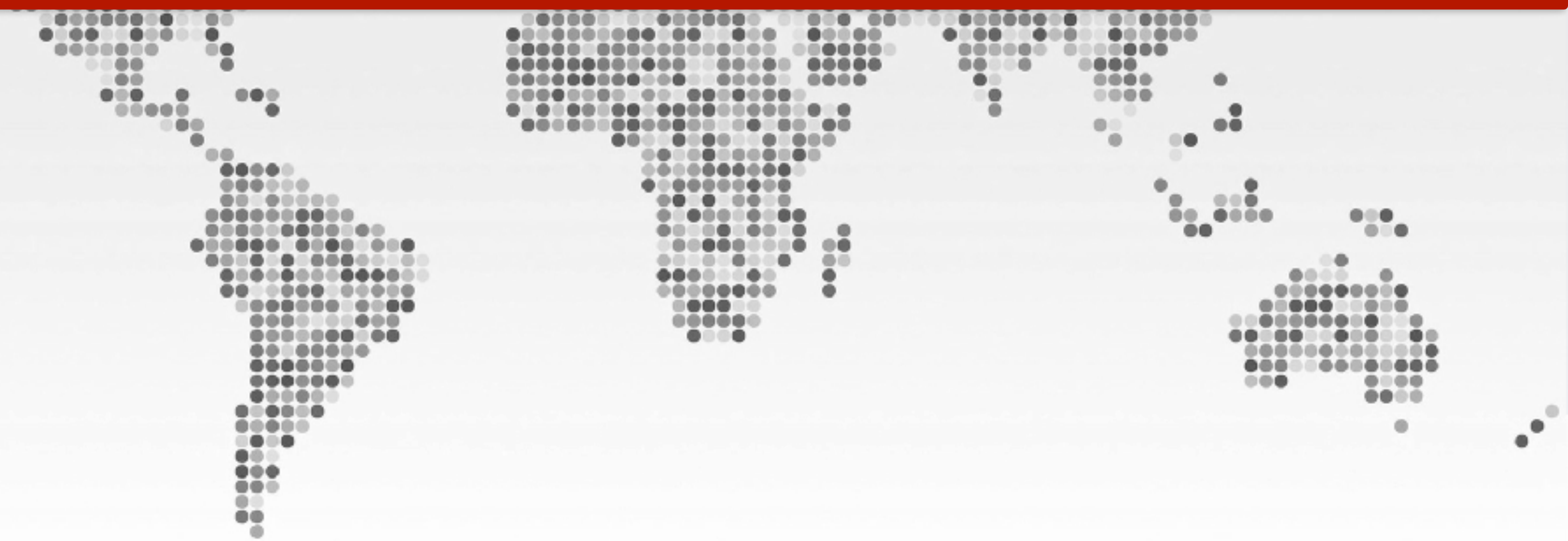
Code smells: detection & refactoring

Slides by Prof. Fabio Palomba



Who I am

Fabiano Pecorelli
Associate Professor
Pegaso Digital University

fabiano.pecorelli@unipegaso.it
https://fabiano-pecorelli.github.io

Who I am

UNIVERSITÀ DEGLI STUDI
DI SALERNO

Who I am

UCL

UNIVERSITÀ DEGLI STUDI
DI SALERNO

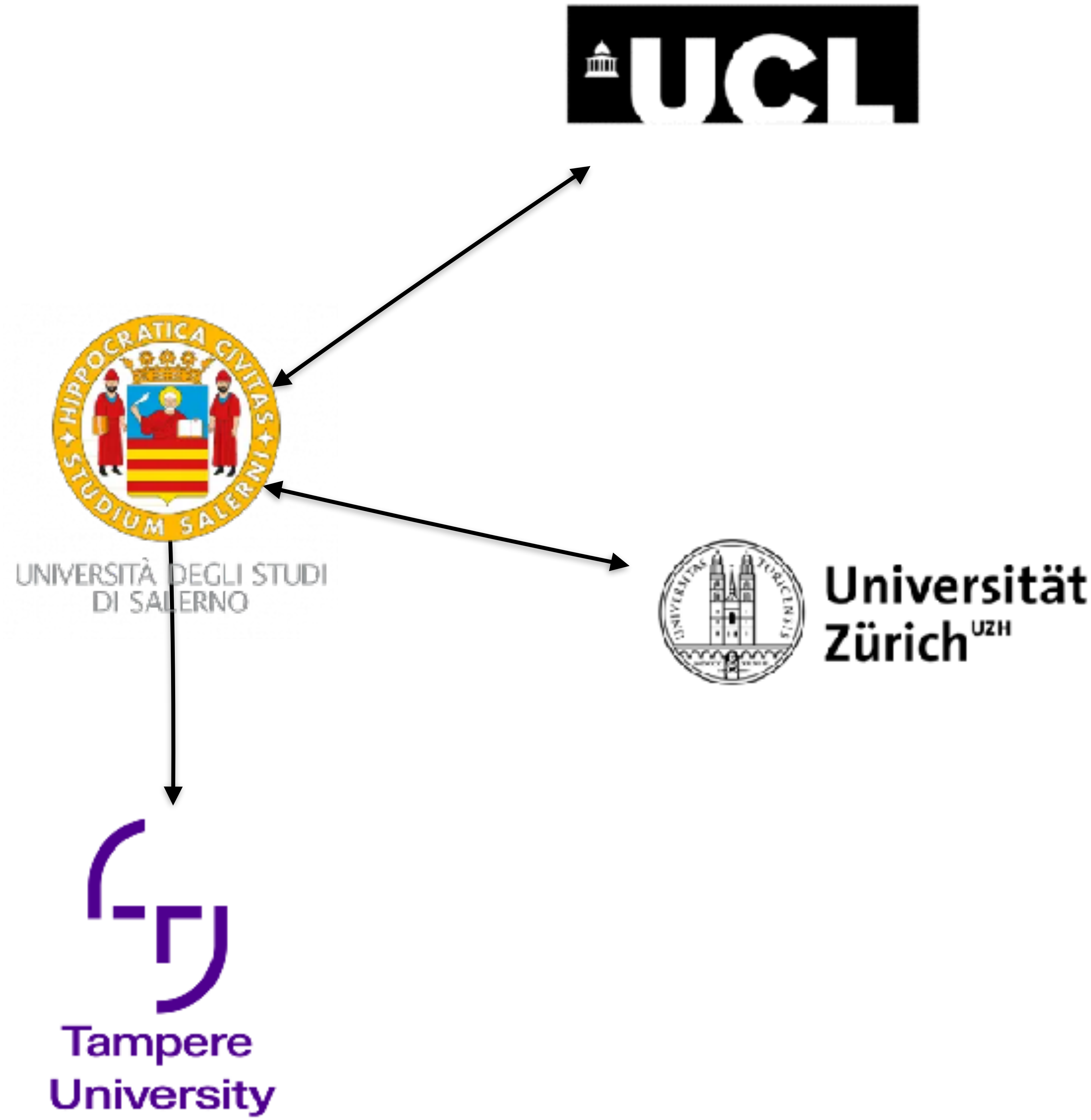
Who I am

UCL

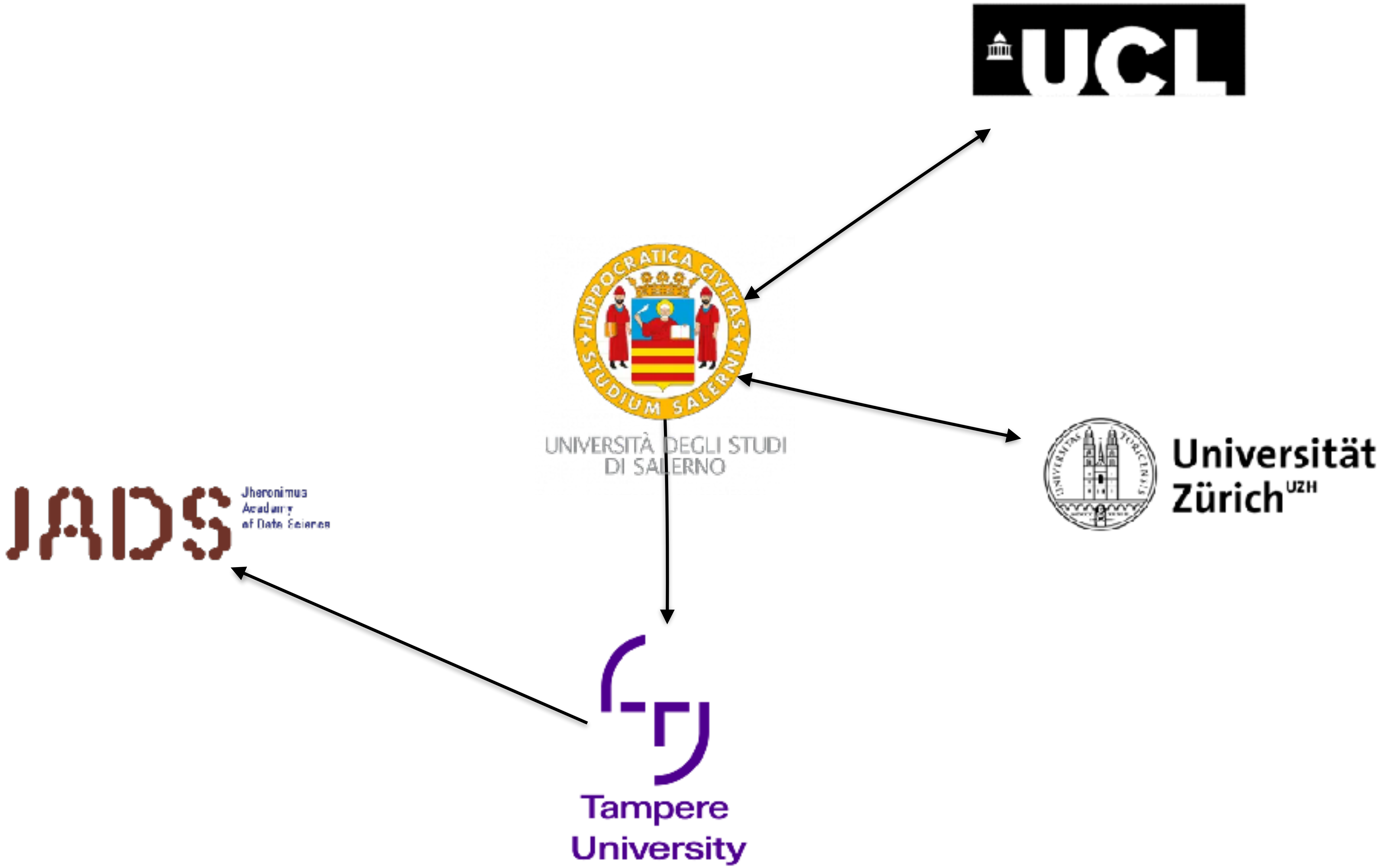
UNIVERSITÀ DEGLI STUDI
DI SALERNO

Universität
Zürich UZH

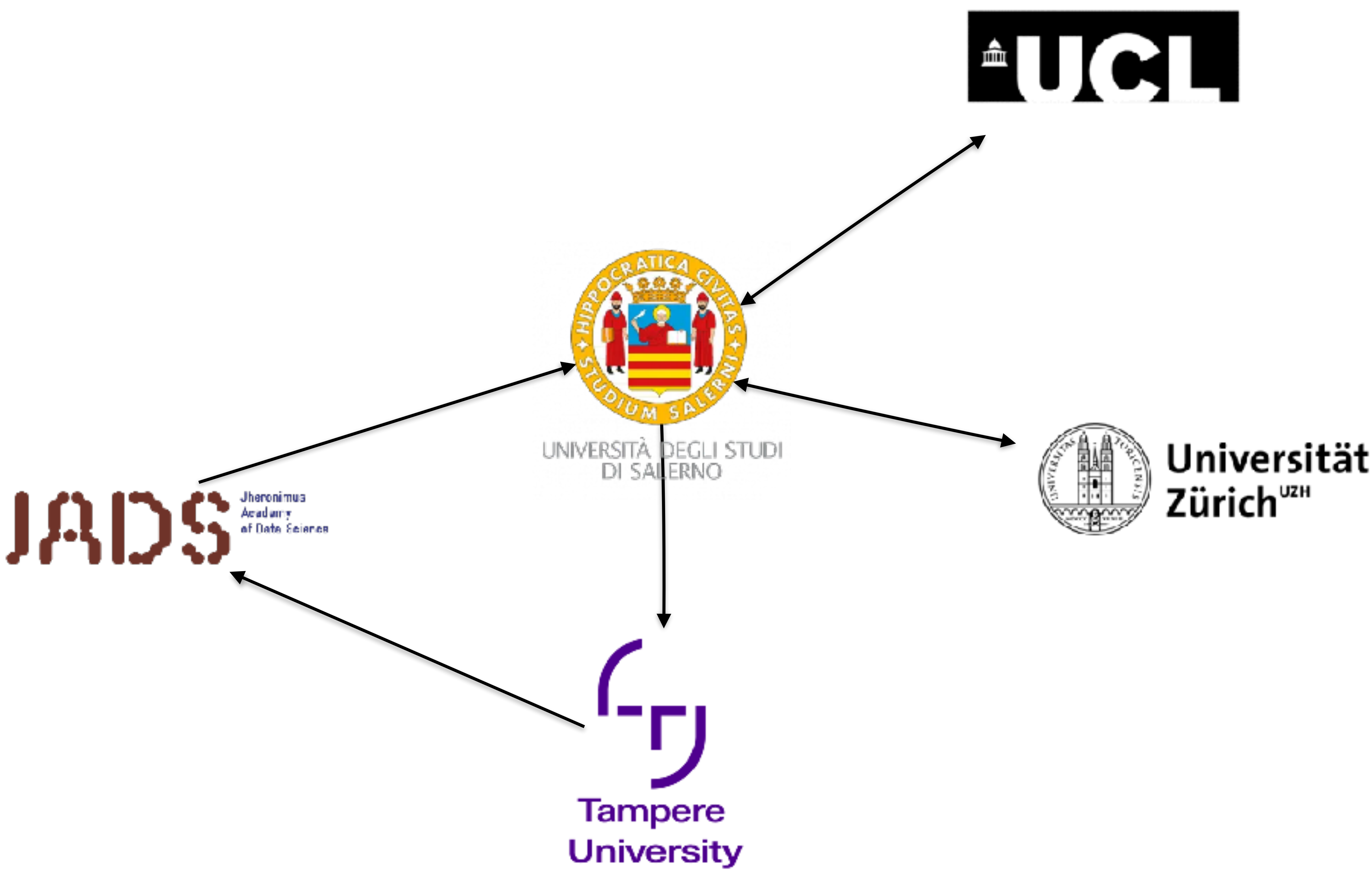
Who I am


UCL
UNIVERSITÀ DEGLI STUDI DI SALERNO
HIPPOCRATICA CIVITAS
STUDIUM SALERNI
Universität Zürich UZH
Tampere University

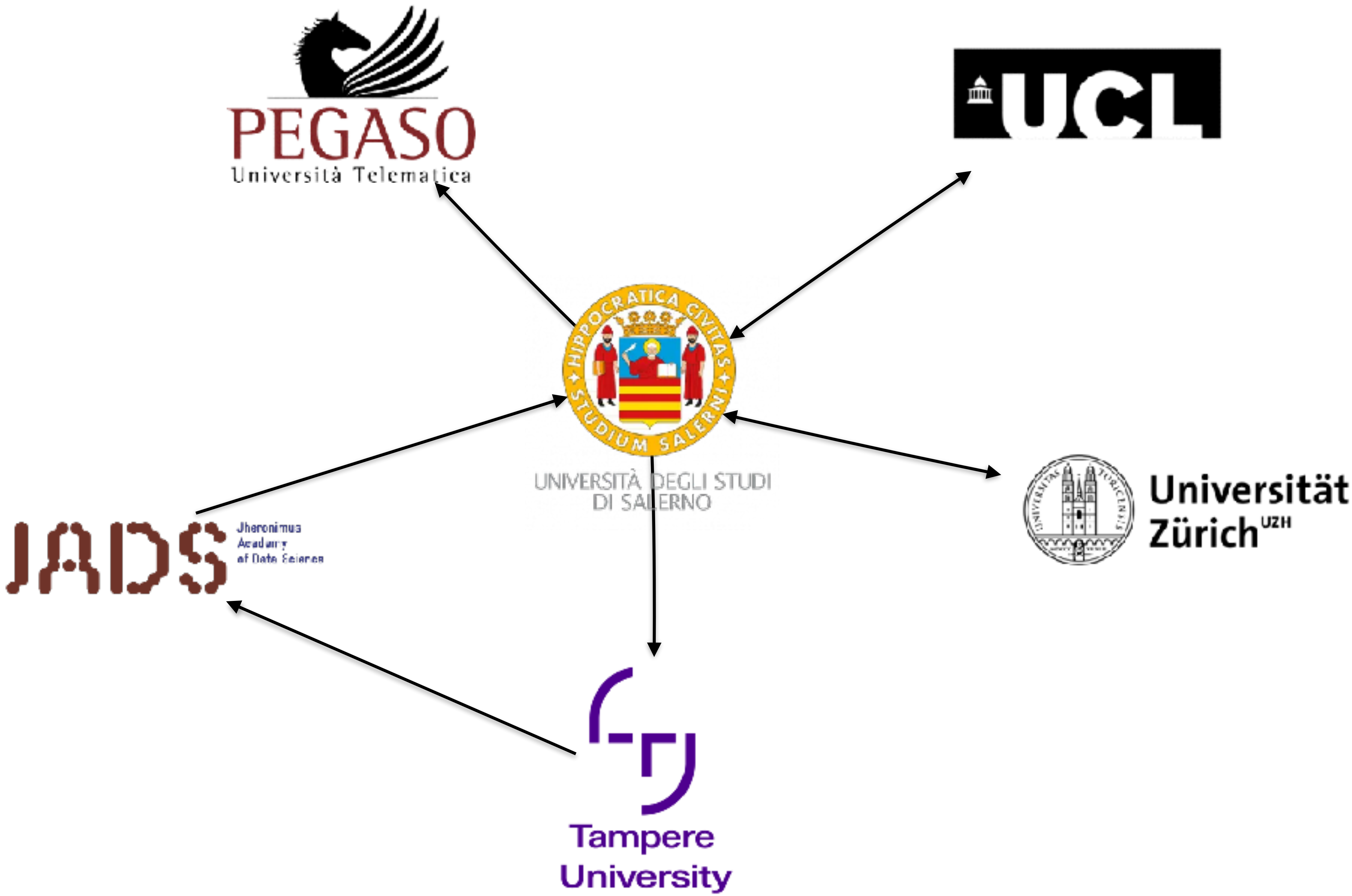
Who I am


UCL
HIPPOCRATICA CIVITAS
STUDIUM SALERNI
UNIVERSITÀ DEGLI STUDI
DI SALERNO
Universität
Zürich UZH
JADS
Jheronimus
Academy
of Data Science
Tampere
University

Who I am


UCL
HIPPOCRATICA CIVITAS
STUDIUM SALERNI
UNIVERSITÀ DEGLI STUDI
DI SALERNO
Universität
Zürich UZH
JADS
Jheronimus
Academy
of Data Science
Tampere
University

Who I am


PEGASO
Università Telematica
UCL
HIPPOCRATICA CIVITAS
STUDIUM SALERNI
UNIVERSITÀ DEGLI STUDI
DI SALERNO
Universität
Zürich UZH
JADS
Jheronimus
Academy
of Data Science
Tampere
University

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**.

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

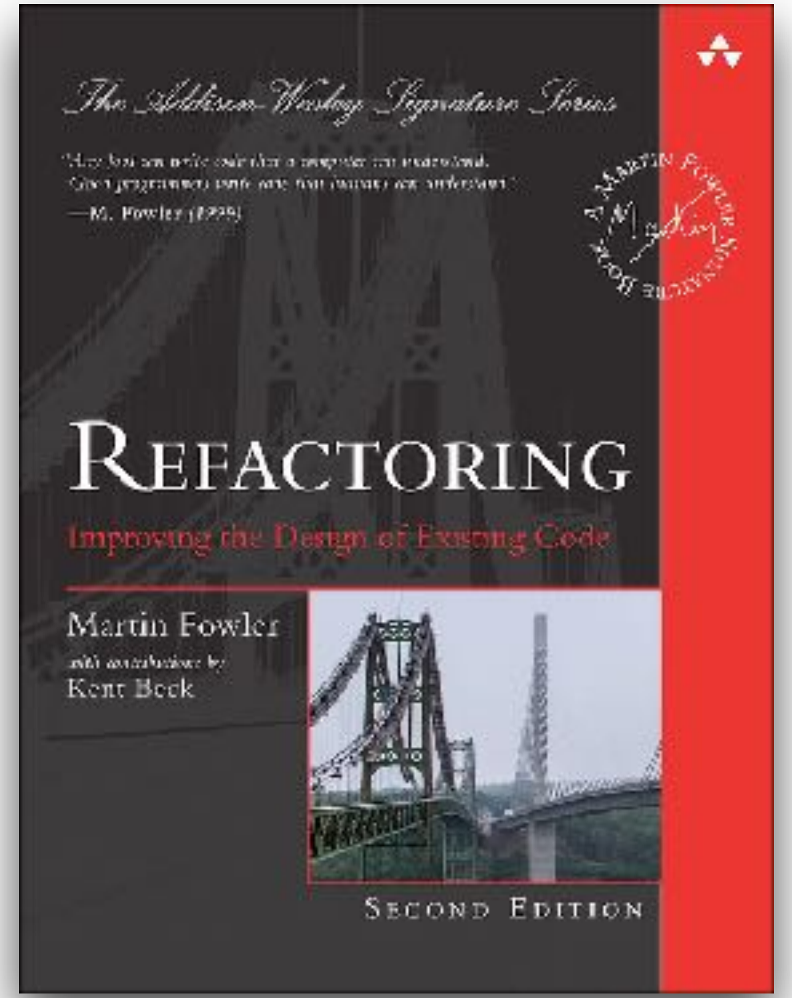
Bad Code Smells - Definition

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 - 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).

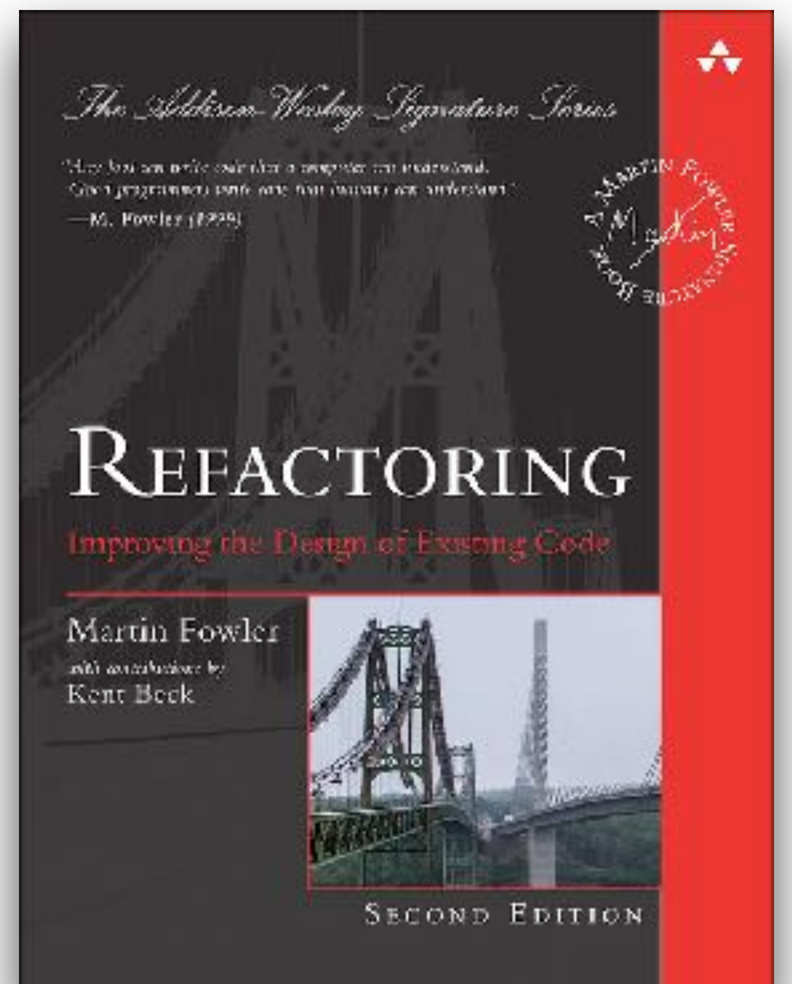


Bad Code Smells

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).



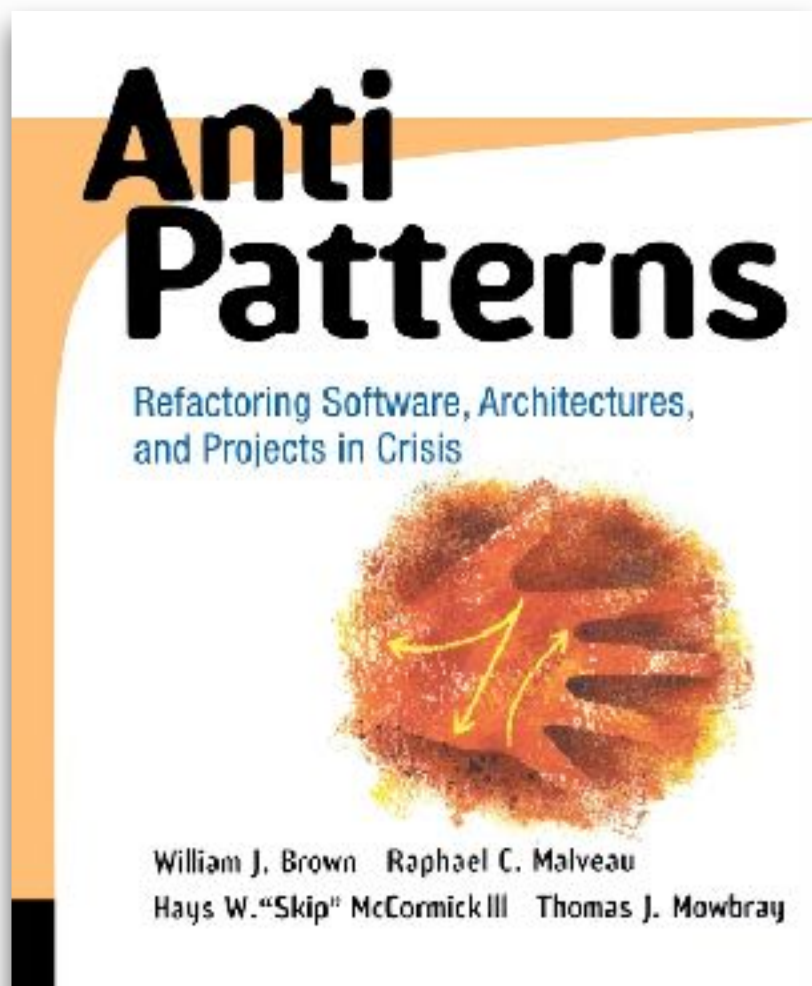
The book describes **more than 20 different types of code smells**, which have to do with violations to all Object-Oriented principles, i.e., cohesion, coupling, complexity, and so on.

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.

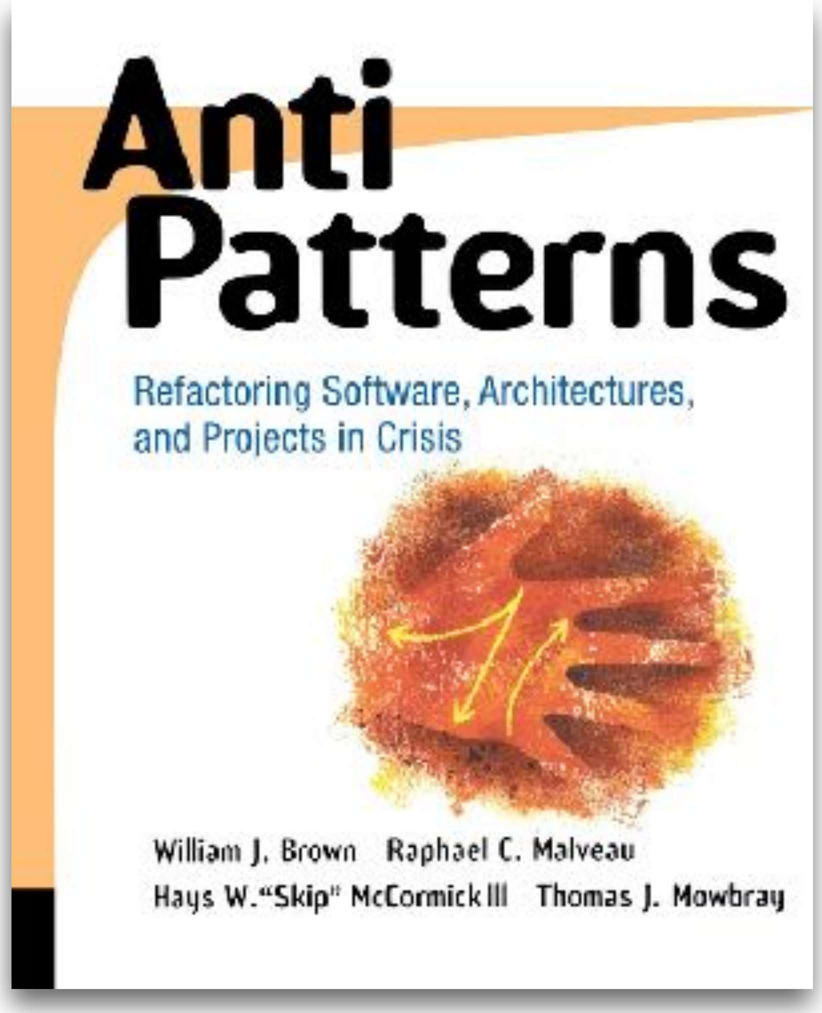
Brown et al., in the same years, defined the concept of **anti-patterns**, i.e., bad programming practices applied in the source code.

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

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 - 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 - 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 - 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 - 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 - 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 - 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

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 - 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 - 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 - 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 - 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

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

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

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.

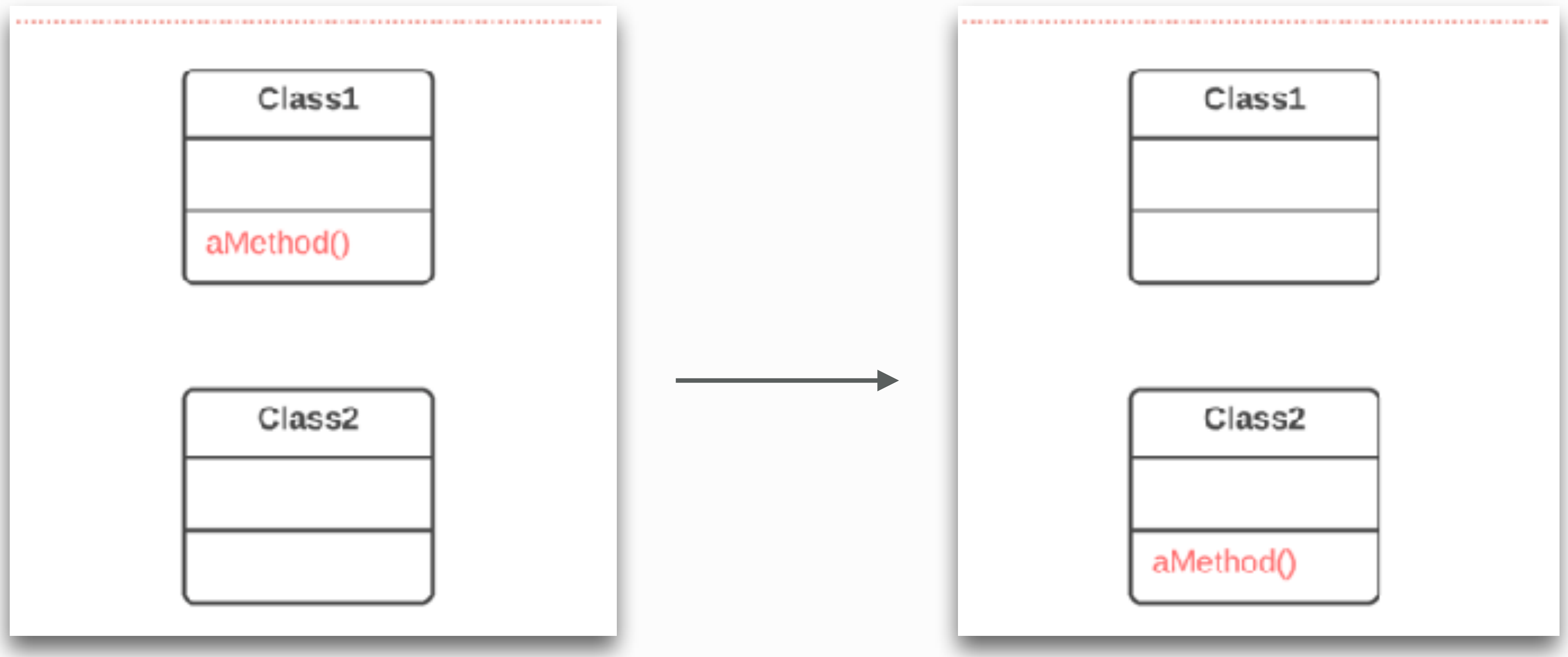
<= In the example, the details are printed in another method.

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.

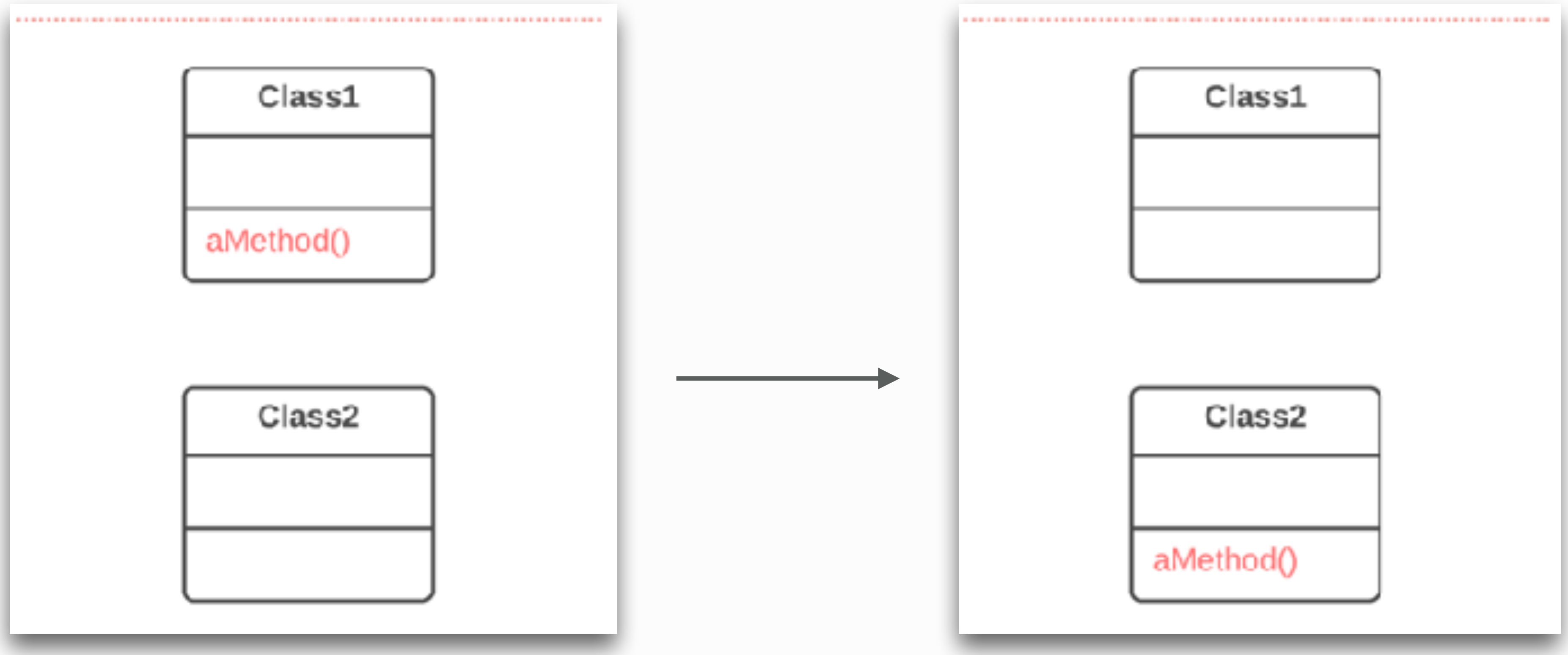
Refactoring - Moving Features Between Objects

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

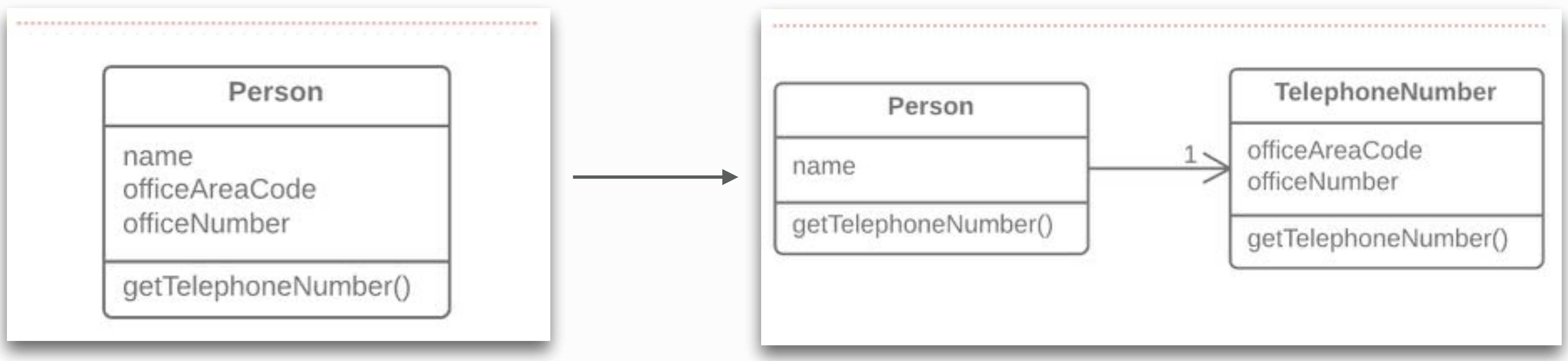
- Make sure that the method **is not** declared in superclasses and subclasses. If this is the case, you will either have to refrain from moving or else implement a kind of polymorphism in the recipient class in order to ensure varying functionality of a method split up among donor classes.

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.

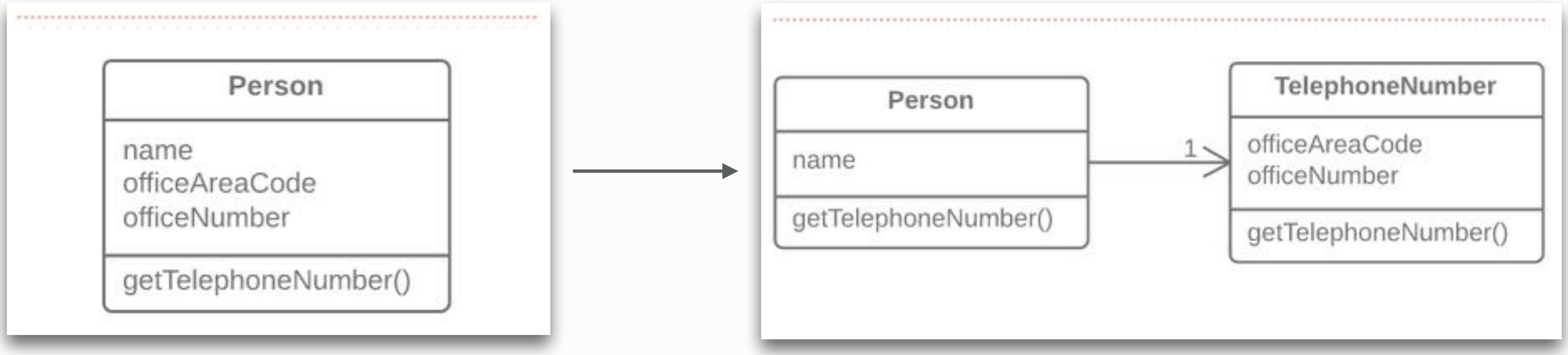
Refactoring - Moving Features Between Objects

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.

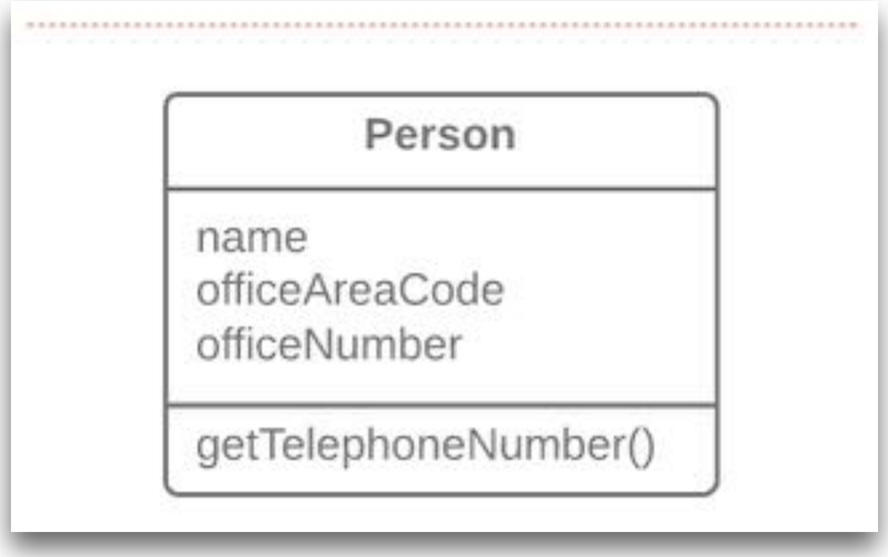


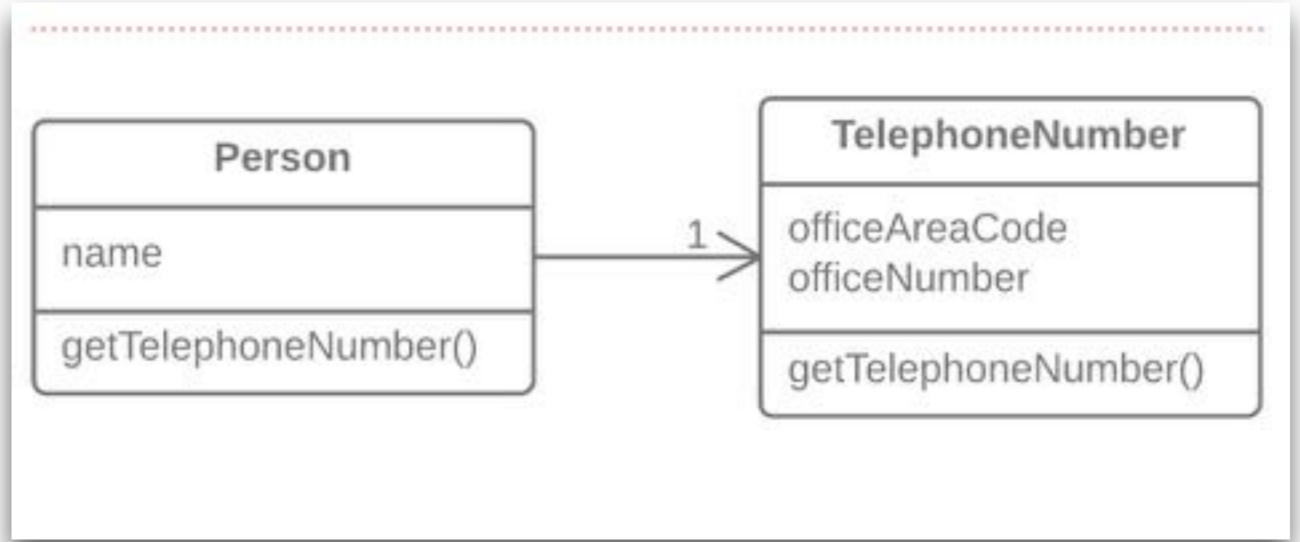
Some precautions

- 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.



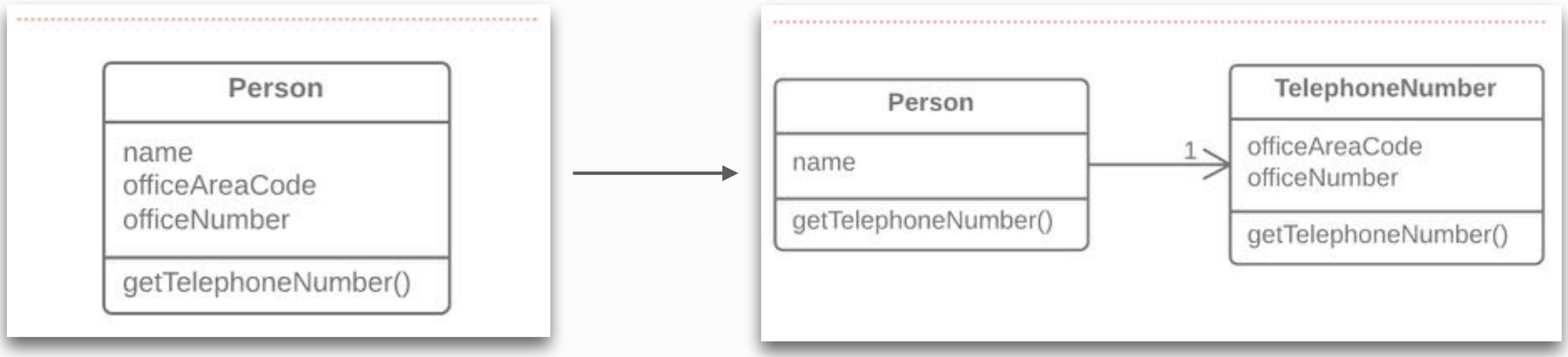


Some precautions

- 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 - 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 - 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 - 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 - 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**.

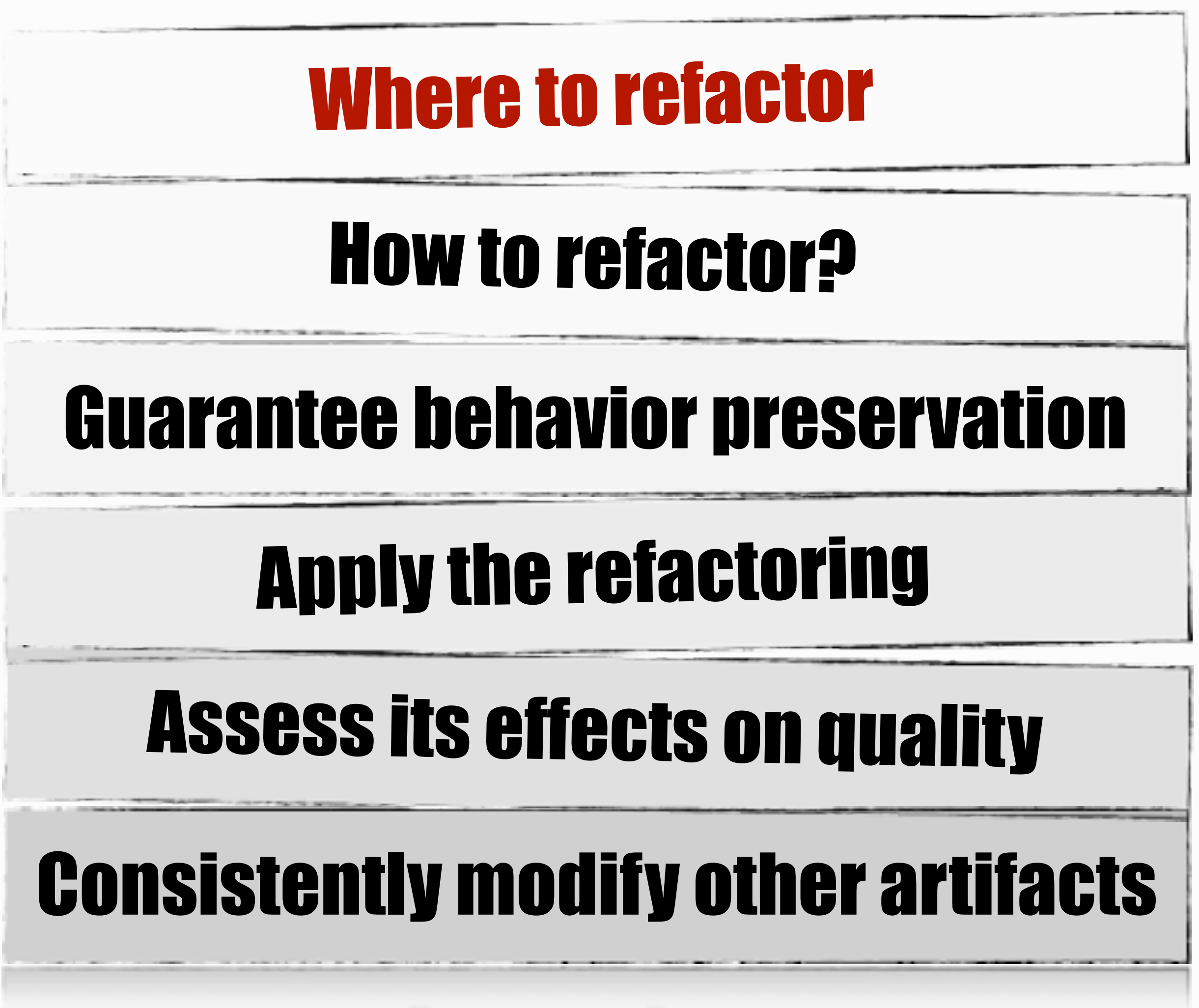
- Where to refactor
- How to refactor?
- Guarantee behavior preservation
- Apply the refactoring
- Assess its effects on quality
- Consistently modify other artifacts

Mens and Tourwé
A Survey of Software Refactoring
TSE 2004

- Identification of portions of code in need of refactoring.

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**.



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.

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.

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

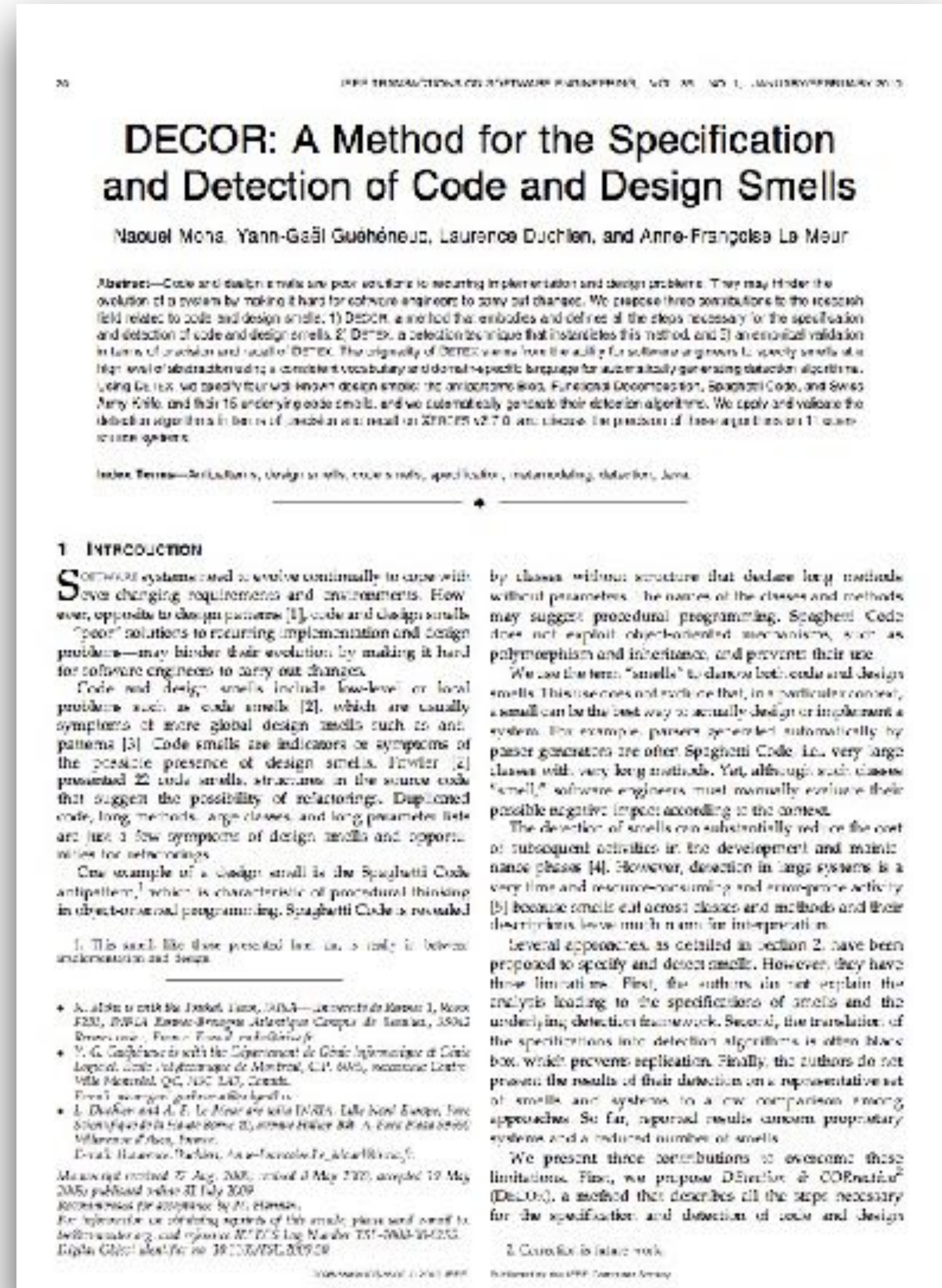
Structural, historical, conceptual information have been assessed.

Method

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

Bad Code Smells

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

1 Introduction

Text-based descriptions of smells

↓

Domain Analysis

↓

Vocabulary, Taxonomy

↓

Specification

↓

Rule Cards

↓

Algorithm Generation

↓

Detection Algorithm

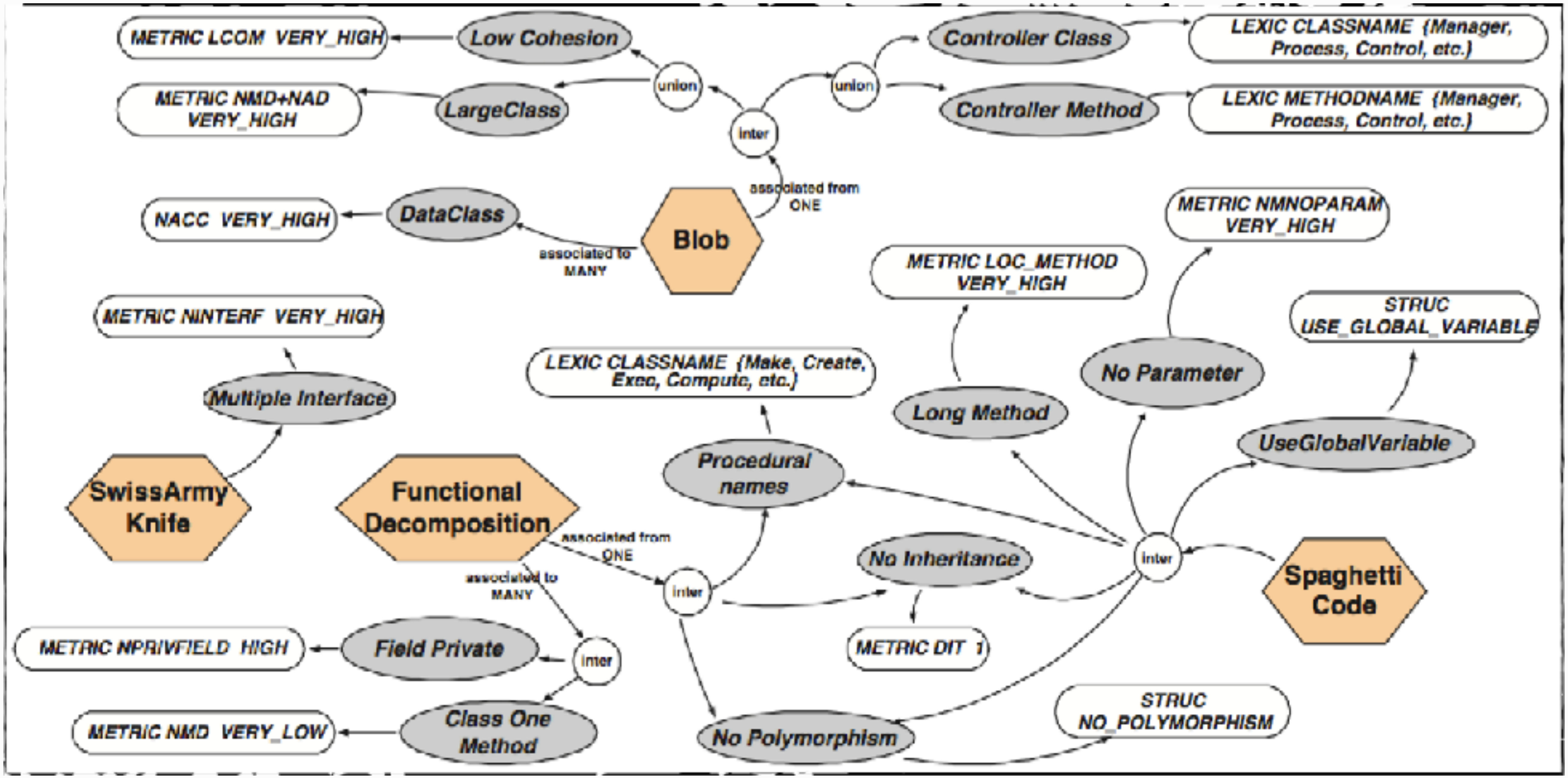
Bad Code Smells - Where to Refactor

- 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 - Where to Refactor

- Step II: Taxonomy creation and specification.

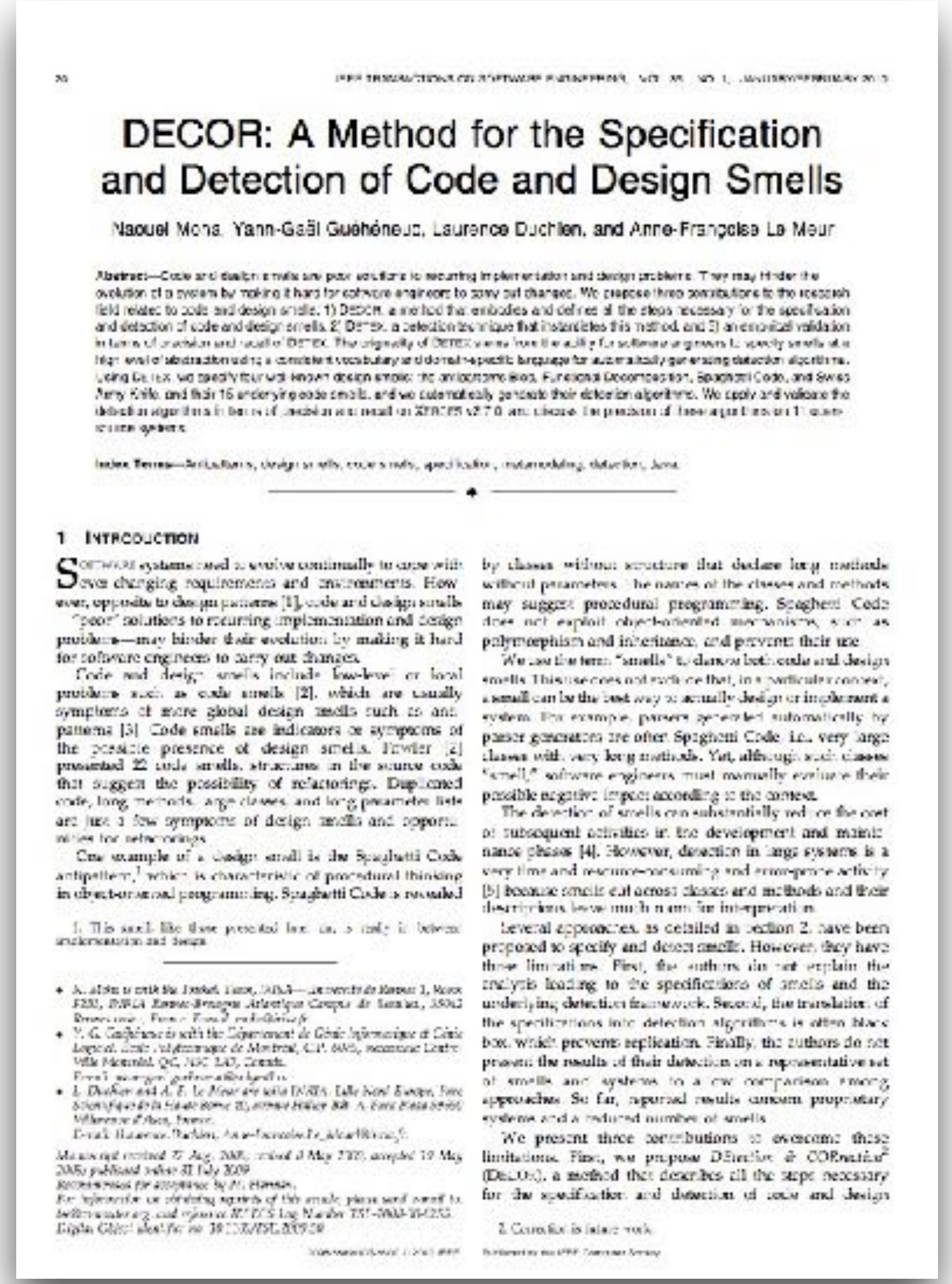


Bad Code Smells - Where to Refactor

- 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



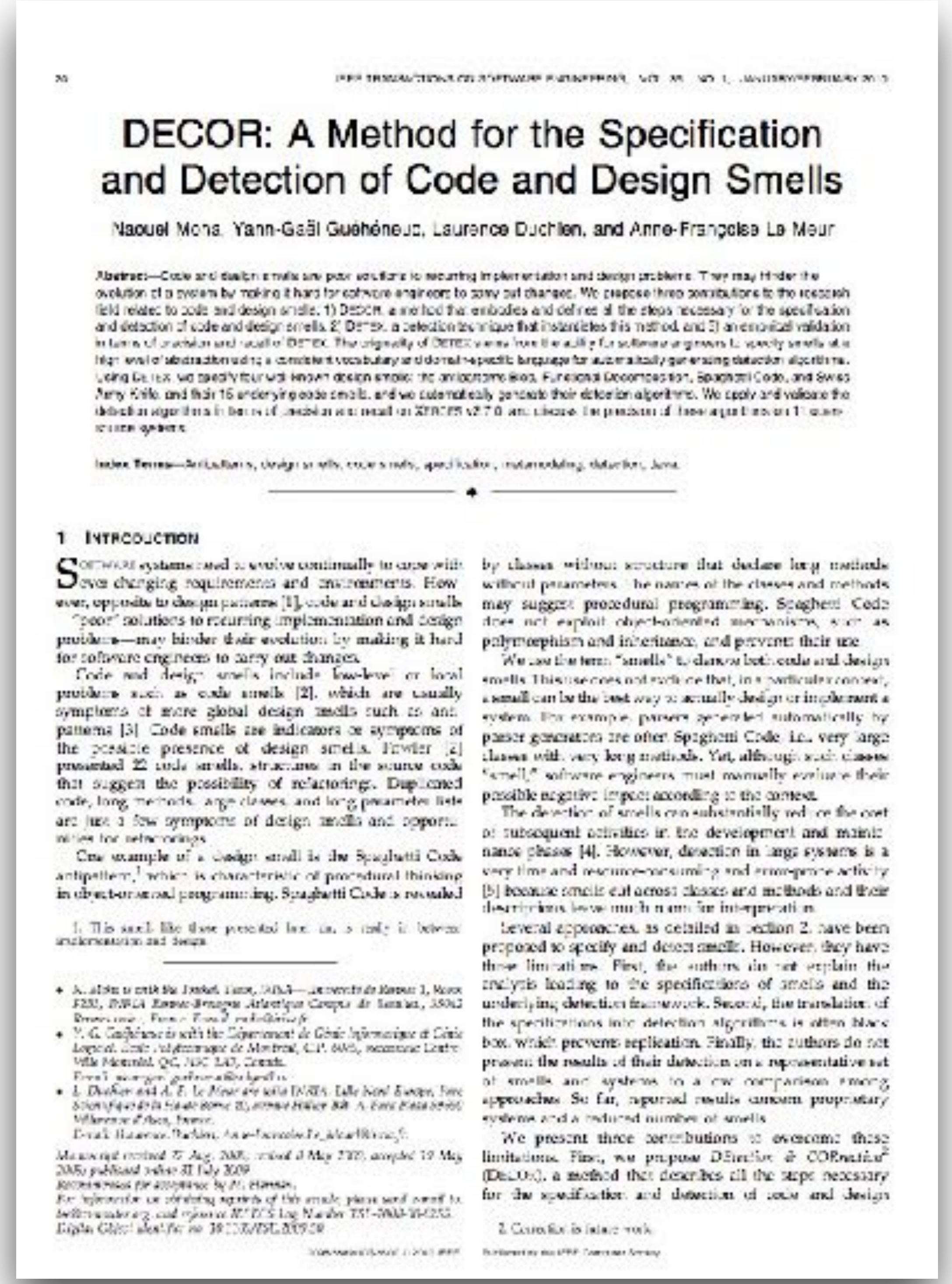
DECOR: A Method for the Specification and Detection of Code and Design Smells

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

1 Introduction

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



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

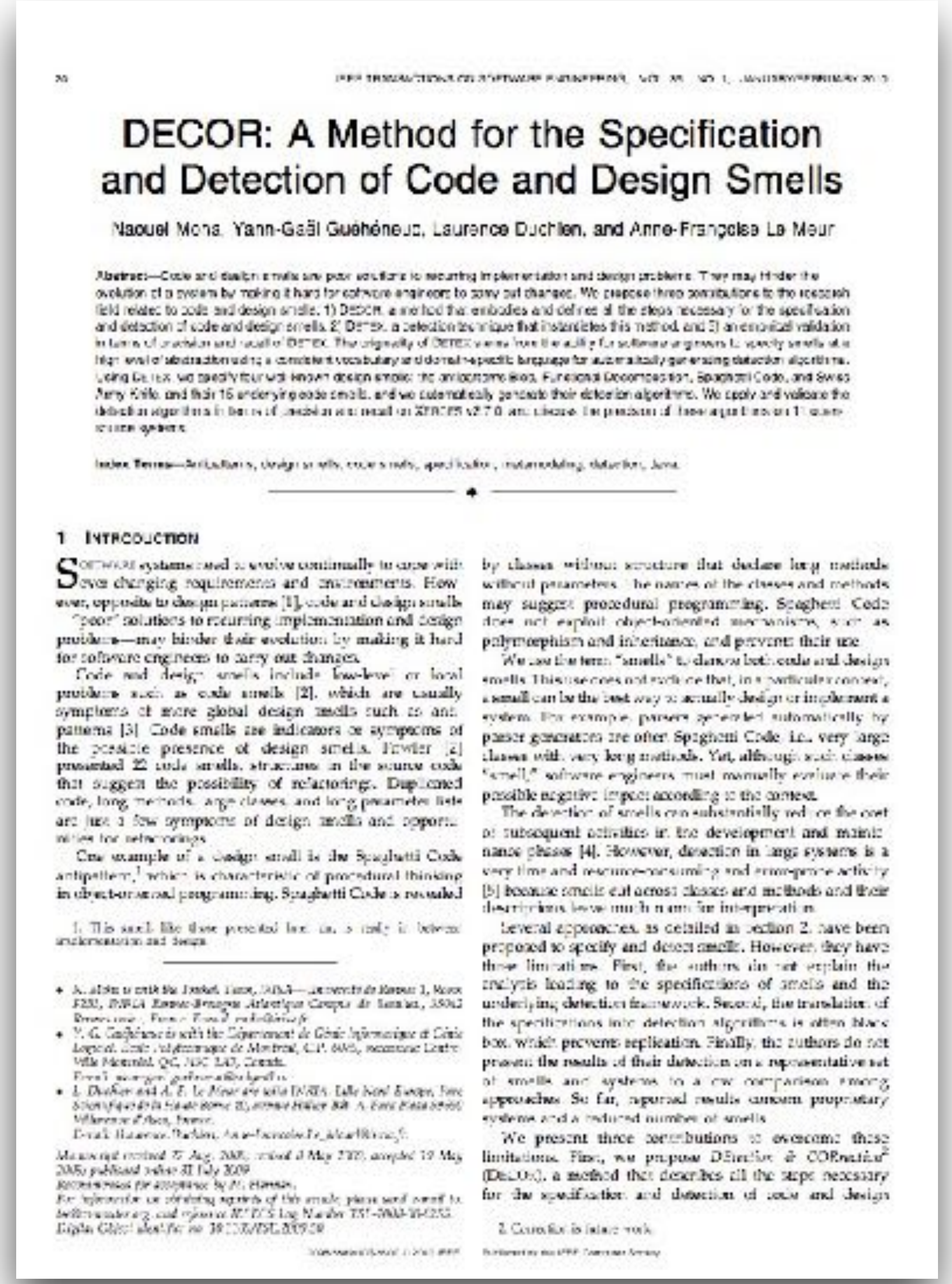
1 INTRODUCTION

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

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

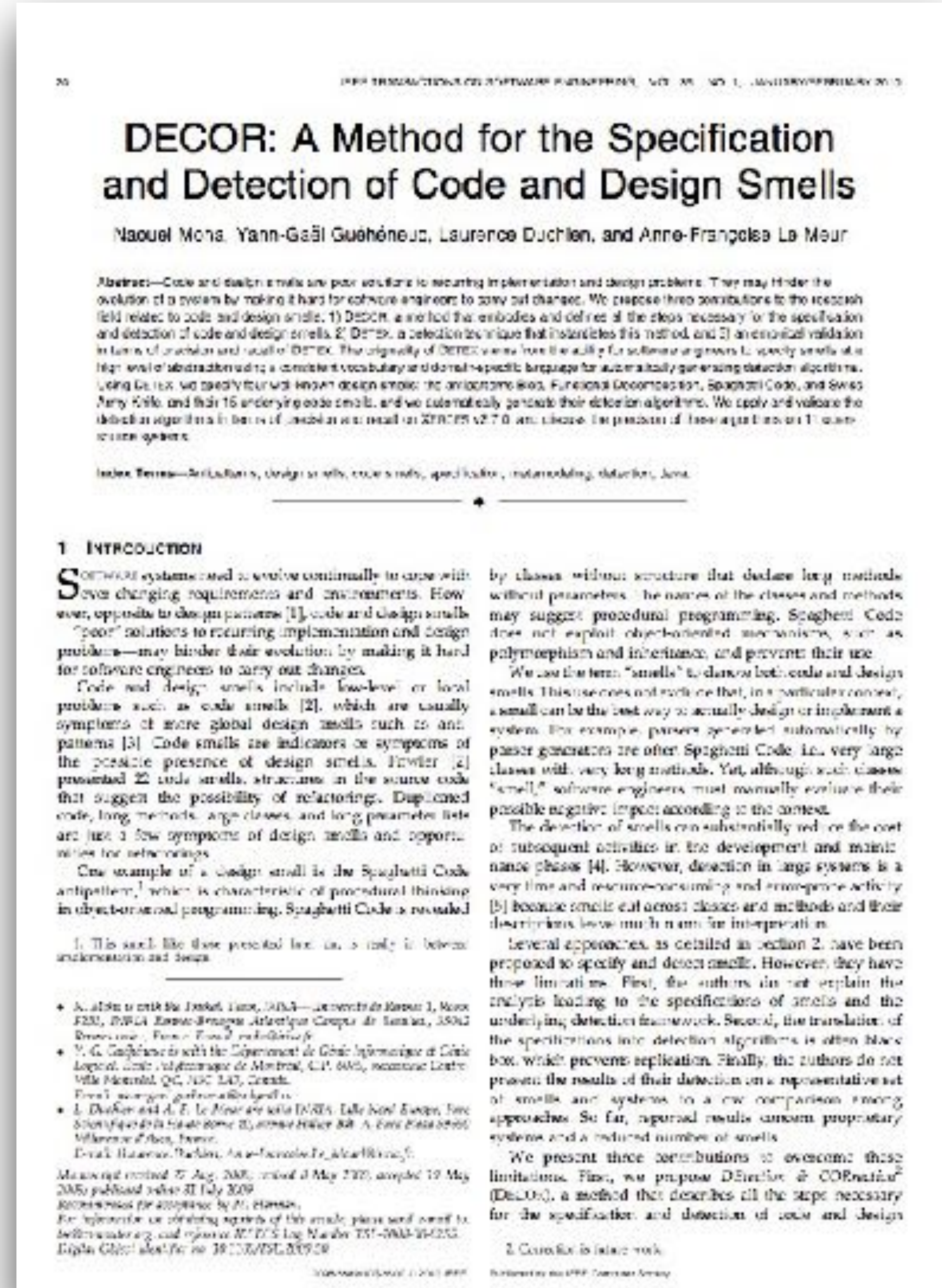
1 INTRODUCTION

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

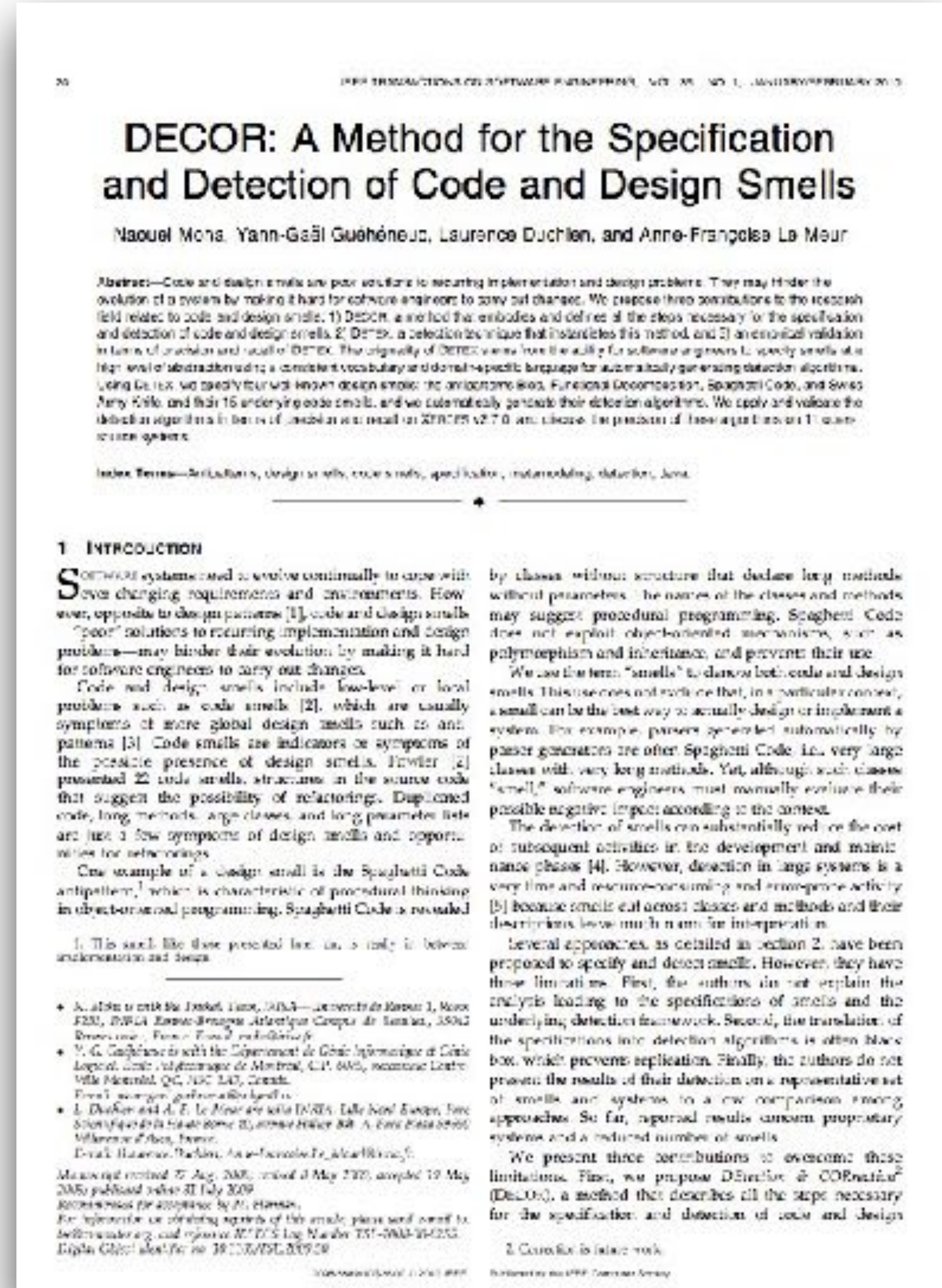
Naouel Moha, Yann-Gaël Guéhéneuc, Laurence Duchien, and Anne-Françoise Le Meur

1 Introduction

- DECOR has been originally validated on one software project;

Bad Code Smells

Bad Code Smells - Where to Refactor



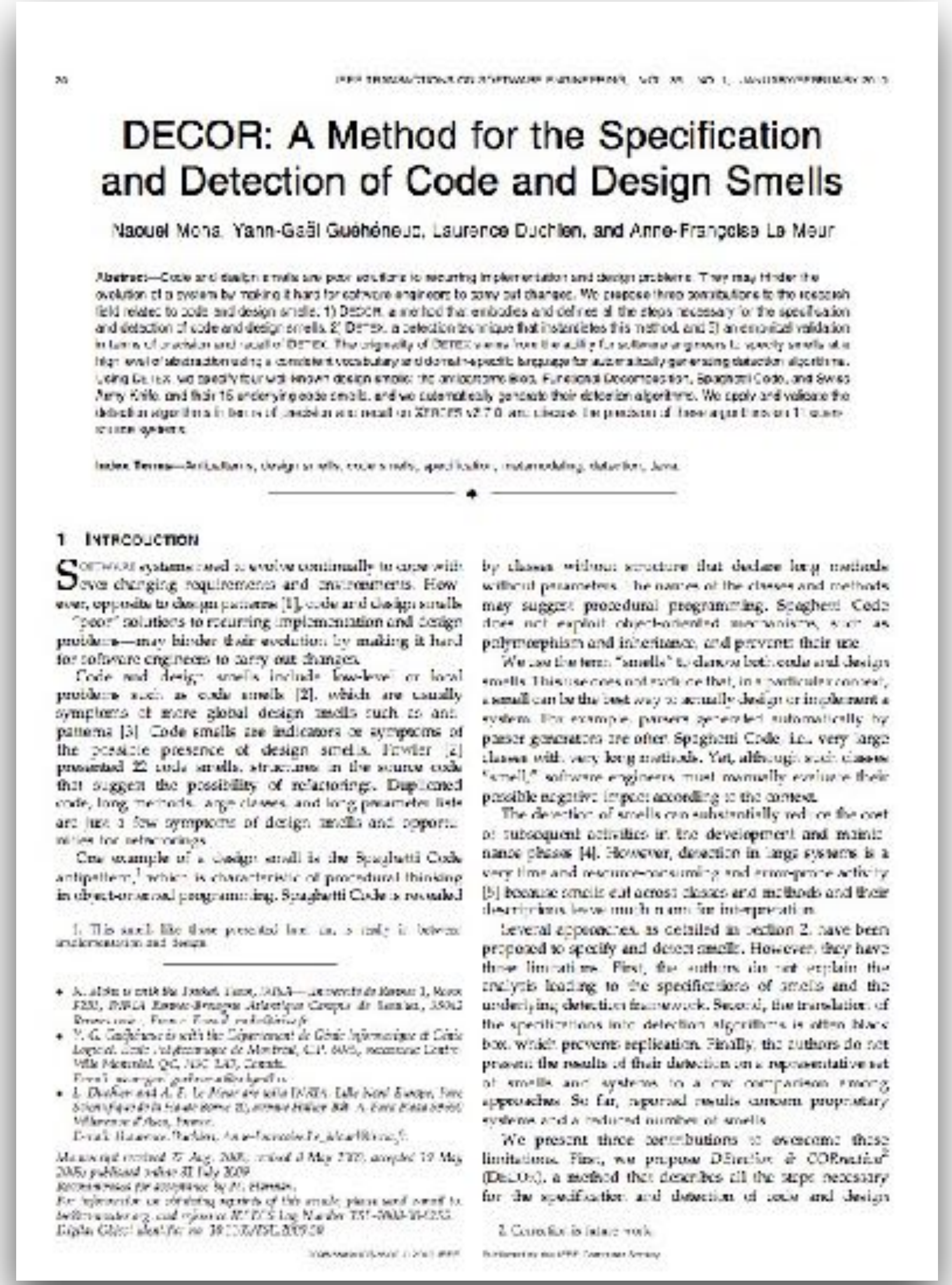
DECOR: A Method for the Specification and Detection of Code and Design Smells

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

1 INTRODUCTION

- 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

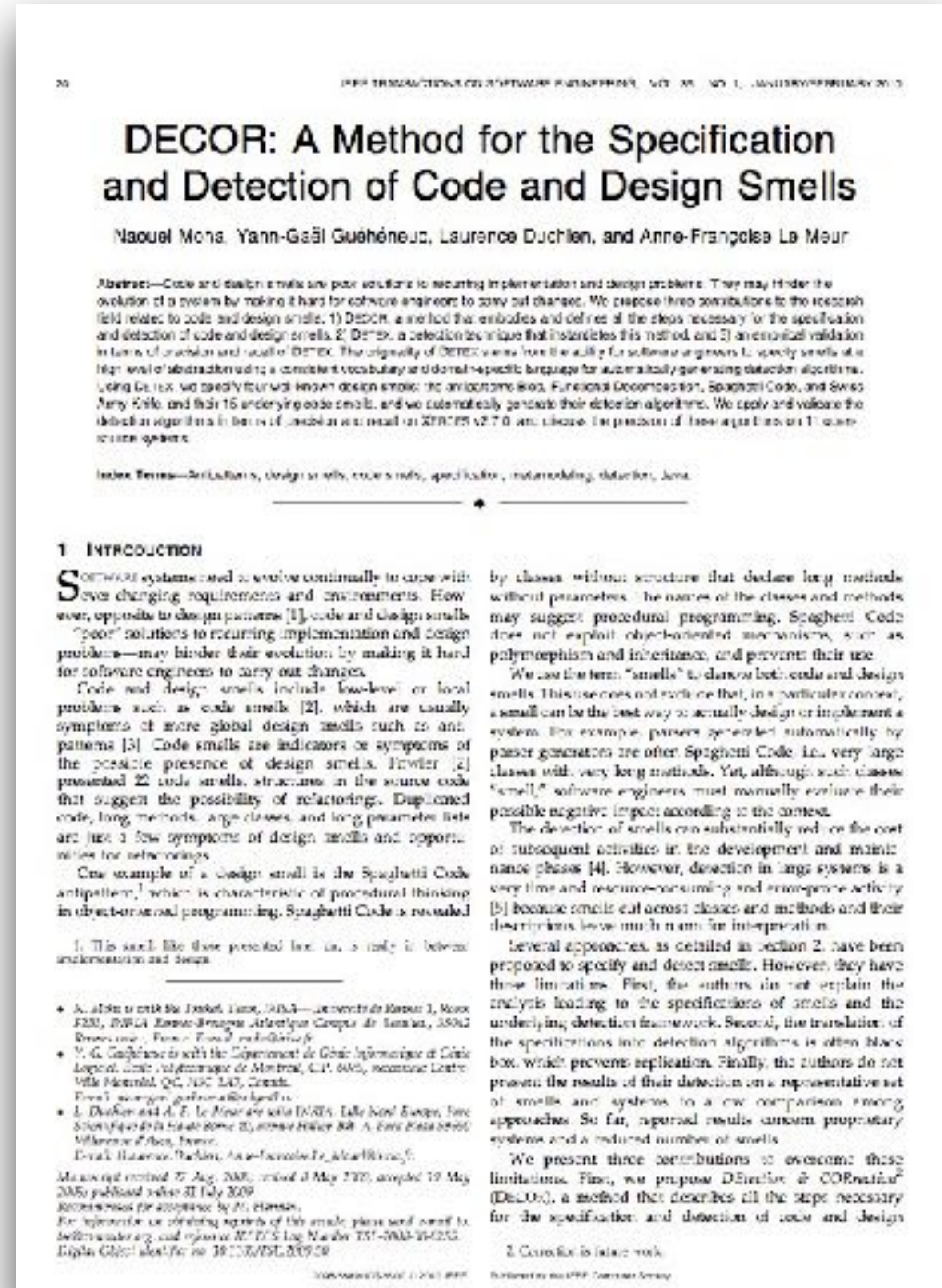

DECOR: A Method for the Specification and Detection of Code and Design Smells

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

1 INTRODUCTION

- 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;

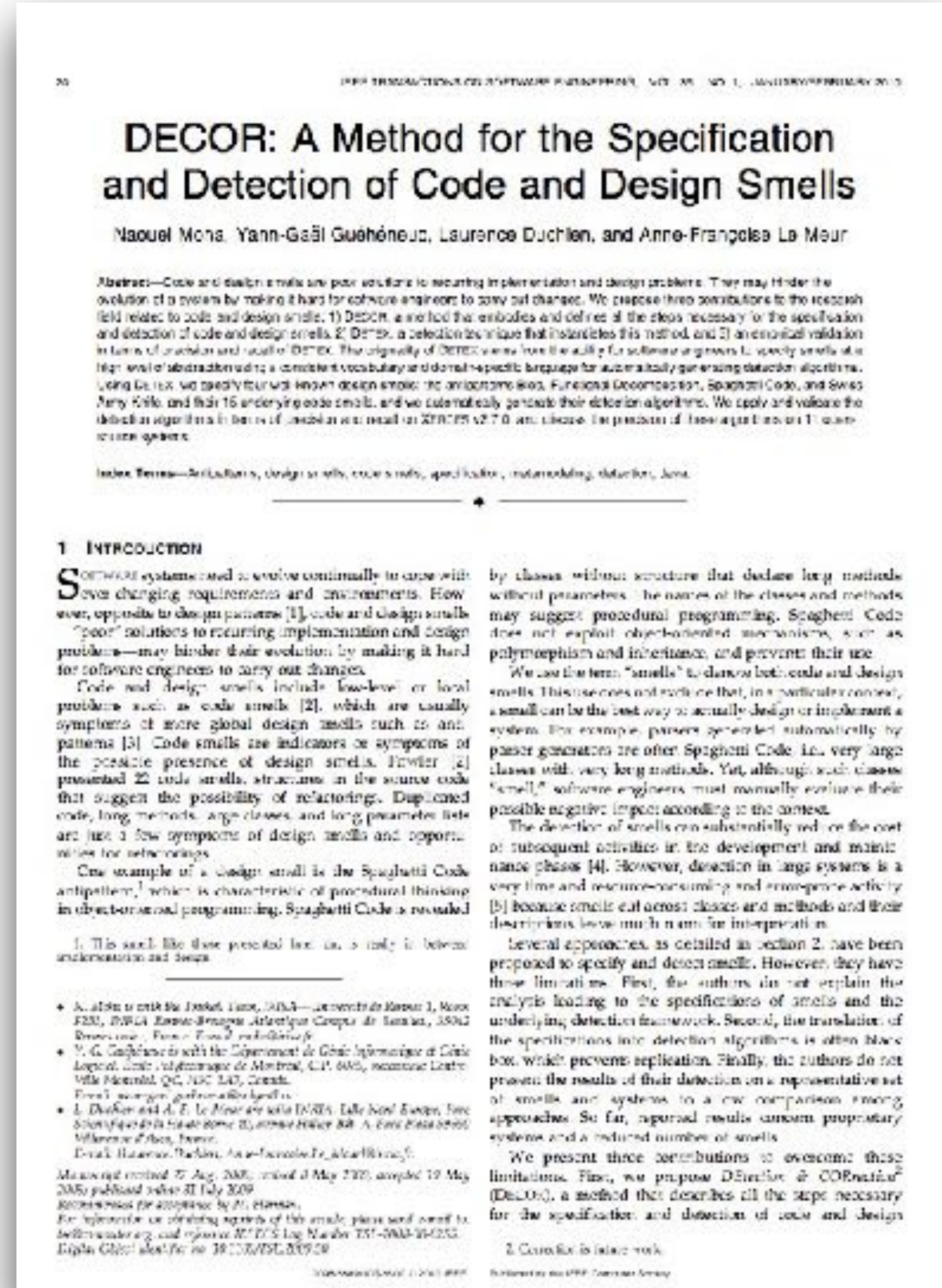
Bad Code Smells - Where to Refactor


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

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

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

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

1 INTRODUCTION

- 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 - 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

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

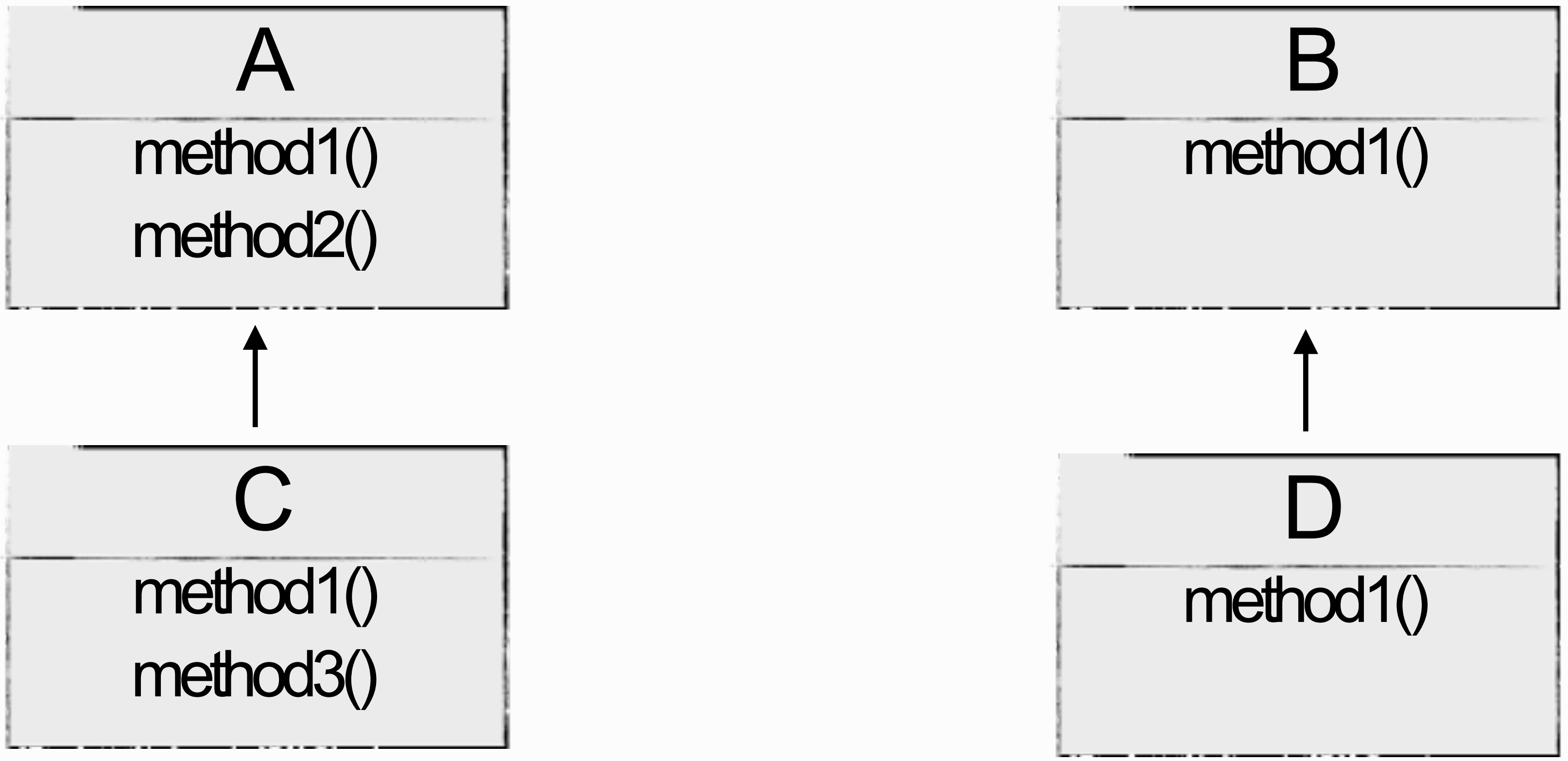
Parallel Inheritance. This code smell appears when, every time a developer adds a subclass of one class, s/he has to add a subclass of another class.

Bad Code Smells - Where to Refactor

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.

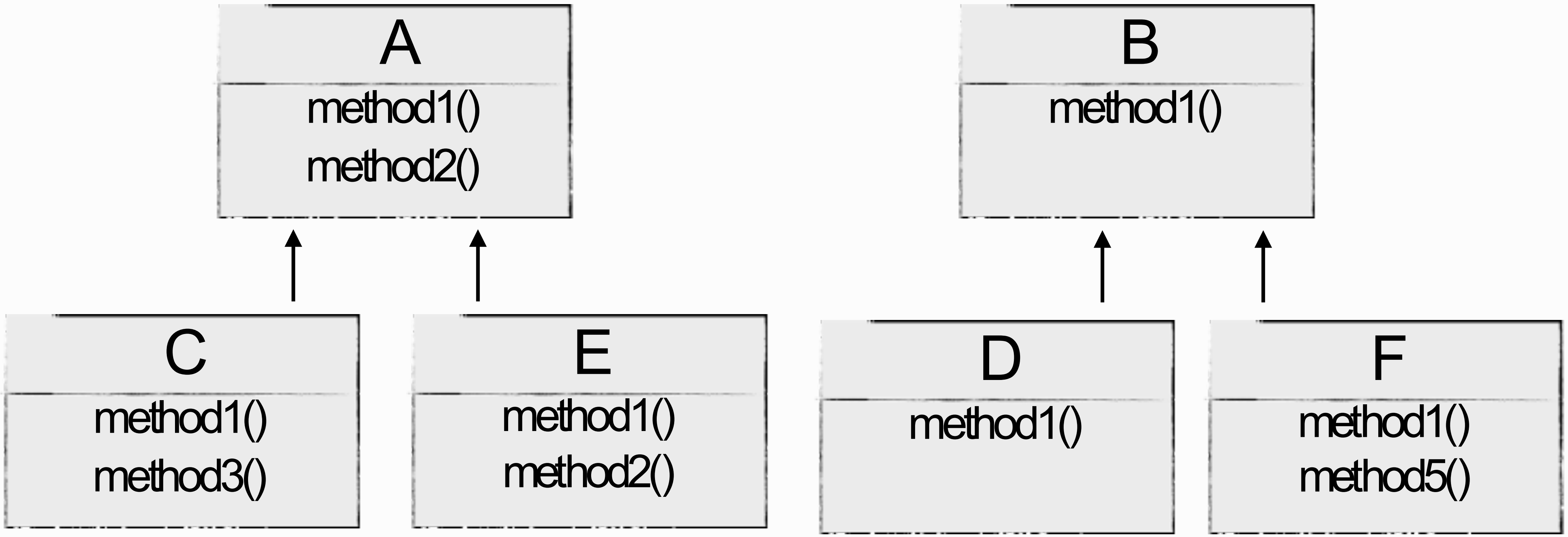


Bad Code Smells - Where to Refactor

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

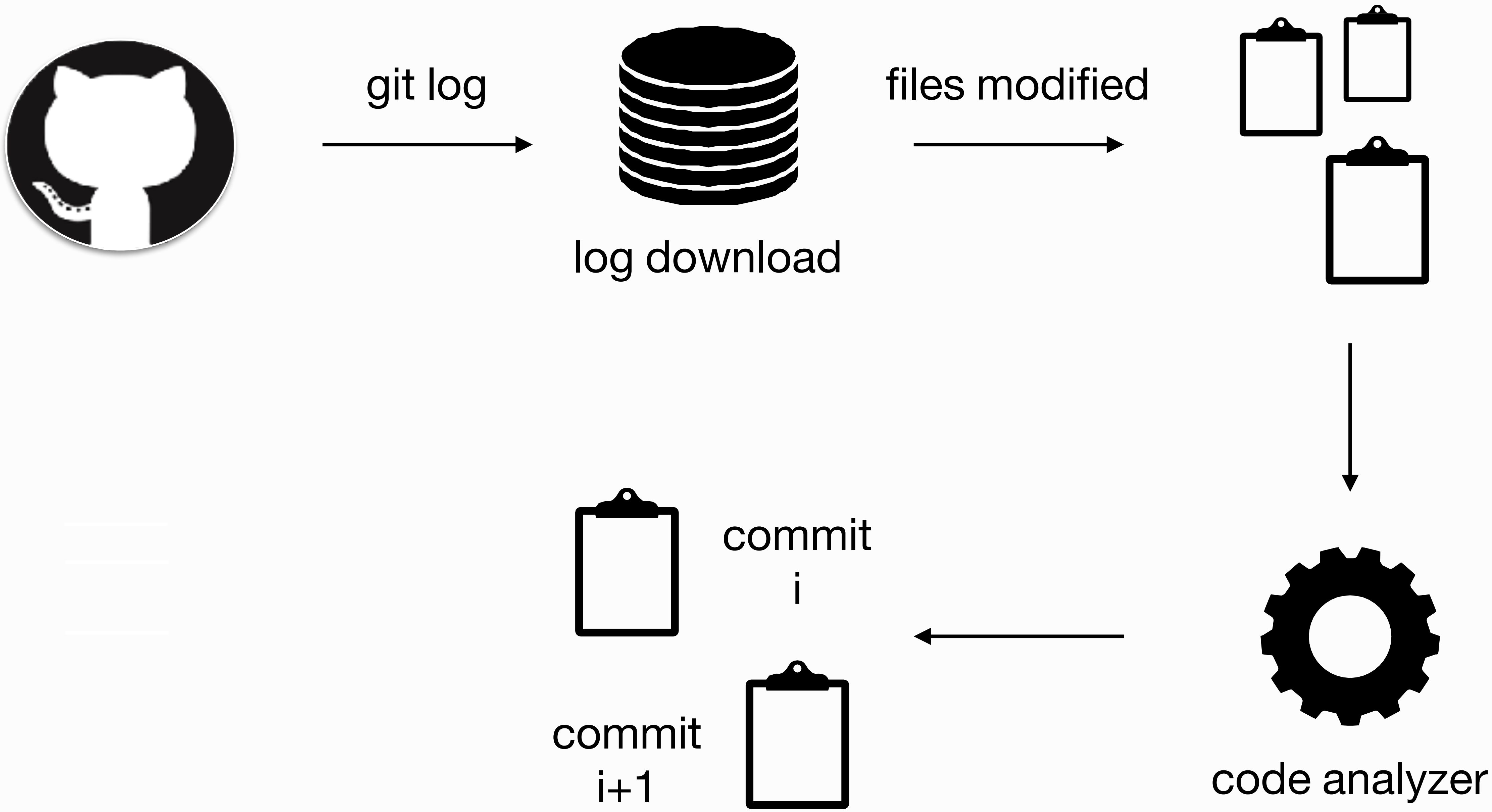
Hence, in some cases the detection of code smells shall make use of change history information. This has been introduced by HIST (Historical Information for Smell deTection).

Bad Code Smells - 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).

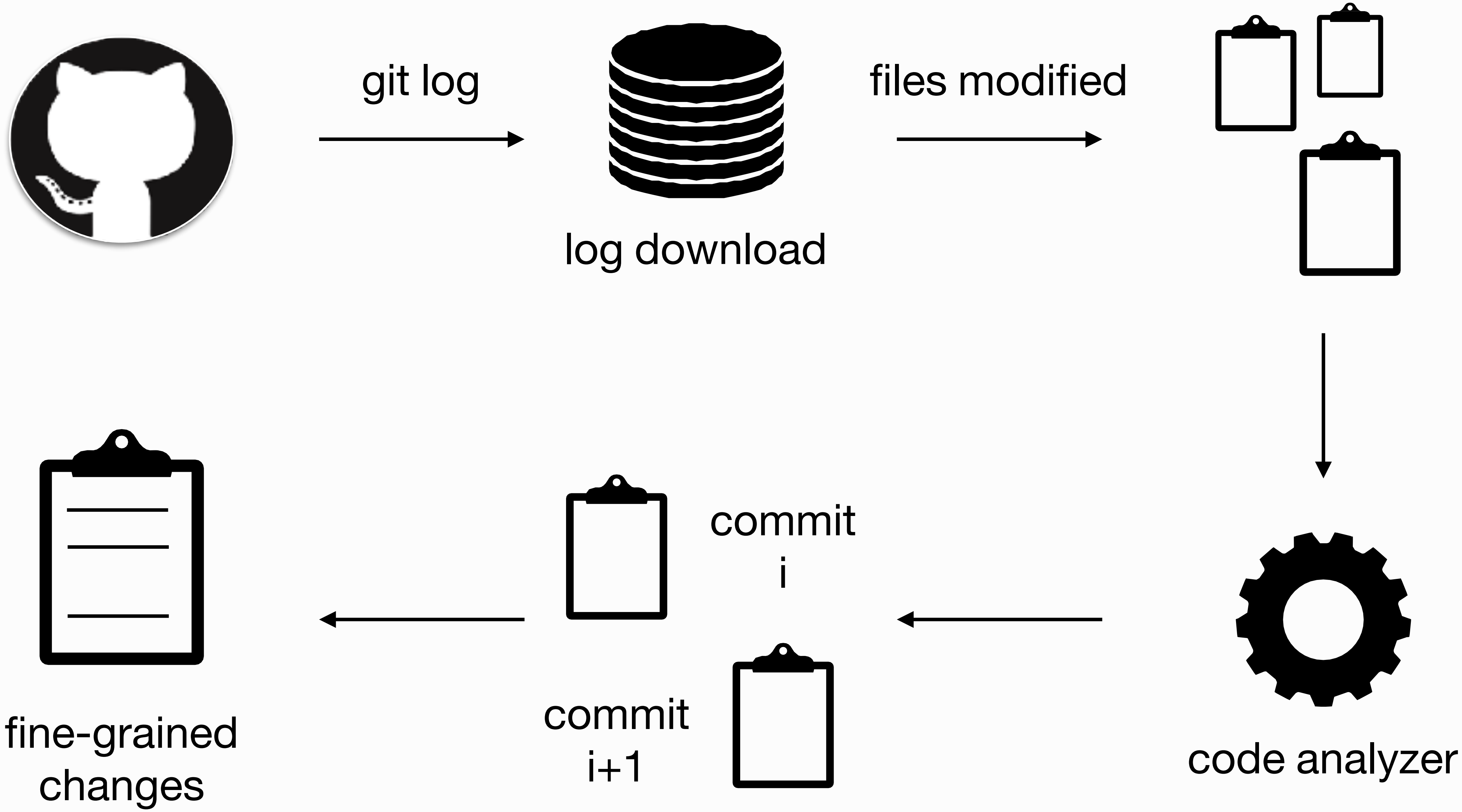
Bad Code Smells - Where to Refactor

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).



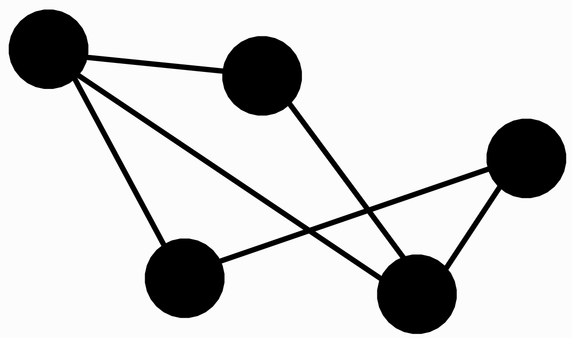
Bad Code Smells - Where to Refactor

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).

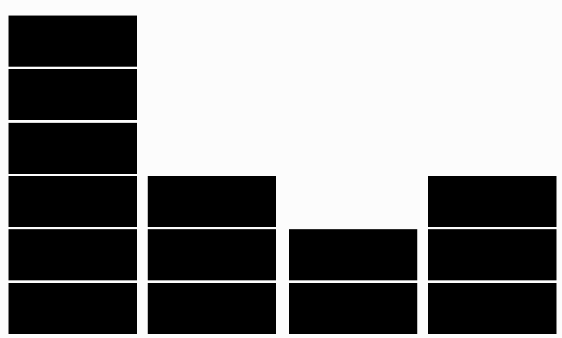


Bad Code Smells - Where to Refactor

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

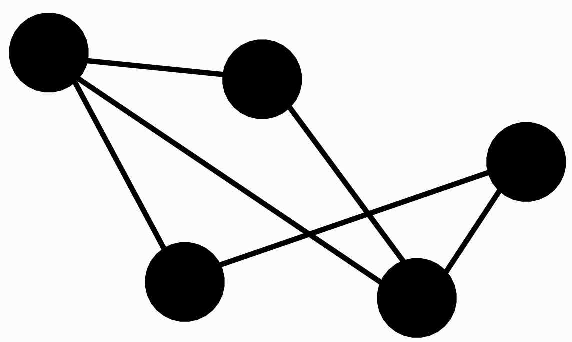


Analysis of change frequency of some specific entities

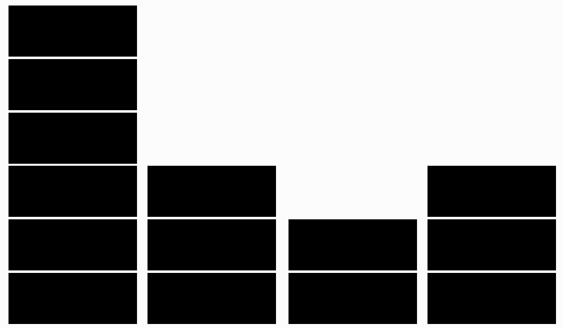
- Once extracted the fine-grained changes, the detector employs two different strategies to identify code smells

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



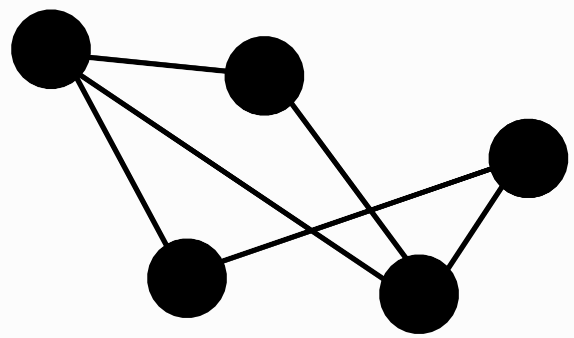
Analysis of change frequency of some specific entities

- 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.

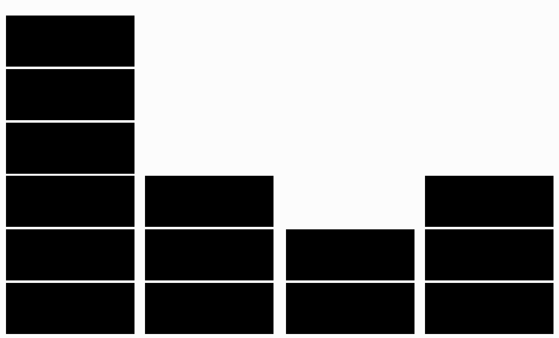
Bad Code Smells

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 - Where to Refactor

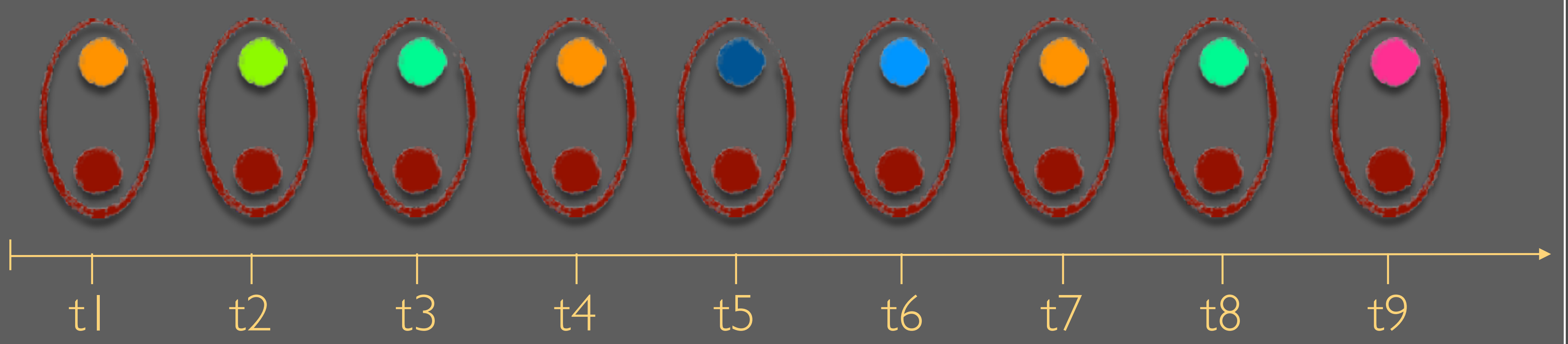
Hence, in some cases the detection of code smells shall make use of change history information. This has been introduced by HIST (Historical Information for Smell deTection).

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).

Mining Version Histories for Detecting Code Smells

Fabio Palomba[1], Gabriele Bavota[2], Massimiliano Di Penta[2], Rocco Oliveto[3], Denys Poshyvanyk[4], Andrea De Lucia[1]

[1]University of Salerno, Fisciano (SA), Italy
[2]University of Sannio, Benevento, Italy
[3]University of Molise, Pesche (IS), Italy
[4]The College of William and Mary, Williamsburg, VA, USA

1 INTRODUCTION

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

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

Fabio Palomba[1], Gabriele Bavota[2], Massimiliano Di Penta[2], Rocco Oliveto[3], Denys Poshyvanyk[4], Andrea De Lucia[1]

[1]University of Salerno, Fisciano (SA), Italy
[2]University of Sannio, Benevento, Italy
[3]University of Molise, Pesche (IS), Italy
[4]The College of William and Mary, Williamsburg, VA, USA

1 INTRODUCTION

- 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).

- 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

Fabio Palomba[1], Gabriele Bavota[2], Massimiliano Di Penta[2], Rocco Oliveto[3], Denys Poshyvanyk[4], Andrea De Lucia[1]

[1]University of Salerno, Fisciano (SA), Italy
[2]University of Sannio, Benevento, Italy
[3]University of Molise, Pesche (IS), Italy
[4]The College of William and Mary, Williamsburg, VA, USA

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

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 change- 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.

There exist a number of approaches for detecting smells in source code to alert developers of their presence [30], [33], [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

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].

- 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

Fabio Palomba[1], Gabriele Bavota[2], Massimiliano Di Penta[2], Rocco Oliveto[3], Denys Poshyvanyk[4], Andrea De Lucia[1]

[1]University of Salerno, Fisciano (SA), Italy
[2]University of Sannio, Benevento, Italy
[3]University of Molise, Pesche (IS), Italy
[4]The College of William and Mary, Williamsburg, VA, USA

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 exploiting change history information to detect instances of five different code smells, namely Divergent Change, Shotgun Surgery, Parallel Inheritance, Blob, and Feature Envy. We evaluate HIST in two empirical studies. 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 [illegible]%, 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 of 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 change- 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.

There exist a number of approaches for detecting smells in source code to alert developers of their presence [30], [33], [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

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].

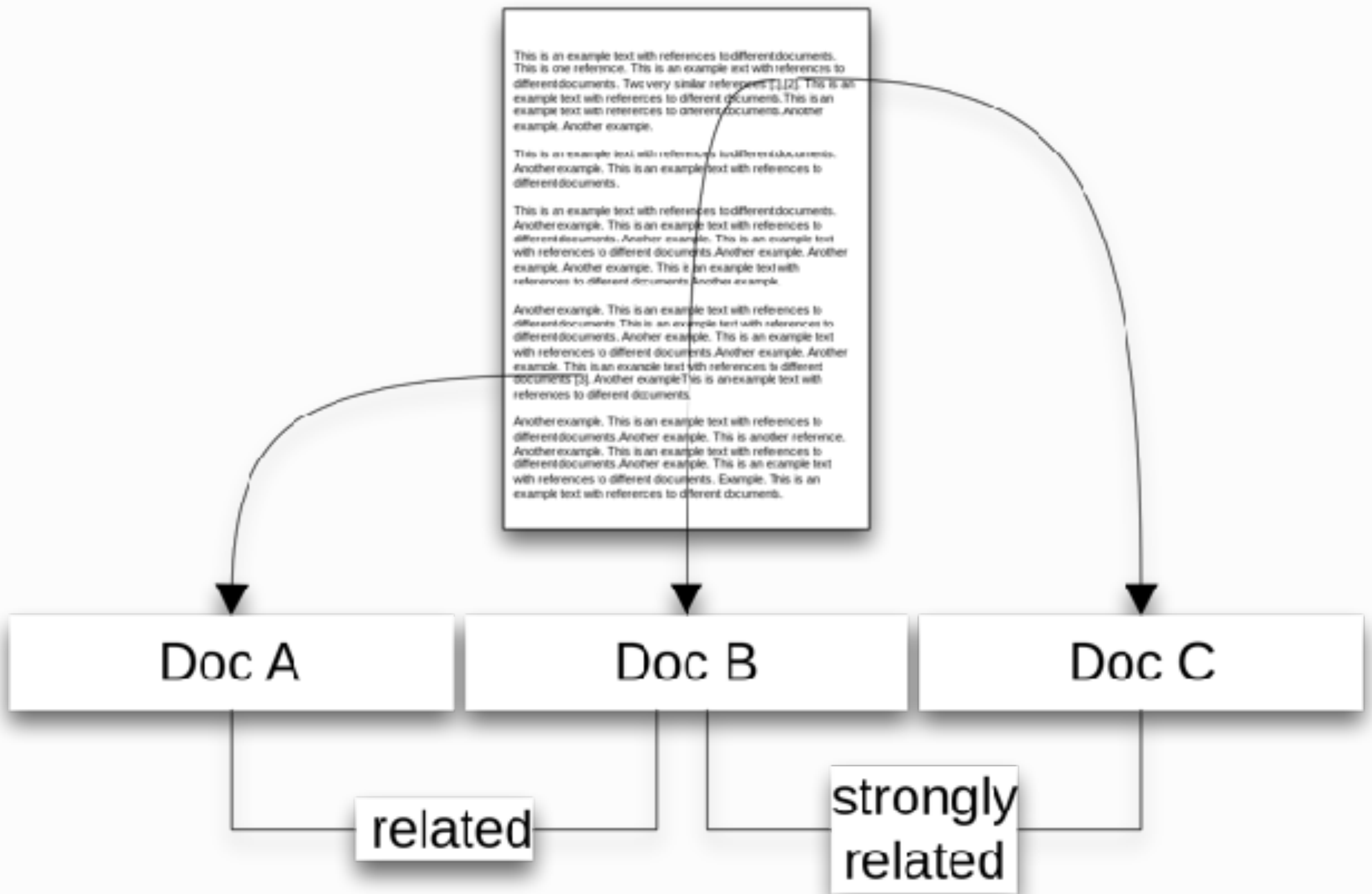
- HIST has been originally validated on twenty open-source software projects;
- According to the achieved results, HIST outperforms structural-based alternatives in terms of precision and recall;
- HIST and the alternative approaches are complementary: they can correctly detect different code smell instances.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

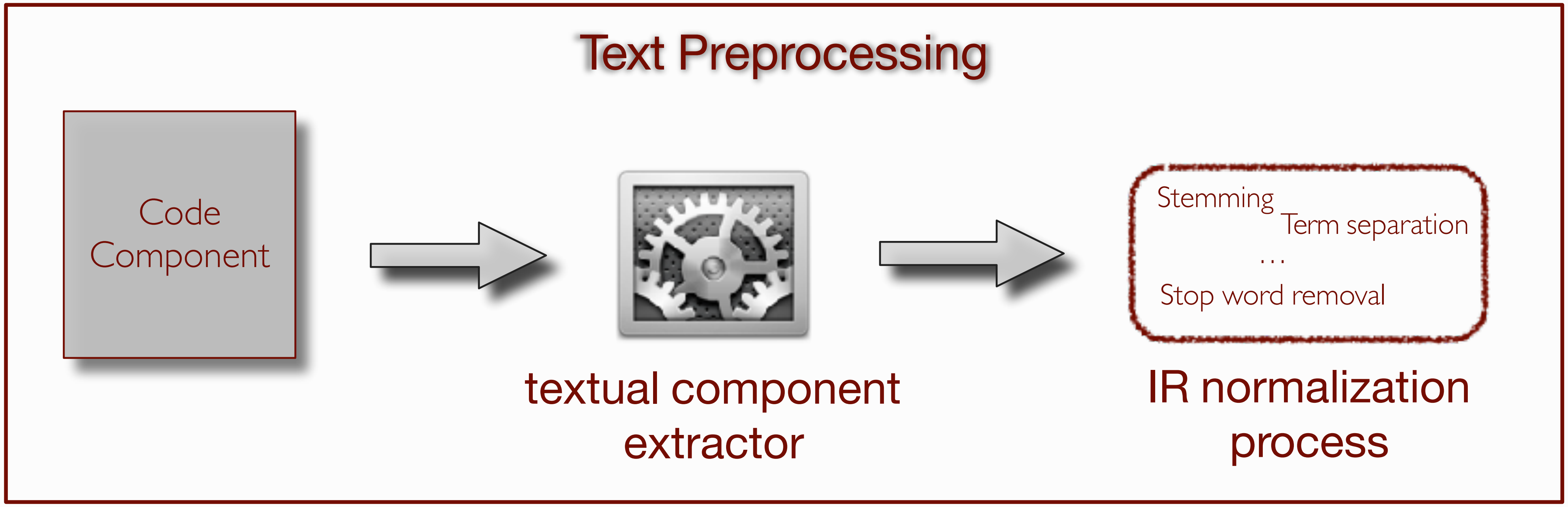
What about **textual analysis**? DECOR introduced this concept for smell detection, yet it limited textual analysis to class and method names.



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

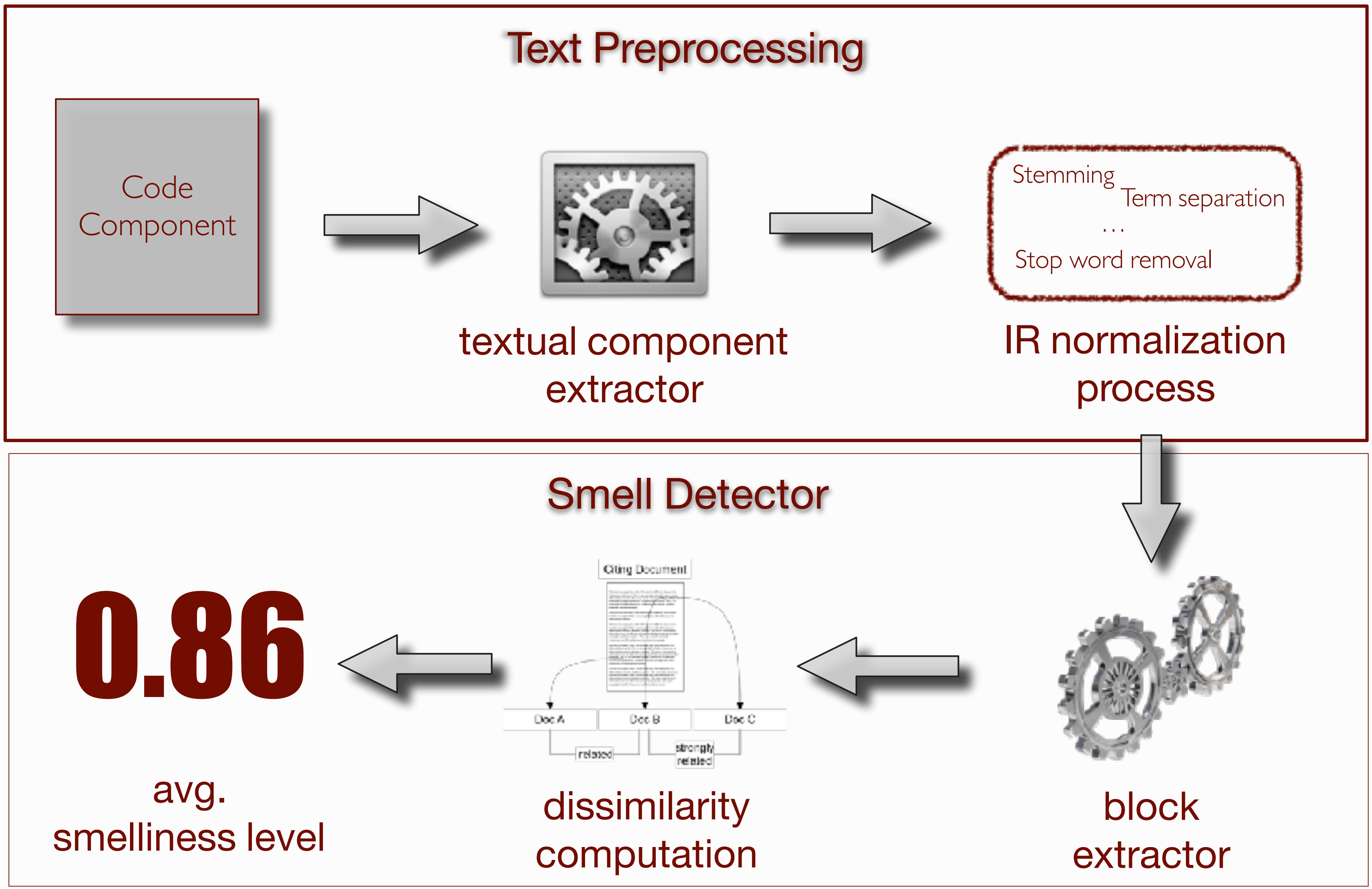
Bad Code Smells - Where to Refactor

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.



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 Palomba[1], Annibale Panichella[2], Andrea De Lucia[1], Rocco Oliveto[3], Andy Zaidman[2]

[1]University of Salerno, Italy – [2]Delft University of Technology, The Netherlands – [3]University of Molise, Italy

- 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 Palomba[1], Annibale Panichella[2], Andrea De Lucia[1], Rocco Oliveto[3], Andy Zaidman[2]
[1]University of Salerno, Italy – [2]Delft University of Technology, The Netherlands – [3]University 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 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.

I. Introduction

- 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 Palomba[1], Annibale Panichella[2], Andrea De Lucia[1], Rocco Oliveto[3], Andy Zaidman[2]
[1]University of Salerno, Italy – [2]Delft University of Technology, The Netherlands – [3]University of Molise, Italy

- 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 - 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.

```
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.

```
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.

A Textual-based Technique for Smell Detection

Fabio Palomba[1], Annibale Panichella[2], Andrea De Lucia[1], Rocco Oliveto[3], Andy Zaidman[2]

[1]University of Salerno, Italy – [2]Delft University of Technology, The Netherlands – [3]University of Molise, Italy

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

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 Palomba[1], Annibale Panichella[2], Andrea De Lucia[1], Rocco Oliveto[3], Andy Zaidman[2]
[1]University of Salerno, Italy – [2]Delft University of Technology, The Netherlands – [3]University of Molise, Italy

- 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 Palomba[1], Annibale Panichella[2], Andrea De Lucia[1], Rocco Oliveto[3], Andy Zaidman[2]
[1]University of Salerno, Italy – [2]Delft University of Technology, The Netherlands – [3]University of Molise, Italy

- 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 Palomba[1], Annibale Panichella[2], Andrea De Lucia[1], Rocco Oliveto[3], Andy Zaidman[2]

[1]University of Salerno, Italy – [2]Delft University of Technology, The Netherlands – [3]University of Molise, Italy

- 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 Palomba[1], Annibale Panichella[2], Andrea De Lucia[1], Rocco Oliveto[3], Andy Zaidman[2]
[1]University of Salerno, Italy – [2]Delft University of Technology, The Netherlands – [3]University of Molise, Italy

- 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

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.


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

Dario Di Nucci[1,2], Fabio Palomba[3], Damian A. Tamburri[4], Alexander Serebrenik[4], Andrea De Lucia[1]

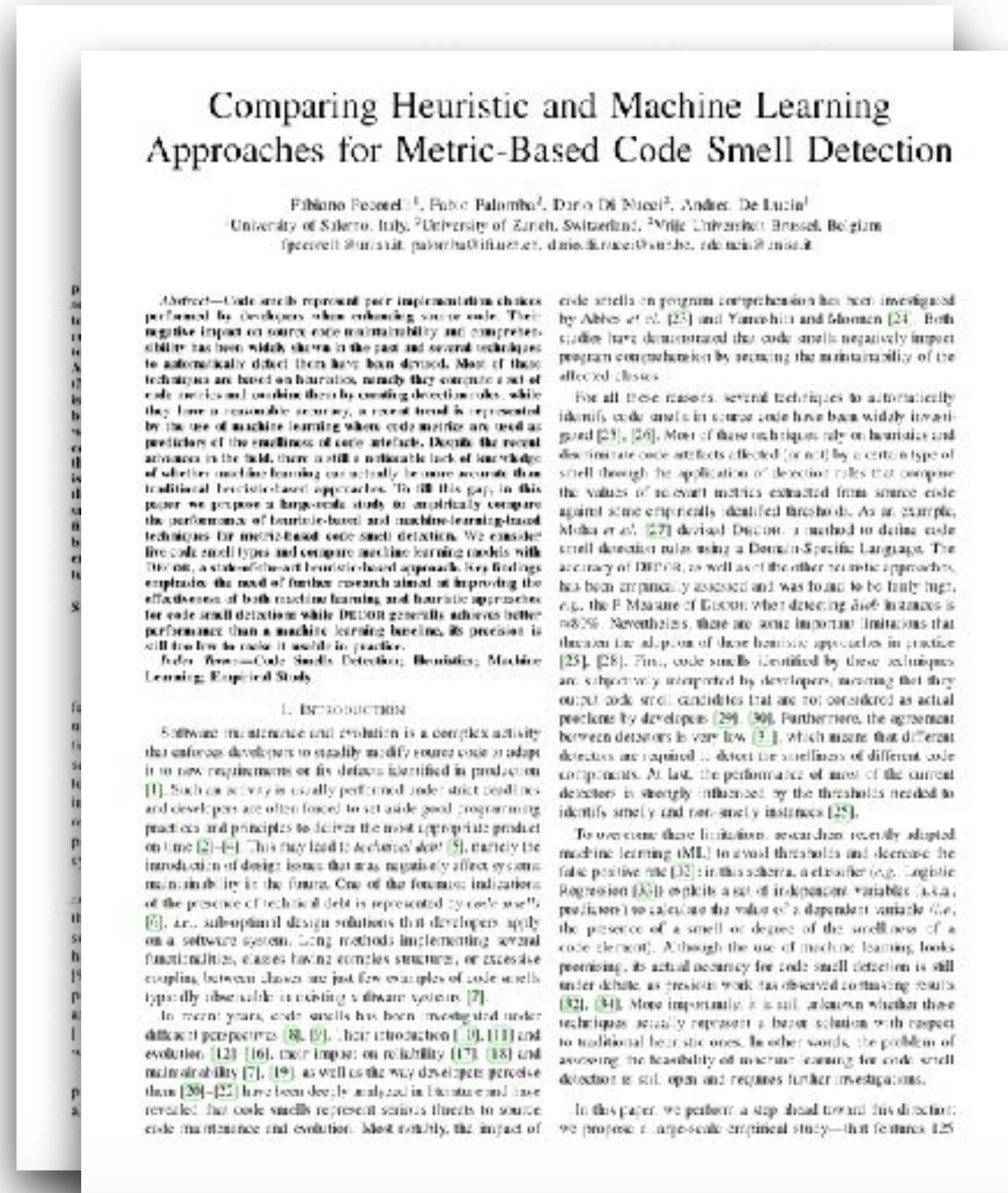
[1]University of Salerno, Italy — [2]Vrije Universiteit Brussel, Belgium

[3]University of Zurich, Switzerland — [4]Eindhoven University of Technology, The Netherlands

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

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

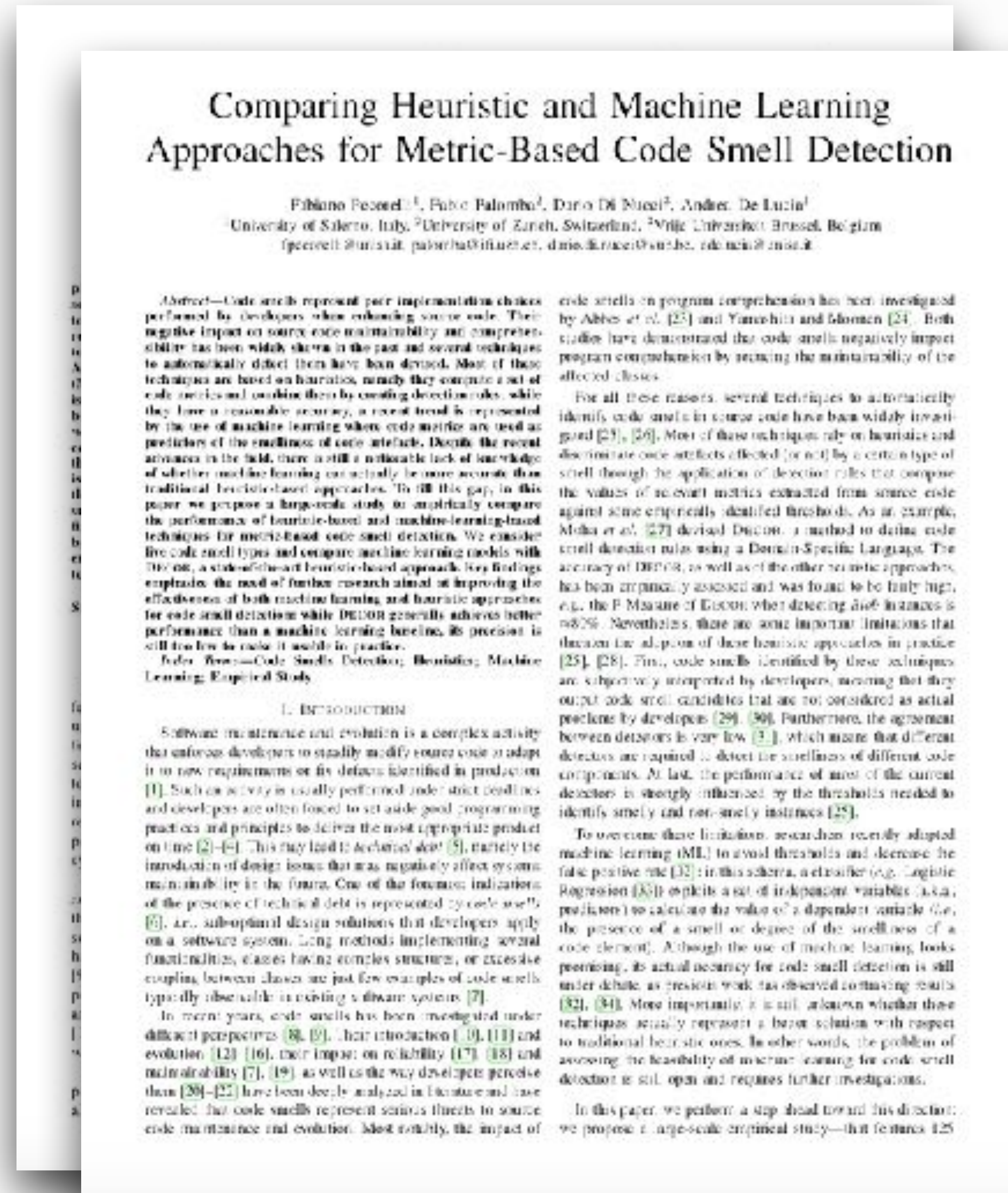
Fabiano Pecorelli[1], Fabio Palomba[1], Dario Di Nucci[2], Andrea De Lucia[1]

I. Introduction

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

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.

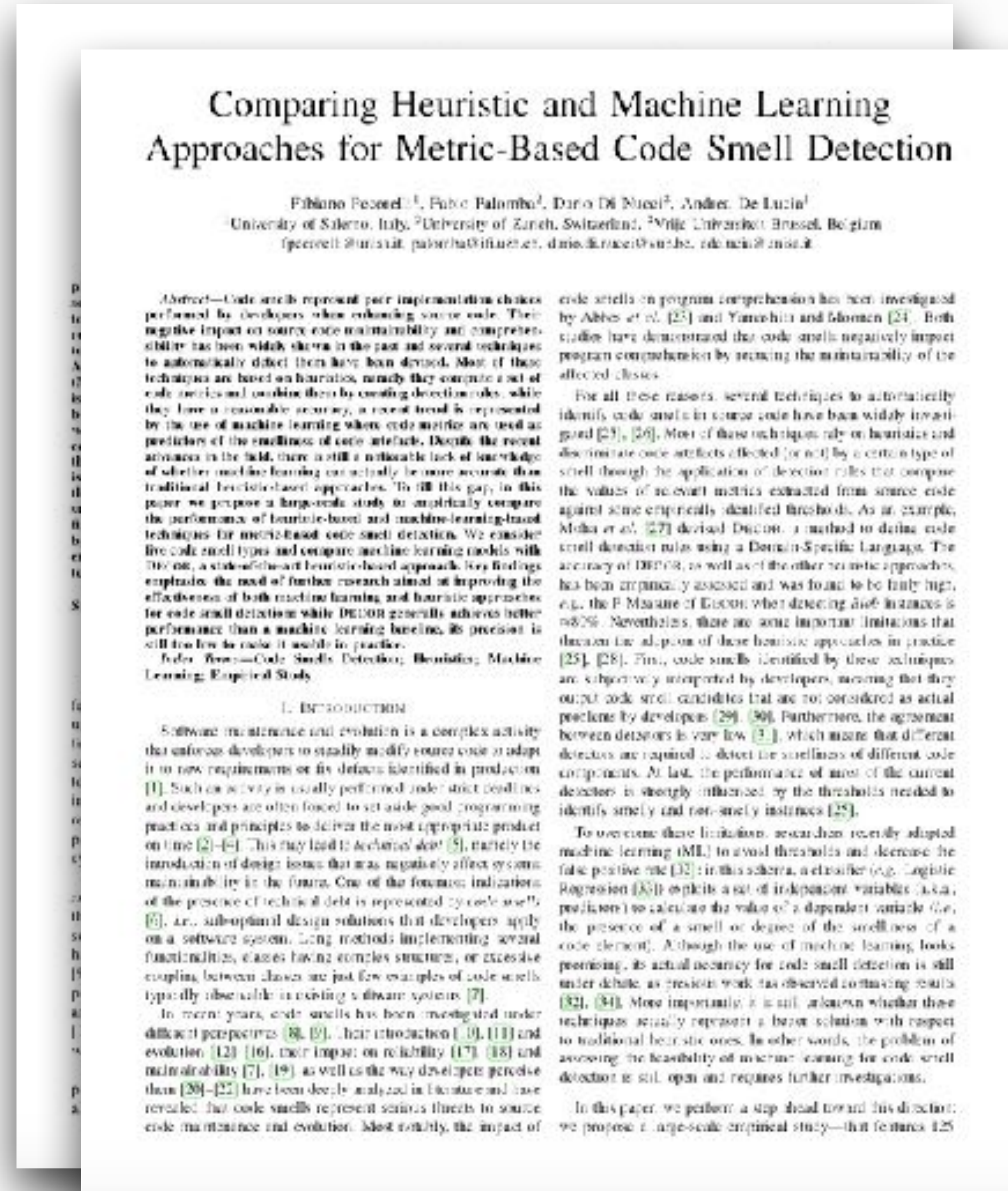


Comparing Heuristic and Machine Learning Approaches for Metric-Based Code Smell Detection

- 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

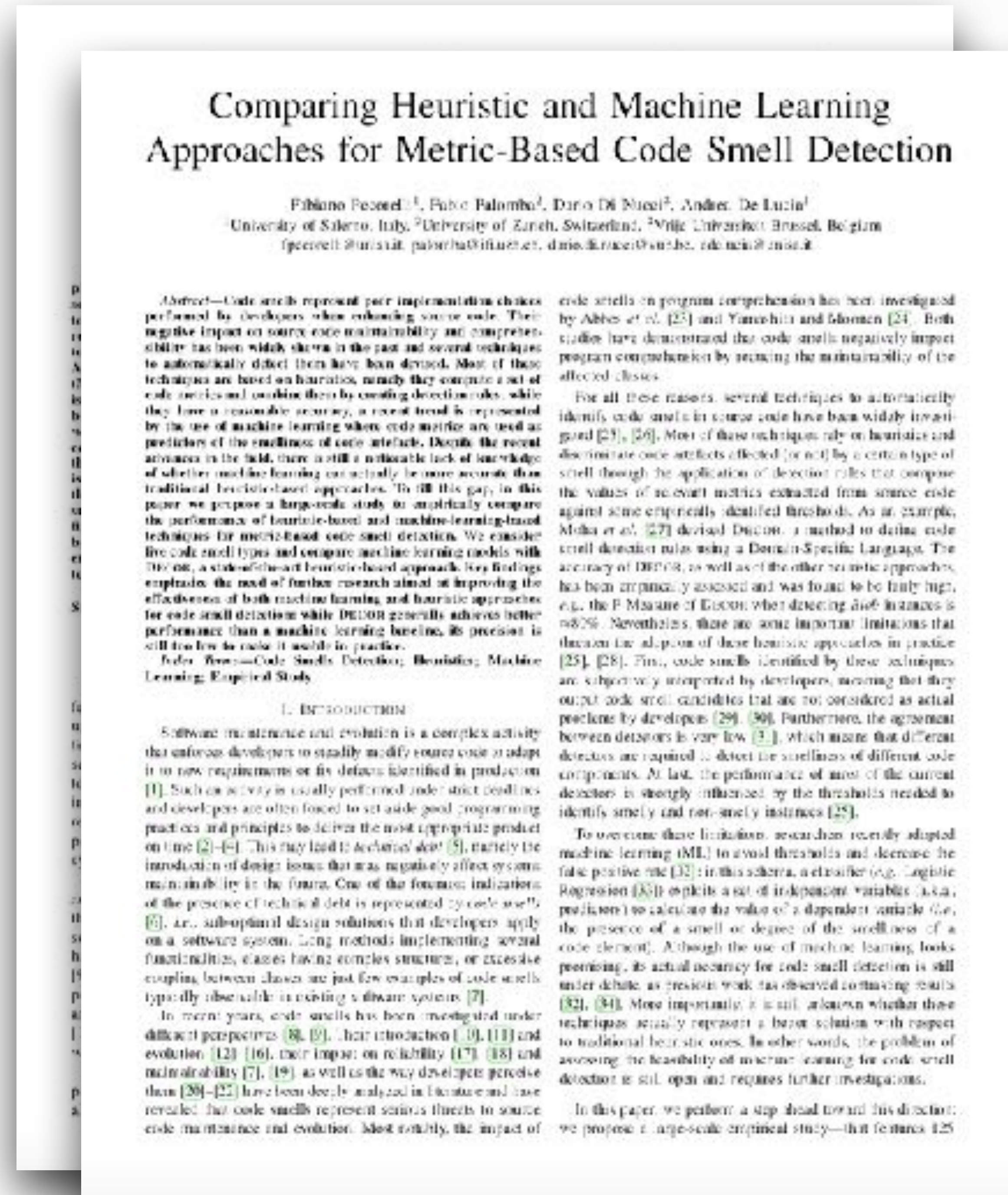
Theoretically speaking, machine learning has the potential to address all the limitations of currently available code smell detection techniques.


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

- 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

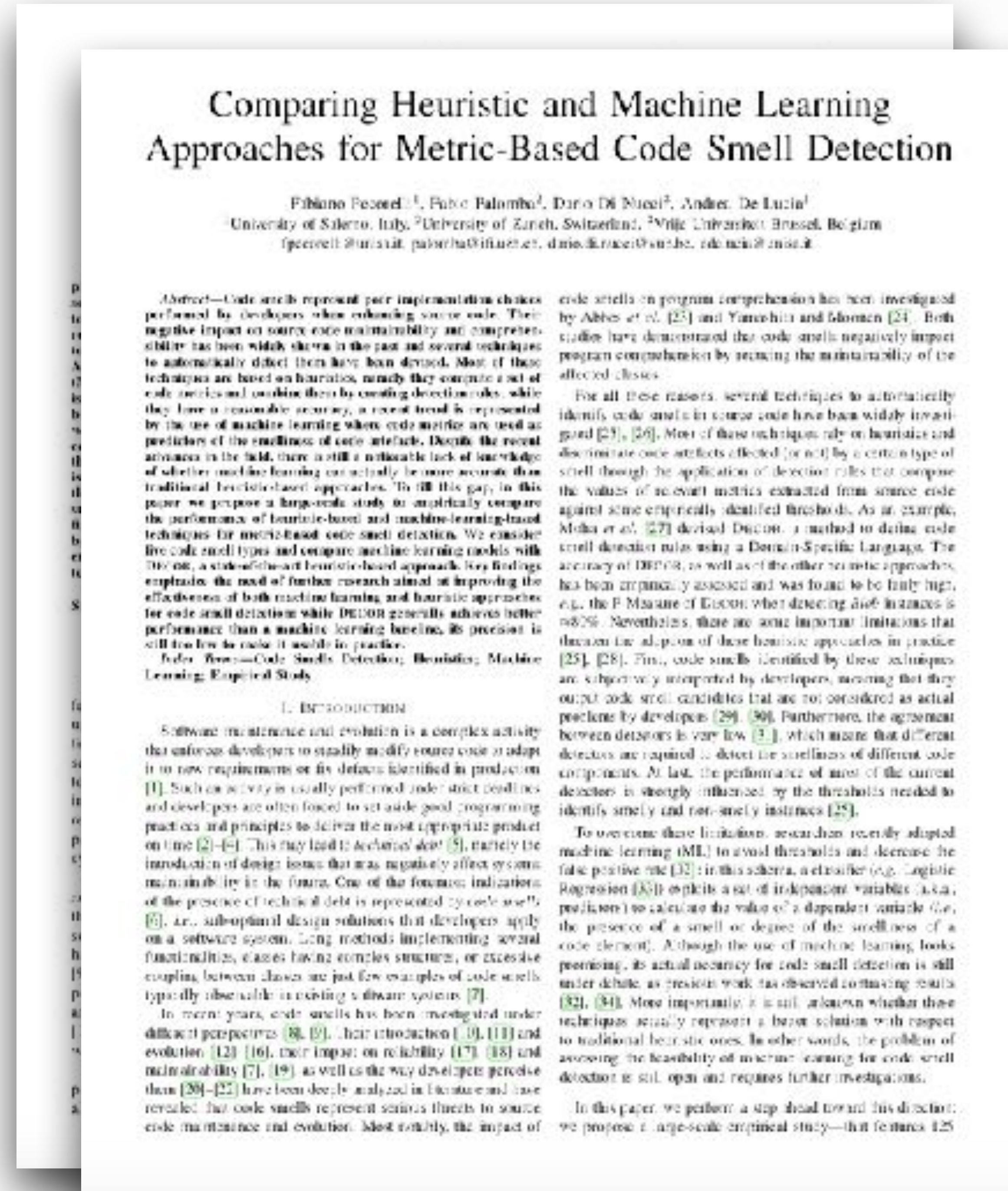
Fabiano Pecorelli, Fabio Palomba, Dario Di Nucci, Andrea De Lucia

- 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

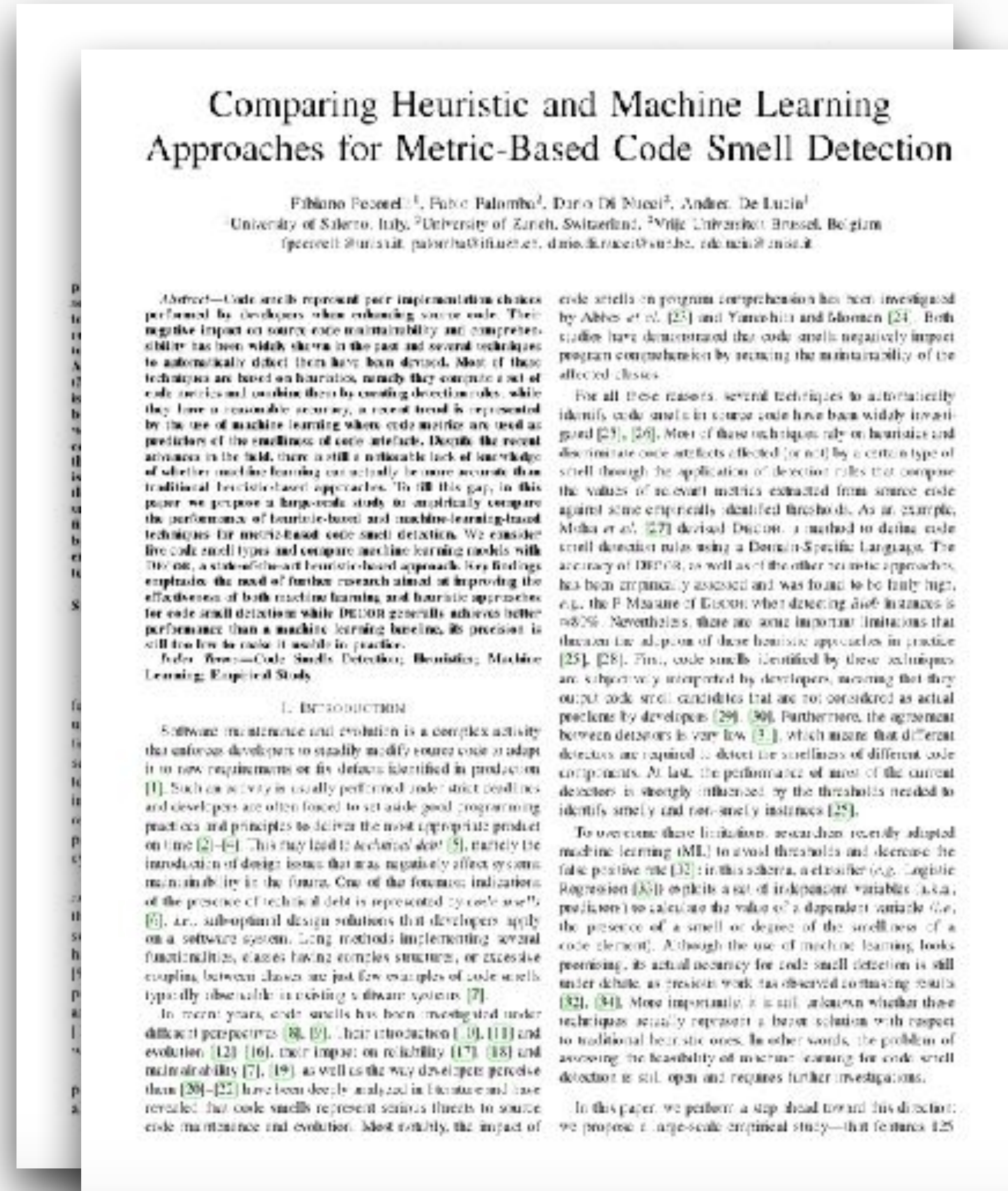
- The composition of the training set can bias the performance by up to 90%.
- Machine learning models **do not** perform better than heuristic ones.
- Machine learning models **work similarly to** a random approach.
- Machine learning models **work worse than** a pessimistic classifier.

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 Pecorelli, Fabio Palomba, Dario Di Nucci, Andrea De Lucia

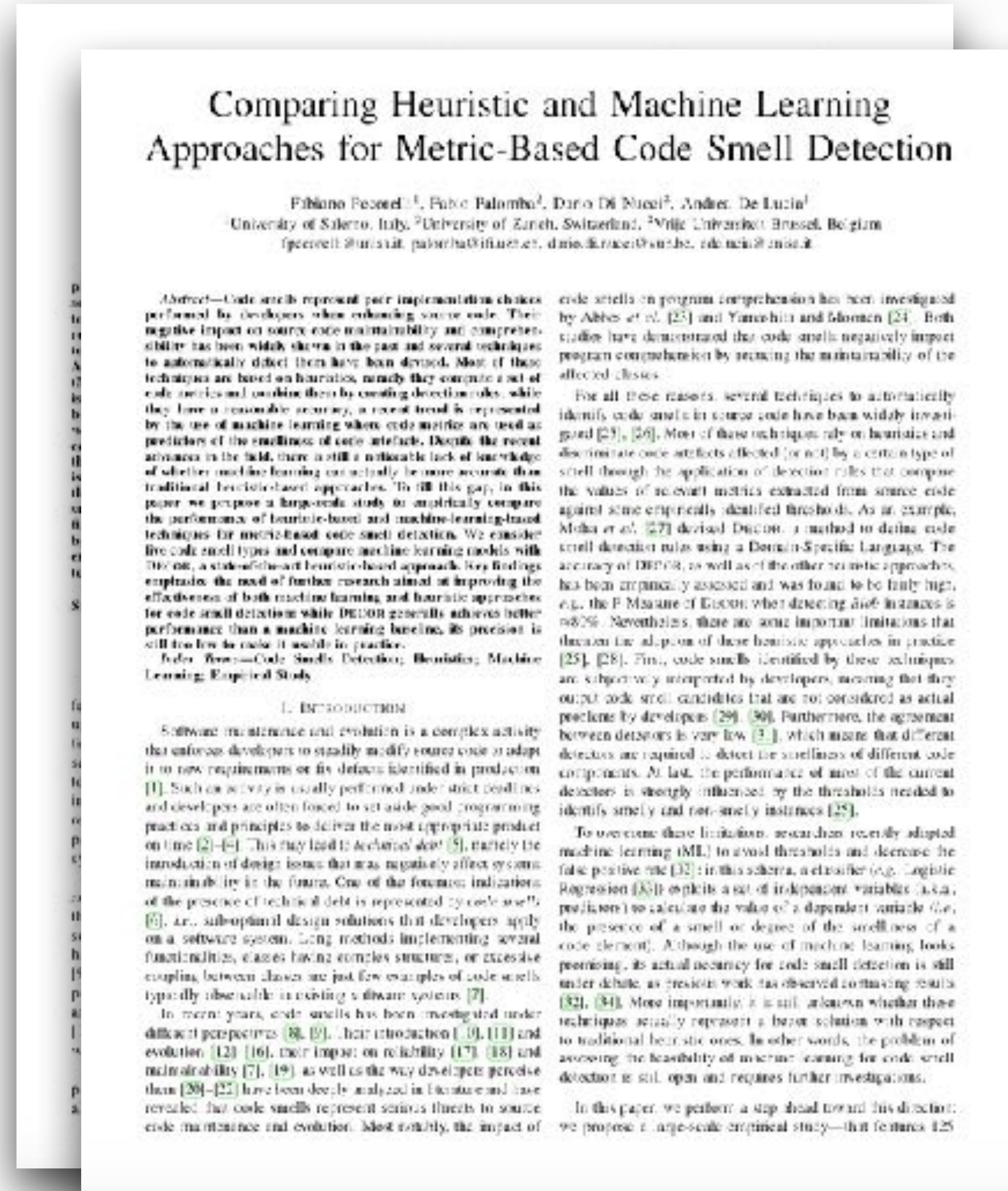
- The composition of the training set can bias the performance by up to 90%.
- Machine learning models **do not** perform better than heuristic ones.
- Machine learning models **work similarly to** a random approach.
- Machine learning models **work worse than** a pessimistic classifier.

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

- 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.

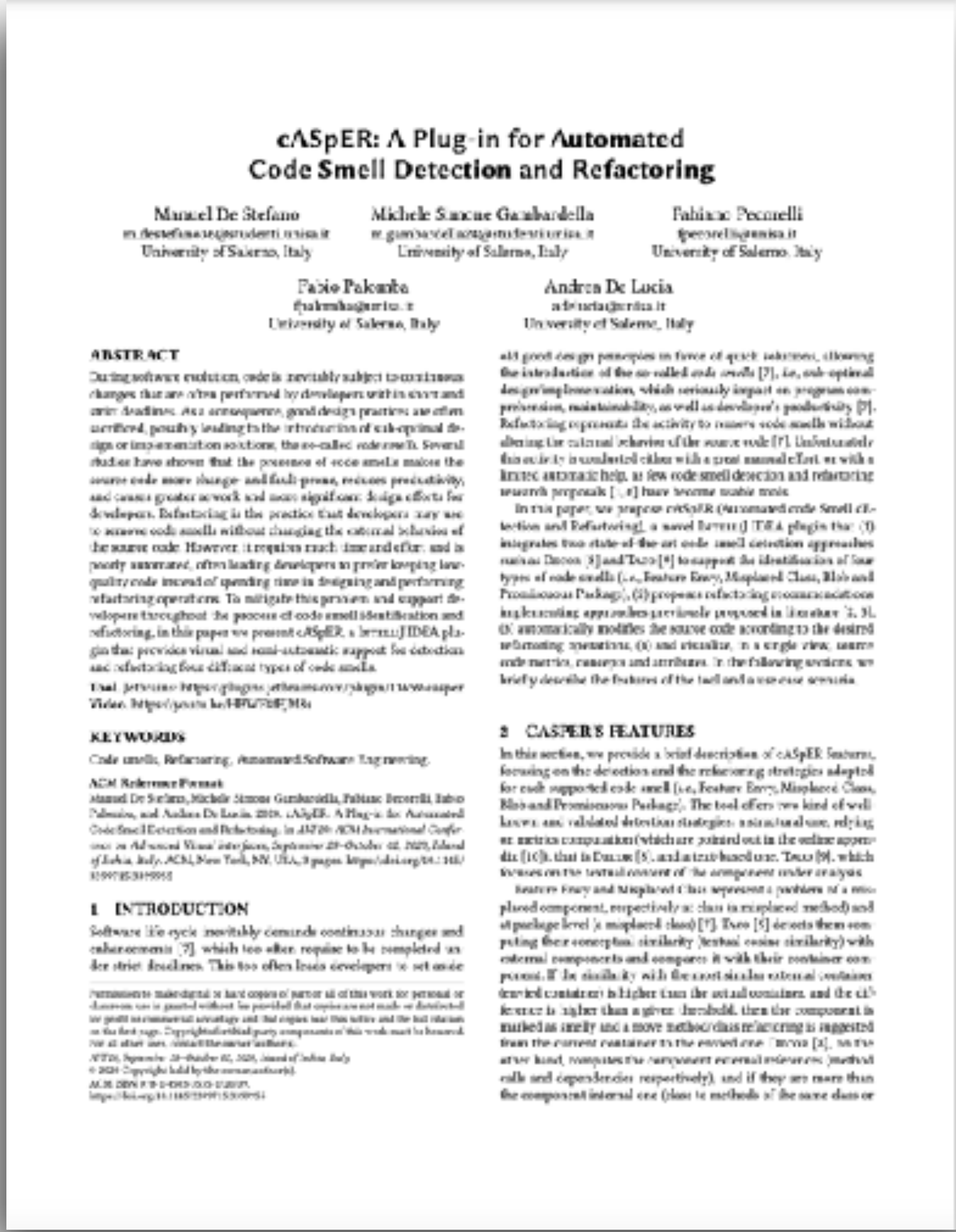
- They can be detected in multiple ways and using different metrics.
- Most of the techniques defined so far are based on structural analysis of source code, yet alternative sources of information seem to be even more efficient and promising - especially because of their orthogonality.
- There are, still,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



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

Manuel De Stefano, University of Salerno, Italy
Michele Simone Gambardella, University of Salerno, Italy
Fabiano Pecorelli, University of Salerno, Italy
Fabio Palomba, University of Salerno, Italy
Andrea De Lucia, University of Salerno, Italy

ABSTRACT

KEYWORDS

1 INTRODUCTION

2 CASPER'S FEATURES

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

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

Code smells: detection & refactoring

Slides by Prof. Fabio Palomba