where and How to refactor**.

Where to refactor

How to refactor?

Guarantee behavior preservation

Apply the refactoring

Assess its effects on quality

Consistently modify other artifacts

- Identification of portions of code in need of refactoring.
- Various approaches to solve this problem have been proposed.

Mens and Tourwé
A Survey of Software Refactoring
TSE 2004

Bad Code Smells - Automating the process

A typical refactoring process is composed of the steps below. In this lecture, we focus on the first two: **Where and How to refactor**.

Where to refactor

How to refactor?

Guarantee behavior preservation

Apply the refactoring

Assess its effects on quality

Consistently modify other artifacts

- Identification of portions of code in need of refactoring.
- Various approaches to solve this problem have been proposed.

Source

Structural, historical, conceptual information have been assessed.

Mens and Tourwé
A Survey of Software Refactoring
TSE 2004

Bad Code Smells - Automating the process

A typical refactoring process is composed of the steps below. In this lecture, we focus on the first two: **Where and How to refactor**.

Where to refactor

How to refactor?

Guarantee behavior preservation

Apply the refactoring

Assess its effects on quality

Consistently modify other artifacts

Mens and Tourwé
A Survey of Software Refactoring
TSE 2004

- Identification of portions of code in need of refactoring.
- Various approaches to solve this problem have been proposed.

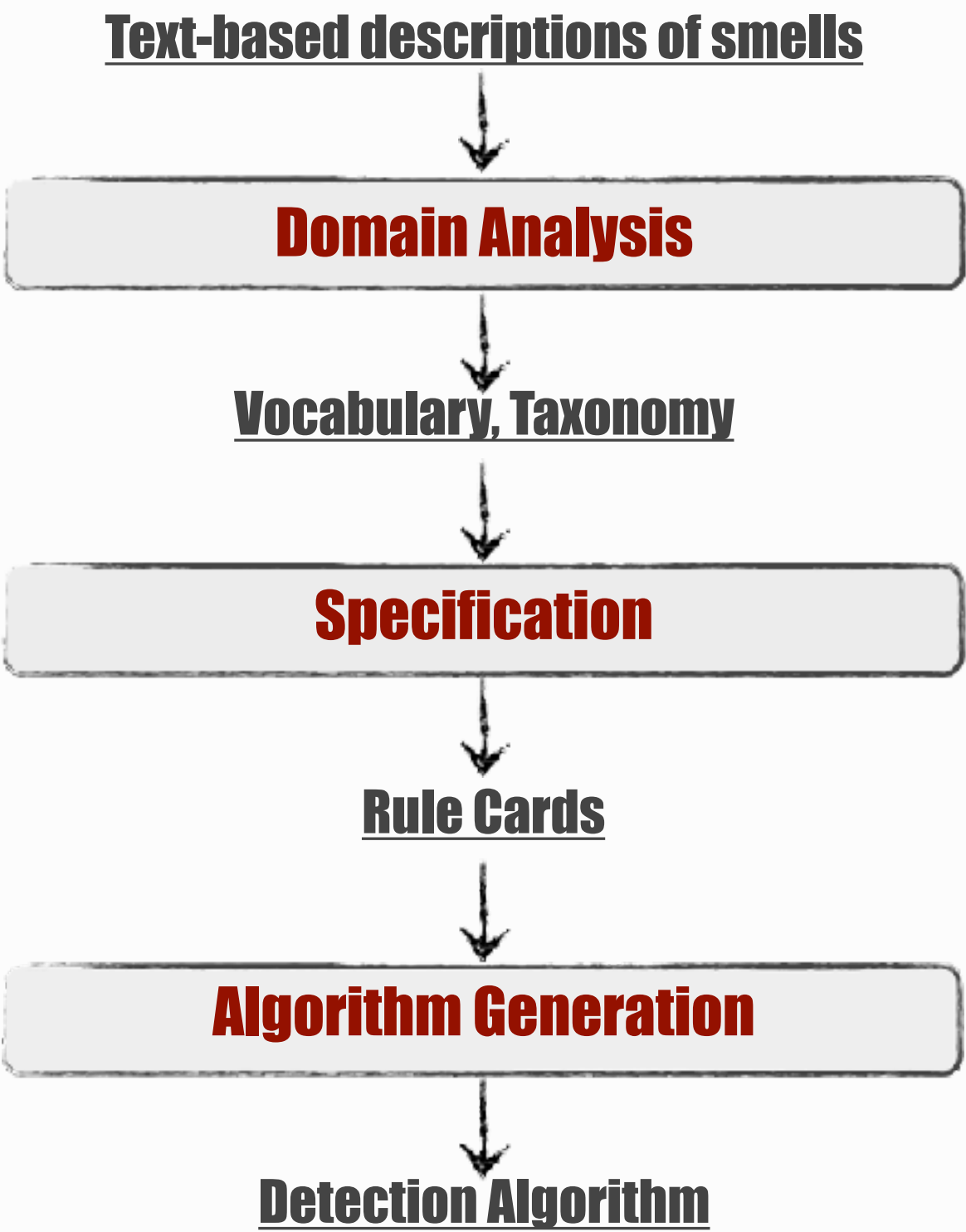
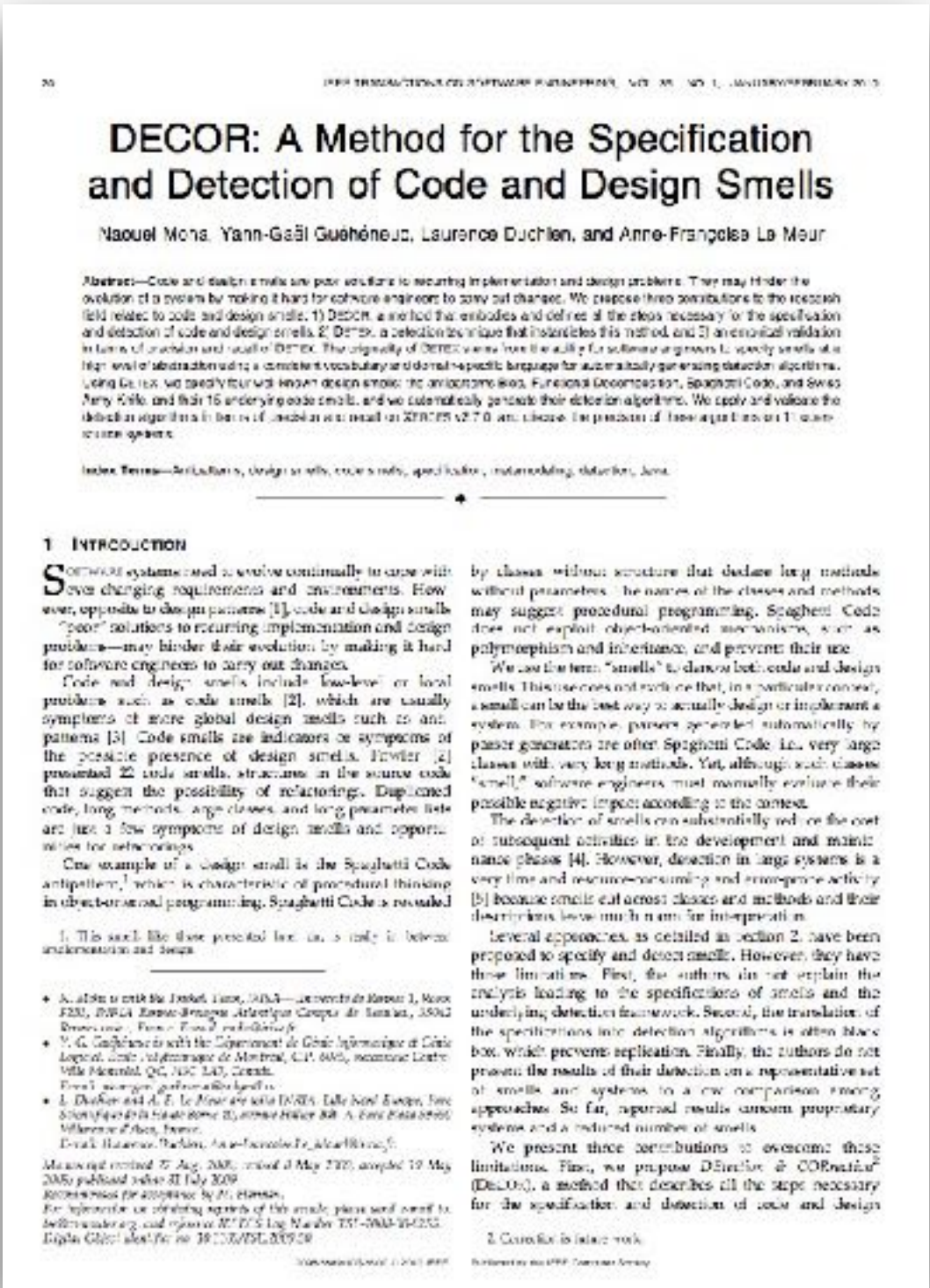
Source

Structural, historical, conceptual information have been assessed.

Method

Most of the available detectors are based on heuristics, some of them use machine learning or search-based solutions.

Bad Code Smells - Where to Refactor



Bad Code Smells - Where to Refactor

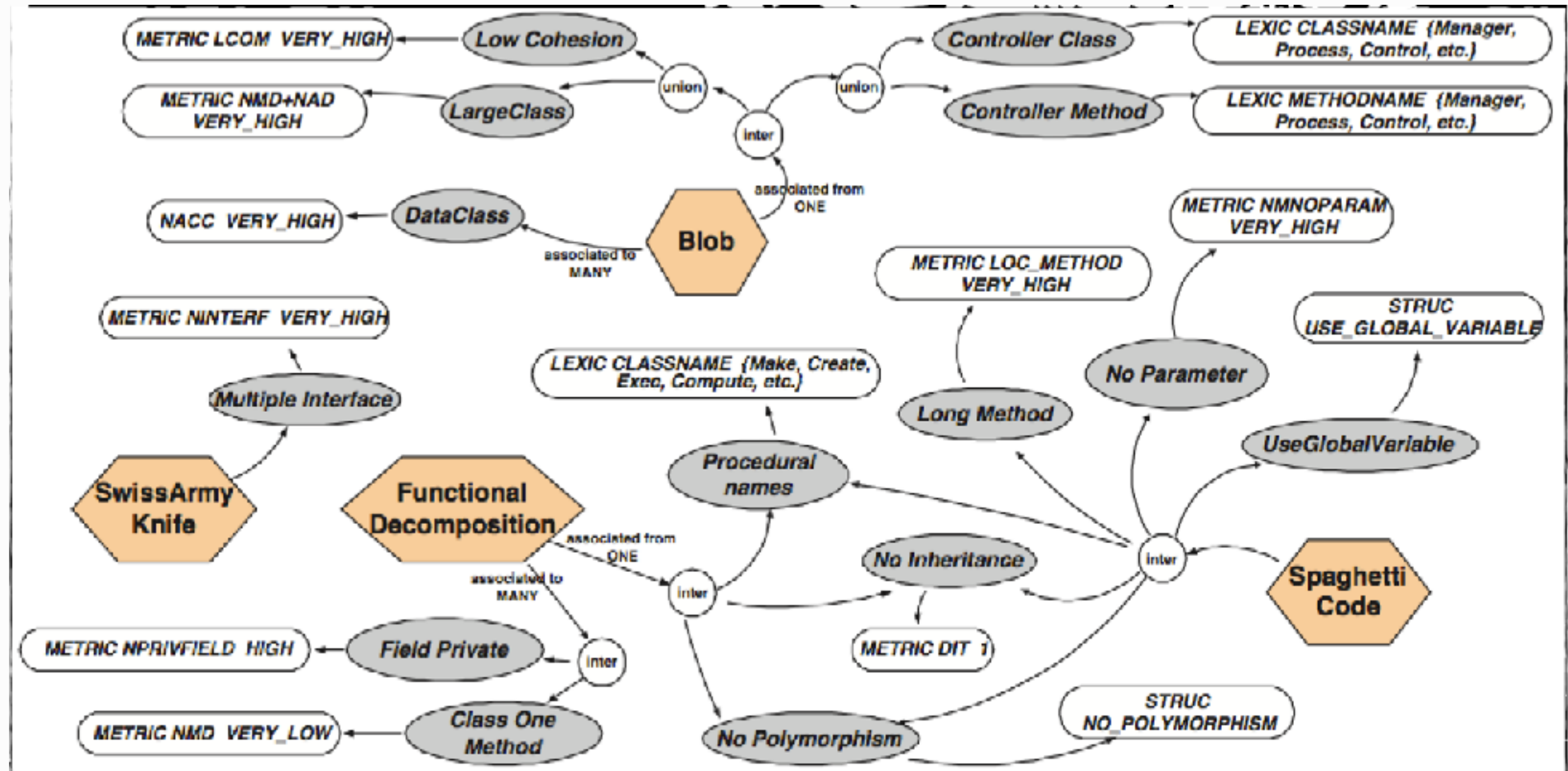
- Step I: Input example - A textual description of Blob.

The Blob (also called God class) corresponds to a large controller class that depends on data stored in surrounding data classes. A large class declares many fields and methods with a low cohesion. A controller class monopolizes most of the processing done by a system, takes most of the decisions, and closely directs the processing of other classes. Controller classes can be identified using suspicious names such as Process, Control, Manage, System, and so on. A data class contains only data and performs no processing on these data. It is composed of highly cohesive fields and accessors.

Bad Code Smells

Bad Code Smells - Where to Refactor

- Step II: Taxonomy creation and specification.



Bad Code Smells - Where to Refactor

- Step III: Algorithm definition and specification.

RULE_CARD : Blob {

RULE : Blob {ASSOC: associated FROM : mainClass ONE TO : DataClass MANY};

RULE : MainClass {UNION LargeClass, LowCohesion, ControllerClass};

RULE : LargeClass {(METRIC : NMD + NAD, VERY_HIGH, 20) } ;

RULE : LowCohesion { (METRIC : LCOM5, VERY_HIGH , 20) } ;

RULE : ControllerClass { UNION (SEMANTIC : METHODNAME,

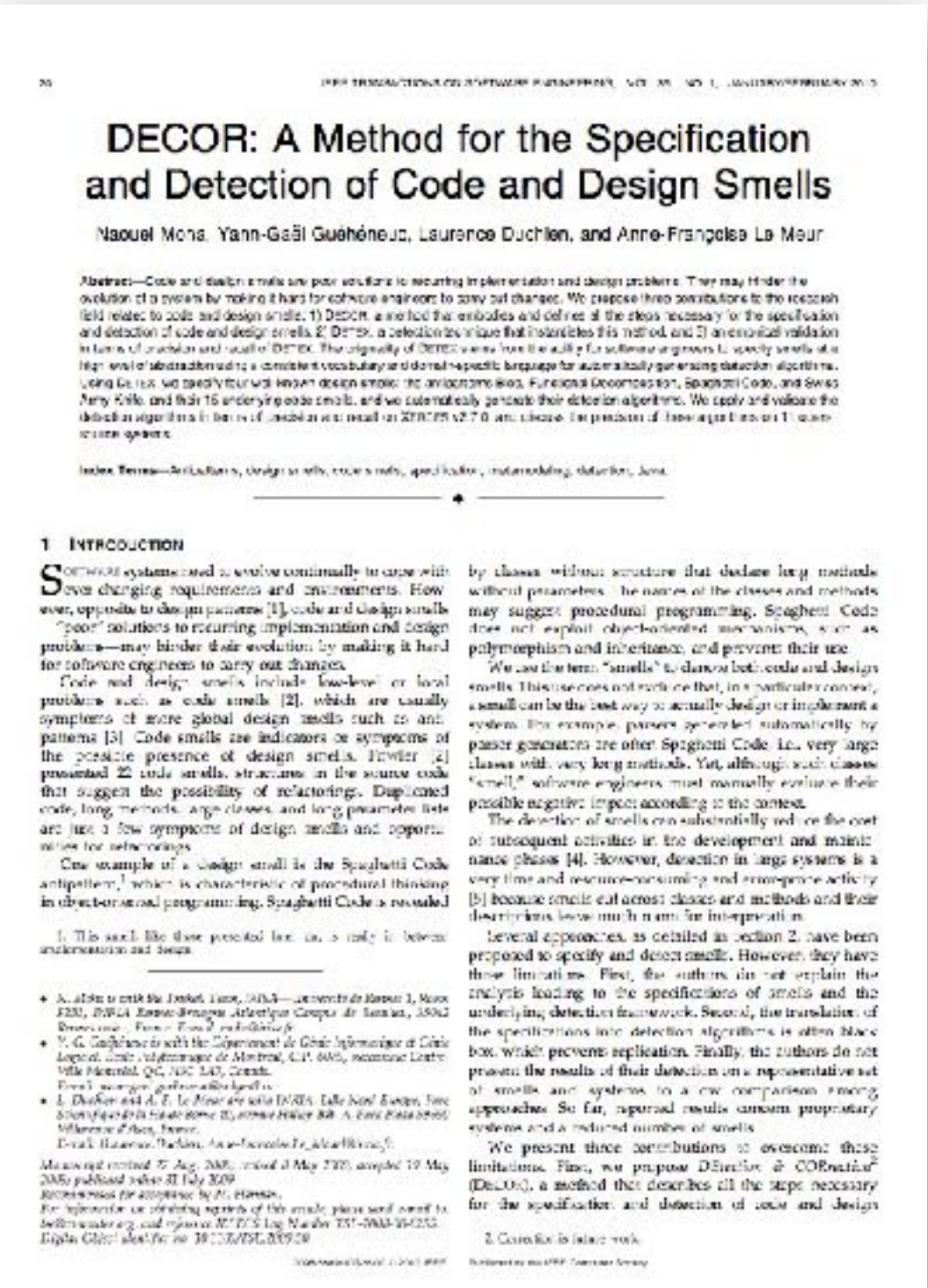
{Process, Control , Ctrl , Command , Cmd, Proc, UI, Manage, Drive})

(SEMANTIC : CLASSNAME, { Process, Control, Ctrl, Command , Cmd, Proc , UI, Manage, Drive , System, Subsystem }) } ;

RULE : DataClass {(STRUCT: METHOD_ACCESSOR, 90%)} ;

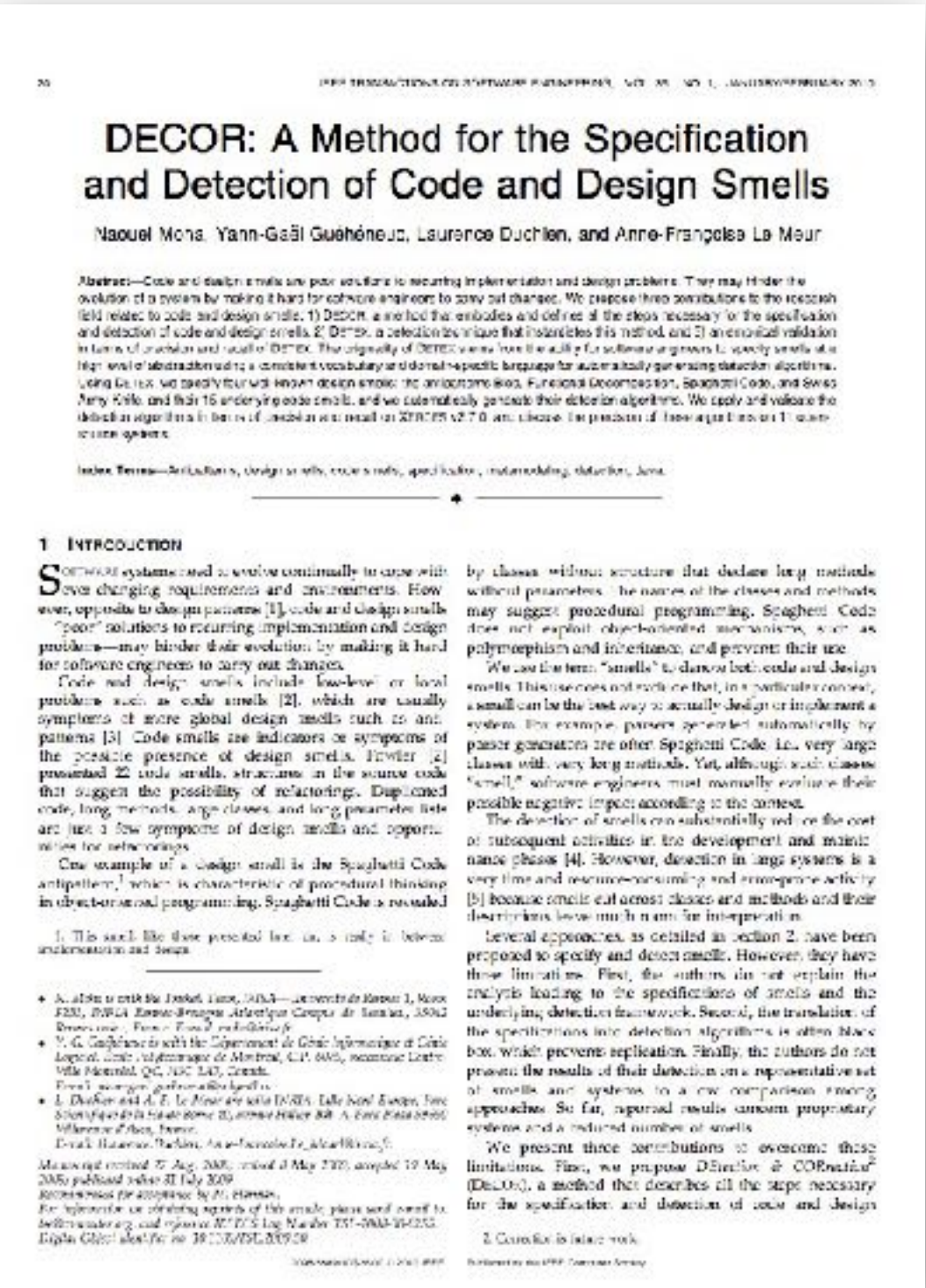
};

Bad Code Smells - Where to Refactor



To sum up, DECOR is a **general method** to specify detection rules. Starting from the textual description of a code smell, one can apply DECOR by analyzing its domain and specifying suitable metrics to detect it.

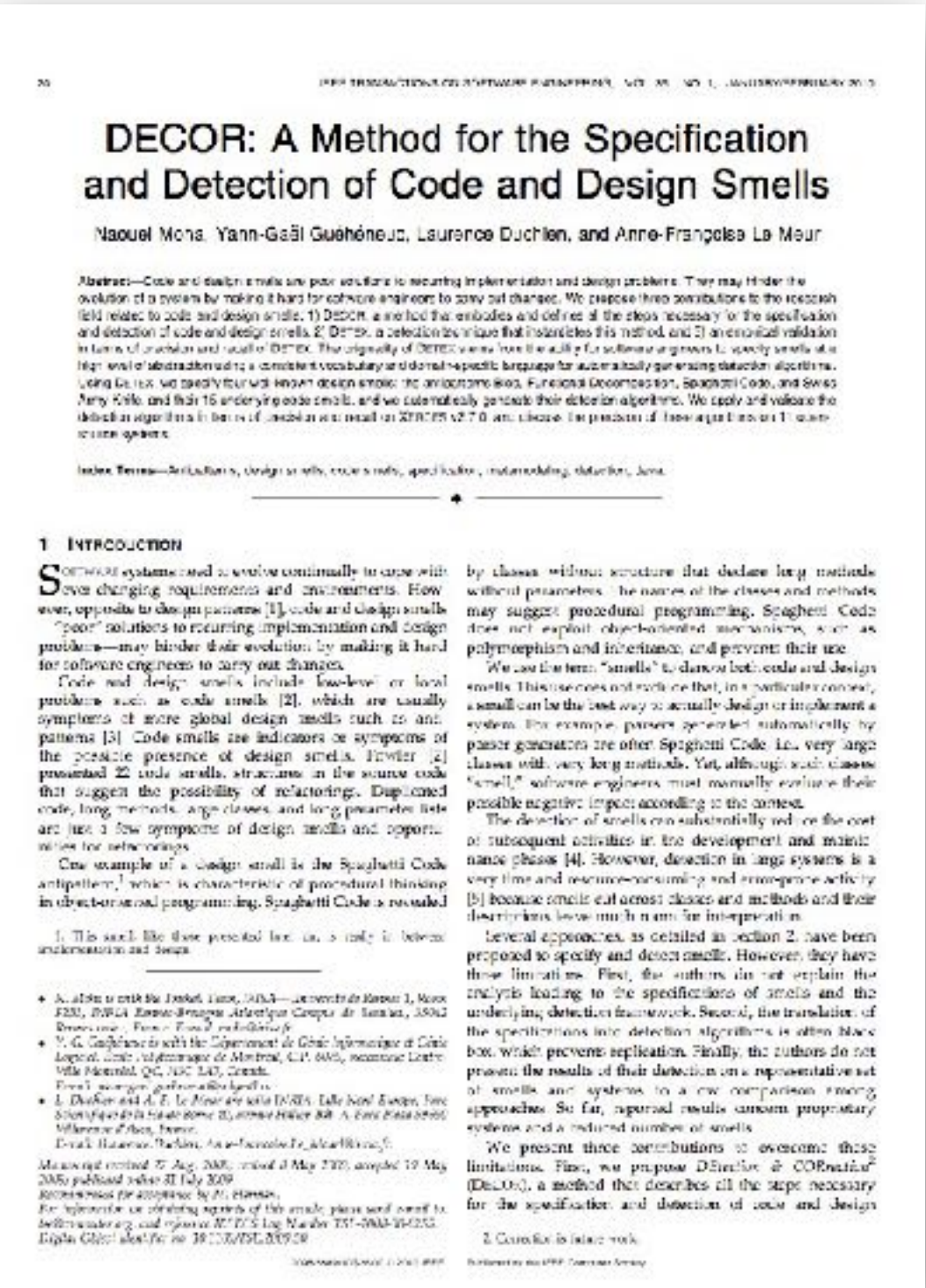
Bad Code Smells - Where to Refactor



To sum up, DECOR is a **general method** to specify detection rules. Starting from the textual description of a code smell, one can apply DECOR by analyzing its domain and specifying suitable metrics to detect it.

DECOR is mainly about structural analysis, as most of the metrics are computed on source code. Nevertheless, it was the first including the concept of **lexical analysis** for detecting code smells, even though it is limited to the evaluation of class and method names.

Bad Code Smells - Where to Refactor



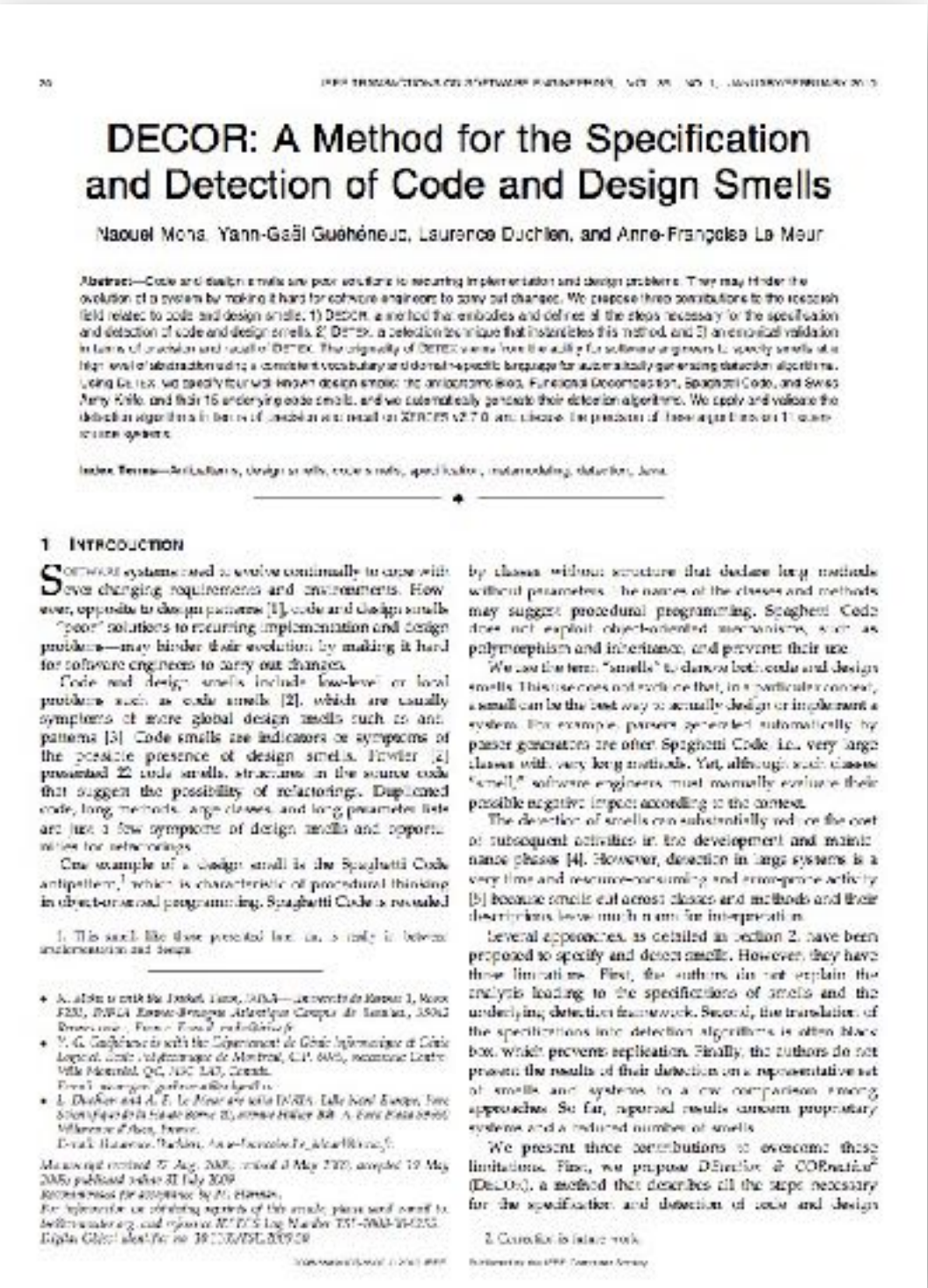
To sum up, DECOR is a **general method** to specify detection rules. Starting from the textual description of a code smell, one can apply DECOR by analyzing its domain and specifying suitable metrics to detect it.

DECOR is mainly about structural analysis, as most of the metrics are computed on source code. Nevertheless, it was the first including the concept of **lexical analysis** for detecting code smells, even though it is limited to the evaluation of class and method names.

Given the **high extensibility** of the approach, DECOR can be used to include additional perspectives and mix together various sources of information. Also, it may be employed within other detection tools.

Bad Code Smells - Where to Refactor

- DECOR has been originally validated on one software project;



Bad Code Smells - Where to Refactor

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 errors are poor vehicles for securing implementation and design problems. They may hinder the evolution of a system by making it hard for developers to make any changes. We propose three contributions to the research field related to code and design errors: 1) DETECT, a method that embodies and defines all the steps necessary to the specification and detection of code and design errors; 2) DETECT, a detection technique that instantiates this method; and 3) an empirical validation in terms of precision and recall of DETECT. The originality of DETECT comes from its ability for software engineers to specify errors at a high level of abstraction using a complete vocabulary and domain-specific language for automatically generating detection algorithms. Using DETECT, we specify four real-world design errors: the arithmetic error, Functional Decomposition, Spaghetti Code, and Switch Error. We, and their 16 underlying code errors, and we automatically generate their detection algorithms. We apply and validate the detection algorithms in terms of precision and recall on DETECT v2.0. We discuss the problem of false-alarms and of false-negative errors.

Index Terms—Artificial neural networks, design synthesis, error analysis, speed analysis, metamodelling, abstraction, 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 refactoring. Duplicated code, long methods, large classes, and long parameter lists are just a few symptoms of design smells and opportunities for refactoring.

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

1. This scale, like those presented here, can be used in between implementation and design.

- A. Allard est maître de conférences, 1983-1987 — Université de Moncton 1, Moncton F2B, 7814A Avenue-Bourgeois, Université du Nouveau Brunswick, 5502 Boulevard de la France, 2nd étage, Saint-Jean.
- Y. G. Gagnon est avec le Département de Génie Informatique et Génie Logiciel, Université polytechnique de Montréal, C.P. 6080, succursale Centre Ville Montréal, QC, H3C 3A7, Canada.
- J. D. Gagnon, professeur adjoint à l'Université de Moncton.
- J. D. Gagnon et A. L. Le Moine ont écrit *INAP, L'Art de la Recherche, Texte Scientifique de la Faculté de Génie 22, Avenue Hédou, 22, C. P. 6080 Centre Ville de Montréal, Québec, Canada.*
- David H. Harewood, Ph.D., est professeur de génie chimique.

Manuscript received 27 Aug. 2005; revised 3 May 2006; accepted 19 May 2006; published online 31 July 2006

For information on obtaining a copy of this article, please send e-mail to: journal@wiley.com, and reference *J. Clin. Psychol.* 65: 289-290, 2009. Digitals only; doi: 10.1111/j.1469-7610.2009.02395.x

by classes without structure that declare long methods without parameters. The names of the classes and methods may suggest procedural programming. Squeak 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 mean 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 normally evaluate their possible negative impact according to the context.

The direction of smells can substantially reduce the cost of subsequent activities in the development and maintenance phases [4]. However, direction 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 have much in common for integrational

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 a few comparisons among approaches. So far, reported results concern proprietary systems and a reduced number of smells.

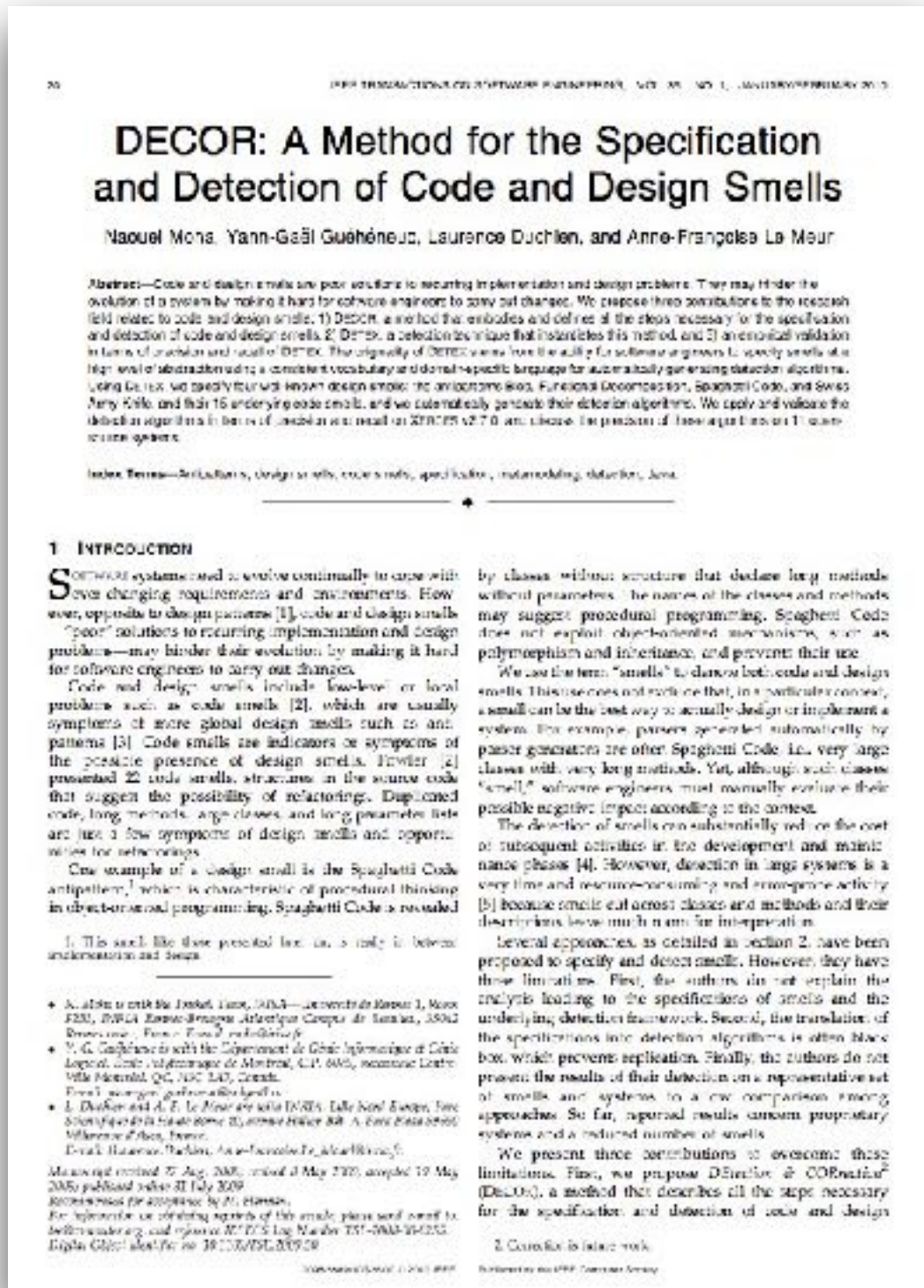
We present three contributions to overcome these limitations. First, we propose D_{Encoder} & CORnet² (DECo), a method that describes all the steps necessary for the modification and detection of code and design

2. Confidential is being used.

Reprinted by permission of the Copyright Clearance Center

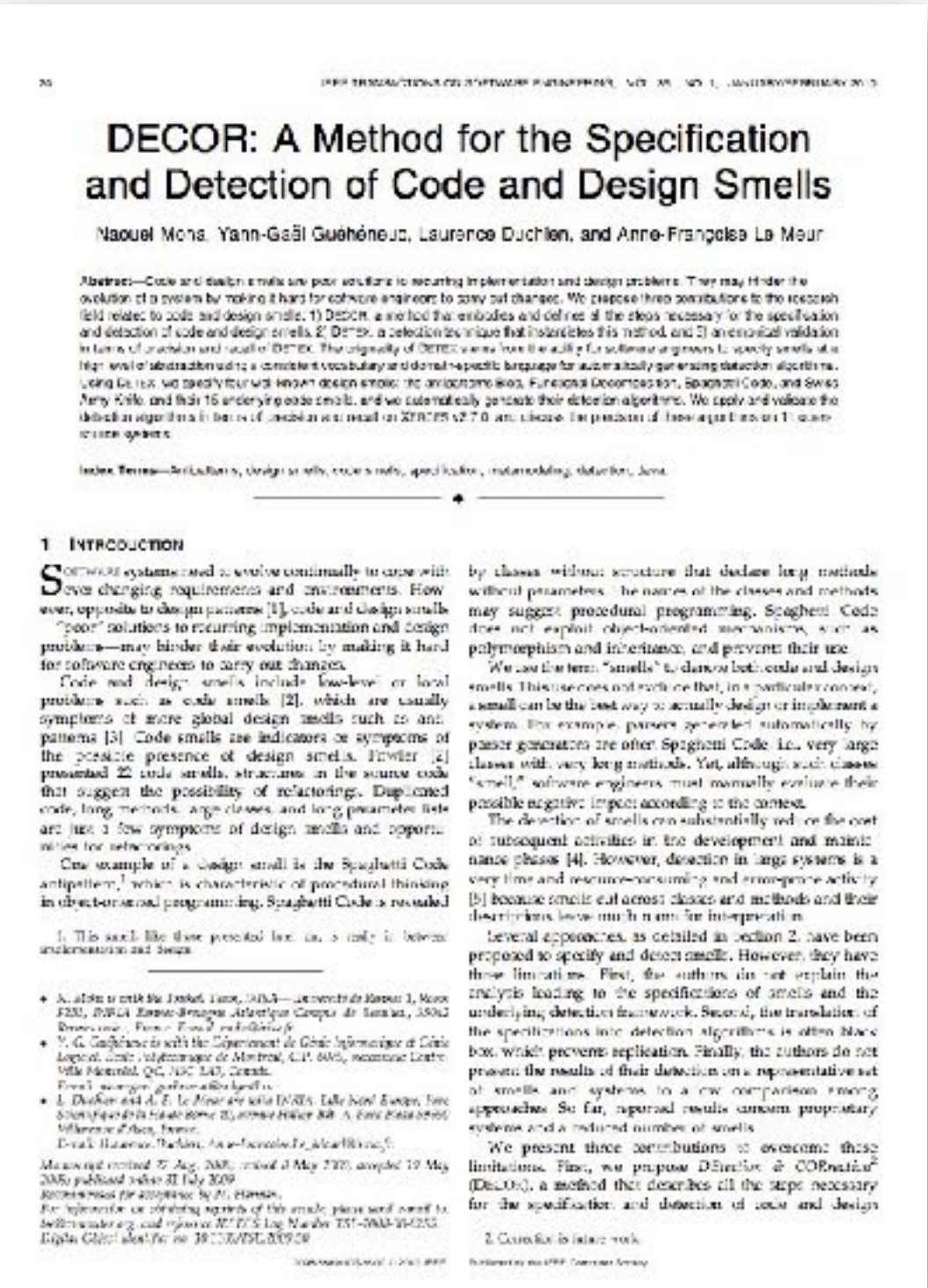
- DECOR has been originally validated on one software project;
- DECOR has been originally instantiated to detect five code smell types;

Bad Code Smells - Where to Refactor



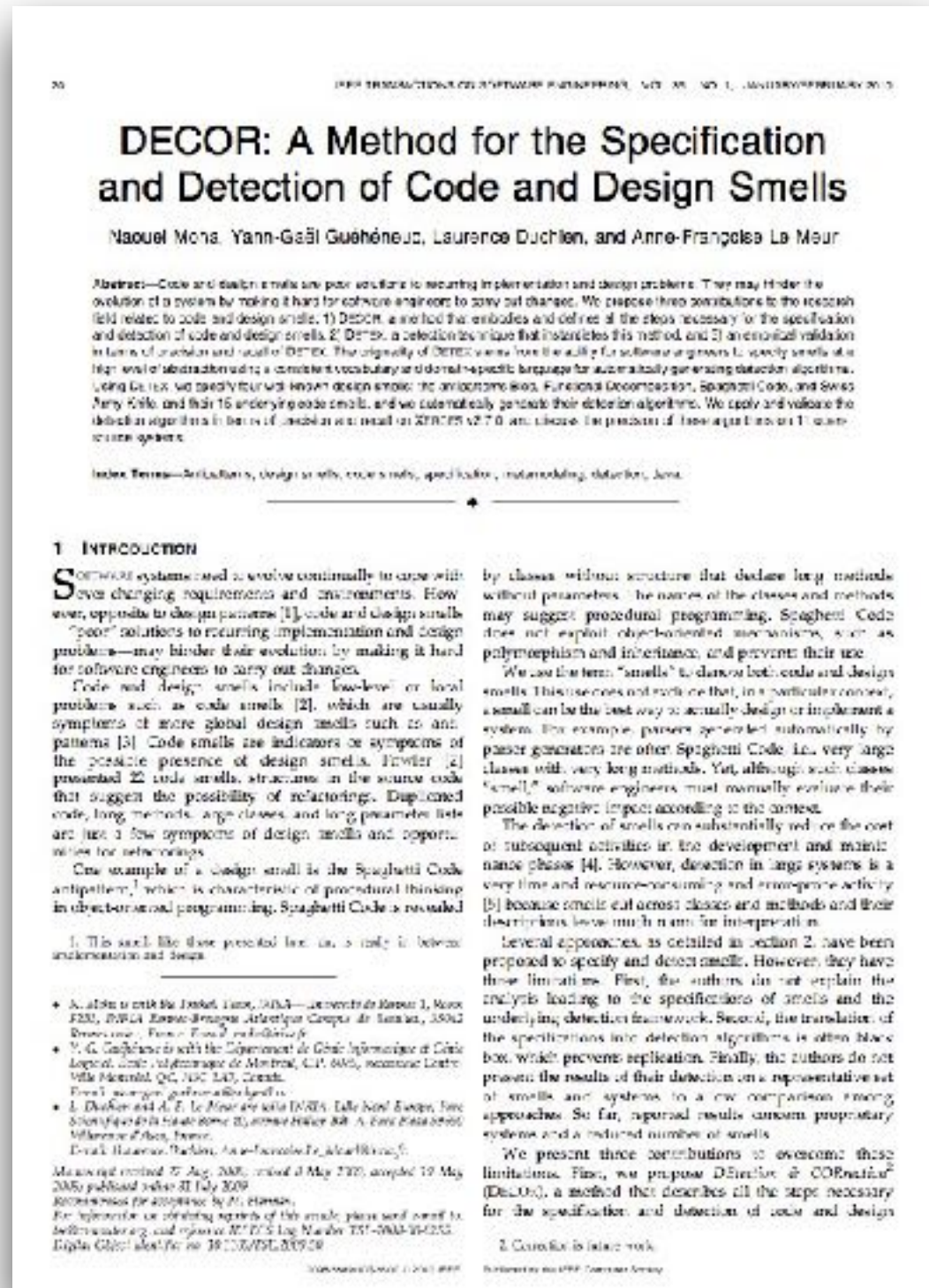
- DECOR has been originally validated on one software project;
- DECOR has been originally instantiated to detect five code smell types;
- According to the achieved results, DECOR has precision and recall of 60% and 100%, respectively;

Bad Code Smells - Where to Refactor



- DECOR has been originally validated on one software project;
- DECOR has been originally instantiated to detect five code smell types;
- According to the achieved results, DECOR has precision and recall of 60% and 100%, respectively;
- Further evaluations done in other studies lowered the expectations: depending on the dataset, DECOR has a precision of 30%-70% and a recall of 40%-60%.

Bad Code Smells - Where to Refactor



- DECOR has been originally validated on one software project;
- DECOR has been originally instantiated to detect five code smell types;
- According to the achieved results, DECOR has precision and recall of 60% and 100%, respectively;
- Further evaluations done in other studies lowered the expectations: depending on the dataset, DECOR has a precision of 30%-70% and a recall of 40%-60%.
- It is, however, one of the most widely used tools. A big pro is represented by its scalability, which allows it to be used in the large.

Bad Code Smells - Where to Refactor

Unfortunately, structural analysis is not bulletproof. There exist code smells which are, by nature, **not “structural”** and that, therefore, are hard to identify with structural metrics.

Bad Code Smells

Bad Code Smells - Where to Refactor

Unfortunately, structural analysis is not bulletproof. There exist code smells which are, by nature, **not “structural”** and that, therefore, are hard to identify with structural metrics.

Some more definitions needed

Parallel Inheritance. This code smell appears when, every time a developer adds a subclass of one class, s/he has to add a subclass of another class.

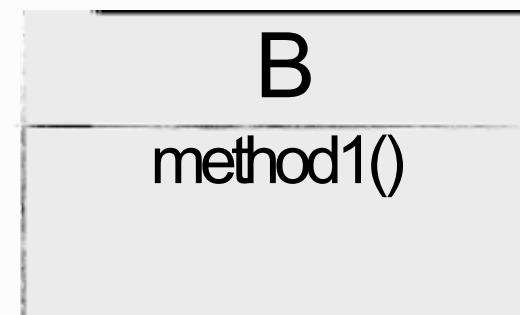
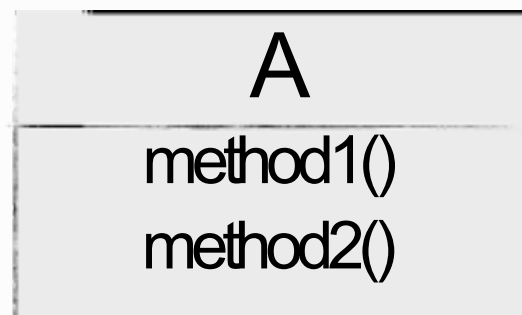
Bad Code Smells

Bad Code Smells - Where to Refactor

Unfortunately, structural analysis is not bulletproof. There exist code smells which are, by nature, **not “structural”** and that, therefore, are hard to identify with structural metrics.

Some more definitions needed

Parallel Inheritance. This code smell appears when, every time a developer adds a subclass of one class, s/he has to add a subclass of another class.



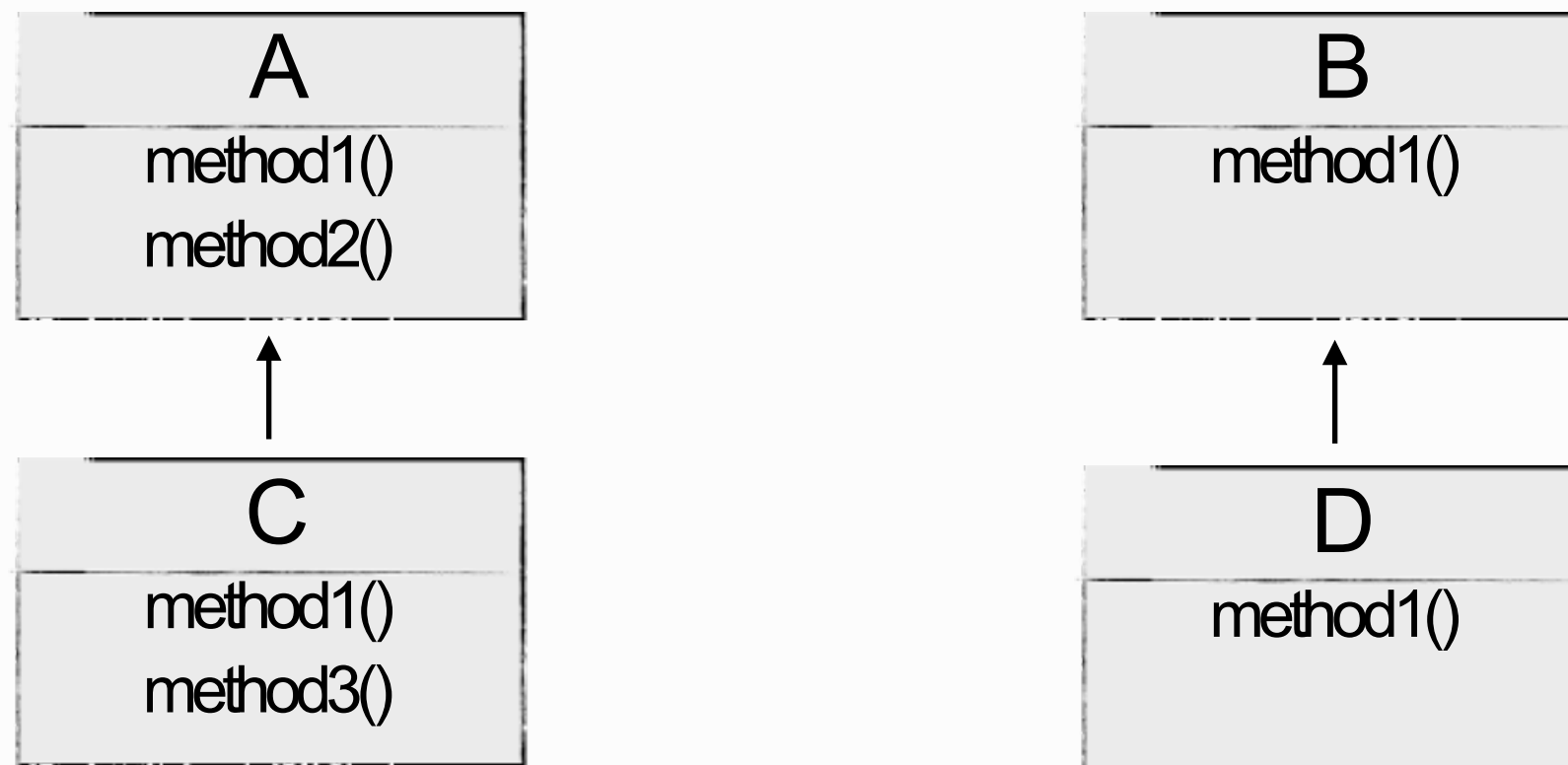
Bad Code Smells

Bad Code Smells - Where to Refactor

Unfortunately, structural analysis is not bulletproof. There exist code smells which are, by nature, **not “structural”** and that, therefore, are hard to identify with structural metrics.

Some more definitions needed

Parallel Inheritance. This code smell appears when, every time a developer adds a subclass of one class, s/he has to add a subclass of another class.



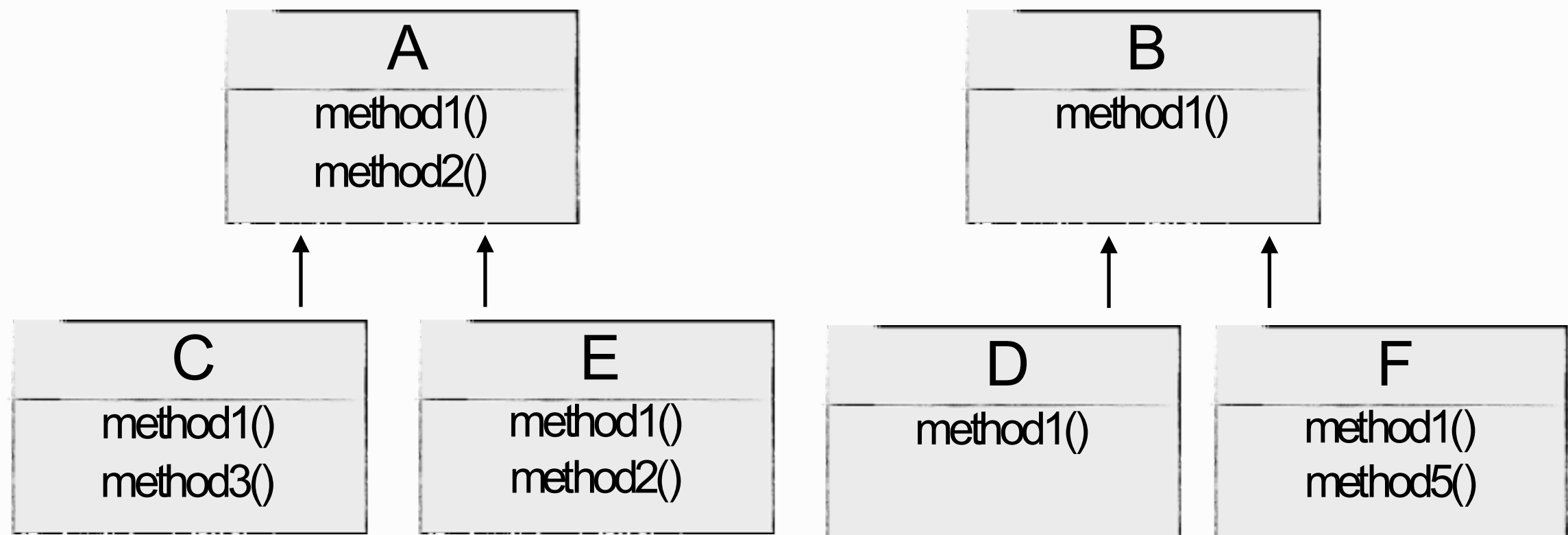
Bad Code Smells

Bad Code Smells - Where to Refactor

Unfortunately, structural analysis is not bulletproof. There exist code smells which are, by nature, **not “structural”** and that, therefore, are hard to identify with structural metrics.

Some more definitions needed

Parallel Inheritance. This code smell appears when, every time a developer adds a subclass of one class, s/he has to add a subclass of another class.



Bad Code Smells - Where to Refactor

Hence, in some cases the detection of code smells shall make use of change history information. This has been introduced by HIST (Historical Information for Smell deTection).

Bad Code Smells

Bad Code Smells - Where to Refactor

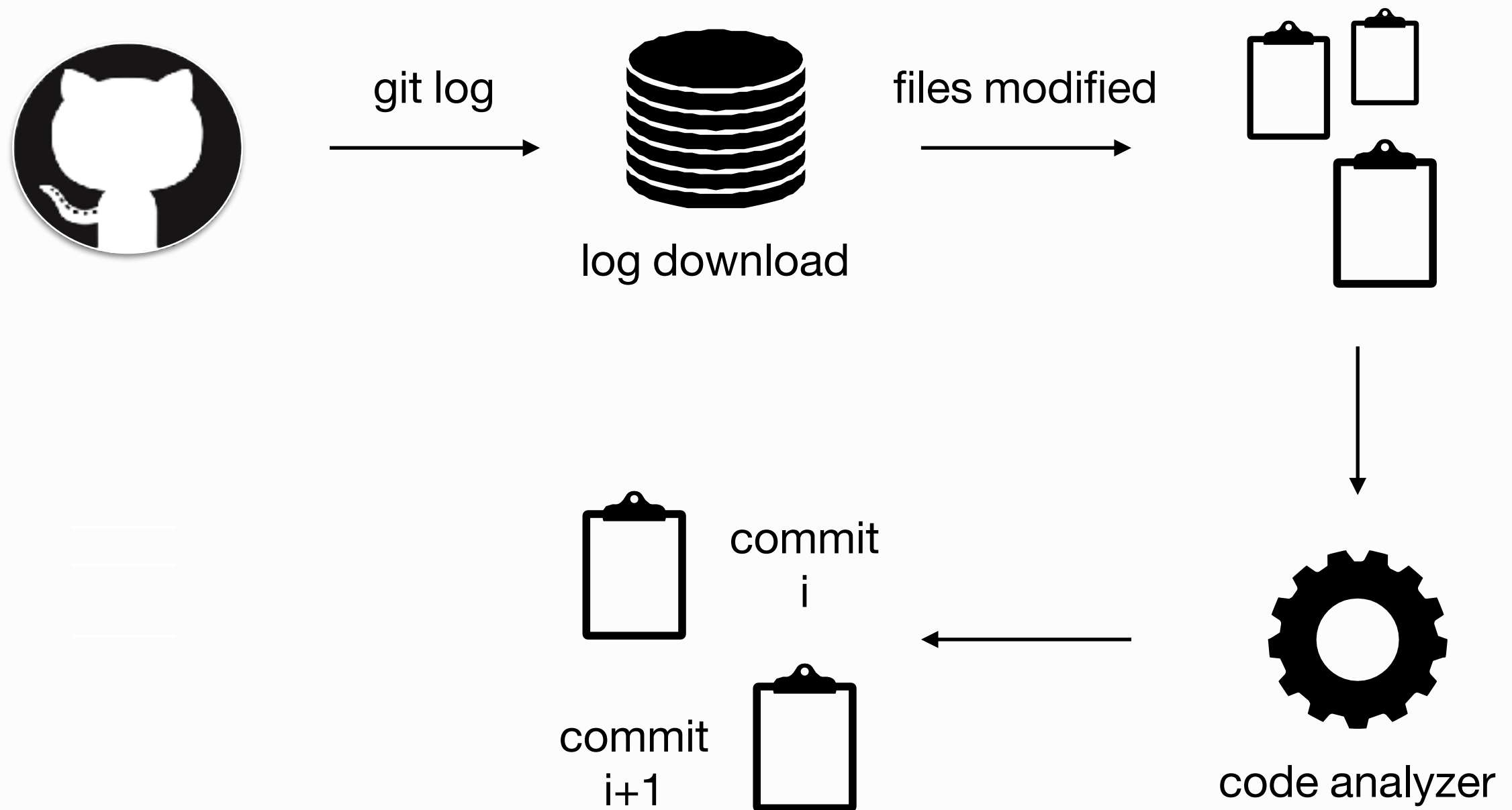
Hence, in some cases the detection of code smells shall make use of change history information. This has been introduced by HIST (Historical Information for Smell deTection).



Bad Code Smells

Bad Code Smells - Where to Refactor

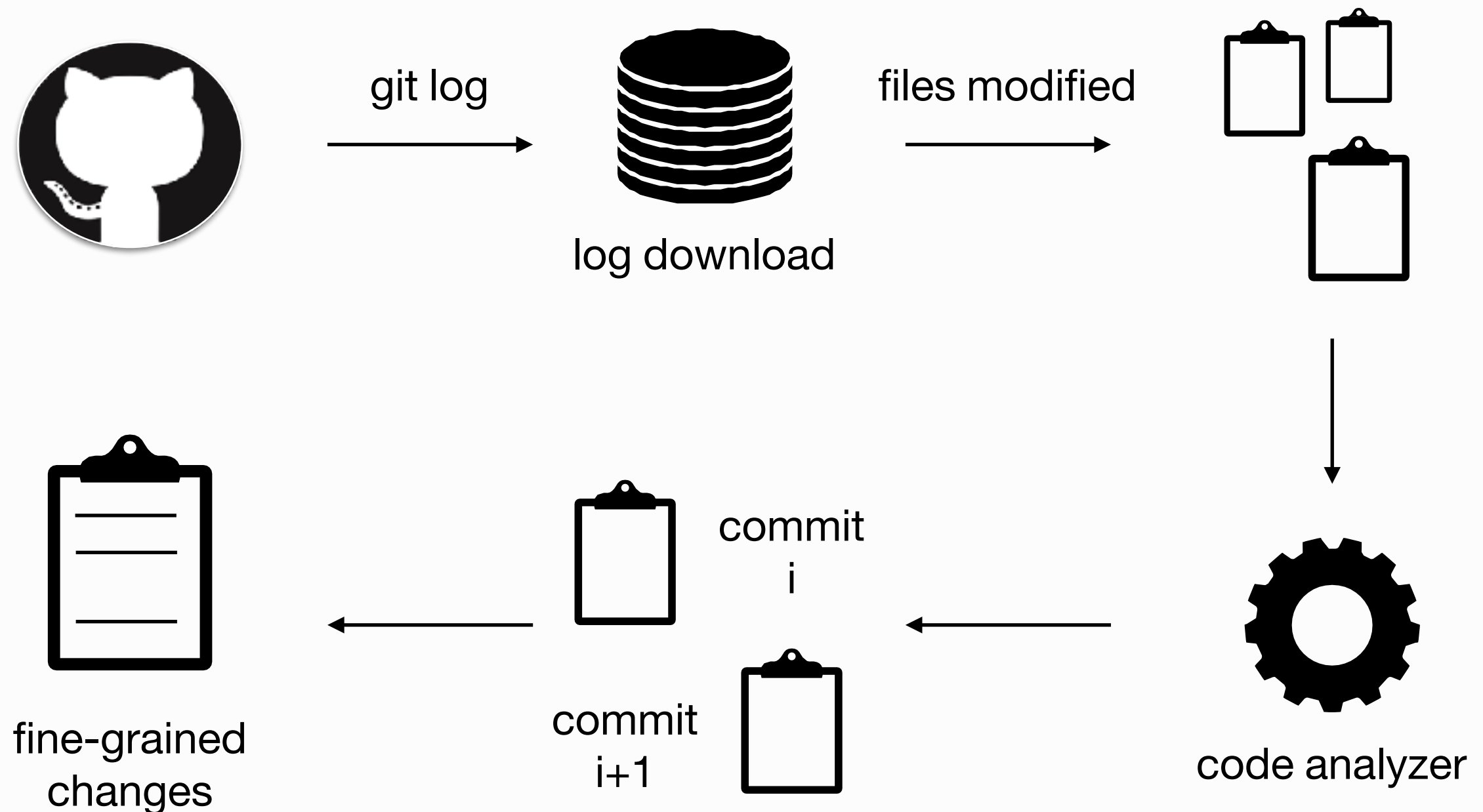
Hence, in some cases the detection of code smells shall make use of change history information. This has been introduced by HIST (Historical Information for Smell deTection).



Bad Code Smells

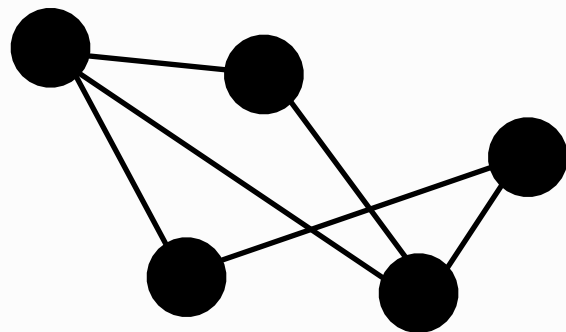
Bad Code Smells - Where to Refactor

Hence, in some cases the detection of code smells shall make use of change history information. This has been introduced by HIST (Historical Information for Smell deTection).

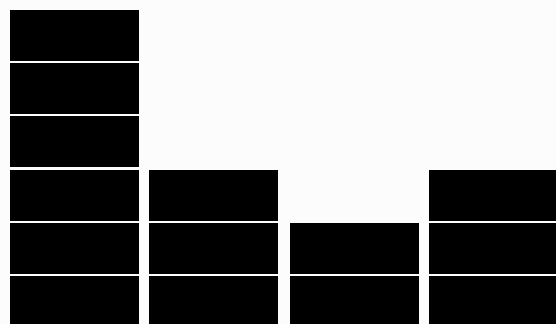


Bad Code Smells - Where to Refactor

Hence, in some cases the detection of code smells shall make use of change history information. This has been introduced by HIST (Historical Information for Smell deTection).



Association rule discovery to capture co-changes between entities

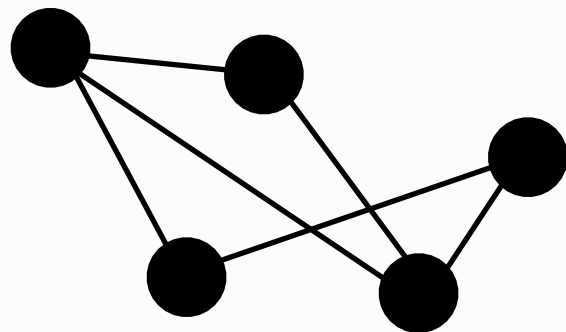


Analysis of change frequency of some specific entities

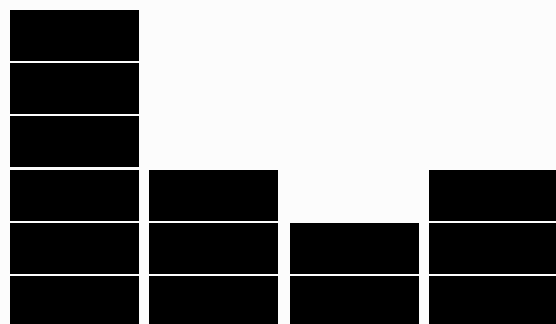
- Once extracted the fine-grained changes, the detector employs two different strategies to identify code smells

Bad Code Smells - Where to Refactor

Hence, in some cases the detection of code smells shall make use of change history information. This has been introduced by HIST (Historical Information for Smell deTection).



Association rule discovery to capture co-changes between entities

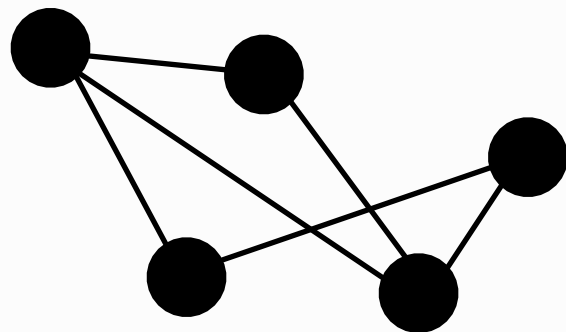


Analysis of change frequency of some specific entities

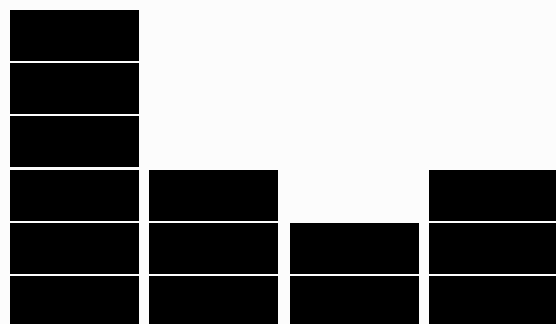
- Once extracted the fine-grained changes, the detector employs two different strategies to identify code smells
- HIST has been instantiated to identify three code smell types naturally historical: Divergent Change, Shotgun Surgery, and Parallel Inheritance.

Bad Code Smells - Where to Refactor

Hence, in some cases the detection of code smells shall make use of change history information. This has been introduced by HIST (Historical Information for Smell deTection).



Association rule discovery to capture co-changes between entities



Analysis of change frequency of some specific entities

- Once extracted the fine-grained changes, the detector employs two different strategies to identify code smells
- HIST has been instantiated to identify three code smell types naturally historical: Divergent Change, Shotgun Surgery, and Parallel Inheritance.
- HIST has been also instantiated to identify two traditional code smell types, Blob and Feature Envy, with the aim of understanding the extent to which historical analysis can be used as an alternative of structural one.

Bad Code Smells - Where to Refactor

Hence, in some cases the detection of code smells shall make use of change history information. This has been introduced by HIST (Historical Information for Smell deTection).

Divergent Change

A class is changed in different ways for different reasons

Solution:
Extract Class Refactoring

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



Bad Code Smells

Bad Code Smells - Where to Refactor

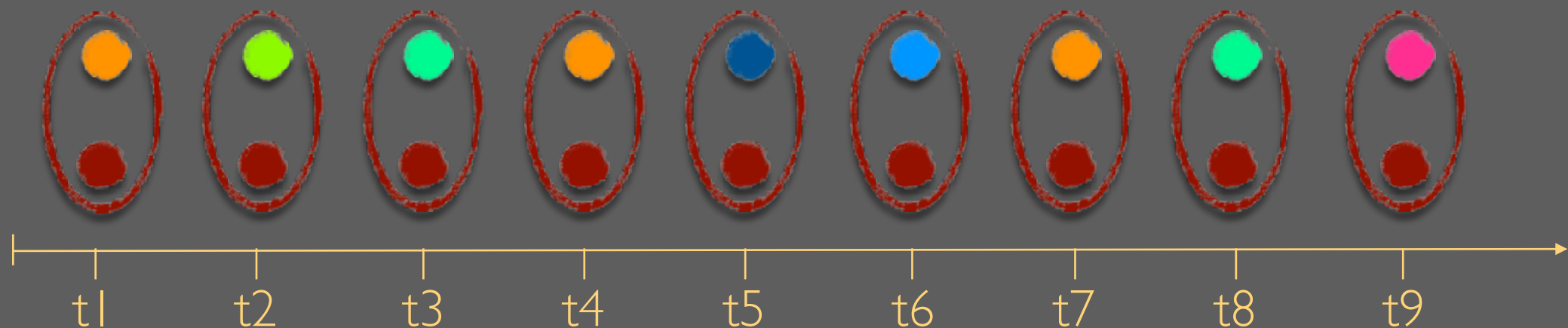
Hence, in some cases the detection of code smells shall make use of change history information. This has been introduced by HIST (Historical Information for Smell deTection).

Blob

A large class implementing several responsibilities

Solution:
Extract Class Refactoring

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



Bad Code Smells - Where to Refactor

Hence, in some cases the detection of code smells shall make use of change history information. This has been introduced by HIST (Historical Information for Smell deTection).

- HIST has been originally validated on twenty open-source software projects;

Mining Version Histories for Detecting Code Smells

Pablo Palomba¹, Gabriele Bavota², Massimiliano Di Penta²,
Rocco Oliveto³, Denys Poshyvanyk⁴, Andrea De Lucia¹

¹University of Salerno, Fisciano (SA), Italy

²University of Salerno, Benevento, Italy

³University of Molise, Pesche (IS), Italy

⁴The College of William and Mary, Williamsburg, VA, USA

ppalomba@unisai.it, gbavota@unisai.it, dipenta@unisai.it,
rocco.oliveto@unimol.it, denys@cs.wm.edu, adelucia@unisai.it

ABSTRACT. Code smells are symptoms of poor design and implementation choices that may hinder code comprehension, and possibly increase change- and fault-proneness. While most of the detection techniques just rely on structural information, many code smells are intrinsically characterized by how code elements change over time. In this paper, we propose HIST (Historical Information for Smell deTection), an approach exploring change history information to detect instances of two different code smells, namely Divergent Groups, Shotgun Surgery, Parallel Inheritance, Blob, and Feature Envy. Whereas HIST is an empirical study. The first, conducted on twenty open source projects, aimed at assessing the accuracy of HIST in detecting instances of the code smells mentioned above. The results indicate that the precision of HIST ranges between 72% and 80%, and its recall ranges between 58% and 100%. Also, results of the first study indicate that HIST is able to identify code smells that cannot be detected by competitive approaches solely based on code analysis or a single system's snapshot. Then, we conducted a second study aimed at investigating to what extent the code smells detected by HIST (and by competitive code analysis techniques) reflect developers' perception of poor design and implementation choices. We involved twelve developers of four open source projects that recognized more than 75% of the code smell instances identified by HIST as actual design/implementation problems.

Index Terms. Code Smells, Mining Software Repositories, Empirical Studies.

1 INTRODUCTION

Code smells have been defined by Fowler [14] as symptoms of poor design and implementation choices. In some cases, such symptoms may originate from activities performed by developers while in a hurry, e.g., implementing urgent patches or simply making suboptimal choices. In other cases, smells come from some recurring, poor design solutions, also known as anti-patterns [9]. For example, a *Blob* is a large and complex class that centralizes the behavior of a portion of a system and only uses other classes as data holders. Blob classes can rapidly grow out of control, making it harder and harder for developers to understand them, to fix bugs, and to add new features.

Previous studies have found that smells hinder comprehension [1], and possibly increase changes and fault-proneness [23], [24]. In summary, smells need to be carefully detected and monitored and, whenever necessary, refactoring actions should be planned and performed to deal with them.

This paper is an extension of "Detecting Bad Smells in Source Code Using Change History Information" that appeared in the Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE 2013), Palo Alto, California, pp. 268-278, 2013 [29].

There exist a number of approaches for detecting smells in source code to alert developers of their presence [8], [34], [48]. These approaches rely on structural information extracted from source code, for example, by means of constraints defined on some source code metrics. For instance, according to some existing approaches, such as DECOR [33], *LongMethod* or *LargeClass* smells are based on the size of the source code component in terms of LOC, whereas other smells like *ComplexClass* are based on the McCabe cyclomatic complexity [32]. Other smells, such as *Blob*, might use more complex rules.

Although existing approaches exhibit good detection accuracy, they still might not be adequate for detecting many of the smells described by Fowler [14]. In particular, there are some smells that, rather than being characterized by source code metrics or other information extracted from source code snapshots, are intrinsically characterized by how source code changes over time. For example, a *Parallel Inheritance* means that two or more class hierarchies evolve by adding code to both classes at the same time. Also, there are smells that are traditionally detected using structural information, where historical information can aid in capturing complementary, additionally useful properties. For example, a *Feature Envy*

Bad Code Smells - Where to Refactor

Hence, in some cases the detection of code smells shall make use of change history information. This has been introduced by HIST (Historical Information for Smell deTection).

Mining Version Histories for Detecting Code Smells

Pablo Palomba¹, Gabriele Bavota², Massimiliano Di Penta²,
Rocco Oliveto³, Denys Poshyvanyk⁴, Andrea De Lucia¹

¹University of Salerno, Fisciano (SA), Italy

²University of Salerno, Benevento, Italy

³University of Molise, Pesche (IS), Italy

⁴The College of William and Mary, Williamsburg, VA, USA

palomba@unisa.it, gbavota@unisa.it, dipenta@unisa.it,
rocco.oliveto@unimol.it, denys@cs.wm.edu, adelucia@unisa.it

ABSTRACT. Code smells are symptoms of poor design and implementation choices that may hinder code comprehension, and possibly increase change- and fault-proneness. While most of the detection techniques just rely on structural information, many code smells are intrinsically characterized by how code elements change over time. In this paper, we propose HIST (Historical Information for Smell deTection), an approach exploring change history information to detect instances of two different code smells, namely Divergent Groups and Shotgun Surgery. Parallel Inheritance, Blob, and Feature Envy. Whereas HIST is an empirical study. The first, conducted on twenty open source projects, aimed at assessing the accuracy of HIST in detecting instances of the code smells mentioned above. The results indicate that the precision of HIST ranges between 72% and 80%, and its recall ranges between 58% and 100%. Also, results of the first study indicate that HIST is able to identify code smells that cannot be detected by competitive approaches solely based on code analysis or a single system's snapshot. Then, we conducted a second study aimed at investigating to what extent the code smells detected by HIST (and by competitive code analysis techniques) reflect developers' perception of poor design and implementation choices. We involved twelve developers of four open source projects that recognized more than 75% of the code smell instances identified by HIST as actual design/implementation problems.

Index Terms. Code Smells, Mining Software Repositories, Empirical Studies.

1 INTRODUCTION

Code smells have been defined by Fowler [14] as symptoms of poor design and implementation choices. In some cases, such symptoms may originate from activities performed by developers while in a hurry, e.g., implementing urgent patches or simply making suboptimal choices. In other cases, smells come from some recurring, poor design solutions, also known as anti-patterns [9]. For example, a *Blob* is a large and complex class that centralizes the behavior of a portion of a system and only uses other classes as data holders. Blob classes can rapidly grow out of control, making it harder and harder for developers to understand them, to fix bugs, and to add new features.

Previous studies have found that smells hinder comprehension [1], and possibly increase changes and fault-proneness [23], [24]. In summary, smells need to be carefully detected and monitored and, whenever necessary, refactoring actions should be planned and performed to deal with them.

This paper is an extension of "Detecting Bad Smells in Source Code Using Change History Information" that appeared in the Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE 2013), Palo Alto, California, pp. 268-278, 2013 [29].

There exist a number of approaches for detecting smells in source code to alert developers of their presence [8], [34], [48]. These approaches rely on structural information extracted from source code, for example, by means of constraints defined on some source code metrics. For instance, according to some existing approaches, such as DECOR [33], *LongMethod* or *LargeClass* smells are based on the size of the source code component in terms of LOC, whereas other smells like *ComplexClass* are based on the McCabe cyclomatic complexity [32]. Other smells, such as *Blob*, might use more complex rules.

Although existing approaches exhibit good detection accuracy, they still might not be adequate for detecting many of the smells described by Fowler [14]. In particular, there are some smells that, rather than being characterized by source code metrics or other information extracted from source code snapshots, are intrinsically characterized by how source code changes over time. For example, a *Parallel Inheritance* means that two or more class hierarchies evolve by adding code to both classes at the same time. Also, there are smells that are traditionally detected using structural information, where historical information can aid in capturing complementary, additionally useful properties. For example, a *Feature Envy*

- HIST has been originally validated on twenty open-source software projects;
- According to the achieved results, HIST outperforms structural-based alternatives in terms of precision and recall;

Bad Code Smells - Where to Refactor

Hence, in some cases the detection of code smells shall make use of change history information. This has been introduced by HIST (Historical Information for Smell deTection).

Mining Version Histories for Detecting Code Smells

Pablo Palomba¹, Gabriele Bavota², Massimiliano Di Penta²,
Rocco Oliveto³, Denys Poshyvanyk⁴, Andrea De Lucia¹

¹University of Salerno, Fisciano (SA), Italy

²University of Salerno, Benevento, Italy

³University of Molise, Pesche (IS), Italy

⁴The College of William and Mary, Williamsburg, VA, USA

{palomba@unisa.it, gbavota@unisa.it, dipenta@unisa.it,
rocco.oliveto@unimol.it, denys@cs.wm.edu, adelacina@unisa.it}

ABSTRACT. Code smells are symptoms of poor design and implementation choices that may hinder code comprehension, and possibly increase change- and fault-proneness. While most of the detection techniques just rely on structural information, many code smells are intrinsically characterized by how code elements change over time. In this paper, we propose HIST (Historical Information for Smell deTection), an approach exploring change history information to detect instances of two different code smells, namely Divergent Groups, Shotgun Surgery, Parallel Inheritance, Blob, and Feature Envy. Whereas HIST is an empirical study. The first, conducted on twenty open source projects, aimed at assessing the accuracy of HIST in detecting instances of the code smells mentioned above. The results indicate that the precision of HIST ranges between 72% and 80%, and its recall ranges between 58% and 100%. Also, results of the first study indicate that HIST is able to identify code smells that cannot be detected by competitive approaches solely based on code analysis or a single system's snapshot. Then, we conducted a second study aimed at investigating to what extent the code smells detected by HIST (and by competitive code analysis techniques) reflect developers' perception of poor design and implementation choices. We involved twelve developers of four open source projects that recognized more than 75% of the code smell instances identified by HIST as actual design/implementation problems.

Index Terms.—Code Smells, Mining Software Repositories, Empirical Studies.

1 INTRODUCTION

Code smells have been defined by Fowler [14] as symptoms of poor design and implementation choices. In some cases, such symptoms may originate from activities performed by developers while in a hurry, e.g., implementing urgent patches or simply making suboptimal choices. In other cases, smells come from some recurring, poor design solutions, also known as anti-patterns [9]. For example, a *Blob* is a large and complex class that centralizes the behavior of a portion of a system and only uses other classes as data holders. Blob classes can rapidly grow out of control, making it harder and harder for developers to understand them, to fix bugs, and to add new features.

Previous studies have found that smells hinder comprehension [1], and possibly increase changes and fault-proneness [23], [24]. In summary, smells need to be carefully detected and monitored and, whenever necessary, refactoring actions should be planned and performed to deal with them.

This paper is an extension of "Detecting Bad Smells in Source Code Using Change History Information" that appeared in the Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015), Palo Alto, California, pp. 268-278, 2015 [29].

There exist a number of approaches for detecting smells in source code to alert developers of their presence [31], [34], [48]. These approaches rely on structural information extracted from source code, for example, by means of constraints defined on some source code metrics. For instance, according to some existing approaches, such as DECOR [33], *LongMethod* or *LargeClass* smells are based on the size of the source code component in terms of LOC, whereas other smells like *ComplexClass* are based on the McCabe cyclomatic complexity [32]. Other smells, such as *Blob*, might use more complex rules.

Although existing approaches exhibit good detection accuracy, they still might not be adequate for detecting many of the smells described by Fowler [14]. In particular, there are some smells that, rather than being characterized by source code metrics or other information extracted from source code snapshots, are intrinsically characterized by how source code changes over time. For example, a *Parallel Inheritance* means that two or more class hierarchies evolve by adding code to both classes at the same time. Also, there are smells that are traditionally detected using structural information, where historical information can aid in capturing complementary, additionally useful properties. For example, a *Feature Envy*

- HIST has been originally validated on twenty open-source software projects;
- According to the achieved results, HIST outperforms structural-based alternatives in terms of precision and recall;
- HIST and the alternative approaches are complementary: they can correctly detect different code smell instances.

Bad Code Smells - Where to Refactor

Hence, in some cases the detection of code smells shall make use of change history information. This has been introduced by HIST (Historical Information for Smell deTection).

Mining Version Histories for Detecting Code Smells

Pablo Palomba¹, Gabriele Bavota², Massimiliano Di Penta²,
Rocco Oliveto³, Denys Poshyvanyk⁴, Andrea De Lucia¹

¹University of Salerno, Fisciano (SA), Italy

²University of Salerno, Benevento, Italy

³University of Molise, Pesche (IS), Italy

⁴The College of William and Mary, Williamsburg, VA, USA

{palomba@unisa.it, gbavota@unisa.it, dipenta@unisa.it,
rocco.oliveto@unimol.it, denys@cs.wm.edu, adelacina@unisa.it}

ABSTRACT. Code smells are symptoms of poor design and implementation choices that may hinder code comprehension, and possibly increase change- and fault-proneness. While most of the detection techniques just rely on structural information, many code smells are intrinsically characterized by how code elements change over time. In this paper, we propose HIST (Historical Information for Smell deTection), an approach exploring change history information to detect instances of five different code smells, namely Divergent Groups, Shotgun Surgery, Parallel Inheritance, Blob, and Feature Envy. Whereas HIST is an empirical study, the first, conducted on twenty open source projects, aimed at assessing the accuracy of HIST in detecting instances of the code smells mentioned above. The results indicate that the precision of HIST ranges between 72% and 80%, and its recall ranges between 58% and 100%. Also, results of the first study indicate that HIST is able to identify code smells that cannot be detected by competitive approaches solely based on code analysis or a single system's snapshot. Then, we conducted a second study aimed at investigating to what extent the code smells detected by HIST (and by competitive code analysis techniques) reflect developers' perception of poor design and implementation choices. We involved twelve developers of four open source projects that recognized more than 75% of the code smell instances identified by HIST as actual design/implementation problems.

Index Terms. Code Smells, Mining Software Repositories, Empirical Studies.

1 INTRODUCTION

Code smells have been defined by Fowler [14] as symptoms of poor design and implementation choices. In some cases, such symptoms may originate from activities performed by developers while in a hurry, e.g., implementing urgent patches or simply making suboptimal choices. In other cases, smells come from some recurring, poor design solutions, also known as anti-patterns [9]. For example, a *Blob* is a large and complex class that centralizes the behavior of a portion of a system and only uses other classes as data holders. Blob classes can rapidly grow out of control, making it harder and harder for developers to understand them, to fix bugs, and to add new features.

Previous studies have found that smells hinder comprehension [1], and possibly increase changes and fault-proneness [23], [24]. In summary, smells need to be carefully detected and monitored and, whenever necessary, refactoring actions should be planned and performed to deal with them.

This paper is an extension of "Detecting Bad Smells in Source Code Using Change History Information" that appeared in the Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE 2013), Palo Alto, California, pp. 268-278, 2013 [29].

There exist a number of approaches for detecting smells in source code to alert developers of their presence [8], [34], [48]. These approaches rely on structural information extracted from source code, for example, by means of constraints defined on some source code metrics. For instance, according to some existing approaches, such as DECOR [33], *LongMethod* or *LargeClass* smells are based on the size of the source code component in terms of LOC, whereas other smells like *ComplexClass* are based on the McCabe cyclomatic complexity [32]. Other smells, such as *Blob*, might use more complex rules.

Although existing approaches exhibit good detection accuracy, they still might not be adequate for detecting many of the smells described by Fowler [14]. In particular, there are some smells that, rather than being characterized by source code metrics or other information extracted from source code snapshots, are intrinsically characterized by how source code changes over time. For example, a *Parallel Inheritance* means that two or more class hierarchies evolve by adding code to both classes at the same time. Also, there are smells that are traditionally detected using structural information, where historical information can aid in capturing complementary, additionally useful properties. For example, a *Feature Envy*

- HIST has been originally validated on twenty open-source software projects;
- According to the achieved results, HIST outperforms structural-based alternatives in terms of precision and recall;
- HIST and the alternative approaches are complementary: they can correctly detect different code smell instances.
- The complementarity would make hybrid approaches possible.

Bad Code Smells - Where to Refactor

Hence, in some cases the detection of code smells shall make use of change history information. This has been introduced by HIST (Historical Information for Smell deTection).

Mining Version Histories for Detecting Code Smells

Pablo Palomba¹, Gabriele Bavota², Massimiliano Di Penta²,
Rocco Oliveto³, Denys Poshyvanyk⁴, Andrea De Lucia¹

¹University of Salerno, Fisciano (SA), Italy

²University of Salerno, Benevento, Italy

³University of Molise, Pesche (IS), Italy

⁴The College of William and Mary, Williamsburg, VA, USA

palomba@unisa.it, gbavota@unisa.it, dipenta@unisa.it,
rocco.oliveto@unimol.it, denys@cs.wm.edu, adelucia@unisa.it

ABSTRACT. Code smells are symptoms of poor design and implementation choices that may hinder code comprehension, and possibly increase change- and fault-proneness. While most of the detection techniques just rely on structural information, many code smells are intrinsically characterized by how code elements change over time. In this paper, we propose HIST (Historical Information for Smell deTection), an approach exploring change history information to detect instances of five different code smells, namely Divergent Groups, Shotgun Surgery, Parallel Inheritance, Blob, and Feature Envy. Whereas HIST is an empirical study, the first, conducted on twenty open source projects, aimed at assessing the accuracy of HIST in detecting instances of the code smells mentioned above. The results indicate that the precision of HIST ranges between 72% and 80%, and its recall ranges between 58% and 100%. Also, results of the first study indicate that HIST is able to identify code smells that cannot be identified by competitive approaches solely based on code analysis or a single system's snapshot. Then, we conducted a second study aimed at investigating to what extent the code smells detected by HIST (and by competitive code analysis techniques) reflect developers' perception of poor design and implementation choices. We involved twelve developers of four open source projects that recognized more than 75% of the code smell instances identified by HIST as actual design/implementation problems.

Index Terms. Code Smells, Mining Software Repositories, Empirical Studies.

1 INTRODUCTION

Code smells have been defined by Fowler [14] as symptoms of poor design and implementation choices. In some cases, such symptoms may originate from activities performed by developers while in a hurry, e.g., implementing urgent patches or simply making suboptimal choices. In other cases, smells come from some recurring, poor design solutions, also known as anti-patterns [9]. For example, a *Blob* is a large and complex class that centralizes the behavior of a portion of a system and only uses other classes as data holders. Blob classes can rapidly grow out of control, making it harder and harder for developers to understand them, to fix bugs, and to add new features.

Previous studies have found that smells hinder comprehension [1] and possibly increase changes and fault-proneness [23] [24]. In summary, smells need to be carefully detected and monitored and, whenever necessary, refactoring actions should be planned and performed to deal with them.

This paper is an extension of "Detecting Bad Smells in Source Code Using Change History Information" that appeared in the Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE 2015), Palo Alto, California, pp. 268-278, 2015 [29].

There exist a number of approaches for detecting smells in source code to alert developers of their presence [8], [34], [48]. These approaches rely on structural information extracted from source code, for example, by means of constraints defined on some source code metrics. For instance, according to some existing approaches, such as DECOR [33], *LongMethod* or *LargeClass* smells are based on the size of the source code component in terms of LOC, whereas other smells like *ComplexClass* are based on the McCabe cyclomatic complexity [32]. Other smells, such as *Blob*, might use more complex rules.

Although existing approaches exhibit good detection accuracy, they still might not be adequate for detecting many of the smells described by Fowler [14]. In particular, there are some smells that, rather than being characterized by source code metrics or other information extracted from source code snapshots, are intrinsically characterized by how source code changes over time. For example, a *Parallel Inheritance* means that two or more class hierarchies evolve by adding code to both classes at the same time. Also, there are smells that are traditionally detected using structural information, where historical information can aid in capturing complementary, additionally useful properties. For example, a *Feature Envy*

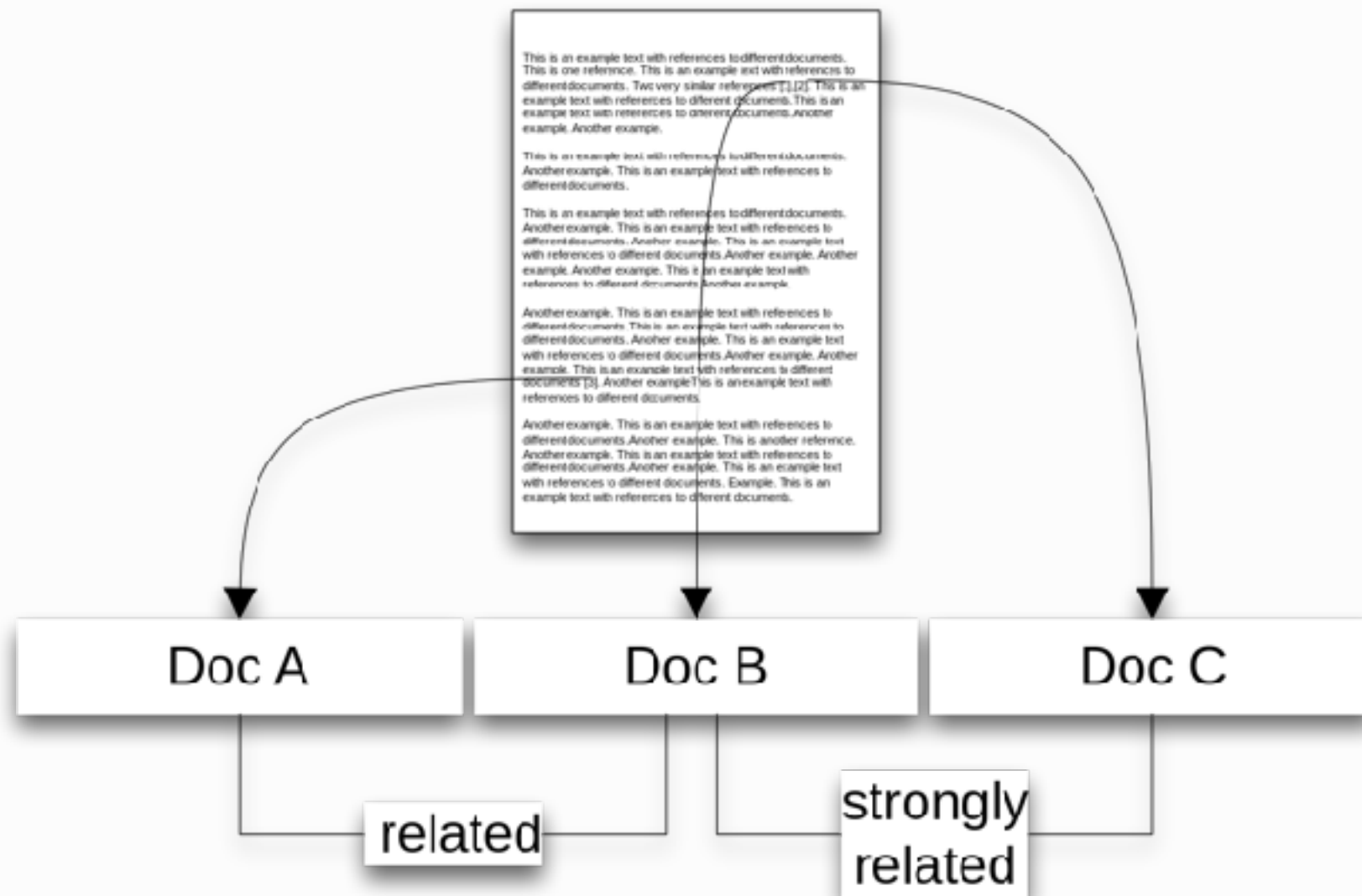
- HIST has been originally validated on twenty open-source software projects;
- According to the achieved results, HIST outperforms structural-based alternatives in terms of precision and recall;
- HIST and the alternative approaches are complementary: they can correctly detect different code smell instances.
- The complementarity would make hybrid approaches possible.
- The code smell instances given by HIST are perceived as more meaningful from developers with respect to those output by other detectors.

Bad Code Smells - Where to Refactor

What about **textual analysis**? DECOR introduced this concept for smell detection, yet it limited textual analysis to class and method names.

Bad Code Smells - Where to Refactor

What about **textual analysis**? DECOR introduced this concept for smell detection, yet it limited textual analysis to class and method names.

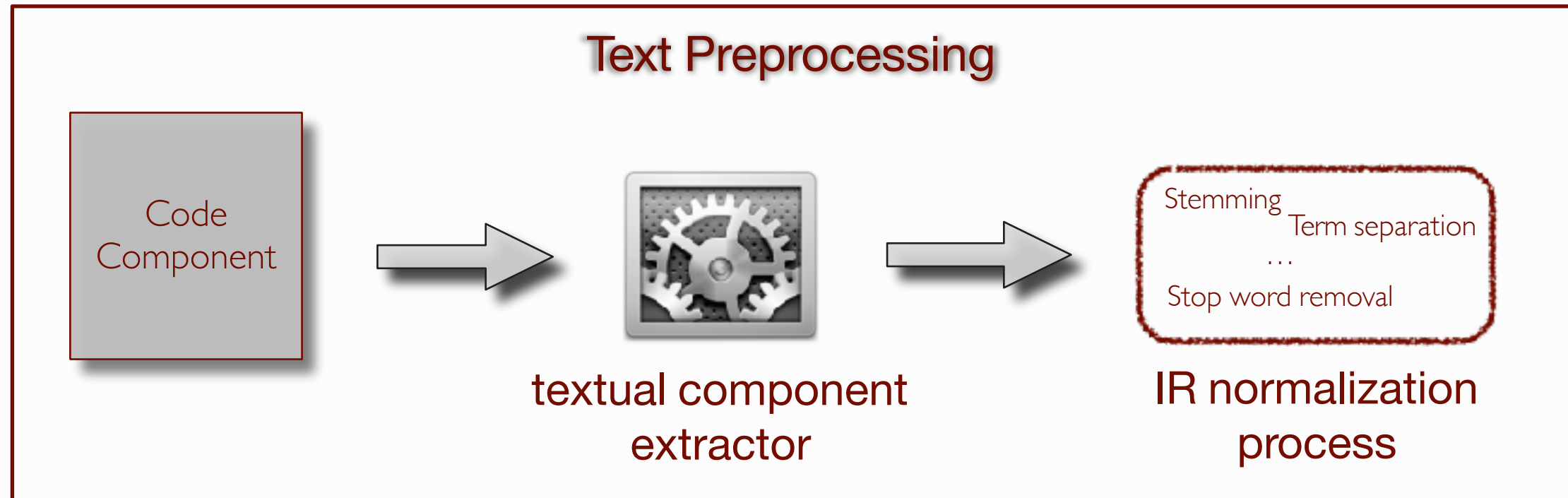


The conjecture is that, if there **unrelated text** in the source code, this may be symptom of the presence of some cohesion-related problems.

Bad Code Smells

Bad Code Smells - Where to Refactor

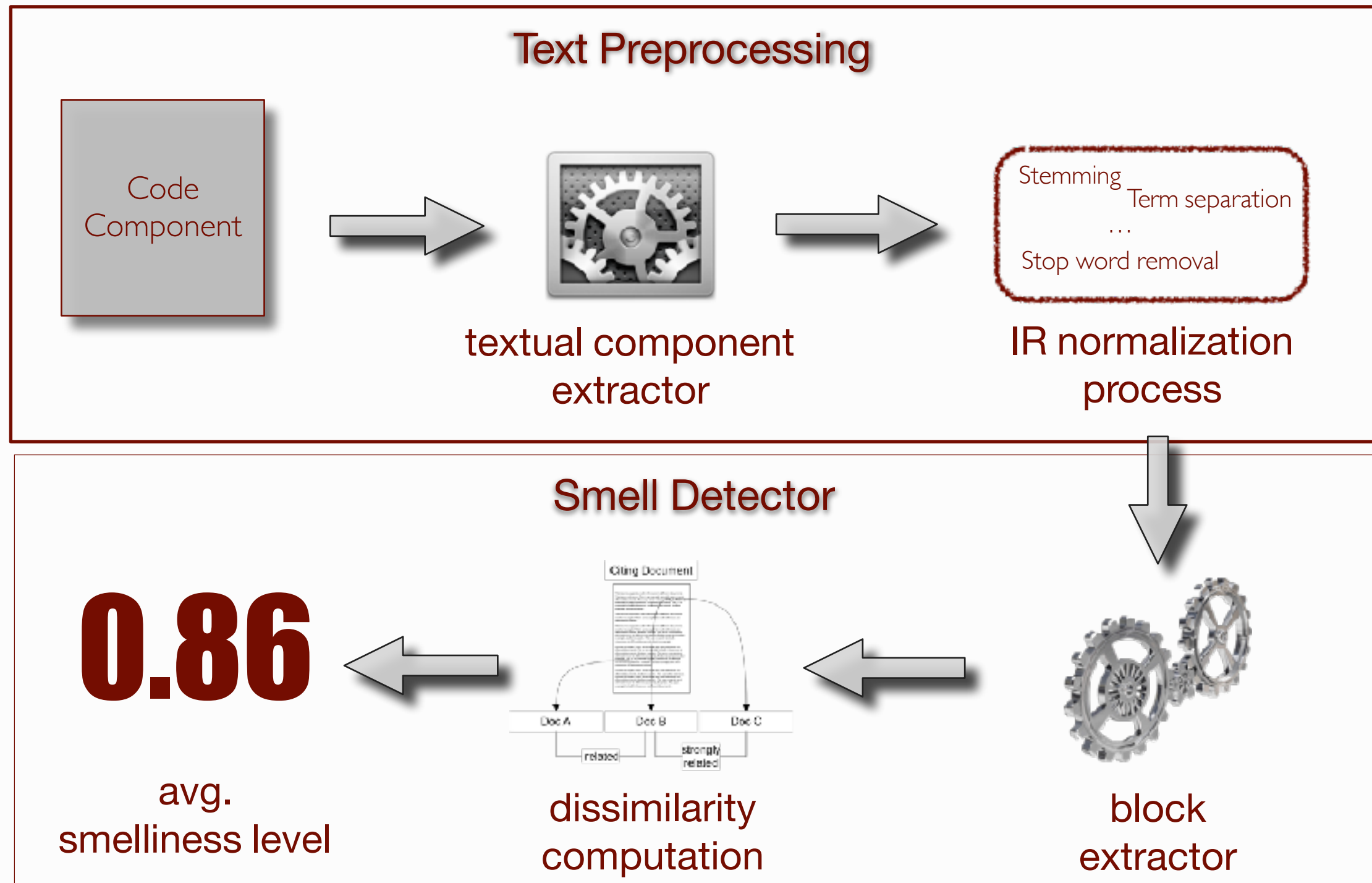
What about **textual analysis**? DECOR introduced this concept for smell detection, yet it limited textual analysis to class and method names.



Bad Code Smells

Bad Code Smells - Where to Refactor

What about **textual analysis**? DECOR introduced this concept for smell detection, yet it limited textual analysis to class and method names.



Bad Code Smells - Where to Refactor

What about **textual analysis**? DECOR introduced this concept for smell detection, yet it limited textual analysis to class and method names.

A Textual-based Technique for Smell Detection

Fabio Palomba¹, Annibale Panichella², Andrea De Lucia¹, Rocco Oliveto², Andy Zaidman³
¹University of Salento, Italy – ²Delft University of Technology, The Netherlands – ³University of Milan, Italy

Abstract—In this paper, we present TACO (Textual Analysis for Code Smell Detection), a technique that exploits textual analysis to detect a family of smells of different nature and different levels of granularity. We run TACO on 10 open source projects, comparing its performance with existing smell detectors purely based on structural information extracted from code components. The analysis of the results indicates that TACO's precision ranges between 67% and 77%, while its recall ranges between 72% and 84%. Also, TACO often outperforms alternative structural approaches confirming, once again, the usefulness of information that can be derived from the textual part of code components.

1. INTRODUCTION

Continuous change requests, strict and close deadlines, the need to preserve the quality of source code to ease maintenance are just some of the challenges that developers must face every day. In such a scenario, finding the solution that provides the maximum gain from each point of view is quite impossible. Very often, due to time constraints or absence of software design documentation, developers decide to set aside good programming guidelines and implement a new change request in the most straightforward way. This way of working under the original design of the system and introduces technical debt [1]. The erosion of the original design is generally represented by “poor design or implementation choices” [2], usually referred to as bad code smells (also named “code smells” or simply “smells”). Over the last decade, researchers investigated several aspects related to the presence of code smells, demonstrating (i) their relevance from the developer’s perspective [3], [4], (ii) their longevity [5], [6], [7], [8], [9], and (iii) their impact on non-functional properties of source code, such as program comprehension [10], change and fault proneness [11], [12], and, more in general, on maintainability [13], [14], [15], [16]. For these reasons the research community devoted a lot of effort to define methods to detect code smells in source code and, whenever possible, trigger refactoring operations [17], [18], [19], [20], [21], [22], [23]. These tools generally apply constraint-based detection rules defined on some source code metrics, i.e., the majority of existing approaches try to detect code smells through the analysis of structural properties of code components (e.g., methods).

Analyzing the catalogue of smells defined in the literature, it is easy to identify a specific family of smells that are represented by source code components with promiscuous responsibilities. For instance, a *Blob* is a giant class that centralizes the behavior of a portion of the system and has a lot of different responsibilities, while a *Feature Envy* refers to a method more related to a different class with respect the one it is actually in. Even if these smells, such as *Blob*, are

generally identified by considering structural properties of the code (see for instance [22]), there is still room for improving their detection by exploring other sources of information. For instance, Palomba et al. [24] recently proposed the use of historical information for detecting several bad smells, including *Blob*. However, components with promiscuous responsibilities can be identified also considering the textual coherence of the source code vocabulary (i.e., terms extracted from comments and identifiers). Previous studies have indicated that lack of coherence in the code vocabulary can be successfully used to identify poorly cohesive [25] or more complex [26] classes. Following the same underlying assumption, in this paper we aim at investigating to what extent textual analysis can be used to detect smells related to promiscuous responsibilities. It is worth noting that textual analysis has already been used in several software engineering tasks [27], [28], [29], including refactoring [30], [31], [32]. However, our goal is to define an approach able to detect a family of smells rather than recommending refactoring solution for a specific smell. To this aim, we define TACO (Textual Analysis for Code smell detection), a smell detector purely based on Information Retrieval (IR) methods. We instantiated TACO for detecting five code smells, i.e., *Long Method*, *Feature Envy*, *Blob*, *Promiscuous Package* and *Misplaced Class*. We conducted an empirical study involving 10 open source projects in order to (i) evaluate the accuracy of TACO when detecting code smells, and (ii) compare TACO with state-of-the-art structural-based detectors, namely DECOR [22], JDeodorant [23], and the approaches proposed in [33] and [34]. The results of our study indicate that TACO's precision ranges between 67% and 77% while its recall is between 72% and 84%. When compared with the alternative structural-based detectors, we experienced that most of the times TACO outperforms these existing approaches. Finally, we observed some complementarities between textual and structural information suggesting that better performance can be achieved by combining the two sources of information.

II. OVERVIEW AND RELATED WORK

Starting from the definition of design defects proposed in [24], [25], [26], [37], researchers have proposed semi-automated tools and techniques to detect code smells, such as ad-hoc manual inspection rules [38], tools to simulate code smells re-refactoring opportunities [39], [40]. Further studies proposed to detect code smells by (i) identifying key signatures that characterize particular bad smells using a set of thresholds based on the measurement of structural metrics (e.g., if Lines Of Code $\geq X$), (ii) conflicting the

- TACO has been instantiated on five code smell types: *Blob*, *Feature Envy*, *Long Method*, *Misplaced Class*, *Promiscuous Package*

Bad Code Smells - Where to Refactor

What about **textual analysis**? DECOR introduced this concept for smell detection, yet it limited textual analysis to class and method names.

A Textual-based Technique for Smell Detection

Fabio Palomba¹, Annibale Panichella², Andrea De Lucia¹, Rocco Oliveto², Andy Zaidman³
¹University of Salerno, Italy – ²Delft University of Technology, The Netherlands – ³University of Molise, Italy

Abstract—In this paper, we present TACO (Textual Analysis for Code Smell Detection), a technique that exploits textual analysis to detect a family of smells of different nature and different levels of granularity. We run TACO on 13 open source projects, comparing its performance with existing smell detectors purely based on structural information extracted from code components. The analysis of the results indicates that TACO's precision ranges between 67% and 77%, while its recall ranges between 72% and 84%. Also, TACO often outperforms alternative structural approaches confirming, once again, the usefulness of information that can be derived from the textual part of code components.

1. INTRODUCTION

Continuous change requests, strict and close deadlines, the need to preserve the quality of source code to ease maintenance are just some of the challenges that developers must face every day. In such a scenario, finding the solution that provides the maximum gain from each point of view is quite impossible. Very often, due to time constraints or absence of software design documentation, developers decide to set aside good programming guidelines and implement a new change request in the most straightforward way. This way of working under the original design of the system and introduces technical debt [1]. The erosion of the original design is generally represented by "poor design or implementation choices" [2], usually referred to as bad code smells (also named "code smells" or simply "smells"). Over the last decade, researchers investigated several aspects related to the presence of code smells, demonstrating (i) their relevance from the developer's perspective [3], [4], (ii) their longevity [5], [6], [7], [8], [9], and (iii) their impact on non-functional properties of source code, such as program comprehension [10], change and fault proneness [11], [12], and, more in general, on maintainability [13], [14], [15], [16]. For these reasons the research community devoted a lot of effort to define methods to detect code smells in source code and, whenever possible, trigger refactoring operations [7], [18], [19], [20], [21], [22], [23]. These tools generally apply constraint-based detection rules defined on some source code metrics, i.e., the majority of existing approaches try to detect code smells through the analysis of structural properties of code components (e.g., methods).

Analyzing the catalogue of smells defined in the literature, it is easy to identify a specific family of smells that are represented by source code components with promiscuous responsibilities. For instance, a *Blob* is a giant class that centralizes the behavior of a portion of the system and has a lot of different responsibilities, while a *Feature Envy* refers to a method more related to a different class with respect the one it is actually in. Even if these smells, such as *Blob*, are

generally identified by considering structural properties of the code (see for instance [22]), there is still room for improving their detection by exploring other sources of information. For instance, Palomba et al. [24] recently proposed the use of historical information for detecting several bad smells, including *Blob*. However, components with promiscuous responsibilities can be identified also considering the textual coherence of the source code vocabulary (i.e., terms extracted from comments and identifiers). Previous studies have indicated that lack of coherence in the code vocabulary can be successfully used to identify poorly cohesive [25] or more complex [26] classes. Following the same underlying assumption, in this paper we aim at investigating to what extent textual analysis can be used to detect smells related to promiscuous responsibilities. It is worth noting that textual analysis has already been used in several software engineering tasks [27], [28], [29], including refactoring [30], [31], [32]. However, our goal is to define an approach able to detect a family of smells rather than recommending refactoring solutions for a specific smell. To this aim, we define TACO (Textual Analysis for Code smell detection), a smell detector purely based on Information Retrieval (IR) methods. We instantiated TACO for detecting five code smells, i.e., *Long Method*, *Feature Envy*, *Blob*, *Promiscuous Package* and *Misplaced Class*. We conducted an empirical study involving 13 open source projects in order to (i) evaluate the accuracy of TACO when detecting code smells, and (ii) compare TACO with state-of-the-art structural-based detectors, namely DECOR [22], JDeodorant [23], and the approaches proposed in [33] and [34]. The results of our study indicate that TACO's precision ranges between 67% and 77% while its recall is between 72% and 84%. When compared with the alternative structural-based detectors, we experienced that most of the times TACO outperforms these existing approaches. Finally, we observed some complementarities between textual and structural information suggesting that better performance can be achieved by combining the two sources of information.

II. OVERVIEW AND RELATED WORK

Starting from the definition of design defects proposed in [24], [25], [26], [37], researchers have proposed semi-automated tools and techniques to detect code smells, such as ad-hoc manual inspection rules [38], tools to visualize code smells or refactoring opportunities [39], [40]. Further studies proposed to detect code smells by (i) identifying key syntagmas that characterize particular bad smells using a set of thresholds based on the measurement of structural metrics (e.g., if Lines Of Code $\geq X$), (ii) correlating the

- TACO has been instantiated on five code smell types: *Blob*, *Feature Envy*, *Long Method*, *Misplaced Class*, *Promiscuous Package*

Some more definitions needed

Misplaced Class. This code smell indicates a class that is in a package that contains other classes not related to it.

Bad Code Smells - Where to Refactor

What about **textual analysis**? DECOR introduced this concept for smell detection, yet it limited textual analysis to class and method names.

A Textual-based Technique for Smell Detection

Fabio Palomba¹, Annibale Panichella², Andrea De Lucia¹, Rocco Oliveto², Andy Zaidman³
¹University of Salento, Italy – ²Delft University of Technology, The Netherlands – ³University of Milan, Italy

Abstract—In this paper, we present TACO (Textual Analysis for Code Smell Detection), a technique that exploits textual analysis to detect a family of smells of different nature and different levels of granularity. We run TACO on 10 open source projects, comparing its performance with existing smell detectors purely based on structural information extracted from code components. The analysis of the results indicates that TACO's precision ranges between 67% and 77%, while its recall ranges between 72% and 84%. Also, TACO often outperforms alternative structural approaches confirming, once again, the usefulness of information that can be derived from the textual part of code components.

1. INTRODUCTION

Continuous change requests, strict and close deadlines, the need to preserve the quality of source code to ease maintenance are just some of the challenges that developers must face every day. In such a scenario, finding the solution that provides the maximum gain from each point of view is quite impossible. Very often, due to time constraints or absence of software design documentation, developers decide to set aside good programming guidelines and implement a new change request in the most straightforward way. This way of working under the original design of the system and introduces technical debt [1]. The erosion of the original design is generally represented by “poor design or implementation choices” [2], usually referred to as bad code smells (also named “code smells” or simply “smells”). Over the last decade, researchers investigated several aspects related to the presence of code smells, demonstrating (i) their relevance from the developers’ perspective [3], [4], (ii) their longevity [5], [6], [7], [8], [9], and (iii) their impact on non-functional properties of source code, such as program comprehension [10], change and fault proneness [11], [12], and, more in general, on maintainability [13], [14], [15], [16]. For these reasons the research community devoted a lot of effort to define methods to detect code smells in source code and, whenever possible, trigger refactoring operations [17], [18], [19], [20], [21], [22], [23]. These tools generally apply constraint-based detection rules defined on some source code metrics, i.e., the majority of existing approaches try to detect code smells through the analysis of structural properties of code components (e.g., methods).

Analyzing the catalogue of smells defined in the literature, it is easy to identify a specific family of smells that are represented by source code components with promiscuous responsibilities. For instance, a *Blob* is a giant class that centralizes the behavior of a portion of the system and has a lot of different responsibilities, while a *Feature Envy* refers to a method more related to a different class with respect the one it is actually in. Even if these smells, such as *Blob*, are

generally identified by considering structural properties of the code (see for instance [22]), there is still room for improving their detection by exploring other sources of information. For instance, Palomba et al. [24] recently proposed the use of historical information for detecting several bad smells, including *Blob*. However, components with promiscuous responsibilities can be identified also considering the textual coherence of the source code vocabulary (i.e., terms extracted from comments and identifiers). Previous studies have indicated that lack of coherence in the code vocabulary can be successfully used to identify poorly cohesive [25] or more complex [26] classes. Following the same underlying assumption, in this paper we start investigating to what extent textual analysis can be used to detect smells related to promiscuous responsibilities. It is worth noting that textual analysis has already been used in several software engineering tasks [27], [28], [29], including refactoring [30], [31], [32]. However, our goal is to define an approach able to detect a family of smells rather than recommending refactoring solution for a specific smell. To this aim, we define TACO (Textual Analysis for Code smell detection), a smell detector purely based on Information Retrieval (IR) methods. We instantiated TACO for detecting five code smells, i.e., *Long Method*, *Feature Envy*, *Blob*, *Promiscuous Package* and *Misplaced Class*. We conducted an empirical study involving 10 open source projects in order to (i) evaluate the accuracy of TACO when detecting code smells, and (ii) compare TACO with state-of-the-art structural-based detectors, namely DECOR [22], JDeodorant [23], and the approaches proposed in [33] and [34]. The results of our study indicate that TACO's precision ranges between 67% and 77% while its recall is between 72% and 84%. When compared with the alternative structural-based detectors, we experienced that most of the times TACO outperforms these existing approaches. Finally, we observed some complementarities between textual and structural information suggesting that better performance can be achieved by combining the two sources of information.

II. OVERVIEW AND RELATED WORK

Starting from the definition of design defects proposed in [24], [25], [26], [37], researchers have proposed semi-automated tools and techniques to detect code smells, such as ad-hoc manual inspection rules [38], tools to simulate code smells re-refactoring opportunities [39], [40]. Further studies proposed to detect code smells by (i) identifying key syntagmas that characterize particular bad smells using a set of thresholds based on the measurement of structural metrics (e.g., all Lines Of Code $\geq X$), (ii) conflicting the

- TACO has been instantiated on five code smell types: *Blob*, *Feature Envy*, *Long Method*, *Misplaced Class*, *Promiscuous Package*

Some more definitions needed

Misplaced Class. This code smell indicates a class that is in a package that contains other classes not related to it.

Promiscuous Package. This code smell arises when a package contains classes implementing too many features, making it too hard to understand and maintain.

Bad Code Smells

Bad Code Smells - Where to Refactor

What about **textual analysis**? DECOR introduced this concept for smell detection, yet it limited textual analysis to class and method names.

```
public void insert(User pUser){  
  
    connect = DBConnection.getConnection();  
  
    String sql = "INSERT INTO USER"  
        + "(login,first_name,last_name,password"  
        + ",email,cell,id_parent) " + "VALUES ("  
        + pUser.getLogin() + ","  
        + pUser.getFirstName() + ","  
        + pUser.getLastName() + ","  
        + pUser.getPassword() + ","  
        + pUser.getEmail() + ","  
        + pUser.getCell() + ","  
        + pUser.getIdParent() + ")";  
  
    String sql = "DELETE FROM USER "  
        + "WHERE id_user = "  
        + pUser.getId();
```

The example of Long Method

Bad Code Smells

Bad Code Smells - Where to Refactor

What about **textual analysis**? DECOR introduced this concept for smell detection, yet it limited textual analysis to class and method names.

```
public void insert(User pUser){  
    connect = DBConnection.getConnection();  
  
    String sql = "INSERT INTO USER"  
        + "(login,first_name,last_name,password"  
        + ",email,cell,id_parent) " + "VALUES ("  
        + pUser.getLogin() + ","  
        + pUser.getFirstName() + ","  
        + pUser.getLastName() + ","  
        + pUser.getPassword() + ","  
        + pUser.getEmail() + ","  
        + pUser.getCell() + ","  
        + pUser.getIdParent() + ")";  
  
    String sql = "DELETE FROM USER "  
        + "WHERE id_user = "  
        + pUser.getId();
```

The example of Long Method

Bad Code Smells

Bad Code Smells - Where to Refactor

What about **textual analysis**? DECOR introduced this concept for smell detection, yet it limited textual analysis to class and method names.

```
public void insert(User pUser){  
    connect = DBConnection.getConnection();  
  
    String sql = "INSERT INTO USER"  
        + "(login,first_name,last_name,password"  
        + ",email,cell,id_parent) " + "VALUES ("  
        + pUser.getLogin() + ","  
        + pUser.getFirstName() + ","  
        + pUser.getLastName() + ","  
        + pUser.getPassword() + ","  
        + pUser.getEmail() + ","  
        + pUser.getCell() + ","  
        + pUser.getIdParent() + ")";  
  
    String sql = "DELETE FROM USER "  
        + "WHERE id_user = "  
        + pUser.getId();
```

The example of Long Method

Bad Code Smells - Where to Refactor

What about **textual analysis**? DECOR introduced this concept for smell detection, yet it limited textual analysis to class and method names.

- TACO has been validated on 30 open-source software projects;

A Textual-based Technique for Smell Detection

Fabio Palomba¹, Annibale Panichella², Andrea De Lucia¹, Rocco Oliveto², Andy Zaidman³
¹University of Salerno, Italy – ²Delft University of Technology, The Netherlands – ³University of Milan, Italy

Abstract—In this paper, we present TACO (Textual Analysis for Code Smell Detection), a technique that exploits textual analysis to detect a family of smells of different nature and different levels of granularity. We run TACO on 30 open source projects, comparing its performance with existing smell detectors purely based on structural information extracted from code components. The analysis of the results indicates that TACO's precision ranges between 67% and 77%, while its recall ranges between 72% and 84%. Also, TACO often outperforms alternative structural approaches confirming, once again, the usefulness of information that can be derived from the textual part of code components.

1. INTRODUCTION

Continuous change requests, strict and close deadlines, the need to preserve the quality of source code to ease maintenance are just some of the challenges that developers must face every day. In such a scenario, finding the solution that provides the maximum gain from each point of view is quite impossible. Very often, due to time constraints or absence of software design documentation, developers decide to set aside good programming guidelines and implement a new change request in the most straightforward way. This way of working under the original design of the system and introduces technical debt [1]. The erosion of the original design is generally represented by “poor design or implementation choices” [2], usually referred to as bad code smells (also named “code smells” or simply “smells”). Over the last decade, researchers investigated several aspects related to the presence of code smells, demonstrating (i) their relevance from the developer’s perspective [3], [4], (ii) their longevity [5], [6], [7], [8], [9], and (iii) their impact on non-functional properties of source code, such as program comprehension [10], change and fault-proneness [11], [12], and, more in general, on maintainability [13], [14], [15], [16]. For these reasons the research community devoted a lot of effort to define methods to detect code smells in source code and, whenever possible, trigger refactoring operations [7], [18], [19], [20], [21], [22], [23]. These tools generally apply constraint-based detection rules defined on some source code metrics, i.e., the majority of existing approaches try to detect code smells through the analysis of structural properties of code components (e.g., methods).

Analyzing the catalogue of smells defined in the literature, it is easy to identify a specific family of smells that are represented by source code components with promiscuous responsibilities. For instance, a *Blob* is a giant class that centralizes the behavior of a portion of the system and has a lot of different responsibilities, while a *Feature Envy* refers to a method more related to a different class with respect the one it is actually in. Even if these smells, such as *Blob*, are

generally identified by considering structural properties of the code (see for instance [22]), there is still room for improving their detection by exploring other sources of information. For instance, Palomba et al. [24] recently proposed the use of historical information for detecting several bad smells, including *Blob*. However, components with promiscuous responsibilities can be identified also considering the textual coherence of the source code vocabulary (i.e., terms extracted from comments and identifiers). Previous studies have indicated that lack of coherence in the code vocabulary can be successfully used to identify poorly cohesive [25] or more complex [26] classes. Following the same underlying assumption, in this paper we aim at investigating to what extent textual analysis can be used to detect smells related to promiscuous responsibilities. It is worth noting that textual analysis has already been used in several software engineering tasks [27], [28], [29], including refactoring [30], [31], [32]. However, our goal is to define an approach able to detect a family of smells rather than recommending refactoring solution for a specific smell. To this aim, we define TACO (Textual Analysis for Code smell detection), a smell detector purely based on Information Retrieval (IR) methods. We instantiated TACO for detecting five code smells, i.e., *Long Method*, *Feature Envy*, *Blob*, *Promiscuous Package* and *Migrating Class*. We conducted an empirical study involving 30 open source projects in order to (i) evaluate the accuracy of TACO when detecting code smells, and (ii) compare TACO with state-of-the-art structural-based detectors, namely DECOR [22], JDeodorant [23], and the approaches proposed in [33] and [34]. The results of our study indicate that TACO's precision ranges between 67% and 77% while its recall is between 72% and 84%. When compared with the alternative structural-based detectors, we experienced that most of the times TACO outperforms these existing approaches. Finally, we observed some complementarities between textual and structural information suggesting that better performance can be achieved by combining the two sources of information.

II. OVERVIEW AND RELATED WORK

Starting from the definition of design defects proposed in [24], [25], [26], [37], researchers have proposed semi-automated tools and techniques to detect code smells, such as ad-hoc manual inspection rules [38], tools to visualize code smells or refactoring opportunities [39], [40]. Further studies proposed to detect code smells by (i) identifying key signatures that characterize particular bad smells using a set of thresholds based on the measurement of structural metrics (e.g., all Lines Of Code $\geq X$), (ii) conflicting the

Bad Code Smells - Where to Refactor

What about **textual analysis**? DECOR introduced this concept for smell detection, yet it limited textual analysis to class and method names.

A Textual-based Technique for Smell Detection

Fabio Palomba¹, Annibale Panichella², Andrea De Lucia¹, Rocco Oliveto², Andy Zaidman³
¹University of Salento, Italy – ²Delft University of Technology, The Netherlands – ³University of Milan, Italy

Abstract—In this paper, we present TACO (Textual Analysis for Code Smell Detection), a technique that exploits textual analysis to detect a family of smells of different nature and different levels of granularity. We run TACO on 30 open source projects, comparing its performance with existing smell detectors purely based on structural information extracted from code components. The analysis of the results indicates that TACO's precision ranges between 67% and 77%, while its recall ranges between 72% and 84%. Also, TACO often outperforms alternative structural approaches confirming, once again, the usefulness of information that can be derived from the textual part of code components.

1. INTRODUCTION

Continuous change requests, strict and close deadlines, the need to preserve the quality of source code to ease maintenance are just some of the challenges that developers must face every day. In such a scenario, finding the solution that provides the maximum gain from each point of view is quite impossible. Very often, due to time constraints or absence of software design documentation, developers decide to set aside good programming guidelines and implement a new change request in the most straightforward way. This way of working under the original design of the system and introduces technical debt [1]. The erosion of the original design is generally represented by “poor design or implementation choices” [2], usually referred to as bad code smells (also named “code smells” or simply “smells”). Over the last decade, researchers investigated several aspects related to the presence of code smells, demonstrating (i) their relevance from the developer’s perspective [3], [4], (ii) their longevity [5], [6], [7], [8], [9], and (iii) their impact on non-functional properties of source code, such as program comprehension [10], change and fault-proneness [11], [12], and, more in general, on maintainability [13], [14], [15], [16]. For these reasons the research community devoted a lot of effort to define methods to detect code smells in source code and, whenever possible, trigger refactoring operations [7], [18], [19], [20], [21], [22], [23]. These tools generally apply constraint-based detection rules defined on some source code metrics, i.e., the majority of existing approaches try to detect code smells through the analysis of structural properties of code components (e.g., methods).

Analyzing the catalogue of smells defined in the literature, it is easy to identify a specific family of smells that are represented by source code components with promiscuous responsibilities. For instance, a *Blob* is a giant class that centralizes the behavior of a portion of the system and has a lot of different responsibilities, while a *Feature Envy* refers to a method more related to a different class with respect the one it is actually in. Even if these smells, such as *Blob*, are

generally identified by considering structural properties of the code (see for instance [22]), there is still room for improving their detection by exploring other sources of information. For instance, Palomba et al. [24] recently proposed the use of historical information for detecting several bad smells, including *Blob*. However, components with promiscuous responsibilities can be identified also considering the textual coherence of the source code vocabulary (i.e., terms extracted from comments and identifiers). Previous studies have indicated that lack of coherence in the code vocabulary can be successfully used to identify poorly cohesive [25] or more complex [26] classes. Following the same underlying assumption, in this paper we aim at investigating to what extent textual analysis can be used to detect smells related to promiscuous responsibilities. It is worth noting that textual analysis has already been used in several software engineering tasks [27], [28], [29], including refactoring [30], [31], [32]. However, our goal is to define an approach able to detect a family of smells rather than recommending refactoring solution for a specific smell. To this aim, we define TACO (Textual Analysis for Code smell detection), a smell detector purely based on Information Retrieval (IR) methods. We instantiated TACO for detecting five code smells, i.e., *Long Method*, *Feature Envy*, *Blob*, *Promiscuous Package* and *Migrating Class*. We conducted an empirical study involving 30 open source projects in order to (i) evaluate the accuracy of TACO when detecting code smells, and (ii) compare TACO with state-of-the-art structural-based detectors, namely DECOR [22], JDeodorant [23], and the approaches proposed in [33] and [34]. The results of our study indicate that TACO's precision ranges between 67% and 77% while its recall is between 72% and 84%. When compared with the alternative structural-based detectors, we experienced that most of the times TACO outperforms these existing approaches. Finally, we observed some complementarities between textual and structural information suggesting that better performance can be achieved by combining the two sources of information.

II. OVERVIEW AND RELATED WORK

Starting from the definition of design defects proposed in [24], [25], [26], [37], researchers have proposed semi-automated tools and techniques to detect code smells, such as ad-hoc manual inspection rules [38], tools to simulate code smells to refactoring opportunities [39], [40]. Further studies proposed to detect code smells by (i) identifying key signatures that characterize particular bad smells using a set of thresholds based on the measurement of structural metrics (e.g., if Lines Of Code $\geq X$), (ii) conflicting the

- TACO has been validated on 30 open-source software projects;
- According to the achieved results, TACO outperforms structural-based alternatives in terms of precision and recall;

Bad Code Smells - Where to Refactor

What about **textual analysis**? DECOR introduced this concept for smell detection, yet it limited textual analysis to class and method names.

A Textual-based Technique for Smell Detection

Fabio Palomba¹, Annibale Panichella², Andrea De Lucia¹, Rocco Oliveto², Andy Zaidman³
¹University of Salento, Italy – ²Delft University of Technology, The Netherlands – ³University of Milan, Italy

Abstract—In this paper, we present TACO (Textual Analysis for Code Smell Detection), a technique that exploits textual analysis to detect a family of smells of different nature and different levels of granularity. We run TACO on 30 open source projects, comparing its performance with existing smell detectors purely based on structural information extracted from code components. The analysis of the results indicates that TACO's precision ranges between 67% and 77%, while its recall ranges between 72% and 84%. Also, TACO often outperforms alternative structural approaches confirming, once again, the usefulness of information that can be derived from the textual part of code components.

1. INTRODUCTION

Continuous change requests, strict and close deadlines, the need to preserve the quality of source code to ease maintenance are just some of the challenges that developers must face every day. In such a scenario, finding the solution that provides the maximum gain from each point of view is quite impossible. Very often, due to time constraints or absence of software design documentation, developers decide to set aside good programming guidelines and implement a new change request in the most straightforward way. This way of working makes the original design of the system and introduces technical debt [1]. The erosion of the original design is generally represented by “poor design or implementation choices” [2], usually referred to as bad code smells (also named “code smells” or simply “smells”). Over the last decade, researchers investigated several aspects related to the presence of code smells, demonstrating (i) their relevance from the developers’ perspective [3], [4], (ii) their longevity [5], [6], [7], [8], [9], and (iii) their impact on non-functional properties of source code, such as program comprehension [10], change and fault-proneness [11], [12], and, more in general, on maintainability [13], [14], [15], [16]. For these reasons the research community devoted a lot of effort to define methods to detect code smells in source code and, whenever possible, trigger refactoring operations [7], [18], [19], [20], [21], [22], [23]. These tools generally apply constraint-based detection rules defined on some source code metrics, i.e., the majority of existing approaches try to detect code smells through the analysis of structural properties of code components (e.g., methods).

Analyzing the catalogue of smells defined in the literature, it is easy to identify a specific family of smells that are represented by source code components with promiscuous responsibilities. For instance, a *Blob* is a giant class that centralizes the behavior of a portion of the system and has a lot of different responsibilities, while a *Feature Envy* refers to a method more related to a different class with respect the one it is actually in. Even if these smells, such as *Blob*, are

generally identified by considering structural properties of the code (see for instance [22]), there is still room for improving their detection by exploring other sources of information. For instance, Palomba et al. [24] recently proposed the use of historical information for detecting several bad smells, including *Blob*. However, components with promiscuous responsibilities can be identified also considering the textual coherence of the source code vocabulary (i.e., terms extracted from comments and identifiers). Previous studies have indicated that lack of coherence in the code vocabulary can be successfully used to identify poorly cohesive [25] or more complex [26] classes. Following the same underlying assumption, in this paper we aim at investigating to what extent textual analysis can be used to detect smells related to promiscuous responsibilities. It is worth noting that textual analysis has already been used in several software engineering tasks [27], [28], [29], including refactoring [30], [31], [32]. However, our goal is to define an approach able to detect a family of smells rather than recommending refactoring solutions for a specific smell. To this aim, we define TACO (Textual Analysis for Code smell detection), a smell detector purely based on Information Retrieval (IR) methods. We instantiated TACO for detecting five code smells, i.e., *Long Method*, *Feature Envy*, *Blob*, *Promiscuous Package* and *Migrating Class*. We conducted an empirical study involving 30 open source projects in order to (i) evaluate the accuracy of TACO when detecting code smells, and (ii) compare TACO with state-of-the-art structural-based detectors, namely DECOR [22], JDeodorant [23], and the approaches proposed in [33] and [34]. The results of our study indicate that TACO's precision ranges between 67% and 77% while its recall is between 72% and 84%. When compared with the alternative structural-based detectors, we experienced that most of the times TACO outperforms these existing approaches. Finally, we observed some complementarities between textual and structural information suggesting that better performance can be achieved by combining the two sources of information.

II. OVERVIEW AND RELATED WORK

Starting from the definition of design defects proposed in [24], [25], [26], [37], researchers have proposed semi-automated tools and techniques to detect code smells, such as ad-hoc manual inspection rules [38], tools to visualize code smells as refactoring opportunities [39], [40]. Further studies proposed to detect code smells by (i) identifying key syntactic constructs that characterize particular bad smells using a set of thresholds based on the measurement of structural metrics (e.g., all Lines Of Code $\geq X$), (ii) conflicting the

- TACO has been validated on 30 open-source software projects;
- According to the achieved results, TACO outperforms structural-based alternatives in terms of precision and recall;
- TACO and the alternative approaches are complementary: they can correctly detect different code smell instances.

Bad Code Smells - Where to Refactor

What about **textual analysis**? DECOR introduced this concept for smell detection, yet it limited textual analysis to class and method names.

A Textual-based Technique for Smell Detection

Fabio Palomba¹, Annibale Panichella², Andrea De Lucia¹, Rocco Oliveto², Andy Zaidman³
¹University of Salerno, Italy – ²Delft University of Technology, The Netherlands – ³University of Milan, Italy

Abstract—In this paper, we present TACO (Textual Analysis for Code Smell Detection), a technique that exploits textual analysis to detect a family of smells of different nature and different levels of granularity. We run TACO on 30 open source projects, comparing its performance with existing smell detectors purely based on structural information extracted from code components. The analysis of the results indicates that TACO's precision ranges between 67% and 77%, while its recall ranges between 72% and 84%. Also, TACO often outperforms alternative structural approaches confirming, once again, the usefulness of information that can be derived from the textual part of code components.

1. INTRODUCTION

Continuous change requests, strict and close deadlines, the need to preserve the quality of source code to ease maintenance are just some of the challenges that developers must face every day. In such a scenario, finding the solution that provides the maximum gain from each point of view is quite impossible. Very often, due to time constraints or absence of software design documentation, developers decide to set aside good programming guidelines and implement a new change request in the most straightforward way. This way of working makes the original design of the system and introduces technical debt [1]. The erosion of the original design is generally represented by “poor design or implementation choices” [2], usually referred to as bad code smells (also named “code smells” or simply “smells”). Over the last decade, researchers investigated several aspects related to the presence of code smells, demonstrating (i) their relevance from the developer’s perspective [3], [4], (ii) their longevity [5], [6], [7], [8], [9], and (iii) their impact on non-functional properties of source code, such as program comprehension [10], change and fault-proneness [11], [12], and, more in general, on maintainability [13], [14], [15], [16]. For these reasons the research community devoted a lot of effort to define methods to detect code smells in source code and, whenever possible, trigger refactoring operations [7], [18], [19], [20], [21], [22], [23]. These tools generally apply constraint-based detection rules defined on some source code metrics, i.e., the majority of existing approaches try to detect code smells through the analysis of structural properties of code components (e.g., methods).

Analyzing the catalogue of smells defined in the literature, it is easy to identify a specific family of smells that are represented by source code components with promiscuous responsibilities. For instance, a Blob is a giant class that centralizes the behavior of a portion of the system and has a lot of different responsibilities, while a Feature Envy refers to a method more related to a different class with respect the one it is actually in. Even if these smells, such as Blob, are

generally identified by considering structural properties of the code (see for instance [22]), there is still room for improving their detection by exploring other sources of information. For instance, Palomba et al. [24] recently proposed the use of historical information for detecting several bad smells, including Blobs. However, components with promiscuous responsibilities can be identified also considering the textual coherence of the source code vocabulary (i.e., terms extracted from comments and identifiers). Previous studies have indicated that lack of coherence in the code vocabulary can be successfully used to identify poorly cohesive [25] or more complex [26] classes. Following the same underlying assumption, in this paper we aim at investigating to what extent textual analysis can be used to detect smells related to promiscuous responsibilities. It is worth noting that textual analysis has already been used in several software engineering tasks [27], [28], [29], including refactoring [30], [31], [32]. However, our goal is to define an approach able to detect a family of smells rather than recommending refactoring solutions for a specific smell. To this aim, we define TACO (Textual Analysis for Code smell detection), a smell detector purely based on Information Retrieval (IR) methods. We instantiated TACO for detecting five code smells, i.e., Long Method, Feature Envy, Blob, Promiscuous Package and Migrating Class. We conducted an empirical study involving 30 open source projects in order to (i) evaluate the accuracy of TACO when detecting code smells, and (ii) compare TACO with state-of-the-art structural-based detectors, namely DECOR [22], JDeodorant [23], and the approaches proposed in [33] and [34]. The results of our study indicate that TACO's precision ranges between 67% and 77% while its recall is between 72% and 84%. When compared with the alternative structural-based detectors, we experienced that most of the times TACO outperforms these existing approaches. Finally, we observed some complementarities between textual and structural information suggesting that better performance can be achieved by combining the two sources of information.

II. OVERVIEW AND RELATED WORK

Starting from the definition of design defects proposed in [24], [25], [26], [37], researchers have proposed semi-automated tools and techniques to detect code smells, such as ad-hoc manual inspection rules [38], tools to visualize code smells and refactoring opportunities [39], [40]. Further studies proposed to detect code smells by (i) identifying key syntagmas that characterize particular bad smells using a set of thresholds based on the measurement of structural metrics (e.g., all Lines Of Code $\geq X$), (ii) conflicting the

- TACO has been validated on 30 open-source software projects;
- According to the achieved results, TACO outperforms structural-based alternatives in terms of precision and recall;
- TACO and the alternative approaches are complementary: they can correctly detect different code smell instances.
- The complementarity would make hybrid approaches possible.

Bad Code Smells - Where to Refactor

What about **textual analysis**? DECOR introduced this concept for smell detection, yet it limited textual analysis to class and method names.

A Textual-based Technique for Smell Detection

Fabio Palomba¹, Annibale Panichella², Andrea De Lucia¹, Rocco Oliveto², Andy Zaidman³
¹University of Salerno, Italy – ²Delft University of Technology, The Netherlands – ³University of Milan, Italy

Abstract—In this paper, we present TACO (Textual Analysis for Code Smell Detection), a technique that exploits textual analysis to detect a family of smells of different nature and different levels of granularity. We run TACO on 30 open source projects, comparing its performance with existing smell detectors purely based on structural information extracted from code components. The analysis of the results indicates that TACO's precision ranges between 67% and 77%, while its recall ranges between 72% and 84%. Also, TACO often outperforms alternative structural approaches confirming, once again, the usefulness of information that can be derived from the textual part of code components.

1. INTRODUCTION

Continuous change requests, strict and close deadlines, the need to preserve the quality of source code to ease maintenance are just some of the challenges that developers must face every day. In such a scenario, finding the solution that provides the maximum gain from each point of view is quite impossible. Very often, due to time constraints or absence of software design documentation, developers decide to set aside good programming guidelines and implement a new change request in the most straightforward way. This way of working makes the original design of the system and introduces technical debt [1]. The erosion of the original design is generally represented by “poor design or implementation choices” [2], usually referred to as bad code smells (also named “code smells” or simply “smells”). Over the last decade, researchers investigated several aspects related to the presence of code smells, demonstrating (i) their relevance from the developers’ perspective [3], [4], (ii) their longevity [5], [6], [7], [8], [9], and (iii) their impact on non-functional properties of source code, such as program comprehension [10], change and fault-proneness [11], [12], and, more in general, on maintainability [13], [14], [15], [16]. For these reasons the research community devoted a lot of effort to define methods to detect code smells in source code and, whenever possible, trigger refactoring operations [7], [18], [19], [20], [21], [22], [23]. These tools generally apply constraint-based detection rules defined on some source code metrics, i.e., the majority of existing approaches try to detect code smells through the analysis of structural properties of code components (e.g., methods).

Analyzing the catalogue of smells defined in the literature, it is easy to identify a specific family of smells that are represented by source code components with promiscuous responsibilities. For instance, a *Blob* is a giant class that centralizes the behavior of a portion of the system and has a lot of different responsibilities, while a *Feature Envy* refers to a method more related to a different class with respect the one it is actually in. Even if these smells, such as *Blob*, are

generally identified by considering structural properties of the code (see for instance [22]), there is still room for improving their detection by exploring other sources of information. For instance, Palomba et al. [24] recently proposed the use of historical information for detecting several bad smells, including *Blob*. However, components with promiscuous responsibilities can be identified also considering the textual coherence of the source code vocabulary (i.e., terms extracted from comments and identifiers). Previous studies have indicated that lack of coherence in the code vocabulary can be successfully used to identify poorly cohesive [25] or more complex [26] classes. Following the same underlying assumption, in this paper we aim at investigating to what extent textual analysis can be used to detect smells related to promiscuous responsibilities. It is worth noting that textual analysis has already been used in several software engineering tasks [27], [28], [29], including refactoring [30], [31], [32]. However, our goal is to define an approach able to detect a family of smells rather than recommending refactoring solutions for a specific smell. To this aim, we define TACO (Textual Analysis for Code smell detection), a smell detector purely based on Information Retrieval (IR) methods. We instantiated TACO for detecting five code smells, i.e., *Long Method*, *Feature Envy*, *Blob*, *Procrastinate Package* and *Misplaced Class*. We conducted an empirical study involving 30 open source projects in order to (i) evaluate the accuracy of TACO when detecting code smells, and (ii) compare TACO with state-of-the-art structural-based detectors, namely DECOR [22], JDeodorant [23], and the approaches proposed in [33] and [34]. The results of our study indicate that TACO's precision ranges between 67% and 77% while its recall is between 72% and 84%. When compared with the alternative structural-based detectors, we experienced that most of the times TACO outperforms these existing approaches. Finally, we observed some complementarities between textual and structural information suggesting that better performance can be achieved by combining the two sources of information.

II. OVERVIEW AND RELATED WORK

Starting from the definition of design defects proposed in [24], [25], [26], [37], researchers have proposed semi-automated tools and techniques to detect code smells, such as ad-hoc manual inspection rules [38], tools to visualize code smells and refactoring opportunities [39], [40]. Further studies proposed to detect code smells by (i) identifying key syntactic constructs that characterize particular bad smells using a set of thresholds based on the measurement of structural metrics (e.g., all Lines Of Code $\geq X$); (ii) conflicting the

- TACO has been validated on 30 open-source software projects;
- According to the achieved results, TACO outperforms structural-based alternatives in terms of precision and recall;
- TACO and the alternative approaches are complementary: they can correctly detect different code smell instances.
- The complementarity would make hybrid approaches possible.
- The code smell instances given by TACO are perceived as more meaningful from developers, which are more able to identify correct refactoring operations.

Bad Code Smells - Issues and Challenges

All code smell detection techniques discussed before have some key common limitations that potentially make them unsuitable for developers.

Bad Code Smells - Issues and Challenges

All code smell detection techniques discussed before have some key common limitations that potentially make them unsuitable for developers.

1

Subjectiveness. Code smells output by different detectors are subjectively interpreted by developers, i.e., not all smells are really smells!

Bad Code Smells - Issues and Challenges

All code smell detection techniques discussed before have some key common limitations that potentially make them unsuitable for developers.

1

Subjectiveness. Code smells output by different detectors are subjectively interpreted by developers, i.e., not all smells are really smells!

2

Agreement. The agreement on code smell candidates between detectors is generally low to very low, i.e., who's correct?

Bad Code Smells - Issues and Challenges

All code smell detection techniques discussed before have some key common limitations that potentially make them unsuitable for developers.

1

Subjectiveness. Code smells output by different detectors are subjectively interpreted by developers, i.e., not all smells are really smells!

2

Agreement. The agreement on code smell candidates between detectors is generally low to very low, i.e., who's correct?

3

Thresholds. All available detection techniques rely on the definition of some thresholds to discriminate which are the smelly elements, i.e., how to select them?

Bad Code Smells - Issues and Challenges

All code smell detection techniques discussed before have some key common limitations that potentially make them unsuitable for developers.

1

Subjectiveness. Code smells output by different detectors are subjectively interpreted by developers, i.e., not all smells are really smells!

2

Agreement. The agreement on code smell candidates between detectors is generally low to very low, i.e., who's correct?

3

Thresholds. All available detection techniques rely on the definition of some thresholds to discriminate which are the smelly elements, i.e., how to select them?

Question: Is there a way/technique that can lead to solve them all?

Bad Code Smells - Machine Learning-based Code Smell Detection

Theoretically speaking, machine learning has the potential to address all the limitations of currently available code smell detection techniques.

Bad Code Smells - Machine Learning-based Code Smell Detection

Theoretically speaking, machine learning has the potential to address all the limitations of currently available code smell detection techniques.

1

Subjectiveness. Machine learning can be used to learn from the past what is actually relevant for developers.

Bad Code Smells - Machine Learning-based Code Smell Detection

Theoretically speaking, machine learning has the potential to address all the limitations of currently available code smell detection techniques.

1

Subjectiveness. Machine learning can be used to learn from the past what is actually relevant for developers.

2

Agreement. Machine learning can be used to put different metrics together, in order to better characterize smelly elements.

Bad Code Smells - Machine Learning-based Code Smell Detection

Theoretically speaking, machine learning has the potential to address all the limitations of currently available code smell detection techniques.

1

Subjectiveness. Machine learning can be used to learn from the past what is actually relevant for developers.

2

Agreement. Machine learning can be used to put different metrics together, in order to better characterize smelly elements.

3

Thresholds. Machine learning models do not require thresholds, and so one can use a model to avoid the specification of unknown parameters.

Bad Code Smells - Machine Learning-based Code Smell Detection

Theoretically speaking, machine learning has the potential to address all the limitations of currently available code smell detection techniques.

1

Subjectiveness. Machine learning can be used to learn from the past what is actually relevant for developers.

2

Agreement. Machine learning can be used to put different metrics together, in order to better characterize smelly elements.

3

Thresholds. Machine learning models do not require thresholds, and so one can use a model to avoid the specification of unknown parameters.

Question: Does it work?

Bad Code Smells - Machine Learning-based Code Smell Detection

Theoretically speaking, machine learning has the potential to address all the limitations of currently available code smell detection techniques.

- The composition of the training set can bias the performance by up to 90%.

Detecting Code Smells using Machine Learning Techniques: Are We There Yet?

Dario Di Nucci^{1,2}, Fabio Palomba³, Damir A. Tambur⁴, Alexander Serebrenik⁴, Andrei De Lucia¹

¹University of Salerno, Italy – ²Vrije Universiteit Brussel, Belgium

³University of Zurich, Switzerland – ⁴Radboud University of Technology, The Netherlands

Abstract—Code smells are symptoms of poor design and implementation choices weighing heavily on the quality of produced source code. During the last decades several code smell detection tools have been proposed. However, the literature shows that the results of these tools can be subjective and are intrinsically tied to the nature and approach of the detection. In a recent work Arcelli Fontana et al. [1] proposed the use of Machine Learning (ML) techniques for code smell detection, possibly solving the issue of tool subjectivity giving us a better ability to discern between smelly and non-smelly source code elements. While this work opened a new perspective for code smell detection, in the context of our research we found a number of possible limitations that might threaten the results of this study. The most important issue is related to the metric distribution of smelly instances in the used dataset, which is strongly different from the one of non-smelly instances. In this work, we investigate this issue and our findings show that the high performance achieved in the study by Arcelli Fontana et al. was in fact due to the specific dataset employed rather than the actual capabilities of machine learning techniques for code smell detection.

Index Terms—Code Smells; Machine Learning; Empirical Studies; Replication Study.

1. INTRODUCTION

Nowadays, the complexity of software systems is growing fast and software companies are required to continuously update their source code [2]. These continuous changes frequently occur under time pressure and lead developers to set aside good programming practices and principles in order to deliver the most appropriate but still immature product in the shortest time possible [3]–[5]. This process can often result in the introduction of so-called *technical debt* [6], design problems likely to have negative consequences during the system maintenance and evolution.

One of the symptoms of the technical debt are *code smells* [7], suboptimal design decisions applied by developers that can negatively affect the overall maintainability of a software system. Over the last decades, the research community heavily investigated (i) how code smells are introduced [8], [9], (ii) how they evolve [10]–[13], (iii) what is their effect on program comprehension [14], [15] as well as on the change and bug proneness of the affected source code elements [16], [17], and (iv) the perception and ability of developers to deal with them [18]–[20].

Moreover, several code smell detection have been proposed [21], [22]: the detectors mainly differ in the underlying algorithm (e.g., metric-based [23]–[26] vs. search-based tech-

niques [27], [28]) and for the specific metric types considered (e.g., product metrics [23], [24] vs. process metrics [25]).

Despite the good performance shown by the detectors, recent studies highlight a number of important limitations threatening adoption of the detectors in practice [21], [29]. In the first place, code smells detected by existing approaches can be subjectively perceived and interpreted by developers [30], [31]. Secondly, the agreement between the detectors is low [32], meaning that different tools can identify the smelliness of different code elements. Last, but not least, most of the current detectors require the specification of thresholds that allow them to distinguish smelly and non-smelly instances [21], as a consequence, the selection of thresholds strongly influence the detectors' performance.

To overcome these limitations, machine learning (ML) techniques are being adopted to detect code smells [1]. Usually a supervised method is exploited, i.e., a set of independent variables (aka. *features*) are used to determine the value of a dependent variable (i.e., presence of a smell or degree of the smelliness of a code element) using a machine learning classifier (e.g., Logistic Regression [33]).

In order to empirically assess the actual capabilities of ML techniques for code smell detection, Arcelli Fontana et al. [1] conducted a large-scale study where 32 different ML algorithms were applied to detect four code smell types, i.e., Data Class, Large Class, Feature Envy and Long Method. The authors reported that most of the classifiers exceeded 99% both in terms of accuracy and of F-Measure, with *Random Forest* obtaining the best performance. The authors see in these results an indication that “using machine learning algorithms for code smell detection is an appropriate approach” and that “performances are already so good that we think it does not really matter in practice what machine learning algorithm one chooses for code smell detection” [1].

In our research, we have observed an important limitation of the work by Arcelli Fontana et al. [1] that might affect the generalizability of their findings. Specifically, the high performance reported might be due to the way the dataset was constructed: for each type of code smell analyzed, the dataset contains only instances affected by this type of smell or non-smelly instances with a non-realistic balance of smelly and non-smelly instances [8], [30] and a strongly different distribution of the metrics between the two groups of instances, which is far from reality.

In this paper, we propose a replicated study on the usage of

Bad Code Smells - Machine Learning-based Code Smell Detection

Theoretically speaking, machine learning has the potential to address all the limitations of currently available code smell detection techniques.

- The composition of the training set can bias the performance by up to 90%.
- Machine learning models **do not** perform better than heuristic ones.

Comparing Heuristic and Machine Learning Approaches for Metric-Based Code Smell Detection

Fabiano Piccoli¹, Fabio Palomba², Dario Di Nucci², Andrea De Lucini¹
¹University of Salerno, Italy; ²University of Zurich, Switzerland; ³Vrije Universiteit Brussel, Belgium
{fpiccoli,di_nucci, palomba}@unizh.ch, dario.di_nucci@unizh.ch, andrea@unisa.it

Abstract—Code smells represent poor implementation choices performed by developers when enhancing source code. Their negative impact on source code maintainability and comprehensibility has been widely shown in the past and several techniques to automatically detect them have been devised. Most of these techniques are based on heuristics, namely they compute a set of code metrics and combine them by creating detection rules, while they have a reasonable accuracy, a recent trend is represented by the use of machine learning where code metrics are used as predictors of the smelliness of code artifacts. Despite the recent advances in the field, there is still a noticeable lack of knowledge of whether machine learning can actually be more accurate than traditional heuristic-based approaches. To fill this gap, in this paper, we propose a large-scale study to empirically compare the performance of heuristic-based and machine-learning-based techniques for metric-based code smell detection. We consider five code smell types and compare machine learning models with DEDON, a state-of-the-art heuristic-based approach. Key findings emphasize the need of further research aimed at improving the effectiveness of both machine learning and heuristic approaches for code smell detection while DEDON generally achieves better performance than a machine learning baseline. Its precision is still too low to make it useful in practice.

Index Terms—Code Smells Detection; Heuristics; Machine Learning; Empirical Study

1. INTRODUCTION

Software maintenance and evolution is a complex activity that requires developers to steadily modify source code to adapt it to new requirements or fix defects identified in production [1]. Such an activity is usually performed under strict deadlines and developers are often forced to set aside good programming practices and principles to deliver the most appropriate product on time [2]–[4]. This may lead to *technical debt* [5], namely the introduction of design issues that may negatively affect system maintainability in the future. One of the foremost indications of the presence of technical debt is represented by code smells [6], i.e., suboptimal design solutions that developers apply on a software system. Long methods implementing several functionalities, classes having complex structures, or excessive coupling between classes are just few examples of code smells typically observable in existing software systems [7].

In recent years, code smells has been investigated under different perspectives [8], [9], their introduction [9], [10] and evolution [12]–[16], their impact on reliability [17], [18] and maintainability [7], [19] as well as the way developers perceive them [20]–[22] have been deeply analyzed in literature and have revealed that code smells represent serious threats to source code maintenance and evolution. Most notably, the impact of

code smells on program comprehension has been investigated by Abbas et al. [23] and Yamashita and Bloomen [24]. Both studies have demonstrated that code smells negatively impact program comprehension by reducing the maintainability of the affected classes.

For all these reasons, several techniques to automatically identify code smells in source code have been widely investigated [25], [26]. Most of these techniques rely on heuristics and discriminate code artifacts affected (or not) by a certain type of smell through the application of detection rules that compare the values of relevant metrics extracted from source code against some empirically identified thresholds. As an example, Moha et al. [27] devised DEDON, a method to define code smell detection rules using a Domain-Specific Language. The accuracy of DEDON, as well as of the other heuristic approaches, has been empirically assessed and was found to be fairly high, e.g., the F-Measure of DEDON when detecting *Long* instances is 88.2%. Nevertheless, there are some important limitations that threaten the adoption of these heuristic approaches in practice [25], [28]. First, code smells identified by these techniques are subjectively interpreted by developers, meaning that they output code smell candidates that are not considered as actual problems by developers [29], [30]. Furthermore, the agreement between detectors is very low [31], which means that different detectors are required to detect the smelliness of different code components. At last, the performance of most of the current detectors is strongly influenced by the thresholds needed to identify smelly and non-smelly instances [25].

To overcome these limitations, researchers recently adapted machine learning (ML) to avoid thresholds and decrease the false positive rate [32]: in this schema, a classifier (e.g., Logistic Regression [33]) exploits a set of independent variables (i.e., predictors) to calculate the value of a dependent variable (i.e., the presence of a smell or degree of the smelliness of a code element). Although the use of machine learning looks promising, its actual accuracy for code smell detection is still under debate, as previous work has observed contrasting results [32], [34]. More importantly, it is still unknown whether these techniques actually represent a better solution with respect to traditional heuristic ones. In other words, the problem of assessing the feasibility of machine learning for code smell detection is still open and requires further investigations.

In this paper, we perform a step ahead toward this direction: we propose a large-scale empirical study—that features 125

Bad Code Smells - Machine Learning-based Code Smell Detection

Theoretically speaking, machine learning has the potential to address all the limitations of currently available code smell detection techniques.

Comparing Heuristic and Machine Learning Approaches for Metric-Based Code Smell Detection

Fabiano Fregene¹, Fabio Palomba², Dario Di Nucci², Andrea De Lucini¹
¹University of Salerno, Italy; ²University of Zurich, Switzerland; ³Vrije Universiteit Brussel, Belgium
{fregene@unisa.it, palomba@unisa.it, dario.di.nucci@unizh.ch, andrea@unisa.it}

Abstract—Code smells represent poor implementation choices performed by developers when enhancing source code. Their negative impact on source code maintainability and comprehensibility has been widely shown in the past and several techniques to automatically detect them have been devised. Most of these techniques are based on heuristics, namely they compute a set of code metrics and combine them by creating detection rules, while they have a reasonable accuracy, a recent trend is represented by the use of machine learning where code metrics are used as predictors of the smelliness of code artifacts. Despite the recent advances in the field, there is still a noticeable lack of knowledge of whether machine learning can actually be more accurate than traditional heuristic-based approaches. To fill this gap, in this paper, we propose a large-scale study to empirically compare the performance of heuristic-based and machine-learning-based techniques for metric-based code smell detection. We consider five code smell types and compare machine learning models with DEDON, a state-of-the-art heuristic-based approach. Key findings emphasize the need of further research aimed at improving the effectiveness of both machine learning and heuristic approaches for code smell detection while DEDON generally achieves better performance than a machine learning baseline. Its precision is still too low to make it useful in practice.

Index Terms—Code Smells Detection; Heuristics; Machine Learning; Empirical Study

1. INTRODUCTION

Software maintenance and evolution is a complex activity that requires developers to steadily modify source code to adapt it to new requirements or fix defects identified in production [1]. Such an activity is usually performed under strict deadlines and developers are often forced to set aside good programming practices and principles to deliver the most appropriate product on time [2]–[4]. This may lead to *technical debt* [5], namely the introduction of design issues that may negatively affect system maintainability in the future. One of the foremost indications of the presence of technical debt is represented by code smells [6], i.e., suboptimal design solutions that developers apply on a software system. Long methods implementing several functionalities, classes having complex structures, or excessive coupling between classes are just few examples of code smells typically observable in existing software systems [7].

In recent years, code smells have been investigated under different perspectives [8], [9], their introduction [9], [10] and evolution [12]–[16], their impact on reliability [17], [18] and maintainability [7], [19] as well as the way developers perceive them [20]–[22] have been deeply analyzed in literature and have revealed that code smells represent serious threats to source code maintenance and evolution. Most notably, the impact of

code smells on program comprehension has been investigated by Abbas et al. [23] and Yamashita and Bloomen [24]. Both studies have demonstrated that code smells negatively impact program comprehension by reducing the maintainability of the affected classes.

For all these reasons, several techniques to automatically identify code smells in source code have been widely investigated [25], [26]. Most of these techniques rely on heuristics and discriminate code artifacts affected (or not) by a certain type of smell through the application of detection rules that compare the values of relevant metrics extracted from source code against some empirically identified thresholds. As an example, Moha et al. [27] devised DEDON, a method to define code smell detection rules using a Domain-Specific Language. The accuracy of DEDON, as well as of the other heuristic approaches, has been empirically assessed and was found to be fairly high, e.g., the F-Measure of DEDON when detecting *Long* instances is 88.2%. Nevertheless, there are some important limitations that threaten the adoption of these heuristic approaches in practice [25], [28]. First, code smells identified by these techniques are subjectively interpreted by developers, meaning that they output code smell candidates that are not considered as actual problems by developers [29], [30]. Furthermore, the agreement between detectors is very low [31], which means that different detectors are required to detect the smelliness of different code components. At last, the performance of most of the current detectors is strongly influenced by the thresholds needed to identify smelly and non-smelly instances [25].

To overcome these limitations, researchers recently adapted machine learning (ML) to avoid thresholds and decrease the false positive rate [32]: in this schema, a classifier (e.g., Logistic Regression [33]) exploits a set of independent variables (i.e., predictors) to calculate the value of a dependent variable (i.e., the presence of a smell or degree of the smelliness of a code element). Although the use of machine learning looks promising, its actual accuracy for code smell detection is still under debate, as previous work has observed contrasting results [32], [34]. More importantly, it is still unknown whether these techniques actually represent a better solution with respect to traditional heuristic ones. In other words, the problem of assessing the feasibility of machine learning for code smell detection is still open and requires further investigations.

In this paper, we perform a step ahead toward this direction: we propose a large-scale empirical study—that features 125

- The composition of the training set can bias the performance by up to 90%.
- Machine learning models **do not** perform better than heuristic ones.
- Machine learning models **work similarly** to a random approach.

Bad Code Smells - Machine Learning-based Code Smell Detection

Theoretically speaking, machine learning has the potential to address all the limitations of currently available code smell detection techniques.

Comparing Heuristic and Machine Learning Approaches for Metric-Based Code Smell Detection

Fabiano Piccoli¹, Fabio Palomba², Dario Di Nucci³, Andrea De Lucini⁴
¹University of Salerno, Italy, ²University of Zurich, Switzerland, ³Vrije Universiteit Brussel, Belgium
fpiccoli@unisa.it, palomba@inf.unizh.ch, dario.di.nucci@unibz.it, andrea@unisa.it

Abstract—Code smells represent poor implementation choices performed by developers when enhancing source code. Their negative impact on source code maintainability and comprehensibility has been widely shown in the past and several techniques to automatically detect them have been devised. Most of these techniques are based on heuristics, namely they compute a set of code metrics and combine them by creating detection rules, while they have a reasonable accuracy, a recent trend is represented by the use of machine learning where code metrics are used as predictors of the smelliness of code artifacts. Despite the recent advances in the field, there is still a noticeable lack of knowledge of whether machine learning can actually be more accurate than traditional heuristic-based approaches. To fill this gap, in this paper, we propose a large-scale study to empirically compare the performance of heuristic-based and machine-learning-based techniques for metric-based code smell detection. We consider five code smell types and compare machine learning models with DEDON, a state-of-the-art heuristic-based approach. Our findings emphasize the need of further research aimed at improving the effectiveness of both machine learning and heuristic approaches for code smell detection while DEDON generally achieves better performance than a machine learning baseline. Its precision is still too low to make it useful in practice.

Index Terms—Code Smells Detection; Heuristics; Machine Learning; Empirical Study

1. INTRODUCTION

Software maintenance and evolution is a complex activity that enforces developers to steadily modify source code to adapt it to new requirements or fix defects identified in production [1]. Such an activity is usually performed under strict deadlines and developers are often forced to set aside good programming practices and principles to deliver the most appropriate product on time [2]–[4]. This may lead to *technical debt* [5], namely the introduction of design issues that may negatively affect system maintainability in the future. One of the foremost indications of the presence of technical debt is represented by code smells [6], i.e., sub-optimal design solutions that developers apply on a software system. Long methods implementing several functionalities, classes having complex structures, or excessive coupling between classes are just few examples of code smells typically observable in existing software systems [7].

In recent years, code smells have been investigated under different perspectives [8], [9], their introduction [9], [10] and evolution [12]–[16], their impact on reliability [17], [18] and maintainability [7], [19] as well as the way developers perceive them [20]–[22] have been deeply analyzed in literature and have revealed that code smells represent serious threats to source code maintenance and evolution. Most notably, the impact of

code smells on program comprehension has been investigated by Abbas et al. [23] and Yamashita and Bloomen [24]. Both studies have demonstrated that code smells negatively impact program comprehension by reducing the maintainability of the affected classes.

For all these reasons, several techniques to automatically identify code smells in source code have been widely investigated [25], [26]. Most of these techniques rely on heuristics and discriminate code artifacts affected (or not) by a certain type of smell through the application of detection rules that compare the values of relevant metrics extracted from source code against some empirically identified thresholds. As an example, Moha et al. [27] devised DEDON, a method to define code smell detection rules using a Domain-Specific Language. The accuracy of DEDON, as well as of the other heuristic approaches, has been empirically assessed and was found to be fairly high, e.g., the F-Measure of DEDON when detecting *Long* instances is 88%. Nevertheless, there are some important limitations that threaten the adoption of these heuristic approaches in practice [25], [28]. First, code smells identified by these techniques are subjectively interpreted by developers, meaning that they output code smell candidates that are not considered as actual problems by developers [29], [30]. Furthermore, the agreement between detectors is very low [31], which means that different detectors are required to detect the smelliness of different code components. At last, the performance of most of the current detectors is strongly influenced by the thresholds needed to identify smelly and non-smelly instances [25].

To overcome these limitations, researchers recently adapted machine learning (ML) to avoid thresholds and decrease the false positive rate [32]: in this schema, a classifier (e.g., Logistic Regression [33]) exploits a set of independent variables (i.e., predictors) to calculate the value of a dependent variable (i.e., the presence of a smell or degree of the smelliness of a code element). Although the use of machine learning looks promising, its actual accuracy for code smell detection is still under debate, as previous work has observed contrasting results [32], [34]. More importantly, it is still unknown whether these techniques actually represent a better solution with respect to traditional heuristic ones. In other words, the problem of assessing the feasibility of machine learning for code smell detection is still open and requires further investigations.

In this paper, we perform a step ahead toward this direction: we propose a large-scale empirical study—that features 125

- The composition of the training set can bias the performance by up to 90%.
- Machine learning models **do not** perform better than heuristic ones.
- Machine learning models **work similarly** to a random approach.
- Machine learning models **work worse** than a pessimistic classifier.

Bad Code Smells - Machine Learning-based Code Smell Detection

Theoretically speaking, machine learning has the potential to address all the limitations of currently available code smell detection techniques.

Comparing Heuristic and Machine Learning Approaches for Metric-Based Code Smell Detection

Fabiano Focardi¹, Fabio Palomba², Dario Di Nucci², Andrea De Lucini¹
¹University of Salerno, Italy; ²University of Zurich, Switzerland; ³Vrije Universiteit Brussel, Belgium
{focardi,di_nucci, palomba}@unizh.ch, dario.di_nucci@unz.ch, andrea@unisa.it

Abstract—Code smells represent poor implementation choices performed by developers when enhancing source code. Their negative impact on source code maintainability and comprehensibility has been widely shown in the past and several techniques to automatically detect them have been devised. Most of these techniques are based on heuristics, namely they compute a set of code metrics and combine them by creating detection rules, while they have a reasonable accuracy, a recent trend is represented by the use of machine learning where code metrics are used as predictors of the smelliness of code artifacts. Despite the recent advances in the field, there is still a noticeable lack of knowledge of whether machine learning can actually be more accurate than traditional heuristic-based approaches. To fill this gap, in this paper, we propose a large-scale study to empirically compare the performance of heuristic-based and machine-learning-based techniques for metric-based code smell detection. We consider five code smell types and compare machine learning models with DEDON, a state-of-the-art heuristic-based approach. Our findings emphasize the need of further research aimed at improving the effectiveness of both machine learning and heuristic approaches for code smell detection while DEDON generally achieves better performance than a machine learning baseline. Its precision is still too low to make it useful in practice.

Index Terms—Code Smells Detection; Heuristics; Machine Learning; Empirical Study

1. INTRODUCTION

Software maintenance and evolution is a complex activity that enforces developers to steadily modify source code to adapt it to new requirements or fix defects identified in production [1]. Such an activity is usually performed under strict deadlines and developers are often forced to get aside good programming practices and principles to deliver the most appropriate product on time [2]–[4]. This may lead to *technical debt* [5], namely the introduction of design issues that may negatively affect system maintainability in the future. One of the foremost indications of the presence of technical debt is represented by code smells [6], i.e., suboptimal design solutions that developers apply on a software system. Long methods implementing several functionalities, classes having complex structures, or excessive coupling between classes are just few examples of code smells typically observable in existing software systems [7].

In recent years, code smells has been investigated under different perspectives [8], [9], their introduction [9], [10] and evolution [12]–[16], their impact on reliability [17], [18] and maintainability [7], [19] as well as the way developers perceive them [20]–[22] have been deeply analyzed in literature and have revealed that code smells represent serious threats to source code maintenance and evolution. Most notably, the impact of

code smells on program comprehension has been investigated by Abbas et al. [23] and Yamashita and Bloomen [24]. Both studies have demonstrated that code smells negatively impact program comprehension by reducing the maintainability of the affected classes.

For all these reasons, several techniques to automatically identify code smells in source code have been widely investigated [25], [26]. Most of these techniques rely on heuristics and discriminate code artifacts affected (or not) by a certain type of smell through the application of detection rules that compare the values of relevant metrics extracted from source code against some empirically identified thresholds. As an example, Moha et al. [27] devised DEDON, a method to define code smell detection rules using a Domain-Specific Language. The accuracy of DEDON, as well as of the other heuristic approaches, has been empirically assessed and was found to be fairly high, e.g., the F-Measure of DEDON when detecting *Long* instances is 88.2%. Nevertheless, there are some important limitations that threaten the adoption of these heuristic approaches in practice [25], [28]. First, code smells identified by these techniques are subjectively interpreted by developers, meaning that they output code smell candidates that are not considered as actual problems by developers [29], [30]. Furthermore, the agreement between detectors is very low [31], which means that different detectors are required to detect the smelliness of different code components. At last, the performance of most of the current detectors is strongly influenced by the thresholds needed to identify smelly and non-smelly instances [25].

To overcome these limitations, researchers recently adapted machine learning (ML) to avoid thresholds and decrease the false positive rate [32]: in this schema, a classifier (e.g., Logistic Regression [33]) exploits a set of independent variables (i.e., predictors) to calculate the value of a dependent variable (i.e., the presence of a smell or degree of the smelliness of a code element). Although the use of machine learning looks promising, its actual accuracy for code smell detection is still under debate, as previous work has observed contrasting results [32], [34]. More importantly, it is still unknown whether these techniques actually represent a better solution with respect to traditional heuristic ones. In other words, the problem of assessing the feasibility of machine learning for code smell detection is still open and requires further investigations.

In this paper, we perform a step ahead toward this direction: we propose a large-scale empirical study—that features 125

- The composition of the training set can bias the performance by up to 90%.
- Machine learning models **do not** perform better than heuristic ones.
- Machine learning models **work similarly** to a random approach.
- Machine learning models **work worse** than a pessimistic classifier.

Why so bad?

Bad Code Smells - Machine Learning-based Code Smell Detection

Theoretically speaking, machine learning has the potential to address all the limitations of currently available code smell detection techniques.

Comparing Heuristic and Machine Learning Approaches for Metric-Based Code Smell Detection

Fabiano Focardi¹, Fabio Palomba², Dario Di Nucci², Andrea De Lucini¹
¹University of Salerno, Italy; ²University of Zurich, Switzerland; ³Vrije Universiteit Brussel, Belgium
{focardi,di_nucci, palomba}@unisa.it, dario.di_nucci@unizh.ch, andrea@unisa.it

Abstract—Code smells represent poor implementation choices performed by developers when enhancing source code. Their negative impact on source code maintainability and comprehension ability has been widely shown in the past and several techniques to automatically detect them have been devised. Most of these techniques are based on heuristics, namely they compute a set of code metrics and combine them by creating detection rules, while they have a reasonable accuracy, a recent trend is represented by the use of machine learning where code metrics are used as predictors of the smelliness of code artifacts. Despite the recent advances in the field, there is still a noticeable lack of knowledge of whether machine learning can actually be more accurate than traditional heuristic-based approaches. To fill this gap, in this paper, we propose a large-scale study to empirically compare the performance of heuristic-based and machine-learning-based techniques for metric-based code smell detection. We consider five code smell types and compare machine learning models with DEDON, a state-of-the-art heuristic-based approach. Our findings emphasize the need of further research aimed at improving the effectiveness of both machine learning and heuristic approaches for code smell detection while DEDON generally achieves better performance than a machine learning baseline. Its precision is still too low to make it useful in practice.

Index Terms—Code Smells Detection; Heuristics; Machine Learning; Empirical Study

1. INTRODUCTION

Software maintenance and evolution is a complex activity that requires developers to steadily modify source code to adapt it to new requirements or fix defects identified in production [1]. Such an activity is usually performed under strict deadlines and developers are often forced to set aside good programming practices and principles to deliver the most appropriate product on time [2]–[4]. This may lead to *technical debt* [5], namely the introduction of design issues that may negatively affect system maintainability in the future. One of the foremost indications of the presence of technical debt is represented by code smells [6], i.e., suboptimal design solutions that developers apply on a software system. Long methods implementing several functionalities, classes having complex structures, or excessive coupling between classes are just few examples of code smells typically observable in existing software systems [7].

In recent years, code smells have been investigated under different perspectives [8, 9], their introduction [9, 10] and evolution [12]–[16], their impact on reliability [17], [18] and maintainability [7], [19] as well as the way developers perceive them [20]–[22] have been deeply analyzed in literature and have revealed that code smells represent serious threats to source code maintenance and evolution. Most notably, the impact of

code smells on program comprehension has been investigated by Abbas et al. [23] and Yamashita and Bloomen [24]. Both studies have demonstrated that code smells negatively impact program comprehension by reducing the maintainability of the affected classes.

For all these reasons, several techniques to automatically identify code smells in source code have been widely investigated [25], [26]. Most of these techniques rely on heuristics and discriminate code artifacts affected (or not) by a certain type of smell through the application of detection rules that compare the values of relevant metrics extracted from source code against some empirically identified thresholds. As an example, Moha et al. [27] devised DEDON, a method to define code smell detection rules using a Domain-Specific Language. The accuracy of DEDON, as well as of the other heuristic approaches, has been empirically assessed and was found to be fairly high, e.g., the F-Measure of DEDON when detecting *Long* instances is 88.2%. Nevertheless, there are some important limitations that threaten the adoption of these heuristic approaches in practice [25], [28]. First, code smells identified by these techniques are subjectively interpreted by developers, meaning that they output code smell candidates that are not considered as actual problems by developers [29], [30]. Furthermore, the agreement between detectors is very low [31], which means that different detectors are required to detect the smelliness of different code components. At last, the performance of most of the current detectors is strongly influenced by the thresholds needed to identify smelly and non-smelly instances [25].

To overcome these limitations, researchers recently adapted machine learning (ML) to avoid thresholds and decrease the false positive rate [32]: in this schema, a classifier (e.g., Logistic Regression [33]) exploits a set of independent variables (i.e., predictors) to calculate the value of a dependent variable (i.e., the presence of a smell or degree of the smelliness of a code element). Although the use of machine learning looks promising, its actual accuracy for code smell detection is still under debate, as previous work has observed contrasting results [32], [34]. More importantly, it is still unknown whether these techniques actually represent a better solution with respect to traditional heuristic ones. In other words, the problem of assessing the feasibility of machine learning for code smell detection is still open and requires further investigations.

In this paper, we perform a step ahead toward this direction: we propose a large-scale empirical study—that features 125

- The composition of the training set can bias the performance by up to 90%.
- Machine learning models **do not** perform better than heuristic ones.
- Machine learning models **work similarly** to a random approach.
- Machine learning models **work worse** than a pessimistic classifier.

Why so bad?

- Smelly elements are **by far** the minority in a software project.

Bad Code Smells - Machine Learning-based Code Smell Detection

Theoretically speaking, machine learning has the potential to address all the limitations of currently available code smell detection techniques.

Comparing Heuristic and Machine Learning Approaches for Metric-Based Code Smell Detection

Fabiano Focardi¹, Fabio Palomba², Dario Di Nucci³, Andrea De Lucini⁴
¹University of Salerno, Italy, ²University of Zurich, Switzerland, ³Vrije Universiteit Brussel, Belgium
ffocardi@unisa.it, palomba@iit.unizh.ch, dario.di.nucci@unibz.it, andrea@unisa.it

Abstract—Code smells represent poor implementation choices performed by developers when enhancing source code. Their negative impact on source code maintainability and comprehensibility has been widely shown in the past and several techniques to automatically detect them have been devised. Most of these techniques are based on heuristics, namely they compute a set of code metrics and combine them by creating detection rules, while they have a reasonable accuracy, a recent trend is represented by the use of machine learning where code metrics are used as predictors of the smelliness of code artifacts. Despite the recent advances in the field, there is still a noticeable lack of knowledge of whether machine learning can actually be more accurate than traditional heuristic-based approaches. To fill this gap, in this paper, we propose a large-scale study to empirically compare the performance of heuristic-based and machine-learning-based techniques for metric-based code smell detection. We consider five code smell types and compare machine learning models with DEDON, a state-of-the-art heuristic-based approach. Our findings emphasize the need of further research aimed at improving the effectiveness of both machine learning and heuristic approaches for code smell detection while DEDON generally achieves better performance than a machine learning baseline. Its precision is still too low to make it useful in practice.

Index Terms—Code Smells Detection; Heuristics; Machine Learning; Empirical Study

1. INTRODUCTION

Software maintenance and evolution is a complex activity that requires developers to steadily modify source code to adapt it to new requirements or fix defects identified in production [1]. Such an activity is usually performed under strict deadlines and developers are often forced to get aside good programming practices and principles to deliver the most appropriate product on time [2]–[4]. This may lead to *technical debt* [5], namely the introduction of design issues that may negatively affect system maintainability in the future. One of the foremost indications of the presence of technical debt is represented by code smells [6], i.e., suboptimal design solutions that developers apply on a software system. Long methods implementing several functionalities, classes having complex structures, or excessive coupling between classes are just few examples of code smells typically observable in existing software systems [7].

In recent years, code smells have been investigated under different perspectives [8], [9], their introduction [10], [11] and evolution [12]–[16], their impact on reliability [17], [18] and maintainability [7], [19] as well as the way developers perceive them [20]–[22] have been deeply analyzed in literature and have revealed that code smells represent serious threats to source code maintenance and evolution. Most notably, the impact of

code smells on program comprehension has been investigated by Abbas et al. [23] and Yamashita and Bloomen [24]. Both studies have demonstrated that code smells negatively impact program comprehension by reducing the maintainability of the affected classes.

For all these reasons, several techniques to automatically identify code smells in source code have been widely investigated [25], [26]. Most of these techniques rely on heuristics and discriminate code artifacts affected (or not) by a certain type of smell through the application of detection rules that compare the values of relevant metrics extracted from source code against some empirically identified thresholds. As an example, Moha et al. [27] devised DEDON, a method to define code smell detection rules using a Domain-Specific Language. The accuracy of DEDON, as well as of the other heuristic approaches, has been empirically assessed and was found to be fairly high, e.g., the F-Measure of DEDON when detecting *Long* instances is 88.2%. Nevertheless, there are some important limitations that threaten the adoption of these heuristic approaches in practice [25], [28]. First, code smells identified by these techniques are subjectively interpreted by developers, meaning that they output code smell candidates that are not considered as actual problems by developers [29], [30]. Furthermore, the agreement between detectors is very low [31], which means that different detectors are required to detect the smelliness of different code components. At last, the performance of most of the current detectors is strongly influenced by the thresholds needed to identify smelly and non-smelly instances [25].

To overcome these limitations, researchers recently adapted machine learning (ML) to avoid thresholds and decrease the false positive rate [32]: in this schema, a classifier (e.g., Logistic Regression [33]) exploits a set of independent variables (i.e., predictors) to calculate the value of a dependent variable (i.e., the presence of a smell or degree of the smelliness of a code element). Although the use of machine learning looks promising, its actual accuracy for code smell detection is still under debate, as previous work has observed contrasting results [32], [34]. More importantly, it is still unknown whether these techniques actually represent a better solution with respect to traditional heuristic ones. In other words, the problem of assessing the feasibility of machine learning for code smell detection is still open and requires further investigations.

In this paper, we perform a step ahead toward this direction: we propose a large-scale empirical study—then featuring 125

- The composition of the training set can bias the performance by up to 90%.
- Machine learning models **do not** perform better than heuristic ones.
- Machine learning models **work similarly** to a random approach.
- Machine learning models **work worse** than a pessimistic classifier.

Why so bad?

- Smelly elements are **by far** the minority in a software project.
- Machine learning algorithms **often fail** to learn the features characterizing smelly elements.

Bad Code Smells - Machine Learning-based Code Smell Detection

Theoretically speaking, machine learning has the potential to address all the limitations of currently available code smell detection techniques.

Comparing Heuristic and Machine Learning Approaches for Metric-Based Code Smell Detection

Fabiano Fregene¹, Fabio Palomba², Dario Di Nucci², Andrea De Lucchi¹
¹University of Salerno, Italy; ²University of Zurich, Switzerland; ³Vrije Universiteit Brussel, Belgium
{fregene@unisa.it, palomba@iit.unizh.ch, dario.di.nucci@unizh.ch, andrea@unisa.it}

Abstract—Code smells represent poor implementation choices performed by developers when enhancing source code. Their negative impact on source code maintainability and comprehensibility has been widely shown in the past and several techniques to automatically detect them have been devised. Most of these techniques are based on heuristics, namely they compute a set of code metrics and combine them by creating detection rules, while they have a reasonable accuracy, a recent trend is represented by the use of machine learning where code metrics are used as predictors of the smelliness of code artifacts. Despite the recent advances in the field, there is still a noticeable lack of knowledge of whether machine learning can actually be more accurate than traditional heuristic-based approaches. To fill this gap, in this paper, we propose a large-scale study to empirically compare the performance of heuristic-based and machine-learning-based techniques for metric-based code smell detection. We consider five code smell types and compare machine learning models with DEDON, a state-of-the-art heuristic-based approach. Our findings emphasize the need of further research aimed at improving the effectiveness of both machine learning and heuristic approaches for code smell detection while DEDON generally achieves better performance than a machine learning baseline, its precision is still too low to make it useful in practice.

Index Terms—Code Smells Detection; Heuristics; Machine Learning; Empirical Study

1. INTRODUCTION

Software maintenance and evolution is a complex activity that requires developers to steadily modify source code to adapt it to new requirements or fix defects identified in production [1]. Such an activity is usually performed under strict deadlines and developers are often forced to get aside good programming practices and principles to deliver the most appropriate product on time [2]–[4]. This may lead to *technical debt* [5], namely the introduction of design issues that may negatively affect system maintainability in the future. One of the foremost indications of the presence of technical debt is represented by code smells [6], i.e., suboptimal design solutions that developers apply on a software system. Long methods implementing several functionalities, classes having complex structures, or excessive coupling between classes are just few examples of code smells typically observable in existing software systems [7].

In recent years, code smells have been investigated under different perspectives [8], [9], their introduction [10], [11] and evolution [12]–[16], their impact on reliability [17], [18] and maintainability [7], [19], as well as the way developers perceive them [20]–[22] have been deeply analyzed in literature and have revealed that code smells represent serious threats to source code maintenance and evolution. Most notably, the impact of

code smells on program comprehension has been investigated by Abbas et al. [23] and Yarnishin and Bloomen [24]. Both studies have demonstrated that code smells negatively impact program comprehension by reducing the maintainability of the affected classes.

For all these reasons, several techniques to automatically identify code smells in source code have been widely investigated [25], [26]. Most of these techniques rely on heuristics and discriminate code artifacts affected (or not) by a certain type of smell through the application of detection rules that compare the values of relevant metrics extracted from source code against some empirically identified thresholds. As an example, Moha et al. [27] devised DEDON, a method to define code smell detection rules using a Domain-Specific Language. The accuracy of DEDON, as well as of the other heuristic approaches, has been empirically assessed and was found to be fairly high, e.g., the F-Measure of DEDON when detecting *Long* instances is 98.2%. Nevertheless, there are some important limitations that threaten the adoption of these heuristic approaches in practice [25], [28]. First, code smells identified by these techniques are subjectively interpreted by developers, meaning that they output code smell candidates that are not considered as actual problems by developers [29], [30]. Furthermore, the agreement between detectors is very low [31], which means that different detectors are required to detect the smelliness of different code components. At last, the performance of most of the current detectors is strongly influenced by the thresholds needed to identify smelly and non-smelly instances [25].

To overcome these limitations, researchers recently adapted machine learning (ML) to avoid thresholds and decrease the false positive rate [32]: in this schema, a classifier (e.g., Logistic Regression [33]) exploits a set of independent variables (i.e., predictors) to calculate the value of a dependent variable (i.e., the presence of a smell or degree of the smelliness of a code element). Although the use of machine learning looks promising, its actual accuracy for code smell detection is still under debate, as previous work has observed contrasting results [32], [34]. More importantly, it is still unknown whether these techniques actually represent a better solution with respect to traditional heuristic ones. In other words, the problem of assessing the feasibility of machine learning for code smell detection is still open and requires further investigations.

In this paper, we perform a step ahead toward this direction: we propose a large-scale empirical study—then featuring 125

- The composition of the training set can bias the performance by up to 90%.
- Machine learning models **do not** perform better than heuristic ones.
- Machine learning models **work similarly** to a random approach.
- Machine learning models **work worse** than a pessimistic classifier.

Why so bad?

- Smelly elements are **by far** the minority in a software project.
- Machine learning algorithms **often fail** to learn the features characterizing smelly elements.
- Data balancing techniques **often fail** when it comes to code smell prediction.

Bad Code Smells - Summing Up

Bad code smells (a.k.a., code smells or simply smells) represent **symptoms of the presence of poor design and/or implementation choices** that may lead to additional unforeseen software development costs.

- They can be detected in multiple ways and using different metrics.

Bad Code Smells - Summing Up

Bad code smells (a.k.a., code smells or simply smells) represent **symptoms of the presence of poor design and/or implementation choices** that may lead to additional unforeseen software development costs.

- They can be detected in multiple ways and using different metrics.
- Most of the techniques defined so far are based on structural analysis of source code, yet alternative sources of information seem to be even more efficient and promising - especially because of their orthogonality.

Bad Code Smells - Summing Up

Bad code smells (a.k.a., code smells or simply smells) represent **symptoms of the presence of poor design and/or implementation choices** that may lead to additional unforeseen software development costs.

- They can be detected in multiple ways and using different metrics.
- Most of the techniques defined so far are based on structural analysis of source code, yet alternative sources of information seem to be even more efficient and promising - especially because of their orthogonality.
- There are, still, a number of limitations related to subjectiveness, agreement, and thresholds to use when detecting code smells.

Bad Code Smells - Summing Up

Bad code smells (a.k.a., code smells or simply smells) represent **symptoms of the presence of poor design and/or implementation choices** that may lead to additional unforeseen software development costs.

- They can be detected in multiple ways and using different metrics.
- Most of the techniques defined so far are based on structural analysis of source code, yet alternative sources of information seem to be even more efficient and promising - especially because of their orthogonality.
- There are, still, a number of limitations related to subjectiveness, agreement, and thresholds to use when detecting code smells.
- While machine learning seems to be an ideal solution to address these limitations, it is still immature to handle code smell detection.

Bad Code Smells - Summing Up

Bad code smells (a.k.a., code smells or simply smells) represent **symptoms of the presence of poor design and/or implementation choices** that may lead to additional unforeseen software development costs.

- They can be detected in multiple ways and using different metrics.
- Most of the techniques defined so far are based on structural analysis of source code, yet alternative sources of information seem to be even more efficient and promising - especially because of their orthogonality.
- There are, still, a number of limitations related to subjectiveness, agreement, and thresholds to use when detecting code smells.
- While machine learning seems to be an ideal solution to address these limitations, it is still immature to handle code smell detection.

So what?

Bad Code Smells - Summing Up

Bad code smells (a.k.a., code smells or simply smells) represent **symptoms of the presence of poor design and/or implementation choices** that may lead to additional unforeseen software development costs.

- They can be detected in multiple ways and using different metrics.
- Most of the techniques defined so far are based on structural analysis of source code, yet alternative sources of information seem to be even more efficient and promising - especially because of their orthogonality.
- There are, still, a number of limitations related to subjectiveness, agreement, and thresholds to use when detecting code smells.
- While machine learning seems to be an ideal solution to address these limitations, it is still immature to handle code smell detection.

So what?

- The mere application of machine learning models to different contexts **does not work** that much; More effort on **adapting** and **specializing** machine learning.

Bad Code Smells - Summing Up

Bad code smells (a.k.a., code smells or simply smells) represent **symptoms of the presence of poor design and/or implementation choices** that may lead to additional unforeseen software development costs.

- They can be detected in multiple ways and using different metrics.
- Most of the techniques defined so far are based on structural analysis of source code, yet alternative sources of information seem to be even more efficient and promising - especially because of their orthogonality.
- There are, still, a number of limitations related to subjectiveness, agreement, and thresholds to use when detecting code smells.
- While machine learning seems to be an ideal solution to address these limitations, it is still immature to handle code smell detection.

So what?

- The mere application of machine learning models to different contexts **does not work** that much; More effort on **adapting** and **specializing** machine learning.
- The problems of heuristic-based approaches **are still there**, even though practitioners may be more confident with respect to their output.

Bad Code Smells - Summing Up

Bad code smells (a.k.a., code smells or simply smells) represent **symptoms of the presence of poor design and/or implementation choices** that may lead to additional unforeseen software development costs.

- They can be detected in multiple ways and using different metrics.
- Most of the techniques defined so far are based on structural analysis of source code, yet alternative sources of information seem to be even more efficient and promising - especially because of their orthogonality.
- There are, still, a number of limitations related to subjectiveness, agreement, and thresholds to use when detecting code smells.
- While machine learning seems to be an ideal solution to address these limitations, it is still immature to handle code smell detection.

So what?

- The mere application of machine learning models to different contexts **does not work** that much; More effort on **adapting** and **specializing** machine learning.
- The problems of heuristic-based approaches **are still there**, even though practitioners may be more confident with respect to their output.
- Mixing together different sources of information seems to provide **more meaningful recommendations** without necessarily decrease the detection performance.

Bad Code Smells - cASpER (Automated code Smell dEtection and Refactoring)

cASpER is an IntelliJ plug-in for the automatic detection and refactoring of Code Smells.

<https://www.youtube.com/watch?v=HBWF8fFJM8s#t=1m10s>

<https://mdestefano.github.io/files/C1.pdf>

cASpER: A Plug-in for Automated Code Smell Detection and Refactoring

Manuel De Stefano
m.destefano@studenti.unisa.it
University of Salerno, Italy

Michele Simone Gambardella
m.gambardella@studenti.unisa.it
University of Salerno, Italy

Fabiano Pecorelli
fpecorelli@unisa.it
University of Salerno, Italy

Fabio Palomba
fpalomba@unisa.it
University of Salerno, Italy

Andrea De Lucia
a.de Lucia@unisa.it
University of Salerno, Italy

ABSTRACT

During software evolution, code is inevitably subject to continuous changes that are often performed by developers with inexact and strict deadlines. As a consequence, good design practices are often sacrificed, possibly leading to the introduction of suboptimal design or implementation solutions, the so-called code smells. Several studies have shown that the presence of code smells makes the source code more change- and fault-prone, reduces productivity, and causes greater network and more significant design efforts for developers. Refactoring is the practice that developers may use to remove code smells without changing the external behavior of the source code. However, it requires much time and effort, and is poorly accurate, often leading developers to prior keeping low-quality code instead of spending time in designing and performing refactoring operations. To mitigate this problem and support developers throughout the process of code smell identification and refactoring, in this paper we present cASpER, a IntelliJ IDEA plug-in that provides visual and semi-automatic support for detection and refactoring four different types of code smells.

Full Text: <https://github.com/mdestefano/casper/blob/master/README.md>

KEYWORDS

Code smells, Refactoring, Automated Software Engineering.

ACM Reference Format

Manuel De Stefano, Michele Simone Gambardella, Fabiano Pecorelli, Fabio Palomba, and Andrea De Lucia. 2020. cASpER: A Plug-in for Automated Code Smell Detection and Refactoring. In *ASE'20: ACM International Conference on Automated Visual Inspection*, September 20–October 02, 2020, Island of Ischia, Italy. ACM, New York, NY, USA, 3 pages. <https://doi.org/10.1145/3399785.3399951>

1 INTRODUCTION

Software life cycle inevitably demands continuous changes and enhancements [7], which too often require to be completed under strict deadlines. This too often leads developers to set aside

all good design principles in force of quick solutions, allowing the introduction of the so-called code smells [7], i.e., sub-optimal design/implementation, which seriously impact on program comprehension, maintainability, as well as developer's productivity [7]. Refactoring represents the activity to remove code smells without altering the external behavior of the source code [7]. Unfortunately, this activity is conducted either with a great manual effort, or with a limited automatic help, as few code smell detection and refactoring research proposals [1, 4] have become usable tools.

In this paper, we propose cASpER (Automated code Smell Detection and Refactoring), a novel IntelliJ IDEA plugin that: (i) integrates two state-of-the-art code smell detection approaches such as Dorian [3] and Tecto [8] to support the identification of four types of code smells (i.e., Feature Envy, Mismatched Class, Blob and Promiscuous Package); (ii) proposes refactoring recommendations implementing approaches previously proposed in literature [1, 3]; (iii) automatically modifies the source code according to the desired refactoring operations; (iv) and visualizes, in a single view, source code metrics, comments and attributes. In the following sections, we briefly describe the features of the tool and a use case scenario.

2 CASPERS FEATURES

In this section, we provide a brief description of cASpER features, focusing on the detection and the refactoring strategies adopted for each supported code smell (i.e., Feature Envy, Mismatched Class, Blob and Promiscuous Package). The tool offers two kind of well-known and validated detection strategies: structural one, relying on metrics computation (which are pointed out in the online appendix [10]), that is Dorian [5], and a text-based one, Tecto [8], which focuses on the textual content of the component under analysis.

Feature Envy and Mismatched Class represent a problem of a misplaced component, respectively at class or misplaced method level and at package level (a misplaced class) [7]. Tecto [5] detects them computing their conceptual similarity (lexical cosine similarity) with external components and compares it with their container component. If the similarity with the most similar external container (derived container) is higher than the actual container, and the difference is higher than a given threshold, then the component is marked as smelly and a move method/class refactoring is suggested from the current container to the derived one. Tecto [5], on the other hand, compares the component external references (method calls and dependencies, respectively), and if they are more than the component internal one (class or methods of the same class or

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

ASTOS, September 20–October 02, 2020, Island of Ischia, Italy
© 2020. Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-7312-0/20/09...\$15.00
<https://doi.org/10.1145/3399785.3399951>

Code smells: detection & refactoring

Slides by Prof. Fabio Palomba

