



GSDetector: a tool for automatic detection of bad smells in GRL goal models

Mawal A. Mohammed¹ · Jameleddine Hassine¹ · Mohammad Alshayeb¹

Accepted: 7 June 2022 / Published online: 13 August 2022

© The Author(s), under exclusive licence to Springer-Verlag GmbH Germany, part of Springer Nature 2022

Abstract

Goal models play a significant role in the early stages of the requirements engineering process. These models are subject to quality problems (a.k.a., bad smells) that may disseminate to the later stages of the requirements engineering process and even to the other stages in software development. To avoid this negative impact, it is important to detect and correct these problems as early as possible. However, the manual detection of these smells is generally tedious, cumbersome, and error-prone. In this paper, we report on an Eclipse plugin tool, called GSDetector (GRL Smells Detector), that automates the detection of Goal-oriented Requirements Language (GRL) bad smells. We first introduce and articulate four new GRL-based bad smells. To detect the instances of these smells, a set of metric-based rules is introduced. Factors that affect setting thresholds are also presented and explained to help modelers specify these rules by setting effective thresholds. GSDetector was evaluated using 5 case studies of different sizes that consider the different scenarios in building GRL models. The obtained results show that GSDetector was able to detect all the existing instances of bad smells with respect to the specified thresholds. The manual inspection of these instances revealed that the modelers were giving the system to be developed more attention than the strategic needs of the stakeholder leading to the appearance of these instances. In conclusion, the proposed bad smells and developed tool provide a useful approach to help identify and analyze quality improvement opportunities in GRL models.

Keywords Bad smells · Goal models · GRL · GSDetector

1 Introduction

Requirement engineering is the first and most important stage in the software development process. The objective of this stage is to elicit, manage, and document system requirements making sure that they are complete, correct, and concise [1]. The outcome of this stage is a set of requirements artifacts that can be categorized into three levels in terms of abstraction: goals, scenarios, and solution-based requirements [1].

Goal artifacts are developed in the early stage of the requirements engineering process to model the strategic alignment of system requirements with the objectives of the key stakeholders of the system. The quality of these artifacts has a significant impact on the later stages of the software development process. The problems in goal artifacts, often, propagate to the subsequent requirements artifacts, and downward the software development process, to the design and implementation artifacts [2, 3]. Thus, the earlier the problems are discovered, the lower the cost of rectifying them [4, 5]. The quality problems of goal artifacts are critical because goal artifacts are concerned with the system's context, and the most expensive requirements problems to rectify, in terms of time and resources, are context problems [6]. The impact of requirements quality problems also goes beyond the software development process to reach other elements in the organization's ecosystem, including customer dissatisfaction and damaging the company's reputation [7].

Goals are used to model the transformation from the business domain to the system domain by linking the objectives

✉ Mohammad Alshayeb
alshayeb@kfupm.edu.sa

Mawal A. Mohammed
g201102570@kfupm.edu.sa

Jameleddine Hassine
jhassine@kfupm.edu.sa

¹ Information and Computer Science Department,
Interdisciplinary Research Center for Intelligent Secure
Systems, King Fahd University of Petroleum and Minerals,
Dhahran 31261, Saudi Arabia

of the system's stakeholders to the features of the system to be developed [8, 9]. Hence, goals are used to justify the need for the specified features by linking them to the strategic or operational objectives of the stakeholders. This linkage is essential in understanding the impact of the developed system on the different aspects of the system's context and vice versa [10, 11]. To this end, several goal modeling languages and frameworks are proposed in the literature including KAOS [12, 13], NFR [14, 15], AGORA [16], i*(iStar) [17], Tropos [18], and Goal-oriented Requirements Language (GRL) [19]. However, goal models developed using these languages and frameworks are subject to quality problems.

The quality assurance of requirements artifacts addresses, among others, quality problems. The quality assurance processes often depend on reviews to spot such quality problems. Reviews are manual processes that require a high degree of domain understanding and expertise to be performed effectively [20]. They also require involving all relevant stakeholders [20], who have to read, understand, and review each requirements artifact. Review processes tend to be difficult and complicated. Besides, their effectiveness depends on the reviewers' competency [21]. Reviewers may spend the majority of their valuable time detecting minor quality problems, which may hinder the effectiveness of the overall review process. Hence, it is crucial to empower requirements engineers with an automatic detection technique to help unveil these quality problems early on. This, in turn, helps the reviewer focus their efforts and resources on deeper semantic problems that require human intervention.

In this work, we address the detection of quality problems that we refer to as bad smells in goal models developed using the GRL framework, given its status as an ITU-T standard [19]. The problem of detecting the instances of bad smells in GRL models is hard due to the scalability problem associated with i* based frameworks, such as the GRL framework. In fact, scalability is the most widely known problem associated with these frameworks [22]. It affects the different quality attributes of these models, including analyzability [23], making manual analysis, such as metrics analysis for bad smells detection, impractical; hence, there is a need for automated analysis tools to support the quality assurance of GRL goal models.

To help alleviate the scalability problem and its impact on the analyzability of GRL goal models, we propose 4 GRL bad smells and develop a tool to automatically detect the instances of these smells. Specifically, we make the following contributions:

- Propose and articulate 4 GRL bad smells. These smells are concerned with metrics values.
- Propose 4 metrics to detect the instances of the proposed 4 bad smells.

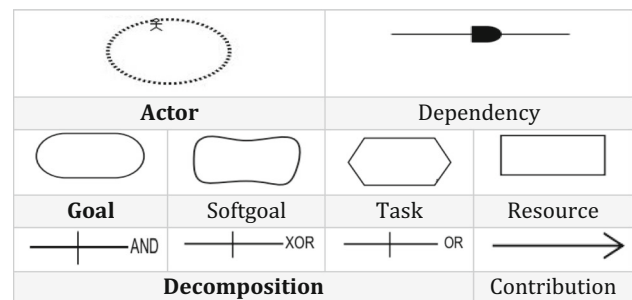


Fig. 1 Graphical representation of various features of GRL language

- Develop and evaluate GSDetector (GRL Smells Detector), a new Eclipse plugin tool to fully automate the detection of the instances of the proposed GRL bad smells. GSDetector is integrated with jUCMNav [24], the most comprehensive GRL tool available.

The rest of this paper is organized as follows: Sect. 2 presents the background of this work. The related work is presented in Sect. 3. Section 4 introduces the proposed GRL bad smells. The detection technique is presented in Sect. 5. The architecture and the internal operations of the developed tool (i.e., GSDetector) are introduced and explained in Sect. 6. The evaluation data and results are presented and discussed in Sect. 7, and the threats to the validity of this work are presented in Sect. 8. Finally, conclusions and future work are presented in Sect. 9.

2 Background

In this section, we introduce the main constructs of the Goal Requirements Language (GRL) [25] using a running example and provide a brief overview of the notion of bad smells.

2.1 Goal requirements language (GRL)

GRL is a goal-oriented requirements language, part of the ITU-T URN standard [25]. Figure 1 illustrates the main constructs of the GRL language, i.e., actors, dependencies, intentional elements (i.e., goal, softgoal, task, and resource), and links (i.e., decomposition and contribution).

In the following, we describe the main GRL constructs using a GRL model of an unmanned aircraft system (see Fig. 2) that has been adapted from [26] (with slight modifications).

Actors are active entities in the domain of the system that have objectives to be satisfied by the system. An actor is represented as a dotted circle with the stick-man next to its name, (see Fig. 1). For example, the GRL model of Fig. 2 consists

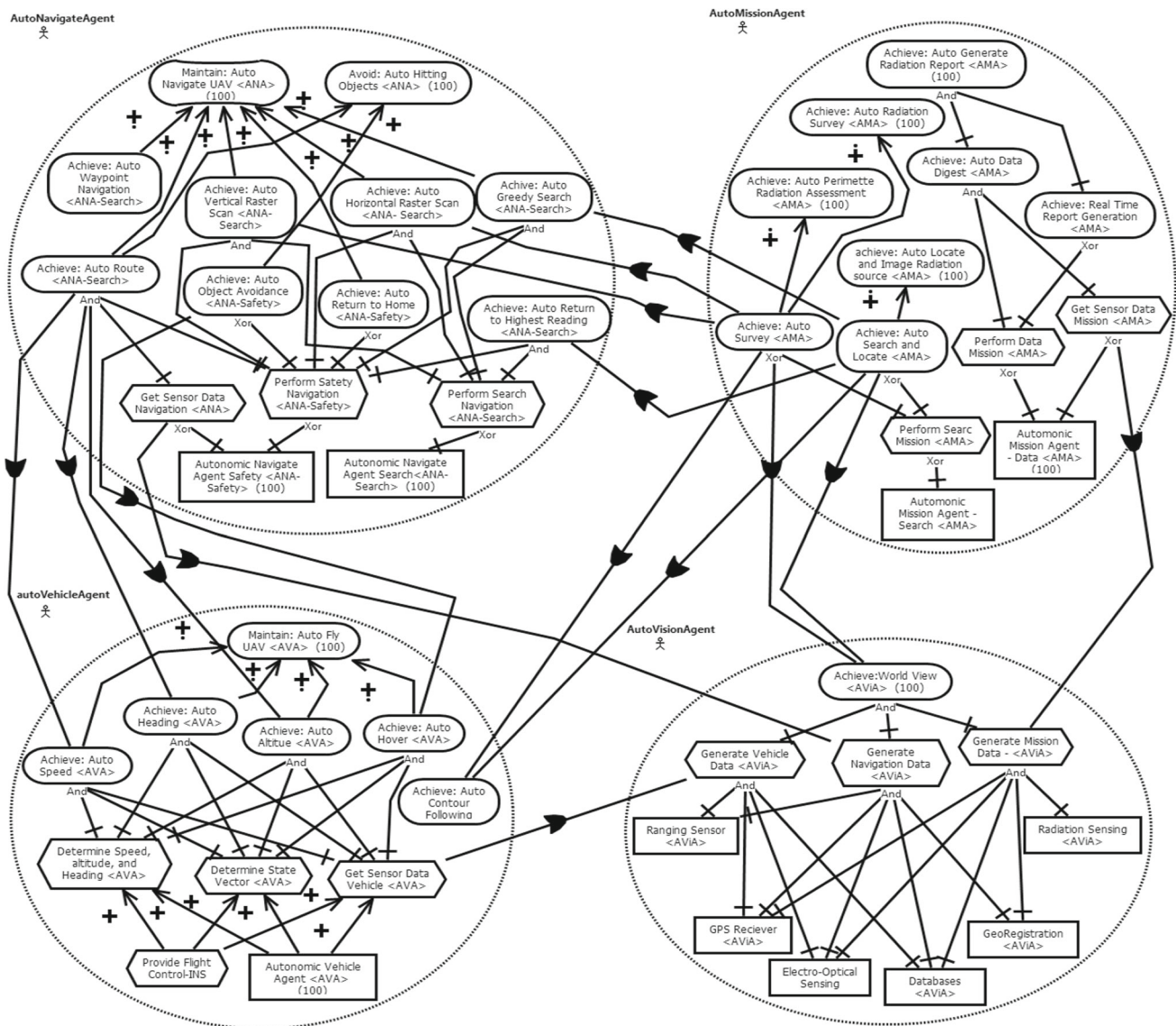


Fig. 2 A running example GRL model of an unmanned aircraft system adopted from [26]

of four actors: “AutoNavigateAgent”, “AutoMissionAgent”, “AutoVehicleAgent”, and “AutoVisionAgent”.

Actors have objectives. These objectives are modeled using goals and softgoals. The difference between goals and softgoals is that goals have clear-cut criteria (i.e., binary) to measure their satisfaction while softgoals do not have clear-cut criteria to measure their satisfaction. Therefore, the satisfaction of softgoals is evaluated as sufficiently or non-sufficiently satisfied. In Fig. 2, the “Achieve: World View <AVA>” goal in the “AutoVisionAgent” actor is an example of a goal element, and the “Maintain: Auto Navigate UAV <AVA>” softgoal in the “AutoNavigateAgent” actor is an example of a softgoal element.

After identifying the high-level goals and softgoals, these goals and softgoals can be refined further using goals or

softgoals elements until they are granular enough to be operationalized. For example, in the “AutoNavigateAgent” actor in Fig. 2, the “Maintain: Auto Navigate UAV <AVA>” softgoal is refined into six goals. The operationalization of these goals and softgoals can be represented by tasks and resources (see Fig. 1). The task element models a course of action to be performed and the resource element models physical or informational resources that are needed to be available to perform a task or to achieve a goal/softgoal. For example, in Fig. 2, operationalizing the “Achieve: World View <AVA>” goal in the “AutoVisionAgent” actor is done using three tasks. One of these tasks is “Generate Vehicle Data <AViA>”, which requires four different resources, one of them is “GPS Receiver <AViA>”.

The refinement of elements into other elements is done using various types of links. These links show how an element can be decomposed into other elements and how an element contributes to other elements. Decomposition links have three types: AND, XOR, and OR. These decomposition types are graphically represented as a solid line with a small perpendicular intersecting dash and labeled with the type of decomposition (see Fig. 1). Contribution links are represented as arrows (Fig. 1) and labeled using contribution values to express the nature and strength of the contribution. These contribution values can be positive or negative and can be represented quantitatively or qualitatively. Quantitative contributions are represented using integer numbers between -100 and 100, and qualitative contributions values are represented categorically as Make, Help, Some +, Break, Hurt, Some-, and Unknown. Figure 2 shows several examples of decomposition and contribution links. For example, operationalizing the “Achieve: World View <AVA>” goal in the “AutoVisionAgent” actor is decomposed into three tasks using the AND-decomposition links. We can also see that “Achieve: Auto Survey <AMA>” goal in the “AutoMissionAgent” actor contributes to “Achieve: Auto Radiation Survey <AMA>” goal using a Make contribution link.

Actors may depend on each other for achieving a goal/softgoal, performing a task, or providing a resource. This dependency is represented using dependency links (see Fig. 1). For example, in Fig. 2, actor “AutoVehicleAgent” depends on “AutoVisionAgent” actor to generate vehicle data, described as a task “Generate Vehicle Data <AVIA>”.

For a comprehensive introduction to GRL, readers can refer to the language standard [25]. It is worth noting that all GRL models used in this paper (i.e., as examples or for evaluation) are created using jUCMNav [24], the most comprehensive modeling and analysis tool for the URN language.

2.2 Bad smells

Bad smells are problems that affect the quality of software artifacts, such as source code and software models [27]. They are not errors in the subject artifacts; however, they might lead to errors as these smells are indicators of a poorly structured, designed, or highly complex artifact [28]. For example, a lengthy method is a code smell [27]. This smell is not an error, but it affects the reusability and readability of the source code [29]. The same definition applies to other software artifacts such as models. Bad smells in software models are symptoms of bad quality as well. For example, in a UML use case diagram, having too many use cases is considered a bad smell [30, 31] and an indicator of a quality problem. For example, an inaccurate definition of the system’s context might have led to the definition of irrelevant requirements resulting in too many use cases, many of which are irrelevant to the real requirements of the stakeholders.

3 Related work

In this section, we present and discuss related work. First, we provide a brief overview of the existing bad smells detection techniques. Second, we present the most widely used bad smells detection tools. Finally, we discuss techniques and tools for bad smells detection in goal-oriented requirements models.

3.1 Bad smells detection techniques

Many bad smells detection techniques have been proposed in the literature. These techniques include rule-based techniques, search-based techniques, machine learning techniques, metric-based techniques, etc. [32–35].

Rule-based techniques are among the most widely used techniques [36–42]. These techniques are built using rules to encapsulate the addressed bad smells and deterministically search for their instances in the subject artifact. In search-based techniques, the problem of detecting the instances of bad smells is converted into a search problem [43–48] in which the search space is stochastically searched for the instances of the addressed bad smells. In machine learning techniques, the problem of detecting bad smells is converted into a classification problem [49–61], where bad smells represent features that are learned by the classification model.

Metrics were also used in the detection of bad smells [62–74]. The basic concept behind the metric-based approach is to create rules to compare metrics values to prespecified thresholds. This concept was applied to detect the instances of bad smells in various software artifacts, including code [66], UML models [64], etc.

3.2 Bad smells detection tools

Manual bad smells detection tends to be very cumbersome and time-consuming [75]. Therefore, several techniques have been developed in the literature to automate the process of bad smells detection in the various software artifacts. Several tools were introduced to detect bad smells in the literature. These tools can be classified based on the software artifact they address, e.g., code-level bad smells, and model-level bad smells [76]. Code-level tools can be classified by the programming language they target, the programming paradigm (i.e., structural, object-oriented) considered, etc. Model-level tools can be classified based on the targeted model type, e.g., UML Class diagram, UML Sequence diagram, etc. [76].

A total of 148 code smells detection tools and prototypes were introduced in the systematic mapping study by Alkharabsheh et al. [33]. However, only a few are widely accepted based on citation counts. For example, according to [33], the most widely cited code smell detection tools

are JDeodorant [77], DÉCOR [78], PMD,¹ and InFusion [79]. These four tools are cited in 80% of the 395 reviewed papers in [33]. Infusion and DÉCOR use metrics to detect bad smells, while JDeodorant uses cluster analysis, and PMD uses rules.

For model bad smells detection tools, only a few tools are found in the literature. Ruhroth et al. developed an extension [80] to the MRC tool [81], targeting several UML diagrams (e.g., class diagrams, state charts, component diagrams) and Z specifications. The objective of this extension is to help identify refactoring opportunities through metrics measurement in the context of the formal quality cycles, which consist of three stages: measure, diagnose, and improve. The diagnosis part (i.e., detection) is developed using OCL (Object Constraints Language) rules [82].

Kim developed a new tool as an extension to the IBM Rational Rose to improve design models by introducing design patterns to the class and sequence diagrams [83]. In this extension, design patterns are defined in terms of RBML (Role-Based Meta-modeling Language) rules. These patterns have three components: the problem specification, the solution specification, and the transformation specification. The problem specification characterizes the problem model, which is used to check for opportunities to apply patterns (i.e., detection), and the solution specification characterizes the solution model, which defines the pattern to be used (i.e., correction). The transformation specification describes how the problem model can be transformed into the solution model.

Arendt et al. developed an Eclipse plugin called MSR (Metric, Smell, and Refactoring) tool [84], which consists of three tools: EMF (Eclipse Modeling Framework) Metric tool, EMF Smell tool, and EMF Refactoring tool. These tools combined perform syntactical quality checks based on metrics to detect instances of bad smells and work on UML models generated based on EMF. In addition, this tool proposes refactorings based on the detected instances of bad smells.

El-Attar and Miller developed a stand-alone tool called ARBIUM (Automated Risk-Based Inspector of UC Models) to detect bad smells in use case diagrams [30]. They used OCL rules to retrieve instances of the defined bad smells. ARBIUM is equipped with a set of predefined bad smells and can be extended by adding user-defined bad smells.

Mohamed et al. developed a tool to refactor the class and sequence diagrams called M-REFACTOR [85]. The detection component of this tool is based on metrics and heuristics (i.e., user-specified thresholds) to define bad smells. Upon detecting an instance of a bad smell, the model is tagged with that bad smell. This tagging is used to enable the modeler to evaluate the detected instance. If the modeler finds the

detected instance as a valid instance, the planned refactoring can then be initiated by the modeler.

Enckevort et al. developed a prototype tool to detect bad smells in class models [86]. This tool is built on top of the Eclipse framework and OpenArchitectureWare Eclipse plugin. It uses metrics and rules to detect instances of bad smells in the target model. These metrics and rules are formulated using OCL.

Xu developed a new prototype tool, called PB (Performance Booster), to detect and suggest curative actions for performance flaws in the design level [87]. In this prototype, a performance model is created from UML sequence and deployment diagrams that can be analyzed to detect performance flaws using rules. Once performance flaws are detected, other rules are used to suggest mitigating changes in the performance model. Then, a tree of altered performance models is generated and used to select the next performance. The process continues until all flaws are detected.

Arcelli developed a tool for performance smells detection and software system refactoring in UML models [88]. This tool is called PADRE (Performance Antipattern Detection and REfactoring) and addresses Sequence, Component, and Deployment Diagrams. PADRE is developed and implemented within Epsilon (an Eclipse-based framework) [89]. In this tool, the target UML models are transformed into a JMVA (Java Mean Value Analysis) textual format, which contains an XMI-based performance analytic model. After the performance analysis (using the JMVA), the collected indices are used to evaluate the presence of bad smells. This tool uses EVL (Epsilon Validation Language) engine in the Epsilon framework to perform bad smells detection and refactoring.

Štolc and Poláček developed a visual tool to detect and correct bad smells in class diagrams [90]. This tool is called VisTra (Visual Transformation) and provides a visual editor to create detection rules and correction scripts. To detect instances of bad smells, OCL rules are used and applied to the target model. Once an instance is identified, the respective correction script is applied.

We can see throughout this section that several tools for detecting bad smells at the model level were developed in the literature. These tools address the various types of models, including class diagrams, sequence diagrams, etc. In these tools, several bad smells detection techniques were applied, including rule-based, metric-based, pattern-based, etc. For goal models bad smells detection, only two tools were found in the literature which are discussed in the next section.

3.3 Goal-oriented bad smells detection

In the context of goal-oriented languages and frameworks, to the best of our knowledge, only two studies present techniques and tools to detect bad smells. The first work targets

¹ <http://pmd.sourceforge.net/>

AGORA goal models [91], and the second targets GRL goal models [92–94].

Asano et al. introduced four bad smells in the AGORA framework, which concern the quality of goals refinements [91]. These smells are (1) Low semantic relation, (2) Too many siblings, (3) Too few siblings, and (4) Coarse-grain leaf. The low semantic relation bad smell is concerned with how children are relevant to their parents. The too many/few siblings bad smells are concerned with whether the children are adequate and enough to satisfy their parents. The coarse-grain leaf bad smell is concerned with whether a leaf element is concrete enough to be interpreted by system-oriented operationalizations. To detect instances of low semantic relation bad smell, a hierarchical dictionary is used to calculate the similarity between parents and their children. For detecting instances of too few siblings and too many siblings bad smell, the number of children metric is used. To detect instances of coarse grain leaf, the number of refinement levels metric is used. These techniques were implemented as an extension to another tool used to analyze AGORA models [93].

With respect to GRL, only one work is found to address bad smells [92]. In this work, a rule-based technique was developed to detect bad smells, and it was implemented as an Eclipse plugin and integrated into the JUCMNav tool. This tool is not restricted to GRL bad smells. It involves many other static semantic analysis types of URN notations, including GRL. It is used in various other analyses such as legal compliance analysis, *i** profile checking, styles checking, etc. Most of the introduced bad smells (i.e., quality defects) in this tool are concerned with the completeness and consistency quality attributes of GRL models. An example of one of these smells is the absence of elements that are assigned an important value in an actor. This smell says that it is better to have at least one element with nonzero importance value in each actor.

The proposed tool in [92] is not extendable for our purpose as it does not allow flexible settings of metrics thresholds. To set thresholds using this tool, thresholds should be hardcoded into the code of the rules, which makes the analysis of metrics under different thresholds difficult. Therefore, we developed GSDetector as an Eclipse plugin, that can be extended and allows for flexible settings of metrics thresholds using a graphical user interface. In GSDetector, we use a metric-based technique to detect the instances of 4 new GRL bad smells.

4 The proposed GRL bad smells

In this section, we propose and articulate a suite of four GRL bad smells, namely, overly ambitious actor, overly operationalized actor, deep hierarchical refinement, and highly

coupled element. These smells highlight some of the possible deviations in the values of some metrics in GRL models.

Each bad smell is presented according to the following structure: (1) The smell description, (2) GRL constructs from which the proposed bad smell originates, (3) The possible interpretations of the reasons for the appearance of the instances of the proposed bad smell, (4) The potential impacts of the bad smell, (5) The definition and mathematical formulation of the bad smell, and (6) An example of how the proposed metric is calculated.

4.1 Overly ambitious actor

Description: An overly ambitious actor is an actor with too many objectives. It represents a stakeholder with high expectations.

Origin: This smell originates from the excessive number of the lowest level goals and softgoals in an actor. Goals and softgoals in an actor can be organized into three levels: High, intermediate, and low. The high-level and intermediate-level goals and softgoals are not a precise measure of the real expectations of the subject actor in terms of system-oriented requirements. This is because their satisfaction in terms of system-oriented solutions (i.e., operationalizations) is not clear yet as they are refined into other goals and softgoals. On the other hand, the lowest level goals and softgoals can quantify the real expectations of the subject actor. These goals and softgoals are refined into system-oriented requirements. Therefore, the excessive number of the low-level goals and softgoals indicates high expectations.

Possible interpretations: There might be several interpretations of the presence of this smell. First, some modelers might be inexperienced in goal modeling. These modelers might tend to exhaustively list goals and softgoals regardless of their relevance to the real requirements of the stakeholders resulting in involving too many goals and softgoals. Second, it might also be a result of failing to correctly specify the roles of the actors in the model by combining more than one role in the same actor. This failure might result in combining several roles into a single actor of a big size and a large number of low-level goals and softgoals.

Impact: From the scalability point of view, this bad smell has several impacts. The first impact of this bad smell on the infected actor and, consequently, on the containing model is the undesirable increase in size as the low-level goals and softgoals require operationalization using tasks and resources. The size of the overly ambitious actor (measured by the number of elements) tends to grow quickly. This is because the included goals and softgoals require to be refined and operationalized into tasks and resources exacerbating the scalability problem. The second impact of the appearance of this bad smell is the expected increase in the number of connections between the infected actor and the other actors in the

model. This increase in the number of connections is associated with an increase in the coupling of the infected actor, making the model more difficult to understand and modify. From the project management point of view, irrelevant requirements that might be the reason for the appearance of this smell will increase the time and cost of the project in developing irrelevant features.

Definition Let lse be a function that returns the number of the lowest level goals and softgoals in the refinement tree of actor A within GRL model G ,

$iflse(A) > LSE, (LSE > 0)$, where LSE is an integer number that represents the threshold beyond which A is deemed as an instance of the overly ambitious actor bad smell.

Example of the metric calculation: In the running example in Fig. 2, the number of the lowest level goals and softgoals in the “AutoMissionAgent” actor is 4. These goals and softgoals are: “Achieve: Auto Data Digest <AMA>”, “Achieve: Real-Time Report Generation <AMA>”, “Achieve: Auto Survey <AMA>”, and “Achieve: Auto Search and Locate <AMA>”.

This number of low-level goals and softgoals is not big enough to consider the corresponding actor as a bad smell. We provide this example to show how to calculate the metric used to detect the instances of the overly ambitious actor bad smell as the real instances of this bad smell are difficult to fit in the paper. These goals/softgoals share the same rule. They are either non-refined (i.e., they do not have children) or they are refined into tasks or resources (i.e., into operationalization elements). This example is used only to show how lse is computed. However, in practice an lse of 4 may not represent a valid bad smell.

4.2 Overly operationalized actor

Description: An overly operationalized actor is an actor with too many tasks and resources compared to the low-level goals and softgoals. It represents a stakeholder with too many system-oriented solutions compared to its objectives.

Origin: This bad smell originates from the use of an excessive number of operationalization elements (i.e., tasks and resources) in an actor. The notions of tasks and resources are brought to model operationalizations of the actors. These operationalizations represent system-oriented solutions. Therefore, the number of operationalization elements should be relative to the number of the low-level goals and softgoals as operationalizing the low-level goals and softgoals leads to operationalizing the higher levels goals and softgoals.

Possible interpretations: This smell might happen when the modeler pays too much attention to technical and implementation details. In such a case, the percentage of

operationalization elements tends to be very high compared to the low-level goals and softgoals in the subject actor.

Impact: The impact of this bad smell is threefold. First, the overly operationalized actor might be subject to frequent changes. Tasks and resources are technical details, that are usually subject to frequent changes. Second, overly operationalized actors tend to be restrictive regarding their satisfaction. If the modeler tends to specify the very specific details of the implementation, the resulting actor will be very restrictive as the modeler left few options to developers and designers of the system. Third, modeling technical details diverts the attention of the modeler from focusing on the system’s context into design or implementation details that should be left to later stages.

Definition As we described in the previous bad smell, Operationalization of the low-level goals and softgoals will lead to operationalizing the objectives of the actor. Therefore, the number of operationalization elements should be relative to the number of low-level goals and softgoals.

Let poe be a function that returns the percentage of operationalization elements (i.e., tasks and resources) to the number of low-level goals and softgoals of actor A within GRL model G ,

$ifpoe(A) > POE, (poe = 100 * \frac{\text{number of tasks and resources in } A}{\text{low_level goals and softgoals in } A})$ and $POE > 0$), where, POE is a real number that represents the threshold beyond which A is deemed to be an instance of the overly operationalized actor.

Example of the metric calculation: In the running example in Fig. 2, the number of the low-level goals and softgoals in the “AutoMissionAgent actor” is 4. These goals and softgoals are: “Achieve: Auto Data Digest <AMA>”, “Achieve: Real Time Report Generation <AMA>”, “Achieve: Auto Survey <AMA>”, and “Achieve: Auto Search and Locate <AMA>”. On the other hand, the number of the operationalization elements is 5: “Perform Data Mission <AMA>” task, “Get Sensor Data Mission <AMA>” task, “Perform Search Mission <AMA>” task, “Automonic Mission Agent—Data <AMA>” resource, and “Automonic Mission Agent—Search <AMA>” resource. Based on that, we can calculate the percentage of operationalization with respect to the low-level goals and softgoals in that actor as follows: $poe = 100 * \frac{5}{4} = 125\%$.

4.3 Deep hierarchical refinement

Description: This smell represents a very deep refinement hierarchy in an actor. It represents a case in which the high-level objectives of a stakeholder are overly refined.

Origin: This bad smell originates from the refinement hierarchy of the elements in an actor. The refinement process is intended to explain how the high-level goals/softgoals

can be refined into lower-level goals/softgoals and how, eventually, these low-level goals and softgoals can be operationalized using tasks and resources. However, this process can be misused by excessive refinement.

Possible interpretations: This smell might result from detailed modeling where the modeler tends to bring too much implementation or technical details. These details should be left to later stages of the requirements engineering process or to later stages in the software development process.

Impact: The effect of this bad smell on the impacted actor is twofold. First, this smell is associated with crowding the model with too many elements losing the benefits of high-level modeling and exaggerating the scalability problem. Second, this smell might be associated with premature restrictive design decisions if details are modeled. That is, going too deep into the details of operationalizations is, usually, associated with the design and implementation details that should be left to later stages of the software development process.

Definition Let l be a function that returns the depth (the longest path between the top-level elements and the lowest level elements + 1) of actor A within GRL model G ,

$ifl(A) > L, (L > 0)$, Where L is an integer that represents the threshold beyond which A is deemed to be an instance of the deep refinement bad smell.

Example of the metric calculation: In the running example in Fig. 2, one of the longest paths in the “autoVehicleAgent” actor is: Maintain: Auto Fly UAV <AVA> Achieve: Auto Altitude <AVA> Determine State Vector <AVA> Autonomic Vehicle Agent <AVA>. The length of this sequence (i.e., l) of elements is 3. Based on that, the depth of refinement in this actor is 4 (i.e., $3 + 1$).

4.4 Highly coupled element

Description: A highly coupled element is an element having a large number of links (incoming or outgoing links).

Origin: Links between elements.

Possible interpretations: There are several interpretations for the appearance of this smell. First, it might be due to hidden information that needs to be revealed. That is, the highly coupled element might not be granular enough; hence, it needs to be refined further, which can lead to distributing the links associated with the parent to the children element. Second, it might be due to detailed modeling, as technical details are associated with excessive interactions that should be left to later stages.

Impact: This smell might be associated with several issues. First, the highly coupled element might be more difficult to understand as it might not be granular enough, making the understanding of the information modeled by that element more difficult. Second, a highly coupled element may

be difficult to change since changes made to a highly coupled element (such as refinement) propagate to the other elements linked to it.

Definition: Let nel be a function that returns the number of links associated with element E in GRL model G ,

$ifnel(E) > NEL, (NEL > 0)$, Where NEL is an integer number that represents the threshold beyond which element E is deemed to be an instance of the highly coupled element bad smell.

Example of the metric calculation: In the running example in Fig. 2, the task “Perform Safety Navigation <ANA-Safety>” in the “AutoNavigateAgent” actor has 8 links associated with it. Based on that, the coupling of this element (i.e., nel) is 8.

5 Bad smells detection technique

A metric-based technique to detect the instances of the proposed bad smells in this work. This choice was natural given the definition of the proposed smells as shown in Sect. 4. The proposed bad smells are defined as rules of the following form:

if metric value > metric threshold,

then the measured model/actor/element is deemed as an instance of the respective bad smell

This rule guard condition has two components: metric value (calculated from the subject GRL model) and metric threshold. The threshold is chosen based on the modeler’s preference given the circumstances of the project (see Sect. 5.2). If the condition in this rule is held, the measured object (i.e., actor or intentional element) is deemed as an instance of the respective bad smell.

5.1 Metric values

In this section, the approach taken to calculate the metrics used to detect the instances of the proposed bad smells is presented. To this end, first, the GRL model is transformed into an Entity-Relation Diagram (ERD); then several algorithms are used to calculate the needed metrics by querying the ERD model.

5.1.1 GRL to ERD model transformation

To facilitate the calculation of the proposed metrics, GRL models are transformed into ERD models as the current tree-like representation of the GRL language has several problems. First, the current GRL representation is not relational enough as several of the relations are compacted as


```

< actors name = "Meeting Initiator" id = "11" contRefs = "142 252"/>
< contRefs xsi:type = "grl:ActorRef" name = "ActorRef142" id = "142" x
    = "42" y = " - 26" width = "334" height = "327" contDef
    = "11" nodes = "136 137 138 139 140 141"/>
< contRefs xsi:type = "grl:ActorRef" name = "ActorRef252" id = "252" x
    = "247" y = "35" width = "334" height = "327" contDef
    = "11" nodes = "256 258 261 262 263 264"/>

```

Fig. 3 An example of the GRL model of an actor with two references

Table 1 The tabular representation of the actors as in the GRL representation

id	Name	contRefs
11	Meeting Initiator	"142 252"

strings. For example, if an element x has three children, k , l , and m , the current representation models these relations as ‘element x has “ $k\ l\ m$ ” children’. This string representation requires further processing every time the model is queried. Second, the notion of actors adds an additional layer of network connectivity and containment increasing the complexity of the models. Third, each actor or element might have more than one reference (i.e., occurrence). Each actor can have more than a reference in every graph in the same model. Each reference can contain the same or different elements. Similarly, each element can have more than one reference (i.e., occurrence). Each reference can be linked to the same or different elements. To cope with these issues, the GRL model is transformed into an Entity Relationship Diagram (ERD). ERD is well-known and widely used to facilitate querying data.

An excerpt of a GRL model is shown in Fig. 3 to help explain how the ERD representation is used to solve the already discussed issues. This excerpt contains one actor with an $id = 11$, as shown in the “actors” tag. This actor has two references, as shown in “contRefs” tags, with the ids 142 and 252. Each of these references has a list of nodes (i.e., references of elements), as shown in the nodes attribute. The first reference has 6 nodes: “136 137 138 139 140 141” and the second reference has 6 nodes as well: “256 258 261 262 263 264”.

Tables 1 and 2 show in a more abstract form the representation of the actor shown in Fig. 3 and its references as in the GRL representation. In Table 1, the references of the actor are represented as a string. In Table 2, the nodes (i.e., elements references) in each of these references are represented as strings as well. We can see that the relations between these two tables are compacted and indistinguishable. To establish clear and queryable representation, the GRL compact and

Table 2 The tabular representation of the references of actors as in the GRL representation

id	nodes
142	“136 137 138 139 140 141”
252	“256 258 261 262 263 264”

Table 3 The ERD representation of actors

id	Name
11	Meeting Initiator

Table 4 The ERD representation of the references of actors

actor	contRefs
11	142
11	252

Table 5 The ERD representation of the references of elements

contRefs	nodes
142	136
142	137
142	138
142	139
142	140
142	141
252	256
252	258
252	261
252	262
252	263
252	264

tree-like representation is transformed into an ERD representation.

ERD models are designed specifically to facilitate queries by creating relational entities. The application of this transformation on the model shown in Fig. 3 is as follows: First, a separated table is created to store actors by their ids, as shown in Table 3. Second, a table is created to store each reference of the actor separately, as shown Table 4. Third, a table is created to store each node of each reference separately, as shown in Table 5. In this representation, the created tables are relational and can be directly queried without further processing each time the model is queried.

It is important to mention that these tables show only part of the ERD model to keep the size of the paper manageable. The objective of presenting this part is to show why the transformation is needed and how the transformation was performed. In addition to these tables, other tables are created for elements and links. These tables are also augmented with additional attributes such as the type of the elements, source elements, and destination elements. Furthermore, in terms of programming constructs, 1-D arrays, 2-D arrays, and a list of lists are used to implement these tables. Lastly, the developed model transformation is linear in the number of elements. It requires a linear time to map the source GRL

In the first step, all the elements belonging to the same actor are collected by querying Tables 3, 4, and 5. Once the list of these elements is ready, each element is examined to collect the low-level goals and softgoals using two filters. First, out of this list of elements, keep only goals and softgoals. Second, out of these goals and softgoals, keep only goals and softgoals that either have no children or have only tasks and resources as children. The application of this algorithm on the actor “autoNavigateAgent” in the running example, Fig. 2, results in 8 low-level goals and softgoals.

Algorithm 1 The query associated with the overly ambitious actor bad smell

Input: ERD Model

Input: The subject actor A

Process: Collect all of the contRefs of actor A by querying the ERD model:

NoOfLowLevelGoalsANDSoftgoals=0;

For each ContRefs, collect all of its nodes:

If (ContRefs.nodes → size() != 0) then

Retrieve intentional elements of the collected nodes

For each intentional element IE:

If (IE.type=goal OR IE.type=softgoal) and

(If (IE.linksDest → size() = 0) OR

If (IE → children → types() are (tasks V resources)

then NoOfLowLevelGoalsANDSoftgoals++;

Return NoOfLowLevelGoalsANDSoftgoals;

Output: Number of low-level goals and softgoals in actor A

model to the target ERD model. Hence, this transformation is not detrimentally time demanding.

5.1.2 Detection queries

Detection queries are used to query the model to calculate the metrics needed to detect the instances of the proposed bad smells. These queries work on the ERD model and range between simple rule-based queries and deterministic graph search-based queries, specifically, breadth-first search. The query corresponding to each bad smell is presented as follows:

Overly ambitious actor bad smell detection query In Algorithm 1, a rule-based query is developed to calculate the number of low-level goals and softgoals in the subject actor by querying the ERD model. Algorithm 1 takes the ERD model and the subject actor as an input and returns the number of low-level goals and softgoals in that actor as an output. The idea behind this algorithm can be summarized in two steps.

Overly operationalized actor bad smell detection query In Algorithm 2, a rule-based query is developed to calculate the percentage of operationalization metric in the subject actor. Algorithm 2 takes the ERD model and the subject actor as an input and returns the percentage of operationalization in that actor as an output. As we described earlier, the percentage of operationalization is calculated by dividing the number of operationalization elements by the number of low-level goals and softgoals. Therefore, Algorithm 2 uses Algorithm 1 in its denominator to calculate the low-level goals and softgoals. The application of this algorithm on the actor “au-

toNavigateAgent” in the running example, Fig. 2, results in $(5/8)*100 = 62.5$.

ments belonging to the subject actor are collected. Since the subject actor can have several appearances in the different

Algorithm 2 The query associated with the overly operationalized actor bad smell

Input: ERD Model
Input: The subject actor A
Process: Collect all of the contRefs of actor A by querying the ERD model:
 NoOfOperationalizationElements = 0;
 For each ContRefs, collect all of its nodes:
 If (ContRefs.nodes \rightarrow size() \neq 0) then
 Retrieve intentional elements of the collected nodes
 For each intentional element IE:
 If (IE.type=task OR IE.type=resource),
 then NoOfOperationalizationElements++;
 Return NoOfOperationalizationElements/algorithm1*100;
Output: Percentage of operationalization in actor A

Deep Hierarchical Refinement bad smell detection query

In Algorithm 3, a more complicated query is developed. This query uses the breadth-first search algorithm to traverse the GRL model based on the ERD model to find the maximum depth of the subject actor. Algorithm 3 takes the ERD model and the subject actor as an input and returns the maximum depth (i.e., the maximum number of refinement levels) of that actor as an output. The idea behind this algorithm can be summarized in two steps. In the first step, high-level ele-

graphs in the same model, the high-level elements should be collected in all these appearances. Tables 3, 4, and 5 are used to retrieve all these elements. In the second step, each high-level element is treated as a root of a branch and traversed to find its depth. Once all branches are traversed and their depths are calculated, the maximum depth among these depths is returned as the depth of the subject actor. The application of this algorithm on actor “autoVisionAgent” in the running example, Fig. 2, results in 3 as the depth of this actor.

Algorithm 3 The query associated with the deep refinement bad smell

Input: ERD Model
Input: The subject actor A
Process: Collect all of the contRefs of actor A by querying the ERD model:
 branches [] ; // High-Level Goals and Softgoals
 For each ContRefs, collect all of its nodes:
 If (ContRefs.nodes \rightarrow size() \neq 0) then
 Retrieve intentional elements of the collected nodes
 For each intentional element IE:
 (If (IE.linksSrc \rightarrow size() = 0) OR
 If(IE.linksSrc \rightarrow type() = dependency))
 then branches [] \rightarrow add (IE)
 depthOfBranches [];
 For each branch in branches B:
 depthOfBranches [] \rightarrow breadthFirstSearch(B)
 return depthOfBranches [] \rightarrow max()
Output: Maximum depth in actor A

Table 6 Detection thresholds

Bad smell	Threshold
Overly ambitious actor	15
Overly operationalized actor	250%
Deep hierarchy of an actor	5
Highly coupled element	8

Highly coupled element bad smell detection query In Algorithm 4, a rule-based query is developed to calculate the coupling of the subject element. Algorithm 4 takes the ERD model and the subject element as an input and returns the coupling of that element as an output. To calculate the coupling of the subject element, the number of the source and destination elements associated with that element is calculated. The application of this algorithm on element “Perform safety navigation <Safety-ANA >” in actor “autoVisionAgent” in the running example, Fig. 2, results in 8.

Algorithm 4 The query associated with the highly coupled element bad smell

Input: ERD Model
Input: The subject intentional element IE
Process: By querying the ERD model:
 couplingOfElement=0;
 noOfSourceElements= IE.linksSrc→ size();
 noOfDestinationElements= IE.linksDest→ size();
 couplingOfElement=noOfSourceElements+noOfDestinationElements;
 return coupling of element;
Output: coupling of element IE

5.2 Metrics thresholds

There are three approaches for setting metric thresholds [95]: statistical-based, metric-based, and experience-based. The Statistical-based approach requires a large number of models to be able to establish a representative sample to conduct the statistical analysis. The Metric-based approach requires relating the metric value to other quality attributes, such as defects that are not available in our context. Therefore, setting thresholds using the statistical-based or metric-based approaches is impractical. The experience-based approach derives thresholds from the modeler’s experience. Hence, in our context, we use the experience-based approach to set the thresholds.

To evaluate the developed tool, we used the threshold values suggested by the experts shown in Table 6. To set effective thresholds, modelers need to consider the specificity of each project. In order to help modelers set appropriate thresholds,

modelers may consider the model size and bad smell tolerance level.

5.2.1 Model size

The model size influences the GRL model size. If the project is big, the GRL model is expected to be big as well, i.e., a large number of actors, intentional elements, and links. Hence, the bigger the project, the larger the expected threshold values.

5.2.2 Bad smell tolerance level

Depending on the bad smell’s importance to the modeler, he can set the threshold value to represent his tolerance of the bad smell. Setting a small value to the threshold will give low tolerance to detect the bad smell, while a large value will give high tolerance in detecting the bad smell.

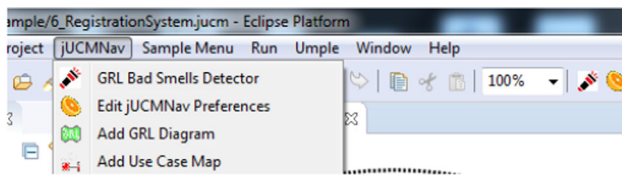
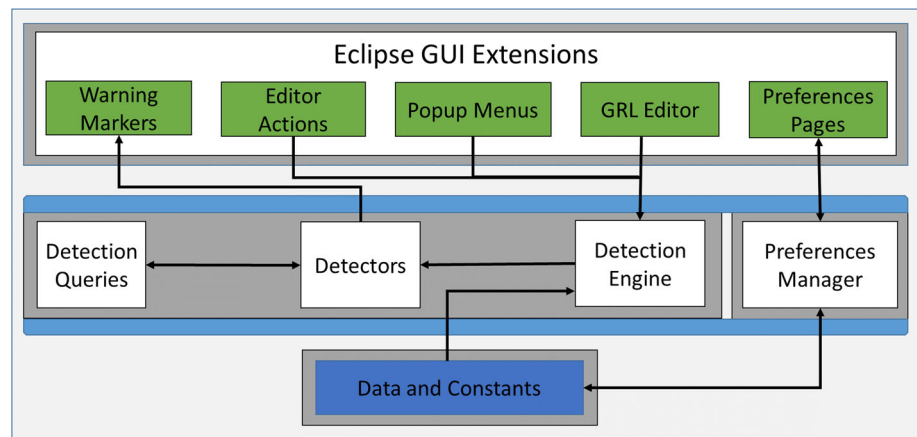
6 GSDetector tool

In this section, we present the GSDetector architecture and its internal operations. GSDetector is developed as an Eclipse plugin. Its source code and installation instructions are publicly available in GitHub.²

6.1 GSDetector architecture

The architecture of GSDetector follows a layered architecture to allow for the separation of concerns. The layered architecture provides flexibility and extensibility in designing software systems by dividing the overall functionalities of the system into smaller components, which eases their design and implementation. The design of cohesive layers allows the extension of the implementation to be made with minimum modification to the other components. Figure 4 illustrates

² <https://github.com/MawalMohammed/GRL-Bad-Smell-Detection>

Fig. 4 Architecture of GSDetector plugin**Fig. 5** Editor actions added to the Eclipse framework

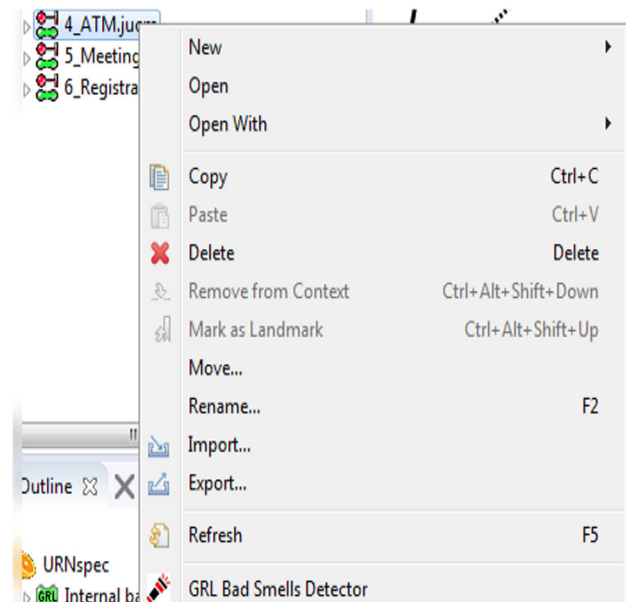
GSDetector architecture which is composed of three layers: the presentation layer (i.e., upper layer), the business logic layer (i.e., middle layer), and the supporting layer (i.e., bottom layer).


6.1.1 Presentation layer


The presentation layer is responsible for providing the user interface elements needed by the user to invoke the capabilities of the tool. Since GSDetector is an Eclipse plugin, the presentation layer is implemented as an extension of the Eclipse framework. Therefore, the basic functions related to the user interface are provided by the Eclipse framework.

GSDetector works on top of the jUCMNav plugin and on top of the Eclipse framework. To effectively integrate the functionalities of GSDetector with the functionalities of the jUCMNav and the functionalities of the Eclipse framework, the GSDetector plugin extended specific Eclipse extensions with the needed user interface elements for presenting and managing its functionality. These extensions, shown in Fig. 5, are added as extensions to the Eclipse interface and presented in the following subsections.

Editor actions Editor actions extension is used to add actions and menu items to the associated editor (i.e., GRL editor in our case). These actions and menu items become active when the GRL editor is active and applied to the file open in the GRL editor. They appear in the jUCMNav menu and toolbar

**Fig. 6** The Popup menu action added to the Eclipse framework

as a torch icon  as shown in Fig. 5. They are used, when clicked, to invoke the detection process.

Popup menus Popup menu actions are the actions added to the popup menus associated with specific types of files. In GSDetector, these are the actions associated with the files of type “.jucm” (i.e., files created using jUCMNav tool which contain GRL models). The result of adding this action to the popup menus is shown in Fig. 6. This action can be accessed by right-clicking on the files of type “.jucm” and locating the menu item associated with the torch icon  and the label “GRL Bad Smell Detector”. Similar to the editor actions, this item is also used to invoke the detection process.

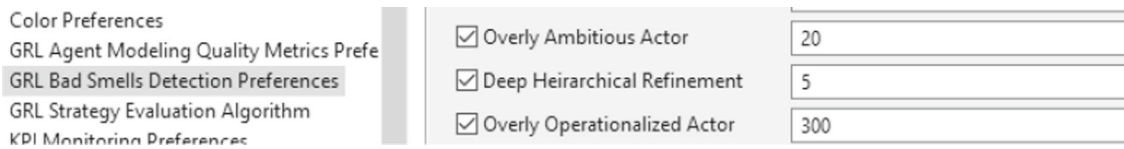


Fig. 7 GSDetector preferences page

Preference pages Preference pages are used to enable users to set and update their preferences. In GSDetector, a preference page is added to the preferences pages of the Eclipse framework and is associated with the preference pages of the jUCMNav plugin. This page is used to allow users to select which bad smells to be detected and to set and update thresholds, as shown in Fig. 7.

Markers In the Eclipse framework, markers represent a mechanism to annotate objects with information as per the developer's needs. In GSDetector, markers are used as an output mechanism. They are used to display the detected instances of the proposed bad smells as warnings in the "problems view" of the Eclipse framework (see Fig. 8). The detected instances of bad smells can be highlighted by clicking on the respective marker.

6.1.2 Business logic layer

The business logic layer is responsible, among others, for processing commands initiated by the UI, coordinating the workflow to achieve the objectives of the developed tool, accessing data in the lower layer, and displaying data in the user interface. In GSDetector, these functionalities are achieved through the cooperation of the detection engine, bad smell detectors, detection queries, and preferences managers, as shown in Fig. 4. The detection engine coordinates requests from the user based on the preferences and sends them to the detectors. The detection engine transforms the GRL model into the ERD model (see Sect. 5.1.1). Detectors use the queries to query the model and detect the instances of the selected bad smells. In addition, detectors report the detected instances of bad smells to the user as warnings in the problems view of the Eclipse framework. The preferences managers are responsible for setting, getting, and storing preferences.

Detection engine The detection engine coordinates the workflow of the detection process in the developed tool. It starts by locating the location of the active file. Once the location is found, the detection engine reads the file containing the GRL model as an XML document object. This XML document object is transformed into an ERD (as shown in Sect. 5.1.1) to extract the required information to detect the instances of bad smells. Once the required information

is extracted and prepared, the detection engine consults the preferences store on the selected bad smells to be analyzed for bad smell detection. Then, the detectors of the selected bad smells are called, and the needed data are passed.

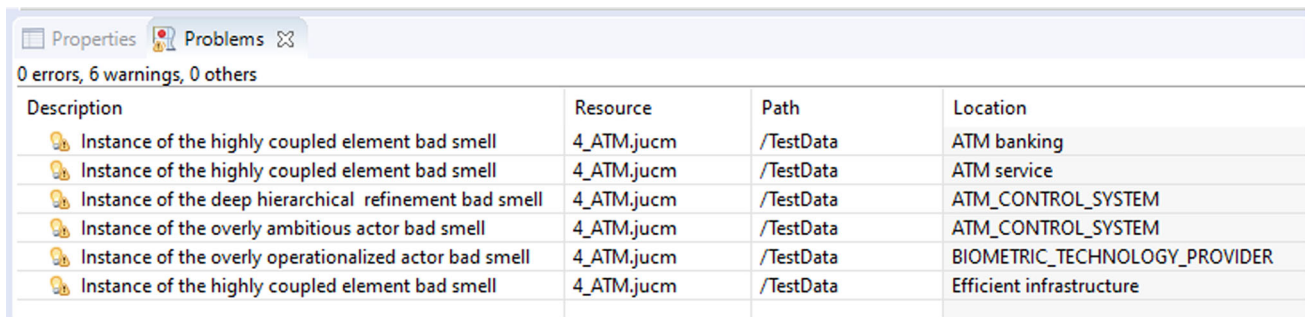
Detectors Each bad smell has its own detector. The detector is employed to retrieve the instances of the respective bad smell and report them to the problems view of the Eclipse framework. To this end, the detectors call the detection queries on each object (i.e., actor, element, etc.). The detection queries return true or false, indicating whether the object is an instance of the respective bad smell. If an instance of a bad smell is detected, the detector reports it as a warning in the problems view of the Eclipse framework.

Detection queries A detection query is developed for each bad smell. This query is used to calculate the metric used to detect the instances of the respective bad smell. These queries receive an object (i.e., actor, element, etc.) each time and check whether that object represents an instance of the respective bad smell. The details of these queries are previously introduced in Sect. 5.1.2.

Preference manager The preferences are added to the developed tool to allow for higher control over the detection process and the specification of bad smells. The preference manager manages preferences setting, getting, and storing. It is responsible for deploying the required controls (i.e., checkboxes, textboxes, and labels) in the GSDetector preferences page. In addition, the preference manager is responsible for storing and updating the preferences in the GSDetector data and constants class.

6.1.3 Data layer

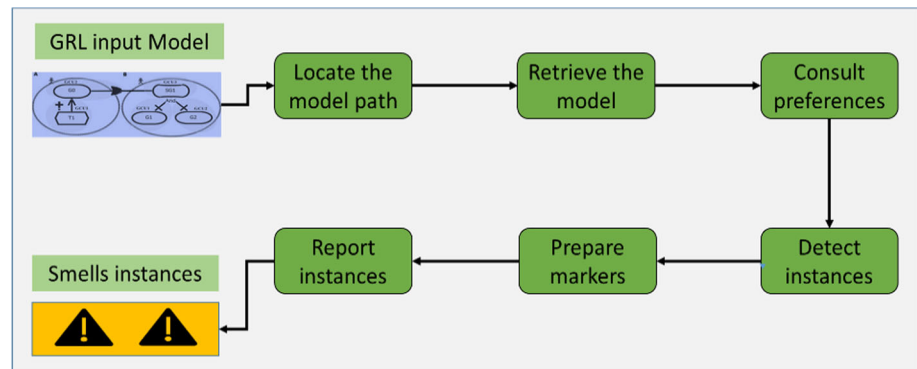
The data layer is brought to support the functions of the business logic layer. The data used to store and manage preferences reside in this layer. These data are stored as a data class and managed by the preferences managers. This class stores the preferences data, which include the information of bad smells selections and the specified thresholds. The detection engine uses the data stored in this class to coordinate the workflow of the detection process.



Description	Resource	Path	Location
Instance of the highly coupled element bad smell	4_ATM.jucm	/TestData	ATM banking
Instance of the highly coupled element bad smell	4_ATM.jucm	/TestData	ATM service
Instance of the deep hierarchical refinement bad smell	4_ATM.jucm	/TestData	ATM_CONTROL_SYSTEM
Instance of the overly ambitious actor bad smell	4_ATM.jucm	/TestData	ATM_CONTROL_SYSTEM
Instance of the overly operationalized actor bad smell	4_ATM.jucm	/TestData	BIOMETRIC_TECHNOLOGY_PROVIDER
Instance of the highly coupled element bad smell	4_ATM.jucm	/TestData	Efficient infrastructure

Fig. 8 The reported instances of bad smells using Eclipse markers

Fig. 9 The detection process



6.2 GSDetector internal operation

The operation of GSDetector goes through several stages, as shown in Fig. 9. It starts with the GRL model opened in the jUCMNav editor. The detection engine retrieves the location of the GRL model (as a file path). Then, it reads the content of the file as an XML document object. Before starting the bad smells detection process, the detection engine checks the stored preferences in the GSDetector data and constants class for the selected bad smells to be detected and the specified thresholds. For each of the selected bad smells, the detector and detection query cooperate to detect the instances of the respective bad smell. If an instance of the respective bad smell is detected, a marker is prepared to report it to the user. Once ready, the prepared marker is reported into the problems view of the Eclipse framework.

6.3 Tool extensibility

The tool can be extended by modifying the GUI to accommodate the new bad smell properties and their corresponding data placeholders. To extend the tool by adding a new bad smell, you need to add the functions to detect the bad smell instances. Then, add the function call to the bad smell detection engine. The technical details can be found in the developers' guide.³

³ <https://github.com/MawalMohammed/GRL-Bad-Smell-Detection/blob/master/Developers'%20Guide.md>

7 Evaluation

In this section, we start by describing the tool testing strategy. Next, we present the models used to evaluate the performance of the proposed technique and tool, followed by a discussion of the evaluation results.

7.1 Tool testing

GSDetector was thoroughly tested using a combination of white and black box testing. Black box test cases were designed using Input Space Partitioning (ISP) to cover various GRL models' characteristics, such as (1) single vs. multiple graphs, (2) actor bound vs. unbound intentional elements, (3) referenced vs. unreferenced actors, (4) referenced vs. unreferenced intentional elements, (5) inter-actor dependencies vs. decomposition/contributions, (6) presence vs. absence of dependum in dependency relationships, (7) models with vs. without bad smells, etc. All development bugs were fixed, and the failed test cases were retested.

7.2 Evaluation data

The proposed tool and technique were evaluated against 5 models of different sizes (i.e., number of intentional elements and links) that belong to different domains (health sector, food industry, sports, financial services, and organizational sector), as shown in Table 7. Model 2 and Model 5 are adopted

Table 7 Descriptive statistics

Model #	System modeled	No. of actors	No. of elements	No. of links	No. of goals	No. of Softgoal	No. of Tasks	No. of resources
Model 1	Hospital ^a	2	84	89	32	0	52	0
Model 2	Navigation [26]	4	49	90	25	0	13	11
Model 3	Fitness App	11	54	52	14	15	24	1
Model 4	ATM	4	159	181	30	33	92	4
Model 5	Meeting Scheduler [96]	3	29	39	5	7	15	2

^a<https://github.com/mngola/ECSE539A1/blob/3cfc7934de8cc3839ece22d34ccd58e7fc90d047/healthTracker.jucm>

from research papers [26, 96], while models 1, 3, and 4 were developed by students. The evaluation dataset is available online.⁴

These models take several modeling practices and scenarios into consideration. Model 1 consists of several graphs and has many intentional elements referenced across different graphs in the same model. In addition, Model 1 covers the case where some of the elements are unbound to GRL actors. Model 2 covers the case of inter-actor dependencies using dependency links without dependums. Model 3 represents a GRL model that is free of the instances of the proposed bad smells. In addition, model 3 uses compositions/contributions to describe inter-actor dependencies. Model 4 consists of several graphs and has actors referenced in different graphs within the model. Finally, Model 5 is used to evaluate the tool when the modeler adopts a less common refinement practice. Indeed, in general, the refinement tree starts by refining the goals and softgoals (as tree roots) and ends with tasks and resources (as leaf elements). Model 5 uses tasks as root elements, which are refined into goals and other tasks.

Furthermore, the choice of the selected models takes into consideration the difference in the distribution of strategic elements (goals and softgoals) vs. the operationalization elements (tasks and resources). The distribution of the strategic elements compared to the operationalization elements in each evaluation model is shown in Fig. 10.

7.3 Results and discussion

Table 8 summarizes the results of evaluating GSDetector and the proposed technique on the case studies described in Sect. 7.2, using the thresholds defined in Table 6. Model 3 was found free of any instance of the proposed bad smells

⁴ <https://github.com/MawalMohammed/GRL-Bad-Smell-Detection/tree/master/DataSet>

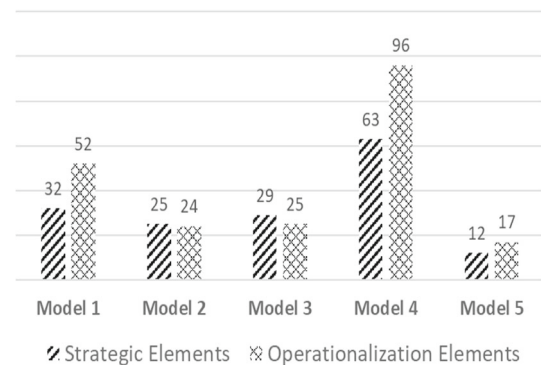


Fig. 10 The distributions of the strategic elements vs. the operationalization elements

as expected. This model is used to ensure that the developed tool does not retrieve false instances. The other models were infected with one or more instances. These instances were successfully detected by the tool and manually compared with the actual instances. Table 8 also shows the values of the metrics associated with the detected instances. For example, Model 5 is infected with a single instance of the overly operationalized actor bad smell, the value of the metric associated with this instance (i.e., the percentage of operationalizations as described in the definition of bad smells) is 400.0. Since the developed tool successfully detects all the instances of bad smells as designed, the precision and recall of the developed tool are 100% and 100%, respectively, as the developed technique is deterministic.

The developed tool was able to detect 4 instances in Model 1 and 3 instances in Model 4 of the highly coupled element bad smell, as shown in Table 8. Four elements in Model 1 and three elements in Model 4 are found to be linked with more than 8 links exceeding the prespecified threshold. The modeler or the model analyst can use this information to refactor

Table 8 Evaluation results

Model	No. of instances	Bad smell	No. of existing instances	No. of detected instances	Metric values
Model 1	6	Overly Operationalized Actor	2	2	333.33, 300.0
		Highly Coupled Element	4	4	13, 13, 11, 9
Model 2	1	Overly Operationalized Actor	1	1	900.0
Model 3	0	No Smells	0	0	-
Model 4	7	Overly Ambitious Actor	1	1	24
		Overly Operationalized Actor	2	2	295.83, 400.0
		Deep Hierarchy	1	1	6
		Highly Coupled Element	3	3	11, 9, 11
Model 5	1	Overly Operationalized Actor	1	1	400.0

these elements to improve the comprehensibility and complexity of the infected model. The modeler might refine these elements and move some or all of the links associated with the newly created elements. The modeler might also decompose the highly coupled element into two or more elements and distribute the links among the newly created elements.

The largest number of instances of bad smells are associated with Model 4 (i.e., 7 instances). It is infected with one or more instance(s) of all the proposed bad smells, as shown in Table 8. This can be attributed to the size of this model. This model is the largest model among the evaluation models. One of the instances associated with Model 4 is an instance of the deep refinement hierarchy bad smell. The infected actor is found to be overly refined exceeding the prespecified threshold (i.e., 5, as shown in Table 6). The modeler or the model analyst can use this information to refactor the infected actor. The modeler or the model analyst needs to revisit the model to assess the level of the technical details in the infected actor. If the infected actor is found to be too detailed, the modeler needs to remove details that are not needed at this level stage of the requirements engineering process. The modeler might also consider performing some merges when the removal is not an option.

The developed tool was able to detect one instance of the overly ambitious actor bad smell in Model 4, as shown in Table 8. The number of the lowest level goals or softgoals in the infected actor was found to exceed the specified threshold (i.e., 15 as specified in Table 6). The modeler or the model analyst can use this information to refactor the infected actor.

The modeler needs to ensure that these goals or softgoals are relevant to the actor. If a goal/softgoal is found to be irrelevant, it should be removed from the infected actor. If the reason for the appearance of this instance is due to modeling the low-level details, the modeler needs to revisit the infected actor and remove the detailed requirements (merges can be applied in some cases as well). If these goals or softgoals are relevant and the level of details is adequate, then, the infected actor should be revisited to ensure that the granularity of the actor is adequate (i.e., each actor represents a single stakeholder). If the infected actor is found to represent more than a single stakeholder, this actor should be decomposed to reflect the actual number of stakeholders.

The developed tool was also able to detect the instances of the overly operationalized actor bad smell in Model 1, Model 2, Model 4, and Model 5, as shown in Table 8. These instances represent actors in which the percentages of operationalization elements to the lowest-level goals or softgoals exceed the prespecified threshold (i.e., 250%, as shown in Table 6). Detecting an instance of the overly operationalized actor bad smell can be used by the modeler or model analyst to refactor the infected actor. This might result from focusing on modeling detailed requirements that should be left to later stages of the requirements engineering process. If this is the case, the modeler needs to revisit the model to remove the details or to perform merges.

By inspecting the detected instances of bad smells and the associated models, we found that the main reason for the appearance of these instances can be attributed to the focus

of the modelers on the system to be developed more than the strategic needs of the stakeholders. The indicators of this focus can be found in the poor refinement of the strategic needs of the stakeholders or in the increased attention on the system and its functionality. The effective refinement of the strategic needs of the stakeholders usually requires refining the goals of the stakeholders into subgoals, and keeping refining these subgoals into sub-subgoals until the needs of the stakeholder are clarified enough for operationalization using the system and its functionality. Besides, the operationalization of the needs of the stakeholders using the system and its functionality should be kept at a high level as goal modeling is an early-stage requirement modeling framework. This stage is usually followed by more detailed and low-level modeling stages, including scenario modeling and software specification.

As an early-stage requirements modeling framework, GRL is intended to align the strategic needs of the stakeholders with the functionality of the system to be developed. In this framework, the strategic elements (i.e., goals and softgoals) are the elements that model the strategic needs of the stakeholders, and the operationalization elements (i.e., tasks and resources) are the elements that translate the strategic needs into system-oriented solutions. However, analyzing the distribution of the strategic elements vs. the operationalization elements for the evaluation models reveals that the focus of the modelers was given to the system and its functionality. We can see in Fig. 10 that the number of operationalization elements is higher compared to the number of strategic elements in three of the smelly models, including Model 1, Model 4, and Model 5. Although this is not the case with Model 2 (i.e., smelly, but the number of operationalization elements is less than the number of strategic elements), by inspecting Model 2, Fig. 2, we found that the distribution of operationalization elements was higher in the overly operationalized actor (i.e., “AutoVisionAgent”) compared to the other actors in the same model. On the other hand, in Model 3, the non-smelly model, we can see that the number of operationalization elements is less than the number of strategic elements.

In addition, we can see that all the smelly models (i.e., Model 1, Model 2, Model 4, and Model 5, as shown in Table 8) are infected with instances of the overly operationalized bad smell. The overly operationalized bad smell, by definition, indicates that the focus was higher on the system and its functionality compared to the strategic needs. To strengthen our analysis, we manually inspected the evaluation models and the detected instances of bad smells. However, since Model 2 and Model 5 are infected only with instances of the overly operationalized bad smell that were already discussed, our focus will be given the Model 1 and Model 4 especially since these two models are associated with the majority of the detected instances of bad smells (13 out of 15

instances). Model 1 and Model 4 are developed by students, as mentioned earlier, which can be used to justify why most of the smells were associated with these models. However, this justification is not sufficient to identify the problems and how they can be solved.

If we take a quick look at Model 1, we can see that the needs of the stakeholders are not adequately refined. Most of the goals and softgoals in this model are refined into one or two levels only. For example, the “Ensure confidentiality” goal⁵ is directly refined into tasks. This is not adequate enough to justify why this goal is needed or to clarify the meaning of that goal to the stakeholder. Therefore, this actor is found infected by 4 instances of the highly coupled element bad smell and 2 instances of the overly operationalized actor bad smell. Refining goals and softgoals adequately could have led to reducing the number of links associated with the highly coupled elements and improving the strategic clarity and alignment of the overly operationalized actors. Similarly, when we take a look at Model 4, we can see that the modeler was focusing on the system. Instead of focusing on the stakeholders, the modeler modeled the system to be developed as an actor (i.e., “ATM_CONTROL_SYSTEM”).⁶ In this actor, which contains more than 90%, of the elements in this model, the modeler provides a description of the system and its functionality. As a result, this model grows large, leading to the appearance of the associated instances of smells.

8 Threats to validity

GSDetector and its related techniques are subject to several threats to validity that we categorize according to four important types as identified by Wohlin et al. [97].

Construct threats: The developed tool and its related techniques are evaluated on models developed in academic settings; Some of these models are taken from papers, and the others are developed by students. Their sizes might not be as real-world models. However, we think this does not affect the effectiveness of the developed tool and techniques as the models used in the validation cover several configurations. The developed tool is an open-source tool, and it is available online (see Sect. 6) for whoever wants to evaluate its effectiveness on a different dataset.

Internal threats: The internal threats to validity concern the degree to which the outcomes of the study depend on the experimental variables. The first internal threat to the validity of this work stems from the specification of the addressed bad smells. This specification depends on metrics thresholds. In

⁵ https://github.com/MawalMohammed/GRL-Bad-Smell-Detection/blob/master/DataSet/1_Hospital.jucm

⁶ https://github.com/MawalMohammed/GRL-Bad-Smell-Detection/blob/master/DataSet/4_ATM.jucm

this work, we used thresholds derived based on our experience. Others might also choose different values. However, we believe that the adopted thresholds provide guidance on setting thresholds. Besides, we provided the developed tool with a feature to allow setting different thresholds when needed. Another internal threat to the validity of this work stems from the implementation of the developed tool. There might be some latent faults in the implementation of the developed tool. We conducted manual black-box testing to ensure that the tool can be adequately used in practical settings. However, we cannot guarantee the absence of code bugs.

External threats: External threats to validity concern the ability to generalize the outcomes of the study. The proposed bad smells and detection techniques are built to address GRL goal models. To generalize the outcomes of this work to models developed using other goal modeling frameworks, several adaptations to the outcomes of this work are needed. However, we think that only a few changes are required, especially with goal modeling frameworks such as *i**.

Conclusion threats: Conclusion threats to validity are concerned with the threats associated with the conclusions made in this work. The objective of this work was to develop a tool to identify GRL quality improvement opportunities based on bad smells. To this end, we proposed metric-based bad smells. We used metrics thresholds derived based on our experience. The retrieved quality improvement opportunities based on these thresholds might be seen differently by different modelers. Some might find them as bad smells, and some might not as they might not agree with the threshold. To help alleviate this problem, the developed tool provides a user interface to modify and set different thresholds as per user preferences.

9 Conclusion

GRL Goal models are used in the early stage of the requirements engineering process. The quality of these models influences the other stages in the process of requirements engineering. Therefore, the higher the quality of goal models, the higher the quality of the subsequent requirements artifacts. To help improve the quality of these models, the notion of bad smells is adopted in this work. We proposed a list of metrics-based bad smells. These smells represent symptoms of bad quality that can affect GRL goal models. The automatic detection of the instances of the proposed bad smells can provide quick feedback that can help model developers and reviewers improve the quality of the addressed model. It will also help them focus on quality problems that require deeper analysis and human intervention.

To enable automatic and flexible detection of the instances of the proposed bad smells, we introduce GSDetector. In this tool, we developed metrics-based detection rules to detect the

instances of the proposed bad smells. These rules consist of two components: metrics and thresholds. We developed four metrics and four algorithms to calculate their values based on metric thresholds. In addition, the developed tool is augmented with a preferences manager. It allows selecting and deselecting bad smells to be detected. It also allows setting and changing thresholds to specify bad smells. To help modelers set effective thresholds, the factors that affect thresholds setting is presented and explained.

The developed tool is evaluated on several case studies based on the adopted thresholds. As a result, the developed tool was able to detect all the instances of the proposed bad smells. The investigation of these instances revealed that these instances can mostly be attributed to the increased modelers' focus on the system and its functionality compared to the focus on modeling the strategic needs of the stakeholders. These instances are considered quality improvement opportunities. Modelers can use these instances to improve the quality of goal models and avoid quality defects early in the requirements engineering process.

Future work can have several directions. First, additional bad smells can be proposed and added to the developed tool. The developed tool is available online for further analysis and extension. To help others adapt and extend this tool, its architecture and operation are introduced and explained in this work. Second, developing (semi-)automatic refactoring techniques to rectify the instances of the proposed bad smells is another direction of future work. Third, the outcomes of this work can be adapted to the other goal modeling framework such as *i** and NFR.

Acknowledgements The authors acknowledge the support of King Fahd University of Petroleum and Minerals in the development of this work.

Author contributions Mawal A. Mohammed: Conceptualization, methodology, analysis, and writing of the manuscript. Jameleddine Hassine: Conceptualization, methodology, analysis, and writing of the manuscript. Mohammad Alshayeb: Conceptualization, methodology, analysis, and writing of the manuscript.

Funding The author declare that there is no funding.

Availability of data and material <https://github.com/MawalMohammed/GRL-Bad-Smell-Detection>

Code availability Not applicable.

Declarations

Conflict of interest The authors confirm that they do not have any Conflicts of interest nor competing interests.

References

- Pohl, K.: Requirements Engineering: Fundamentals, Principles, and Techniques. Springer. (2010)
- Denger, C. and Olsson, T.: Quality assurance in requirements engineering. In: Engineering and Managing Software Requirements. p. 163–185, Springer, Berlin, Heidelberg. (2005)
- Knauss, E., El Boustani, C., and Flohr, T.: Investigating the impact of software requirements specification quality on project success. In: International Conference on Product-Focused Software Process Improvement. Springer. (2009)
- Boehm, B.W., Papaccio, P.N.: Understanding and controlling software costs. *IEEE Trans. Softw. Eng.* **14**(10), 1462–1477 (1988)
- Frederick P. Brooks, J.: No Silver Bullet - Essence and Accidents of Software Engineering. *IEEE Comput.* **20**(4). (1997)
- Bubenko, J.A.: Challenges in requirements engineering. In: Proceedings of 1995 IEEE International Symposium on Requirements Engineering (RE'95). IEEE. (1995)
- Femmer, H.: Requirements engineering artifact quality: definition and control. Technische Universität München. (2017)
- Yu, E.S., Mylopoulos, J.: From ER To “aR”—modelling strategic actor relationships for business process reengineering. *Int. J. Cooperative Inf. Syst.* **4**(02n03), 125–144 (1995)
- Jacobs, S., Holten, R.: Goal driven business modelling: supporting decision making within information systems development. In: Proceedings of conference on Organizational computing systems. (1995)
- Clements, P., Bass, L.: Relating business goals to architecturally significant requirements for software systems. Software Engineering Institute, Carnegie Mellon University. (2010)
- Ali, R., Dalpiaz, F., Giorgini, P.: A goal-based framework for contextual requirements modeling and analysis. *Requir. Eng.* **15**(4), 439–458 (2010)
- Dardenne, A., van Lamsweerde, A., Fickas, S.: Goal-directed requirements acquisition. *Sci. Comput. Program.* **20**(1–2), 3–50 (1993)
- Van Lamsweerde, A. and Letier, E.: From object orientation to goal orientation: a paradigm shift for requirements engineering. *Radical Innovations of Software and Systems Engineering in the Future*. Springer. p. 325–340. (2004)
- Mylopoulos, J., Chung, L., Nixon, B.: Representing and using non-functional requirements: a process-oriented approach. *IEEE Trans. Softw. Eng.* **18**(6), 483–497 (1992)
- Chung, L., et al.: Non-functional requirements in software engineering. Vol. 5. Springer Science & Business Media. (2012)
- Kaiya, H., Horai, H., Saeki, M.: AGORA: Attributed goal-oriented requirements analysis method. In: Proceedings IEEE joint international conference on requirements engineering. IEEE. (2002)
- Yu, E.S.: Towards modelling and reasoning support for early-phase requirements engineering. In: Proceedings of ISRE'97: 3rd IEEE International Symposium on Requirements Engineering. IEEE. (1997)
- Bresciani, P., et al.: Tropos: an agent-oriented software development methodology. *Auton. Agent. Multi Agent Syst.* **8**(3), 203–236 (2004)
- Pacheco, C., Garcia, I.: A systematic literature review of stakeholder identification methods in requirements elicitation. *J. Syst. Softw.* **85**(9), 2171–2181 (2012)
- Salger, F.: Requirements reviews revisited: residual challenges and open research questions. In: 21st IEEE International Requirements Engineering Conference (RE). IEEE. (2013)
- Zelkowitz, M.V., et al.: The Software Industry: A State of the Art Survey. (1983)
- Lima, P., et al.: An extended systematic mapping study about the scalability of i* models. *CLEI Electron. J.* **19**(3), 7–7 (2016)
- Yu, E., et al.: Strengths and Weaknesses of the i* Framework: An Empirical Evaluation. *Social Modeling for Requirements Engineering*, MIT Press. (2011)
- Amyot, D., et al.: GRL Modeling and Analysis with jUCMNav. *iStar* **766**, 160–162 (2011)
- ITU-T, Z.: 151 User Requirements Notation (URN)—Language Definition. ITU-T. (2018)
- Neace, K., Roncace, R., Fomin, P.: Goal model analysis of autonomy requirements for Unmanned aircraft systems. *Req. Eng.* **23**(4), 509–555 (2018)
- Fowler, M.: Refactoring: improving the design of existing code. Addison-Wesley Professional. (2018)
- Sharma, T., Spinellis, D.: A survey on software smells. *J. Syst. Softw.* **138**, 158–173 (2018)
- Yang, L., Liu, H., and Niu, Z.: Identifying fragments to be extracted from long methods. In: 16th Asia-Pacific Software Engineering Conference. IEEE. (2009)
- El-Attar, M., Miller, J.: Improving the quality of use case models using antipatterns. *Softw. Syst. Model.* **9**(2), 141–160 (2010)
- El-Attar, M., Miller, J.: Constructing high quality use case models: a systematic review of current practices. *Req. Eng.* **17**(3), 187–201 (2012)
- Misbhaudhin, M., Alshayeb, M.: UML model refactoring: a systematic literature review. *Empir. Softw. Eng.* **20**(1), 206–251 (2015)
- Alkharabsheh, K., et al.: Software design smell detection: a systematic mapping study. *Softw. Qual. J.* **27**(3), 1069–1148 (2019)
- Baqais, A., Alshayeb, M.: Automatic software refactoring: a systematic literature review. *Softw. Qual. J.* **2**(28), 459–502 (2020)
- AbuHassan, A., Alshayeb, M., Ghouti, L.: Software smell detection techniques: a systematic literature review. *J. Softw. Evol. Process.* **33**(3), e2320 (2021)
- Czibula, G., Marian, Z., Czibula, I.G.: Detecting software design defects using relational association rule mining. *Knowl. Inf. Syst.* **42**(3), 545–577 (2015)
- Lee, S.-J., et al.: Co-changing code volume prediction through association rule mining and linear regression model. *Expert Syst. Appl.* **45**, 185–194 (2016)
- Kessentini, M., et al.: Design defect detection rules generation: a music metaphor. In: 15th European Conference on Software Maintenance and Reengineering. IEEE. (2011)
- Maddeh, M. and Ayouni, S.: Extracting and modeling design defects using gradual rules and UML profile. In: IFIP International Conference on Computer Science and its Applications. Springer. (2015)
- Palomba, F., et al.: Lightweight detection of android-specific code smells: the aDoctor project. In: IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE. (2017)
- Kim, T.-W., Kim, T.-G., Seu, J.-H.: Specification and automated detection of code smells using ocl. *Int. J. Softw. Eng. Appl.* **7**(4), 35–44 (2013)
- Kim, T.-W., Kim, T.-G.: Automated code smell detection and refactoring using OCL. *KIPS Trans. PartD.* **15**(6), 825–840 (2008)
- Boussaa, M., et al.: Competitive coevolutionary code-smells detection. In: International Symposium on Search Based Software Engineering. Springer. (2013)
- Ghannem, A., Kessentini, M., El Boussaidi, G.: Detecting model refactoring opportunities using heuristic search. In: Proceedings of the 2011 Conference of the Center for Advanced Studies on Collaborative Research. IBM Corp. (2011)
- Kessentini, W., et al.: A cooperative parallel search-based software engineering approach for code-smells detection. *IEEE Trans. Softw. Eng.* **40**(9), 841–861 (2014)
- Ouni, A., et al.: Multi-criteria code refactoring using search-based software engineering: an industrial case study. *ACM Trans. Softw. Eng. Methodol. (TOSEM)* **25**(3), 23 (2016)

47. Ouni, A., et al.: Search-based web service antipatterns detection. *IEEE Trans. Serv. Comput.* **10**(4), 603–617 (2017)
48. Serikawa, M.A., et al.: Towards the characterization of monitor smells in adaptive systems. In: X Brazilian Symposium on Software Components, Architectures and Reuse (SBCARS). IEEE. (2016)
49. Macia, I., Garcia, A., and von Staa, A.: Defining and applying detection strategies for aspect-oriented code smells. In: Brazilian Symposium on Software Engineering. IEEE. (2010)
50. Fontana, F.A., et al.: Comparing and experimenting machine learning techniques for code smell detection. *Empir. Softw. Eng.* **21**(3), 1143–1191 (2016)
51. Hassaine, S., et al.: IDS: An immune-inspired approach for the detection of software design smells. In: Seventh International Conference on the Quality of Information and Communications Technology. IEEE. (2010)
52. Khomh, F., et al.: A bayesian approach for the detection of code and design smells. In: Ninth International Conference on Quality Software. IEEE. (2009)
53. Maiga, A., et al.: Support vector machines for anti-pattern detection. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. ACM. (2012)
54. Oliveto, R., et al.: Identifying method friendships to remove the feature envy bad smell: NIER track. In: 33rd International Conference on Software Engineering (ICSE). IEEE. (2011)
55. Hozano, M., et al.: Evaluating the accuracy of machine learning algorithms on detecting code smells for different developers. In: ICEIS (2). (2017)
56. Jiang, Y., Li, M., Zhou, Z.-H.: Software defect detection with R ocus. *J. Comput. Sci. Technol.* **26**(2), 328–342 (2011)
57. Maneerat, N. and Muenchaisri, P.: Bad-smell prediction from software design model using machine learning techniques. In: 2011 Eighth International Joint Conference on Computer Science and Software Engineering (JCSSE). IEEE. (2011)
58. Alkharabsheh, K., et al.: Influence of nominal project knowledge in the detection of design smells: an exploratory study with god class. *Int. J. Adv. Stud. Comput. Sci. Eng.* **5**(11), 120 (2016)
59. Oliveto, R., et al.: Numerical signatures of antipatterns: An approach based on b-splines. In: 14th European Conference on Software Maintenance and Reengineering. IEEE. (2010)
60. Kaur, K., Jain, S.: Evaluation of machine learning approaches for change-proneness prediction using code smells. In: Proceedings of the 5th International Conference on Frontiers in Intelligent Computing: Theory and Applications. Springer. (2017)
61. Vaucher, S., et al.: Tracking design smells: lessons from a study of god classes. In: 16th Working Conference on Reverse Engineering. IEEE. (2009)
62. Bertran, I.M.: Detecting architecturally-relevant code smells in evolving software systems. In: 33rd International Conference on Software Engineering (ICSE). IEEE. (2011)
63. Dextrun, J., et al.: Detecting bad smells with weight based distance metrics theory. In: Second International Conference on Instrumentation, Measurement, Computer, Communication and Control. IEEE. (2012)
64. Fourati, R., Bouassida, N., Abdallah, H.B.: A metric-based approach for anti-pattern detection in uml designs. *Computer and Information Science* 2011. Springer. p. 17–33. (2011)
65. Nongpong, K.: Feature envy factor: A metric for automatic feature envy detection. In: 7th International Conference on Knowledge and Smart Technology (KST). IEEE. (2015)
66. Singh, S., Kahlon, K.: Effectiveness of encapsulation and object-oriented metrics to refactor code and identify error prone classes using bad smells. *ACM SIGSOFT Softw. Eng. Notes* **36**(5), 1–10 (2011)
67. Tahvildari, L. and Kontogiannis, K.: A metric-based approach to enhance design quality through meta-pattern transformations. In: Seventh European Conference on Software Maintenance and Reengineering, 2003. Proceedings.: IEEE. (2003)
68. Tahvildar, L., Kontogiannis, K.: Improving design quality using meta-pattern transformations: a metric-based approach. *J. Softw. Maint. Evol. Res. Pract.* **16**(4–5), 331–361 (2004)
69. Fontana, F.A., Maggioni, S.: Metrics and antipatterns for software quality evaluation. In: IEEE 34th Software Engineering Workshop. IEEE. (2011)
70. Marinescu, R.: Detecting design flaws via metrics in object-oriented systems. In: Proceedings 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems. TOOLS 39. IEEE. (2001)
71. Olbrich, S., et al.: The evolution and impact of code smells: a case study of two open source systems. In: 3rd international symposium on empirical software engineering and measurement. IEEE. (2009)
72. Salehie, M., Li, S., Tahvildari, L.: A metric-based heuristic framework to detect object-oriented design flaws. In: 14th IEEE International Conference on Program Comprehension (ICPC'06). IEEE. (2006)
73. Srivisut, K., Muenchaisri, P.: Bad-smell metrics for aspect-oriented software. In: 6th IEEE/ACIS International Conference on Computer and Information Science (ICIS 2007). IEEE. (2007)
74. Padilha, J., et al.: Detecting god methods with concern metrics: an exploratory study. In: Latin-American Workshop on Aspect-Oriented Software Development. (2013)
75. Dextrun, J., et al.: Detection and refactoring of bad smell caused by large scale. *Int. J. Softw. Eng. Appl.* **4**(5), 1 (2013)
76. Fernandes, E., et al.: A review-based comparative study of bad smell detection tools. In: Proceedings of the 20th International Conference on Evaluation and Assessment in Software Engineering. ACM. (2016)
77. Fokaefs, M., Tsantalos, N., Chatzigeorgiou, A.: Jdeodorant: Identification and removal of feature envy bad smells. In: 2007 IEEE International Conference on Software Maintenance. IEEE. (2007)
78. Moha, N., et al.: Decor: A method for the specification and detection of code and design smells. *IEEE Trans. Softw. Eng.* **36**(1), 20–36 (2009)
79. Marinescu, R.: Assessing technical debt by identifying design flaws in software systems. *IBM Journal of Research and Development.* **56**(5): p. 9: 1–9: 13. (2012)
80. Ruhroth, T., Voigt, H., Wehrheim, H.: Measure, diagnose, refactor: a formal quality cycle for software models. In: 2009 35th Euromicro Conference on Software Engineering and Advanced Applications. IEEE. (2009)
81. Voigt, H., Ruhroth, T.: A quality circle tool for software models. In: International Conference on Conceptual Modeling. Springer. (2008)
82. Cabot, J., Gogolla, M.: Object constraint language (OCL): a definitive guide. In: International school on formal methods for the design of computer, communication and software systems. Springer. (2012)
83. Kim, D.-K.: Software quality improvement via pattern-based model refactoring. In: 2008 11th IEEE High Assurance Systems Engineering Symposium. IEEE. (2008)
84. Arendt, T., et al.: Towards syntactical model quality assurance in industrial software development: process definition and tool support. *Software Engineering – Fachtagung des GI-Fachbereichs Softwaretechnik.* (2011)
85. Mohamed, M., Romdhani, M., Ghedira, K.: M-REFACTOR: A new approach and tool for model refactoring. *ARPN J. Syst. Softw. Syst. Model.* **1**(4), 117–122 (2011)
86. Enkevort, T.V.: Refactoring UML models: using openarchitectureware to measure uml model quality and perform pattern matching on UML models with OCL queries. In: Proceedings of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications. ACM. (2009)

87. Xu, J.: Rule-based automatic software performance diagnosis and improvement. *Perform. Eval.* **69**(11), 525–550 (2012)
88. Arcelli, D., Cortellessa, V., Di Pompeo, D.: Automating performance antipattern detection and software refactoring in UML models. In: 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE. (2019)
89. Kolovos, D.S., Paige, R.F., Polack, F.A.: Eclipse development tools for epsilon. In: Eclipse Summit Europe, Eclipse Modeling Symposium. Citeseer. (2006)
90. Štolc, M., Poláček, I.: A visual based framework for the model refactoring techniques. In: 2010 IEEE 8th International Symposium on Applied Machine Intelligence and Informatics (SAMI). IEEE. (2010)
91. Asano, K., Hayashi, S., Saeki, M.: Detecting bad smells of refinement in goal-oriented requirements analysis. In: International Conference on Conceptual Modeling. Springer. (2017)
92. Yan, J.B.: Static Semantics Checking Tool for jUCMNav. Master's project, SITE, University of Ottawa. (2008)
93. Saeki, M., Hayashi, S., Kaiya, H.: A tool for attributed goal-oriented requirements analysis. In: 2009 IEEE/ACM International Conference on Automated Software Engineering. IEEE. (2009)
94. Mohammed, M.A., Alshayeb, M., Hassine, J.: A search-based approach for detecting circular dependency bad smell in goal-oriented models. *Software and Systems Modeling*. (2022)
95. Alqmase, M., et al.: Threshold Extraction Framework for Software Metrics. *34*(5): p. 1063–1078. (2019)
96. Yu, E.: Modelling Strategic Relationships for Process Reengineering. *Social Modeling for Requirements Engineering*. p. 2011. (2011)
97. Wohlin, C., et al.: Experimentation in Software Engineering. Springer Science & Business Media. (2012)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.